

**PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
GRADUATE PROGRAM IN COMPUTER SCIENCE**

**SECURITY SERVICES
PROVISION FOR SOA-BASED
IOT MIDDLEWARE SYSTEMS**

RAMÃO TIAGO TIBURSKI

Dissertation presented as partial requirement
for obtaining the degree of Master in
Computer Science at Pontifical Catholic
University of Rio Grande do Sul.

Advisor: Prof. Fabiano Passuelo Hessel

**Porto Alegre
2016**

Dados Internacionais de Catalogação na Publicação (CIP)

T554s Tiburski, Ramão Tiago

Security services provision for SOA-Based IoT Middleware Systems / Ramão Tiago Tiburski. – 2016.

102 f.

Dissertação (Mestrado) – Faculdade de Informática, PUCRS.
Orientadores: Prof. Dr. Fabiano Hessel.

1. Segurança da Informação. 2. Sistemas distribuídos.
3. Serviços Web. 4. Sistemas de Informação. 5. Arquitetura da
Informação. 6. Informática. I. Hessel, Fabiano. II. Título.

CDD 23 ed. 004.2


Ramon Ely CRB 10/2165
Setor de Tratamento da Informação da BC-PUCRS



Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

STATEMENT OF PRESENTATION OF THE DISSERTATION

Dissertation entitled "Security Services Provision for SOA-based IoT Middleware Systems" presented by Ramão Tiago Tiburski as part of the requirements to achieve the degree of Master in Computer Science approved on March 16, 2016 by the Examination Committee:



Prof. Dr. Fabiano Passuelo Hessel – PPGCC/PUCRS
Advisor

Prof. Dr. Leonardo Albernaz Amaral PPGEE/PUCRS

Prof. Dr. Avelino Francisco Zorzo PPGCC/PUCRS

Prof. Dr. Franck Rousseau Grenoble INP-Ensimag

Ratified on 16/06/2016, according to Minute No. 012 by the Coordinating Committee.



Prof. Dr. Luiz Gustavo Leão Fernandes
Graduate Program Coordinator

PUCRS

Campus Central

Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br

PROVISÃO DE SERVIÇOS DE SEGURANÇA PARA SISTEMAS DE MIDDLEWARE DA IOT BASEADOS EM SOA

RESUMO

A evolução da IoT (do inglês, Internet of Things) requer uma infra-estrutura de sistemas que seja capaz de fornecer serviços tanto para abstração de dispositivos e gerenciamento de dados, quanto para suporte ao desenvolvimento de aplicações. Middleware para IoT tem sido reconhecido como o sistema capaz de prover esta infra-estrutura necessária de serviços e vem se tornando cada vez mais importante para a Internet das Coisas ao longo dos últimos anos. A arquitetura de um sistema de middleware para IoT geralmente está baseada no padrão SOA (do inglês, Service-Oriented Architecture) e tem o requisito de segurança como um dos seus principais desafios. A grande quantidade de dados que trafega nesse tipo de sistema exige serviços de segurança capazes de garantir a proteção dos dados em toda a extensão do sistema. Além disso, algumas aplicações para IoT, principalmente aquelas voltadas para ambientes de saúde, fizeram surgir novos requisitos em termos de comunicação segura e tempo de resposta aceitável para serviços críticos. Embora diversas tecnologias de middleware para IoT têm sido utilizadas para lidar com os requisitos mais relevantes exigidos pelas diferentes aplicações existentes para IoT, segurança ainda é um tema especial que não está maduro o suficiente neste tipo de tecnologia. Os desafios de segurança relacionados à cenários de saúde estão concentrados, principalmente, nas questões relacionadas com a camada de comunicação, especialmente nos casos em que dados de pacientes são transmitidos em redes abertas, onde são mais vulneráveis a ataques. Neste sentido, existe a necessidade de garantir confidencialidade e integridade de dados nas camadas do middleware para permitir um entendimento mais confiável a respeito do estado de vida de um paciente. Este trabalho propõe a definição de quatro serviços de segurança voltados para proteção de dados a fim de minimizar os problemas de segurança encontrados em sistemas de middleware para IoT baseados em SOA. Apenas um dos serviços de segurança propostos foi implementado neste trabalho (o CCP - Communication Channel Protection), o qual é um serviço composto pela implementação de dois protocolos de segurança: TLS e DTLS. Ambas abordagens estão baseadas em protocolos de segurança já conhecidos e capazes de garantir confidencialidade, integridade e autenticidade. O serviço implementado visa proteger a transmissão de dados em um sistema de middleware para IoT (COMPaaS - Cooperative Middleware Platform as a Service), e foi

validado através de um cenário de aplicação específico para a área da saúde. O principal objetivo da validação foi verificar se as implementações dos serviços de segurança estavam comprometendo, em termos de tempo de resposta, o desempenho das camadas de comunicação dos sistemas do middleware COMPaaS, o qual é o requisito fundamental do cenário de saúde. Testes revelaram resultados satisfatórios visto que as abordagens implementadas respeitaram o requisito de tempo de resposta da aplicação e protegeram os dados transmitidos.

Palavras Chave: Internet das Coisas, Middleware, SOA, Segurança, Confidencialidade, Integridade, Autenticidade, Confiabilidade, TLS, DTLS, COMPaaS, Serviço Médico de Emergência, E-health.

SECURITY SERVICES PROVISION FOR SOA-BASED IOT MIDDLEWARE SYSTEMS

ABSTRACT

The evolution of the Internet of Things (IoT) requires an infrastructure of systems that can provide services for devices abstraction and data management, and also support the development of applications. IoT middleware has been recognized as the system that can provide this necessary infrastructure of services and has become increasingly important for IoT over the last years. The architecture of an IoT middleware is usually based on SOA (Service-Oriented Architecture) standard and has security requirement as one of its main challenges. The large amount of data that flows in this kind of system demands security services able to ensure data protection in the entire system. In addition, some IoT applications, mainly those from e-health environments, have brought new requirements in terms of secure communication and acceptable response time for critical services. Although IoT middleware technologies have been used to cope with the most relevant requirements demanded by different IoT applications, security is a special topic that is not mature enough in this kind of technology. The security challenges regarding e-health scenarios are concentrated mainly on issues surrounding the communication layer, specially those cases in which patient data are transmitted in open networks where they are more vulnerable to attacks. In this sense, there is a need for ensure data confidentiality and integrity in middleware system layers to enable a reliable understanding of a patient current life state. This work proposes the definition of four security services focused on data protection in order to minimize security problems found in SOA-based IoT middleware systems. We implemented only one of these services (CCP - Communication Channel Protection) which is composed of two security approaches: TLS and DTLS. Both approaches are known security protocols able to provide confidentiality, integrity, and authenticity. The implemented service was focused on protecting data transmission in an IoT middleware system (COMPaaS - Cooperative Middleware Platform as a Service) and was validated through a specific e-health scenario. The main goal was to verify if our security implementations compromise, in terms of response time, the communication performance of the middleware system, which is the key requirement of the e-health scenario. Tests revealed a satisfactory result since the implemented approaches respected the response time requirement of the application and protected the transmitted data.

Keywords: Internet of Things, Middleware, Security, Confidentiality, Integrity, Authenticity, Reliability, TLS, DTLS, COMPaaS, Emergence Medical Service, E-health.

LIST OF FIGURES

Figure 1.1 – Smart city and healthcare scenario.	23
Figure 1.2 – Work scope.	25
Figure 2.1 – IoT architecture.	27
Figure 2.2 – SOA-based architecture for IoT middleware.	28
Figure 2.3 – COMPaaS architecture overview.	31
Figure 3.1 – Security taxonomy for SOA-based IoT middleware.	33
Figure 3.2 – DTLS layer.	38
Figure 3.3 – Cipher suite composition.	39
Figure 4.1 – Composition of the security services.	48
Figure 4.2 – Security services on SOA-based IoT middleware.	49
Figure 4.3 – CCP service architecture.	50
Figure 4.4 – Schematic overview of the TLS approach.	51
Figure 4.5 – Schematic overview of the DTLS approach.	52
Figure 4.6 – Creating a SecureRandom object.	55
Figure 4.7 – Creating KeyStore objects.	55
Figure 4.8 – Reading client key pair.	56
Figure 4.9 – Creating the TrustManagerFactory.	56
Figure 4.10 – Creating the KeyManagerFactory.	56
Figure 4.11 – Creating an SSLContext.	56
Figure 4.12 – Connecting to the TLSServer.	57
Figure 4.13 – Creating an SSLServerSocket channel.	57
Figure 4.14 – Receiving a TLSClient connection.	57
Figure 4.15 – Using alias to load certificate chain and private key.	58
Figure 4.16 – Extracting public key.	59
Figure 4.17 – Configuring a DTLSConnector instance.	59
Figure 4.18 – Calling the encryption method.	60
Figure 4.19 – Calling the decryption method.	60
Figure 5.1 – COMPaaS middleware applied in an EMS scenario.	63
Figure 5.2 – Environment configuration for tests.	65
Figure 5.3 – DataMessage content.	66
Figure 5.4 – User interface used for medical devices simulation.	67
Figure 5.5 – TLS handshake to get authentication.	68

Figure 5.6 – DTLS handshake to get authentication. 69

Figure 5.7 – TLS message encrypted during transmission. 70

Figure 5.8 – DTLS message encrypted during transmission. 70

Figure 5.9 – Scenario goals on secure COMPaaS. 71

Figure 5.10 – Overhead added by the security approaches. 72

Figure 5.11 – Response time for applications with and without security. 73

Figure 5.12 – Overhead added when compared to non-secure approaches (%). 74

Figure 5.13 – Example of a data message and its respective ACK response. 75

Figure APPENDIX B.1 – DTLS Full Handshake (adapted from [22]). 91

Figure APPENDIX B.2 – DTLS Abbreviated Handshake (adapted from [22]). 95

Figure APPENDIX C.1 – TLS Full Handshake. 97

LIST OF TABLES

Table 3.1 – Relationship between attacks and requirements.	36
Table 3.2 – Security in IoT middleware systems.	42
Table 3.3 – Security in IoT communication protocols.	45
Table 5.1 – Used configuration for computers.	65
Table 5.2 – Summary of the functionality tests on COMPaaS.	68
Table 5.3 – Performance Tests (ms).	71

LIST OF ACRONYMS

6LowPAN – IPv6 over Low power Wireless Personal Area Networks

AAC – Authorization and Access Control

ADA – Applications and Devices Authentication

AEAD – Authenticated Encryption with Associated Data

AES – Advanced Encryption Standard

API – Application Programming Interface

AUT – Authentication

CA – Certificate Authority

CCM – Counter with Cipher Block Chaining Message Authentication Code

CCP – Communication Channel Protection

CoAP – Constrained Application Protocol

COMPaaS – Cooperative Middleware Platform as a Service

CON – Confidentiality

CoRE – Constrained RESTful Environments

CSR – Certificate Signing Request

DCI – Data Confidentiality and Integrity

DH – Diffie-Hellman

DoS – Denial-of-Service

DSA – Digital Signature Algorithm

DTLS – Datagram Transport Layer Security

ECC – Elliptic Curve Cryptography

ECDHE – Elliptic-curve Diffie-Hellman Ephemeral

ECDSA – Elliptic-curve Digital Signature Algorithm

EMS – Emergency Medical Service

GCM – Galois/Counter Mode

GSE – Grupo de Sistema Embarcados

GSN – Global Sensor Networks

HMAC – Hash-based Message Authentication Code

HTTP – Hypertext Transfer Protocol

HTTPS – Hypertext Transfer Protocol Secure

ID – Identifier

IETF – Internet Engineering Task Force

INT – Integrity

IoT – Internet of Things

IP – Internet Protocol

JCA – Java Cryptography Architecture

JKS – Java KeyStore

JSSE – Java Secure Socket Extension

KB – KiloByte

LAN – Local Area Network

LWM2M – LightWeight Machine-to-Machine

M2M – Machine-to-Machine

MQTT – Message Queuing Telemetry Transport

NoSecTCP – No Security TCP

NoSecUDP – No Security UDP

PKI – Public Key Infrastructure

PRF – Pseudo-Random Function

PSK – Pre-Shared Key

PUCRS – Pontifícia Universidade Católica do Rio Grande do Sul

REST – Representational State Transfer

RFC – Request for Comments

RFID – Radio-Frequency IDentification

RSA – Rivest, Shamir and Adleman

SASL – Simple Authentication and Security Layer

Sc – Scandium

SCP – Security Communication Protocol

SHA – Secure Hash Algorithm

SMARTIE – Secure and sMARter ciTIEs data management

SOA – Service-Oriented Architecture

SOAP – Simple Object Access Protocol

SPTP – Secure, Private and Trustworthy Protocol

TCP – Transmission Control Protocol

TLS – Transport Layer Security

UDP – User Datagram Protocol

URI – Uniform Resource Identifier

WHO – World Health Organization

WS – Web Services

WSN – Wireless Sensor Network

XML – eXtensible Markup Language

XMPP – eXtensible Messaging and Presence Protocol

CONTENTS

1	INTRODUCTION	21
1.1	MOTIVATION	22
1.2	CONTRIBUTION	24
1.3	WORK SCOPE	24
1.4	OUTLINE	25
2	IOT MIDDLEWARE SYSTEMS	27
2.1	IOT SYSTEMS ARCHITECTURE	27
2.2	SOA-BASED MIDDLEWARE	28
2.3	COMPAAS MIDDLEWARE	30
2.4	FINAL CONSIDERATIONS	32
3	SECURITY SURVEY	33
3.1	THREATS AND SECURITY REQUIREMENTS	33
3.2	SECURITY BACKGROUND	36
3.2.1	TRANSPORT LAYER SECURITY	36
3.2.2	DATAGRAM TRANSPORT LAYER SECURITY	37
3.2.3	PUBLIC-KEY CRYPTOGRAPHY	38
3.2.4	CERTIFICATES AND CERTIFICATE AUTHORITY	39
3.2.5	CIPHER SUITES	39
3.3	SECURITY IN IOT MIDDLEWARE SYSTEMS	40
3.4	EMERGING SECURITY STANDARDS AND PROTOCOLS	43
3.5	SECURITY IN IOT COMMUNICATION PROTOCOLS	45
3.6	FINAL CONSIDERATIONS	46
4	PROPOSED APPROACH	47
4.1	SECURITY FOR SOA-BASED IOT MIDDLEWARE	47
4.1.1	SECURITY SERVICES	47
4.1.2	TLS ON COMPAAS	50
4.1.3	DTLS ON COMPAAS	51
4.2	IMPLEMENTATION	53
4.2.1	TLS APPROACH	53
4.2.2	DTLS APPROACH	58

4.3	DISCUSSION	60
5	EVALUATION	63
5.1	USE CASE DESCRIPTION	63
5.1.1	EMERGENCY MEDICAL SERVICE (EMS)	63
5.1.2	ENVIRONMENT SETUP	64
5.2	FUNCTIONALITY TESTS	68
5.3	PERFORMANCE TESTS	70
5.3.1	DISCUSSION	74
6	FINAL CONSIDERATIONS	77
6.1	ATTACK ANALYSIS	77
6.2	SECURITY SERVICES	78
6.3	NON-INTRUSIVE SECURITY PROTOCOLS IN IOT ENVIRONMENTS	79
6.4	FUTURE IMPLEMENTATIONS	80
6.5	CONCLUSION	80
	REFERENCES	83
	APPENDIX A – PUBLICATIONS	89
	APPENDIX B – DTLS HANDSHAKE PROTOCOL	91
	APPENDIX C – TLS HANDSHAKE PROTOCOL	97
	APPENDIX D – KEY MANAGEMENT	101

1. INTRODUCTION

A new worldwide trend in Computer Science is becoming real. The computer systems are ceasing to be centralized systems and oriented to personal computers to become a portable computing based on mobile devices with a diversity of services and communication channels. This new trend is called Internet of Things (IoT), a computing paradigm that aims to interconnect our everyday life objects (i.e., computing devices or things) using the Internet as the communication medium. Beyond Internet-based communication, the IoT also aims to provide information processing capabilities to enable things to sense, integrate, and present data, reacting to all aspects of the physical world [6].

An IoT ecosystem is based on a layered architecture style and uses this view to abstract and automate the integration of objects, and also to provide smart services solutions to applications [21]. In IoT, high-level system layers, as the application layer, are composed of IoT applications and middleware system that is an entity that simplifies the development of applications by supporting services to cope with the interoperability requirement among heterogeneous devices (i.e., objects or things) [17].

Although there are IoT middleware systems designed to be applied in specific areas of application, most of the existing systems are being designed to cover different domains, like industry, environment, and society, and lack of a generic and well-defined system architecture able to allow the interoperability of these different areas. Consequently, the SOA (Service-Oriented Architecture) standard has been used as a viable choice to cope with this gap, since it helps to ensure high levels of systems interoperability, providing system services that are based on devices functionality and can be used by applications [6].

An important application field for SOA-based IoT middleware is the e-health environment [6]. It benefits from the use of a set of interconnected devices to create an IoT network devoted to healthcare assessment, including monitoring of patients and automatically detecting critical situations where medical interventions are required.

Many applications for e-health have been created in scenarios like home care and Emergency Medical Services (EMS). Home care is usually related to monitoring systems for the elderly or post trauma patients. An application example is the automated movement monitoring system that allows the identification of falls and notification of medical personnel without any user intervention. The combination of voice and video allows for verification and a more appropriate response in case of alarms [28].

In a case of disaster relief operations or EMS, an ambulance-to-hospital based e-health system is an appropriated example of how IoT technology can help save lives. In this case, by providing real-time patient information to the hospital via wireless communications, this e-health system can enable remote diagnoses and primary care, reducing rescue response time.

An¹ SOA-based IoT middleware system can be used to abstract the devices or medical equipment integration in a house or an ambulance, and also to allow the proper interaction with hospital systems. In EMS, for example, the middleware can locate the ambulance and provide the shortest path routing, and thus, patients can be carried to the hospital as quickly as possible. In this sense, it is mandatory to have an effective IoT middleware system able to ensure the response time between send and interpret data fast enough to guarantee that all decisions of a physician are based on the current health condition of a patient.

However, the described scenarios have some security issues since most of the transmissions from a house or an ambulance to a hospital are made through open networks using no security protocols or mechanisms. Thus, the data transmitted in these networks can be targets of attacks that seek to spy or change these data. In this sense, there is a need to ensure data confidentiality and integrity as a means to enable a reliable understanding of the patient's current life state.

Security is considered one of the most important requirements for SOA-based IoT middleware systems [6] since they are very susceptible to attacks that may occur in the systems entities (i.e., applications, middleware and devices), data, or communication channels. Even though there are SOA-based IoT middleware systems that have some kind of security in their architectures, the security approaches provided by these systems are specific to some application requirements, and, in most cases, do not present implementation details. The definition of security services that can protect data in all extension of the systems of the middleware is extremely important and required. In addition, IoT applications like smart cities and healthcare have vulnerable scenarios, which reinforces how the protection of these systems is crucial.

1.1 MOTIVATION

SOA-based IoT middleware systems can be applied to various scenarios. Recently, two scenarios have been emphasized: healthcare and smart cities. Both are being a constant focus in IoT. A real example is illustrated in Figure 1.1: a smart city with monitoring cameras integrated with a healthcare system. The use of cameras is necessary to verify the current status of an injured person on the street and to request the presence of an ambulance. In this case, the cameras transmit video in real-time to the hospital system helping in the interpretation of the real-life condition of the patient and also in the first care assistance. This scenario illustrates a real case where the use of an SOA-based IoT middleware is recommended. The problem is that the transmitted data can constantly be target of threats and if the middleware system does not have all its architecture protected, it may face some security attacks (e.g., man-in-the-middle, eavesdropping, etc.).

The endless search for security, despite being old, remains a topic of extreme importance. According to [6] and [12], security and privacy support is crucial for the functions of an SOA-based IoT middleware solution. Currently, several systems try to provide some security in their

¹We read an S-O-A.

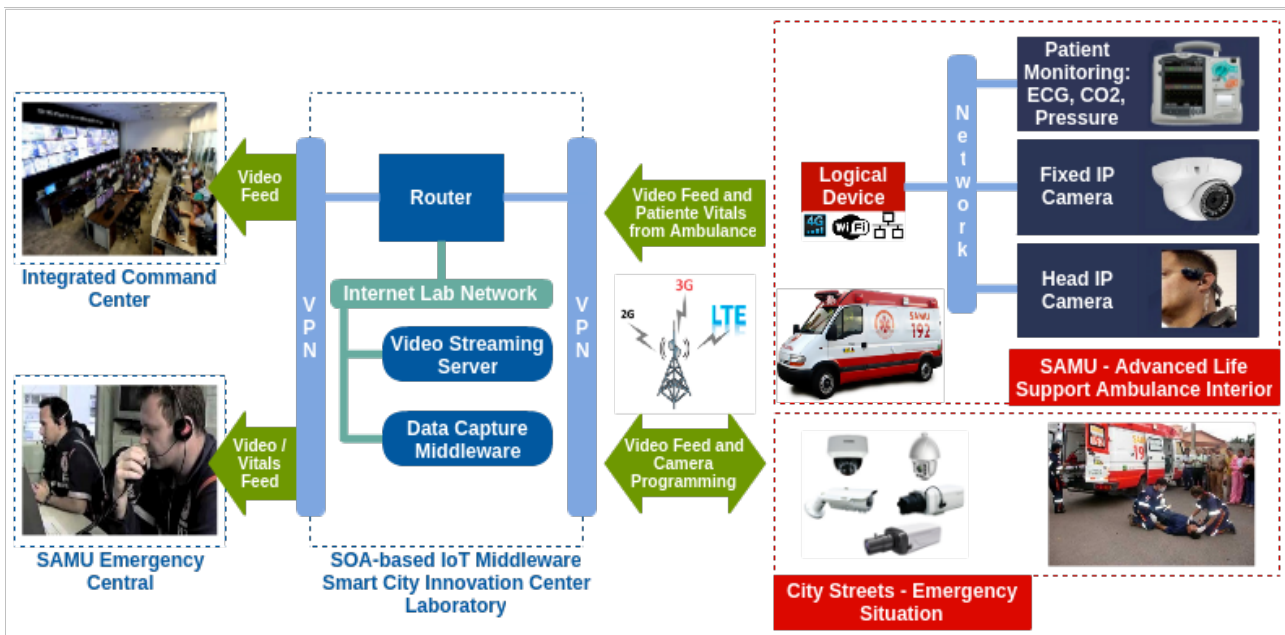


Figure 1.1 – Smart city and healthcare scenario.

architectures. However, they do not supply it in a proper way. In this sense, there is a need for the definition of a standard security architecture for SOA-based IoT middleware systems [55].

In IoT applications, like smart cities and healthcare, the concern is about to provide security for the entire systems architecture rather than just for a single piece of software or a single IoT layer. Moreover, the fact that each middleware solution is usually designed to implement only the necessary requirements to solve the application problems may not ensure it is able to be applied to other application domain as if it were a generic or a standard security architecture. Indeed, there is a lack of details that can hinder the standardization of a specific security architecture for SOA-based IoT middleware systems.

In addition, there are still significant challenges to be overcome regarding the potential protocols that may be used in the development of a security architecture. DTLS is a good example of this challenge. In order to DTLS become a valuable technique for the security provision in resource-constrained environments, it needs to be simplified, stripped off all unnecessary features to achieve a useful trade-off between security and a lightweight implementation suitable for the constraints of the environment [50]. In this sense, [21] and [46] have mentioned that the most significant challenge related to security in IoT is the necessity of having non-intrusive security solutions, which means to provide security in a way that it does not change or affect the system performance to the point of harming any interested part. Provide solutions such as key management, authentication, confidentiality, and integrity is considered a significant challenge, mainly when applied to resource-constrained environments like the IoT. In this way, to find an efficient strategy to deal with the massive data generated by IoT systems and to provide secure protocols to efficiently handle and organize all this information is a tremendous challenge.

