University of Deusto

Boosting the adoption of Industry 4.0 through Internet of Things, industrial standards and software engineering

Doctoral thesis by

MARKEL IGLESIAS URKIA

Supervised by

Dr. AITOR URBIETA ARTECHE

Dr. DIEGO CASADO MANSILLA

Bilbao, February 2020

UNIVERSITY OF DEUSTO

BOOSTING THE ADOPTION OF INDUSTRY 4.0 THROUGH INTERNET OF THINGS, INDUSTRIAL STANDARDS AND SOFTWARE ENGINEERING

Doctoral thesis by
MARKEL IGLESIAS URKIA

within the program
ENGINEERING FOR THE INFORMATION SOCIETY
AND SUSTAINABLE DEVELOPMENT

Supervised by
Dr. AITOR URBIETA ARTECHE
Dr. DIEGO CASADO MANSILLA

PhD Candidate	Advisor	Advisor

Bilbao, February 2020

Noche eta Sherpyri

Abstract

With the advent of the Internet of Things (IoT), everyday objects are getting interconnected. These electronic devices are not only microcontrollers or microprocessors, but also other kind of objects such as food or clothing. In this new technological wave, lightweight network protocols are needed to connect these objects to the IoT. The main features of these are low network, energy and resource requirements. Relevant examples are the Constrained Application Protocol (CoAP) or the Message Queue Telemetry Transport (MQTT). When applying the IoT to the industrial domain, it appears the concept of Industrial Internet of Things (IIoT), which is one of the main enablers of the fourth industrial revolution, also known as Industry 4.0. However, as industrial life cycles are longer than consumer ones, and the industry is more conservative, it is slower adopting new technologies, and the used systems for monitoring and actuating are usually siloed from other systems.

Taking this issue in mind, the work carried out during this thesis aims to alleviate this latter fact, helping to develop interoperable and reliable systems for industry in a faster

and more efficient way, with the final goal of accelerating the adoption of Industry 4.0. To do that, first, MQTT and CoAP are surveyed and compared, aiming to select the most suitable to continue carrying on the work with one of them. This first iteration of the work resulted in selecting CoAP. From there, different open and available libraries of CoAP have been compared, including security features which are paramount for industry adoption. Then, the industrial standard IEC 61850 used for modelling, controlling and monitoring electrical substations, has been selected to add interoperability to the systems and avoid siloed deployments. It has been mapped to CoAP on a first stage, and then, the performance of the mapping was compared against other approaches from the body of literature that applied HTTP and SOAP. This mapping has sufficed some barriers because of the CoAP protocol limitations. More concretely, on its Observe extension which allows using CoAP following a publish-subscribe model. These limitations are focused on the subscription process. Hence, the next contribution of this thesis has been to propose and evaluate enhanced methods for subscribing to CoAP resources, to apply them to the IEC 61850 mapping. Finally, the last contribution of this thesis has been to apply software engineering techniques to improve the software development process for industrial devices, allowing faster and better software generation and reuse. To this end, two different specifications have been tested. On the one hand, the already mentioned IEC 61850. On the other hand, the

more generic Web of Thing (WoT), which aims to use already existing standards to overcome the interoperability issues and transform the IoT on the WoT. That is, the same way the Internet was transformed on the World Wide Web (WWW) in the early nineties.

Resumen

Con el Internet de las Cosas (Internet of Things, IoT), los objetos cotidianos se están interconectando. Estos dispositivos electrónicos no solo son microcontroladores o microprocesadores, si no todo tipo de objectos como comida o ropa. En esta nueva ola tecnológica son necesarios protocolos de comunicación ligeros para poder conectar estos objetos al IoT. Las principales características de tales protocolos son bajos requisitos en cuanto a red, energía y recursos. Algunos de los ejemplos más relevantes son Constrained Application Protocol (CoAP) o Message Queue Telemetry Transport (MQTT). Cuando se aplica el IoT al dominio industrial, un nuevo concepto aparece, Industrial Internet of Things (IIoT), el cual es uno de los principales facilitadores para la cuarta revolución industrial, también conocida como Industria 4.0. Sin embargo, los ciclos de vida industriales son más largos que los de los consumidores, además de que la industria es más conservadora, siendo más lenta en la adopción de las nuevas tecnologías, por lo que los sistemas utilizados para la monitorización y actuación están aisladas de otros sistemas.

Con este problema en mente, el trabajo realizado durante esta tesis tiene como objetivo aliviar este último hecho,

ayudando a desarrollar sistemas interoperables y fiables para la industria de una manera rápida y eficiente, con el propósito final de acelerar la adopción de la Industria 4.0. Para ello, primero se analizan y comparan MQTT y CoAP, para seleccionar el más adecuado de ellos para continuar el trabajo. Esta primera iteración del trabajo resulta en la selección de CoAP. Después de seleccionarlo, se han analizado diferentes librerías libres y disponibles de CoAP, incluyendo características de seguridad, las cuales son de vital importancia para la adopción industrial. A continuación, se ha seleccionado el estándar industrial IEC 61850 usada para el modelado, control y monitorización de subestaciones eléctricas con el objetivo de añadir interoperabilidad a los sistemas y evitar despliegues aislados. Como primera etapa, se ha mapeado a CoAP para después comparar el rendimiento del mapeo con otras propuestas en la literatura que aplican HTTP y SOAP. Este mapeo presenta algunas barreras debido a las limitaciones del protocolo CoAP. Concretamente, en su extensión de Observe, el cual permite usar CoAP con un modelo de comunicaciones de publicador-suscriptor. Estas limitaciones se enfocan en el proceso de suscripción, por lo que la siguiente contribución de esta tesis ha sido proponer y evaluar métodos más avanzados para suscribirse a recursos CoAP, y aplicarlos al mapeo del estándar IEC 61850. Finalmente, la última contribución de esta tesis ha sido aplicar técnicas de ingeniería de software para mejorar el proceso de desarrollo de software para dispositivos industriales, permitiendo generación

de software mejor y más rápido y reutilización de código. Para ello, se han probado dos especificaciones diferentes. Por un lado, la mencionada IEC 61850. Por el otro lado, una más genérica, la Red de las Cosas (Web of Things, WoT), la cual tiene como objetivo utilizar estándares ya existentes para solucionar problemas de interoperabilidad y transformar el IoT en el WoT. Esto es, de la misma manera que Internet se transformó en la World Wide Web (WWW) a principios de los noventa.

// Laburpena

Gauzen Internet-en (Internet of Things, IoT) etorrerarekin eguneroko objektuak elkarren artean interkonektatzen ari dira. Gailu elektroniko hauek ez dira mikokontrolagailu edo mikroprozesagailuak soilik, era guztietako objektuak dira, adibidez janaria edo arropa. Olatu teknologiko berri honetan sare protokolo arinak beharrezkoak dira objektu hauek IoTra konektatzeko. Protokolo hauen ezaugarri nagusiak sare, energia eta gaitasun eskaera baxuak dira. Protokolo hauen adibide azpimarragarrienak Constrained Application Protocol (CoAP) edo Message Queue Telemetry Transport (MQTT) dira. IoT industri domeinuetara aplikatzen denean kontzeptu berri bat agertzen da, Industrial Internet of Things (IIoT), zeinek laugarren iraultza industriala edo Industria 4.0 ahalbidetzen duen. hala ere, bizi zikloak luzeagoak dira industrian kontsumitzaleenak baino, industria kontserbadoreagoa izanik, polikiago barruratzen ditu teknologia berriak, monitorizazio eta eragingailu sistemak askotan isolatuta geratzen direlarik.

Arazo hau kontutan hartuta, tesi honetan egindako lanaren helburua industriarako sistema interoperable eta fidagarrien garapena era azkar eta eragikor batean egiten laguntzea

da, horrela Industria 4.0-aren adopzioa azkartuz. Horretarako, lehenengo MQTT eta CoAP aztertu eta konparatu dira, tesi honetako lana jarraitzeko egokiena aukeratzeko helburuarekin. Egindako lanaren lehenengo iterazio honetan CoAP izan da aukeratutakoa. Behin CoAP aukeratuta, libreak eta eskuragarri dauden hainbat CoAP liburutegi konparatu dira, segurtasun ezaugarriak ere kontutan hartuz, industriarako ezinbestekoak baitira. Ondoren, subestazio elektrikoen modelaketa, kontrola eta monitorizaziorako IEC 61850 estandar industriala aukeratu da, sistemen interoperabilitatea ahalbidetzeko eta isolatutako hedapenak ekiditeko. Lehen urrats bezala, CoAPera mapeatu da, gero mapeoaren errendimendua HTTP eta SOAP erabiltzen duten beste lan batzuekin konparatzeko. Mapeo honek hainbat muga dauzka, zehazki, Observe luzapenean, zeinek CoAP-i publikatzaile-harpide komunikazio gaitasuna ematen dion. Muga hauek harpidetza prozesuan nabarmentzen dira, eta horregatik, tesi honen hurrengo ekarpena harpidetza prozesua hobetzeko metodo berriak proposatu eta ebaluatzea da, IEC 61850 mapeoan aplikatuta. Bukatzeko, tesi honetako azkenengo kontribuzioa gailu industrialen softwarea garatzeko prozesuan software ingeneritzako teknikak aplikatzea izan da, kodea berrerabiliz eta software hobea azkarrago sortzeko. Horretarako, bi espezifikazio ezberdin erabili dira. Alde batetik, aurretik aipatutako IEC 61850. Bestetik, generikoagoa den Gauzen Sarea (Web of Things, WoT), zeinek aurretik existitzen diren estandarrak erabiltzea duen helburu, in-

teroperabilitatea lortzeko eta IoT WoT bihurtzeko. Hau da, laurogeita hamarreko hamarkadaren hasieran Internet World Wide Web (WWW)-ean bihurtu zenaren antzera.

Acknowledgments

Tesi hau aurrera atera ahal izateko jende askoren laguntza izan dut, eta hauek gabe ez litzateke posible izango honaino heltzea.

Lehenik eta behin, eskerrik asko Ikerlani, bertan doktore tesia egiteko aukera emateagatik. Eskerrik asko bereziki Aitorri, edozein momentutan laguntzeko eta gidatzeko prest egoteagatik, eta kaña sartu behar izan denean ere sartzeagatik. Ikerlango Cybersecure IoT taldeko kideei ere eskerrak eman nahi dizkiet, lan talde osasuntsua edukitzeagatik, beti laguntzeko prest. Azkenik, Ikerlango beste doktoregaiei ere bai, Orive, Goiuri, Aitziber, Mikel, Mihail, Víctor.... Erderaz esaten den bezala, "mal de muchos, consuelo de tontos", baina momentu txarretan beste batzuk ere antzeko bizipenak izaten ari direla baina aurrera ateratzen direla ikusteak aurrera jarraitzeko indarrak ematen baititu.

