Giulio Peretti

CoAP over DTLS TinyOS Implementation and Performance Analysis

Implementazione di CoAP e DTLS in TinyOS ed Analisi delle Prestazioni

Tesi di laurea magistrale

Advisor: Prof. Michele Zorzi
Co-Advisor: Vishwas Lakkundi, Ph.D.

University of Padova
School of Engineering
Department of Information Engineering
December 10, 2013

Abstract

The IP-based Internet of Things (IoT) and the availability of inexpensive sensing devices capable of wireless communications enable a wide range of applications such as intelligent building automation and control, mobile health care, smart logistics and distributed monitoring. IoT devices are expected to employ Constrained Application Protocol (CoAP) for the integration of such applications with the Internet, which suggests the use of Datagram Transport Layer Security (DTLS) protocol in order to provide authentication functionalities as well as essential end-to-end security for the transmission of sensitive information.
This thesis presents firstly an application called BlinkToSCoAP, obtained through the integration of three libraries implementing lightweight versions of DTLS and CoAP protocols as well as the IPv6/6LoWPAN stack. Secondly, an experimental campaign is presented that evaluates the performance of the DTLS security operations. The experiments analyze the BlinkToSCoAP's communications exchanged between two Zolertia Z1 devices, allowing evaluations in terms of memory footprint, energy consumption, latency and packet overhead. Based on performance analysis results and the experience gained during the implementation phase, this thesis finally presents an outlook on future works that can be developed in order to enhance the application performance.

Sommario

La tecnologia Internet-of-Things (IoT), basata sul protocollo IP, e la disponibilità di dispositivi economici dotati di sensori e funzionalità wireless sono alla base di un'ampia gamma di applicazioni come il controllo intelligente ed automatico di edifici, la supervisione delle funzioni vitali in ambito medico ed il monitoraggio distribuito. I dispositivi IoT implementano il Constrained Application Protocol (CoAP) per integrare tali applicazioni con Internet. Lo standard di tale protocollo consiglia l'utilizzo del protocollo Datagram Transport Layer Security (DTLS) per garantire le essenziali funzionalità di sicurezza end-to-end per trasmissioni di dati sensibili e per autenticare i dispositivi coinvolti nella comunicazione.
La tesi presenta un'applicazione che integra i protocolli DTLS, CoAP e lo stack IPv6/6LoWPAN, basata su implementazioni sviluppate dal SIGNET Group del Dipartimento di Ingegneria dell'Informazione di Padova. In secondo luogo viene presentata una serie di esperimenti con lo scopo di valutare la variazione di performance dovuta alle operazioni di sicurezza del protocollo DTLS. Tale variazione sarà valutata in termini di memoria ed energia richiesta, ritardi e overhead nei pacchetti. Basandosi sui risultati degli esperimenti effettuati e sull'esperienza guadagnata durante la fase di sviluppo dell'applicazione, la tesi presenta infine una serie di suggerimenti per eventuali lavori futuri che estendono il lavoro presentato.

Acknowledgements

Desidero ringraziare tutti coloro che mi hanno aiutato nella realizzazione di questa mia Tesi. Ringrazio la mia famiglia per il quotidiano sostegno morale, per la fiducia riposta nelle mie capacità. Ringrazio Michele Zorzi e Vishwas Lakkundi per il supporto fornito, ringrazio Giulio Marin per avermi aiutato nella revisione della lingua inglese e per i suoi preziosi consigli. Ringrazio inoltre Moreno Dissegna e Matteo Fiorindo per il supporto fornito nella progettazione e valutazione degli esperimenti effettuati, Mario Emilio Cecconato per il suo sostegno prima e durante la stesura del codice dell'applicazione sviluppata. Ringrazio infine tutte le persone che mi sono state vicino in questi mesi impegnativi, le persone che mi hanno fatto sorridere, le persone che non hanno mai smesso di spronarmi ed avere fiducia in me.

Padova, December 10, 2013 Giulio

Contents

List of Figures

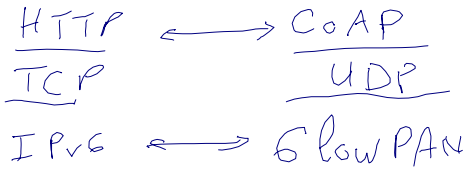
Chapter 1

Introduction

1.1 Motivation

The paradigm Internet of Things (IoT) denotes the Internet-like interconnection of highly heterogeneous and wireless-capable entities such as sensors, actuators and mobile devices, in a Low power and Lossy Network (LLN). These small devices usually have 8-bit microcontrollers with small amounts of ROM and RAM, while constrained networks usually have high packet error rates and very low throughputs. Proposed for the first time by Kevin Ashton in 2009 [9], the IoT allows a wide range of application scenarios with potentially critical actuating and sensing tasks. The integration of such applications with the Internet will contribute to shape a vision of a future Web that is nowadays denoted as the Web of Things (WoT). In this scenario, IoT entities interact with each other, with Internet remote services and with humans carrying an Internet-capable device, like a smartphone.

In order to simplify this integration, the fundamental building blocks used for IoT applications are the web services, the IPv6 protocol and its LLN compressed version obtained through the IPv6 Low power Wireless Personal Area Network (6LoWPAN) protocol. More specifically, the 6LoWPAN protocol defines encapsulation and header compression mechanisms that allow IPv6 packets to be efficiently sent over constrained networks. The Internet Engineering Task Force Constrained Restful Environments (IETF CoRE) working group provides a framework for IoT resource-oriented applications with the standardization of the Constrained Application Protocol (CoAP), that is a LLN optimized version of the HTTP protocol designed to run over the UDP protocol, in order to guarantee efficient communications at the application level.



The standardization of the CoAP protocol suggests the introduction of transmission security. The most common approach to provide security functionalities to the Internet communications is given by the Transport Layer Security (TLS) protocol, together with a public-key infrastructure. Unfortunately, TLS is not suited to constrained networks as it needs reliable channels and due to high power consumption. For this reason, multiple alternatives have been proposed to solve the security requirements in constrained environments. Therefore, in order to protect the transmission of information, the CoAP standard [16] allows either the usage of Datagram Transport Layer Security (DTLS) or Internet Protocol Security (IPsec), which are end-to-end security approaches that achieve replay protection, data integrity and authentication.

IPsec is a network layer protocol and is implemented in the kernel of operating systems. For this reason, it may not be the best choice for all kinds of environments [2]. IPsec is not supported by all the embedded IP stacks and neither by all PC operating systems or back-end web servers. In addition, application developers may not have privileges to add a security gateway to the network or to enable and configure IPsec. Firewalls and NATs might thus compromise the usage of this approach.

A more suitable solution is currently represented by a datagram capable version of TLS, the DTLS protocol. A great advantage of DTLS over IPsec is that it is an application layer protocol, thus implemented in the application space instead of in the kernel, therefore avoiding the aforementioned problems for the IPsec protocol. In addiction, thanks to their similarity, DTLS allows the reusage of existing TLS protocol infrastructure at the cost of a minimal application overhead [15].

1.2 Contribution

This thesis presents an IoT application that includes the CoAP protocol, protected by the DTLS protocol, running over the IPv6/6LowPAN stack. It is realized by merging, adapting and optimizing previously existing implementations of these protocols, all written by different authors within the Department of Information Engineering (DEI) at the University of Padova. Furthermore, it gives an in-depth analysis of the performance variation due to the presence of the security protocol including:

- additional RAM and ROM memory usage of the application, since data

storage and especially RAM are very critical resources on actual sensor node platforms

- computational time and energy overhead required, as they represent two important evaluation criteria of the feasibility of the security implementation
- packet length overhead introduced by the DTLS header.

1.3 Related Work

Recently, a lot of research into end-to-end security protocols for the IoT and WSNs running CoAP has been conducted.

Authors of [10] introduce a DTLS security architecture that performs two-way authentication, which includes both client and server authentication, based on RSA, the most widespread key exchange algorithm. This particular asymmetric encryption algorithm requires too much resources in constrained devices, furthermore the devices considered in that architecture have to guarantee hardware support for secure application and RSA key storage, such as Trusted Platform Modules (TPMs). This architecture, running over the 6LowPAN protocol, achieves proper authentication through the access control server, a trusted non-constrained entity in which the access rights of the sensor device are stored, and through X.509 certificates signed by a trusted third party, called Certificate Authority. The architecture has been then tested in terms of latency and energy consumption in order to evaluate its feasibility.

In [6] the performance impact of several DTLS security modes, proposed by the CoAP standard, is analyzed in order to identify the limitations of node platforms and the requirements of IoT applications. The authors have evaluated the energy, the packet and the computational overhead as well as the memory footprint of the various security modes, showing that the small memory space and the absence of Elliptic Curve Cryptography (ECC) hardware support are a critical aspect for the compatibility of the IoT networks with existing public-key certification infrastructures. However, some DTLS security suites were identified as viable if security and resources usage compromises are allowed by the network application.

In [7] an extensive experimental evaluation is presented to identify the most appropriate secure communications mechanisms between end-to-end network-layer and application-layer security, compared in terms of energy, computational overheads and memory footprints. The authors have described the impact of end-to-end

security on communications rate of sensing devices as well as on the lifetime of the constrained network. The end-to-end approach provides the benefit of enabling secure communications regardless of the application, while the network-layer security may facilitate the integration with certification infrastructures through the usage of ECC, at the cost of more resources.

In [13], it has been shown that DTLS headers can be compressed using 6LoWPAN mechanisms, significantly reducing the number of additional security bits. This result leads to an increment of both the network lifetime and the achievable throughput. The same authors, in a more recent work [14], have presented Lithe, a DTLS secured CoAP implementation that exploits the data compression methods mentioned earlier.

1.4 Outline

After introducing the *Constrained Application Protocol* (CoAP) and the *Datagram Transport Layer Security* (DTLS), the thesis presents the environment set-up used to develop and test the BlinkToSCoAP application, described in Chapter 5. The following Chapter 6 presents the experiment campaign together with the results achieved, while the last Chapter 7 draws the conclusions and reports some suggestions for future work.

Chapter 2

Constrained Application Protocol

2.1 CoAP Overview

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol intended to be used by constrained devices in Machine-to-Machine (M2M) applications. M2M can be seen as a subset of IoT, as it refers to all the technologies that allow machines to communicate with each other, especially over Internet protocols in wireless channels. The CoAP protocol provides a client/server interaction model between application endpoints and includes the same key functionalities of the HTTP protocol. For this reason CoAP is easily interfaced with HTTP, resulting in simplified web integration while also ensuring M2M critical requirements such as low overhead, multicast support, built-in discovery and simplicity.

The Representational State Transfer (ReST), named for the first time by Roy Thomas Fielding in his Ph.D. dissertation [5], is a network architectural style that abstracts the implementation of the network elements within a distributed hypermedia system[1]. Moreover, ReST focuses on the architectural elements role, the constraints in their interactions and interpretation as significant data elements.

The Constrained REST Environments (CoRE) working group aims at the realization of the ReST architecture, exemplified in Figure 2.1, suitable for constrained devices and networks. Their work comprises the specification of the CoAP protocol [16] which has been proposed with the following features:

- constrained web protocol fulfilling M2M requirements.
- UDP binding with optional reliability, supporting unicast and multicast

[1] A multimedia system in which information items are connected and can be presented together

requests.

- asynchronous message exchanges.
- small header overhead and parsing complexity.
- URI and Content-type support.
- 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.
- security binding to Datagram Transport Layer Security (DTLS).

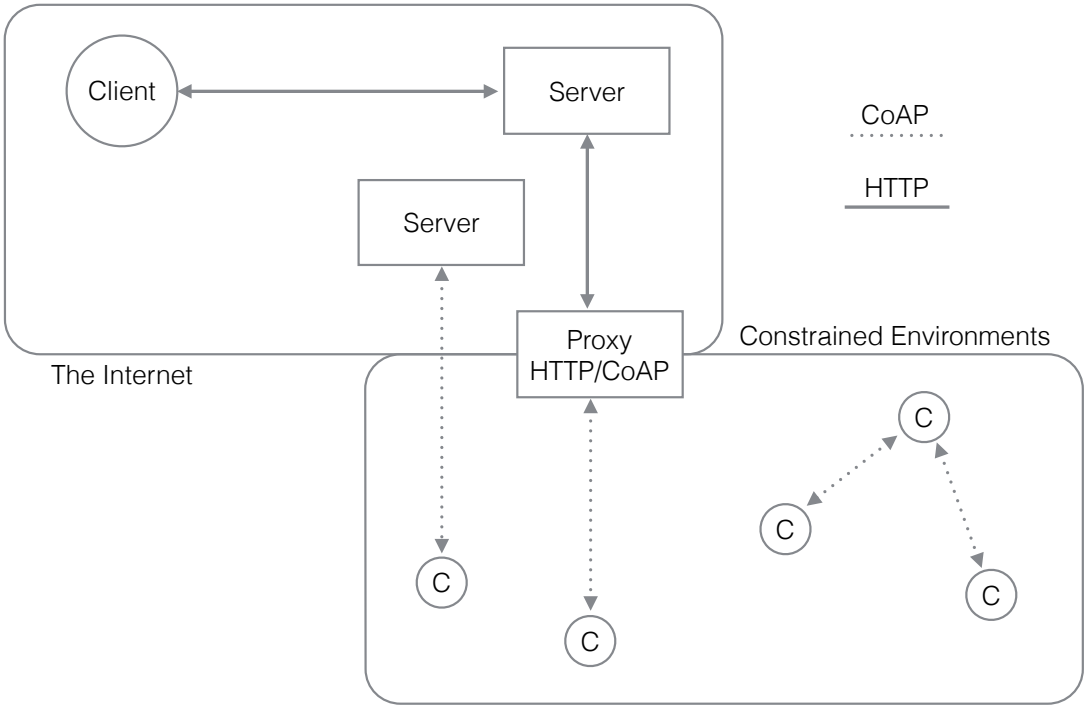


Figure 2.1: The CoRE ReSTful architecture.

The CoAP protocol is divided into two layers, depicted in Figure 2.2, that provide different functionalities. The higher one handles all the mechanisms needed by the protocol to provide web services, while the lower layer implements techniques to handle the unreliability of the channel. The next two sections will present these two layers.

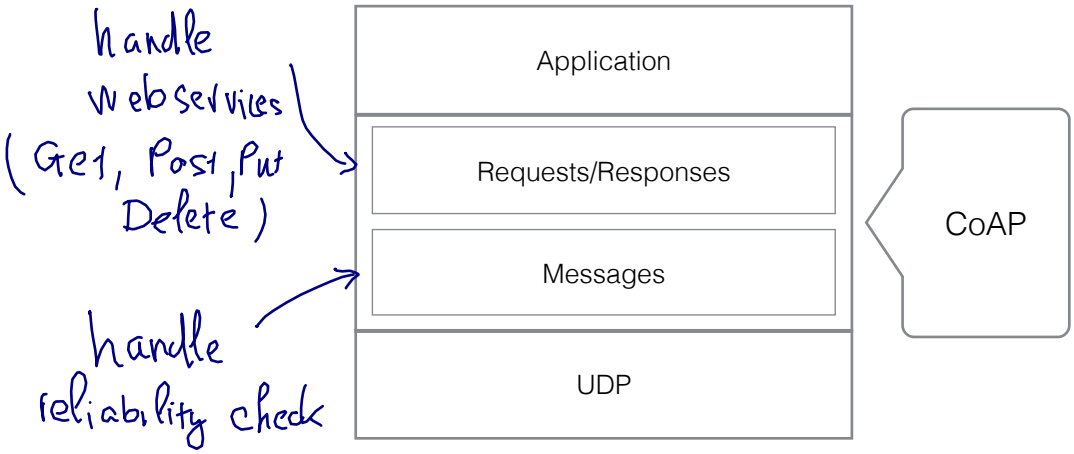


Figure 2.2: CoAP protocol layers.

2.1.1 CoAP Requests and Responses

The CoAP client/server interaction model, depicted in Figure 2.3, imposes that *CoAP requests* are sent by clients in order to request an action on a *resource* of the server. After the request elaboration, the server sends back a *CoAP response* containing an appropriate *response code* and optionally a resource representation.

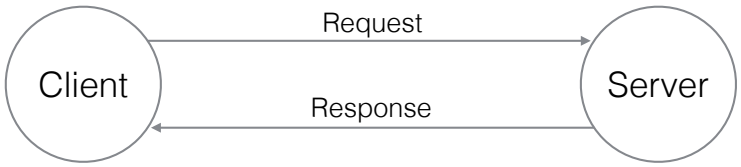


Figure 2.3: Client-server model.

The client request contains a *method* that specifies the action requested, an unique identifier of the server resource called *Uniform Resource Identifier* (URI) and optionally a payload containing meta-data about the request. The CoAP standard defines four different methods:

- GET: retrieves an information representation of the resource.
- POST: carries an information representation and asks the receiver to process it. The output depends on the target resource, usually involving resource creation or update.

- PUT: requests an update operation of the resource identified by the request URI with the carried information representation.
- DELETE: causes the deletion of the resource identified by the request URI.

Upon reception of the request, the server elaborates it and, if no errors occur, sends back to the client its response containing a response code that indicates the result of the request process. Response codes are divided into three classes:

- 2.xx (Success): the request has successfully been received and processed.
- 4.xx (Client Error): the request was not valid or correctly understood by the server.
- 5.xx (Server Error): the server accepted the request but failed to process it.