The motivation of this work is related to the need of security services that can protect all the data processed by an SOA-based IoT middleware system. We also join this lack with existing security problems from key scenarios of e-health that deal with important requirements such as acceptable response time for critical services. Therefore, protecting data of an IoT middleware in a non-intrusive way is our primary motivation. Moreover, the need for security is also a challenge for the COMPaaS [4], an SOA-based IoT middleware of our research group GSE/PUCRS. In this sense, protect COMPaaS with our security services is another important motivation.

1.2 CONTRIBUTION

The main contributions of this work can be summarized in:

- The definition of four security services that can be used for SOA-based IoT middleware systems to mitigate security issues related to the Internet of Things. These services should be composed of security approaches based on the implementation of existing security mechanisms and protocols.
- The implementation of the CCP (Communication Channel Protection) security service to protect the data transmission among the middleware layers. We implemented TLS and DTLS protocols with the intention to provide confidentiality and integrity of data transmitted in non-protected environments.
- A security survey encompassing background, threats, requirements, and the current state-of-the-art research regarding security in IoT middleware systems and also standards and protocols that can be used to mitigate the security issues in the Internet of Things.
- The deployment of the security service (CCP) on COMPaaS middleware.

1.3 WORK SCOPE

The primary goal of this work is the protection of data in application environments of SOA-based IoT middleware systems. We present four security services that could be recommended to be used by these systems in order to get an adequate protection for all system data. As the first step for the definition of the security services, this work presents the implementation of the Communication Channel Protection (CCP) service in the COMPaaS middleware and illustrates its importance in a healthcare environment based on the Emergency Medical Service (EMS) scenario. Moreover, the EMS scenario adds the acceptable response time requirement as a new challenge to the CCP service, since it must provide security in a non-intrusive way.

As can be seen in Figure 1.2, the scope of this work refers to: (1) present security services required for an SOA-based IoT middleware, and (2) the implementation of the CCP service using protocols that can ensure confidentiality and integrity in a relevant e-health scenario.

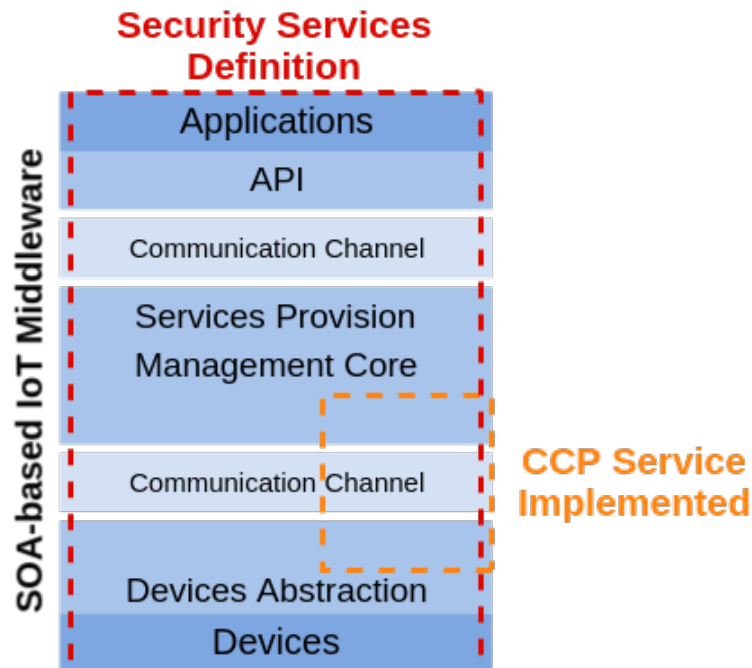


Figure 1.2 – Work scope.

1.4 OUTLINE

We start introducing IoT middleware systems in Chapter 2. Chapter 3 provides a security survey, which involves the definition of threats and security requirements, the description of some important protocols and concepts that were used in this work, and the presentation of the state-of-the-art regarding security in IoT middleware systems, standards, and protocols. Chapter 4 defines the security services and specifies how the CCP service will take place on COMPaaS middleware. In addition, it presents the implementation of the CCP service (TLS and DTLS approaches) on COMPaaS. Chapter 5 analyzes the results according to functionality and performance requirements when applied in an e-health scenario. Finally, Chapter 6 presents the final considerations of this work.

2. IOT MIDDLEWARE SYSTEMS

This chapter presents the main concepts related to the application area of this work. Section 2.1 presents an architecture overview of the IoT systems. Section 2.2 presents SOA as an important standard architecture for IoT middleware. Section 2.3 presents our SOA-based IoT middleware implementation named COMPaaS. We conclude this chapter in Section 2.4.

2.1 IOT SYSTEMS ARCHITECTURE

The Internet of Things brings forth a new phase of the Internet evolution that we can characterize as “the Internet meets the physical world”. The basic vision is that objects present in everyday life can be equipped with sensors which can track useful information. Attached to the Internet, the information of objects can flow through the same protocol that connects our computers to the Internet. These uniquely identified Internet-connected objects can sense whatever they are directed to, and communicate it, enabling collection and analysis of data in an unprecedented flow and volume [6].

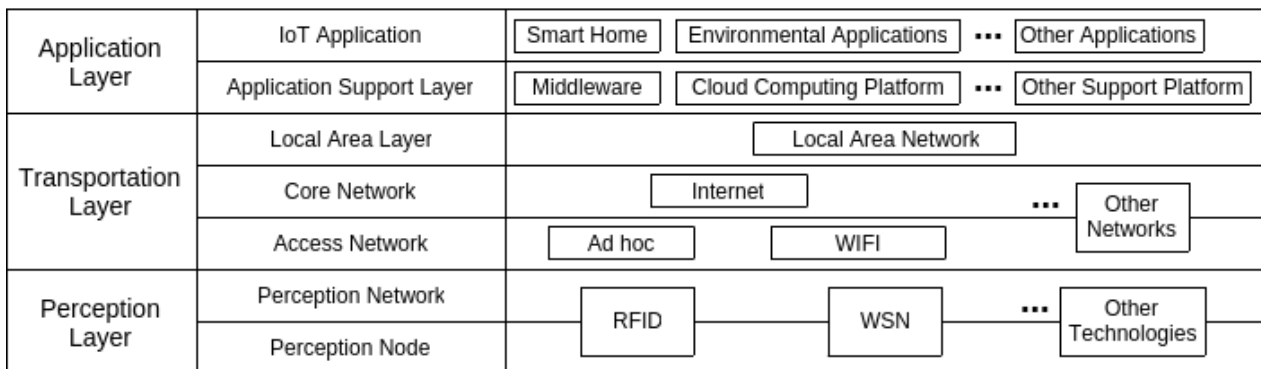


Figure 2.1 – IoT architecture.

IoT systems architecture can be divided into three layers [21]: application, transportation, and perception layer as shown in Figure 2.1.

- **Application layer:** The application layer can be divided into two parts: the IoT application and the application support layer. IoT application refers to the domains that IoT applications can be developed. On the other hand, the application support layer supports all sorts of services and realizes intelligent computation and logical resources allocation. Application support layer can be organized in different ways according to different services. Usually, it includes middleware, M2M², cloud computing platform and service support platform.

²Machine-to-Machine refers to direct and autonomous communication between devices using any communication channel [6].

- **Transportation layer:** The transportation layer provides network ubiquitous access for the elements of the perception layer [62]. This layer is a combination of a variety of heterogeneous networks and can be divided into three layers: access network, core network and local area. Access network provides ubiquitous accessing of the network. Core network is mainly responsible for data transmission. And local area network facilitates the use of the resources of the networks.
- **Perception layer:** This layer is responsible for the perception and control of devices and also for the collection of information. This layer can be divided into two parts: (1) the perception node is used for data acquisition and data control, and (2) the perception network sends collected data to the gateway or sends control instructions to the controller of the devices. Perception layer technologies include RFID, WSNs, etc.

2.2 SOA-BASED MIDDLEWARE

The Internet of Things assisted by wireless communication technologies has connected a lot of physical devices or things toward a variety of heterogeneous applications. Enterprises are still in the exploration phase for implementing vertical applications for IoT and machine-to-machine (M2M) technologies, but the increasing demand for bring-your-own-device and heterogeneous devices has accelerated the growth of IoT middleware systems [57].

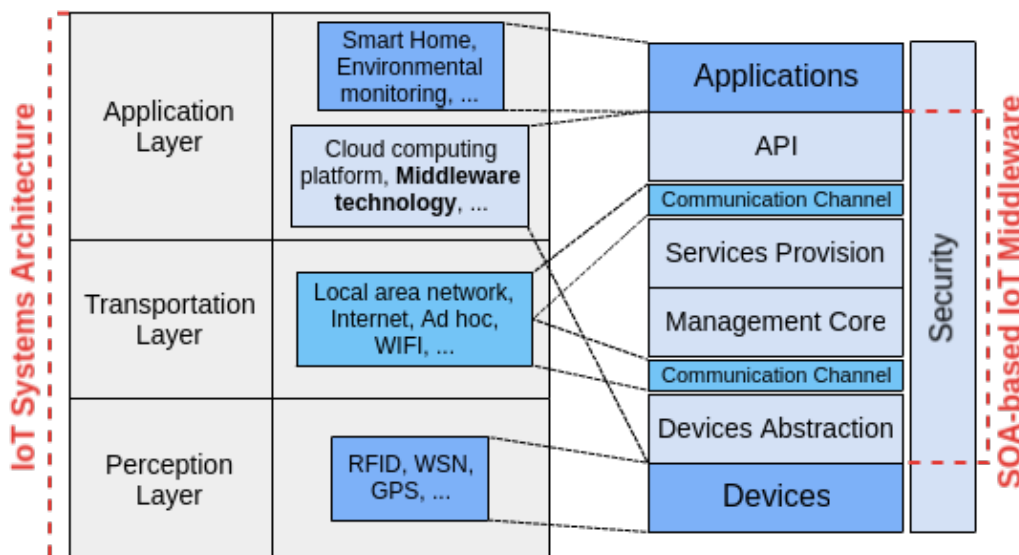


Figure 2.2 – SOA-based architecture for IoT middleware.

IoT middleware is a software layer or a set of sub-layers interposed between technological (perception and transportation layers) and application layers. The middleware's ability to hide the details of different technologies is fundamental to exempt the programmer from issues that are not directly pertinent to his/her focus, which is the development of specific applications enabled by IoT

infrastructures [6]. IoT middleware has received much attention in the last years due to its significant role in simplifying the development of applications and the integration of devices.

According to [57], middleware should also assist in improving the performance and assure the control of IoT applications through modularised access control policies. It is the connective tissue that can make or break many IoT initiatives, given that interoperability, data in motion, security, and scalability would rely heavily on middleware to support global adoption of the IoT products.

Many of the system architectures proposed for IoT middleware comply with the Service-Oriented Architecture (SOA) approach. The adoption of SOA principles allows the decomposition of complex systems into applications consisting of a system of simpler and well-defined components. In SOA architecture, each device can be able to offer its functionality as standard services, while the discovery and invocation of new features from other services can be performed concurrently. Moreover, the SOA architecture supports open and standardized communication through all layers of the web services [6]. Figure 2.2 presents an SOA-based architecture for IoT middleware according to the IoT architecture presented in Figure 2.1. We describe it as follows:

- *Applications*: This layer is dedicated to the end users and is not considered part of the middleware. Applications are able to explore all features of the middleware using an API that is provided to request information and to interact with the middleware web services.
- *API*: Applications need to implement the methods proposed by this layer in order to use the middleware services (i.e., Services Provision layer).
- *Services Provision*: It provides services to be used by applications. Each available service has its own infrastructure of devices connected to the middleware, so the devices function is abstracted into a service and provided in this layer.
- *Management Core*: This is an important layer of the middleware. It is composed of functions that allow the management of data and also each device of the environment. The core set of functions encompasses dynamic discovery of devices, status monitoring, service configuration, data management and context management.
- *Devices Abstraction*: This is the lowest layer of the middleware and facilitates the management of devices through a common language and procedures. This layer is useful because the IoT is composed of a heterogeneous set of devices, each one providing specific functions accessible through their addresses. Thus, since a device can offer discoverable web services on an IP network, it is necessary to insert an abstraction layer, which must be composed of two sub-layers. The first is the service interface, which exposes the methods available through a standard web service interface, and is also responsible for managing all incoming and outgoing messages involving communication with the devices. The second sub-layer is responsible for translating the web service methods into a set of device-specific commands to communicate with the devices. These sub-layers must be embedded into the device, thus, it is a small layer of the middleware that allows its interaction with the upper layers.

- *Devices*: It is not considered part of the middleware. It can be composed of any IoT device. These devices have their own functions provided by APIs that together can compose an IoT gateway, which is responsible for establishing a connection between the physical part and the respective logical classes of the Devices Abstraction layer.
- *Security*: The management support of security is considered one the main functions of IoT middleware systems since they must be able to provide functions related to the security for all exchanged and stored data. These functions can be either built on a single layer or distributed among all other layers of the middleware. Moreover, it must not affect the performance of the system or introduce significant communication overheads.

This SOA structure can be used in order to provide important features like interoperability, scalability, data management, context-awareness and security.

2.3 COMPaaS MIDDLEWARE

COMPaaS (Cooperative Middleware Platform as a Service) is an SOA-based IoT middleware developed by GSE/PUCRS [4]. It provides a simple and well-defined infrastructure of services. Beyond these services, there is a set of system layers that deals with the users and applications requirements, for example, request and notification of data, discovery and management of physical devices, communication channels, and data management.

COMPaaS is composed of two main systems: Middleware Core and Logical Device. Middleware Core is the system responsible for abstracting the interactions between applications and devices and hides all the complexity involved in these activities. Logical Device is the system responsible for hiding all the complexity of physical devices and abstracts their functions to the Middleware Core layer.

Figure 2.3 presents the COMPaaS middleware architecture and highlights some technical details around the integration between Applications, Middleware Core, Logical Devices, and Devices as well as the main flow of information. Each layer is described in the next items:

- *API*: It is provided by the Middleware Core layer and is responsible for managing the access to the COMPaaS services. It must be embedded into the applications. It sends data through SOAP COMPaaS web services and receives data through a TCP-based socket channel.
- *Middleware Core*: The main functions of the Middleware Core range from data management to devices integration and address the provision of high-level services to applications. This layer provides a web service SOAP, which is used by applications. Devices can make the registration process through UDP-based sockets. The Middleware Core layer sends commands to devices through REST device's web services and receives data through a TCP-based socket channel.

- *Logical Device*: This layer provides the abstraction for physical devices and must be embedded into these respective devices. Logical Devices are described through an XML file called “profile”. Each device profile contains attributes to characterize the physical device, such as name, manufacturer, function, model, data type, URI, etc. Besides the profile, the device has methods that expose its interfaces and features to the Middleware Core layer through a web service REST. Logical Device sends data to Middleware Core through TCP-based sockets.

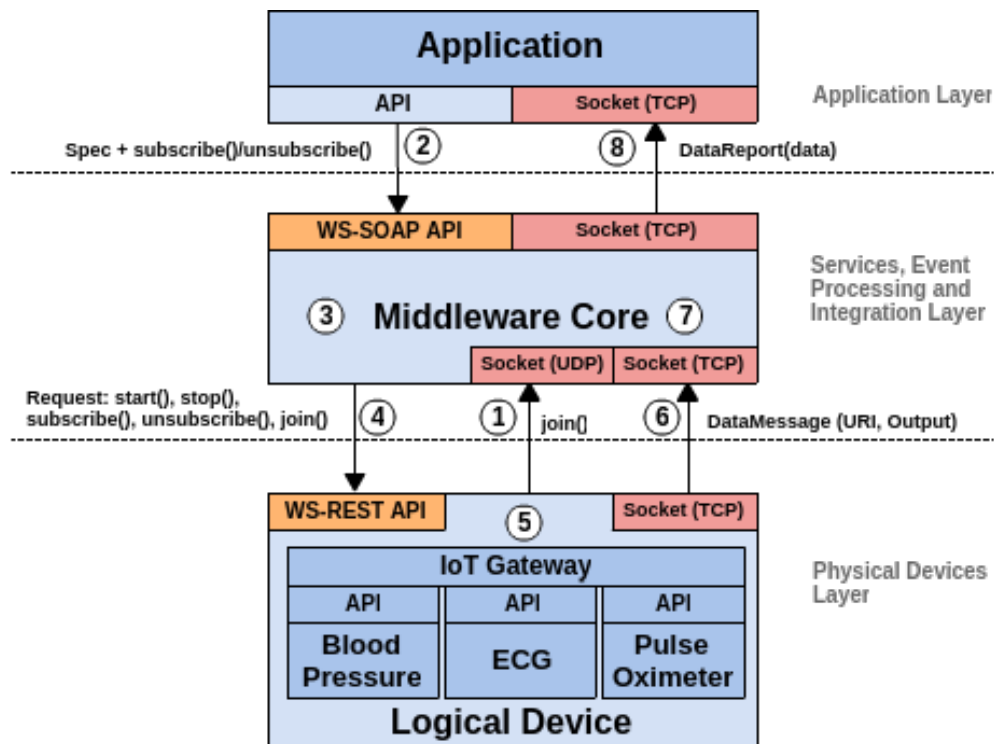


Figure 2.3 – COMPaaS architecture overview.

We explain in Figure 2.3 the interaction of the COMPaaS layers in order to request and provide useful data.

1. Logical Device register itself in the Middleware Core through messages sent via an UDP-based socket. It sends the device’s profile, which is an XML file that characterizes the devices abstracted by the Logical Device.
2. The Application uses the API to define and subscribe a specification in the Middleware Core through a SOAP web service. This specification informs the Middleware Core the subscription time and the devices that the Application desires to require data.
3. Middleware Core receives, understands and processes the Application specification (request).
4. After processing the specification, Middleware Core interacts with the Logical Device previously registered in order to start generating data. Middleware interacts with Logical Device through the methods `start()`, `stop()`, `subscribe()`, `unsubscribe()` and `join()`.

5. Logical Device understands the request through its REST web service and starts generating data from Devices.
6. Logical Device uses a TCP-based socket to send data to Middleware Core. It uses an XML file called DataMessage to encapsulate data from Devices.
7. Middleware Core processes data according to the collection cycle specification, interprets the current state of the data through Drools³ rules and, at the end of each cycle, stops and unsubscribes the Devices. After, it prepares a report to be sent to the Application, which is called DataReport.
8. Middleware Core notifies the requested reports to Application, which unsubscribes it and uses the collected data.

COMPaaS is used in this work as a reference platform. We used this architecture to deploy the security service developed and also to enable the validation of our implementations.

2.4 FINAL CONSIDERATIONS

Although COMPaaS has many features in its architecture, it does not provide security services. According to the literature and based on some important existing IoT middleware systems, security is one of the main challenges regarding the Internet of Things [6] [8] [21] [55]. In this way, we try to address this important challenge providing security services that can be used in COMPaaS and other important IoT middleware systems. In this sense, next chapter presents some important concepts, threats and requirements regarding security for these systems. In addition, we present the related work in which we highlight what has been made and also some relevant considerations related to security in IoT middleware systems.

³<http://www.drools.org/>

3. SECURITY SURVEY

This chapter presents a security background and the state-of-the-art of this work. Section 3.1 presents the threats and security requirements regarding IoT middleware systems. It helps to give a security overview and sharp the focus on what is relevant when trying to secure this kind of system. Section 3.2 presents some important concepts and definitions that will be used in this document. We present the state-of-the-art in Sections 3.3, 3.4, and 3.5. Our intention is to identify what are the current security approaches used in SOA-based IoT middleware systems and also present emerging solutions that have being used in order to mitigate the IoT security issues. Section 3.3 presents how acknowledged IoT middleware systems address security in their architectures. Section 3.4 shows some existing standards and protocols that focus on new solutions regarding IoT security. Section 3.5 presents how existing IoT communication protocols provide protection for their communication channels. Finally, Section 3.6 presents some final considerations.

3.1 THREATS AND SECURITY REQUIREMENTS

An SOA-based middleware is an important part of the IoT and demands security requirements in order to protect the surrounding systems and applications. According to [12], the management support of security has to be considered as the primary function of an SOA-based IoT middleware. In this way, it is mandatory to know what kind of attacks these systems can suffer and then be able to propose or decide which security mechanisms or countermeasures must be implemented in order to protect the system.

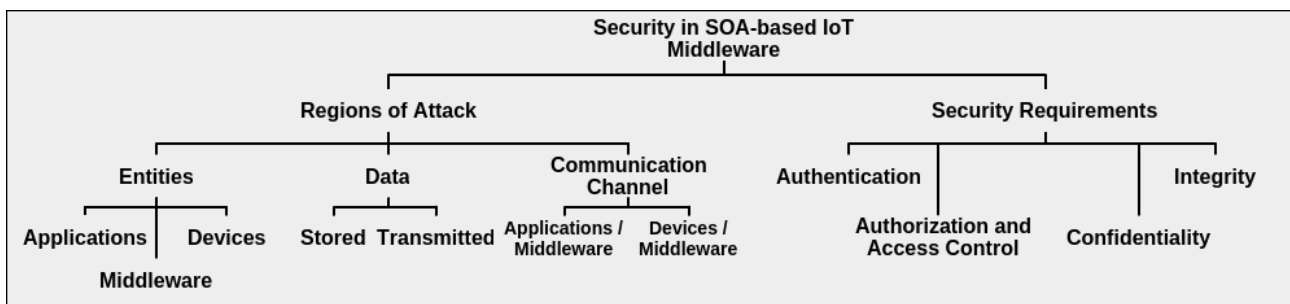


Figure 3.1 – Security taxonomy for SOA-based IoT middleware.

Figure 3.1 illustrates a security taxonomy for SOA-based IoT middleware. It identifies the most attractive targets for attackers and the security requirements for this kind of system. According to the taxonomy, the attacks can occur in entities, data and communication channels [39]. Entities attacks are related to unauthorized access in applications, middleware or devices.

Since IoT is becoming omnipresent, privacy issue has been a real concern. Disclosing users data only to authorized parties is crucial. In this sense, data attacks can happen in two ways: when

data are changed or spied during transmission between entities, and when stored data are spied or illegally modified in the data repository.

SOA-based IoT middleware systems are also vulnerable to communication channel attacks that are common on the Internet. These attacks happen in the communication between system entities. An IoT middleware has two communication channels, one with applications and another with devices. Attacks can explore both channels.

The security threats can be divided into two categories: physical attacks and non-physical attacks. Physical attacks are executed by attackers that force their way into the physically unprotected device and try to compromise it in different ways [18]. We present potential physical attacks related to IoT middleware in the following items [22]:

- *Cloning of Devices*: An untrusted manufacturer can easily clone the physical features, firmware and software, or security configurations during the manufacturing process. The manufacturer could implement additional functionality which could later be used with malicious intent.
- *Firmware Replacement Attack*: The software or firmware of a device may be updated to introduce new functionality or new features. An attacker may be able to exploit such upgrade by replacing it with malicious software. If such attack is successful, the operational behaviour of the device can be influenced and exploited afterwards.
- *Extraction of Security Parameters*: Devices deployed in an environment are usually physically unprotected and can easily be captured by an attacker. When in possession of a device, the attacker may try to extract certain security information such as security keys. Once a key is compromised, the whole network may be compromised as well.

All of the described attacks rely on the physical presence of the attacker. Although our focus is protecting data that flow in SOA-based IoT middleware communication channels, these kinds of attacks have to be taken into consideration, since if any other part of the system is susceptible to physical attacks, all security architecture can be compromised. In terms of non-physical attacks, the threats for this kind of context can be summarized as follows [22] [39]:

- *Eavesdropping Attack*: An entity into a network can be vulnerable to eavesdropping attacks which mean that if data, security parameters or keying material are exchanged in the clear (i.e., unprotected), the attacker might be able to retrieve the information exchanged between the communicating entities. Even a protected network can also be vulnerable to eavesdropping attacks. A situation example could be in the event of session key compromise, which can happen after a longer period of using the same key without renewing it.
- *Man-in-the-Middle Attack*: It is possible when messages (i.e., data, authentication credentials, keying material) are exchanged in a communication between two parties. The attacker secretly intercepts and relays messages between these two parties who believe they are communicating directly with each other.