Deustoko Unibertsitateari ere eskerrak eman nahi dizkiot, doktoretzako programan matrikulatzeko aukera emateagatik. Gracias Diego por tu ayuda como supervisor aunque sea en la distancia, siempre dispuesto a apoyar y echar una mano.

I also want to thank Simon for helping me with some parts of my PhD and for inviting me to go to Pro^{2}Future for my

research stay. I really felt welcome, thank you everyone there. Simon, Heimo, Christina, Sandra, Konrad, Daniel, Josef, Markus, Leo, Georgios, Milot, Nikolina, Amer, Matej, Stanley. Thank you, danke, obrigado, *ευχαριστώ*, hvala.

Gracias a Abel también, por tu ayuda en la ultima parte de la tesis.

Eskerrik asko be bai Aita ta Amai, zuengatik ez bazan izango ez nintzelako honaino ailegauko, ikasteko aukeria emutiatik eta pazientzia eukitxiatik.

Por último, la mas importante, la luz que ilumina mi camino. Raquel, gracias por todo el apoyo, por aguantarme y por hacerme la vida más fácil. ¡Te quiero!

Eskerrik asko,

Markel Iglesias Urkia

February 2020

Contents

List of Figures **xxiii**

List of Tables **xxviii**

1 Introduction **1**

1.1 Motivation 2
1.2 Hypothesis 6
1.3 Objectives 6
1.4 Contributions 7
1.5 Methodology 11
1.6 Publications 13
1.7 Research Context 14
1.7.1 Research Support 14
1.7.2 Research Activities 17
1.8 Outline 19

2 Background **23**

2.1 Introduction 24
2.2 Internet of Things 25
2.2.1 Communication models 29
2.2.2 Hardware 31
2.2.3 Software 34

2.2.4 Transport Layer Protocols 35
2.2.5 Security Protocols 36
2.2.6 Application Layer Protocols 37
2.2.6.1 Internet Protocols 37
2.2.6.2 Industrial Protocols 38
2.2.6.3 IoT Protocols 40
2.2.6.4 Summary of the protocols 42
2.3 Data and Service models 43
2.3.1 IEC 61850 44
2.3.2 Web of Things 46
2.3.2.1 Organization 47
2.3.2.2 Normative Deliverables 47
2.3.2.3 Informative Deliverables 49
2.4 Software Engineering 50
2.4.1 Software Product Lines 51
2.4.2 Domain Specific Language 51
2.4.3 Model Driven Engineering 52

3 Lightweight Communication Protocols 53
3.1 Introduction 54
3.2 Related Work 58
3.3 Protocol Comparison 65
3.3.1 Analyzed Protocols 66
3.3.1.1 MQTT 66
3.3.1.2 CoAP 69
3.3.2 Experiment Setup 72
3.3.2.1 Scenario Definition 72
3.3.2.2 Architecture Design 75
3.3.2.3 Hardware and Software 76

3.3.3 Results 77
3.3.3.1 Overhead 77
3.3.3.2 Scalability 79
3.3.3.3 Latency 83
3.4 Comparison of CoAP Libraries 85
3.4.1 CoAP 86
3.4.2 CoAP implementations 91
3.4.3 Security in CoAP: DTLS 94
3.4.4 Feature Comparison 97
3.4.5 Security Feature Comparison 101
3.4.6 Experiment Setup 103
3.4.7 Results 105
3.4.7.1 Compatibility 105
3.4.7.2 Latency 106
3.4.7.3 Resource Consumption 107
3.5 Conclusion 111

4 Interoperability through IEC 61850 113
4.1 Introduction 115
4.2 Related Work 118
4.3 IEC 61850 122
4.3.1 Basic information model 123
4.3.2 Control blocks for additional functions 124
4.4 Mapping IEC 61850 to CoAP 126
4.4.1 Basic Services 127
4.4.2 Reporting Services 130
4.4.3 Logging Services 132
4.4.4 Setting Services 133
4.4.5 Eventing Services 134

4.4.6 Sampled Value Transmission Services 136
4.4.7 General Functionality Services 137
4.4.8 Other Services 139
4.4.8.1 Control Model Services 139
4.4.8.2 Time and Time Synchronization Services 139
4.5 Implementation of the mapping 139
4.6 Evaluation 145
4.6.1 Latency 147
4.6.2 Overhead 149
4.7 Lightweight Resource Representation 151
4.8 Validation 152
4.8.1 Latency 152
4.8.2 Overhead 155
4.9 Conclusion 158

5 Advanced Subscription Mechanisms 161
5.1 Introduction 162
5.2 Related Work 164
5.2.1 IETF RFCs 164
5.2.2 IETF working drafts 165
5.2.3 Other proposals 166
5.3 Enhancement Proposal 170
5.3.1 Subscribe through PUT/POST requests 172
5.3.2 Lightweight responses 173
5.3.3 Subscribe through third resources 175
5.3.4 New CoAP options and response codes 178
5.4 Evaluation: exploratory example 180
5.5 Implementation in the IEC 61850 Mapping 183

5.6 Validation 184
5.7 Conclusion 186

6 Software Engineering Techniques in the IoT 189
6.1 Introduction 191
6.2 Related Work 194
6.3 Technological Background 199
6.3.1 Model-Driven Engineering and Domain Specific Language 199
6.3.2 Automated code generators 200
6.4 IEC 61850 based model 201
6.4.1 Problem Statement 201
6.4.1.1 Use cases 203
6.4.2 Solution Design: TRILATERAL 206
6.4.3 Implementation 210
6.4.4 Evaluation 216
6.4.5 Validation 218
6.5 WoT based model 224
6.5.1 First Iteration 225
6.5.1.1 Implementation 225
6.5.1.2 Generic Framework Implementation . 226
6.5.1.3 WoT system implementation 230
6.5.1.4 Evaluation 233
6.5.2 Second iteration 236
6.5.2.1 Abstract syntax 236
6.5.2.2 Concrete syntax 242
6.5.2.3 From Concrete Syntax to Abstract Syntax 244

6.5.2.4 Developing a WoT servient using the WoT Toolkit 246
6.6 Conclusion 250

7 Conclusions and Future Work 257
7.1 Conclusions 258
7.1.1 C1: Analysis of IoT 259
7.1.1.1 C1.1: Analysis of IoT communication protocols 259
7.1.1.2 C1.2: Analysis of CoAP 260
7.1.2 C2: Interoperability though IEC 61850 262
7.1.2.1 C2.1: Mapping IEC 61850 to CoAP . 262
7.1.2.2 C2.2: Implementation and comparison of the mapping 263
7.1.3 C3: CoAP enhancement proposal 264
7.1.4 C4: Apply Software Engineering techniques in IoT environments 265
7.1.4.1 C4.1: IEC 61850 based metamodel . 266
7.1.4.2 C4.2: WoT based metamodel 266
7.1.5 General conclusions of the thesis 268
7.2 Future Work 270
7.2.1 C1: Analysis of IoT 271
7.2.1.1 C1.1: Comparison of IoT protocols . 271
7.2.1.2 C1.2: Comparison of CoAP implementations 272
7.2.2 C2: Interoperability though IEC 61850 273
7.2.3 C3: CoAP enhancement proposal 275
7.2.4 C4: Apply Software Engineering techniques in IoT environments 275

7.2.4.1 C4.1: IEC 61850 based metamodel . 275
7.2.4.2 C4.2: Interoperability through WoT . 276
7.2.5 Future work of the thesis 278

List of Figures

1.1 Summary of the four Industrial Revolutions. 3
1.2 Industry 4.0 value chain. 4
1.3 Application domains of Industry 4.0. 5
1.4 Methodology followed during the thesis. 12
1.5 Ikerlan. 14
1.6 University of Deusto. 16
1.7 Pro^{2}Future. 17

2.1 Comparison of TCP/IP and OSI protocol stack with examples. 28
2.2 Device-to-device communication example. 29
2.3 Device-to-cloud communication example. 30
2.4 Device-to-Gateway communication example. 30
2.5 Back-end data sharing example. 31
2.6 Raspberry Pi 3 model B. 32
2.7 STM32F407VGT6 Discovery and the Ethernet expansion board. 33
2.8 Consumer-Thing interaction: a thing exposes its features using the TD. 46

3.1 MQTT broker, publisher and subscribers. 67
3.2 Message exchange with different QoS on MQTT. . . . 69
3.3 Architecture scheme. 75
3.4 Platform scheme. 76
3.5 Overhead on control scenarios for CoAP and MQTT. . 79
3.6 Overhead on monitoring scenarios for CoAP and MQTT. 80
3.7 One-to-many control overhead with 1 to 20 actuators. . 81
3.8 Proportional transmission size per actuator. 81
3.9 One-to-many monitoring overhead. 82
3.10 Proportional size per monitor. 82
3.11 Control latency. 83
3.12 Monitoring latency. 84
3.13 Event detection latency. 85
3.14 CoAP message [34]. 88
3.15 DTLS handshake with no authentication (a), server-side only (b) and mutual authentication (c). 96
3.16 Experiment setup. 104
3.17 Maximum RTT in ms. 106
3.18 Median RTT in ms. 107
3.19 Mean RTT in ms. 108
3.20 Minimum RTT in ms. 108

4.1 The Information Model of the IEC 61850 standard. . . 123
4.2 The Information Model of the IEC 61850 standard with the implemented blocks in black and the not implemented ones in grey. 140
4.3 Screenshot of the tool that generates the IEC 61850 code automatically. 143
4.4 IEC 61850 tool layers. 144

4.5 Model used in the experimentation. 145
4.6 Data and overhead bytes. 150
4.7 Layered overview of the reference design of the IEC 61850 software tool. 152
4.8 Percentage of CoAP's median response time compared to HTTP and SOAP in the functions without block-wise transfer. Values below 100% mean that the protocol is slower than CoAP. 155
4.9 The bars show the number of bytes required for the resource representation and overhead in CoAP, HTTP and WS-SOAP implementations. The lines represent the overheads of HTTP and SOAP as fractions of the CoAP baseline. 156
4.10 Percentage of required bytes for the resource representation of HTTP's (JSON) implementation with CoAP's (CBOR) as baseline. 158
4.11 Percentage of total number of required bytes (payload + protocol overhead) for sending data in HTTP's and WS-SOAP's implementation with CoAP's as baseline. . 159

5.1 Subscribing to a resource and getting notifications using CoAP and the Observe extension. 163
5.2 Update the values of a resource, subscribe to it and get notifications. 173
5.3 Subscription to a resource without getting the current state and get notifications. 174
5.4 Change a resource, subscribe to it but do not receive a payload, then get notifications. 175