The fraction of the response code just denoted with xx does not have any categorization role: it gives instead additional details of the output of the request process. For example, the most common HTTP response code is the 404 or not found error, which indicates that the client request was correct but the server was not able to find the resource pointed by the URI field.

The matching between requests and responses is achieved by means of a *token*, that is an unique identifier of any request/response couple between two specific endpoints. This field is included on every CoAP request as well as in every CoAP response.

2.1.2 Messages

As CoAP is by default bound to UDP, requests and responses can appear duplicated, arrive out of order or go missing. To deal with this issue, the protocol is theoretically divided into two logical layers, where the upper one comprises the request/response mechanisms previously introduced and the lower one handles a lightweight reliability mechanism.

The message layer is totally independent of the request/response layer. It defines four message types:

- *Confirmable* (CON): indicates that the carried data have to be acknowledged from the receiver, providing reliability functionality.
- *Non Confirmable* (NON): carries data that do not require acknowledgments but still has to be protected from message duplication.

- *Acknowledgement* (ACK): acknowledges CON messages.
- *Reset* (RST): signals errors occurred in the reception of a CON or NON message.

The matching of CON/ACK messages and the message duplicate detection is done by means of a Message ID, generated and enclosed in every CON and NON message. As for the token introduced in the previous section, the Message ID has to be unique for every NON or CON/ACK message between two specific endpoints. Until the reception of the matched ACK message from the same destination, the CON message is retransmitted through the channel using a default timeout with an exponential back-off time.

These message types can embody client requests or server responses as shown in Table 2.1. The special combination of a Confirmable message without any request or response included is used only to trigger a Reset message, realizing the CoAP ping application.

	CON	NON	ACK	RST
Request	X	X	-	-
Response	X	X	X	-
Empty	*	-	X	X

Table 2.1: Usage of message types.

When a request arrives in a CON message, and the server has the response immediately available, it can be carried directly in the resulting ACK message saving network resources. These kind of responses are called piggy-backed responses. On the other hand, if the server needs a longer time to process the request, it should prevent client retransmissions by immediately replying with an ACK message. When available, the response will be sent in a CON or NON message. An example of piggy-backed and separated responses is depicted in Figure 2.4.

2.2 Message Format

CoAP message format is illustrated in Figure 2.5; it starts with a 4 bytes header that contains the following fields:

- Version (Ver): indicates the protocol version number.

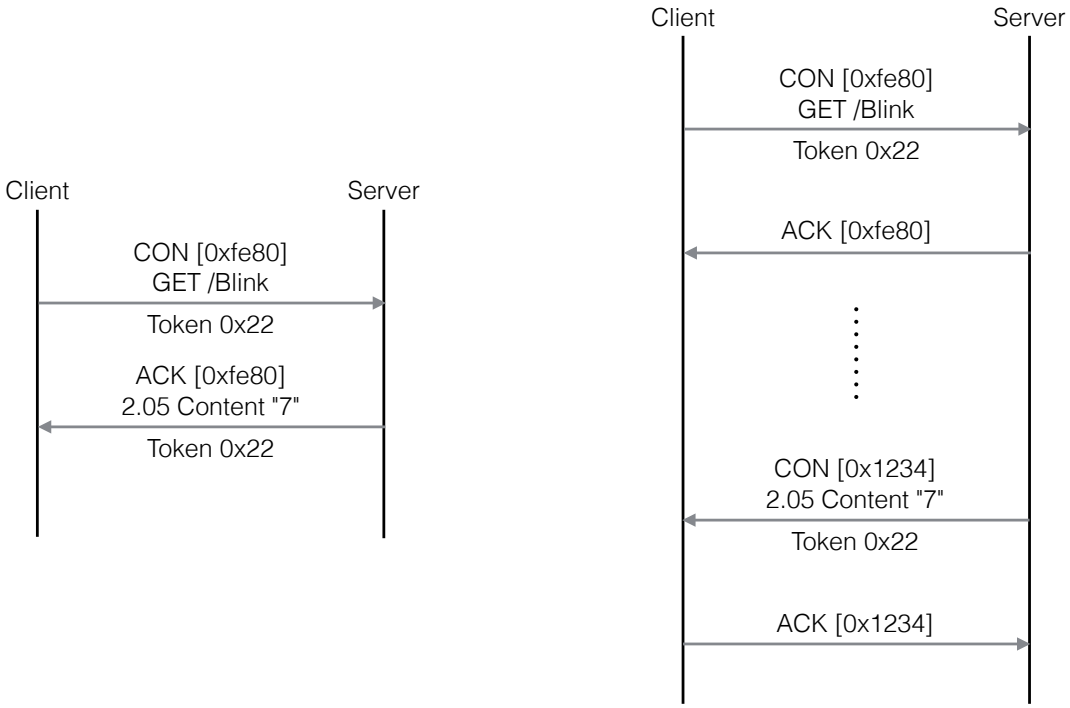


Figure 2.4: Piggy-backed and separated responses in CoAP.

- Type (T): indicates the message type (CON, NON, ACK, RST).
- Token Length (TKL): represents the number of bytes of the Token field.
- Code: signals a client request or a server response.
- Message ID: contains the Message ID used to match CON and ACK messages as well as to detect message duplicates.

The header is followed by the token field, used to correlate requests and responses, which can be from 0 to 8 bytes long. The length can be zero if no other token is currently used for the same destination or when the client sends requests serially and receives only piggy-backed responses. Besides, a long randomized token is generated when the CoAP protocol is not secured by a transport layer security protocol, acting as a protection from response spoofing.
After those fields, a sequence of zero or more CoAP options and the optional payload take place, separated by a one-byte Payload Marker (0xFF). The absence of this marker indicates that no payload is present.

The total message size is upper bounded by the CoAP specification but, to avoid undesirable packet fragmentation at lower layers, it should fit in an IP MTU.

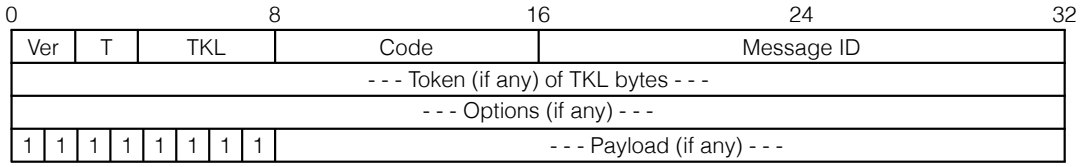


Figure 2.5: CoAP message format.

Chapter 3

Datagram Transport Layer Security

Datagram Transport Layer Security (DTLS), defined in [15], is an extension of the Transport Layer Security (TLS) protocol. This chapter introduces TLS and then describes the main modifications accomplished in DTLS in order to adapt it to unreliable transport protocols.

3.1 TLS Overview

Transport layer security, presented in [4], is nowadays the most widespread web security protocol. It provides messages authentication, integrity and confidentiality. The structure of the protocol, illustrated in Figure 3.1, is layered into two levels. In the higher one four subprotocols interact with the lower layer *Record Protocol* in order to provide different functionalities: *Handshake Protocol*, *Change Cipher Spec* (CCS) Protocol, *Alert Protocol* and *Application Protocol*. Due to this layered structure, at each layer, messages include fields for length, description, and content.

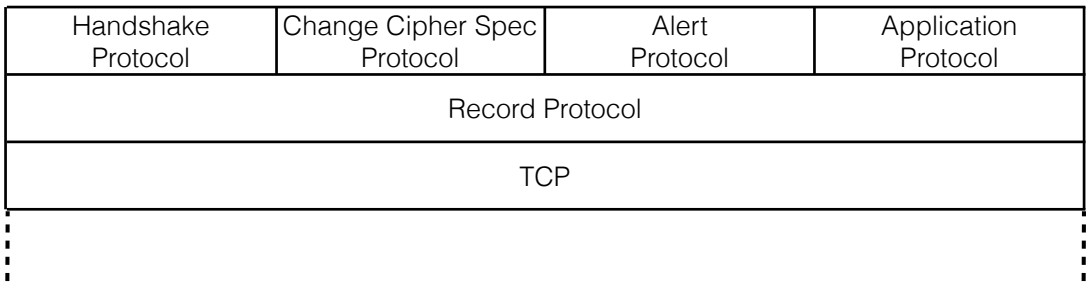


Figure 3.1: TLS protocol stack.

3.1.1 TLS Handshake Protocol

The Handshake Protocol consists of a series of messages exchanged by client and server before any application data transmission takes place. During this initial phase, endpoints reciprocally authenticate each other and negotiate security parameters. In Figure 3.2 a basic TLS handshake, consisting of only the essential messages, is illustrated. There are multiple variants of the TLS Handshake, depending on the specific application, that implement one or two way authentication[1] or include certificate verification messages. The functionality of these optional handshake messages are beyond the scope of this chapter and for this reason only the basic handshake messages are described below. From Figure 3.2 it can also be noticed that handshake messages are grouped in *flights*, defined in the TLS specification as groups of contiguously sent handshake messages.

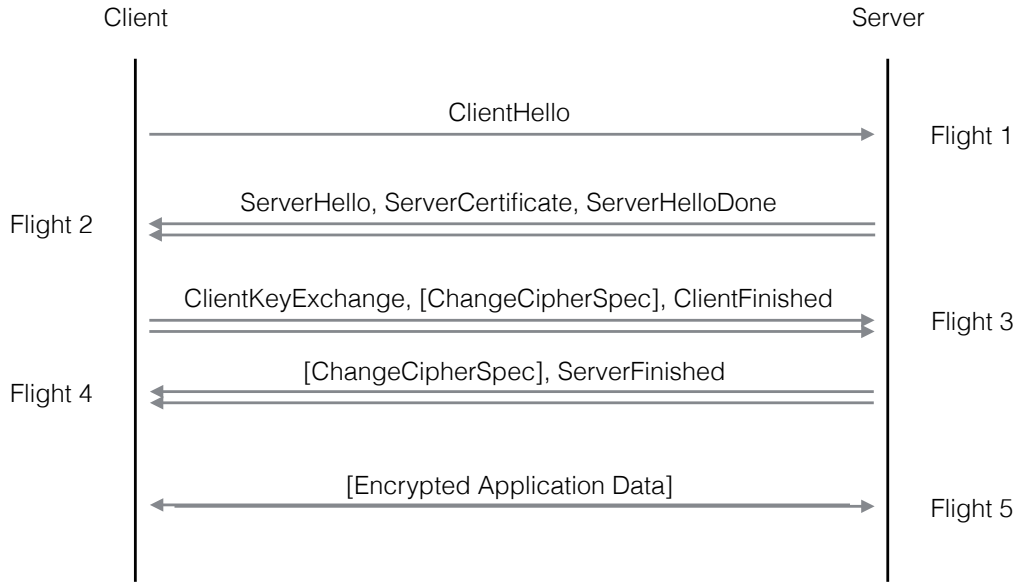


Figure 3.2: TLS handshake.

The essential messages of a TLS handshake are described below.

ClientHello (CH) This is typically the first message of the handshake phase (in some applications the server is allowed to request a ClientHello message). Its structure contains the following fields:

[1]One way authentication includes only server certificate while the two way authentication involves the client certificate as well

- the version of the protocol.
- a *client random token* used later to generate the *premaster secret*.
- a session ID that, if not empty, allows reuse of the security parameters already established in a previous session.
- an optional compression algorithm.
- a list of *cipher suites* available in the client.

In particular, a cipher suite is composed of:

- a symmetric encryption algorithm, used to encrypt and decrypt application data flowing between endpoints after the conclusion of the handshake phase.
- a MAC algorithm that provides message integrity.
- an asymmetric encryption algorithm used to securely exchange the premaster secret.

A typical example can be TLS_RSA_WITH_AES_CBC_SHA, where RSA is the key exchange algorithm, AES_CBC the Advanced Encryption Standard in Cipher-Block Chain mode symmetric encryption algorithm and SHA is the Secure Hash Algorithm used to provide message integrity.

ServerHello (SH) This message is the server response to the ClientHello. It contains:

- the protocol version supported by both endpoints that will be used for the connection.
- a *server random token* that, together with the client random token, will contribute to the generation of the premaster secret from which the *master secret* will be derived.
- a session ID for future session resumptions.
- the strongest client cipher suite (and compression algorithm) also supported by the server that will be used for the transmission.

ServerCertificate (SC) The server, after the ServerHello message, sends to the client its certificate containing the server's *public key*. The server certificate is also used by the client to verify the server identity.

ServerHelloDone (SHD) This is a message with no content that indicates the end of the ServerHello flight. After this message the server awaits the response from the client.

ClientKeyExchange (CKE) This message is sent after computing the premaster secret using both the client and server random tokens. The premaster secret is then enclosed in the ClientKeyExchange message after being encrypted through the chosen key exchange algorithm using the server public key. Both endpoints will use this parameter in order to locally compute the master secret, that will be extended through the PRF function[2] into several keys intended to be used for the encryption and HMAC algorithms. ClientKeyExchange message also includes the client protocol version in order to guard against rollback attacks, which cause the server and the client to use an earlier and thus less secure version of the protocol.

ChangeCipherSpec (CCS) notifies to the other endpoint that all future messages will be encrypted using the keys and algorithms just negotiated. The ChangeCipherSpec message is not really part of the Handshake Protocol as it belongs to the ChangeCipherSpec Protocol.

ClientFinished (CF) This message is the first message being encrypted and hashed by the record layer, and signals that the client has no other handshake messages to send. It contains also a hash of the entire conversation in order to provide further authentication of the client.

ServerFinished (SF) Like the CF message just described, this message is a hash of the entire handshake exchange until this point. If the client is able to successfully decrypt this message and the contained data, the TLS handshake is successful and the two endpoints are ready to exchange application data in a secure manner.

3.1.2 TLS ChangeCipherSpec, Alert and Application Protocol

The CCS Protocol is composed only by the ChangeCipherSpec message described above. This protocol has the purpose of allowing developers to define their own CCS mechanisms.
Messages belonging to the Alert Protocol convey TLS alerts carrying information about errors occurred. Alert messages are divided into two levels, *warning* and *fatal*. Fatal alerts result in the immediate termination of the session, forcing the establishment of a new connection in order to keep the transmissions safe. If otherwise a waring alert is sent or received, the connection can continue normally

[2] A mechanism used to produce a securely generated pseudo-random output of arbitrary length.

or be terminated depending on the specific situation. The Application Protocol refers to the higher level protocol that utilizes the DTLS protocol services.

3.1.3 TLS Record Protocol

The last portion of the TLS protocol, at the bottom of its structure, is the TLS Record Layer which applies security mechanisms and handles data transport. As illustrated in Figure 3.3, this protocol merges and fragments messages coming from the upper protocols into more manageable blocks. If possible, multiple messages of the same type can be enclosed together into a single record, or a single long one can be fragmented across several records. Moreover the TLS Record Protocol optionally compresses the data, adds a MAC field, encrypts and finally transmits the resulting extended fragment. When a new packet is delivered to Record Protocol from lower layers, the data is decrypted, verified, decompressed, reassembled and then passed on to higher layer protocols.

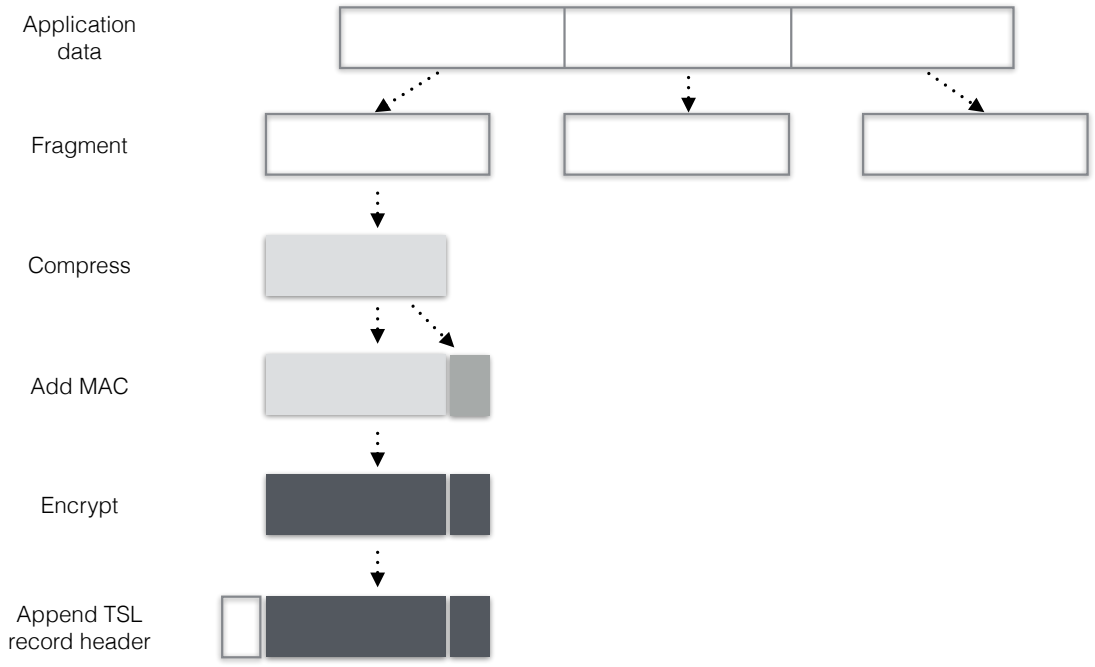


Figure 3.3: TLS record protocol operations.