- *Routing Attack:* Routing attacks include among others: a) Sinkhole attack, where the attacker declares himself to have a high-quality route, allowing him to do anything to all packets passing through. b) Selective forwarding, where the attacker may drop or forward packets selectively.

As we have identified potential security threats, in the next topics we describe a set of security requirements [21] [51]. Our intention is to reduce the surface of attacks in SOA-based IoT middleware systems.

- *Authentication:* It guarantees that all parties involved in the communication are who they claim they are. It is necessary to establish an authentic connection between two entities in order to exchange data and keys in a reliable manner. In IoT context, mutual authentication is required because IoT data are used in different decision making and actuating processes. Therefore, both entities need to ensure that the service is accessed by authentic parties, and it is offered by an authentic source.
- *Authorization and Access Control:* This requirement is strongly related to authentication because when a device or application is authenticated, the system must apply a correct set of rules to these entities that determine their level of access to the system. Authorization is a mandatory requirement for applications and devices since they have different privileges for accessing specific resources and services of the middleware.
- *Confidentiality:* It must be used to preserve the exchanged data in the whole architecture of the middleware. It can also ensure that data stored in entities are protected from unauthorized access. Confidentiality can be achieved to prohibit eavesdropping attacks. The selection of a particular algorithm is highly dependent on the device capability since IoT devices are generally resource-constrained entities.
- *Integrity:* The integrity is violated if a message can actively be altered during transmission without being detected. If message integrity and authenticity are guaranteed, we can significantly reduce the surface of attacks. IoT entities exchange critical data with other entities which demands that the data sent, stored, and transmitted must not be tampered either maliciously or accidentally. Integrity can also be used to protect data stored in entities.

These security requirements can be implemented by a combination of cryptographic mechanisms such as block ciphers, hash functions, signature algorithms, cryptographic mechanisms, or secure communication protocols [21] [22]. They are relevant to the IoT context since there is a very large surface of attacks. For example, when devices are deployed in uncontrolled and distinct areas.

The relationship between potential attacks and security requirements is presented in Table 3.1. To solve cloning, firmware problems, and extraction of security parameters, no simple cryptographic mechanism can be used. Rather, legal methods must be applied, like having contracts with the manufacturer regulating these issues. Of course, this technique does not result in tight security, but during this phase total security cannot be achieved [22].

Table 3.1 – Relationship between attacks and requirements.

Attacks / Requirements	AUT ¹	AAC ²	CON ³	INT ⁴
Cloning of Devices	-	-	-	-
Firmware Replacement Attack	-	-	-	-
Extraction of Security Parameters	-	-	-	-
Man-in-the-Middle	✓	-	✓	✓
Eavesdropping	✓	-	✓	-
Routing Attack	✓	-	✓	✓

¹ Authentication, ² Authorization and Access Control, ³ Confidentiality, ⁴ Integrity

Authentication, confidentiality, and integrity are required to protect against the man-in-the-middle attack [22]. It focuses on message tampering and spying of identities and authentication credentials. Message manipulation can be prevented by integrity checking since it allows to verify if data has been modified during transmission. Message leakage can be avoided by confidentiality mechanisms. Since integrity can only be provided efficiently if both parties know themselves, authentication between them is mandatory. In this sense, the three mentioned requirements can also be used to protect certain routing attacks (i.e., spoofing and altering of messages). However, these requirements are insufficient to avoid other traditional routing attacks like sinkhole and selective forwarding attacks.

Regarding eavesdropping attack, the use of confidentiality is enough to ensure that attackers do not know what data are trafficked on the network. Key management (i.e., generation, renewing and sharing of keys) is an important part of this process since keys are used to encrypt and decrypt messages. In this sense, the use of authentication, although it is not mandatory to provide confidentiality, is recommended, as it gives one more security level for the involved entities. Authentication ensures that both parties are sharing keys only with who have credentials to see them.

Authorization and access control requirement is not used to prevent any of the mentioned attacks. However, it can be important in other attacks that aim to have access to unauthorized privileges. This requirement is closely related to authentication. The middleware system is the responsible for monitoring the security policies of these requirements, that should also protect the system from illegal access coming from applications or devices.

3.2 SECURITY BACKGROUND

3.2.1 TRANSPORT LAYER SECURITY

Transport Layer Security (TLS) is a security protocol that aims to protect Internet communications. It offers security for the transport layer in order to protect applications running on top of the TCP. TLS 1.2 is the latest version defined in RFC 5246 [15]. This version offers more

flexibility for cipher suites that were hard coded in the Pseudo-Random Function (PRF) in earlier versions [29]. This means that cipher suites, which should be used to negotiate the security settings for a network connection, can be bent easily and are more difficult to break than the older versions.

TLS exchanges records, which encapsulate the data to be transferred in a particular format, over the TLS record protocol. A TLS record contains several fields, including version information, application protocol data, and the higher-level protocol used to process the application data. TLS protects the application data using a set of cryptographic algorithms to ensure the confidentiality, integrity, and authenticity of exchanged data. TLS defines several protocols for connection management that sit on top of the record protocol, where each protocol has its own record type [45].

The TLS protocols are the handshake, alert, and change cipher spec. The TLS handshake protocol is used to negotiate the session parameters. The alert protocol is used to notify the other party of an error condition. The change cipher spec protocol is used to change the cryptographic parameters of a session. In addition, the client and the server exchange application data that is protected by the security services provisioned by the negotiated cipher suite. These security services are negotiated and established with the handshake [15].

TLS was designed for reliable transport protocols, thus, it expects no loss or reordering of messages from the transport layer. If a message is lost or appears out of order, it assumes an attack and thus drops the connection. Hence, it cannot be used with unreliable transport protocols that are invariably lossy in nature [29]. This feature led to the emergence of the DTLS protocol.

3.2.2 DATAGRAM TRANSPORT LAYER SECURITY

Datagram Transport Layer Security (DTLS) is a protocol for securing network traffic which has been specified by the IETF⁴ in RFC 6347 [47]. DTLS is based on TLS and inherits some of its characteristics. It allows re-use of TLS security functions on top of User Datagram Protocol (UDP) and does not depend on reliable data transfer. According to [22], the main changes regarding TLS protocol are related to lossy data transfer: (a) handling packet loss and (b) ordering of messages.

Like TLS, DTLS also is composed of the Record Protocol and higher-level protocols. The Record Protocol provides connection security that has two basic properties: a) the connection is private by using symmetric encryption and b) the connection is reliable by including a message integrity check. We show these protocols in Figure 3.2 and describe them as follows [15] [22]:

- *Handshake Protocol*: It is used to negotiate the security parameters of a session later used for protected communication.
- *Change Cipher Spec Protocol*: It exists to signal transitions in ciphering strategies. It is used to notify the receiving party that subsequent records will be protected using the negotiated security parameters.

⁴<https://www.ietf.org/>

- *Alert Protocol*: It can be used at any time during the handshake and up to the closure of a session, signalling either fatal errors or warnings.
- *Application Data Protocol*: Application data messages are carried by the record layer and are fragmented, compressed, and encrypted based on the current connection state.

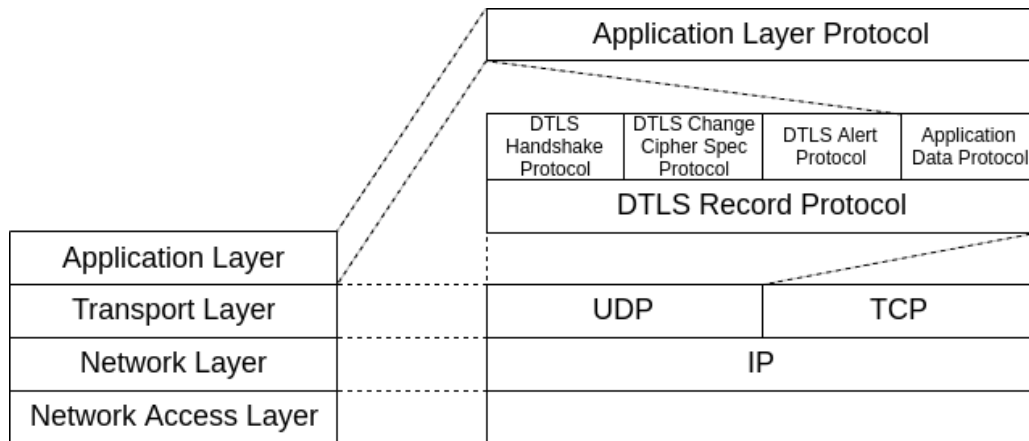


Figure 3.2 – DTLS layer.

Regarding reliability, according to [47], the basic philosophy of DTLS is to build “TLS over datagram”. The reason that TLS cannot be used over datagram is simply because packets may be lost or reordered, i.e., the TLS encryption layer does not allow decryption records alone (if the record N is not received, the record N+1 cannot be decrypted) and the handshake layer assumes that messages are delivered in order and if they are not, it fails.

To address the weaknesses that hinder TLS to be used over UDP, the DTLS uses a relay system based on counters, meaning that a message is relayed automatically if there is no return communication. Regarding the reordering problem, the DTLS enters a specific numeric sequence in each handshake message. When an end system receives a handshake message, it can determine whether the next message is waiting or if it is a different message. If it is the expected message, it will be processed. Otherwise it will be stored for further processing, until they have been received all messages prior to this [47].

3.2.3 PUBLIC-KEY CRYPTOGRAPHY

An existing problem related to many cryptographic algorithms is that they require the distribution of keys. The drawback of using shared keys is that they must be shared between communicating entities before a secure communication can start. The sharing process, however, can be vulnerable to eavesdropping since before exchange data securely we must first exchange secret keys securely.

This problem is solved with public key cryptography. In Diffie-Hellman public-key system [14], each communicating part holds a pair of keys, one public and one private. The private

key is known only by the communicating party while the public key can be given to anyone. Data encrypted using one of the keys can only be decrypted with the other. Thus, if someone wants to create a message to be read only by a particular party, it uses their public key to make the encryption, and then the other part uses its private key to decrypt the message [14]. Likewise, if someone encrypts a message with its private key, then anyone who has a copy of its public key can use it to decrypt the message. This assures the person on the receiving end that the message came from it and not from someone else, since only it has its private key. A message that someone has encrypted in this way bears its digital signature [22].

3.2.4 CERTIFICATES AND CERTIFICATE AUTHORITY

A certificate is a public key that has been digitally signed by a trusted party in order to prove that it is a valid public key [20]. We used certificates in order to prove ownership of a public key. A certificate includes information about the key, information about its owner's identity, and the digital signature of an entity that has verified the certificate's contents are correct. If the signature is valid, and the person examining the certificate trusts the signer, then they know they can use that key to communicate with its owner.

The trusted party is called a certification authority (CA), which provides a testimonial that the public key belongs to the person who owns it. We can use commercial CAs for a fee, or we can create our own. It all depends on how much authority we want to wield when proving the identity in the digital realm. If an entity signs its public key, it's called a self-signed certificate [20].

3.2.5 CIPHER SUITES

A cipher suite is a named combination of key exchange, signature algorithm, encryption, and message authentication algorithms. It is used to negotiate the security settings for a network connection using the TLS/SSL protocol [15]. It is usually expressed as a string as shown in Figure 3.3.

[SSL/TLS]_[keyexchange]_[signaturealgorithm]_WITH_[blockcipher]_[authenticationhash]
 TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256

Figure 3.3 – Cipher suite composition.

Figure 3.3 illustrates how a cipher suite is composed. In this example, it implements Elliptic-curve Diffie-Hellman Ephemeral (ECDHE) key exchange using the Elliptic-curve Digital Signature Algorithm (ECDSA), with AES-128 as the block cipher and SHA-256 HMAC for the authentication hash. In our approaches we used `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256` for the TLS implementation and `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` for the DTLS

implementation. We give a brief explanation of each algorithm that compose these cipher suites in the following items.

- *Elliptic Curves*: Elliptic Curve Cryptography (ECC) requires smaller keys compared to non-ECC cryptography to provide equivalent security. They are applicable for encryption, digital signatures, pseudo-random generators and other tasks.
- *Diffie-Hellman key exchange (DH)*: It is a key exchange method. It allows two actors, each with a public and a private key, to arrive at a shared secret (through math) without either revealing their private key.
- *Ephemeral (E)*: With standard DH key exchange, communications using Ephemeral key exchange will generate a unique secret for each message, making it impossible to bulk-decrypt them.
- *Digital Signature Algorithm (DSA)*: It is a signing algorithm thought to be secure for sufficiently large key sizes (2048 and up).
- *Advanced Encryption Standard (AES-128)*: It is the current symmetric encryption of record. It is a specification for the encryption of electronic data. The 128 denotes the keysize.
- *Authenticated Encryption with Associated Data (AEAD)*: It is a block cipher mode of operation which simultaneously provides confidentiality, integrity and authenticity assurances on data. Decryption is combined in a single step with integrity verification. GCM and CCM are examples of modes of operation.
 - *Galois/Counter Mode (GCM)*: GCM refers to the mode of operation used by the encryption algorithm and provides authentication and confidentiality.
 - *Counter with Cipher Block Chaining-Message Authentication Code (CCM)*: CCM is an authentication encryption block cipher mode designed to provide both authentication and confidentiality. The 8 means that it uses eight octets (64 bits) for authentication resulting in a ciphertext that is eight octets longer than the corresponding plaintext.
- *SHA-256*: It is a one-way hash function used for message verification. It is a member of the SHA-2 family.

3.3 SECURITY IN IOT MIDDLEWARE SYSTEMS

Once security support is essential for IoT middleware systems, it is important to investigate how some relevant works address the security issues. During our research, we found some important surveys regarding security in IoT middleware systems architectures [8] [17] [21]. We decided to update some information about IoT middleware considering only the relevant works for our research. Our goal was to understand what they provide regarding security and how they do it.

The VIRTUS Middleware [13] is an IoT middleware that relies on the open eXtensible Messaging and Presence Protocol (XMPP) to provide a reliable and secure communication channel for distributed applications. It uses TLS protocol to provide communication channel protection, SASL protocol to allow authentication, and XMPP's built-in mechanisms for access control. SASL supports some different systems including token-based approaches such as OAuth2⁵, username/password, X.509 certificates, etc. VIRTUS is not clear when describes security for client-to-server based communications. On the other hand, it describes the use of SASL to ensure full server federation in server-to-server communications.

SOCRADES [52] is a middleware focused on industrial environments, specifically for manufacturing shop floors [17]. It is based on SOAP and Web Services and utilizes the WS-Security standard for encryption and message integrity. It also supports access control and authentication for applications and devices. In this system, devices and back-end services may only be accessed by clients that have certain authorization privileges and provide correct credentials for authentication. However, the lack of explicit support for tokens and federated security and identity models creates a significant challenge in key distributions and centralized identity for this system [17].

COSMOS middleware [24] is based on SOA and provides a services layer to allow the integration of applications and devices. It dedicates its security approach in authentication for applications and devices, access control and data confidentiality. It has a module named security manager that controls the access to sensor networks in the middleware. This module is the responsible for providing the protection of the system. However, this work does not present a technical overview of its security approach.

MOSDEN [44] is a plug-in-based IoT middleware for mobile devices that allows to collect and process sensor data without programming efforts. Its architecture supports sensing as a service model and allows the device management through the Virtual Sensor Manager. MOSDEN is also interoperable with other cloud-based middleware solutions such as GSN [1]. There is an insufficient description of the security architecture to do any meaningful review.

SIRENA [10] is a SOAP/WS-based IoT middleware. It concentrates its security approach in communication channel protection and authentication for applications and devices. It uses the DPWS (Devices Profile for Web Services) technology in its framework, which defines a minimal set of implementation requirements to enable secure Web Service messaging on resource-constrained devices. DPWS uses TLS/SSL to establish a connection between applications and devices. Moreover, it uses the X.509.v3 certificate as a cryptographic credential to allow authentication.

Hydra [7], also known as Linksmart, is a European Union funded project that developed an SOA-based IoT middleware system. Its architecture is based on SOAP web services. The security model utilizes WS-Security [32]. Hydra approach also uses symmetric keys, which is a challenge for IoT because each key must be uniquely created, distributed and updated upon expiry into each device, creating a major key management issue [17]. Hydra also offers a service called TrustManager, which is a system that uses the cryptographic capabilities to support a trusted identity for IoT devices

⁵<http://oauth.net/2/>

using Public Key Infrastructure (PKI) and certificates. Hydra middleware does not offer any policy based access control for IoT data and does not address the secure storage of data for users.

ubiSOAP [11] is an SOA approach that builds a middleware for Ubiquitous Computing and IoT based on Web services standards and SOAP protocol. There is an insufficient description of the security architecture to do any meaningful review.

MUFFIN [58] proposes a middleware framework to interact with devices and their data through the use of Web services. It is based on SOA and has mechanisms that allow the management of devices and interoperability between systems. There is an insufficient description of the security architecture to do any meaningful review.

Table 3.2 – Security in IoT middleware systems.

IoT Middleware	AUT ¹	AAC ²	CON ³	INT ⁴	SCP ⁵
VIRTUS	✓	✓	✓	✓	TLS
SOCRADES	✓	✓	✓	✓	WS-Security
COSMOS	✓	✓	✓	-	-
MOSDEN	-	-	-	-	-
SIRENA	✓	X	✓	✓	TLS
HYDRA	✓	X	✓	✓	WS-Security
UBISOAP	-	-	-	-	-
MUFFIN	-	-	-	-	-
Our Work	✓	X	✓	✓	TLS/DTLS

Legend: ✓ = yes, X = no, - = there is an insufficient description.

¹ Authentication, ² Authorization and Access Control, ³ Confidentiality, ⁴ Integrity, ⁵ Security Communication Protocol.

Table 3.2 presents the relation between the IoT middleware systems and the security requirements. The last column of the table shows how each system provides protection for the communication channel to secure data transmission. MOSDEN, UBISOAP and MUFFIN do not provide information about their security approaches. On the other hand, all the remaining systems provide some kind of protection that complies with three or more security requirements. Authentication, confidentiality and integrity are implemented by almost all systems that provide security. However, authorization and access control requirement is addressed only by VIRTUS, SOCRADES and COSMOS.

We observe that most of IoT middleware systems are focused on protecting transmitted data since they implement authentication, confidentiality and integrity, which are strongly related to information protection. TLS is chosen as the security communication protocol by VIRTUS and SIRENA. On the other hand, SOCRADES and Hydra use WS-Security, which is a standard approach to protecting SOAP web services.

According to [19], WS-Security has as the primary focus the use of XML Signature and XML Encryption to provide end-to-end security. Moreover, it provides support for authentication, confidentiality and integrity. However, WS-Security is very heavyweight and has issues with key distribution, federated identity and access control. In end-to-end situations, authentication, con-

Confidentiality, and integrity can also be enforced on Web services through the use of TLS (e.g., by sending messages over HTTPS). In addition, applying TLS can significantly reduce the overhead involved by removing the need to encode keys and message signatures into XML before sending. In this sense, the work in [21] calls for the need of non-intrusive security solutions for IoT systems such as key management and authentication, which are crucial to have an efficient solution.

Regarding our proposed solution, the definition of the security services are able to provide all security requirements. However, in this table, we are considering only the security service we implemented (i.e., CCP service), which is composed of TLS and DTLS approaches. In this case, the authorization and access control requirement is not addressed. DTLS approach is able to provide the same security requirements as in TLS in order to protect data transmitted over communication channels. However, DTLS ensures a lower data transmission time than TLS in most scenarios, including resource-constrained environments, even being the DTLS not designed for this purpose.

3.4 EMERGING SECURITY STANDARDS AND PROTOCOLS

The use of well-defined and established standards and protocols is essential to provide security for IoT middleware systems. However, there are still some significant challenges to be addressed. In this sense, some emerging works propose new security approaches and architectures that may be used to mitigate the IoT middleware security issues.

The work in [53] presents a snapshot of the latest progress in the oneM2M standardization such as architecture, protocols, security aspects, device management, and abstraction technologies. They adopted a resource-based data model. All services are represented as resources, which are associated with the common service functions to support registration, configuration, management, and security.

In [27] the authors introduce a fully implemented two-way authentication security scheme for the Internet of Things based on existing Internet standards, especially the DTLS protocol. The proposed security approach is based on RSA and works on top of standard low power communication stacks.

Authors in [46] proposed a DTLS header compression scheme for 6LowPAN. This approach mitigates the communication overhead and message fragmentation problems seen on constrained networks. However, to use DTLS header compression, a constrained device has to support this approach, which leads to low scalability. If the payload size is much larger than the header length, as in a certificate-based handshake message, problems can occur due to fragmentation.

The work in [16] proposes a security architecture for an IoT transparent middleware. Its protection measures are based on existing technologies for security, such as AES, TLS, and OAuth⁶. It integrates privacy, authenticity, integrity and confidentiality of exchanged data to provide security

⁶<http://oauth.net/>

for smart objects, services, and users. However, they do not focus on SOA standard and lightweight approaches.

Authors in [63] propose a security architecture for IoT systems that are oriented to the perception layer, and analyze two proposals for IoT security. The first one [30] proposes a self-managed security cells oriented to the middleware layer, which can undertake a series of functions such as strategic management and access control, and establish the security architecture of SOA-based IoT systems. The second one [61] proposes a secure transmission system and develops a three-layer structure consisting of management center, root services, and local services, aiming the IoT as a whole.

In [2] the authors present a reference architecture for building cloud-enabled IoT applications in support of pervasive systems aimed at achieving trustworthiness among end-users in IoT scenarios. They describe a case study that leverages this reference architecture to protect sensitive user data in IoT application implementation. Also, they present a Secure, Private and Trustworthy Protocol (named SPTP), that was prototyped for addressing critical security, privacy and trust concerns surrounding mobile, pervasive and cloud services in collective intelligence scenarios.

An important technology that has become the choice for implementing SOA architectures is Web Services. This communication standard provides a framework for systems integration independent of programming language and operating system. However, the security of Web services depends not only on the security of the services themselves but also on the confidentiality and integrity of the XML-based SOAP messages used for communication [38]. XML Signature and XML Encryption are used in web services to provide integrity and confidentiality, and can be reused to provide SOAP security as well.

The work in [31] designs an information security system architecture for the Internet of Things based on a lightweight cryptography. The authors propose the development of security protocols for authentication, an encryption/decryption and a signature verification. All to verify the legality of access devices and to protect the confidentiality and integrity of transmitted data.

The authors in [60] propose an architecture that leverages the security concepts both from content-centric and traditional connection-oriented approaches. It is based on the concept of object security that introduces security within the application payload. They rely on secure channels established using DTLS for key exchange. They provide a mechanism to protect from replay attacks by coupling their scheme with the CoAP application protocol.

The SMARTIE (Secure and sMARTer ciTIEs data management) project [5] works on security, privacy and trust for data exchange between IoT devices and consumers of their information. SMARTIE is a distributed framework to share large volumes of heterogeneous information for the use in smart city applications, enabling end-to-end security and trust in information delivery for decision-making purposes following data owner's privacy requirements. This framework follows a data-centric paradigm, which offers highly scalable and secure information for IoT applications.

3.5 SECURITY IN IOT COMMUNICATION PROTOCOLS

Along with the development of new security protocols and standards, new communication protocols are arising for the Internet of Things. These protocols implement existing security solutions since this is an essential requirement in IoT environments. In this section, our intention is to show which kind of security is used by important IoT communication protocols to provide protection for data transmitted over open networks.

The CoRE (Constrained RESTful Environments) Working Group within the IETF (Internet Engineering Task Force) has defined the Constrained Application Protocol (CoAP) as a generic web-based protocol for RESTful-constrained environments, targeting M2M applications that cope with HTTP for integration with the existing web [23]. CoAP [50] is a relatively simple request and response protocol providing both reliable and unreliable forms of communication. To protect the transmission of sensitive data, secure CoAP mandates the use of DTLS as the underlying standard security protocol to guarantee communication channel protection on a point to point basis.

LightweightM2M (LWM2M) [54] is a standard for management of IoT devices. It is based on CoAP protocol (DTLS for security) and, therefore, is optimized for communications over sensor or cellular networks. LWM2M provides an extensible object model that allows enabling application data exchanges in addition to the core device management features. LWM2M is a new, still on-going effort to create a new technical standard for remote management of IoT devices and a bit of service enablement and application management.