5.5 Get the representation of a resource, subscribe to a different resource and get notifications. 176
5.6 Update a resource, subscribe to a different resource and get notifications. 177
5.7 Poll a resource without getting the representation, subscribe to a different resource and get notifications. . . . 178
5.8 Update a resource without getting the representation, subscribe to a different resource and get notifications. . 178
5.9 Overhead in the old and new approaches in different usages of a clock and alarm use case. 182
5.10 Overhead in the old and new approaches: SetBRCBValues (1) and SetURCBValues (2) with a big payload; and SetBRCBValues (3) and SetURCBValues with small payload (4). The left bar of each pair represents the current message exchange with CoAP. The right bar, the new approach with new option and response codes. 185

6.1 Manual vs Automated processes. 202
6.2 TRILATERAL components. 207
6.3 TRILATERAL workflow. 208
6.4 Server definition feature model. 210
6.5 TRILATERAL implementation steps. 210
6.6 Simplified version of the metamodel used in TRILATERAL. 211
6.7 Screenshot of TRILATERAL's tree view editor. 212
6.8 Layers of the IEC tool. 213
6.9 Creating a ICPS artifact in TRILATERAL. 216
6.10 Different protocol sizes. 218
6.11 Catenary-free tram and its systems. 219

6.12 Screenshot of TRILATERAL's tree view editor describing a part of the climate system of a catenary-free tram. 221
6.13 Workflow and the used tools, with intermediate results before the final code. 226
6.14 Metamodel based on the WoT specification on the Working Draft of October 21. DataSchema has several subclasses. 228
6.15 Graphical tool to generate the tree structure that allows to generate the .wot file. 230
6.16 Different layers of the code projects. 231
6.17 Small part of the generated code as an example. 235
6.18 EMF implementation of the TD Core vocabulary model. 238
6.19 EMF implementation of the TD Data Schema vocabulary model. 240
6.20 EMF implementation of the TD WoT security vocabulary model. 241
6.21 EMF implementation of the TD Hypermedia controls vocabulary model. 242
6.22 Metamodel derived from the concrete syntax for the WoT TD specification. 245
6.23 Development process of a WoT Servient using the WoT tool. 249
6.24 Different phases on the development of projects. 253

7.1 General view of the time to develop projects from scratch versus projects implemented with TRILATERAL. 277

List of Tables

1.1 Publications with their type and status, and their corresponding chapter, contribution, and research question. . 13

2.1 Different protocols on IoT and Web stacks. 29
2.2 Comparison of IoT protocols. 42
2.3 Specification of the different parts of the IEC 61850 within the different sections of the standard. 45
2.4 WoT WG Normative deliverables. 47
2.5 WoT WG Informative deliverables. 48

3.1 CoAP and other protocols comparisons. 62
3.2 Previous CoAP benchmarks. 64
3.3 MQTT message types and their hexadecimal code. . . 68
3.4 CoAP request codes. 70
3.5 CoAP response codes. 71
3.6 IoT and general application stacks. 87
3.7 Most important options of CoAP. 88
3.8 CoAP libraries' features - Part 1. 98
3.9 CoAP libraries' features - Part 2. 98
3.10 DTLS libraries' features. 102
3.11 Implementation compatibility results. 105
3.12 ROM usage in bytes. 109

3.13 CPU and RAM usage in second and Kbytes. 110

4.1 IEC 61850 mappings: communication protocol, data format, application domain, metrics and number of mapped functions. 118
4.2 Mapping of basic services to CoAP. 127
4.3 Mapping of reporting services to CoAP. 130
4.4 Mapping of logging services to CoAP. 132
4.5 Mapping of setting services to CoAP. 133
4.6 Mapping of eventing services to CoAP. 135
4.7 Mapping of sample value transmission services to CoAP. 136
4.8 Mapping of additional services to CoAP. 137
4.9 The set of the implemented IEC 61850 services mapped to the CoAP communication protocol. 142
4.10 Maximum and median latencies in ms with CoAP, HTTP and SOAP. 148
4.11 Maximum and median of the response times in ms using JSON and CBOR resource representation in the CoAP implementation. 154

5.1 Summary of the related work on CoAP's publish/subscribe. 170
5.2 New proposed options for CoAP. 179
5.3 New proposed response codes for CoAP. 180
5.4 Action for a clock use cases with current mechanisms and with the new approach. 181
5.5 The sizes of the different parts of a CoAP message in the alarm clock use case. 181

5.6 Updating [UB]RCBs and subscribing to Reports with the current CoAP specification and with the new enhanced subscription mechanism. 184

6.1 Previous CoAP benchmarks. 198
6.2 Characteristics of the different domains regarding the connectivity. 204
6.3 Libraries and executables need for introducing communication protocols to an artifact by TRILATERAL. . . 217
6.4 Characteristics of the use cases and the selected communication protocol. 223
6.5 Functions for sending request to servers in order to retrieve or update Properties, invoke Actions, and observe Properties, Actions and Events. 232

7.1 Summary of the contributions, on which chapter they are explained, which RQ they address and the related published articles. 259

Glossary

ACK Acknowledgement

ACSI Abstract Communication Service Interface

AMQP Advanced Message Queuing Protocol

BIM Basic Information Model

BLE Bluetooth Low Energy

BRCP Buffered Report Control Blocks

C Contribution

CB Control Blocks

CBOR Concise Binary Object Representation

CoAP Constrained Application Protocol

CoCoA CoAP Simple Congestion Control/Advanced

CORBA Common Object Request Broker Architecture

CoRE Constrained RESTful Environments

CON CONfirmable

CPS Cyber-Physical System

CPU Central Processing Unit

CRUD Create, Read, Update, and Delete

D2D Device-to-Device

DDoS Distributed Denial of Service

DDS Data Distribution Service

DER Distributed Energy Resources

DoS Denial of Service

DTLS Datagram Transport Layer Security

EM Entity Manager

EMF Eclipse Modeling Framework

EMOF Essential Meta-Object Facility

ETSI European Telecommunications Standards Institute

FC Functional Constraint

FCD Functionally Constrained Data

FCDA Functionally Constrained Data Attribute

FTP File Transfer Protocol

GoCB GOOSE Control Block

GOOSE Generic Object Oriented Substation Event

GsCB GSSE Control Block

GSSE Generic Substation State Event

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IANA Internet Assigned Numbers Authority

IEC International Electrotechnical Commission

IED Intelligent Electronic Device

IETF Internet Engineering Task Force

IDL Interface Definition Language

IG Interest Group

IIoT Industrial Internet of Things

IoT Internet of Things

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

JMS Java Message Service

JSON JavaScript Object Notation

JVM Java Virtual Machine

LAN Local Area Network

LCB Log Control Block

LD Logical Devices

LED Light-Emitting Diode

LLN0 Logical Node Zero

LN Logical Nodes

LPWAN Low-Power Wide-Area Networks

M2M Machine-to-Machine

M2T Model-to-Text

MMS Manufacturing Message Specification

MQTT Message Queue Telemetry Transport

MQTT-SN MQTT for Sensor networks

MQTT-WS MQTT over Web Sockets

MSVCB Multicast Sample Value Control Block

NB-IoT NarrowBand IoT

NDN Named Data Networking

NON NON-Confirmable

OASIS Organization for the Advancement of Structured Information Standards

OLE Object Linking and Embedding

OMG Object Management Group

OPC OLE for Process Control

OPC-UA OPC Unified Architecture

OS Operative System

OTG On-The-Go

PSK Pre-Shared Key

QoS Quality of Service

RAM Random Access Memory

RD Resource Directory

REST REpresentational State Transfer

RFC Request for Comments

RFID Radio Frequency IDentification

ROM Read-Only Memory

RPK Raw Public Key

RTT Round Trip Time

RQ Research Question

SASL Simple Authentication and Security Layer

SBC Single Board Computer

SCADA Supervisory Control and Data Acquisition

SE Software Engineering

SG Setting Group

SGCB Setting Group Control Block

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

TCP Transmission Control Protocol

TD Thing Description

TRILATERAL sofTware pRoduct lIne based muLtidomain iot ArTifact gEneration for industRiAL cps

TLS Transport Layer Security

UDP User Datagram Protocol

URCB Unbuffered Report Control Blocks

URI Uniform Resource Identifier

USCVB Unicast Sample Value Control Block

W3C World Wide Web Consortium

WG Working Group

WoT Web of Things

WPAN Wireless Personal-Area Networks

WS Web Services

WSDL Web Services Description Language

XML eXtensible Markup Language

XMPP eXtensible Messaging and Presence Protocol

The beginning is the most important part of the work.

Plato

Everybody has a plan... until they get punched in the face.

Mike Tyson

1 Introduction

This chapter introduces the dissertation of the PhD thesis. It starts presenting the change that Industry 4.0 is promoting on industrial environments to put the thesis in context. After that, the hypothesis is formulated, to continue with the objectives to validate the hypothesis, formulated as Research Questions (RQ). In order to answer those RQ, several contributions have been made, which are described in the next section. The contributions have been disseminated on a number of conference papers, journal articles and a book chapter, which are also summarized next. Finally, the research activities and the outline of this dissertation are described.

Contents

1.1 Motivation 2

1.2 Hypothesis 6

1.3 Objectives 6
1.4 Contributions 7
1.5 Methodology 11
1.6 Publications 13
1.7 Research Context 14
1.7.1 Research Support 14
1.7.2 Research Activities 17
1.8 Outline 19

1.1 Motivation

At the present time, human kind and especially the industry are in the middle of the transition to the fourth industrial revolution, also known as Industry 4.0 (Figure 1.1 summarizes the four industrial revolutions) [108]. The first one took place on the 18th century with the mechanization and the use of steam powered machines. The second industrial revolution brought mass production, the switch from steam power to electricity and the appearance of assembly lines on the 19th century. The third industrial revolution started on mid 20th century when electronics and computers began to be applied to factory machines, providing automation to the processes. The fourth industrial revolution focuses on connectivity, connecting all kinds of devices and machines, exchanging and managing information, without human intervention [160].

Several organisms have been composed with the objective of generating technologies, standards, methodologies, and techniques that help with this revolution. Some of the most prominent organisms are the International Telecommunication Union (ITU)[1], the International

[1] https://www.itu.int

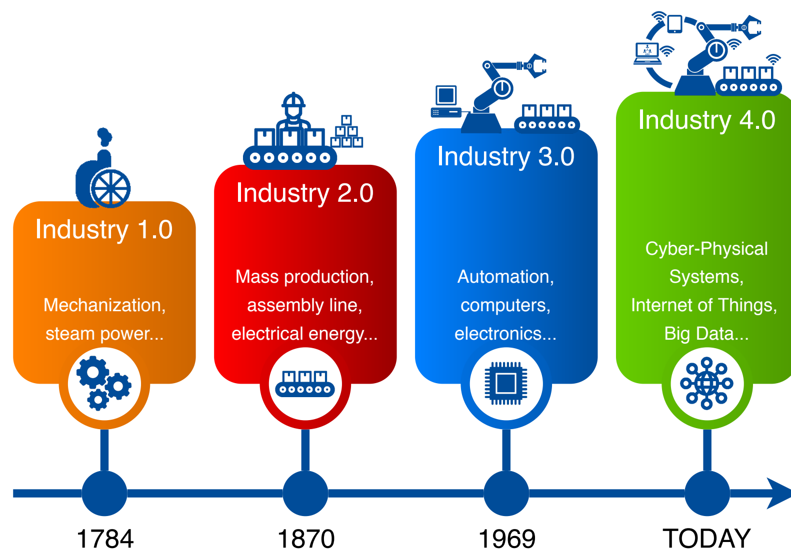


Figure 1.1: Summary of the four Industrial Revolutions.