The TLS protocol is assumed to be interfaced with a reliable transport protocol such as TCP. This requirement prohibits its utilization in LLN networks, where unreliable transport protocols are a more efficient choice. For this reason, as

mentioned in Chapter 2, the CoAP specification suggests to secure communications by means of the DTLS protocol.

3.2 DTLS Overview

Datagram Transport Layer Security is a modified version of TLS that resolves the original protocol issues when running over unreliable transport protocols. DTLS is designed to be as similar to TLS as possible in order to take advantage of pre-existing protocol infrastructures and implementations.
DTLS resolves several incompatibilities of the the TLS protocol running over unreliable protocols [15], and the main changes affect the Handshake Protocol and Record Protocol.

3.2.1 DTLS Handshake Protocol

The TLS handshake is not compatible with unreliable transport layer protocols because it would break the handshake process due to message loss or reordering, resulting in the failure of any successive communication. DTLS must then provide reliability to handshake messages, and this is done by means of two mechanisms: a simple *retransmission timer*, represented by the state machine shown in Figure 3.4, and a *handshake message number* field. Since the handshake messages are grouped in flights, the retransmission mechanism refers to these message groups as a whole instead of keeping track of each single message.
The handshake message number introduced by DTLS allows the receiver to reconstruct the correct order of handshake messages: when a peer receives one of them, it can quickly determine whether that message is the expected one or not. If so, then it is processed, if not, it is queued for future handling, once all previous messages have been received.

Because of the connectionless nature of the UDP protocol, DTLS, contrary to TLS, is vulnerable to several Denial Of Service (DOS) attacks with spoofed IP addresses. To mitigate this threat, the TLS handshake has been extended with a *cookie exchange technique*: before the server allocates resources for a new communication, the client must demonstrate its capability of receive packets addressed to its declared IP address. This is done by replying a *cookie* provided by the server through the HelloVerifyRequest (HVR) message, shown in Figure 3.5. In the first DTLS ClientHello message the new cookie field has zero length. The server,

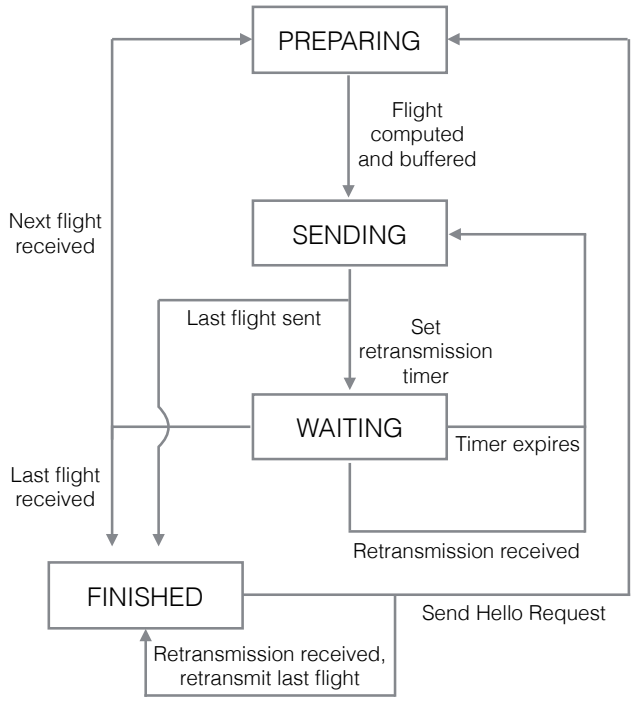


Figure 3.4: DTLS handshake retransmission state machine.

unable to verify it, sends the HelloVerifyRequest message containing a new cookie whose generation process must not allocate resources on the server, in order to avoid resource-consuming DOS attacks. Upon receiving the HelloVerifyRequest, the client retransmits the ClientHello with the received cookie added (indicated with CH*). This time the server can verify the cookie and is allowed to proceed with the handshake.

3.2.2 DTLS Record Protocol

Another incompatibility of the TLS running over unreliable transport protocols is related to the cipher modes: stream ciphers maintain residual state between encryption of records, requiring records to be decrypted in order without missing messages on the receiver side. In addition, the MAC of each record is calculated taking into account an implicit sequence number of the records, requiring again that messages be delivered in order and without losses, the functionalities not provided by UDP.

For these reasons, DTLS Record Protocol is not compatible with stream ciphers, allowing only block ciphers, and uses an explicit *record sequence number* field in

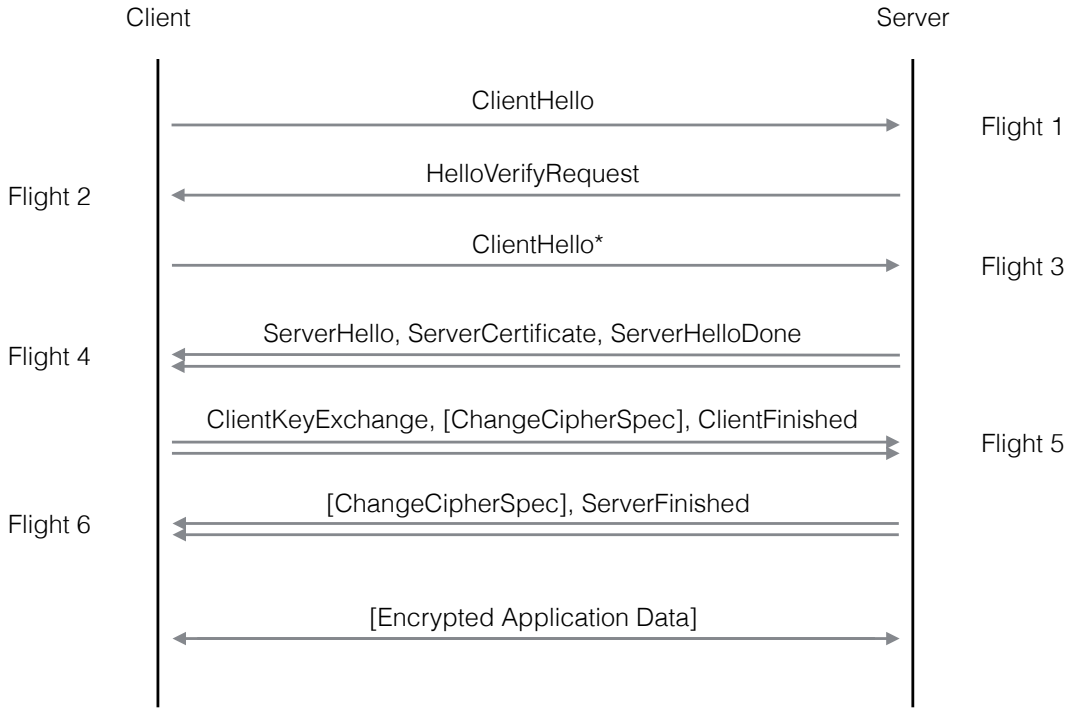


Figure 3.5: DTLS handshake with cookie exchange.

order to resolve the MAC issue.

Sequence number restarts from zero after every ChangeCipherSpec message, consequently causing confusion when several handshakes are performed in close succession. Indeed, if delay events occur, in the channel there can be multiple messages with the same sequence number but belonging to different cipher suites. For this reason, in addition to the sequence number, DTLS Record Protocol also introduces the *epoch* field, whose value is incremented at every ChangeCipherSpec message sent and allows endpoints to distinguish such messages.

Chapter 4

Environment Set-up

The environment used to realize the application developed in this thesis is, for practical reasons, the same environment used for the CoAP, DTLS and 6LowPAN libraries. In particular, the mote available at the Department of Information Engineering of the University of Padova at the start of this work was Zolertia Z1, also used as the testing board during CoAP and DTLS development. The programming language used to build the implementations of the aforementioned protocols is nesC, a dialectal form of the C programming language purposely created for the TinyOS embedded operating system.
Furthermore, this chapter introduces the devices used for implementing the whole protocol stack, the TinyOS operating system and also the system created to compile, install and debug the application source code.

4.1 Zolertia Z1 Module

The Z1 module, shown in Figure 4.1, is a general purpose development platform for wireless sensor networks designed for researchers and developers. This hardware platform does not require any external hardware to be programmed. Indeed, due to its built-in full USB capability, it can be directly programmed allowing easy integration with multiple systems. The mote is equipped with the following hardware:

- second generation MSP430F2617 low power microcontroller
- 16-bit RISC CPU, 16MHz clock speed, built-in clock factory calibration
- 8KB RAM
- 92KB flash memory

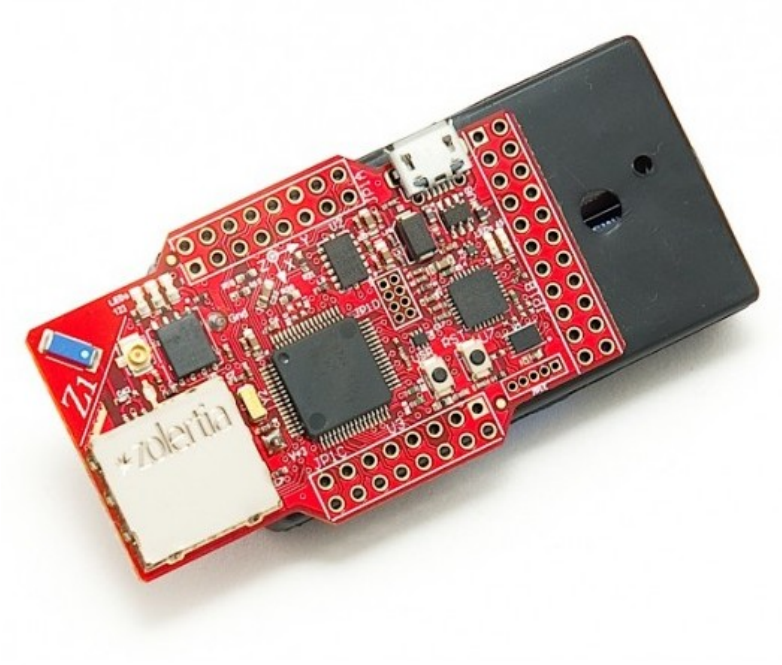


Figure 4.1: Zolertia Z1 module.

- CC2420 transceiver, IEEE 802.15.4 compliant, operates at 2.4GHz with an effective data rate of 250 Kbps
- digital programmable accelerometer (ADXL345)
- digital temperature sensor (TMP102)
- CP2102 USB-to-serial chip from SiLabs
- user and reset buttons
- three RGB LEDs.

The board can be powered by means of a battery pack, a coin cell, a USB cable or directly connected through a power source. It can be enhanced with other analog and digital sensors, for a total of up to 4 additional external devices.
Zolertia Z1 also supports two of the currently most employed open source embedded operating systems, TinyOS and Contiki, which can both take advantage of a huge online community support.

4.2 TinyOS and NesC

TinyOS is a BSD-licensed open source operating system for wireless embedded systems, designed to support the concurrency-intensive and low-power operations

required by the constrained devices of IoT networks. It supports several microcontroller families as well as multiple radio chips, and is composed mainly of a work scheduler and a set of drivers for the most common hardware of wireless embedded platforms.

4.2.1 TinyOS Executive Model

Due to the broad range of hardware capabilities of sensor nodes and their very limited RAM, one of the goals of TinyOS is to have a flexible hardware/software boundary. Synchronous code is a piece of code that runs in a single execution context thus reserving the CPU access and preventing other code (from hardware interrupts or events) execution until its completion, adversely affecting the mote responsiveness especially when the execution time is long. Rather than making everything synchronous, in TinyOS operations that are split-phase (or non-blocking) in hardware are split-phase in software too. Furthermore, all input/output operations that need more than a few hundred microseconds are asynchronous and have a down-call called *command*, that starts the operation, and a call-back, called *event*, that signifies operation completion. This feature enables the operating system to maintain high concurrency with one unique stack but, on the other hand, forces applications to have many small event handlers.
Due to the concurrency-intensive nature of typical operations of IoT networks, TinyOS provides a form of deferred procedure call, called *task*, that allows applications to postpone CPU-intensive computations. Tasks are non-preemptive and run in FIFO order, i.e. their codes run synchronously with respect to each other.

4.2.2 NesC Programming Language

The programming language adopted by the TinyOS operating system is nesC, an extension of the C programming language suited for the hardware limits of sensor networks and designed to embody the structuring concepts and execution model of TinyOS just described.
TinyOS code is statistically linked with the application code by a GNU toolchain. In these kind of programs, all the code is contained in a single executable module, and thus the efficiency of the library referencing operations is improved. However, as a drawback, the memory code size increases.
The model of TinyOS applications is component-based, which means that each application is composed of one or more software components interconnected by

interfaces. Components should embody simple functions while their connection should realize more complex functions. TinyOS provides built-in components and interfaces that represents hardware abstractions, such as packet communication, routing, sensing and storage.
All components have two code blocks: the first describes the component signature and the second its implementation. The implementation section divides components into two categories, modules and configurations, used interchangeably to build TinyOS applications. In the modules component, the implementation section consists of variables and functions, like a classic C program; in the configurations component instead the implementation section consists of nesC wiring code that connects components together.
Component signatures contain zero or more interfaces that the component can provide or use. Each interface describes a functional relationship between two or more different components and defines a set of functions divided into commands and events, that corresponds to the previously introduced TinyOS down-calls and call-backs. This distinction determines which component implements the function and which can call it: user components can call commands implemented by provider components, and conversely provider components can launch events managed by the user components. From this point of view, a component that provides an interface offers a service meant to be accessed by another component wired to it. Implementations of events and commands take place in the module implementation section.

In Figure 4.2 an example of interface wiring is shown: the BlinkC component make use of the Boot, Timer<TMilli> and Leds interfaces provided by MainC, TimerMilliC and LedsC components respectively. A grey triangle inside a rectangle indicates an interface provided by the relative component, while an external triangle indicates an used interface.

4.3 System Set-up

In order to program the Zolertia devices, the TinyOS libraries have to be installed on a compatible system. TinyOS can be installed on the most common operating system such as Linux and Windows or in a virtual machine emulating one of them. The best choice in terms of portability and complexity, thanks to its built-in features and to the great TinyOS support, is the Ubuntu Linux distribution. Specifically, the system is composed of a 32-bit Linux Ubuntu 12.04 LTS, with

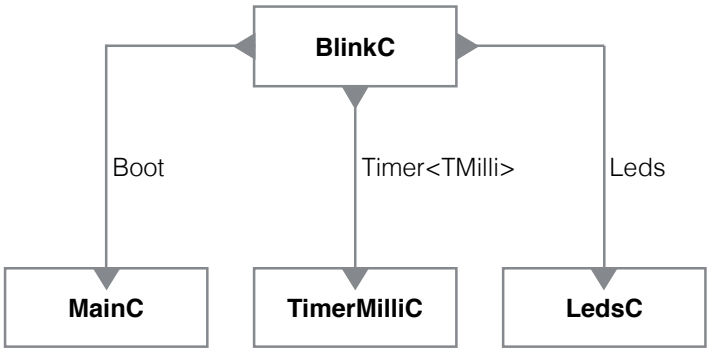


Figure 4.2: BlinkC wiring example.

kernel 3.8.0, virtualized by the VMware Fusion software running on a 2012 Macbook air with 1 GB of RAM and 2 virtual cores of the 1.8 GHz Intel Core i5. The TinyOS version installed on the linux distribution is 2.1.2, coupled with the GCC MSP430 compiler version 4.6.3. The procedure followed for the installation of TinyOS and the MSP430 toolchain is presented in [1].
Furthermore, two Zolertia Z1 motes have been connected and powered via two USB cables, completing the environment setup.

Chapter 5

BlinkToSCoAP Implementation

In Chapter 2 and Chapter 3 two fundamental IoT network protocols have been described, one providing web services and the other end-to-end security functionalities. This chapter presents an application called BlinkToSCoAP, intended to run over 6LoWPAN networks, which includes a DTLS secured CoAP implementation. In the following section, the TinyOS libraries that implement these protocols are briefly presented and then a detailed description the BlinkToSCoAP application is given. The last section of the chapter describes another application, called BlinkToCoAP, obtained by depriving BlinkToSCoAP of the security components. The purpose of this second application is to allow the evaluation of DTLS performance through the experiments that will be presented in the following chapter.

5.1 Protocol Libraries

This section contains a brief description of the libraries utilized to assemble the BlinkToSCoAP and BlinkToCoAP applications. These components implement a lightweight and unoptimized version of the protocols, and all their authors belong to the SIGNET Group of the University of Padova's Department of Information Engineering (DEI).

5.1.1 CoAP Protocol Library

The CoAP library has been developed by Angelo P. Castellani and Mattia Gheda. The code structure, depicted in Figure 5.1, is composed of a main component, called CoAP, and an auxiliary component called TimedPool. The first one provides two different interfaces, CoAPClient and CoAPServer, that offer to other components CoAP client and CoAP server functionalities respectively. While CoAP module handles all the CoAP request/response messages and all the protocol mechanisms,

the TimedPool component stores all the transaction data and implements a timed queue for message retransmissions. By default, this library is wired to the IP_6LP component, written by Matus Harvan, that provides UDP and IPv6/6LoWPAN protocol functionalities. However, BlinkToSCoAP does not utilize this component but uses instead the IPv6/6LoWPAN implementation developed within the SIGNET group.