Message Queuing Telemetry Transport (MQTT) [33] is a protocol designed to connect the physical world devices and networks with middleware and applications. It is useful for mobile applications that require small size, low power usage, minimized data packets, and efficient distribution of information to one or many receivers. It implements TLS protocol as a way to provide authenticity, confidentiality, and integrity between devices, middleware, and applications.

The Extensible Messaging and Presence Protocol (XMPP) [49] uses the XML text format as its native type, making person-to-person communications natural. It runs over TCP and provides security using TLS protocol. In the IoT context, XMPP offers an easy way to address a device. XMPP is especially handy if data is going between distant, mostly unrelated points, just like the person-to-person case. Its strengths in addressing, security, and scalability make it ideal for IoT applications.

Table 3.3 – Security in IoT communication protocols.

IoT Protocol	Security Protocol
CoAP	DTLS
LWM2M	DTLS
MQTT	TLS
XMPP	TLS

Table 3.3 shows the security protocols used in each IoT communication protocol mentioned. We note that TLS and DTLS prevail as being the only ones used. MQTT and XMPP aim to ensure interoperability and security irrespective of the application environment. They use TLS to ensure the protection of their communications. On the other hand, COAP and LWM2M use DTLS since they have lightweight proposals and are focused on resource-constrained environments. However, while DTLS is seen as a viable choice for IoT constrained environments, it is not entirely optimal for resource-constrained networks as it was designed for traditional computer networks [22].

3.6 FINAL CONSIDERATIONS

In this chapter we presented some important concepts regarding security. Also, we presented the state-of-the-art related to IoT middleware, security protocols and standards. We discussed about some important threats and security requirements that should be considered when we look for security in the Internet of Things. In this sense, we also demonstrated what has been made regarding security towards the future. Most of the existing IoT middleware systems does not address security requirements able to provide data protection in the whole system architecture. However, there are some institutions working on the standardization of the Internet of Things, providing new solutions mainly when we talk about security. Next chapter provides the definition and implementation details of our proposed security services that can be used as a basis towards the definition of a security architecture for SOA-based IoT middleware systems.

4. PROPOSED APPROACH

This chapter presents the proposed approach for this work. Section 4.1 presents the definition and description of the proposed security services for SOA-based IoT middleware. Moreover, Section 4.2 presents the implementation of TLS and DTLS approaches that compose one of the proposed services, the CCP (i.e., Communication Channel Protection). Finally, Section 4.3 presents a brief discussion regarding some important questions faced during the implementation process.

4.1 SECURITY FOR SOA-BASED IOT MIDDLEWARE

In this work we defined a set of security services that should be used in an SOA-based IoT middleware in order to provide security countermeasures against some dangerous threats (e.g., man-in-the-middle and eavesdropping). Here we describe how the proposed services should work in an SOA-based IoT middleware system.

As a first step towards the provision of the proposed security services, we decided to start by implementing the CCP service in this work. Our motivation is to try to mitigate an important security problem of the IoT that also has been inherited by e-health scenarios, which is the necessity of having secure communications between system entities respecting an acceptable response time for applications. More details concerning how to provide confidentiality, integrity, and authenticity in the communication channels between Logical Devices and Middleware Core are presented in the next subsections.

4.1.1 SECURITY SERVICES

According to the previous chapters, security and privacy support is definitely crucial for the proper functioning of an SOA-based IoT middleware solution. Besides, although several middleware systems try to provide some kind of security service in their architectures, recurrently they do not supply it in an efficient manner able to mitigate the main threats existing in IoT environments. Thus, there is a gap for the definition of a standard security architecture for SOA-based IoT middleware systems [55].

The definition of security services is an essential step in order to provide the required protection. These services should preferably be based on established security standards and protocols. In this way, the correct choice of the “common set of security standards and protocols” to be used is the basis to protect the system adequately. The accomplishment of the security services for SOA-based IoT middleware means to guarantee both confidentiality, integrity, authenticity, and access control for all entities, channels and exchanged/stored data.

The set of security requirements that must be mandatory for SOA-based IoT middleware can be summarized in four security services: Applications and Devices Authentication (ADA), Authorization and Access Control (AAC), Data Confidentiality and Integrity (DCI), and Communication Channel Protection (CCP). We show in Figure 4.1 how these requirements can compose the proposed services. We believe the implementation of these services in an SOA-based IoT middleware architecture would help to protect the system against some dangerous threats adequately.

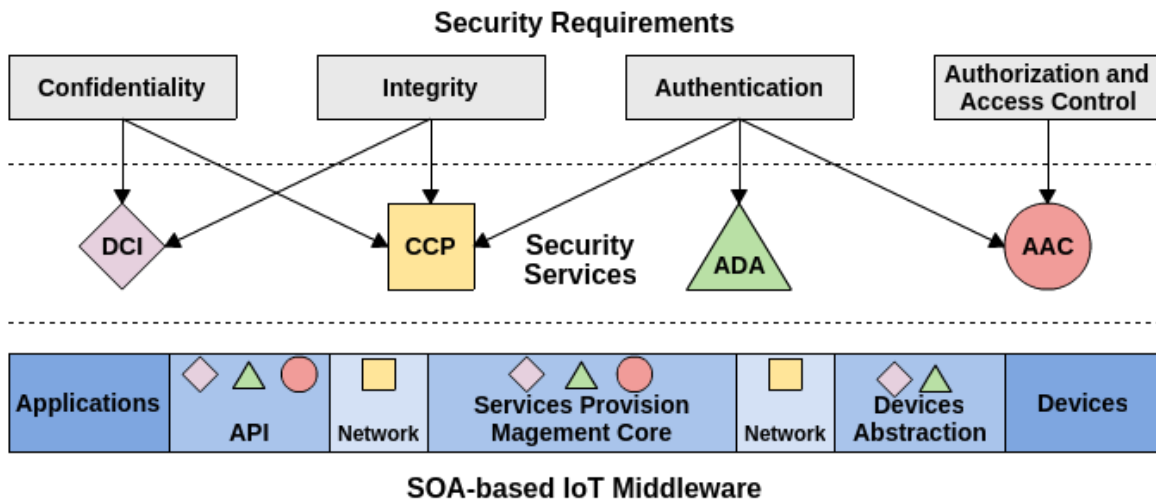


Figure 4.1 – Composition of the security services.

Figure 4.2 identifies where each security service could be embedded in an SOA-based IoT middleware system to ensure the protection of entities, data and channels. ADA should provide authentication between applications/devices and middleware. AAC service is related only with applications. CCP is responsible for protecting all exchanged data between IoT middleware channels. Finally, DCI service should ensure the protection of all data stored and exchanged (together with CCP). Next we present a brief definition for each service.

- *Applications and Devices Authentication (ADA)*: ADA service would be responsible for allowing the authentication for applications, middleware, and devices, ensuring the correct identities for them. As applications and devices interact periodically with the middleware, it is essential to make sure that there is no intruder in the system. This service should be used along all middleware architecture (i.e., from devices to applications) and could use a public key infrastructure (PKI) with certificates to ensure authenticity between all parties.
- *Authorization and Access Control (AAC)*: This service would be responsible for allowing the access of information for an authenticated user, allowing the management of applications and devices. It can be based on permissions and roles, and is directly related to the authentication service since it only performs its functions based on previously authentication. This service should prohibit the unauthorized access to any object or information present in the middleware. Such information is confidential and should be accessed only by authorized users.

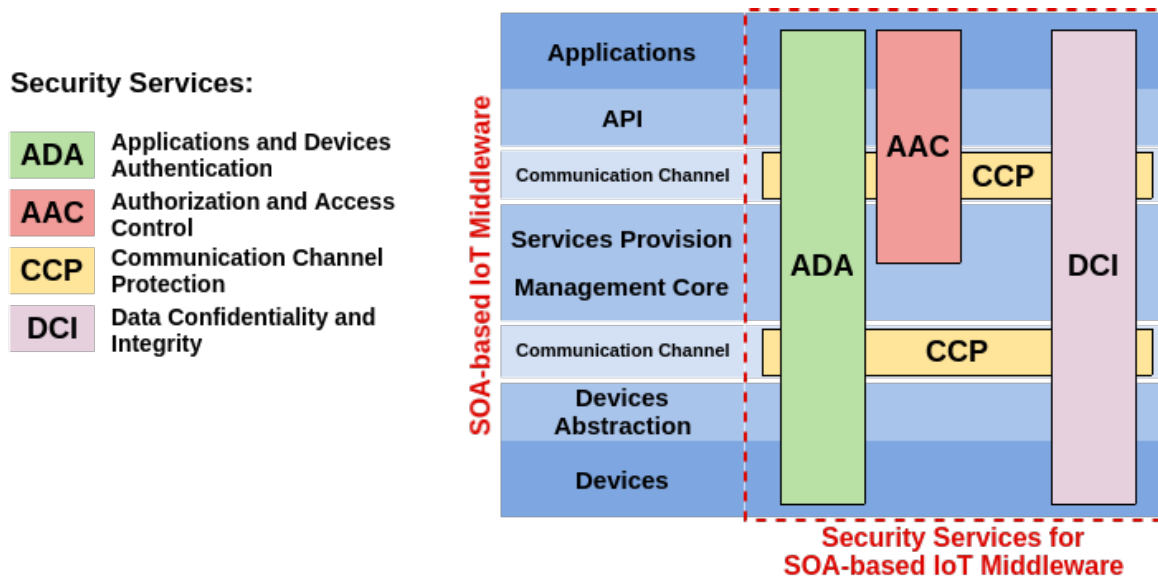


Figure 4.2 – Security services on SOA-based IoT middleware.

- *Data Confidentiality and Integrity (DCI)*: DCI service would involve all layers of the architecture. Part of it should be based on data encoding and decoding mechanisms, which would be responsible for encrypting and decrypting data related to storage. The other part of this service would be related to data integrity mechanisms. It would be responsible for allowing integrity checking of the stored data. DCI would also be responsible for protecting exchanged data. However, this is directly related to CCP service, since some protocols provide confidentiality and integrity. In this sense, DCI inherits exchanged data protection from CCP.
- *Communication Channel Protection (CCP)*: CCP service would be responsible for protecting the communication channel in order to ensure the protection of data sent by devices, middleware or applications against eavesdropping and tampering during transmissions. It should take place between devices and middleware and also between middleware and applications. It can be provided by the implementation of security protocols such as TLS and DTLS.

We chose to define the CCP service on COMPaaS as a first step toward the definition of all the other services in the future. We decided to implement TLS and DTLS protocols to protect data sent from devices to middleware because this communication layer is vulnerable to security attacks. Moreover, due to the fact that IoT devices and applications are becoming extremely heterogeneous and widely used, a security service for the communication layer is a mandatory choice. Incorporate security to this particular communication layer into the COMPaaS middleware implies in add a TLS/DTLS layer on both sides of the system architecture. Figure 4.3 presents the CCP service architecture composed of these protocols. We explain both implementations in the next subsections.

COMPaaS was based originally on an unprotected channel to send data from Logical Device to Middleware Core: the messages were transmitted by socket through a TCP-based channel. We decided to provide two new secure ways of sending data from Logical Devices to Middleware Core (i.e., only one direction). The first one continues using a TCP-based channel, however, with the

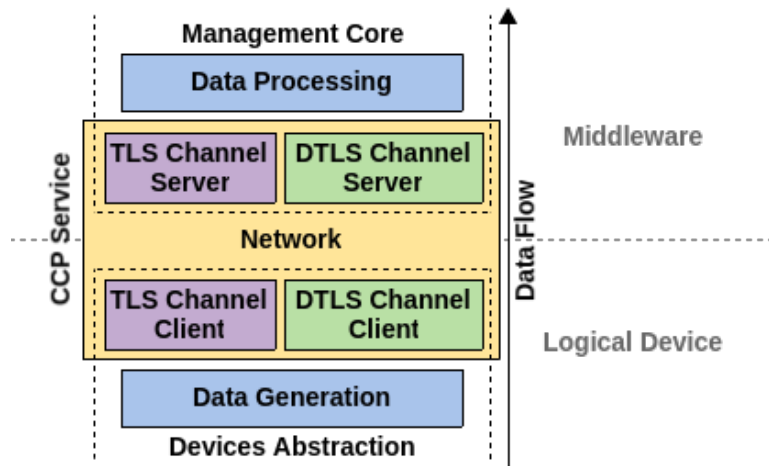


Figure 4.3 – CCP service architecture.

TLS protocol (i.e., all messages are sent by secure sockets in a protected way). For the second one, as we were looking for a protocol capable of minimizing the elapsed time during the communication transmission, we decided to use the DTLS protocol, which is based on TLS. However, it runs over a UDP-based channel, which helps reducing the data transmission time.

In addition, the implemented version of DTLS ensures similar characteristics to the TLS protocol, including the managing of packet loss [22]. We chose these two protocols because they are already established in the research community. Besides, both protocols fit well with our goal that tries to mitigate the overhead imposed by security communication approaches when applied to critical environments, as the e-health scenario. The e-health field demands reliability and acceptable response time as essential requirements since a wrong interpretation of data can directly influence in the life condition of a patient.

4.1.2 TLS ON COMPaaS

The TLS implementation on COMPaaS communication channels is concentrated in three main classes: TLSServer, TLSClient and TLSChannel. TLSServer listens for incoming connections on a specified port. Each time a connection comes in, TLSServer creates a new TLSChannel instance to process the connection. Processing a connection means receiving text messages from TLSClient and sending them to the upper layers of the Middleware Core. When TLSClient starts up, it initiates a connection with TLSServer. TLSClient keeps this connection open throughout the middleware-device session. Each message is sent along this connection. Figure 4.4 provides a schematic overview of the TLS approach that was implemented. We describe each of the steps in next items.

1. TLSServer and TLSClient have to establish a connection between them. After this, they are able to run a full handshake (see APPENDIX C for more detail) started by the client side to get mutual authentication. In this way, the next data generated by the device will be sent in a protected manner. A new authentication is requested always they close their connection.

2. Data is generated by the devices, encapsulated in an XML file called DataMessage, and sent to the TLSClient, which is responsible for providing security and sending data.
3. TLSClient ensures the message protection through the inner classes and sends it to the TLSChannel as an application data protected.
4. Data is transmitted in a secure way over the network.
5. After arriving at the Middleware Core side, the message is decrypted and passed to the TLSChannel class. It reconstructs the original DataMessage and sends it to TLSServer.
6. Data is sent in a decrypted way to the upper layer, which is responsible for processing it and notifying the application.

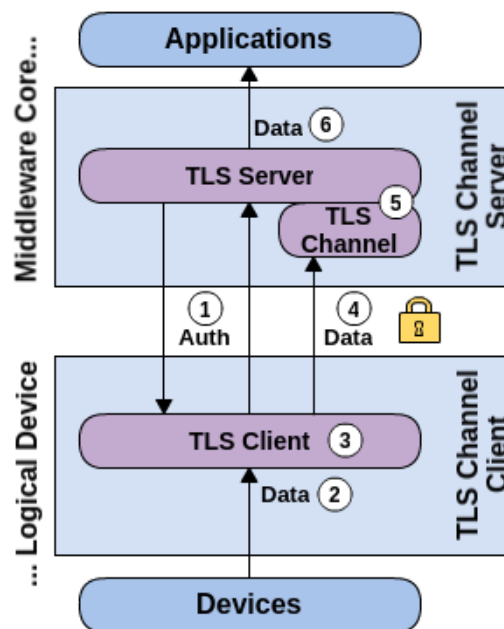


Figure 4.4 – Schematic overview of the TLS approach.

4.1.3 DTLS ON COMPaaS

As in TLS, the DTLS implementation in the COMPaaS communication channels is concentrated in three main classes: DTLS Server, DTLS Client and DTLS Connector, that is shared by both sides (i.e., Middleware Core and Logical Device). DTLS Server and DTLS Client open their channels and provide a DatagramSocket to be used during the communication. DTLS Client is responsible for starting the handshake. On the other hand, DTLS Connector is responsible for all the processing (encrypting, decrypting), key and messages exchange, etc. Figure 4.5 provides a schematic overview of the DTLS approach that was implemented. We describe each of the steps in next items. Some steps are the same as in TLS implementation.

1. When DTLSConnector at client side receives a message from DTLSClient (device's message), it checks whether a DTLS session exists with the required DTLS Server at the server side:
 - (a) If an active DTLS session is found, the DTLSConnector sends the device's message as encrypted application data.
 - (b) If a DTLS session exists with the server side that has been closed already, an abbreviated handshake (see APPENDIX B for more detail) must be executed to resume the session.
 - (c) If there is no such DTLS session, a full handshake (see APPENDIX B for more detail) must be executed.
2. Data is generated by devices, encapsulated in an XML file called DataMessage, and sent to the classes responsible for providing security and sending data.
3. Data goes through the DTLSClient and reaches the DTLSConnector, which encodes the message and sends it as an application data protected with the negotiated security parameters.
4. Data is transmitted in a secure way over the network.
5. Once the data arrives at Middleware Core side, it is decrypt by the DTLSConnector and passed on to the DTLS Server.
6. DTLS Server only sends the decrypted data to the upper layer, which is responsible for processing it and notifying the application.

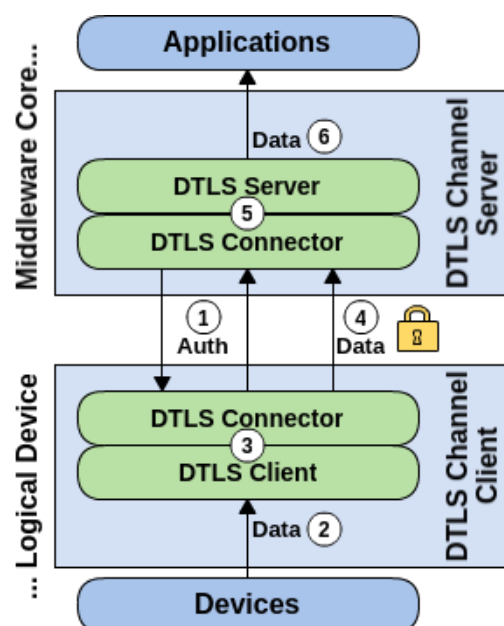


Figure 4.5 – Schematic overview of the DTLS approach.

4.2 IMPLEMENTATION

The implementation of TLS and DTLS security approaches is based on the functional description presented in Section 4.1. We decided to implement these two protocols in the proposed CCP service in order to protect the communication channels of COMPaaS middleware. In this way, we intend to have more than one option to send data in a secure way and also respecting the acceptable response time required by the e-health scenario illustrated in this work. Sections 4.2.1 and 4.2.2 present the technical description of the implemented approaches. Once DTLS is based on TLS, both implementations are quite similar.

4.2.1 TLS APPROACH

For the implementation of the TLS protocol we have used the Java Secure Socket Extension (JSSE)⁷, a Java package that enables secure Internet communications. It provides a development framework and an implementation for a Java version of the TLS protocol and includes functionality for data encryption, server and client authentication, and message integrity. It uses secure sockets to protect the communications. Secure sockets are used the same way that standard sockets are, except that they transparently encrypt any data that pass through them.

We used public-key encryption to ensure the privacy of messages sent over the Internet. Both Middleware Core and Logical Device must have a pair of keys, one public and one private. However, before they can exchange messages, they must generate and store these keys. We give a brief description of how to generate keys in APPENDIX D. For the loading of keys we used the Java keytool, which is a key and certificate management tool that manages a keystore (database) of cryptographic keys, X.509 certificate chains, and trusted certificates.

Middleware Core and Logical Device have a file containing its own public and private keys. Middleware Core also has the Logical Device's public key certificate, as well as Logical Device has the Middleware Core's public key certificate. There is no need to hide the public keys from any other party, they can be given out freely.

Before data can be sent across the TLS connection, Middleware Core and Logical Device must negotiate and exchange key information. The complete description of the handshake can be seen in APPENDIX C. Basically, it involves the following steps:

1. TLSClient sends a ClientHello message to TLSServer.
2. TLSServer answers with a ServerHello message, followed by a ServerCertificate message, a ServerKeyExchange message, and a ServerHelloDone message.

⁷<https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html>

3. TLSClient sends a ClientCertificate message to TLSServer followed by a ClientKeyExchange message, a CertificateVerify message, a ChangeCipherSpec message, and a finished message.
4. In response, TLSServer sends a ChangeCipherSpec message and a finished message. At this point, TLSClient and TLSServer can exchange data in a protected way.

During the TLS handshake and securing application data, a lot of different cryptographic algorithms are needed, such as for digital signatures, encryption, certificates and certificate validation, message digests (hashes), secure random number generation, and key generation and management. The Java Cryptography Architecture (JCA)⁸ provides a large set of APIs which helps integrating security into the application code.

Before show how we implemented the TLS protocol on COMPaaS, we present some important steps that are required to provide protection using JSSE. In order to TLSClient initiate a secure socket connection with TLSServer, we must carry out the following steps [15]:

1. Create a SecureRandom number, which is random enough that it will not make the encryption vulnerable to attack.
2. Create a KeyStore object containing the TLSServer's public key.
3. Create a KeyStore object containing the TLSClient's public/private key pair, including its public key certificate.
4. Create a TrustManagerFactory from the TLSServer's KeyStore. This is used to authenticate the TLSServer.
5. Create a KeyManagerFactory from the TLSClient's KeyStore. This is used for encrypting and decrypting data.
6. Create an SSLContext object, using the KeyManagerFactory, the TrustManagerFactory, and the SecureRandom.
7. Use the SSLContext to create an SSLSocketFactory.
8. Use the SSLSocketFactory to create an SSLSocket, which acts just like a regular Socket, except that it is secure.

In order to TLSServer listen for incoming connections, we must carry out a similar set of steps:

1. Create a SecureRandom number.
2. Create a KeyStore object containing the TLSClient's public key.

⁸<http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html>

3. Create a KeyStore object containing the TLSServer's public/private key pair, including its public key certificate.
4. Create a TrustManagerFactory from the TLSClient's KeyStore. This is used to authenticate the TLSClient.
5. Create a KeyManagerFactory from the TLSServer's KeyStore. This is used for encrypting and decrypting data.
6. Create an SSLContext object, using the KeyManagerFactory, the TrustManagerFactory, and the SecureRandom.
7. Use the SSLContext to create an SSLServerSocketFactory.
8. Use the SSLServerSocketFactory to create an SSLServerSocket, which acts just like a regular ServerSocket, except that it is secure.

Now we show how each one of these steps was implemented. We examine only the client-side (Logical Device) process in detail. However, the server-side process is nearly the same. After, we demonstrate the differences between the two sides.

Creating a SecureRandom object consists in use only two lines as shown in Figure 4.6. The first line creates the SecureRandom. The `nextInt()` method in the second line generates the next random object.

```

1  secureRandom = new SecureRandom();
2  secureRandom.nextInt();

```

Figure 4.6 – Creating a SecureRandom object.

Next, we must create some KeyStore objects. We created an empty KeyStore using the static method `KeyStore.getInstance()`, and initialize it using its `load()` method, as illustrated in Figure 4.7. We are also reading the key information from "server.public", which contains the server side's public key.

```

1  private void setupServerKeystore()
2      throws GeneralSecurityException, IOException {
3      serverKeystore = KeyStore.getInstance("JKS");
4      serverKeystore.load(new FileInputStream("server.public"),
5                          passphrase.toCharArray());
6  }

```

Figure 4.7 – Creating KeyStore objects.

We created a KeyStore of type "JKS", which is the standard keystore format used in JSSE. We also must read the client key pair from "client.private" as shown in Figure 4.8.

```

1 private void setupClientKeyStore()
2     throws GeneralSecurityException , IOException {
3     clientKeyStore = KeyStore.getInstance("JKS");
4     clientKeyStore.load(new FileInputStream("client.private"),
5                         "public".toCharArray());
6 }

```

Figure 4.8 – Reading client key pair.