Electrotechnical Commission (IEC)[1], the Institute of Electrical and Electronics Engineers (IEEE)[2], and the Internet Engineering Task Force (IETF)[3].

To be able to enable the changes required for Industry 4.0, and following the mentioned organisms' proposals, several concepts arise. These concepts allow getting data from sensors that feel the physical world and bring the data to devices that analyze it to make decisions, forming a value chain, presented in Figure 1.2. When a decision is made, the inverse chain can be followed to apply the decision to actuators and have the desired effect in the physical world.

[1] `https://www.iec.ch/`
[2] `https://www.ieee.org/`
[3] `https://www.ietf.org/`

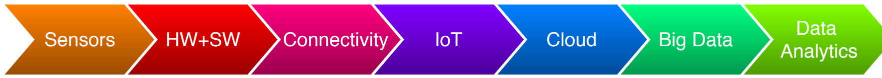


Figure 1.2: Industry 4.0 value chain.

One of the most important parts of the value chain is the Internet of Things (IoT), which is the way that different devices are connected and exchange information. As an evolution of the Internet, IoT started using regular Internet protocols such as HTTP, FTP, or XMPP. When applied to industry, the IoT is often named Industrial Internet of Things (IIoT), and same as regular Internet, previous existing protocols such as OPC-UA, DDS, or AMQP have also been used in this domain. However, taking into account that IoT devices are more heterogeneous and that some of the devices can be really small with hard resource constraints, new protocols needed to be developed. These new protocols are specifically designed for networks with a huge number of small devices that transmit small pieces of data, hence, they have low overhead and are very scalable. This makes them suitable for resource constrained devices, that have small computational capability and are often battery powered. Some of the protocols for the application layer developed for the IoT are CoAP and MQTT.

To achieve interconnectivity between systems, interoperability is one of the most important keys. This way, using common models to represent the elements of a system while using standard resource representation formats and communication protocols enables the interconnectivity of different systems, avoiding siloed solutions.

However, even though the technological development is agile, the adoption pace by industrial companies is not so fast [136]. There are several reasons for this, which include the lack of skill and off-the-

shelf solutions, that some industries are not agile to changes, and that the required investments are high. However, when Industry 4.0 and its entire value chain are fully implemented, they will produce deep changes that will be applied in several domains, giving intelligence to such domains, presented in Figure 1.3.

Figure 1.3: Application domains of Industry 4.0.

For one of the showed domains, i.e., Smart Grids, one important standard has been specified, more concretely for describing the communication networks and systems in electrical substation. This standard is IEC 61850 designed by the IEC. At Ikerlan, several projects have been developed using this or other related standards and it has been applied not only for Smart Grid environments, but also other applications such as catenary-free tram energy efficiency monitoring and control for Smart Mobility, home appliance energy efficiency monitoring and control for Smart Homes, or monitoring and control of energy harvesting devices on elevator systems for Smart Buildings. These usages prove that some standards can go beyond their original purpose and can be applied in environments outside of their initial intended use.

The main objective of this thesis is to evaluate the applicability of IoT protocols, other industrial standards, and software engineering techniques to industrial domains to enable a faster development, knowl-

edge and technology reusability, and interoperability. This way, the industrial adoption of new technologies may be accelerated and fulfill the vision of Industry 4.0.

1.2 Hypothesis

With the explained motivation in mind, the following hypothesis has been formulated to work on during the duration of the thesis:

"IoT-based lightweight communication protocols, industrial standards, and software engineering techniques can accelerate the process of Industry 4.0 adoption in order to bring together siloed systems enabling more interoperable and reliable communications."

All the work and the contributions carried out in during this thesis have been developed aiming to validate this hypothesis.

1.3 Objectives

With the goal of proving the formulated hypothesis, some objectives have been established, and inside those objectives several Research Questions (RQ). Answering these questions will help to achieve the objectives, as the answers provide steps in the direction to fulfil the main motivation of the thesis, to evaluate the applicability of the IoT and other standards in industrial domains with the goal of advancing in the state of the art regarding the Industry 4.0 adoption. Following are described the objectives and their related research questions:

- O1: Analysis of lightweight communication protocols.
  - RQ1.1: Which IoT protocols are available?
  - RQ1.2: Which is the protocol that best suits for IIoT?

  - RQ1.3: Which libraries are available for the selected protocol?

- O2: Implementing the selected protocol in industrial scenarios.

  - RQ2.1: How can such protocol be deployed on a real Industrial scenario?
  - RQ2.2: Is the implementation faster/lighter/less power demanding than other approaches?

- O3: Enhancing the selected IoT protocol.

  - RQ3.1: Does the selected protocol a missing capability to fit better within an Industrial scenario?
  - RQ3.2: How can the missing capability be implemented?

- O4. Apply software engineering techniques to the development of IoT.

  - RQ4.1: Can software engineering be applied to industrial standards?
  - RQ4.2: Can software engineering be applied to more general IoT specifications?

1.4 Contributions

To validate the formulated hypothesis, several contributions have been carried out during this thesis. These contributions are directly related to the RQs presented in Section 1.3. The contributions are named on the following and then described:

- C1: Analysis of IoT.
  - C1.1: Comparison of IoT protocols.
  - C1.2: Comparison of CoAP implementations.
- C2: Interoperability through IEC 61850.
  - C2.1: Mapping IEC 61850 to CoAP.
  - C2.2: Implementation and comparison of the mapping.
- C3: CoAP enhancement proposal.
- C4: Apply Software Engineering techniques in IoT environments.
  - C4.1: IEC 61850 based metamodel.
  - C4.2: WoT based metamodel.

C1: Analysis of IoT

The first contribution of this thesis is an analysis of the different available IoT communication protocols. This contribution aims to answer *"RQ1.1: Which IoT protocols are available?"*, *"RQ1.2: Which is the protocol that best suits for IIoT?"* and *"RQ1.3: Which libraries are available for the selected protocol?"*, and it is described in Chapter 3. It includes two subcontributions, one that compares IoT protocols and another one that compares nine of the available CoAP implementations.

In the first one, i.e., *"C1.1: Comparison of IoT protocols"*, two of the most important IoT protocols have been selected for the comparison, MQTT and CoAP. To carry out the experiments, a STM32F407 Discovery board has been used, and the overhead, scalability and latency

have been analyzed in three different scenarios, i.e., control, monitoring and eventing, aiming to answer *"RQ1.1: Which IoT protocols are available?"* and *"RQ1.2: Which is the protocol that best suits for IIoT?"*.

The second subcontribution, i.e., *"C1.2: Comparison of CoAP implementations"* focuses on CoAP. The goal for this subcontribution is to answer *"RQ1.3: Which libraries are available for the selected protocol?"* and it analyzes the features and performance on nine different CoAP implementations. The experiments have been carried out using the Raspberry Pi platform and the libraries have been analyzed theoretically covering a variety of properties, and practically, analyzing the performance in terms of compatibility, latency times and resource usage (i.e., ROM, RAM and CPU).

C2: Interoperability through IEC 61850

The second contribution uses the IEC 61850 standard to provide interoperability. This contribution is divided in two subcontributions, one to propose a mapping of IEC 61850 to CoAP, and another one to implement and analyze the mapping and its performance. This contribution targets *"RQ2.1: How can such protocol be deployed on a real Industrial scenario?"* and *"RQ2.2: Is the implementation faster/lighter/less power demanding than other approaches?"* and it is the focus for Chapter 4.

The first subcontribution, *"C2.1: Mapping IEC 61850 to CoAP"*, first surveys other approaches that use different communication protocols with the IEC 61850 standard. After that, it presents a mapping proposal to use CoAP for the communication functions of IEC 61850.

The second one, *"C2.2: Implementation of the mapping"*, implements the mapping with some changes and compares its performance to other approaches using HTTP and SOAP. The compared metrics have been latency and overhead.

C3: CoAP enhancement proposal

Contribution C2 suffices some limitations on CoAP, which have been addressed here, aiming to answer *"RQ3.1: Does the selected protocol a missing capability to fit better within an Industrial scenario?"* and *"RQ3.2: How can the missing capability be implemented?"*. The details of C3 are further explained in Chapter 5.

For this contribution, enhanced mechanisms have been proposed to provide more advanced features in the subscription process of CoAP's Observe extension. In order to do that, new CoAP option and response codes are proposed, to enable lightweight subscription responses, single step subscriptions when creating or updating resources, and subscriptions through related resources.

C4: Apply Software Engineering techniques in IoT environments

The last contribution aims to accelerate and improve the development and deployment process of IoT devices using Software Engineering techniques, answering *"RQ4.1: Can software engineering be applied to industrial standards?"* and *"RQ4.2: Can software engineering be applied to more general IoT specifications?"*. This contribution uses SPL, DSL, and MDE and Chapter 6 describes it in depth.

In this contribution, there are two subcontributions. The first one explains TRILATERAL, a tool that uses SPL, DSL, and MDE, and automatically generates the code for ICPSs using a tree view editor

which uses a metamodel based on IEC 61850, answering *"RQ4.1: Can software engineering be applied to industrial standards?"*.

The second subcontribution changes the base to generate the metamodel from IEC 61850 to the WoT TD, focusing on *"RQ4.2: Can software engineering be applied to more general IoT specifications?"*. The WoT TD is a more generic specification that is not so focused to specific domains, like IEC 61850. This subcontribution aims to automatically generate WoT servients, once again, using a tree view editor, but with an additional JSON editor.

1.5 Methodology

With the purpose of answering the research questions form the previous section, an incremental contribution methodology has been applied during the work of this thesis. Figure 1.4 shows the steps that have been followed for each contribution.

These steps have been followed for each contribution, and each contribution builds on the previous ones. The steps are:

- Literature review: a study analyzing the related work, and find points of improvement on the state of the art.
- Preliminary study: analyze the possible solutions for improving the state of the art and define the description and the objectives for the contributions.
- Design: design the proposed solution, its implementation and the experiments to validate the proposal.
- Implementation: develop the proposed solution.