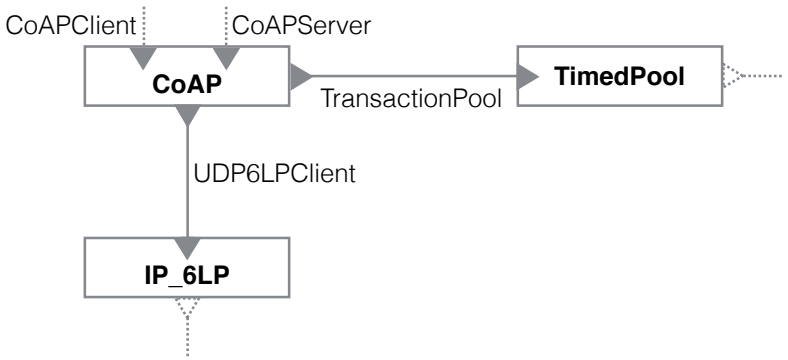


Figure 5.1: CoAP library structure.

The CoAP library is designed to handle, by default, up to 5 concurrent transactions for each mote. Since it implements only piggy-backed responses, there is no need to use two different identifiers to distinguish requests/responses and different message types. Furthermore the matching is done by means of a transaction ID, provided by the TimedPool component every time a new transaction is stored. In addition to this mechanism, the CoAP library uses different UDP port numbers for different transactions, realizing a second matching mechanism that can be adopted when separate responses are implemented.

5.1.2 DTLS Protocol Library

The DTLS library has been developed by Cristiano Tapparello. It is a lightweight implementation of the DTLS protocol and is based on the interconnection of multiple components with the *dtls* main module, which represents the entry point of DTLS implementation and contains all its logic required to handle a secure communication such as sessions data, handshake protocol definition and structures of different messages belonging to the security protocol.

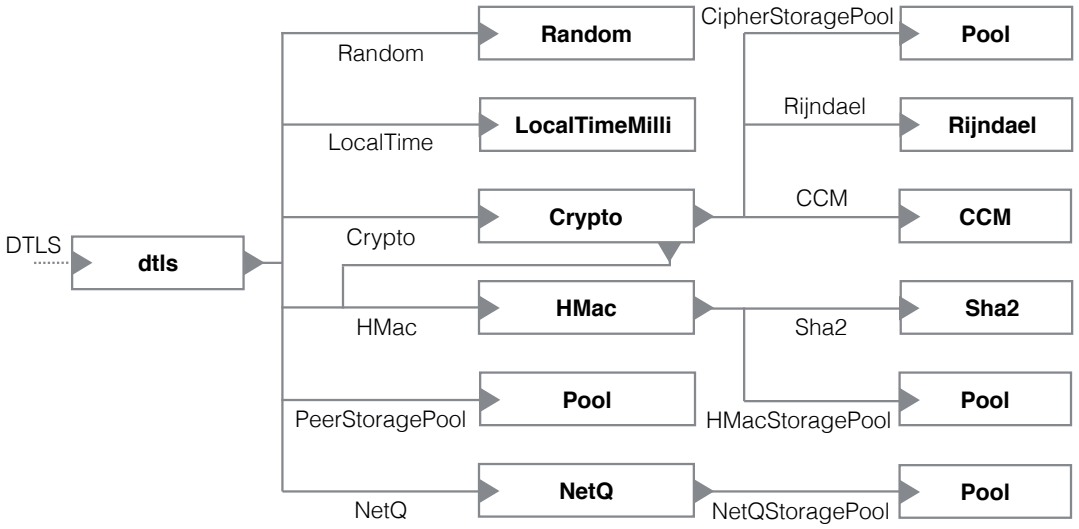


Figure 5.2: DTLS library structure.

The DTLS library structure is depicted in Figure 5.2 and is composed of several modules. Crypto is the cryptographic module that handles all authentication and encrypt/decrypt operations. Since this is a lightweight DTLS implementation, the crypto component provides the only cipher suite supported DTLS_PSK_WITH_AES_128_CBC_SHA-256, which is composed of the Pre Shared Key (PSK) exchange algorithm and the 128 bit Advanced Encryption Standard (AES) algorithm in Counter CBC-MAC (CCM) mode. AES is a symmetric block cipher based on the Rijndael algorithm that, in CCM mode, provides both message authentication and confidentiality. This algorithm is implemented and provided by the Rijndael and CCM components.
Message integrity and a second check for message authentication is achieved by a keyed Hash Message Authentication Code (HMAC), provided by the HMac component, which calculates a MAC through the 256-bit Secure Hash Algorithm (SHA-256) function in combination with a secret cryptographic key generated from the master secret, elaborated during the handshake phase.
The generation of the client and server random tokens that will contribute to the computation of the premaster secrets, as proposed by the DTLS standard, involves random values as well as the current time. These functionalities are provided by the Random module and the LocalTimeMilli component respectively.
The multiple Pool component instances are used to store concatenated lists of data structures: each component utilizes one instance in order to store their specific data.

The NetQ module implements a concatenated list meant to provide, coupled with a timer, reliability feature to handshake messages. However this feature is still not fully developed.
From Figure 5.2, it can be seen that, differently from the CoAP structure, the DTLS component does not have a vertical architecture. Instead of providing its functionalities upwards for components of higher layers, and utilize functionalities provided by lower layer components, the *dtls* component can be wired only through its unique interface DTLS.

The DTLS library implements the basic version of the DTLS Handshake Protocol, which consists of the message flights described in Table 5.1.

Flight number	Number of records	Handshake messages included
1	1	CH
2	1	HVR
3	1	CH*
4	1	SH, SHD
5	3	CKE, CCS, CF
6	1	CCS
7	1	SF

Table 5.1: DTLS library handshake flights.

The first three flights are the same as the DTLS basic handshake shown in Figure 3.5. The fourth flight instead is composed of one single record sent by the server that includes the ServerHello and ServerHelloDone messages. The fifth flight instead is composed of three different records, one for each handshake message involved, because the implementation requires that the ChangeCipherSpec is carried alone by a single record. In the end, once the server receives the client CCS message it changes the cipher specification and immediately sends back its own CCS message. In order to send its last handshake message, the server has to elaborate both the CF and SF messages, resulting in a non-negligible computation time. For this reason, the server's CCS and SF messages belong to different flights.

5.1.3 IPv6/6LoWPAN Protocol Stack Library

The implementation of IPv6/6LoWPAN enclosed in the BlinkToSCoAP and BlinkToCoAP applications is provided by the SiGLoWPAN library [3], developed by

Giulio Ministeri. The SiGLoWPAN architecture, shown in Figure 5.3, is composed of several protocols spanning from the transport layer to the IP adaptation layer, which stands between the network and the Medium Access Control (MAC) layers. UDP and ICMPv6 implementations take place at the fourth layer, the third layer contains the IPv6 component with other supporting modules and at the bottom layer there are 6LoWPAN and IP adaptation components, which perform the adaptations required to transmit IP messages over specific link layer protocols such as IEEE 802.15.4 or Point-to-Point Protocol (PPP). This library is enhanced by an advanced memory management approach, the link layer independence and an optimized memory footprint.

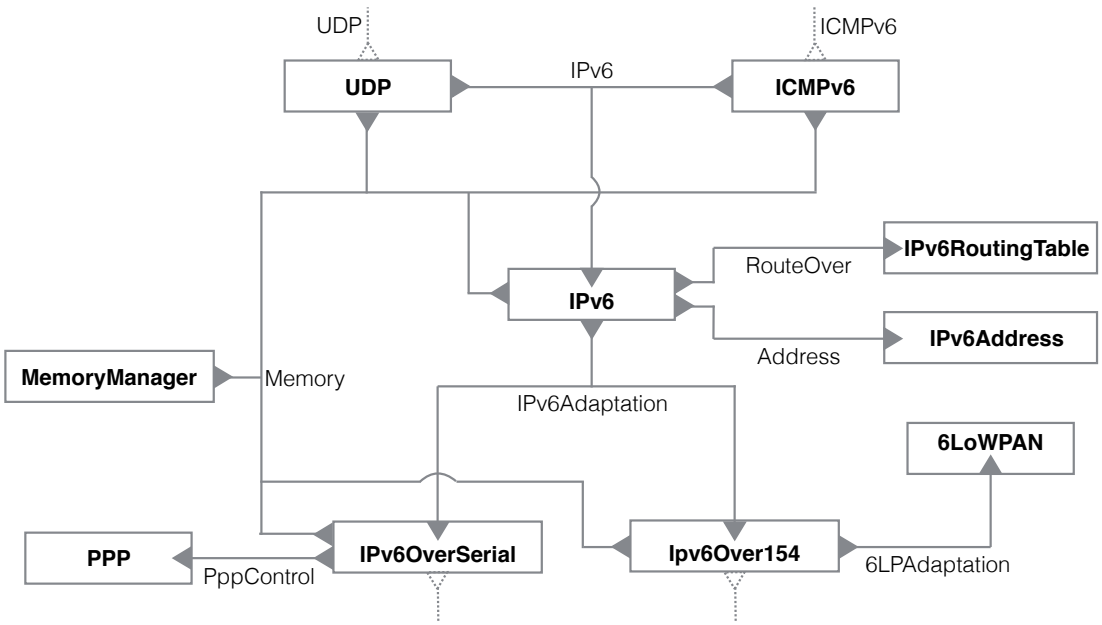


Figure 5.3: SiGLoWPAN architecture.

The link layer independence is achieved through the aforementioned IP adaptation layer, at the lowest level of the SiGLoWPAN stack, while the advanced memory management approach of SiGLoWPAN is provided by a very useful component called MemoryManager. Instead of allocating the necessary static buffers inside of each TinyOS module that has to be big enough to accommodate the maximum size of data supported by the protocol represented, all the SiGLoWPAN modules share a fraction of the available RAM that is statistically allocated to the MemoryManager component at link time. When necessary, the module dynamically reallocates this memory to the requesting component.

Other details of the SiGLoWPAN stack are beyond the scope of this thesis and can be found in [3].

5.2 BlinkToSCoAP Application

The design of the BlinkToSCoAP application architecture, depicted in Figure 5.4, was driven by the pre-existent wiring structures of the three libraries introduced earlier. The development of this application, discussed below, has involved mainly the BTSCTest, CoAP, SSLP, dtls and MemoryManager components.

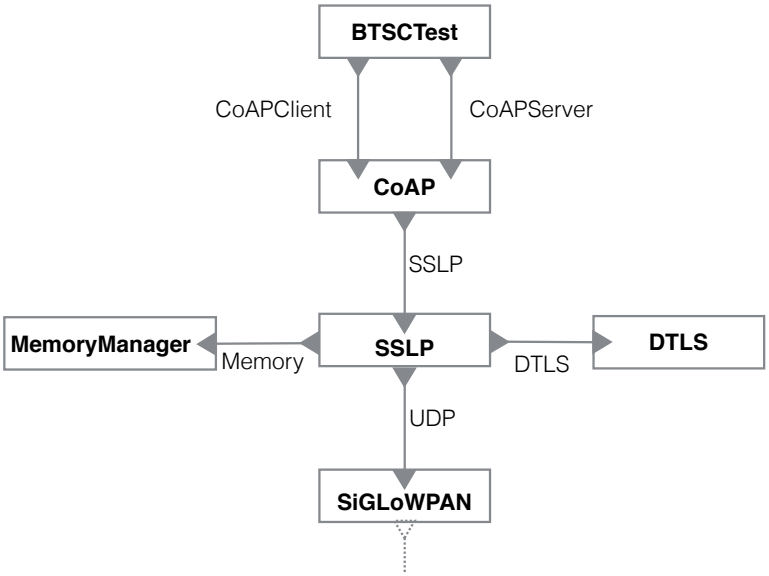


Figure 5.4: BlinkToSCoAP architecture.

5.2.1 BTSCTest Component and Wiring

At the top of the BlinkToSCoAP stack there is the BTSCTest component, designed to test the whole application and to act as a CoAP Server or Client. The role interpreted by this component depends on the IPv6 address of the mote, assigned during the compiling phase: one specific address is assigned to the server, the other belongs to clients. If it acts as a CoAP client, it sends a Blink request to the CoAP server. If otherwise, the component interprets a CoAP server and waits for a Blink request. Every time it correctly receives this kind of a request, an internal counter is incremented and sent back to the client carried by the server's response. Both server and client nodes can optionally turn on the LEDs

of the Zolertia platform in a configuration that represents the binary value of the exchanged counter. Therefore, to exploit the CoAP functionalities, the BTSCTest component is wired to the CoAP module through both of its interfaces, namely CoAPClient and CoAPServer.
The BTSCTest module, as wired to the MainC component via the Boot interface, is the first component that has the control of the node. This includes responsibility of the initialization of all the necessary circuitries, modules and variables. For this reason, in order to activate the radio and all the SiGLoWPAN variables, BTSCTest is wired to the IPv6 module of the SiGLoWPAN library through the SplitControl interface that is intended to be used in contrast for this purpose. The IPv6 component then handles the initialization of all the SiGLoWPAN library plus the radio circuitry. The BTSCTest wiring is illustrated in Figure 5.5.

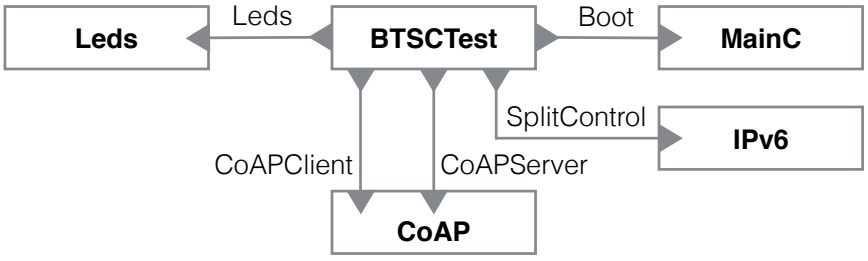


Figure 5.5: BTSCTest wiring.

5.2.2 CoAP Wiring

The BTSCTest component exploits the CoAP module functionalities by means of the CoAPClient and CoAPServer interfaces.

The CoAPClient interface, shown in Figure 5.6, provides only a request command and a response event. The first function returns the transaction ID of the created transaction used to further match the server response or a NULL value if some errors occurs, and requests three parameters:

- *absuri - location of the C structure containing URI, destination host IPv6 address and destination UDP port number;
- method - code representing one of the CoAP methods (GET, POST, PUT, DELETE);

- *content - location of the memory buffer containing the payload of the CoAP request.

The response event is launched by the CoAP module when a received message is correctly recognized as a server response to a previously sent client request. It provides to the event handler function the following data:

- tid - transaction ID of the response transaction that must match the tid of the relative request;
- status - code representing the server response, as described in Chapter 2 ;
- *content - location of the buffer containing the payload of the CoAP response.

```
1 interface CoAPClient {
2
3      command coap_tid_t request (
4           coap_absuri_t* absuri,
5           coap_method_t method,
6           coap_content_t* content,
7           bool acked
8      );
9
10     event void response (
11          coap_tid_t tid,
12          coap_status_t status,
13          coap_content_t* content
14     );
15 }
```

Figure 5.6: CoAPClient interface.

The CoAPServer interface is shown in Figure 5.7 and provides, in contrast to the CoAPClient interface, the response command and the request event. The request event is signaled by the CoAP module when a received packet has been recognized as a new client request, and provides the following parameters to the component that interprets the CoAP server role:

- rid - transaction ID of the received request;
- *uri - location of the requested resource's URI;
- method - code that represents the requested method;
- *content - location of the buffer containing the payload of the received CoAP request;

- toack - boolean value that indicates if the request was carried by a CON or NON message.

When the application component finishes to elaborate the request, the response command is called to transmit the results of the elaboration via a CoAP response. The function requires the following data as parameters:

- rid - transaction ID of the response;
- status - code that represents the server response;
- *content - location of the buffer containing the payload of the CoAP response.

```
1 interface CoAPServer {
2
3      event void request (
4           coap_rid_t rid,
5           coap_absuri_t* uri,
6           coap_method_t method,
7           coap_content_t* content,
8           bool toack
9      );
10
11     command error_t response (
12          coap_rid_t rid,
13          coap_status_t status,
14          coap_content_t* content
15     );
16
17 }
```

Figure 5.7: CoAPServer interface.

At the bottom of the original CoAP library structure, the CoAP component is interfaced with the UDP6LPClient interface, shown in Figure 5.8, in order to send CoAP messages by means of the UDP protocol. The UDP6LPClient interface provides one command, sendTo, and two events, sendDone and receive. The sendTo function requires the following parameters to send a CoAP message through the UDP protocol:

- *addr - location of the destination IPv6 address;
- port - destination UDP port number;
- *buf - location of the UDP payload to send;
- len - length of the UDP payload expressed in number of bytes.