The next step is to use the KeyStore objects created to initialize key and trust manager factories. We created a TrustManagerFactory from the server's keystore, which will be used to authenticate the remote server as shown in Figure 4.9. The TrustManagerFactory is of type "SunX509", "509" is the name of the certification protocol we used in our implementation. In the second code line, the TrustManagerFactory is loaded with the server's keystore.

```

1 TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
2 tmf.init(serverKeyStore);

```

Figure 4.9 – Creating the TrustManagerFactory.

We also created a KeyManagerFactory from the client's KeyStore, as shown in Figure 4.10.

```

1 KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
2 kmf.init(clientKeyStore , passphrase.toCharArray());

```

Figure 4.10 – Creating the KeyManagerFactory.

Next we must create an SSLContext. It contains all the key and certificate information mentioned so far. It is used to create an SSLSocketFactory, which creates secure sockets. Once an SSLContext is created at the start of an application, it can be used for each connection needed, as long as each connection uses the same keys. To create an SSLContext, we use our factories and the SecureRandom, as shown in Figure 4.11. We created an SSLContext of type "TLSv1.2". We then initialize it with the TrustManagerFactory and KeyManagerFactory objects we created a few steps back.

```

1 sslContext = SSLContext.getInstance("TLSv1.2");
2 sslContext.init( kmf.getKeyManagers() ,
3                 tmf.getTrustManagers() ,
4                 secureRandom );

```

Figure 4.11 – Creating an SSLContext.

All these mentioned steps give an SSLContext, which is used to make a connection to the TLSServer, as shown in the code of Figure 4.12. We have just created a secure connection to the port on the Middleware Core host. The last two lines indicate the cipher suite that the client desires

to use in the connection with the server. More details of the cipher suite algorithms are given in subsection 3.2.5.

```

1  SSLSocketFactory sf = sslContext.getSocketFactory();
2  SSLSocket socket = (SSLSocket)sf.createSocket(host, port);
3  String[] string = {"TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256"};
4  socket.setEnabledCipherSuites(string);

```

Figure 4.12 – Connecting to the TLSServer.

Setting up the server side (Middleware Core) is almost the same as setting up the client side (Logical Device). Of course, the server reads its key information from “client.public” and “server.private”, rather than from “server.public” and “client.private”. Also, the code to carry out the final step (establishing a connection) and some classes are a little different for the server side. We use the server classes such as SSLServerSocket and SSLServerSocketFactory instead of SSLSocket and SSLSocketFactory, as shown in Figure 4.13.

```

1  SSLServerSocketFactory sf = sslContext.getServerSocketFactory();
2  SSLServerSocket ss = (SSLServerSocket) sf.createServerSocket(port);
3  ss.setNeedClientAuth(true);

```

Figure 4.13 – Creating an SSLServerSocket channel.

Note that we used `SSLServerSocket.setNeedClientAuth()`. This is the way to indicate that the client should authenticate itself. Client applications do not authenticate themselves by default, so with this call, the client authentication will be part of the handshaking process.

By the time the TLSServer receives a connection from a TLSClient, it creates an exclusive channel (i.e., an instance of the TLSChannel class) to receive data from that client. This code is shown in Figure 4.14.

```

1  Socket client = ss.accept();
2  System.out.println("Got connection from " + client);
3  TLSChannel clientChannel = new TLSChannel(client);
4  clientChannel.start();

```

Figure 4.14 – Receiving a TLSClient connection.

Once the connection has been established, the TLSChannel is able to receive data sent from TLSClient. All transmitted data after the well-established connection are protected according to the algorithms used by the cipher suite.

Regarding error handling, all possibly thrown exceptions are caught at a single point in the TLS implementation. To make the classes stable against unexpected errors, the whole method that receives the message, parses the message and reacts according to the message, is surrounded with a try/catch block that catches all possible exceptions.

4.2.2 DTLS APPROACH

For the implementation of the DTLS protocol we have used Scandium (Sc)⁹ [22], a part of the Californium Eclipse Project¹⁰ (Java-based implementation of CoAP) that provides security for CoAP¹¹ [50]. It implements DTLS 1.2 to secure applications through ECC with pre-shared keys, certificates or raw public keys. It is an open source project that provides an implementation for a Java version of DTLS protocol and includes functionality for data encryption, server and client authentication and message integrity. It uses datagram sockets to send data from one node to another since messages are sent through a UDP-based channel.

Before sending the first data to Middleware Core, Logical Device should start the handshake to achieve authentication between DTLS`Server` and DTLS`Client`. The handshake is basically the same as in TLS, except for two messages, the `HelloVerifyRequest` (has been added by the DTLS specification as a DoS countermeasure) and `ClientHello 2` (identical to the first one sent, except for the added cookie taken from the `HelloVerifyRequest`). The full handshake can be seen in APPENDIX B. After the handshake, data can be sent in a secure way.

In order to achieve authentication, the DTLS handshake protocol specifies the `Certificate` message. It contains a certificate chain of X.509 certificates where the first certificate in the chain contains the entity's public key. The loading, as in the TLS implementation, is done with the Java `keytool`. In this case, there exist two keystore files: the keystore containing a private key and a certificate chain with the first certificate having the corresponding public key, and the truststore that contains all trusted certificates. The work in [22] shows how to use `keytool` and `openssl` commands to generate a keystore containing a certificate chain signed by a trusted certificate authority.

Loading of the certificate chain and private key can be easily achieved by the use of an "alias" given to the private key and certificate chain. The Java code in Figure 4.15 illustrates it.

```

1  KeyStore keyStore = KeyStore.getInstance("JKS");
2  InputStream in = new FileInputStream(KEY_STORE_LOCATION);
3  keyStore.load(in, KEY_STORE_PASSWORD.toCharArray());
4
5  Certificate[] certificates = keyStore.getCertificateChain("alias");
6  PrivateKey privateKey = keyStore.getKey("alias",
7  KEY_STORE_PASSWORD.toCharArray());

```

Figure 4.15 – Using alias to load certificate chain and private key.

According to [25], the use of raw public keys could be a good solution for DTLS used in constrained environments, since instead of sending a large X.509 certificate chain to achieve authentication, an entity can send only its public key in the `Certificate` message while the validation

⁹<https://github.com/eclipse/californium.scandium>

¹⁰<https://www.eclipse.org/californium/>

¹¹<http://coap.technology/>

of the public key is achieved by an out-of-band method. This case can be handled almost the same way as sending the whole certificate chain. The loading of the certificate through the keystore can be done the same way as already explained. Extracting the public key can be achieved as shown in Figure 4.16.

```
1 byte [] rawPublicKey = certificateChain[0].getPublicKey().getEncoded();
```

Figure 4.16 – Extracting public key.

The *getEncoded()* method applies the right encoding to the public key and, therefore, the raw public key can be added to the payload of the Certificate message without any further modifications.

Now we show how the DTLSConnector is configured. It is the responsible for understanding and processing messages that arrive or leave the channel. Remember that DTLS uses UDP, so we can not establish a connection between DTLSserver and DTLSclient. DTLSserver should open its channel creating a DatagramSocket and linking it to an “IP:port” through a DTLSConnector instance and wait for a DTLSclient request for authentication. The way DTLSclient uses to send their data is basically through a network address (i.e., “IP:port”). Figure 4.17 presents the server-side code to configure a DTLSConnector. The client-side is nearly the same, we have to change only the network address.

```
1 DtlsConnectorConfig.Builder builder =
2   new DtlsConnectorConfig.Builder(
3     new InetSocketAddress(DEFAULT_PORT));
4 builder.setIdentity((PrivateKey) keyStore.getKey("server",
5   KEY_STORE_PASSWORD.toCharArray()),
6   keyStore.getCertificateChain("server"), true);
7 builder.setTrustStore(trustedCertificates);
8 dtlsConnector = new DTLSConnector(builder.build(), null);
9 dtlsConnector.setRawDataReceiver(new RawDataChannelImpl(dtlsConnector));
```

Figure 4.17 – Configuring a DTLSConnector instance.

We created a *DtlsConnectorConfig* instance with the corresponding *InetSocketAddress*. We set the connector’s identifying properties using a private key and a corresponding issuer certificates chain. In DTLSserver, the key and certificates are used to prove the server’s identity to the client. In DTLSclient, the key and certificates are used to prove the client’s identity to the server. We also set the root certificates that the connector should use when verifying a system’s identity based on an X.509 certificate chain. We instantiate a *DTLSConnector* object for securing data exchanged between DTLSserver and DTLSclient. It creates and manages (i.e., starts, sends, receives, etc.) a *DatagramSocket*. Finally, we open a data receiver channel where the DTLSserver receives data from DTLSclient.

DTLSConnector is also responsible for creating an instance from *Record* class. It is mainly used for encrypting messages that are leaving and also for decrypting messages that are arriving. For

any message type that leaves through the method `setFragment()`, it calls the method responsible for encrypting the message, as shown in Figure 4.18.

```

1      ...
2      case ALERT:
3      case APPLICATION_DATA:
4      case HANDSHAKE:
5      case CHANGE_CIPHER_SPEC:
6          byteArray = encryptFragment(byteArray);
7          break;
8      ...

```

Figure 4.18 – Calling the encryption method.

For any message type that arrives through the method `getFragment()`, it calls the respective method to decrypt the message, as shown in Figure 4.19.

```

1      ...
2      case ALERT:
3          fragment = decryptAlert(currentReadState);
4          break;
5      case APPLICATION_DATA:
6          fragment = decryptApplicationMessage(currentReadState);
7          break;
8      ...

```

Figure 4.19 – Calling the decryption method.

For both encryption and decryption, the Record instance uses a cipher suite to know what algorithms should be employed. In the DTLS implementation, we used the cipher suite `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8`. More details of the cipher suite algorithms are given in subsection 3.2.5.

Related to error handling, as in TLS implementation, all possibly thrown exceptions are caught at a single point in the DTLS implementation. To make the classes stable against unexpected errors, the whole method that receives the message, parses the message, and reacts according to the message, is surrounded with a try/catch block that catches all possible exceptions.

4.3 DISCUSSION

In this Section we point some important issues regarding our implementations. The goal of the implementations is to provide security in a non-intrusive way using existing and well-defined security protocols. The best way we found to implement the CCP service was dividing each implementation in Java classes, as explained in the above sections. The integration of the security layer with the middleware was allowed by these classes. We only instantiated them and set some required

parameters (e.g., protocol version, cipher suite, authentication mode, etc.) in order to integrate the security approaches with a communication channel of the COMPaaS middleware.

Regarding the implementation, we used Java 8 and the version 1.2 of the protocols TLS and DTLS from the packages JSSE and Scandium, respectively. For the TLS implementation we used the `TLSChannel` class to manage each connection from different TLS clients. In TLS, a connection should be established between server and client to allow the handshake between them. The `TLSClient` is responsible for starting the handshake. In this sense, we had difficulty to set the desired cipher suite. After some research, we discovered that the difficulty was related to the Java version, so we installed Java 8 and used the method `setEnabledCipherSuites()` passing the desired cipher suite as parameter. This should be made at the `TLSClient` side since the `TLS`Server adapts its methods to the chosen cipher suite.

For the DTLS implementation, we used the `DTLSConnector` class for both sides (i.e., server and client) since there is no previous connection between the two parts because DTLS is based on UDP. In this sense, both sides are able to use the methods created in `DTLSConnector` (e.g., `send()`, `receive()`, etc.). On the other hand, `DTLS`Server and `DTLS`Client have some different characteristics, for example, as in TLS, the client is responsible for starting the handshake. Regarding the cipher suite, it is set in the `DTLSConnector` class.

We also had some difficulty to load and use the keys and certificates. But again, after some research, we found relevant references that help us to understand and implement this important part of the approaches [22]. We used `KeyStore` java class in order to load/store/use keys and certificates. Regarding the generation of keys, we used some commands that are illustrated in APPENDIX D. We strongly used the work in [22] as a reference for key management.

5. EVALUATION

This chapter presents the evaluation of the CPP service implemented in this work. Section 5.1 describes the use case used in this validation. We divided the validation into two main test requirements: functionality and performance. We describe and present goals, results, and also discuss each test in Sections 5.2 and 5.3.

5.1 USE CASE DESCRIPTION

5.1.1 EMERGENCY MEDICAL SERVICE (EMS)

Emergency Medical Services (EMS) is a type of emergency service dedicated to provide out-of-hospital acute medical care, transport to definitive care, and other medical transport to patients with illnesses and injuries which prevent the patient from transporting themselves [26]. The goal of most emergency medical services is to provide treatment to people who demands urgent medical care.

This use case allows examining one of the main threat in the security area that is the attack on data during transmission. In addition, since we are talking about e-health environments, we have to deal with an essential requirement that is to ensure an acceptable response time during the transmission of data. We intend to show how non-intrusive security approaches are important in communications involving patient's data.

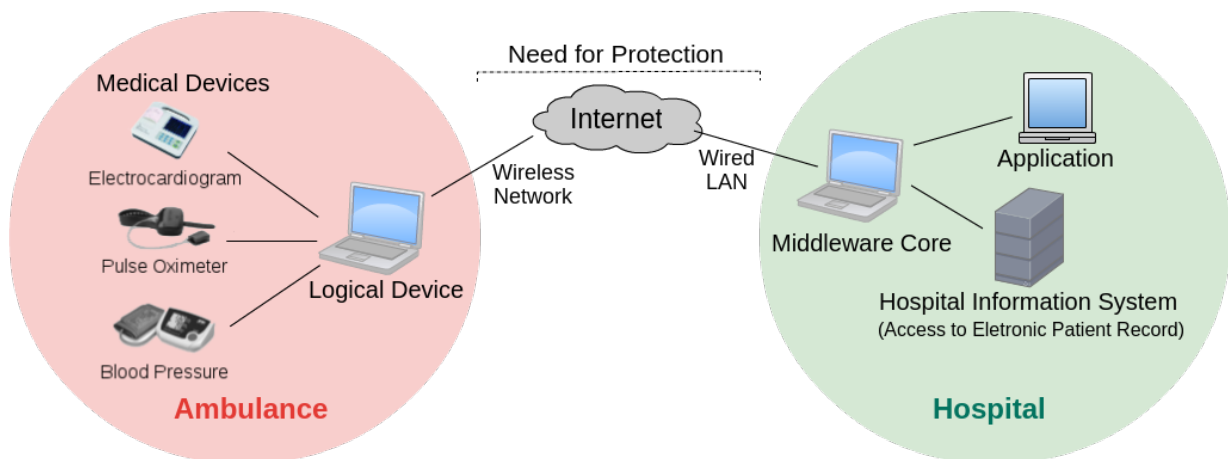


Figure 5.1 – COMPaaS middleware applied in an EMS scenario.

Let's consider the scenario presented in Figure 5.1. We assume that an ambulance answers an emergency call in a street. Middleware Core system already knows about the existence of the ambulance (i.e., the ambulance is already registered) and waits for a connection request. The ambulance arrives at the scene and uses a wireless network infrastructure to establish a connection

and authenticate itself with the Middleware Core. Since Logical Device system is already configured in the ambulance, someone has only to start the system when it arrives at the scene. Thus, after the patient is connected to medical devices, they start generating data.

The Logical Device system is connected to three medical devices (i.e., an electrocardiogram machine, a pulse oximeter, and a blood pressure monitor) through their APIs, that compose what we call IoT gateway (see COMPaaS architecture in Section 2.3). This system collects the devices data of a patient every 1.5 seconds and sends it to the Middleware Core through a wireless network. The Middleware Core, which is located in the hospital, receives the data through a wired connection and then processes, stores, and sends these data to the corresponding application. The emergency room physicians receive the data and can analyze the current life condition of the patient and also can prepare themselves adequately to receive the patient in an adequate way. Although simple, this EMS use case illustrates some vulnerability points that must be addressed in this scenario, as such: (1) authentication between Logical Device and Middleware Core; (2) confidentiality and integrity to protect data of any tampering and/or eavesdropping during the communication sessions; and (3) key management of the secret keys.

The implemented service focuses exclusively to ensure the protection of data sent from Logical Device to Middleware Core. As all the data sent between these systems flows in open networks, all the communications between them should be protected to ensure data confidentiality and integrity. In this sense, the communication between Middleware Core and Applications should also be protected. However, in this example, we assume that Applications and Middleware Core are communicating through an already protected internal network of the hospital.

5.1.2 ENVIRONMENT SETUP

In this subsection we present all the configuration used in the EMS use case. We describe in the following items the role of Applications, Middleware Core, Logical Devices, and Devices.

- *Applications*: They are responsible for receiving data from Middleware Core and for showing them in graphics to the physicians.
- *Middleware Core*: It is responsible for: (1) authenticating with the Logical Device and (2) receiving, processing and sending data to Applications.
- *Logical Devices*: It is responsible for: (1) authenticating with the Middleware Core, (2) abstracting devices' functions to the Middleware Core, and (3) sending the data generated by Devices.
- *Devices*: They are abstracted by Logical Devices and provide patient's data to the upper systems.

To perform the tests we used an infrastructure composed of three computers. Table 5.1 shows the used configuration for each one.

Table 5.1 – Used configuration for computers.

Features / Systems	Applications/API	Middleware Core	Logical Devices/Devices
Processor	Intel Core 2 Duo	Intel Xeon(R) W3530	Intel Core i3-2330M
Cores	2	4	2
GHz	2.20	2.80	2.20
Hyperthreading	0	8	4
Main Memory	4GB	4GB	6GB
Cache	2M	8M	3M
Operating System	Ubuntu 14.04 LTS	Ubuntu 14.04 LTS	Ubuntu 14.04 LTS

We simulated 1, 5, 10 and 15 Applications and Logical Devices, and only 1 Middleware Core. Each Logical Device hosted three medical devices that were also simulated. A wired LAN connection was used between Applications and Middleware Core while a wireless LAN connection was used between Logical Devices and Middleware Core. Logical Devices were set to receive data from the medical devices and to forward it to the Middleware Core every 1.5 seconds (i.e., the corresponding interval of data sending). Each message had about 1 KB. The goal of each Logical Device was to represent an ambulance in a city infrastructure. According to estimations reported by the World Health Organization (WHO)¹², the optimal amount of ambulances for each city is one unit for every 150 thousand inhabitants. In this case, our use case can cover a city with a population around 2 million inhabitants. We show the test environment in Figure 5.2.

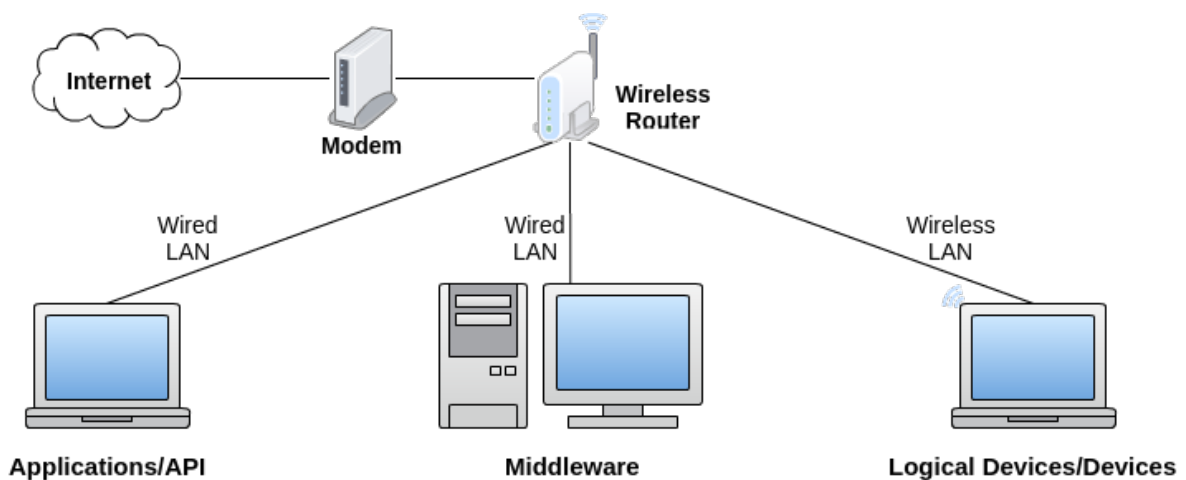


Figure 5.2 – Environment configuration for tests.

The messages sent from Logical Device to Middleware Core are composed of data from all medical devices. The XML file that encapsulates these data is exemplified in Figure 5.3. Each XML is composed of ambulance ID and its respective devices.

The devices were simulated in agreement with specification standards [40] [41] [42] [43]. We can simulate various levels of severity for each one. Figure 5.4 shows the user interface we used to simulate real data generation.