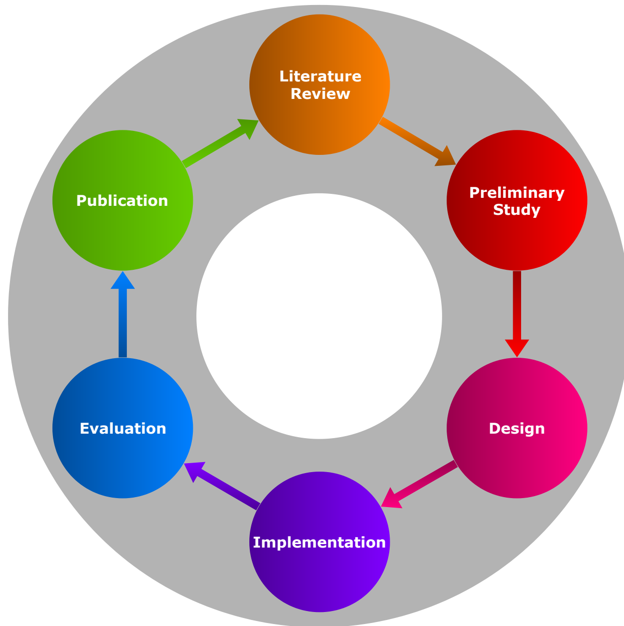


Figure 1.4: Methodology followed during the thesis.

- Evaluation: validate the proposed solution and its implementation with experiments.
- Publications: disseminate the obtained results publishing them on conference or journal papers.

1.6 Publications

During the work carried out for this thesis, several papers have been published. Not all of them are explained in this dissertation, as they are not directly related to the PhD topic, but most of them are direct results of the contributions. Table 1.1 summarizes the achieved publications, the type of publication, the status, and to which chapter, contribution and RQ they correspond.

Publication	Type	Status	Chapter	Contribution	RQ
[93]	Journal	Published	3	C1.2	RQ1.3
[85]	Journal	Published	4, 5	C2, C3	RQ2.1, RQ2.2, RQ3.1, RQ3.2
[88]	Journal	Submitted	6	C4.2	RQ4.2
[91]	Conference	Published	3	C1.1	RQ1.1, RQ1.2
[92]	Conference	Published	3	C1.2	RQ1.3
[94]	Conference	Published	4	C2.1	RQ2.1
[87]	Conference	Published	4	C2.2	RQ2.1, RQ2.2
[86]	Conference	Published	5	C3	RQ3.1, RQ3.2
[81]	Conference	Published	6	C4.1	RQ4.1
[89]	Conference	Published	6	C4.2	RQ4.2
[90]	Book Chapter	Published	6	C4.1	RQ4.1
[161]	Conference	Published	-	-	-
[62]	Conference	Published	-	-	-

Table 1.1: Publications with their type and status, and their corresponding chapter, contribution, and research question.

1.7 Research Context

This document is the doctoral dissertation for obtaining a PhD in the "Engineering for the Information Society and Sustainable Development" program at the University of Deusto. It is an Industrial Thesis that have been worked on from 2016 to 2019, under the supervision of Dr. Aitor Urbieta Arteche and Dr. Diego Casado Mansilla.

1.7.1 Research Support

Three different institutions have taken part on the development of this work: Ikerlan Technology Research Centre, University of Deusto and Pro²Future GmbH.

Ikerlan Technology Research Centre

The main working place for the work carried out in this thesis has been the Cybersecure IoT team at Ikerlan[1] (Figure 1.5), under the supervision of Dr. Aitor Urbieta Arteche.

(a) Ikerlan logo.

(b) Ikerlan's main building.

Figure 1.5: Ikerlan.

[1] https://www.ikerlan.es/en/

Ikerlan is a technological research centre inside the Mondragon Corporation[1]. It was founded in 1974 and since then, its aim has been to seek excellence in R&D&i, adapting to its customers and being close to the business reality.

Using a collaboration model that combines technology transfer, internal research and the training of highly qualified personnel, Ikerlan has become a trusted technological partner of major companies in the country. In order to achieve these objectives, it has three units of technological specialization, i.e., electronics, information and communication technologies; energy and power electronics; and advanced manufacturing.

University of Deusto

The thesis has been registered at the "Engineering for the Information Society and Sustainable Development" program at the University of Deusto[2] (Figure 1.6). The required courses and seminars have been taken there. Dr. Diego Casado Mansilla has been the supervisor at the University of Deusto.

The University of Deusto was founded in 1886 by the Society of Jesus in Bilbao. The concerns and cultural interest of the Basque Country in having their own university, as well as the interest of the Jesuits in establishing higher studies in some part of the Spanish State coincided in its conception. Bilbao, a seaport and commercial city which was undergoing considerable industrial growth during that era, was chosen as the ideal location. Since then, it has been expanded

[1] https://www.mondragon-corporation.com/

[2] https://www.deusto.es/cs/Satellite/deusto/en/university-deusto?cambioidioma=si

(a) University of Deusto logo.

(b) University of Deusto.

Figure 1.6: University of Deusto.

to have faculties also in Donostia, Vitoria-Gasteiz and Madrid, and currently has around 12,500 students.

Its guidelines are love of wisdom, desire for knowledge and rigour in scientific research and methodologies. Therefore, its main focus is on achieving excellence in research and education. Another objective is to provide the background for free persons, who are responsible citizens and competent professionals, equipped with the knowledge, values and skills needed to take on the commitment to foster learning and transform society.

Pro^{2}Future GmbH

During the PhD, an international research stay has been made at Pro^{2}Future[1] (Figure 1.7) in Graz, Austria, under the supervision of Dr. Simon Mayer, from June to October 2018.

The name of Pro^{2}Future derives from **Pro**ducts and **Pro**duction Systems of the **Future**. It is a COMET Centre funded by the Austrian competence centres funding program COMET (Competence Centres for Excellent Technologies), and it is established alongside the axis

[1] http://www.pro2future.at/start-en/

(a) Pro^{2}Future logo.

(b) Pro^{2}Future, Graz building.

Figure 1.7: Pro^{2}Future.

Upper Austria – Styria. In these two industrially most-active provinces of Austria, they carry out industrial research co-operation at the three sites Linz, Graz and Steyr.

Pro^{2}Future was established in 2017 and its share holders are universities (i.e., Johannes Kepler University Linz and Graz University of Technology) and companies (i.e., Upper Austrian Research GmbH, AVL List GmbH and FRONIUS International GmbH), forming the nucleus of an excellent consortium of company partners and scientific partners that include other company partners.

1.7.2 **Research Activities**

Several research activities have been done during the work of this thesis, in addition to the mandatory courses and seminars at the University of Deusto.

The main research activity has been the international stay for four months at Pro^{2}Future, as previously explained. There, apart from doing

research, an event called *Pro^{2}Future in Progress* took place, where a brief presentation was made about the project.

The results obtained during this thesis have been disseminated in several conferences. A presentation has been made in the following conferences:

- IEEE International Workshop of Electronics, Control, Measurement, Signals and their application to Mechatronics (2017) in San Sebastián, Spain[1], with the title *Towards a lightweight protocol for Industry 4.0: An implementation based benchmark*.
- The 8th International Conference on Ambient Systems, Networks and Technologies (2017) in Madeira, Portugal[2], with the title *Analysis of CoAP Implementations for Industrial Internet of Things: A Survey*.
- The 7th International Conference on the Internet of Things (2017) in Linz, Austria[3], with the title *IEC 61850 meets CoAP: Towards the integration of Smart Grids and IoT standards*.
- The IEEE 23rd International Conference on Emerging Technologies and Factory Automation (2018) in Torino, Italia[4], with the title *Validation of a CoAP to IEC 61850 Mapping and Benchmarking vs HTTP-REST and WS-SOAP*.
- The 8th International Conference on the Internet of Things (2018) in Santa Barbara, California[5], with the title *Enhanced publish/sub-*

[1] http://ecmsm2017.mondragon.edu/en
[2] http://cs-conferences.acadiau.ca/ant-17/
[3] https://iot-conference.org/iot2017/
[4] http://ieee-etfa2018.com/index.php
[5] https://iot-conference.org/iot2018/

scribe in CoAP: Describing advanced subscription mechanisms for the Observe extension.

- The 9th International Conference on the Internet of Things (2019) in Bilbao, Spain[1], with the title *Enabling easy Web of Things compatible device generation using a Model-Driven Engineering approach.*

At the IoT 2017 conference, in addition to the plenary presentation, a presentation was also made as part of the Doctoral Colloquium, with the title *Towards boosting the Industry 4.0 and Smart Grids through IoT-based lightweight reliable communication protocols.* Likewise, at the IoT 2018 conference a poster was also presented in the poster session, titled *Lightweight IoT Communication Protocols for Industry 4.0 and Smart Grids.*

Finally, Ikerlan also organized three *Jornada de Doctorandos*. The participation in the three of them has had different forms. In the first one, as an attendee; for the second one a poster titled *Towards boosting the Industry 4.0 and Smart Grids through IoT-based lightweight communication protocols* was made that a supervisor presented as the event took place during the research stay; and for the last one, a poster with the title *Interoperability in Industry 4.0 systems: model driven automated code generation* was presented.

1.8 Outline

This dissertation consists on seven chapters including this introduction. The following lines present the rest of the chapters:

[1] `https://iot-conference.org/iot2019/`

- **Chapter 2** presents the main technological tools that have been used for the work during this thesis. It is divided in three subsections that categorize the used tools, one for IoT technologies, another one for data and service models and a last one for software engineering concepts.

- **Chapter 3** describes the first contribution carried out during this thesis, i.e., *C1: Analysis of IoT*. This chapter analyzes IoT protocols, with the objective of answering the research questions on objective *O1: Analysis of lightweight communication protocols*. For that, two subcontributions are described, the first one offers a comparison on MQTT and CoAP. Based on the offered comparison, CoAP has been selected to continue the work, and the second subcontribution surveys and compares nine available CoAP implementations. The results obtained in this chapter have been published in [91, 92, 93]

- **Chapter 4** explains contribution *C2: Interoperability through IEC 61850*, which aims to enable interoperability using the IEC 61850 standard. With that objective, a mapping of IEC 61850 to CoAP has been proposed. Then, the said mapping has been implemented and its performance has been compared to other approaches that use HTTP and SOAP. With the content of this chapter, two conference papers [94, 87] and a journal article [85] have been published.

- **Chapter 5** deepens on a downside that CoAP has when mapping the IEC 61850 standard and is pointed out in Chapter 4. This downside is that the subscription mechanism of the Observe extension of CoAP is limited, and IEC 61850 does not entirely

fit. This is why in this chapter, more advanced subscription mechanisms are proposed, to overcome the issues with the IEC 61850 mapping, but that can also be applicable to other use cases. [86] has been published as a conference paper where the new subscription mechanisms are described, and they are applied to the IEC 61850 mapping in a journal article [85].

- **Chapter 6** builds on the work of the previous chapter to propose to use software engineering techniques for developing industrial systems. For applying software engineering techniques, models that are used to represent the system structure are required. This chapter evaluates how to apply two different models. The first one is based on the IEC 61850 used in Chapter 4, and it has allowed to create a tool named TRILATERAL. This has been disseminated on a conference paper [81] and a book chapter [90]. The second used model is based on the WoT TD, which is a specification that aims to describe things using already existing standards aiming to achieve higher interoperability. This work has been published on a conference paper [89], and then extended on a submitted journal paper [88].