Once the lower layer finishes the sending process or encounters an error, the event sendDone is launched. It gives the result of the operation and the pointer to the payload sent, in order to eventually free the memory allocated to it.
When a new packet arrives, the component providing the UDP6LPClient interface launches the receive event to provide the following information:

- *addr - location of the source IPv6 address;
- port - source port;
- *buf - location of the UDP payload received;
- len - length of the received UDP payload expressed in number of bytes.

```
1  interface UDP6LPClient {
2
3      command error_t sendTo(const ip6_addr_t *addr,
4                             uint16_t port,
5                             const uint8_t *buf,
6                             uint16_t len
7      );
8
9      event void sendDone(error_t result,
10                         void* buf
11     );
12
13     event void receive(const ip6_addr_t *addr,
14                        uint16_t port,
15                        uint8_t *buf,
16                        uint16_t len
17     );
18 }
```

Figure 5.8: UDP6LPClient interface.

The direction of the functional relationship established by this interface between the CoAP module and the UDP/IPv6 stack has to be changed in order to include both the DTLS security protocol as well as the SiGLoWPAN library in the BlinkToSCoAP application. The fact that the DTLS interface provided by the DTLS library is completely different from the UDP6LPClient interface, and the horizontal architecture of the library, has led to the definition of a new module. This component intercepts the CoAP transmission requests in order to elaborate them via the DTLS interface and, once finished, redirect the encrypted data to the UDP module of the SiGLoWPAN library. Vice-versa, when a new UDP datagram is received it intercepts the payload, passes it to the DTLS protocol and redirects the eventual decrypted application data to the CoAP component. Therefore, this

new module is directly wired with CoAP, SiGLoWPAN and DTLS components, acting like a gateway for the data flowing between them.

5.2.3 SSLP Component and DTLS, SiGLoWPAN Wiring

The new component, called Secure SiGLoWPAN (SSLP), abstracts the functionalities of the DTLS protocol and the SiGLoWPAN stack, also resolving some issues of the DTLS library architecture. The first problem, already introduced in the DTLS library overview, stands on its horizontal architecture. Rather than taking care of plain data by securing and passing it to lower layer for the transmission and signaling of the operations concluded through specific events, the DTLS interface is designed to provide security functionalities through its commands and to return the result of operations done by means of its signaling events. This structure makes the DTLS library highly modular and completely independent of other components, but at the same time forces the component wired to the DTLS interface to handle both the plain data as well as the encrypted data ready for transmission. In addition, the DTLS interface involves another issue of the security component: DTLS data structures are not restricted to the DTLS library. Commands and events have, as parameters, the main data structures that contain variables strictly belonging to the protocol.
Therefore the SSLP component is designed to handle the plain data coming from the CoAP library, the obfuscated data from the DTLS module ready to be transmitted through the SiGLoWPAN stack and also the DTLS data structures such as the context, peers and sessions. Its wiring scheme is depicted in Figure 5.9, where it can be seen that SSLP provides the SSLP interface, which is basically the previously presented UDP6LPClient interface renominated. In this way, the CoAP module does not have to be modified to handle different primitives and the new name of the interface clarifies that it abstracts a DTLS secured IPv6/6LoWPAN stack.

Moreover, SSLP is wired to the DTLS library through the DTLS interface presented in Figure 5.10, in order to handle all its security functionalities. At the start-up of the platform, the first action that has to be done at the security layer is to initialize the dtls context that will be used to store buffers, list of connected endpoints and session states. This data structure is allocated and returned by the dtls_new_context command, that accepts as parameter a memory address intended to be the physical location of the application buffer containing the data which has to be secured and transmitted.

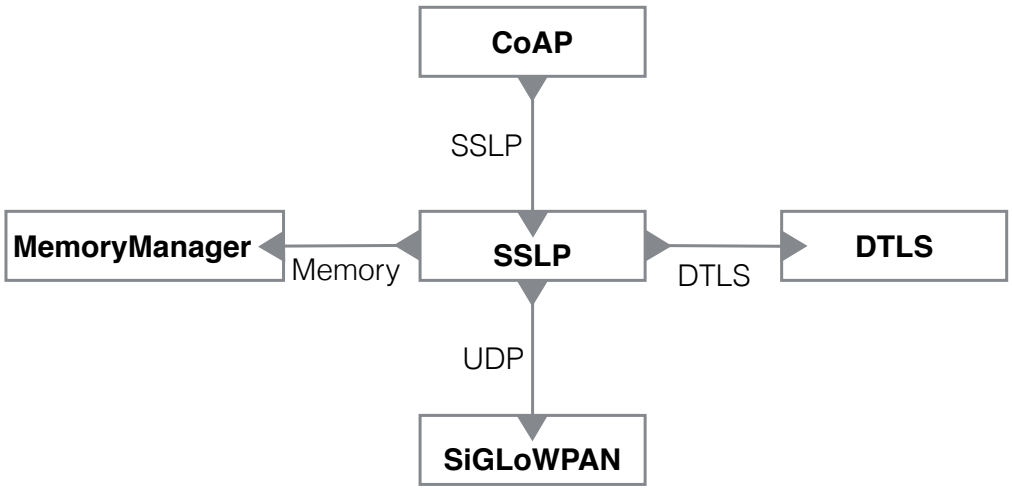


Figure 5.9: SSLP wiring

When a CoAP packet is received, the SSLP component checks if the destination of the message has already a DTLS connection active via the dtls_isConnected command, that returns the data structure of the peer if found or a null pointer otherwise. This command needs the following parameters in order to execute the search:

- *ctx - location of the DTLS context structure;
- *dst - location of the data structure containing information of the destination address such as its IPv6 address and UDP port.

If the SSLP component receives a null pointer from the command, the DTLS protocol has to establish a new session or recover an expired one. The dtls_connect command of the DTLS interface starts a new handshake with a remote host, specified as a parameter, in order to establish the secure channel. This function accepts the same parameters described for the dtls_isConnected command, and returns an integer value that indicates eventual errors.
If, otherwise, the session is found active, the message can be immediately handled by the DTLS component. This is done by the dtls_write command that needs, in addition to the previously described parameters of the dtls_isConnected function, the location of the CoAP message and its length expressed in number of bytes.
Once the DTLS module has finished the elaboration of the message, it signals the send_to_peer event in order to provide to the SSLP component the pointer of the encrypted data ready for transmission. The parameters of this function are the

```
1  interface DTLS{
2
3       command int16_t dtls_connect(dtls_context_t *ctx,
4                                    const session_t *dst
5                                    );
6
7       command dtls_context_t * dtls_new_context(void *app_data);
8
9       command int16_t dtls_write(struct dtls_context_t *ctx,
10                                   session_t *dst,
11                                   uint8_t *buf,
12                                   size_t len
13                                   );
14
15      command int16_t dtls_handle_message(dtls_context_t *ctx,
16                                          session_t *session,
17                                          uint8_t *msg,
18                                          int16_t msglen
19                                          );
20
21      command dtls_peer_t * dtls_isConnected(dtls_context_t *ctx,
22                                          session_t *dst
23                                          );
24
25      event int16_t read_from_peer(struct dtls_context_t *ctx,
26                                   session_t *session,
27                                   uint8_t *data,
28                                   size_t len
29                                   );
30
31      event int16_t send_to_peer(struct dtls_context_t *ctx,
32                                   session_t *session,
33                                   uint8_t *data,
34                                   size_t len
35                                   );
36
37      event int16_t get_key(struct dtls_context_t *ctx,
38                                const session_t *session,
39                                const unsigned char *id,
40                                size_t id_len,
41                                const dtls_key_t **result
42                                );
43
44      event int16_t signal_event(struct dtls_context_t *ctx,
45                                session_t *session,
46                                dtls_alert_level_t level,
47                                uint8_t code
48                                );
49 }
```

Figure 5.10: DTLS interface.

same as that of the dtls_write command: the dtls context, the session structure of transmission, the memory address of the obfuscated message and its total length.
When a new message arrives from the UDP protocol, the SSLP component expects that it is a DTLS packet. Furthermore, it passes the data directly to the DTLS module through the dtls_handle_message command.
If DTLS recognizes the received packet as a valid application message, it decrypts it and launches the read_to_peer event, where the SSLP forwards the decrypted data to the CoAP layer. If the received message is recognized as a DTLS alert message, the security module signals it via the signal_event function, which provides as parameters the code and the level of the alert message in addition to the dtls context and the session structure. This particular functionality is not yet implemented in the code. As the last case, if the received packet is a handshake message, the DTLS library will process it and follow the handshake protocol specification.
The last function of the DTLS interface is the get_key event, used to retrieve the pre-shared key and its ID from the mote. Actually, the key recovery functionality is not yet implemented, and the operation is thus bypassed by the SSLP module by providing the values written directly on its source code.

In order to transmit messages over the wireless channel, the SSLP component is interfaced with the SiGLoWPAN stack through the UDP interface shown in Figure 5.11. To exploit the primitives of the UDP module, the SSLP component

```
1 interface UDP  {
2
3      command error_t send (memory_id_t ID,
4                            slp_ip6_entry_t* dest,
5                            uint16_t dstPort);
6
7      event void sendDone (memory_id_t ID,
8                            error_t error);
9
10     event void receive (memory_id_t ID,
11                         slp_ip6_entry_t* src,
12                         uint16_t srcPort);
13 }
```

Figure 5.11: UDP interface.

needs to adopt the same mechanism used by the SiGLoWPAN library to handle shared buffers between its components. The MemoryManager component makes its reserved RAM virtual by assigning a virtual memory ID (vmID) to each specific memory allocation rather than identifying it with its memory address. The vmID

can be shared between layers avoiding the need for static memory allocations, since it is a global identifier for the buffer space. MemoryManager provides its functionalities through the Memory interface, shown in Figure 5.12. This interface provides several commands: the alloc and smartalloc functions allocate a given number of RAM bytes and return its virtual memory identifier, the release of a vmID is performed by the free command and the id2p converts a vmID into a physical address. The last two commands of the memory interface are designed to

```
1 interface Memory {
2
3       command memory_id_t alloc(memory_size_t size);
4
5       command memory_id_t smartalloc (memory_size_t size, uint8_t
            layer);
6
7       command void free(memory_id_t id);
8
9       command void * id2p(memory_id_t id, memory_size_t * size);
10
11      command error_t realloc(memory_id_t id, int16_t size);
12
13      command error_t hrealloc(memory_id_t id, int16_t size);
14
15 }
```

Figure 5.12: Memory interface.

resize memory buffers: the realloc function appends at their ends a given number of bytes, while hrealloc adds bytes in front of the buffers. This last function is particularly useful when new headers need to be added to memory buffers containing payloads coming from upper layers.

The MemoryManager module provides to the SSLP component the necessary tools to be correctly interfaced with the SiGLoWPAN library: every time a transmission is requested by the DTLS protocol, the SSLP module converts the pointer of the buffer containing the data to send into a vmID, successively passed to the UDP interface. On the receive handler function of the SSLP instead the reverse operation is done in order to obtain a physical address of the UDP payload, successively handled by the DTLS component.

5.2.4 Practical Issues

During the development of the BlinkToSCoAP application, some practical issues have been encountered, as described below.

Compiler dependent implementation Before the BlinkToSCoAP application has been finished, when the DTLS component was being analyzed and tested, the system setup had installed the most recent version of the GNU mspgcc toolchain, specifically the 4.7.0 version. This was causing the failure of the DTLS decryption process, problem no more encountered if the 4.6.3 compiler was used. This issue has evidenced the presence of some compiler-dependent function or operation of the DTLS encrypt/decrypt section. Actually this issue is not yet resolved.

Data structure incompatibilities The first issue is related to the data structure incompatibility between the different libraries used to assemble the whole application. In particular, these incompatibilities have taken place in the SSLP component where the destination endpoint is represented by the DTLS component as a session_t structure and by the UDP module as a slp_ip6_entry_t structure.
In order to fix this problem, the optimal solution should have been the conversion of all the destination representations to an unique data structure, either modifying the DTLS or the SiGLoWPAN source code. Unfortunately, this involves a lot of modifications and time, independently from the data structure chosen, and for this reason a sub-optimal solution has been adopted: the SSLP module performs data structure conversion every time it is needed.

RAM usage The total RAM available on the Zolertia Z1 platform is 8KB. Once the BlinkToSCoAP application has been finished, with the configurations of all the libraries set to their default values, the compilation procedure was failing due to a RAM region overflowing of about 500 bytes. In order to reduce the memory consumption of the application, some configuration parameters have been reduced, more specifically, in the CoAP library:

- COAP_MAX_CONNECTIONS = 2 - maximum CoAP contemporaneous active transactions;

while in the SiGLoWPAN library:

- SLP_IPV6_QUEUE = 3 - maximum queue dimension for IPv6 packets;

- SLP_IPV6_MAX_ADDR = 1 - maximum number of IPv6 addresses a node can have at the same time;
- SLP_ROUTES = 3 - number of entries in the SiGLoWPAN IPv6 routing table;
- SLP_MEMORY_SIZE = 500 - number of RAM bytes dedicated to the dynamic management of the MemoryManager component.

In addiction of these modifications, the DTLS has been optimized by eliminating some redundant code and variables.

UDP ports As already mentioned, the CoAP library implements a request/response matching mechanism based on different adjacent UDP port numbers. Unfortunately, the SiGLoWPAN implementation uses these parameters to offer multiple instances of the UDP interface, allowing multiple components to be linked to the same protocol but forcing them to use a predetermined number of UDP ports. Since the CoAP library implements only piggy-backed responses, this matching mechanism is momentarily disabled until separate responses are added to implementation. The CoAP module therefore uses a single UDP port to handle all its transactions.

5.3 BlinkToCoAP

The BlinkToCoAP application is equivalent to BlinkToSCoAP, except for the presence of the DTLS security layer. Its structure is shown in Figure 5.13. Since its purpose is to evidence the performance variation brought in by the DTLS module, the BlinkToCoAP application is designed to be as close as possible to its secured version. Specifically, their differences involve only the wiring of the CoAP module, which – in this application – is interfaced directly to the UDP protocol. As can be seen from the BlinkToCoAP architecture, and as already discussed above, CoAP is wired to the MemoryManager component in order to correctly relate with the SiGLoWPAN library.

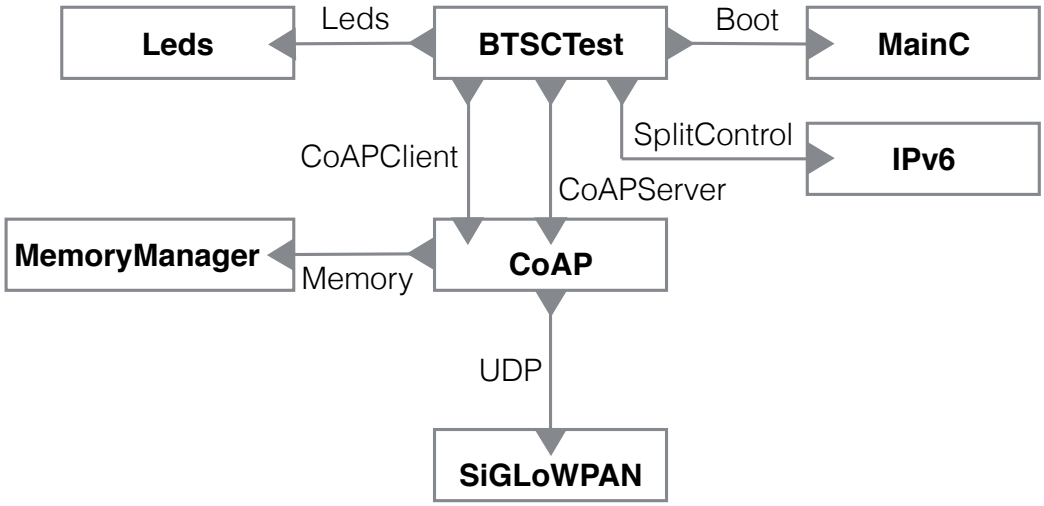


Figure 5.13: BlinkToCoAP Architecture.

Chapter 6

Performance Analysis and Results

An experimental evaluation campaign is performed using the proposed BlinkToSCoAP and BlinkToCoAP implementations in a 6LoWPAN network to evaluate the performance variation due to the DTLS security operations. The network setup consists of two Zolertia motes that communicate directly through the radio by means of the IEEE 802.15.4 Medium Access Control (MAC) protocol with no Radio Duty Cycling (RDC) enabled. The performance is evaluated in terms of parameters such as energy consumption, execution time, packet overhead and memory footprint, whose experiments are described in the following sections together with their results.

6.1 Memory Footprint

The memory footprint of the BlinkToSCoAP and BlinkToCoAP applications is provided by the GCC MSP430 Toolchain, which during the Zolertia device programming phase, displays the total bytes that are written in to the ROM and RAM memories. Since the two applications differ only in terms of presence of the DTLS protocol, the difference in memory utilization precisely provides the memory space occupation of the SSLP component and DTLS library.

Memory Footprint Results

The memory footprints provided by the GCC MSP430 Toolchain, used to compile and link the code, are shown in Table 6.1. The SSLP and DTLS components that include all the cryptographic functionalities and the DTLS state-machine, require 16298 bytes of ROM and 2428 bytes of RAM, which represent 36% and 44% of RAM and ROM usage of the BlinkToSCoAP application respectively. Figure 6.1 and Figure 6.2 show graphical representations of these results along with free Zolertia Z1 memory dimensions still available for use.