¹²<http://www.who.int/>

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <DataMessage name="Ambulance X" id="4531">
3   <devices>
4     <device name="Blood Pressure">
5       <uri>http://IP:port/CompositeResource/monitors/blood_pressure</uri>
6       <output corrupt="false">
7         <value type="ArrayOfValues">
8           <values>
9             <value type="Output" priority="LOWEST" corrupt="false">
10              <value type="Double">100.5</value>
11            </value>
12            <value type="Output" priority="HIGHEST" corrupt="false">
13              <value type="Double">50.3</value>
14            </value>
15          </values>
16        </value>
17        <timestamp>1450200628040</timestamp>
18      </output>
19    </device>
20    <device name="ECG">
21      <uri>http://IP:port/CompositeResource/monitors/ecg</uri>
22      <output type="Double" priority="HIGHEST" corrupt="false">
23        <value type="Double">115.0</value>
24        <timestamp>1450200628051</timestamp>
25      </output>
26    </device>
27    <device name="Pulse Oximeter">
28      <uri>http://IP:port/CompositeResource/monitors/spo2</uri>
29      <output type="Double" priority="HIGHEST" corrupt="false">
30        <value type="Double">83.8</value>
31        <timestamp>1450200628032</timestamp>
32      </output>
33    </device>
34  </devices>
35 </DataMessage>

```

Figure 5.3 – DataMessage content.

Medical Devices Simulation

Heart beat, blood pressure and pulse oximeter monitor at Ambulance 4531

Middleware

Connected

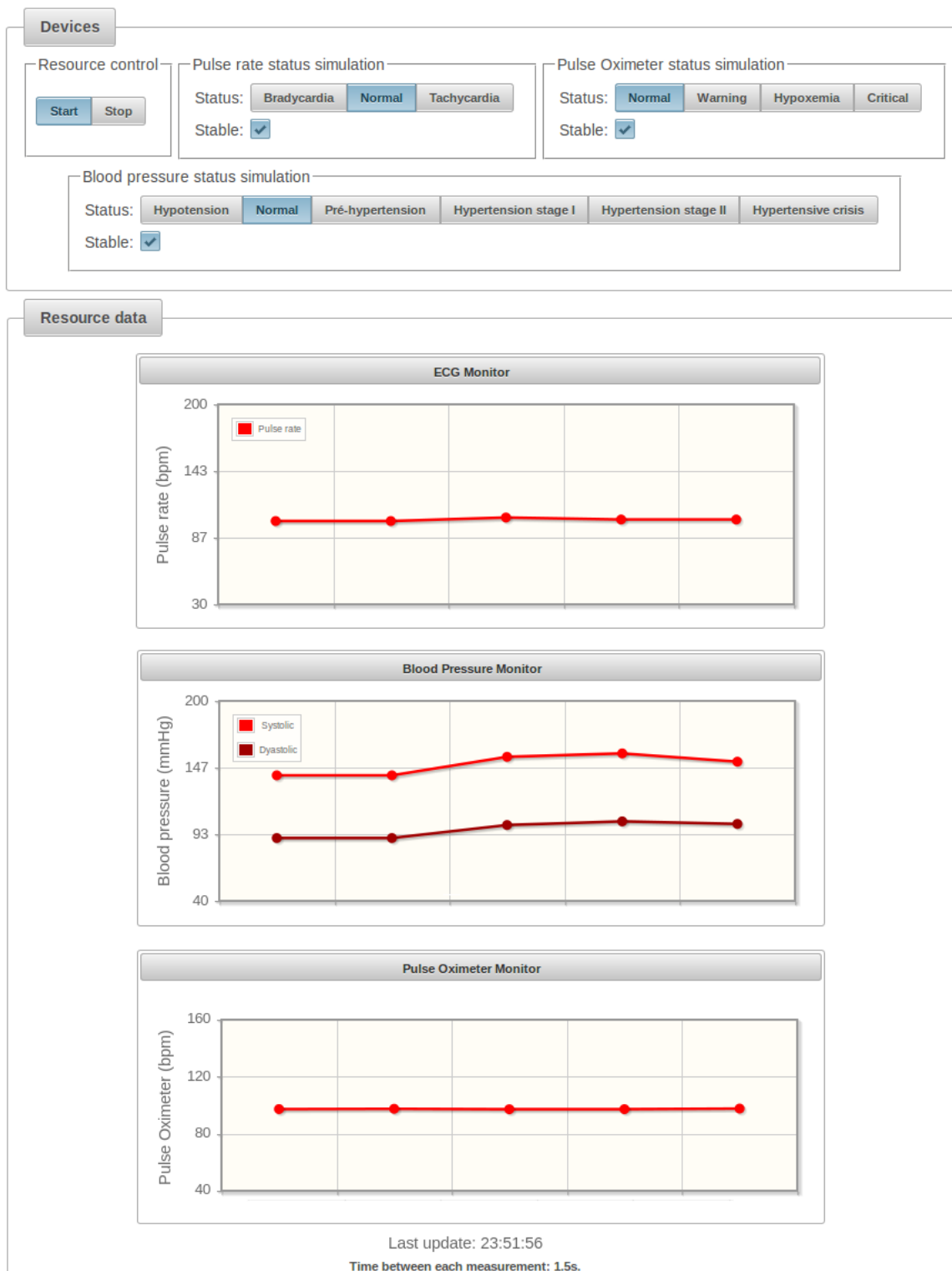


Figure 5.4 – User interface used for medical devices simulation.

5.2 FUNCTIONALITY TESTS

The software packages of TLS and DTLS used in our approaches are based on their respective specifications [15] [47]. According to them, an implementation is secure as long as the specification has been followed correctly. Since the used packages were already tested during their development, we consider that each specification has been followed in a proper way, and we don't need to perform exhaustive tests around the packages' functionality. Instead of this, we conducted some tests as a means to increase the confidence that we have implemented the CCP service (i.e., TLS and DTLS approaches) correctly.

Our goal with these tests was to verify if: (1) Middleware Core and Logical Device are authenticating with each other correctly; and (2) data are transmitted in a safe way. We performed all the tests on COMPaaS and used Wireshark¹³, a network protocol analyzer, to get information about authentication handshakes and transmitted messages.

Table 5.2 – Summary of the functionality tests on COMPaaS.

Protocol	Mutual Authentication	Data Protection
TLS	✓	✓
DTLS	✓	✓

Table 5.2 uses a mark (✓) to represent if the tests' goals were achieved in an adequate way for both TLS and DTLS implementations. We detail each test in the next paragraphs and figures.

Regarding the authentication requirement, we captured the messages exchanged during the full handshake of TLS and DTLS. The only difference between both authentication schemes protocols is that TLS does not provide the HelloVerifyRequest from Server side. Next figures illustrate that both handshakes are exchanging the required messages to get authentication.

Source	Destination	Protocol	Info
Client	Server	TLSv1.2	Client Hello (1)
Server	Client	TCP	5685 > 37612 [ACK] Seq=1 Ack=150 Win=44800 Len=0 TSval=6642549 TSecr=6642549 (2)
Server	Client	TLSv1.2	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
Client	Server	TCP	37612 > 5685 [ACK] Seq=150 Ack=1016 Win=45824 Len=0 TSval=6642552 TSecr=6642552
Client	Server	TLSv1.2	Certificate, Client Key Exchange (3)
Server	Client	TCP	5685 > 37612 [ACK] Seq=1016 Ack=814 Win=46208 Len=0 TSval=6642564 TSecr=6642554
Client	Server	TLSv1.2	Certificate Verify, Change Cipher Spec, Hello Request, Hello Request (3)
Server	Client	TCP	5685 > 37612 [ACK] Seq=1016 Ack=950 Win=47488 Len=0 TSval=6642564 TSecr=6642564
Server	Client	TLSv1.2	Change Cipher Spec (4)
Client	Server	TCP	37612 > 5685 [ACK] Seq=950 Ack=1022 Win=45824 Len=0 TSval=6642576 TSecr=6642566
Server	Client	TLSv1.2	Hello Request, Hello Request (4)
Client	Server	TCP	37612 > 5685 [ACK] Seq=950 Ack=1067 Win=45824 Len=0 TSval=6642576 TSecr=6642576
Client	Server	TLSv1.2	Application Data
Server	Client	TCP	5685 > 37612 [ACK] Seq=1067 Ack=983 Win=47488 Len=0 TSval=6642586 TSecr=6642576

Figure 5.5 – TLS handshake to get authentication.

Figure 5.5 presents the TLS handshake captured by Wireshark. We understand Middleware Core and Logical Device by Server and Client respectively. The handshake is divided into four flights: (1) Logical Device sends a ClientHello message to Middleware Core; (2) Middleware Core

¹³<https://www.wireshark.org/>

answers with the messages ServerHello, Certificate, ServerKeyExchange, CertificateRequest and ServerHelloDone; (3) Logical Device sends the Certificate to Middleware Core, besides the messages ClientKeyExchange, CertificateVerify, ChangeCipherSpec and a finished message; (4) Middleware Core sends ChangeCipherSpec and finished messages. We noted that Middleware Core and Logical Device exchanged all the required messages according to TLS handshake specification [15]. More details about each exchanged message are presented in APPENDIX C.

Source	Destination	Protocol	Info
Client	Server	DTLSv1.2	Client Hello (1)
Server	Client	DTLSv1.2	Hello Verify Request (2)
Client	Server	DTLSv1.2	Client Hello (3)
Server	Client	DTLSv1.2	Server Hello (4)
Server	Client	DTLSv1.2	Certificate (Fragment) (4) (4)
Server	Client	DTLSv1.2	Certificate (Reassembled), Server Key Exchange, Certificate Request, Server Hello Done
Client	Server	DTLSv1.2	Certificate (Fragment) (5)
Client	Server	DTLSv1.2	Certificate (Reassembled), Client Key Exchange, Certificate Verify (5)
Client	Server	DTLSv1.2	Change Cipher Spec, Hello Request (5)
Server	Client	DTLSv1.2	Change Cipher Spec, Hello Request (6)
Client	Server	DTLSv1.2	Application Data
Server	Client	DTLSv1.2	Application Data

Figure 5.6 – DTLS handshake to get authentication.

Figure 5.6 presents the DTLS handshake also captured by Wireshark. We understand Middleware Core and Logical Device by Server and Client respectively. Different from TLS, the DTLS handshake is divided into six flights: (1) Logical Device sends the first ClientHello message to Middleware Core; (2) Middleware Core answers with a HelloVerifyRequest message; (3) Logical Device sends the second ClientHello message to Middleware Core and starts the handshake; (4) Middleware Core sends important messages to Logical Device, including the Certificate; (5) Logical Device processes the previous messages and answers the Middleware Core with another important messages that will establish the authentication by the Middleware Core side; (6) Logical Device receives the last messages from Middleware Core and also agrees with the session establishment. Middleware Core and Logical Device exchanged all the required messages according to DTLS handshake specification [47]. More details about each exchanged message are presented in APPENDIX B.

Regarding data protection, we captured and analyzed the exchanged “Application Data” messages generated by devices with Wireshark. Both protocols sent their messages in a safe way since they are encrypted during the transmission, ensuring confidentiality and integrity. Figure 5.7 presents the TLS message while Figure 5.8 shows the DTLS exchanged message. Each captured message is composed of some fields like protocol version, length, epoch, sequence number and encrypted application data. The protocol version presented in each message confirms that we used the last and more consolidated version of TLS and DTLS. The length of the exchanged messages was around 1KB. It is related to the XML file that composes the message exemplified in Figure 5.3. Depending on how large is the message size, it can affect the final time directly. Epoch and sequence number are used to manage ordering and delivery of messages. More details about their specific functions are given in [22]. Finally, the “Encrypted Application Data” field has the encoded data. We decrypted some of the exchanged messages using the respective keys in Wireshark user interface to be sure that each encrypted message corresponds to the data sent by the devices.

```

▼Secure Sockets Layer
  ▼TLSv1.2 Record Layer: Application Data Protocol: http
    Content Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 1048
    Encrypted Application Data: 000000000000002dd47a46ba6dbf761c003dcc011c47852...

```

Figure 5.7 – TLS message encrypted during transmission.