- **Chapter 7** concludes this dissertation providing conclusions and future lines for continuing the work, both for each individual contributions and general ones.

If we knew what we were doing it wouldn't be research.

Albert Einstein

I skate to where the puck is going to be, not where it has been.

Wayne Gretzky

2 Background

After the introduction to give the main context to the work carried out during this thesis, in this chapter some technological background is provided. Different technologies, protocols, and tools are presented, to help justify the decisions made during this work. The chapter is divided in different sections, which group the different technologies and tools depending on their context. The sections are IoT related technologies, data and service models, and finally, Software Engineering techniques.

Contents

2.1 Introduction 24

2.2 Internet of Things 25

2.2.1 Communication models 29

2.2.2 Hardware 31

2.2.3 Software 34

2.2.4 Transport Layer Protocols 35
2.2.5 Security Protocols 36
2.2.6 Application Layer Protocols 37
2.2.6.1 Internet Protocols 37
2.2.6.2 Industrial Protocols 38
2.2.6.3 IoT Protocols 40
2.2.6.4 Summary of the protocols 42
2.3 Data and Service models 43
2.3.1 IEC 61850 44
2.3.2 Web of Things 46
2.3.2.1 Organization 47
2.3.2.2 Normative Deliverables 47
2.3.2.3 Informative Deliverables 49
2.4 Software Engineering 50
2.4.1 Software Product Lines 51
2.4.2 Domain Specific Language 51
2.4.3 Model Driven Engineering 52

2.1 Introduction

This chapter provides a general overview of the technologies used in this thesis. Chapter 3 makes the most use of the technologies presented in Section 2.2, although others also use them. The presented technologies are grouped in communication models, hardware platforms, software, and different types of protocols.

Section 2.3 presents two specifications that can be used for obtaining more interoperable systems. These specifications provide models to represent data in different domains. The first one, i.e., IEC 61850 is used in Chapter 4 along with IoT and in Chapter 5 to validate the CoAP

enhancement presented in this thesis. The other presented specification is the Web of Things (WoT), which aims to offer interoperability using already existing web technologies. The WoT has not been designed for a concrete use case like IEC 61850, having a more generic model.

Finally, in Section 2.4 some concepts about software engineering are explained, which are the base for the work presented in Chapter 6. These concepts allow speeding up the development and deployment processes of software products. Chapter 6 explains the contribution using software engineering with the WoT and IEC 61850 models.

2.2 Internet of Things

As previously stated, the IoT is one of the main enablers for Industry 4.0. It allows to all kinds of devices to exchange information, which opens a wide range of monitoring and controlling use cases [222]. The convergence of multiple technologies such as wireless communications or the transition from embedded systems to micro-electromechanic systems, has made the IoT vision to evolve [186]. Traditional areas of embedded systems, wireless sensor networks, automation and others have facilitated the development of the IoT.

The IoT is a network of objects or physical *things* that have embedded electronics, software, sensors and networking capabilities, which enables to collect and exchange data. This allows to receive information or control objects remotely, using already existing networks, creating opportunities for a more direct integration of the physical world in computer systems, allowing bigger efficiency, precision and economical benefit. When the IoT is complemented with sensors and actuators, the technology goes even further, to Cyber-Physical Systems (CPS), which

enables smart networks, smart homes, smart mobility, smart factories, or smart cities [132].

The concept of a network with intelligent devices started on 1982 with the first internet connected appliance. It was a modified Coke vending machine[1] at the Carnegie Mellon University, which informed whether it had soda bottles and if they were cold or not. Mark Weiser's paper *The computer of the 21st Century* [220] and important conferences such as UbiComp and PerCom produced the current vision of the IoT. Reza Raji described the concept on 1994 on the IEEE Spectrum journal as "Control networks [...] must shuttle countless small but frequent packets among a relatively large set of nodes" [167]. Companies such as *at Work* from Microsoft and *NEST* from Novell also presented solutions between 1993 and 1996. However, the idea did not take really take off until Bill Joy presented the device-to-device (D2D) communication model as part of his "Six Webs" framework, presented at the World Economic Forum at Davos at 1999[2].

The first one to use the Internet of Things term was Kevin Ashton in 1999 while working on the Auto-ID Labs at the MIT [8], referring to a global network of Radio Frequency IDentification (RFID) connected devices. According to Ashton, if all the objects and people had their own, unique identifier, computers could manage and inventory them. The tagging could be made thanks to technologies such as NFC, bar codes, QR codes, or digital watermarks.

However, the concept has evolved to more advanced techniques, thanks to improvements in the technology. It is expected that the IoT will offer advanced device connectivity, and systems and services that

[1] https://www.cs.cmu.edu/~coke/history_long.txt

[2] https://www.technologyreview.com/s/404694/etc-bill-joys-six-webs/

go beyond machine-to-machine (M2M) communications covering a wide range of protocols and application domains. It is also expected that the interconnectivity between these embedded devices will be integrated in the automation of all types of application domains, allowing advanced applications as intelligent and wide area networks, such as smart factories and smart cities.

Millions of devices have to be connected to the network for the development of the IoT. Evans [51] and Chase [29], expect 50 billion connected devices by 2020. Jeon [102] goes further and claim that there will be 75 billion connected devices. Columbus [38] offers more conservative numbers for 2020 with 30 billion connected devices, but predicts a huge growth with more than 70 billion devices by 2025 [38]. However, these predictions seem too optimistic as by 2015 the expected growth was bigger that the actual one [51], but it is also true that the number of interconnected devices is growing fast in many application domains such as eHealth [10].

To be able to interconnect the expected billions of devices, the IoT follows the same layered approach as the Internet, thus, different technologies and protocols can coexist. Figure 2.1 shows the different protocol stacks comparing TCP/IP and OSI models.

However, with the objective of achieving more lightweight communications, the IoT uses different protocols and resource representations that have lower overhead, reducing the hardware requirements and power usage. Table 2.1 compares some typical protocol and resource representations used on Web applications and the IoT. As it can be seen in the table, there are many layers with protocols and technologies used in the IoT. Traditionally, Web applications do not have resource limitations, as the devices used for them have powerful CPUs, GBs of RAM

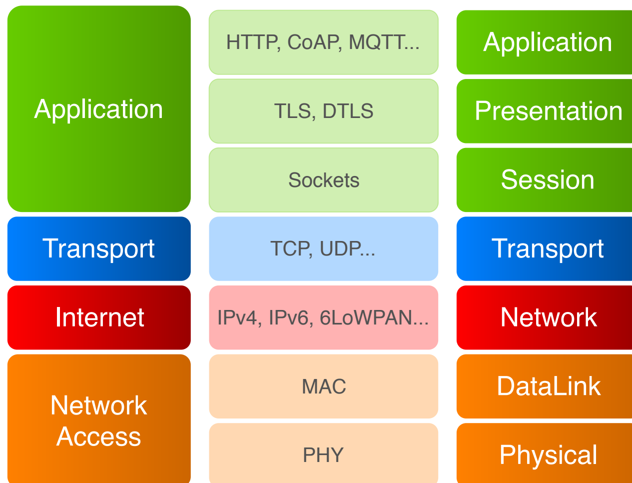


Figure 2.1: Comparison of TCP/IP and OSI protocol stack with examples.

memory, hundreds of GBs if not TBs of ROM. IoT devices, on the other hand, are much more contrained. There are three defined classes for constrained IoT devices, Class 0, Class 1, and Class 2, depending on their capabilities. All the classes have very modest resources [18], with less than 50 KiBs of RAM and 250 KiBs of ROM memory. In order to fit the connectivity capabilities in these type of devices, protocols of all the layers must be taken into account.

The work of this thesis focuses on the application layer and up, although there is also some preliminary analysis of the security layer, and the transport layer is also mentioned in several evaluations as it directly affects the upper layers.

	IoT stack	Web stack
Resource representation	Binary, JSON, CBOR	HTML, XML, JSON
Application	CoAP, MQTT	HTTP
Security	DTLS	TLS
Transport	UDP	TCP
Internet Layer	IPv6	IPv4, IPv6
	6LoWPAN	
Network Access	IEEE 802.15.4 MAC	Ethernet, WiFi
	IEEE 802.15.4 PHY, Radio	

Table 2.1: Different protocols on IoT and Web stacks.

2.2.1 Communication models

In addition to the different protocol layers in the protocol stack, there are four main different communication models between devices in IoT applications, with different features and characteristics.

The first communication model is the **Device-to-Device** model, where the different devices are connected directly to each other, in the same network. Figure 2.2 shows an example of this, an emergency button that connects to a siren on a factory network.

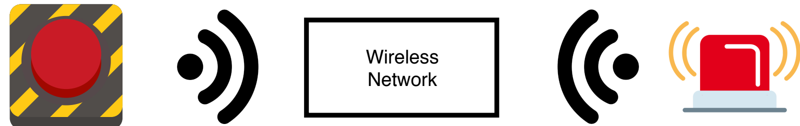


Figure 2.2: Device-to-device communication example.

Another typical communication model is **Device-to-Cloud**. In this case, the devices are directly connected to a cloud, they are not limited to a local network. Figure 2.3 shows an example of this, where two sensors are connected directly to a service provider and upload the data they gathered straight to the cloud.

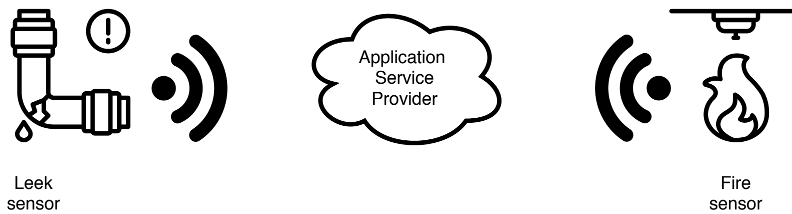


Figure 2.3: Device-to-cloud communication example.

The third communication model is **Device-to-Gateway**, shown in Figure 2.4. The sensors are not directly connected to the service provider, they use a local gateway as an intermediary device.

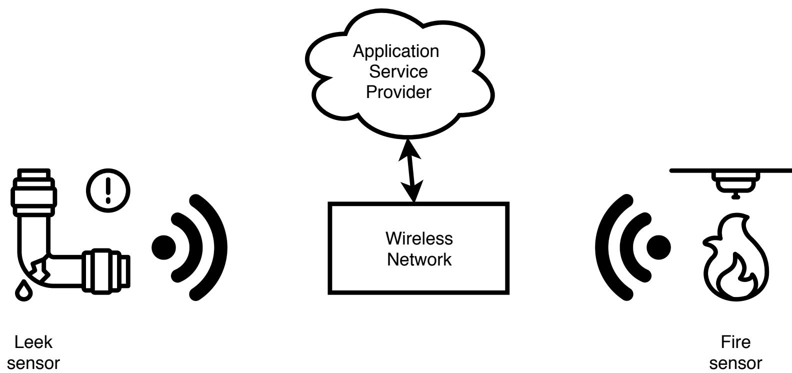


Figure 2.4: Device-to-Gateway communication example.