Application	RAM [bytes]	ROM [bytes]
BlinkToSCoAP	6832	37360
BlinkToCoAP	4404	21062

Table 6.1: Memory footprint.

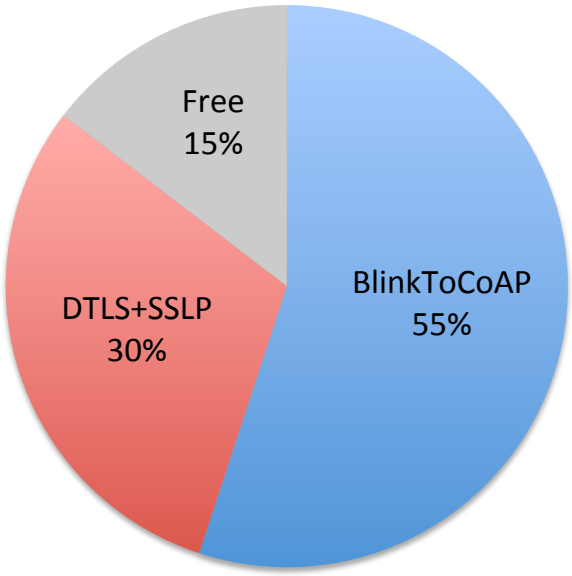


Figure 6.1: DTLS RAM memory overhead.

6.2 Packet Overhead

The packet overhead experiment involves a wireless packet sniffer board interfaced to a Linux system running Wireshark – a packet analyzer software.
This tool is used to capture packets exchanged over the air, between the two nodes under test, such as handshake messages, CoAP secured transactions and CoAP unsecured transactions. Wireshark is capable of parsing the captured data in order to distinguish the various protocols headers, providing access to all their fields as well as their sizes and, in particular, the effective DTLS header dimension. Moreover, this experiment also gives additional time measurements of the message

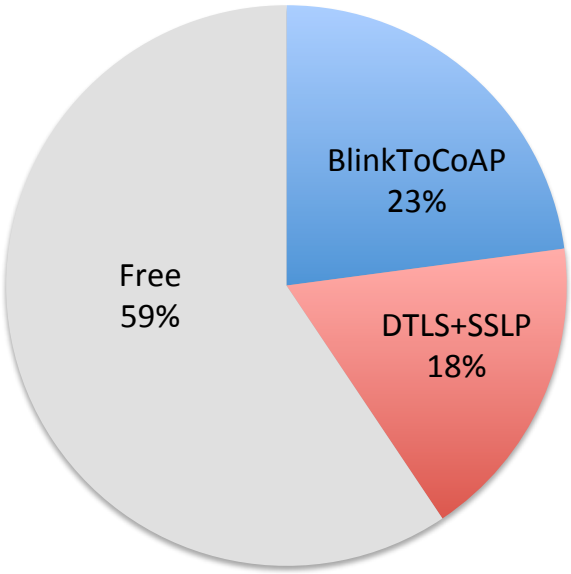


Figure 6.2: DTLS ROM memory overhead.

elaborations and computations, since packet transmissions are events that occur either before or after CPU-intensive periods that correspond to their elaborations.

Packet Overhead Experiment Results

Table 6.2 shows the frame and UDP payload dimensions for unsecured CoAP transactions, whereas Table 6.3 depicts secured CoAP transactions and finally Table 6.4 shows all the handshake messages.

	Frame length [bytes]	UDP payload length [bytes]
Request	30	13
Response	24	7

Table 6.2: Unsecured CoAP transmission lengths.

From the values reported, it can be noticed that the protocols below the application layer add a total overhead of 17 bytes to each frame, while the DTLS protocol adds 29 more bytes. This overhead drastically reduces the 102 octets maximum frame size available at the media access control (without link-layer

	Frame length [bytes]	UDP payload length [bytes]
Request	59	42
Response	53	36

Table 6.3: Secured CoAP transmission lengths.

	Frame length [bytes]	UDP payload length [bytes]
ClientHello	84	67
ClientHelloVerify	61	44
ClientHello (with cookie)	100	83
ServerHello + ServerHelloDone	105	88
ClientKeyExchange	59	42
ChangeCipherSpec	31	14
ClientFinished	70	53
ChangeCipherSpec	31	14
ServerFinished	70	53

Table 6.4: Handshake message lengths.

security) [12]. Figure 6.3 shows the graphical representation of the DTLS header and CoAP message sizes.

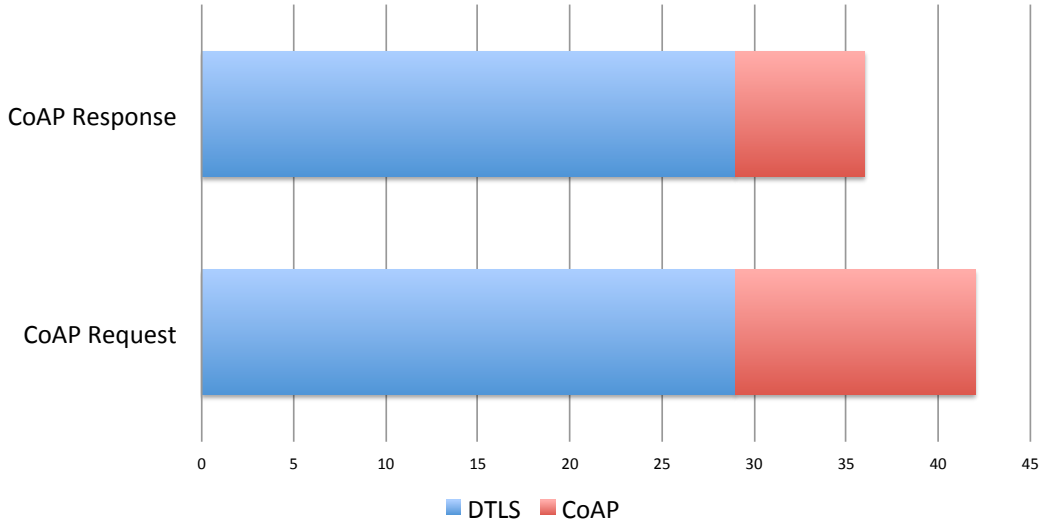


Figure 6.3: DTLS overhead expressed in bytes.

6.3 Energy Consumption

Energy consumption of the employed hardware platform is obtained through experimental measurements of the voltage across a current sensing resistor of 32.8 Ω, placed in series with the Zolertia Z1 board and the USB cable used to supply it, as illustrated by the electrical schematic depicted in Figure 6.4. They are measured using the UTD2102CEL Digital Storage Oscilloscope, characterized by the parameters shown in Table 6.5.

Parameter	Value
Real time sampling rate	$1GS/s$
Bandwidth	$100MHz$
Vertical sensitivity	$1mV \sim 20V/div$
Scan time base	$2ns \sim 50s/div$

Table 6.5: UTD2102CEL oscilloscope parameters.

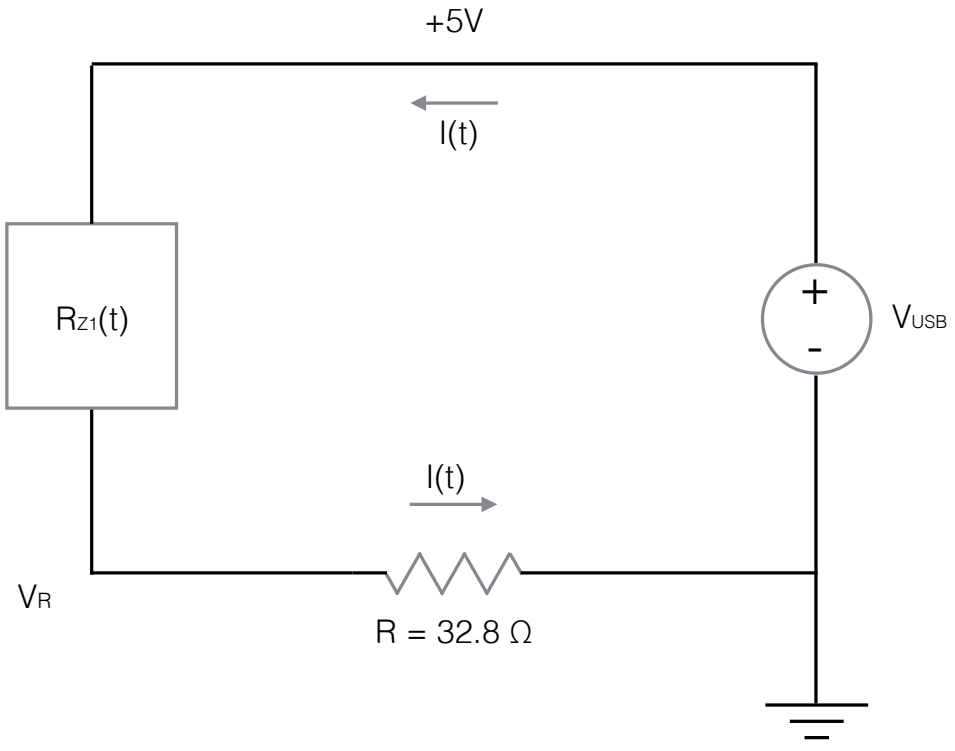


Figure 6.4: Electrical schematic for energy measurement.

The platform can be assumed as a time variable resistor $R_{Z1}(t)$. Its variation affects both its drain current as well as its voltage: if $I(t)$ is the current and $V_{USB} = 5V$ the voltage supplied by the USB cable, $R = 32.8\ \Omega$ the sensing resistor and $R_{Z1}(t)$ the variable resistance of the Zolertia mote, then the Kirchhoff and Ohm laws impose the following equation:

$$V_{USB} = V_R(t) + V_{Z1}(t) \tag{6.1}$$

$$= (R + R_{Z1}(t))I(t) \tag{6.2}$$

$$\Rightarrow I(t) = \frac{V_{USB}}{R + R_{Z1}(t)} \tag{6.3}$$

From the last equation can be seen that if $R_{Z1}(t)$ increases, $I(t)$ decreases and vice-versa. Moreover, the variance of the platform equivalent resistance is sensed by the sensing resistor because its voltage is obtained through the voltage divider equation $V_R = V_{USB}\frac{R}{R+R_{Z1}(t)}$.

By default, the adopted platform is intended to run at 3 Volts [17], so the 5 Volts provided by the USB cable are reduced to the correct value by means of an Automatic Voltage Regulator (AVR) integrated on the mote's CP2102 (USB-to-UART bridge) chip, shown in Figure 6.5.

The AVR chip causes a performance loss in terms of power consumption: this inefficiency is verified considering the current provided by the USB cable, while the Zolertia mote is only listening the wireless channel for new packets, thus suppyling only the radio circuitry. As reported in [8], the CP2102 transceiver requires 18.8 mA in listening mode, but the average current flowing across the sensing resistor, calculated from its average voltage during the inactivity periods, is about 19.7 mA. Since all the other components drain only a few µA while inactive, the residual 0.9 mA is supposed to be drained by the voltage regulator. Therefore, a lower bound of the power efficiency coefficient for the AVR can be estimated as:

$$\eta_{AVR} \geq 1 - \frac{P_{measured} - P_{ideal}}{P_{ideal}} = 0.953 \tag{6.4}$$

where, $P_{measured}$ is the power consumed by the board, calculated with the average measurements of the sensor resistor voltages, and P_{ideal} is the power consumption of the radio circuitry calculated with its nominal current requirements. One cause of the voltage regulator inefficiency can be associated to its leakage current, consumed internally and thus not available to the load, which has a typical value of about 25 µA [11].

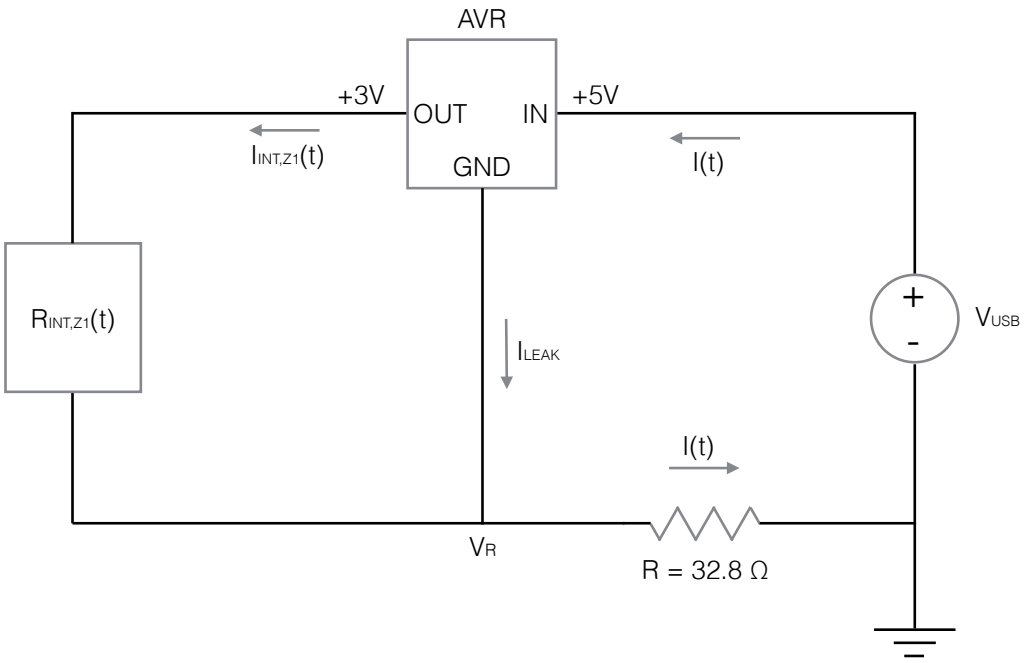


Figure 6.5: Energy experiment electrical schematic with AVR.

The AVR chip inefficiencies thus slightly increment the energy consumption of the Zolertia nodes. However, since the performance evaluation is focused on the impact of the DTLS operations on response time and energy consumption, this performance loss due to the AVR chip will not affect the evaluation of the higher energy requirements of BlinkToSCoAP with respect to BlinkToCoAP. For this reason this issue will be neglected in the following considerations and the estimation of the Zolertia Z1 energy consumption, for a given operation of duration T, is calculated as:

$$E_{Z1} = P_{Z1}T \tag{6.5}$$

$$= V_{Z1}IT \tag{6.6}$$

$$= (V_{USB} - V_R)\frac{V_R}{R}T \tag{6.7}$$

where, $E_{diss,Z1}$ and $P_{diss,Z1}$ are respectively the energy and the power dissipated by the hardware platform.

The experiment is subdivided into three subsections, each of them involving a different communication mode:

- CoAP secured and unsecured transmissions;

- DTLS handshake phases.

CoAP Secured and Unsecured Transmissions

In this experiment the two nodes have already established a secure channel by means of the DTLS Handshake Protocol. The first goal to achieve is to understand the time behaviour of the resistor voltage: theoretically, the client mote activity should be represented on the oscilloscope behaviour as a step, caused by the incremented power need of the Zolertia platform due to the request elaboration and transmission, followed by a certain period of inactivity time where the mote is awaiting the server response and another step that signals the request receipt and elaboration. In a complementary manner, the oscilloscope sensing the server node's activity should display an unique step due to the request reception and elaboration as well as the response computation and transmission.
In order to evidence DTLS operations, the SSLP component is configured, for this experiment, to turn on a LED every time the DTLS module is called to handle a message, involving operations such as encryption, decryption, message computation and message parsing. When the security operations are signaled as concluded, the LED is turned off. Furthermore, while the first probe of the oscilloscope senses the voltage of the sensing resistor, the second one measures the activity of the toggling diode. The behaviour displayed by the electrical tool for a secured transmission while studying a CoAP client is shown in Figure 6.6.

The measurements in yellow indicate LED activity: the steps correspond to the CPU time used by the DTLS library for its security operations. Moreover, request transmission and response receipt are clearly visible on the blue samples that indicate the first oscilloscope probe measures. The security calculations are evidenced by both the diode activity as well as its energy consumption, noticeable on the blue samples as a second step over the main one, which instead indicates the high CPU utilization. The resistor voltage also shows the particular Tx and Rx operations, recognized as two consecutive troughs clearly visible after the elaboration of the packet to send and also present – in a less clear form – just before response elaboration. Between the elaboration of the CoAP request and its transmission, it is also possible to recognize the fraction of time where the radio circuitry of the Zolertia mote is sensing the wireless channel in order to avoid packet collisions.