```

▼Datagram Transport Layer Security
  ▼DTLSv1.2 Record Layer: Application Data Protocol: Application Data
    Content Type: Application Data (23)
    Version: DTLS 1.2 (0xfefd)
    Epoch: 1
    Sequence Number: 1
    Length: 1038
    Encrypted Application Data: 000100000000001510dd9c048038e585a004d1d83bfa2eb...

```

Figure 5.8 – DTLS message encrypted during transmission.

5.3 PERFORMANCE TESTS

Measure the performance of the implemented protocols and try to compare it with other implementation approaches is an interesting step to evaluate them. However, according to [22], this practice is a complicated process since most of the TLS/DTLS implementations available for performance testing comparison only support PSK (i.e., Pre-Shared Key). In addition, no standardized benchmarking has been realized so far. Due to this reason, the results presented by other TLS/DTLS implementations depend exclusively on the hardware used, which difficult the comparison with our implementation.

We decided to compare our two implementations against each other and try to verify how much overhead TLS/DTLS creates in a transmission compared to a non-secure communication channel from the COMPaaS middleware. We are not considering the time spent during the handshake between the parts, only the time spent between sending and receiving messages after the handshake. We decided not to measure this time, as it does not interfere directly in each message exchanged during a connection. On the other hand, we are aware that TLS and DTLS spend considerable time with their handshakes. In this way, we intend to cope with these issues in future steps of our work.

Data is generated by medical devices and, after encoded, they are sent to the Middleware Core, which decodes, processes, and sends them to the corresponding Applications. We measured the elapsed time for four implemented approaches: NoSecTCP (open channel, only sockets over TCP), TLS (protected channel, TLS over TCP), NoSecUDP (open channel, only datagram sockets over UDP), and DTLS (protected channel, DTLS over UDP). To get comparable results, each execution of each implemented approach was performed 10 times. Each time (i.e., process of generation and transmission of data) was performed for 10 minutes. In a case of 15 applications, one middleware, and 15 devices (worst case in our tests), we had around 6000 data packets of 1KB flowing between Logical Devices and Middleware Core, and the same number of data was processed

between Middleware Core and Applications. This amount of data is composed of 400 generated data for each device, which flow for 10 minutes from the Middleware Core to the corresponding application.

We tried to provide security with TLS and DTLS protocols in a non-intrusive way. It means that Middleware Core and Logical Device should interact with each other in an acceptable time regarding the constraint application requirements. We created a scenario to evaluate the performance of the implemented security approaches and divide it into two main goals. Figure 5.9 shows the limits of the scenario in the architecture of the COMPaaS middleware for each goal.

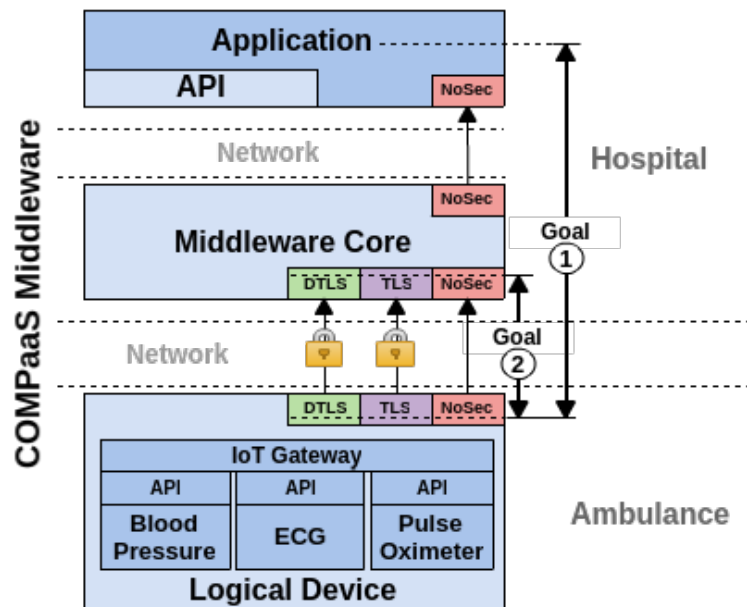


Figure 5.9 – Scenario goals on secure COMPaaS.

This scenario tests the entire flow of information into the middleware system for a variable number of Applications and Logical Devices (i.e., 1, 5, 10 and 15). The goals are twofold:

1. To evaluate if the TLS/DTLS protocols implementations compromise the response time for applications;
2. To evaluate how much overhead is added by TLS and DTLS approaches in this specific channel of COMPaaS;

Table 5.3 – Performance Tests (ms).

Approach/Systems	1/1/1	5/1/5	10/1/10	15/1/15
NoSecTCP	310	450	593	689
TLS	314	456	600	698
NoSecUDP	254	367	482	545
DTLS	257	372	488	552

We measured the time before data leaving from Logical Device until they reach the Application. Table 5.3 presents the results of all tested approaches in milliseconds. The first column

of the table shows the tested approaches while the first row of the table displays the number of systems (i.e., Applications/Middleware Core/Logical Devices) used in the tests. We observed that the main responsible for the elapsed time in each execution of this scenario were both the congestion and processing of data involved in middleware system, and not the addition of the security feature. However, as we are looking for transmitting data in an acceptable response time and in a safe way, all the elapsed time should be considered for analysis in this scenario.

We calculated some statistical data on the results that help to understand the obtained averages. We measured the standard deviation, which is used to quantify the amount of variation or dispersion of a set of data values related to the average. During all the executions the standard deviation ranged between 2 and 7. A low standard deviation indicates that the elapsed times were close to the average. On the other hand, a high standard deviation indicates that the elapsed times were spread out over a wider range of values. In our results, we can consider that the elapsed times were close to the averages.

Regarding Table 5.3, it shows that TLS adds, at least, 4ms (case 1/1/1) when compared to an unsecured TCP connection. In this sense, we observed that DTLS implementation adds, at least, 3ms (case 1/1/1) when compared to an unsecured UDP connection. The biggest overhead was obtained in the case 15/1/15 for both approaches. TLS increased the elapsed time in 9ms when compared to NoSecTCP, while DTLS increased the elapsed time in 7ms when compared to NoSecUDP. This small increase in elapsed time for both approaches is related to the security layer, which is responsible for encoding and decoding each sent data. Figure 5.10 presents the overheads added by the security approaches in a graph way.

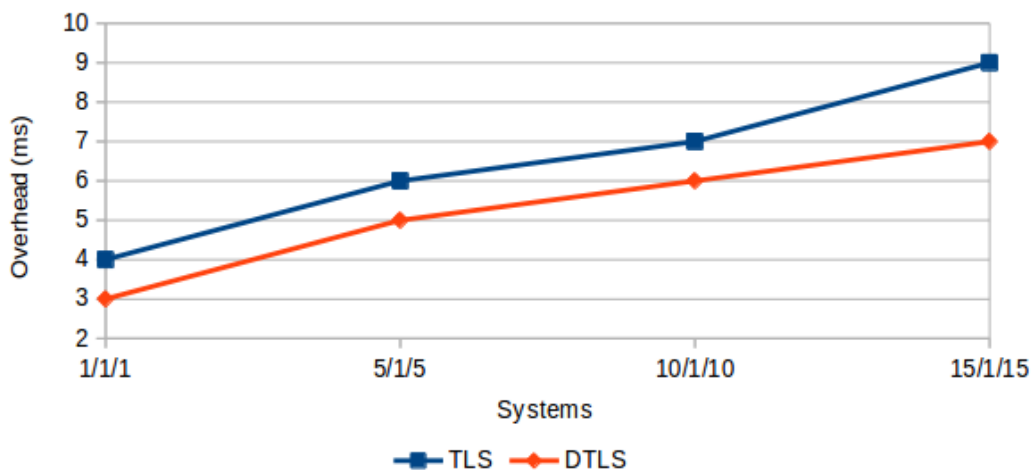


Figure 5.10 – Overhead added by the security approaches.

The additional functions of DTLS approach results in a bigger overhead related to its corresponding non-secure approach when compared to TLS and its corresponding NoSecTCP approach. However, this larger cost is not sufficient to increase the final times of DTLS executions in a meaningful way, since it continues obtaining the lowest times when we look only for runtime.

In fact, the overheads obtained for TLS and DTLS should always be the same for all executions of the respective approach (i.e., TLS or DTLS). However, as we used only one computer to simulate the applications and also only one computer to simulate the Logical Devices and Devices, the final result depends on the processing capacity of the used machines. If we had used 15 computers for simulating the 15 Logical Devices, for example, the parallelism would be the best possible and the overheads added by the TLS and DTLS would be the same for all the respective executions.

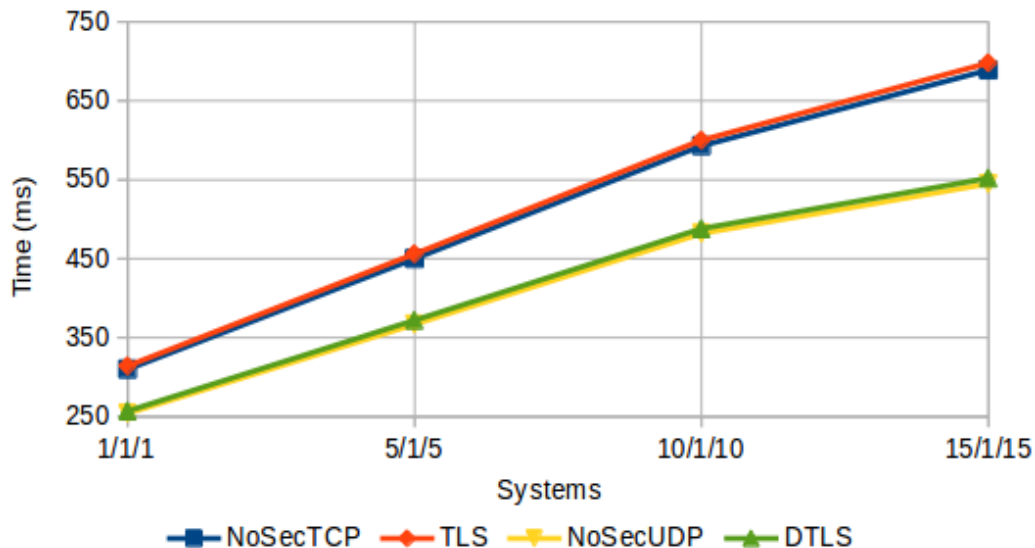


Figure 5.11 – Response time for applications with and without security.

Regarding the response time for applications, Figure 5.11 presents the results of Table 5.3 in a graph way. It illustrates that when we have more devices generating data, the middleware has more data to manage, so it increases the final time. In addition, we can note a considerable difference when compare the elapsed time of TLS and DTLS. We observed that DTLS always had the lowest times, mainly because it is an approach over UDP which has no original feature for reliability. If ordering is required, it has to be managed in the application layer. In addition, this is not a rule, but some features characterize UDP faster than TCP:

- Its header size is 8 bytes (against 20 bytes of TCP);
- It does not have an option for flow control;
- It does error checking, but no recovery options.

As mentioned during this work, the DTLS implemented approach is based on Scandium, which provides some additional functions for ordering and delivery of messages at the application layer. Although these features increase the DTLS time for transmission in some milliseconds, it is not sufficient to exceed the TLS approach in the e-health scenario. This explanation is illustrated in Figure 5.12, where it is possible to notice that the elapsed time for TLS increases 1,3% on average when compared to NoSecTCP. On the other hand, the elapsed time for DTLS increases 1,4% on average when compared to NoSecUDP. These values demonstrate that although DTLS increases the

elapsed time related to its non-secure approach (because of its additional functions), it continues faster than TLS when we look only for elapsed time. Moreover, it is relevant to mention that although there are some divergent results between the implemented approaches, both are still able to ensure the same level of security (i.e., confidentiality and integrity) for the exchanged messages since they used very similar cipher suites.

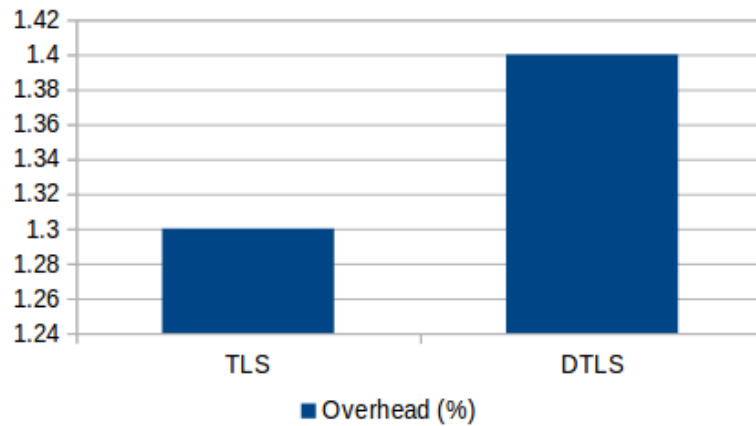


Figure 5.12 – Overhead added when compared to non-secure approaches (%).

5.3.1 DISCUSSION

The tests demonstrated that both implemented protocols can provide security in the lowest layers of the COMPaaS middleware without jeopardizing the system performance. Moreover, when we have used the DTLS approach, we decreased the response time for applications which matches exactly with our main goal regarding try to provide a non-intrusive security approach.

As already mentioned, the DTLS has a larger overhead compared to a non-secure approach than TLS. Despite the additional functions, the cipher suite used by DTLS has some characteristics that point to a slightly slower security than the one used by TLS. Technically, both TLS and DTLS use cipher suites very similar. The only difference is about the module used to ensure authenticated encryption. TLS uses GCM while DTLS uses CCM. We would like to have used GCM in both cases since it enables encryption/decryption of multiple blocks in a parallel way, among other interesting features. However, the Scandium package used to implement the DTLS approach does not provide the GCM module, only the CCM, which despite lacking the parallelization feature and be less used than GCM, is also able to provide authenticated encryption.

When we compare the overhead added by TLS in relation to DTLS, we noticed that it increases the cost significantly than when compared to the NoSecTCP approach. When TLS was used, the time has increased on an average of 24% when compared to DTLS approach. This large difference is related to the TCP protocol, which is used by TLS. TCP has a characteristic to control the data flow. Since our tests simulate a lot of data to be transmitted, it has to control each one.

This makes the elapsed times increase significantly when compared to DTLS, which has not control over the data flow.

Regarding the acceptable response time for Applications, despite DTLS was faster on average compared with TLS approach, both implemented protocols behaved according to the expectations, that was around 1.5 seconds [59]. Both approaches had an acceptable outcome and, depending on the critical response time requirement of the medical systems, can be able to provide data protection with an acceptable response time. This requirement is extremely relevant since it allows the hospital physicians to analyze the current life's condition of a patient, increasing the chance of a correct interpretation in order to save a life. With the results, we can argue that adding this kind of security in an IoT middleware channel for EMS scenarios is not costly once an active session has been established.

Regarding all the tests, DTLS implementation obtained the lowest results when compared to TLS approach. Despite TLS is based on TCP, that is a protocol that in most cases consumes more resources than UDP, the DTLS approach was able to ensure similar characteristics of TCP such as the delivery and ordering of messages. DTLS uses a simple retransmission timer to handle packet loss and also has a few adjustments to the record and handshake header to handle reordering. In fact, the choice for TLS or DTLS should be made based on the constraint requirements of the addressed environment. In this case, the optimal choice would be TLS since TCP provides reliability as a standard. On the other hand, DTLS, although faster in messages exchange and with the same level of security than TLS, does not guarantee reliability as a standard. Indeed, to ensure ordering and delivering of exchanged messages, DTLS needs some adaptations regarding the native UDP protocol. Nevertheless, this adaptation only ensures a solid reliability of the handshake's messages, where it knows exactly the expected message. In an exchange of post-handshake data, an alternative is to use the sequence number to see if the messages are coming in an orderly way and also send an ACK message from the receiver side to warn that the message was received. Figure 5.13 shows a data message sent by Logical Device (Client) and then an ACK sent as "Application Data" by the Middleware Core (Server). This is the way the Scandium library implements the DTLS in CoAP [22] as a means to ensure the reliability of data exchanged on the network. According to [37], to allow receivers to transmit an ACK value, which indicates that they have received the message, is a good approach since it allows timers to be set lower as well as reducing the number of packets that have to be retransmitted (since the sender would know that something had already been received).

No.	Time	Source	Destination	Protocol	Length	Info
67	40.959109000	Client	Server	DTLSv1.2	1087	Application Data
68	40.966454000	Server	Client	DTLSv1.2	74	Application Data
71	42.452661000	Client	Server	DTLSv1.2	1087	Application Data
72	42.462752000	Server	Client	DTLSv1.2	74	Application Data

◀Data Message
◀ACK Message

Figure 5.13 – Example of a data message and its respective ACK response.

In the described scenario an appropriate solution indicates the union of TLS reliability with DTLS performance. However, both implemented approaches can be used in order to ensure security and reliability of data. In addition, all data transmitted from the simulated devices of the ambulance

can be analyzed in an acceptable time, which was less than one second for each data, ensuring the physicians' interpretation on the real life condition of a patient.

Regarding the choice of the network environment used, we conducted our tests on a local network. However, in a real scenario we probably would have used a mobile network 3G/4G. In this sense, despite knowing that this kind of network will probably increase the elapsed time for data transmission, we understand that it is not an affect of the security feature since the obtained results point for low percentages when we look for security overhead. In this way, the mere addition of the security layer does not mean a significant impact on the scenario. Anyway, we intend to improve our validation with new tests in the future.

To understand why we chose to implement the four approaches (i.e., NoSecTCP, TLS, NoSecUDP, and DTLS) is also important. We know that DTLS is based on TLS, and they have almost the same characteristics, including very similar handshakes. The choice of these protocols was related to the response time requirement demanded by the application environment. Moreover, we compared TLS and DTLS approaches because both can provide security and reliability for transmitted data (i.e., TLS as a standard of TCP and DTLS through an adaptation at the application layer provide by Scandium package), which is essential for the e-health scenario. Although the NoSecUDP approach does not provide the reliability feature, we used it as a way to facilitate the comparison with the DTLS approach, since it runs over UDP. Regarding the NoSecTCP approach, it was already used by COMPaaS before the implementation of the security approaches. In addition, it facilitates the comparison with the TLS approach.

6. FINAL CONSIDERATIONS

In this chapter we present the final considerations for the main issues addressed in this work. Section 6.1 presents a relationship between the implemented approaches and the attacks described in Section 3.1. Section 6.2 presents the considerations about the importance of having security services for SOA-based IoT middleware. Section 6.3 presents an analysis related to non-intrusive approaches in IoT environments. Section 6.4 gives an overview about our next steps, while Section 6.5 concludes this work.

6.1 ATTACK ANALYSIS

In Chapter 3 we introduced six potential threats that must be considered when trying to provide security for an SOA-based IoT middleware system. Since TLS and DTLS are security protocols that operate at the application layer, two threats cannot be handled solely by TLS/DTLS and remain a potential risk: device cloning and extraction of security parameters. In the following items we made conclusions about the remaining four threats [18] [22].

- *Firmware Replacement*: A device secured exclusively by TLS/DTLS is not protected against firmware replacement attacks. If an attacker can exploit a firmware upgrade to influence the operational behavior of a device, there is no use for encrypted communication and authentication since the device has been compromised at a higher level rendering this endeavor useless.
- *Eavesdropping*: Both TLS and DTLS provide full protection against eavesdropping. This is true for our implementations as long as the Diffie-Hellman assumption holds. An eavesdropper who does not know the private Diffie-Hellman values has no option to derive a premaster secret¹⁴ and, therefore, is unable to decrypt the protected application data.
- *Man-in-the-Middle*: Whenever the server is successfully authenticated, the channel is secure against man-in-the-middle attacks. Only completely anonymous sessions are inherently vulnerable to such attacks, but we do not negotiate cipher suites that support anonymous sessions. If the server is authenticated using a public key infrastructure, an attacker has no possibility to forge the server's ServerKeyExchange message which is signed with the appropriate private key.
- *Routing Attacks*: Routing attacks take place at the network layer where an attacker can manipulate routes and the traffic over certain nodes. Both TLS and DTLS have no control over the path the packets take in the network. So, both are vulnerable to certain routing

¹⁴In our case it is a shared secret between two communicating systems generated during the handshake protocol (e.g., using Diffie-Hellman).

attacks like sinkhole and selective forwarding attacks, but are secure against spoofing or altering of messages.

As we have noticed before, TLS/DTLS alone will not prevent node capture attacks where an attacker captures a physically unprotected thing and tries to extract security parameters. So, it is important to analyze the worst thing that could happen to a node secured with the TLS/DTLS implementation exposed to such an attack.

Regarding certificates and private keys, they must be stored on the node to be easy accessible by TLS/DTLS. For our implementations we used a password-protected Java keystore. Even if the certificates and keys are password-protected, preventing an intruder from extracting those is hard since the password itself must be stored on the node itself. According to [22], if the flash memory of a node can be accessed, the password and therefore the certificates and keys can be extracted, so the only protection against such attacks is some sort of certificate revocation list to invalidate the certificate once it has been compromised.

Another important target for attacks regarding stored sensitive data is related to sessions. Each time a handshake was successful, the node stores a session containing two sensitive values: the Master Secret and the session identifier (see APPENDIX B and APPENDIX C). If an attacker obtained these values it is able to start future secured connections and impersonate this node using these two values in a resuming handshake. It is a potential security risk that should be examined further in the future.

Use ephemeral keys as in ECDH key exchange guarantees PFS (as long as the Master Secret remains secured). Thus, for a more complex and time-consuming handshake, we gain this security feature.

6.2 SECURITY SERVICES

We addressed the protection of SOA-based IoT middleware during all this work. This is an open challenge of the IoT ecosystem and encompasses some important procedures including key management, establishing access control policies together with authentication to allow access to networks, services and data, usage of security services to protect data against potential attacks, and the development and selection of efficient cryptographic mechanisms. Custom security approaches offered by the IoT research community mostly offer specific improvements or solutions. However, they rarely help to understand the big picture for securing an SOA-based IoT middleware system.

A security architecture that provides a full stack of security services composed of authentication, authorization, integrity, confidentiality and communication channel protection should be defined [55]. Many security protocols have already been standardized and adapting them to be used in different IoT environments would be beneficial for the architecture definition.

The importance of having a security architecture is strongly related to the definition of the correct set of security services. The services proposed in Chapter 4 could compose an SOA-based

security architecture and would be useful for ensuring protection against some powerful attacks such as eavesdropping and man-in-the-middle. A security architecture is intended to ensure protection for an entire system providing a standard that could be used by other middleware systems, since none of the security mechanisms mentioned in this work can guarantee, by themselves, an adequate protection for a whole system with different application requirements.

The use of the security standards protocols described in Sections 3.2 and 3.4 could be the basis for the definition of a security architecture. For example, TLS and DTLS protocols can provide protection for the communication channel, and CoAP, an important IoT communication protocol, can provide interoperability for systems and security through DTLS.

Although the definition of a security architecture for SOA-based IoT middleware is noteworthy, it is mandatory to be careful with the way in which the involved protocols can be deployed, since some IoT devices may not have sufficient computing resources to perform complex security mechanisms. A significant challenge related to IoT middleware is the necessity of having non-intrusive security solutions [21] [46]. Provide solutions such as key management, authentication, access control, confidentiality, and integrity is considered a significant challenge mainly when applied in resource-constrained environments like the physical infrastructure of devices of the IoT.

6.3 NON-INTRUSIVE SECURITY PROTOCOLS IN IOT ENVIRONMENTS

In this current age of information and communication technology, is natural or even mandatory to save information about all our activities [48]. In this way, meaningful information is important as any other valuable asset, especially when the organizations need to secure their data. Apart from the field of application in IoT middleware systems, security issues are inherently important for the whole systems, from data protection to device security, and can be divided into different requirements and scenarios. In e-health, for example, we demonstrated that acceptable response time and data protection requirements are more emphasized. However, these requirements will only be reached with the implementation of the correct security protocol.

Non-intrusive security protocols are important in all IoT environments. However, they are highlighted in e-health domain since it deals with people's health. IoT devices are generally embedded with low-speed processors. Besides, these devices are not designed to perform computationally expensive operations, and they just act as a sensor or actuator. Therefore, finding a security solution that minimizes resource consumption and thus improves security performance is a challenging task [48].

Chapters 4 and 5 demonstrated that TLS and DTLS approaches are viable choices to provide confidentiality, integrity, authenticity and reliability in IoT middleware applied to e-health scenarios. The implementation of the CCP service is a first step thinking in a security architecture that is our primary goal for the next years. However, although these protocols are used in resource-constrained environments, they are not entirely optimal for this since they were designed

for traditional computer networks [22]. For example, DTLS needs large buffers to handle message loss by retransmitting the last flight of the handshake protocol or by storing all the fragments when fragmentation must be applied due to the smaller maximum transmission unit. In other words, we concluded that there are some other significant challenges to overcome. According to [29], IoT nodes often have constraints regarding their resources such as computational power, memory size and power management. Network communication, especially wireless, also imposes additional restrictions such as low bit rates, variable delays, and high packet losses.

6.4 FUTURE IMPLEMENTATIONS

As already mentioned, some features can be improved mainly in the DTLS protocol in order to allow its application in environments with more specific resource-constrained requirements. In the future we intend to improve our DTLS implementation as a mean to enable its use in other resource-constrained environments (e.g., hardware with limited processing capacity, battery lifetime) beyond that applied in this work. We also intend to improve our tests and perform experiments in mobile networks (e.g., 3G, 4G) and with real devices generating data.

Furthermore, another important goal for the future is to implement the security services in other parts of the COMPaaS middleware as a mean to enable the provision of the necessary security requirements to protect applications, stored and exchanged data, and entities. We intend to provide a complete security architecture that can be based on security services such as those mentioned in Chapter 4, which are ADA, AAC, DCI, and CCP, always focusing on a non-intrusive mode. Next step is the technical definition and implementation of these services, which will collaborate not only with middleware systems but also with the whole IoT paradigm.

6.5 CONCLUSION

The objective of this work was to define security services that could be used in SOA-based IoT middleware systems to protect data. We call these services as ADA, AAC, DCI, and CCP. We also gave an overview of some threats and outlined the requirements to ensure a secure communication. As a first step towards the definition of the security services, we implemented the CCP service to provide security in the main scenarios of e-health, which bring critical requirements as acceptable response time and security of data transmitted over open networks. In an EMS scenario, for example, sensitive data is trafficked out of the hospitals networks and need protection mainly against eavesdropping and man-in-the-middle attacks. We have implemented the CCP service on COMPaaS middleware to protect it and also to allow the validation of the proposed security approaches.

The proposed CCP service was composed of two approaches: TLS and DTLS. Both protocols ensure confidentiality, integrity, and authenticity of messages exchanged over the network.

We have used JSSE and Scandium as libraries to implement the security approaches. Regarding the tests, we verified the functionality of the implementations in a successful way. In this sense, the performance tests revealed that the use of DTLS was always faster than TLS. Once a session has been established between Middleware Core and Logical Device, the transmission time for TLS messages was 24% bigger on average when compared to DTLS messages.

Regarding attack analysis, the implemented approaches secured the connection between Middleware Core and Logical Device during the communication phases. It was possible by providing confidentiality, integrity, and authenticity, and, therefore, preventing eavesdropping and man-in-the-middle attacks. Protection against physical attacks, such as device cloning and extraction of security parameters, remains unresolved for these approaches.

At the end of this document we present some important appendices. APPENDIX A presents the publications obtained during the development of this work. APPENDIX B presents the DTLS handshake, while APPENDIX C presents the TLS handshake. Finally, APPENDIX D gives a brief description of how to generate keys.

BIBLIOGRAPHY

- [1] Aberer, K.; Hauswirth, M.; Salehi, A. "Infrastructure for data processing in large-scale interconnected sensor networks". In: International Conference on Mobile Data Management, 2007, pp. 198–205.
- [2] Addo, I. D.; Ahamed, S. I.; Yau, S. S.; Buduru, A. "A reference architecture for improving security and privacy in Internet of Things applications". In: IEEE International Conference on Mobile Services, 2014, pp. 108–115.
- [3] Amaral, L. A.; Matos, E.; Tiburski, R. T.; Hessel, F.; Lunardi, W. T.; Marczak, S. "Internet of Things (IoT) in 5G Mobile Technologies". Cham: Springer International Publishing, 2016, chap. Middleware Technology for IoT Systems: Challenges and Perspectives Toward 5G, pp. 333–367.
- [4] Amaral, L. A.; Tiburski, R. T.; de Matos, E.; Hessel, F. "Cooperative middleware platform as a service for Internet of Things applications". In: 30th Annual ACM Symposium on Applied Computing, 2015, pp. 488–493.
- [5] Antonio, S. "Initial Architecture Specification", Technical Report, SMARTIE project, 2015, 91p.
- [6] Atzori, L.; Iera, A.; Morabito, G. "The Internet of Things: A survey", *Computer Networks*, vol. 54–15, 2010, pp. 2787 – 2805.
- [7] Badii, A.; Khan, J.; Crouch, M.; Zickau, S. "Hydra: Networked embedded system middleware for heterogeneous physical devices in a distributed architecture". In: Final External Developers Workshops Teaching Materials, 2010, pp. 4.
- [8] Bandyopadhyay, S.; Sengupta, M.; Maiti, S.; Dutta, S. "A Survey of Middleware for Internet of Things". Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, chap. 27, pp. 288–296.
- [9] Blake-Wilson, S.; Moeller, B.; Gupta, V.; Hawk, C.; Bolyard, N. "Elliptic curve cryptography (ECC) cipher suites for transport layer security (TLS)". Capturado em: <https://tools.ietf.org/html/rfc4492>, May 2006.
- [10] Bohn, H.; Bobek, A.; Golasowski, F. "SIRENA - service infrastructure for real-time embedded networked devices: A service-oriented framework for different domains". In: International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies, 2006, pp. 43–43.
- [11] Caporuscio, M.; Raverdy, P.-G.; Issarny, V. "ubiSOAP: A Service-Oriented Middleware for Ubiquitous Networking", *IEEE Transactions on Services Computing*, vol. 5–1, Jan 2012, pp. 86–98.

- [12] Chaqfeh, M.; Mohamed, N. "Challenges in middleware solutions for the Internet of Things". In: International Conference on Collaboration Technologies and Systems (CTS), 2012, pp. 21–26.
- [13] Conzon, D.; Bolognesi, T.; Brizzi, P.; Lotito, A.; Tomasi, R.; Spirito, M. "The VIRTUS middleware: An XMPP based architecture for secure IoT communications". In: 21st International Conference on Computer Communications and Networks (ICCCN), 2012, pp. 1–6.
- [14] Cooper, D. "Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile". Capturado em: <https://tools.ietf.org/html/rfc5280>, May 2008.
- [15] Dierks, T. "The transport layer security (TLS) protocol version 1.2". Capturado em: <https://tools.ietf.org/html/rfc5246>, Aug 2008.
- [16] Ferreira, H.; Sousa, R.; Gomes, F.; Canedo, E. "Proposal of a secure, deployable and transparent middleware for Internet of Things". In: 9th Iberian Conference on Information Systems and Technologies (CISTI), 2014, pp. 1–4.
- [17] Fremantle, P.; Scott, P. "A security survey of middleware for the Internet of Things", *PeerJ PrePrints*, vol. 3, 2015, pp. e1521.
- [18] Garcia-Morchon, O.; Kumar, S.; Struik, R.; Keoh, S.; Hummen, R. "Security considerations in the IP-based Internet of Things". Capturado em: <https://tools.ietf.org/html/draft-garcia-core-security-04>, Mar 2012.
- [19] Huston, G.; Michaelson, G.; Kent, S. "Certification authority (CA) key rollover in the resource public key infrastructure (RPKI)". Capturado em: <https://tools.ietf.org/html/rfc6489.html>, Feb 2012.
- [20] Jing, Q.; Vasilakos, A.; Wan, J.; Lu, J.; Qiu, D. "Security of the Internet of Things: perspectives and challenges", *Wireless Networks*, vol. 20–8, 2014, pp. 2481–2501.
- [21] Jucker, S. "Securing the constrained application protocol", Master's Thesis, Department of Computer Science, ETH Zurich, Zurich, Switzerland, 2012, 103p.
- [22] Keoh, S. L.; Kumar, S.; Tschofenig, H. "Securing the Internet of Things: A standardization perspective", *IEEE Internet of Things Journal*, vol. 1–3, June 2014, pp. 265–275.
- [23] Kim, M.; Lee, J. W.; Lee, Y. J.; Ryou, J.-C. "COSMOS: A middleware for integrated data processing over heterogeneous sensor networks", *ETRI Journal*, vol. 30–5, 2008, pp. 696–706.
- [24] Kivinen, T. "Using raw public keys in transport layer security (TLS) and datagram transport layer security (DTLS)". Capturado em: <https://tools.ietf.org/html/rfc7250>, Jun 2014.

- [25] Kobusingye, O. C.; Hyder, A. A.; Bishai, D.; Joshipura, M.; Hicks, E. R.; Mock, C. "Emergency medical services". In: *Disease control priorities in developing countries*, Jamison D., Breman J., M. A. A. G. C. M. E. D. (Editor), Washington (DC): World Bank, 2006, chap. 68, pp. 1261–1279.
- [26] Kothmayr, T.; Schmitt, C.; Hu, W.; Brunig, M.; Carle, G. "A DTLS based end-to-end security architecture for the Internet of Things with two-way authentication". In: IEEE 37th Conference on Local Computer Networks Workshops (LCN Workshops), 2012, pp. 956–963.
- [27] Lake, D.; Milito, R.; Morrow, M.; Vangheese, R. "Internet of Things: Architectural framework for eHealth security", *Journal of ICT*, vol. 3, 2014, pp. 301–330.
- [28] Lakkundi, V.; Singh, K. "Lightweight DTLS implementation in CoAP-based Internet of Things". In: 20th Annual International Conference on Advanced Computing and Communications (ADCOM), 2014, pp. 7–11.
- [29] Leusse, P.; Periorellis, P.; Dimitrakos, T.; Nair, S. "Self Managed Security Cell, a Security Model for the Internet of Things and Services". In: First International Conference on Advances in Future Internet, 2009, pp. 47–52.
- [30] Li, Y.; Du, L. P.; Guo, J.; Zhao, X. "Research on lightweight information security system of the Internet of Things", *Research Journal of Applied Sciences, Engineering and Technology*, vol. 5–19, 2013, pp. 4722–4726.
- [31] Linksmart. "Eulinksmartsecuritycommunicationsecuritymanagersym - linksmart open source middleware - linksmart middleware portal". Capturado em: <https://linksmart.eu/redmine/projects/linksmart-opensource/wiki/Eulinksmartsecuritycommunicationsecuritymanagersym>, Feb 2015.
- [32] Locke, D. "MQTT v3.1 Protocol Specification", Technical Report, International Business Machines Corporation (IBM) and Eurotech, 2010, 42p.
- [33] Lunardi, W.; Matos, E.; Tiburski, R.; Amaral, L.; Marczak, S.; Hessel, F. "Context-based search engine for industrial IoT: Discovery, search, selection, and usage of devices". In: IEEE Conference on Emerging Technology & Factory Automation (ETFA), 2015, pp. 1–8.
- [34] Matos, E.; Amaral, L.; Tiburski, R.; Lunardi, W.; Hessel, F.; Marczak, S. "Context-aware system for information services provision in the internet of things". In: IEEE Conference on Emerging Technologies & Factory Automation, 2015, pp. 1–4.
- [35] Matos, E.; Lunardi, W.; Amaral, L.; Tiburski, R.; Hessel, F.; Marczak, S. "Context-based Framework to Discovery, Search, and Selection of Computing Devices in the Internet of Things". In: CSBC 2015 - SEMISH, 2015, pp. 1–6.

- [36] Modadugu, N.; Rescorla, E. "The design and implementation of Datagram TLS". In: NDSS Symposium, 2004, pp. 1–13.
- [37] Nadalin, A.; Kaler, C.; Monzillo, R.; Hallam-Baker, P. "Web services security: SOAP message security 1.1 (WS-Security 2004)", *Oasis Standard*, vol. 200401, 2004, pp. 1–20010502.
- [38] Nordbotten, N. "XML and Web Services security standards", *IEEE Communications Surveys Tutorials*, vol. 11–3, rd 2009, pp. 4–21.
- [39] oneM2M. "Analysis of Security Solutions for the oneM2M System", Technical Report, ETSI, 2014, 49p.
- [40] Open Mobile Alliance (OMA). "M2M Enablers". Capturado em: <http://openmobilealliance.org/about-oma/work-program/m2m-enablers>, Jan 2016.
- [41] Organization, W. H. "Medical device regulations: Global overview and guiding principles", Technical Report, World Health Organization, Geneva, 2003, 54p.
- [42] Organization, W. H. "Blood pressure monitor". Capturado em: http://www.who.int/medical_devices/innovation/blood_pressure_monitor.pdf, Nov 2011.
- [43] Organization, W. H. "Electrocardiograph (ECG)". Capturado em: http://www.who.int/medical_devices/innovation/electrocardiograph.pdf, Nov 2011.
- [44] Organization, W. H. "Pulse oximeters". Capturado em: http://www.who.int/medical_devices/innovation/hospt equip_21.pdf, Mar 2012.
- [45] Perera, C.; Jayaraman, P. P.; Zaslavsky, A.; Christen, P.; Georgakopoulos, D. "MOSDEN: An Internet of Things middleware for resource constrained mobile devices". In: 47th Hawaii International Conference on System Sciences, 2014, pp. 1053–1062.
- [46] Polk, T.; McKay, K.; Chokhani, S. "Guidelines for the selection, configuration, and use of transport layer security (TLS) implementations", *NIST Special Publication*, vol. 800, 2014, pp. 52.
- [47] Raza, S.; Shafagh, H.; Hewage, K.; Hummen, R.; Voigt, T. "Lite: Lightweight secure CoAP for the Internet of Things", *IEEE Sensors Journal*, vol. 13–10, Oct 2013, pp. 3711–3720.
- [48] Rescorla, E.; Modadugu, N. "Datagram transport layer security version 1.2". Capturado em: <https://tools.ietf.org/html/rfc6347>, Jan 2012.
- [49] Riazul Islam, S.; Kwak, D.; Humaun Kabir, M.; Hossain, M.; Kwak, K.-S. "The Internet of Things for Health Care: A Comprehensive Survey", *IEEE Access*, vol. 3, 2015, pp. 678–708.
- [50] Saint-Andre, P. "Extensible messaging and presence protocol (XMPP): Core". Capturado em: <https://tools.ietf.org/html/rfc6120>, Mar 2011.

- [51] Shelby, Z.; Hartke, K.; Bormann, C. "The constrained application protocol (CoAP)". Capturado em: <https://tools.ietf.org/html/rfc7252>, Jun 2014.
- [52] Sicari, S.; Rizzardi, A.; Grieco, L.; Coen-Porisini, A. "Security, privacy and trust in Internet of Things: The road ahead", *Computer Networks*, vol. 76–0, 2015, pp. 146–164.
- [53] Spiess, P.; Karnouskos, S.; Guinard, D.; Savio, D.; Baecker, O.; Souza, L.; Trifa, V. "SOA-based integration of the Internet of Things in enterprise services". In: IEEE International Conference on Web Services, 2009, pp. 968–975.
- [54] Swetina, J.; Lu, G.; Jacobs, P.; Ennesser, F.; Song, J. "Toward a standardized common M2M service layer platform: Introduction to oneM2M", *IEEE Wireless Communications*, vol. 21–3, June 2014, pp. 20–26.
- [55] Tiburski, R.; Amaral, L.; Matos, E.; Hessel, F. "The importance of a standard security architecture for SOA-based IoT middleware", *IEEE Communications Magazine*, vol. 53–12, Dec 2015, pp. 20–26.
- [56] Tiburski, R. T.; Amaral, L. A.; Hessel, F. "Internet of Things (IoT) in 5G Mobile Technologies". Cham: Springer International Publishing, 2016, chap. Security Challenges in 5G-Based IoT Middleware Systems, pp. 399–418.
- [57] Uckelmann, D.; Harrison, M.; Michahelles, F. "Architecting the Internet of Things". Springer-Verlag Berlin Heidelberg, 2011, 353p.
- [58] Valente, B.; Martins, F. "A middleware framework for the Internet of Things". In: Third International Conference on Advances in Future Internet, 2011, pp. 139–144.
- [59] Vilaplana, J.; Solsona, F.; Abella, F.; Filgueira, R.; Rius, J. "The cloud paradigm applied to e-health", *BMC medical informatics and decision making*, vol. 13–1, 2013, pp. 35.
- [60] Vučinić, M.; Tourancheau, B.; Rousseau, F.; Duda, A.; Damon, L.; Guizzetti, R. "OSCAR: Object security architecture for the Internet of Things", *Ad Hoc Networks*, vol. 32, 2015, pp. 3 – 16.
- [61] Wu, Z.-Q.; Zhou, Y.-W.; Ma, J.-F. "A security transmission model for Internet of Things", *Chinese Journal of Computers*, vol. 34–8, 2011, pp. 1351–1364.
- [62] Zhang, L.; Wang, Z. "Integration of RFID into wireless sensor networks: Architectures, opportunities and challenging problems". In: Fifth International Conference on Grid and Cooperative Computing Workshops, 2006, pp. 463–469.
- [63] Zhang, W.; Qu, B. "Security architecture of the Internet of Things oriented to perceptual layer", *International Journal on Computer, Consumer and Control (IJ3C)*, vol. 2–2, 2013, pp. 37–45.

APPENDIX A – PUBLICATIONS

This appendix addresses the scientific production (or publications) I have been involved during my master degree. Five papers and two book chapters were written. I am the principal author in two of them, a magazine and a book chapter.

Next items present all the papers and a brief description of my role in which one. The papers are presented in order of importance for my work.

1. **The Importance of a Standard Security Architecture for SOA-based IoT Middleware [55], in *IEEE Communications Magazine*, 2015:**

This is my main paper. It discusses the importance of a security architecture for SOA-based IoT middleware systems like COMPaaS and proposes the use of the security services defined in Chapter 4.

2. **Cooperative Middleware Platform as a Service for Internet of Things Applications [4], in *30th Annual ACM Symposium on Applied Computing (SAC 15)*, 2015:**

This is the COMPaaS definition paper. I was responsible for writing the technical part and also for writing and performing the tests that validate the system.

3. **Security Challenges in 5G-based IoT Middleware Systems [56], in *Springer International Publishing (Internet of Things (IoT) in 5G Mobile Technologies)*, 2016:**

This is an important work in which I am the main author. It is a book chapter that presents the security challenges of IoT middleware systems towards 5G.

4. **Middleware Technology for IoT Systems: Challenges and Perspectives Toward 5G [3], in *Springer International Publishing (Internet of Things (IoT) in 5G Mobile Technologies)*, 2016:**

This is another book chapter in which I strongly contributed, mainly in the middleware specification, elaboration, and discussion related to the requirements towards 5G.

5. **Context-based Search Engine for Industrial IoT: Discovery, Search, Selection, and Usage of Devices [34], in *Emerging Technology and Factory Automation (ETFA 2015)*, 2015:**

I was responsible for writing the text related to IoT middleware.

6. **Context-Aware System for Information Services Provision in the Internet of Things [35], in *Emerging Technology and Factory Automation (ETFA 2015)*, 2015:**

I was responsible for writing the text related to IoT middleware.

7. **Context-based Framework to Discovery, Search, and Selection of Computing Devices in the Internet of Things [36], in *CSBC 2015 - SEMISH*, 2015:**

I was responsible for writing the text related to IoT middleware.

APPENDIX B – DTLS HANDSHAKE PROTOCOL

This appendix gives a detailed view of the DTLS handshake protocol. It is strongly based on APPENDIX A of work in [22]. Next we describe a full and an abbreviated handshake.

Figure APPENDIX B.1 illustrates the full DTLS handshake protocol containing all possible handshake messages for ECDH Key Exchange with mutual authentication. The following items provide explanations for each message sent. The handshake happens in 6 flights:

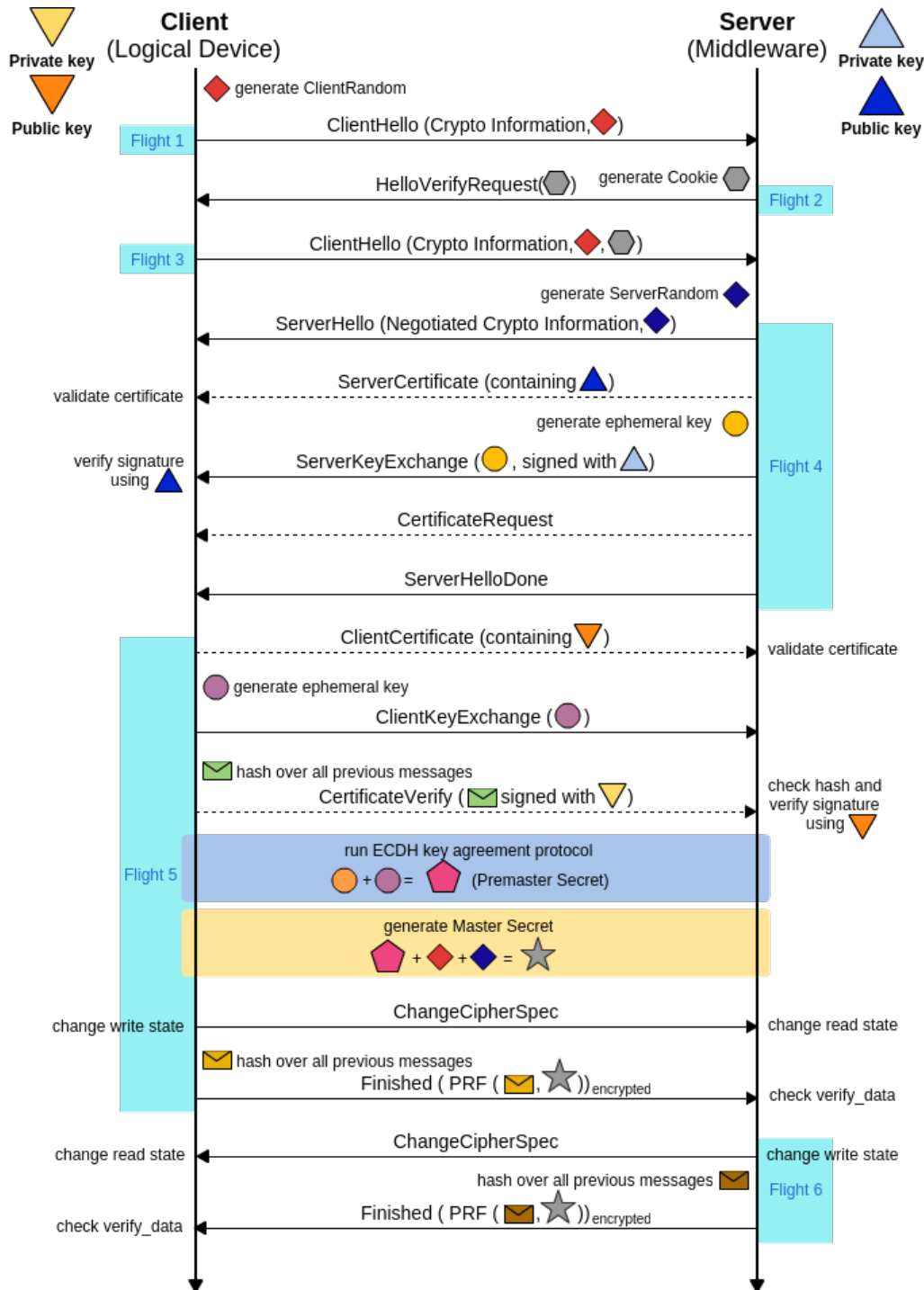


Figure APPENDIX B.1 – DTLS Full Handshake (adapted from [22]).

Flight 1:**ClientHello (1)**

The ClientHello message initiates a full handshake with the server. The client provides the security parameters it wants to use in the handshake (i.e., DTLS version, preferred cipher suite, etc.). Additionally, it provides a 32-byte random value which is used in a later step of the protocol to compute the premaster secret. In the first version of this message, the cookie field will be empty.

Flight 2:**HelloVerifyRequest**

The HelloVerifyRequest has been added by the DTLS specification as a DoS countermeasure [22]. It contains a stateless cookie which is sent back to the client, forcing it to be able to receive the cookie which makes DoS attacks with forged addresses hard.

Flight 3:**ClientHello (2)**

This ClientHello is identical to the first one sent, except for the added cookie taken from the HelloVerifyRequest.

Flight 4:**ServerHello**

The ServerHello message is used by the server to notify the Logical Device about the negotiated security parameters under which the handshake will be executed (i.e., DTLS version, cipher suite, etc.). Additionally, the server provides a 32-byte random value, also used later for calculating the premaster secret.

ServerCertificate

This message contains the server's certificate chain ultimately signed by a certificate authority. The first certificate in this chain contains the server's static ECDH-capable public key and is signed with the ECDSA [22]. Upon receiving this message, the client validates the certificate chain and extracts the server's public key which will be used to verify the ServerKeyExchange message.

ServerKeyExchange

The ServerKeyExchange message is used to convey the server's ephemeral ECDH public key to the client and the used elliptic curve domain parameters. The message is signed with the private key corresponding to the public key sent in the Server-Certificate message. Upon receipt, the client verifies the signature and extracts the server's ephemeral public key and the elliptic curve domain parameters.

CertificateRequest

This message is sent whenever a non-anonymous server requires the client to authenticate itself as well. It specifies a list of certificate types that the client may offer, a list of hash/signature algorithms the server can verify and list of acceptable CAs. Only when this message is sent by the server, mutual authentication can be achieved.

ServerHelloDone

The ServerHelloDone message indicates that the server is done sending messages to support the key exchange. It is needed because otherwise the client cannot be sure if the server's flight is finished since the CertificateRequest message is not mandatory to be sent by the server.

Flight 5:**ClientCertificate**

If required by the server, the client needs to send its certificate chain to the server to convey its static public key. The server validates the certificate chain and extracts the client's public key.

ClientKeyExchange

This message is sent in all key exchange algorithms. In this mode (ECDHE ECDSA) it contains the client's ephemeral ECDH public key which it created corresponding to the parameter it received from the server in the ServerKeyExchange message. Upon receipt, the server extracts the client's ephemeral key and, after having exchanged their public key of the ECDH key pair, the client and the server can run the ECDH key agreement protocol [22] to obtain the premaster secret.

CertificateVerify

This message is only sent following a ClientCertificate that has signing capability. The meaning for it is to prove that the client really possesses the private key corresponding to the public key in the ClientCertificate message. The client computes the hash over all handshake messages sent or received so far and computes the signature of it using its private key corresponding to the public key in the ClientCertificate. The server verifies the signature using the client's public key received in the ClientCertificate message.

ChangeCipherSpec (Client)

After having computed the Master Secret¹⁵ the client signals that each message received after this one will be secured with the negotiated security parameters.

Finished (Client)

The Finished message is the first one protected with the newly negotiated security parameters. The client hashes all previous sent or received handshake messages (excluding

¹⁵It is a 48-byte secret shared between the two systems in the connection. It is generated using the premaster secret, a 32-byte random value provided by each system and a PRF, which takes as input a secret, a seed, and an identifying label and produces an output of arbitrary length.

the first ClientHello, HelloVerifyRequest and ChangeCipherSpec) and generates the verify data by applying it to the Pseudorandom Function (PRF). The server validates the client's verify data.

Flight 6:

ChangeCipherSpec (Server)

After having the client's Finished message validated, the server sends its ChangeCipherSpec with the same semantics as the client's. Therefore, the write pending state is copied to the current write state.

Finished (Server)

The server's Finished message contains the verify data which is computed the same way as before, but the hash of the handshake messages also contains the client's Finished message. This is the server's first message protected with the negotiated security parameters, as the message directly follows its ChangeCipherSpec message. Once a side has sent its Finished message and received and validated the system's Finished message, it may begin to send and receive application data over the connection.

An abbreviated handshake can be executed when the client and server decide to resume a previous session [47]. It can only be executed when the two systems have successfully finished a full handshake in the past, thus, having stored a session containing the session identifier and the Master Secret generated in the full handshake.

The exchanged messages for this scenario are illustrated in Figure APPENDIX B.2. We give a description in next items:

1. The client sends a ClientHello using the session identifier of the session to be resumed.
2. The server checks its session cache for a match. If it is found, and the server is willing to re-establish the connection under the specified session state, it sends a ServerHello with the same session identifier value. The fresh keys are generated from the existing Master Secret and the newly exchanged random values.
3. After the ChangeCipherSpec messages, the connection will be secured with the fresh keys and the security parameters negotiated during the full handshake. If the server does not find a match in the session cache, it generates a new session identifier and initiates a full handshake.

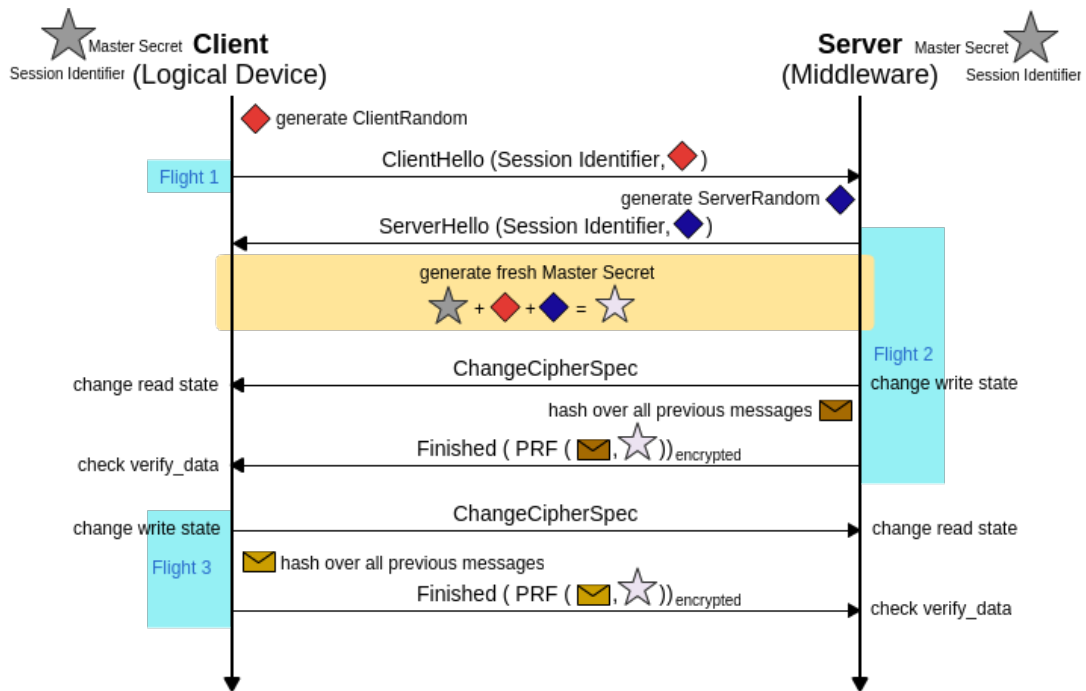


Figure APPENDIX B.2 – DTLS Abbreviated Handshake (adapted from [22]).

For further details, please refer to the specifications of ECC [9], TLS [15], DTLS [47] and [22].

APPENDIX C – TLS HANDSHAKE PROTOCOL

This appendix presents the exchanged messages during the TLS handshake protocol. It is based on TLS specification [15]. The only difference related to the DTLS handshake protocol is that TLS does not provide the HelloVerifyRequest, exchanged in Flight 2 (see Figure APPENDIX B.1). Once they are very similar, we give an overview of the exchanged messages, but we do not explain each one in detail as in APPENDIX B.

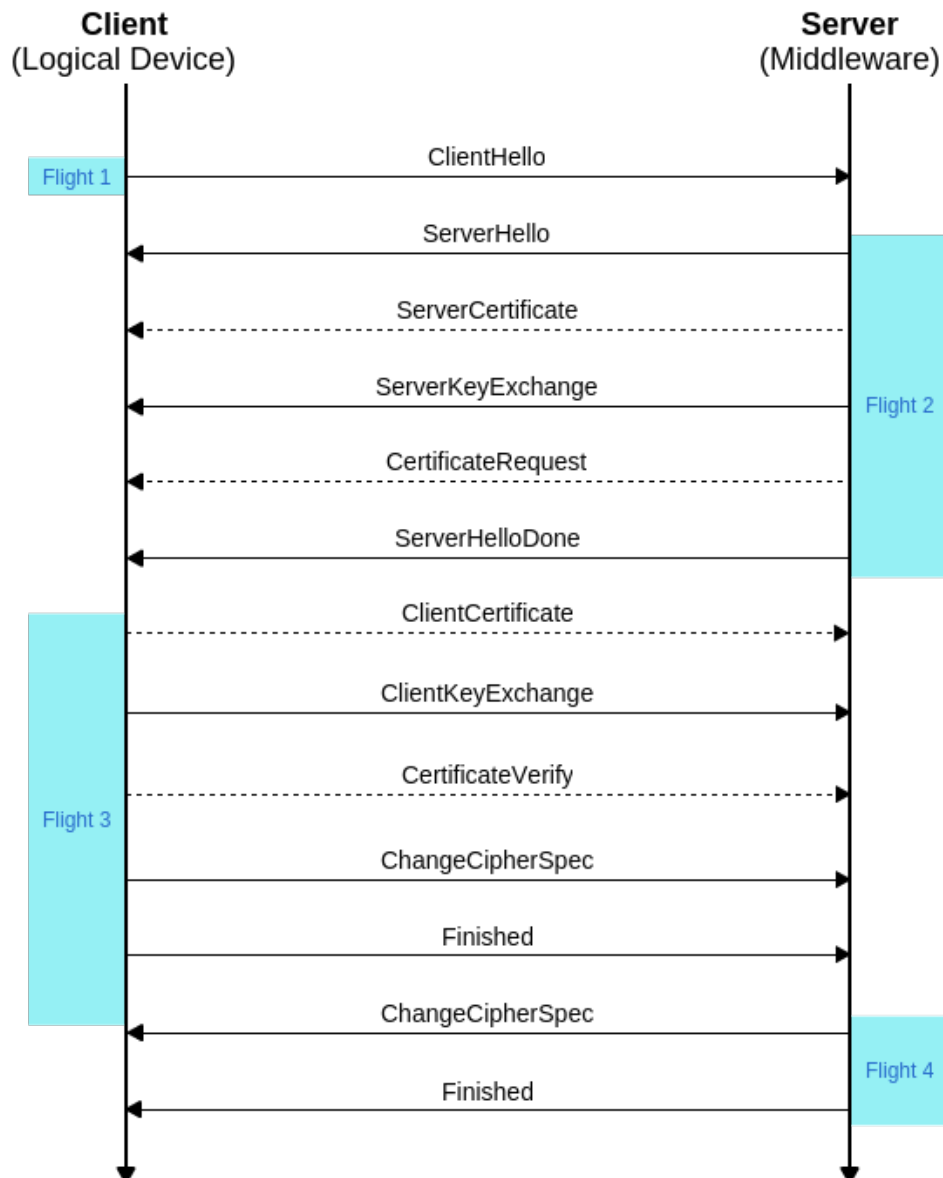


Figure APPENDIX C.1 – TLS Full Handshake.

Figure APPENDIX C.1 illustrates the messages exchanged during the handshake. The handshake happens in 4 flights:

Flight 1:

The client sends a ClientHello message to the server containing some important information such as protocol version, session id, cipher suite, etc.

Flight 2:

The server answers with a ServerHello message if it was able to find an acceptable algorithm to the required information. A ServerHello message is similar to a ClientHello message, containing protocol version, cipher suite, session id, etc.

Immediately after the server sends the ServerHello message, it sends a ServerCertificate and a ServerKeyExchange message to the client. The CertificateRequest message can optionally be sent by the server to request a certificate from the client if the server chooses to authenticate the client (as in DTLS handshake). Then the server sends a ServerHelloDone message and waits for a client answer.

Flight 3:

The client sends a ClientCertificate message to the server if it was required for send. After it, a ClientKeyExchange and a CertificateVerify message are sent by the client. A CertificateVerify message allows the server to verify a client certificate. If the server does not request a certificate, the ClientKeyExchange message is the first message sent by the client. A ChangeCipherSpec and a Finished message are sent by the client.

Flight 4:

In response, the server sends a ChangeCipherSpec message and a Finished message. At this point, the client and server can exchange data in a secure way.

TLS also has the possibility of an abbreviated handshake between client and server. In an ordinary full handshake, the server sends a session id as part of the ServerHello message. The client associates this session id with the server's IP address and TCP port, so that when the client connects again to that server, it can use the session id to abbreviate the handshake. In the server, the session id maps to the cryptographic parameters previously negotiated, specifically the "Master secret". Both sides must have the same "Master secret" or the resumed handshake will fail (this prevents an eavesdropper from using a session id). The random data in the ClientHello and ServerHello messages virtually guarantee that the generated connection keys will be different from in the previous connection. The exchanged messages are very similar with the DTLS abbreviated handshake presented previously (see Figure APPENDIX B.2). For this reason, we only give a brief description of the abbreviated handshake:

1. The client sends a ClientHello message including the session id from the previous TLS connection.

2. The server responds with a ServerHello message and if it recognizes the session id sent, it generates the new keys from the existing “Master secret” and the newly exchanged random values (like in DTLS). The server sends a ChangeCipherSpec and an encrypted Finished message, containing a hash and MAC over the previous handshake messages.
3. The client verifies the hash and MAC of last server’s message. If it fails, the handshake is considered to have failed, and the connection should be torn down. Finally, the client sends a ChangeCipherSpec and an encrypted Finished message. The server performs the same decryption and verification. If it was successful, the handshake is complete and the messages exchanged between client and server will be encrypted.

For further details, please refer to the TLS specification [15].

APPENDIX D – KEY MANAGEMENT

This appendix gives some examples of how to generate and store keys using keytool¹⁶ and openssl¹⁷. Next items show some commands examples.

Generating Client Private Key

We generate a client private key into private keystore using keytool.