Last, the **Back-end data sharing** model is presented in Figure 2.5. in this case, the button is connected to the service provider same as the Device-to-Cloud model. The difference is that the service provider can share the information that it receives from the device with other service provider clouds, so the device does not need to send the data to all of them. This is especially useful in IoT because the devices might not have big capabilities, and they might work on batteries.

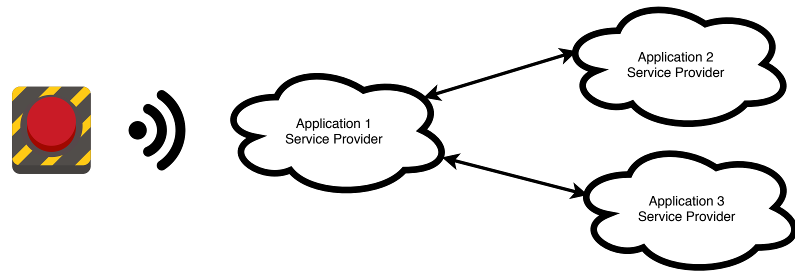


Figure 2.5: Back-end data sharing example.

2.2.2 Hardware

Different hardware platforms have been developed to assist carry out the vision of the IoT. The platforms share similarities in the requirements, such as low price and low energy consumption. This makes them not as powerful as full PC or servers, but they do not have the same computational needs. However, there are different levels of requirements in the price, energy consumption and computational power, hence, different hardware has been designed for different use cases. During the work of this thesis, two hardware platforms have been used.

Raspberry Pi

One of the most important hardware platforms for IoT is the Raspberry Pi[1]. It is a low cost SBC that has had several iterations, which was originally developed by the Raspberry Pi Foundation for educational purposes.

The Raspberry Pi is a good choice for industrial prototyping, as it is very versatile, includes several interfaces (e.g., HDMI, USB, GPIO, etc.) and offers a fast adaptation process due to being able to run

[1] `https://www.raspberrypi.org/`

GNU/Linux and even Windows 10. Even more, it is expanding its use on industrial projects [63].

There are several models of the Raspberry Pi, the newest one being the Raspberry Pi 4 model B. However, the boards that have been used in the contributions in this thesis is the Raspberry Pi 3 model B (see Figure 2.6), as it was the current one when the experimentation was carried out. The Raspberry Pi 3 model B was launched on 2016 and includes a Broadcom BCM2837 processor, a 64-bit quad-core ARMv8 running at 1.2GHz that uses a 64 bit RISC instruction set. The GPU is a Broadcom VideoCore IV and shares the 1 GB RAM with the CPU. The Raspberry Pi 3 model B was the first Raspberry Pi that included WiFi and Bluetooth connectivity, and it also includes 4 USB 2.0, HDMI, DSI interface, RCA connector, 3.5 mm Jack, and a 17 pin GPIO. The included 10/100Mbps RJ-45 port shares the bus with the USB ports.

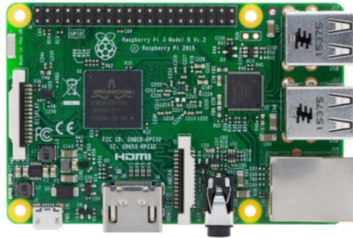


Figure 2.6: Raspberry Pi 3 model B.

STM32F407VGT6 Discovery

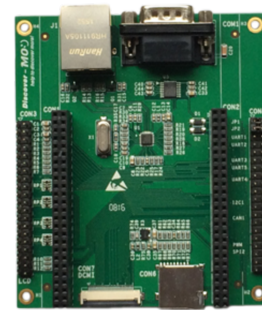
STM32 is a 32 bit microcontroller family by STMicroelectronics. These microcontrollers combine very high performance, real-time capabilities, digital signal processing, and low-power and low-voltage operation, and connectivity, while maintaining full integration and ease of devel-

opment. They are designed for industry applications and aim to be used on small projects and on entire platforms.

For this thesis, a STM32F407VGT6 Discovery [194] has been used (see Figure 2.7a), along with an extension board to add ethernet connectivity (Figure 2.7b), i.e., STM32F4DIS-BB. The STM32F407VGT6 Discovery uses a 32-bit ARM Cortex-M4 microprocessor with FPU core, 1-Mbyte Flash memory, and 192-Kbyte RAM. It also integrates a 3-axis accelerometer, a CS43L22 audio DAC with integrated class D speaker driver, 8 LEDs, (four user LEDs and another four system LEDs), 2 push buttons (one for the user and another one for reset) and a USB OTG micro-AB connector.

This platform is used on the industry as it is a complete solution for a cheap price and the energy consumption is also lower than the Raspberry Pi platform.

(a) STM32F407VGT6 Discovery.



(b) Ethernet expansion board.

Figure 2.7: STM32F407VGT6 Discovery and the Ethernet expansion board.

2.2.3 Software

To use the hardware platforms mentioned above, some software has also been necessary. This software includes the needed Operative Systems (OS) (i.e., Raspbian for Raspberry Pi and FreeRTOS for STM32F407VGT6) and a library to enable Internet connectivity to the STM32F407VGT6 board (i.e., lwIP).

Raspbian

Raspbian[1] is the OS used with the Raspberry Pi board during this thesis. It is a free operative system optimized for Raspberry Pi hardware and based on Debian. Apart from the basic set of programs and utilities to make the Raspberry Pi run, Raspbian also provides thousands of packages, pre-compiled software and an easy installer.

FreeRTOS

The OS used with the STM32F407VGT6 board is FreeRTOS [57]. FreeRTOS is a real time operating system and according to their website[2], the de-facto standard solution for microcontrollers and small microprocessors. It is a free OS and it is supported, with an open source MIT license that allows to protect the intellectual property of the systems developed using it.

lwIP

When using FreeRTOS, the lwIP [55] library has been used for TCP/IP connectivity in the STM32F407VGT6 board. This is a TCP/IP implementation with minimal RAM usage, which makes a perfect choice for

[1] `https://www.raspbian.org/`
[2] `https://www.freertos.org/`

small microcontrollers as it only needs tens of KB RAM and around 40 KB of ROM. It supports IPv4 and IPv6, DHCP, UDP, TCP, DNS among many other networking protocols.

2.2.4 Transport Layer Protocols

Before explaining the application layer protocols used in the IoT, it is important to also describe the protocols that run below. These protocols are TCP and UDP, as presented in the Transport layer in Table 2.1.

TCP

There are two main protocols on the transport layer. One of them is the Transport Control Protocol (TCP) [210], which provides a reliable, ordered and error-checked delivery of bytes, over an IP network. It uses ports to identify application endpoints on a host, and uses those port to open, manage and close sessions between the endpoints. TCP needs an open connection, so before sending any streams, it needs to open a session using a handshake process. When a communication ends, it closes the session with another different handshake process. One of the advantages of TCP is that it is able to slice big data streams if they are too big to fit in TCP segment. TCP recovers the entire stream when it arrives to the destination endpoint. TCP is used on some major internet applications, such as de WWW, email, SSH or FTP.

UDP

The other main transport layer protocol is the User Datagram Protocol (UDP) [211]. UDP packets are called datagrams, and same as TCP, it uses ports to identify application endpoints, but it does not keep any type of session. This makes UDP faster and more lightweight, as it does

not need to open, manage or close the session, but it has the downside that it does not guarantee reliability and ordered message reception. It is also not able to send big streams of data slicing it like TCP. Several internet applications use UDP as the transport layer, i.e., DNS, and voice and video transmission.

2.2.5 Security Protocols

As shown in Table 2.1, there can be security protocols between the transport and application layers, which secure the information to be sent from one endpoint to another. This protocols are TLS for TCP communications and DTLS for UDP communications.

TLS

Security protocols can be added between the transport protocol and the main application protocol. The Transport Layer Security (TLS) [45] is the protocol that can be used to secure connections that run on top of TCP. TLS is the successor of the Secure Sockets Layer (SSL), and uses several cryptographic protocols for encoding the message. TLS's main objective is to provide privacy and data integrity in communications, offering privacy, authentication and reliability. TLS can use symmetric or asymmetric cryptography, which are further explained on Chapter 3.

DTLS

Due to UDP not being able to guarantee message reception and reception order, TLS can not be used on top of UDP. This is why the Datagram Transport Layer Security (DTLS) [171] was described by the IETF, which based on TCP, it adapts it to fit with the characteristics of UDP. The features of DTLS are further described in Chapter 3.

2.2.6 Application Layer Protocols

Communication protocols are necessary in the application layer to enable the communication between different devices, as presented in Table 2.1. Using proprietary or custom protocols makes the system and devices siloed, hence using open or well-known protocols is important. Being IoT a continuation of the Internet and industrial M2M communications, historically, protocols from both of them have been applied to IoT devices. However, there are also new communication protocols specifically designed and developed for IoT, taking into account requirements and limitations that IoT devices might have.

2.2.6.1 Internet Protocols

There are several internet protocols that have historically been applied in IoT domains, such as HTTP, SOAP, and XMPP.

HTTP

The most predominant application layer on the World Wide Web (WWW) is the HyperText Transfer Protocol (HTTP) [144]. It is often used following the REpresentational State Transfer (REST) [52] architectural style. This protocol uses the client-server model and it is stateless, making each request self-contained. Resources on the web are represented by Uniform Resource Identifiers (URI)s, and using methods called verbs (e.g., GET, POST, PUT, etc.), a client can interact with the resources. With the purpose of interacting with the resources, the representation of the resource has to be transmitted and there are several formats for that, such as HTML, XML, or JSON. The resources can be discovered at run time thanks to links in the received responses.

This allows clients to navigate through the resources without the need to know the hypermedia structure of the resources in advance.

SOAP Web-Services

SOAP Web-Services [190] encapsulates client requests and service response objects using the Simple Object Access Protocol (SOAP). The encapsulated objects are then transmitted using HTTP POST requests. The services, their functionality, and the interfaces are described using the Web Services Description Language (WSDL). Complementary standards from the Web-Services family can be used to define addressing schemes, security parameters, and discovery services. Some work has also been carried out to adapt SOAP Web-Services for resource constrained devices, e.g., Priyantha et al. [165].

XMPP

The Extensible Messaging and Presence Protocol (XMPP) [223] is an open protocol originally created for instant messaging systems. It runs on top of TCP and can use both publish-subscribe and client-server communication models. The information to be exchanged is coded using XML. However, this protocol does not include QoS options, hence, it is not very practical for M2M communications.