The same experiment, applied to the CoAP server mote, yields the behaviour depicted in Figure 6.7, which clearly evidences the CPU time phases belonging to

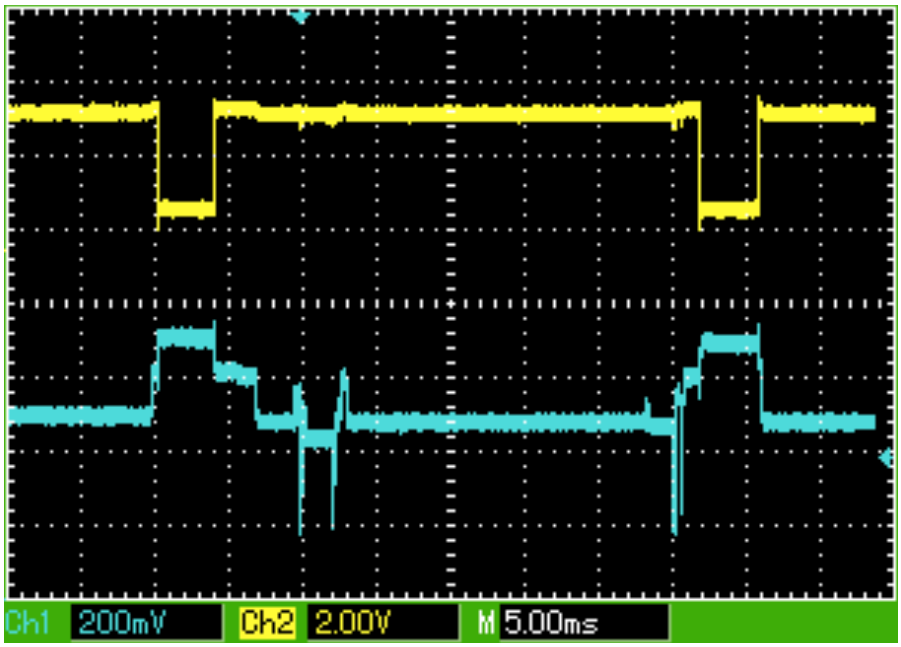


Figure 6.6: Client secured transmission and LED activity.

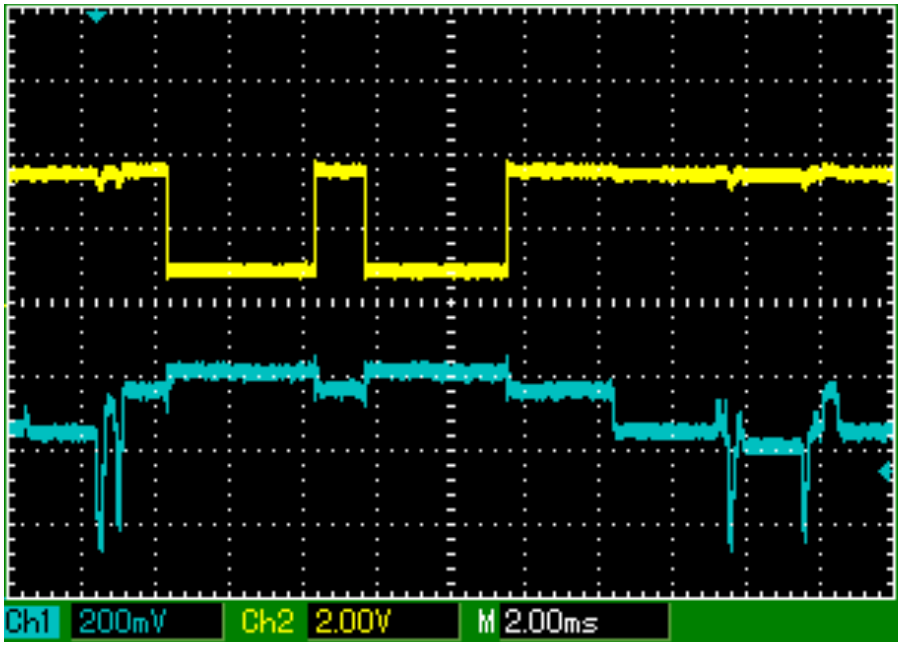


Figure 6.7: Server secured transmission and LED activity.

following protocols:

- SiGLoWPAN receipt operation time before the first LED activity;
- decryption phase during the first LED activity;
- CoAP interval between the two LED activities;

- encryption of the response during the second LED activity;
- SiGLoWPAN transmission operations after the second LED activity.

Also the carrier sense phase of the wireless circuitry can be recognized here just after the computation of the server response.

The only measurements that can be considered reliable from this first experiment are the CPU utilization times taken by various processing phases, since LED activity alters the energy consumption of the mote. For this reason, the experiment is repeated with the LED disabled in order to acquire reliable measurements of the electric potential difference of the sensing resistor during the CPU-intensive and inactivity phases. The resulting behaviors displayed by the oscilloscope for both CoAP client and server secured transmissions are shown in Figure 6.8 and Figure 6.9 respectively.

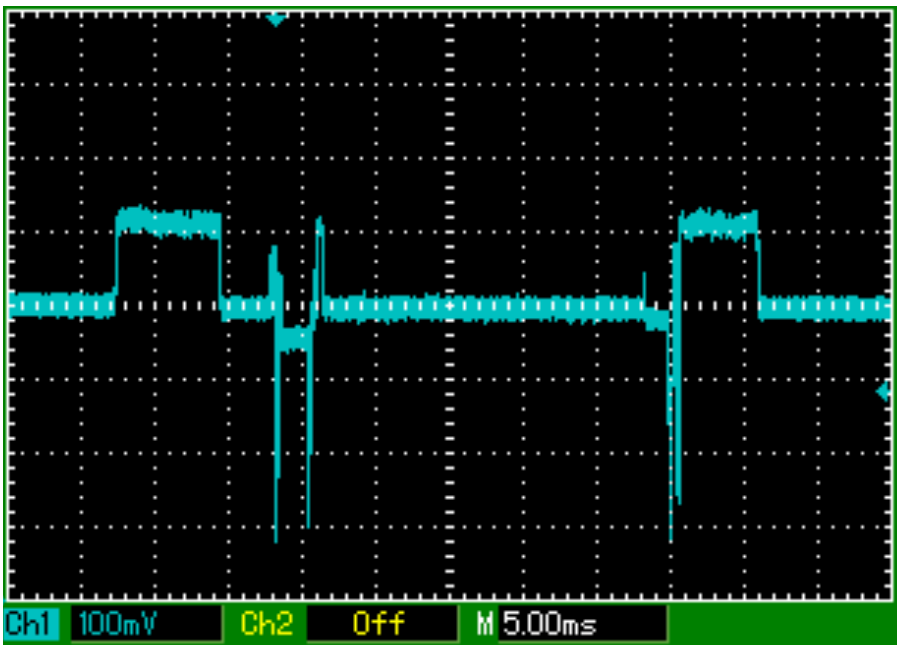


Figure 6.8: Client secured transmission.

At this point, in order to obtain the effective performance variation due to the DTLS implementation, the same experiment is repeated on two nodes programmed with the BlinkToCoAP application. This experiment provides other measurements of the resistor voltage during the CPU-intensive phases and the time required to elaborate the CoAP request and response without security features. The behaviour yielded by the oscilloscope for this experiment is shown in Figure 6.10 and Figure 6.11, where the reduced time fraction of the CPU-intensive tasks can be observed.

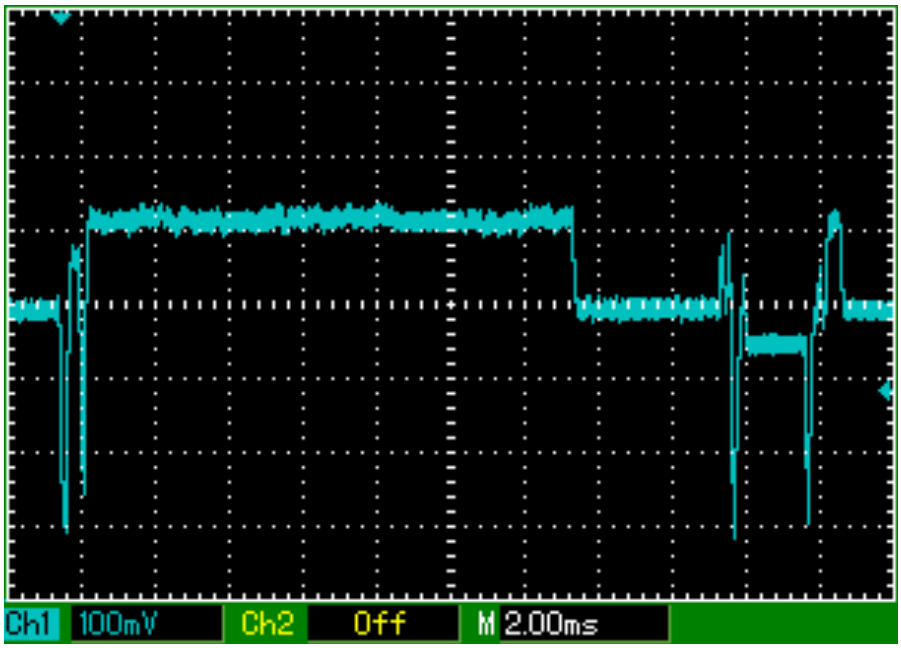


Figure 6.9: Server secured transmission.

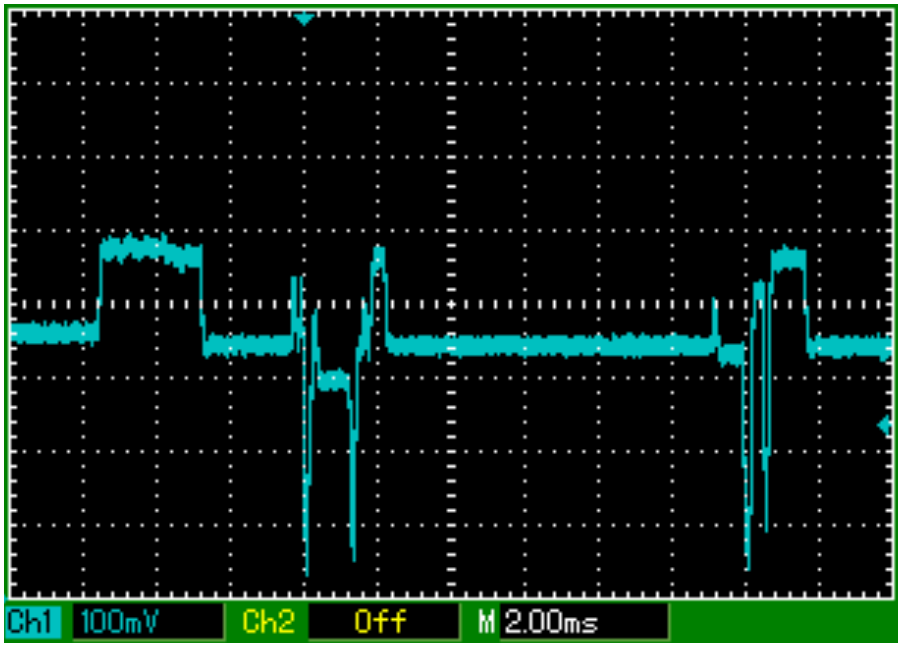


Figure 6.10: Client unsecured transmission.

CoAP Transmission Results

The oscilloscope experiments described above provide various information. The first are the measures of the sensing resistor voltage during the CPU-active and

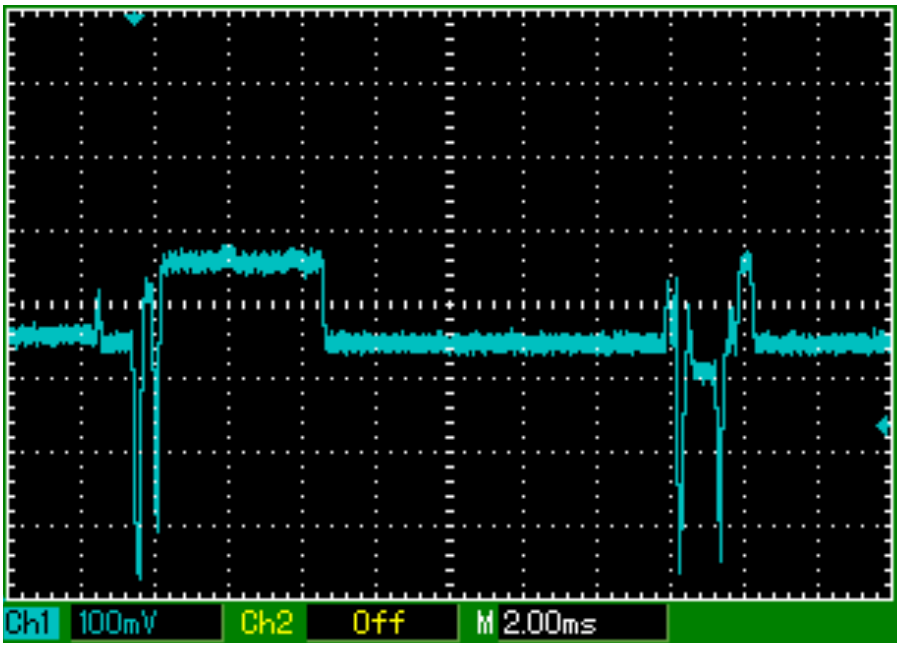


Figure 6.11: Server unsecured transmission.

CPU-inactive phases of the Zolertia platform, whose mean values are as follows:

$$V_{R,CPU-active} = 0.764\ [V] \tag{6.8}$$

$$V_{R,CPU-inactive} = 0.645\ [V] \tag{6.9}$$

The first value is used to get the estimation of energy consumption. The second kind of information gathered are the BlinkToSCoAP and BlinkToCoAP CPU times needed for the computation of CoAP requests, CoAP responses, handshake flights and radio transmissions. From the time values of various CPU-intensive processes the relative energy consumption can be estimated by means of equation 6.7 and related CPU ticks can be calculated using the following formula:

$$ticks_{proc} = \frac{T_{proc}}{T_{tick}} \tag{6.10}$$

$$= T_{proc} f_{CPU} \tag{6.11}$$

where, T_{proc} is the time duration of the considered process and f_{CPU} the CPU clock frequency of the Zolertia board. CPU ticks, average energy consumption and processing time values are presented together in Table 6.6 for CoAP secured transactions and in Table 6.7 for CoAP unsecured transactions.

	Processing time $[ms]$	Energy consumption $[\mu J]$	Ticks
Client request	7.08	698	113.28 K
Client response	5.22	515	83.52 K
Server request	5.82	574	93.12 K
Server response	7.34	722	117.12 K
Client transaction	12.30	1213	196.8 K
Server transaction	13.16	1296	210.24 K
Transaction	/	2509	407.04 K

Table 6.6: CoAP secured transaction performance parameters.

	Processing time $[ms]$	Energy consumption $[\mu J]$	Ticks
Client request	2.70	266	43.2 K
Client response	0.91	88	14.4 K
Client transaction	3.81	355	57.6 K
Server transaction	4.42	436	70.72 K
Transaction	/	791	128.32 K

Table 6.7: CoAP unsecured transaction performance parameters.

Table 6.7 does not provide information about the server unsecured request and response, because, in contrast to the CoAP secured transactions, there are no experiments with LED indications of the CPU time fractions for components other than DTLS.
The experiment done with the LED indicating the DTLS CPU usage time provides precise time values of DTLS security operations, comprehensive of message parsing, creation, encryption and decryption, which are reported in Table 6.8 together with their estimated energy consumption and CPU ticks.

The time requested for the security operations can be evaluated considering that in 3.84 ms the transceiver could transmit about 55 bytes, while in 4 ms about 57 bytes. From the estimated energy consumption values just presented can be calculated the ratio between the total energy spent, by both the client and the server, for a secured CoAP transaction and the total energy spent for an unsecured CoAP transaction. This value indicates how much more energy-hungry is the BlinkToSCoAP application in comparison to its unsecured version BlinkToCoAP

	Processing time $[ms]$	Energy consumption $[\mu J]$	Ticks
Creation, encryption	3.84	379	61.44 K
Parsing, decryption	4	395	64 K
All	7.84	774	125.44 K

Table 6.8: DTLS operation performance parameters.

when exchanging a secure CoAP transaction over an already established and active DTLS session:

$$\frac{E_{trans,sec}}{E_{trans,unsec}} = \frac{E_{trans,sec}(client) + E_{trans,sec}(server)}{E_{trans,unsec}(client) + E_{trans,unsec}(server)} \cong 3.17 \tag{6.12}$$

$E_{trans,sec}$ and $E_{trans,unsec}$ are the energies requested by client and server to complete a secured and unsecured CoAP transaction.

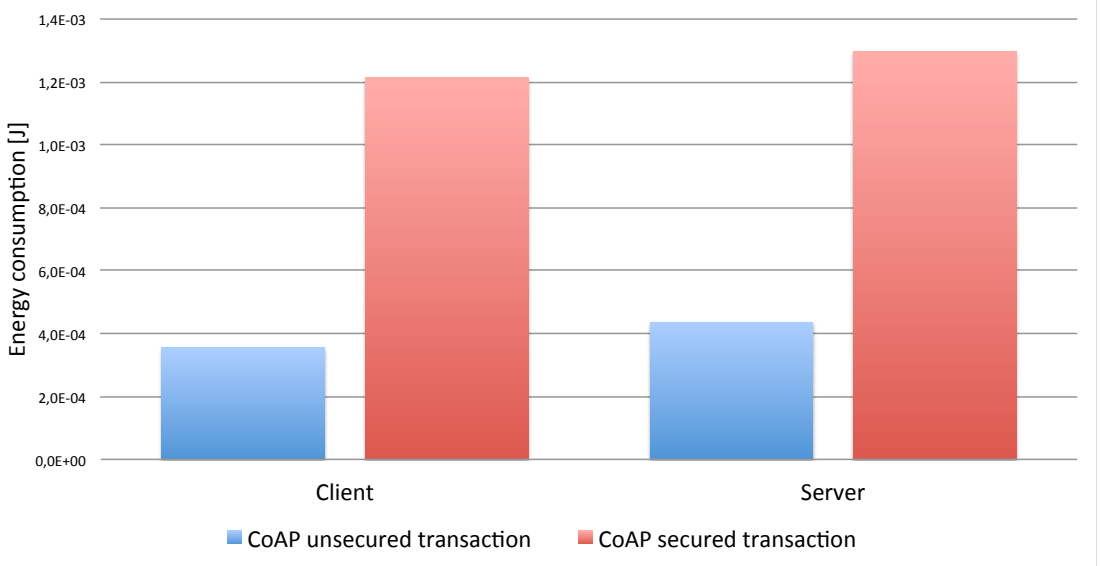


Figure 6.12: Comparison of energy consumption.