```
keytool -genkey -alias clientprivate -keystore client.private -storetype
  JKS -keyalg ec -dname "CN=Your Name, OU=Your Organizational Unit, O=Your
  Organization, L=Your City, S=Your State, C=Your Country" -storepass
  clientpw -keypass clientpw
```

Generating Client Public Key

We generate a client public key and import it into public keystore using keytool.

```
keytool -export -alias clientprivate -keystore client.private -file temp.
  key -storepass clientpw
keytool -import -noprompt -alias clientpublic -keystore client.public -file
  temp.key -storepass public
rm -f temp.key
```

At the first command we received the message “Certificate stored in file <temp.key>”. At the second one we received the message “Certificate was added to keystore”, and at the third one we just remove the temp.key that assisted in client public key generation. The commands to generate the server’s key pair are the same as presented for the client.

Although keytool is used in some cases, it is not capable of signing Certificate Signing Requests (CSRs). In this sense, an alternative is to use openssl, as shown in next items.

Generate a Private Key of the CA

We generate a private key for root.

```
openssl ecparam -name secp521r1 -genkey -out root.key
```

Generate Self-Signed Root Certificate

The trusted client CA (Certificate Authority) signs its certificate with its own private key (root.key).

```
openssl req -new -key root.key -x509 -days 365 -out root.crt
```

This gives us a self-signed X.509 certificate (root.crt) which is valid for 365 days.

¹⁶<http://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html>

¹⁷<https://www.openssl.org/>

The presented commands using openssl are the first steps towards the generation of a certificate chain signed by a trusted authority using keytool and openssl. The complete tutorial and explanations are presented in APPENDIX C of work in [22]. For further details please refer to it.