The security functionalities provided by the DTLS and SSLP components increment the energy requirements of CoAP transactions by a factor of about 3. In order to have a graphical comparison, the visual representation of client and server drained energy values for a complete secured and unsecured transaction can be found in Figure 6.12.

Considering also the energy drained from the DTLS library components, an estimation of energy inefficiency due to the SSLP component can be calculated as:

$$E_{trans,sec}(SSLP) = E_{trans,sec} - E_{trans,unsec} - 2E_{trans,sec}(DTLS) \quad (6.13)$$
$$\cong (1213 + 1296) - (355 + 436) - 2(379 + 395) \quad (6.14)$$
$$\cong 172\ [\mu J] \quad (6.15)$$

A graphical comparison of the drained energy (by both client and server devices) involving also the above value is depicted in Figure 6.13.

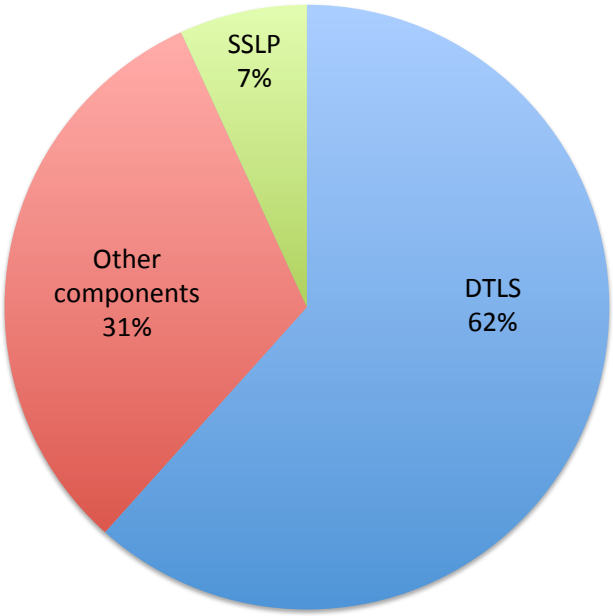


Figure 6.13: Energy consumption of DTLS and SSLP components for a secured CoAP transaction.

DTLS Handshake

Another source of performance degradation introduced by the DTLS protocol is its handshake phase. Every time two nodes interact for the first time or after a long gap, both of them must spend a certain amount of time and energy in order to establish or recover a DTLS secured channel. Since the lightweight DTLS implementation does not include mechanisms for recovering expired sessions, the experiment is applied only to the full handshake. Figure 6.14 and Figure 6.15 show the electric potential difference measured on the sensing resistor of CoAP client

and CoAP server respectively during a full DTLS handshake phase.

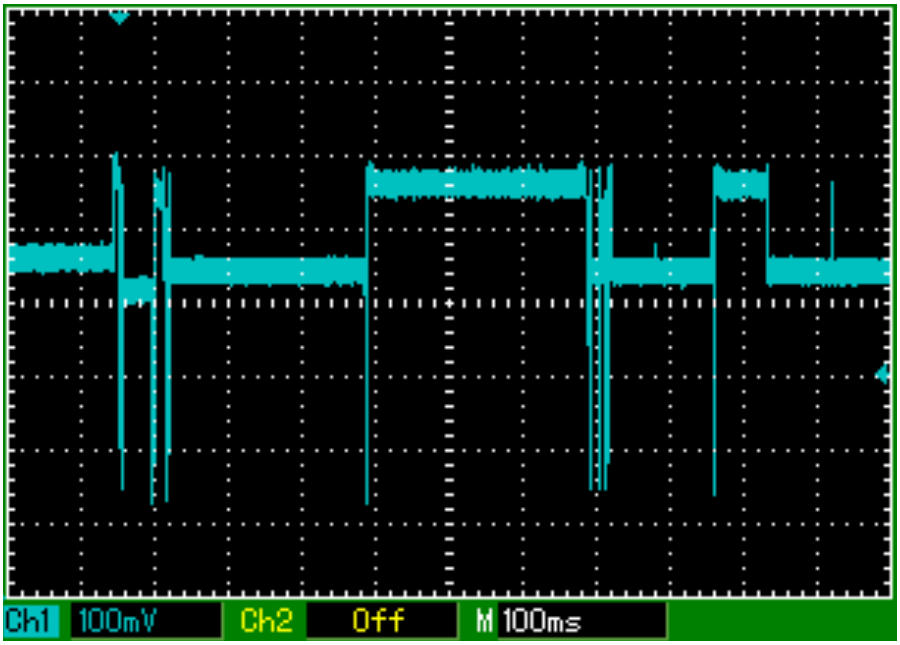


Figure 6.14: Client handshake phase.

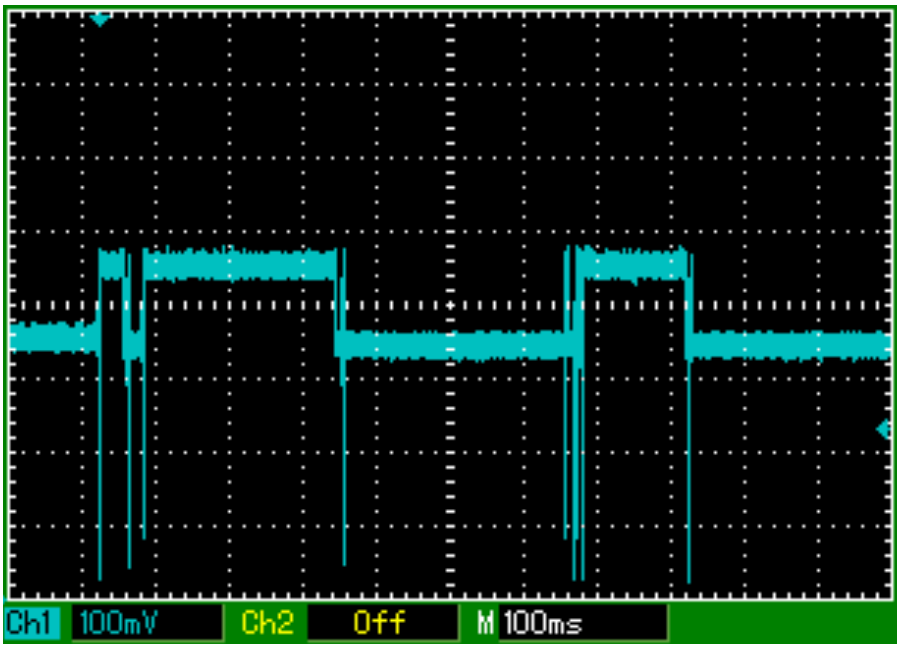


Figure 6.15: Server handshake phase.

The behaviour displayed by the oscilloscope evidences the handshake flight elaborations: the first step (due to the large spatial base of the oscilloscope) among

client measurements is the ClientHello message elaboration, followed by the troughs of the radio activity waveform and a small period of inactivity that corresponds to the ClientHelloVerify computation and transmission of the server as can be seen in Figure 6.15. Upon reception of this message, the client processes it and then retransmits the ClientHello with the received cookie included. Operations corresponding to the second step and its adjacent troughs are shown in Figure 6.14. After that, the server starts the the ServerHello elaboration and ServerHelloDone messages noticeable by the second long step on its measurements, while the client is inactive and waits for the fourth flight. The fifth flight elaboration and transmission, visible on the client behaviour as the third step and the consecutive peaks, involves the computation and transmission of the CKE, CCS and CF messages. Upon reception of the ChangeCipherSpec message, the server immediately switches to the secure transmission mode and responds with its own CCS message. Its receipt corresponds to the last peak after the long step on the client measurements and to the preceding peaks of the last step on the server behaviour. After the security mode switch, the server decrypts and elaborates the received ClientFinished message and starts the computation of its ServerFinished message, as indicated by the last step shown in Figure 6.15. Finally, the last CPU-intensive period of the client handshake indicates the decryption and verification of the hashed information carried by the received SF message.

DTLS Handshake Results

Referring to the flight numbers presented in Table 5.1, Table 6.9 and Table 6.10 show – together with the performance parameters – the average time values needed by client and server operations to handle message flights of a full handshake. These operations include all the elaborations done by the SiGLoWPAN library in order to handle the reception or transmission of the flights involved plus the parsing and construction of received flights and new flights scheduled by the DTLS library. The total time needed by a couple of nodes to perform a full handshake is a channel-dependent parameter, but the time durations required for frame transmissions are negligible compared to the duration of their elaborations, which are the main source of latency. The average delays caused by the handshake phase are reported in Table 6.11.

The time and energy required to establish a secure channel are a major source of performance degradation. In fact, the elaborations of the handshake phase consume an amount of energy that approximately corresponds to the energy requested by a device for exchanging 33 CoAP secured transactions, while within the time required

Flights involved	Processing time [ms]	Energy consumption [μJ]	Ticks
1	7.8	769	124.8 K
2, 3	14.2	1400	22.72 K
4, 5	302.3	29822	4833.6 K
6	3.2	316	51.2 K
7	72.4	7142	1158.4 K
All	400	39450	6398.4 K

Table 6.9: Client handshake flights and their processing parameters.

Flights involved	Processing time [ms]	Energy consumption [μJ]	Ticks
1,2	31.7	3127	507.2 K
3,4	259.9	25639	4158.4 K
5	2.9	286	46.4 K
6,7	142.8	14087	2284.8 K
All	437.3	43140	6996.8 K

Table 6.10: Server handshake flights and their processing parameters.

	Duration [ms]
Client	880
Server	800

Table 6.11: Handshake durations.

to establish the secure channel the device could transmit approximately 200 secured transactions.

Chapter 7

Conclusions

7.1 Concluding Remarks

The goal of this thesis was to present both the implementation as well as the evaluation of a CoAP application secured by DTLS for 6LoWPAN-based IoT networks. This work gives an overview of the underlying protocol implementations and new components merged to assemble the proposed BlinkToSCoAP application and outlines their relationships to ensure their correct collaboration. Experimental results show that despite the unoptimized state of protocols implementation and the high number of their functionalities, the amount of RAM available in the Zolertia Z1 platform is quite sufficient to contain all the necessary data. On the other hand, the security operations of DTLS implementation, consume considerable amounts of energy and introduce delay for every handshake carried out every time two nodes begin a new communication or need to recover an expired session. Once the security session has been established, the security operations increase the energy consumption of CoAP transmissions by a factor of about 3.2 and affect the responsiveness of the nodes, which have to work for additional 8 milliseconds per CoAP transaction in order to deal with obfuscated data and to transmit the additional DTLS header.
The context considered to assess the quality of the proposed application comprises communications between only two devices supplied through a USB cable. Therefore, the next logical step would be the evaluation of the application performance in more realistic conditions involving multiple nodes running on battery supply.

7.2 Future Work

The development of the BlinkToSCoAP application and its performance evaluation has raised some issues that suggest possible directions for future work as an extension of the works presented here.

Delegation of the handshake operations The operations done during the handshake phase cause a considerable amount of energy consumption and latency. A future work could analyze the possibility to delegate these operations to a trusted third entity with no energy constraint and with better performance, reducing the delay and allowing the devices to save precious energy.

Dynamic memory approach The MemoryManager component included in the SiGLoWPAN library provides functionalities that greatly simplify and optimize the memory management for constrained devices. Reviewing the CoAP and DTLS libraries, in order to change their static allocation approach to the dynamic one by the MemoryManager module, can greatly improve the application performance while also reducing source code complexity and incrementing code reutilization.

DTLS vertical orientation The SSLP solution, presented in Chapter 5, that resolves the issues of the DTLS horizontal design may not be the best one. A better alternative would be to rewrite the DTLS interface. This operation firstly will remove the DTLS structures from the signatures of the interface primitives, forcing the *dtls* module to keep its variables private. Secondly, it will replace the events used to signal obfuscated or plain data just after its elaboration with events that signal the transmission of the current message or the receipt of a new one. The rewriting process then would also involve the *dtls* component that has to include the mechanisms implemented in the SSLP module in order to handle the transmission and receipt of packets via the UDP interface provided by the SiGLoWPAN library.

TimedPool for handshake reliability The TimedPool component included in the CoAP library implements a timed queue intended to provide CoAP message reliability. This module could also be exploited by the DTLS library in order to guard against handshake message losses, while also incrementing the source code reutilization.

CoAP separated responses The current implementation of the CoAP protocol does not provide separate responses for confirmable messages. This means

that if the server needs more than a CoAP timeout duration to elaborate the response, the client will retransmit the request, thus wasting precious energy and network resources. Therefore, the implementation of CoAP separated responses could enhance the overall application performance. This feature also raises the need for a second matching mechanism, that has to be implemented as well, in order to have a distinct identifier for CoAP CON, NON, ACK messages and CoAP requests and responses.

DTLS header compression Experimental results show that the DTLS protocol adds 29 bytes of header to each CoAP message. In order to enhance both the transmission efficiency as well as the maximum payload size available without MAC layer fragmentation, the 6LoWPAN header compression for DTLS presented in [13] could be integrated into the SiGLoWPAN library.

Bibliography

[1] 2013. URL: http://www.eetutorials.com/article/28/1/TinyOS-installation-guide-on-Ubuntu.html (cit. on p. 25).

[2] C. Bormann. *Using CoAP with IPsec*. Tech. rep. CoRE Working Group, 2012 (cit. on p. 2).

[3] A. Castellani, G. Ministeri, M. Rotoloni, L. Vangelista, and M. Zorzi. "Interoperable and globally interconnected Smart Grid using IPv6 and 6LoWPAN". In: *Communications (ICC), 2012 IEEE International Conference on*. 2012, pp. 6473–6478 (cit. on pp. 30, 32).

[4] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). Updated by RFCs 5746, 5878, 6176. Internet Engineering Task Force, Aug. 2008. URL: http://www.ietf.org/rfc/rfc5246.txt (cit. on p. 13).

[5] R. T. Fielding. "REST: Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (cit. on p. 5).

[6] J. Granjal, E. Monteiro, and J. Sa Silva. "On the feasibility of secure application-layer communications on the Web of Things". In: *Local Computer Networks (LCN), 2012 IEEE 37th Conference on*. 2012, pp. 228–231 (cit. on p. 3).

[7] J. Granjal, E. Monteiro, and J. Silva. "On the Effectiveness of End-to-End Security for Internet-Integrated Sensing Applications". In: *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*. 2012, pp. 87–93 (cit. on p. 3).

[8] T. Instruments. *CC2420 Datasheet*. URL: http://www.silabs.com/Support%20Documents/TechnicalDocs/CP2102-9.pdf (cit. on p. 50).

[9] A. Kevin. "That 'Internet of Things' Thing, in the real world things matter more than ideas". In: *RFID Journal* (2009) (cit. on p. 1).

[10] T. Kothmayr, C. Schmitt, W. Hu, M. Brunig, and G. Carle. "A DTLS based end-to-end security architecture for the Internet of Things with two-way authentication". In: *Local Computer Networks Workshops (LCN Workshops), 2012 IEEE 37th Conference on*. 2012, pp. 956–963 (cit. on p. 3).

[11] S. Labs. *CP2102 Datasheet*. URL: `http://www.silabs.com/Support%20Documents/TechnicalDocs/CP2102-9.pdf` (cit. on p. 50).

[12] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944 (Proposed Standard). Updated by RFCs 6775, 6282. Internet Engineering Task Force, 2007. URL: `https://datatracker.ietf.org/doc/rfc4944/` (cit. on p. 48).

[13] S. Raza, D. Trabalza, and T. Voigt. "6LoWPAN Compressed DTLS for CoAP". In: *Distributed Computing in Sensor Systems (DCOSS), 2012 IEEE 8th International Conference on*. 2012, pp. 287–289 (cit. on pp. 4, 65).

[14] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt. "Lithe: Lightweight Secure CoAP for the Internet of Things". In: *Sensors Journal, IEEE* 13.10 (2013), pp. 3711–3720 (cit. on p. 4).

[15] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security*. RFC 4347 (Proposed Standard). Updated by RFC 5746. Internet Engineering Task Force, Apr. 2006. URL: `http://www.ietf.org/rfc/rfc4347.txt` (cit. on pp. 2, 13, 18).

[16] Z. Shelby, Sensinode, K. Hartke, and C. Bormann. *Constrained Application Protocol (CoAP) draft-ietf-core-coap-18*. Tech. rep. CoRE Working Group, 2013 (cit. on pp. 2, 5).

[17] Zolertia. *Z1 Datasheet*. 2013. URL: `http://zolertia.sourceforge.net/wiki/images/e/e8/Z1_RevC_Datasheet.pdf` (cit. on p. 50).