2.2.6.2 Industrial Protocols

As previously mentioned, the IoT is also an evolution of industrial M2M communications, and as such, some industrial communication protocols have also been used in IoT domains, as a natural evolution. Some of these protocols are DDS, AMQP, JMS and CORBA.

DDS

Data Distribution Service for real-time systems (DDS) [53] is an standard defined by the Object Management Group (OMG) with the objective of enabling data exchange in a scalable, real time, dependable, high performance, and interoperable way. DDS aims to fulfill the requirements of financial trading, air control, smart network management, and other big data applications. DDS uses the publish-subscribe communication model and can run on top of different transport layer applications, TCP and UDP among others. It offers several levels of QoS that can be configured by the users through parameters.

AMQP

Advanced Message Queuing Protocol (AMQP) [5] is an open protocol on top of TCP that offers asynchronous communication. The defining features of AMQP are message orientation, queuing, point-to-point and publish-subscribe routing, accuracy, and security. It includes three levels of QoS, i.e., at most once, at least one, and exactly once. It can use TLS or Simple Authentication and Security Layer (SASL) for securing the communications.

JMS

The Java Message Service (JMS) [103] is Sun Microsystems' solution for using message queues. It allows to create, send, receive and read messages using the Java 2 platform. The communication can be carried out synchronous and asynchronously.

CORBA

Another industrial communication protocol is Common Object Request Broker Architecture (CORBA) [150]. CORBA is not exactly a communication protocol, it is a middleware that enables to develop distributed applications in heterogeneous systems. It defines an Interface Definition Language (IDL) to represent the interfaces of internal objects to the outside.

2.2.6.3 IoT Protocols

Both regular Internet and industrial communication protocols have been designed for devices with high capabilities. With electronics lowering the price and size, the IoT allows adding electronics to many non electronic objects, enabling information exchange. However, these new electronics embedded on devices may have severe resource constraints, not only on the capabilities of the electronics, but also on energy consumption. This is why new protocols have been designed and proposed, with the aim to drastically reduce the needed resources and energy consumption. Nowadays, there are two main protocols, MQTT and CoAP.

MQTT

Message Queuing Telemetry Transport [12] is a lightweight protocol that usually runs on top of TCP. It uses the publish-subscribe communication model and it is designed for bandwidth communications. It uses an intermediary device called broker that receives the messages the publishing client sends and forwards the messages to all the subscribed clients. It uses topics to define to what resource a client subscribes.

Similar to AMQP, MQTT also has three QoS, i.e., fire and forget, delivered at least once and delivered exactly once. Although MQTT runs on top of TCP, it has been designed to have very low overhead compared to other protocols. For security, MQTT can use passwords or it can also add a TLS layer between MQTT and TCP.

MQTT has a variant named MQTT for Sensor Networks (MQTT-SN) [191] that targets embedded devices. It was originally developed to run using ZigBee instead of TCP/IP stack, and it can also run on top of Bluetooth and UDP.

CoAP

With the goal of offering a RESTful protocol for constrained devices, the IETF designed the Constrained Application Protocol (CoAP) [34]. It is similar to HTTP, with a client-server communication model and uses a subset of HTTP verbs, i.e., GET, PUT, POST and DELETE, and URIs. The main difference is that it runs on top of UDP instead of TCP, hence, for securing the communications, DTLS can be used. As it runs on top of UDP and there is no delivery guarantee on the transport layer, CoAP includes CONfirmable (CON) and NON-confirmable (NON) messages, with the objective of offering a basic QoS. The former requires an acknowledgment while the latter does not.

The IETF also described an extension for CoAP, named Observing Resources in the Constrained Application Protocol (CoAP) [68], which allows to use the publish-subscribe communication in CoAP. A client interested on getting notifications can subscribe to a resource sending an extended GET request. When a server gets such a request, it adds the client to the subscriber list and when the resource changes it sends a notification to all the subscribed clients. The server decides what a

change in the resource is, e.g., a timeout for sending periodic notifications, a change on the value of a variable, or a variable that changes its value out of a range.

2.2.6.4 Summary of the protocols

As describe in the previous paragraphs, there are many communication protocols used for IoT use cases. Table 2.2 summarizes some characteristics they offer.

Protocol	Transport	QoS	Communication model	Security	Performance
HTTP	TCP	No	Client-server	TLS	-
SOAP	HTTP	Yes	Client-server	HTTPS	-
XMPP	TCP	No	Client-server Publish-subscribe	TLS/SSL	-
DDS	TCP/UDP	Yes	Publish-subscribe	TLS/SSL	Microseconds
AMQP	TCP	Yes	Publish-subscribe	TLS/SSL	Seconds
JMS	TCP/RMI/ HTTP...	Yes	Point-to-Point Publish-subscribe	SSL/HTTPS	-
MQTT	TCP	Yes	Publish-subscribe	TLS/SSL	Seconds
MQTT-SN	UDP/BT/ ZigBee	Yes	Publish-subscribe	DTLS	-
CoAP	UDP	Yes	Client-server Publish-subscribe	DTLS	Miliseconds

* Although CORBA has also been mentioned in the description of the protocols, it is not exactly an application layer protocol, hence has not been included in this table.

Table 2.2: Comparison of IoT protocols.

It can be seen that most of them use TCP as the transport layer in spite of being slower that UDP. However, it has the advantage of offering delivery guarantee and ordered reception. In addition, the publish-subscribe model is more present in the table. This is due to the fact that for monitoring applications this model is more appropriate, while control use cases fit better with a client-server model. Another

important aspect is the QoS, which most of the analyzed protocols offer. This is an important aspect for the IoT, as depending on the use case, message loss can be tolerated to ensure a fast delivery and configuring this brings an important benefit. Security is also important on the IoT and all the analyzed protocols offer some kind of ciphering option.

For the work carried out in this thesis, MQTT and mostly CoAP have been used. They are the only pure IoT communication protocols and offer very different features. Both of them are gaining momentum, meaning that many libraries are available and also support is provided. They are more lightweight than the other analyzed protocols, which makes them more suitable for IoT applications.

The first contribution in Chapter 3 makes a comparison on both protocols, in order to select one of them for continuing the work of this thesis, while the second contribution of Chapter 3 analyzes nine implementations of the selected protocols to select one for the implementations of the following chapters.

2.3 Data and Service models

The interoperability between different devices, systems, and protocols can be achieved using data and service models. These models allow to have a common language even for different domains. During the work carried out in this thesis, two models have been used, the one defined by the IEC 61850 standard (used in Chapter 4 and Chapter 6) and the one that the WoT proposes (used in Chapter 6).

2.3.1 IEC 61850

Standards are needed to ensure the interoperability and integration of different devices. They also allow an intuitive modeling of the devices and data that are part of the system, enabling a fast and convenient communication, lowering the costs. In this context, the International Electrotechnical Commission (IEC) defined the IEC 61850 standard [72], as part of its architecture for electrical power systems. This standard was originally designed for modeling, controlling and monitoring electrical substations and their data, along with their communication.

The main computational unit on an electrical substation is the Intelligent Electronic Device (IED) which refers to any device (or *thing* in the IoT vocabulary) in a substation with processing capabilities that manages and exchanges information and commands with other entities inside or outside the substation, e.g., control centers. The design of IEC 61850 was made with the following requirements in mind:

- High-speed IED to IED communication,
- Networkable throughout the utility enterprise,
- High-availability,
- Guaranteed delivery times,
- Standard based,
- Multi-vendor interoperability,
- Support for heterogeneous samples data,
- Support for File Transfer,

- Auto-configurable/configuration support, and
- Support for security.

Although IEC 61850 was initially designed for electrical substation, it has evolved to also support the monitoring, control and protection of Smart Grid applications. To do that, there are three important aspects in the specification [74], summarized in Table 2.3:

System Aspects		**Data Models**	
Part 1:	Introduction and Overview	Part 7-3:	Common Data Classes
Part 2:	Glossary	Part 7-4:	Compatible Logical Node Classes & Data Classes
Part 3:	General Requirements	Part 7-10:	Communication Networks and Systems in Power Utility Automation – Requirements for Web-Based & Structured Access to the IEC 61850 Information Models
Part 4:	System & Project Management		
Part 5:	Communication Requirements for Functions and Device Models		
Configuration		**Specific Communication Service Mapping (SCSM)**	
Part 6:	Configuration Language for Communication in Electrical Substations Related to IEDs	Part 8-1:	Mappings to MMS (ISO/IEC9506-1 and ISO/IEC 9506-2)
		Part 9-1:	Sampled Values over Serial Unidirectional Multidrop Point to Point Link
Abstract Communications		Part 9-2:	Sampled Values over ISO/IEC 802-3
Part 7-1:	Principles and Models	Part 9-3:	Precision Time Protocol Profile for Power Utility Automation (IEEE C37.238.2011)
Part 7-2:	Abstract Communication Service Interface	**Testing**	
		Part 10:	Conformance Testing

Table 2.3: Specification of the different parts of the IEC 61850 within the different sections of the standard.

- An abstract and extensible *Information Model* for defining a virtualization of physical devices and functionalities. Also to represent some control functions associated to the real world, such as logging, reporting, etc.

- A set of services to interact with the model, with abstract service interfaces for each specific class in the information model, called *Abstract Communication Service Interface (ACSI)*.

- A set of mappings of these services to specific communication protocols.

At Ikerlan, this standard has also been successfully used for energy related projects but outside of electrical substation domains such as catenary-free trams, smart elevator, and wind farms, as explained in Chapter 6.

2.3.2 Web of Things

The next step for the IoT is the Web of Things (WoT), which aims to add a new layer on top of IoT with the objective of enable interoperability and avoid fragmentation. The WoT provides a description of the *things*, which includes how the different services of the *thing* can be accessed, as seen in Figure 2.8. This way, a developer does not need to know all the different IoT protocols, technologies and standards, as the WoT offers a way to combine them. The WoT is being developed at the World Wide Web Consortium (W3C)[1].

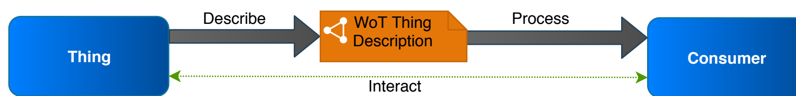


Figure 2.8: Consumer-Thing interaction: a thing exposes its features using the TD.

[1] `https://www.w3.org/`

2.3.2.1 **Organization**

There are two main groups at the W3C working on the WoT. On the one hand, there is the WoT Interest Group (IG)[1], which focuses on seeking the collaboration with external organizations and industrial companies. The main goal of the IG is to explore the ideas before the standardization process starts.

On the other hand, there is the WoT Working Group (WG)[2], which works to provide documents to fulfill the standardization process of the WoT, based on the work of the IG. They already published two normative deliverables as W3C recommendations, as can be seen in Table 2.4. They are still working on another three informative deliverables that have not met the expected completion date, as can be seen in Figure 2.5.

Document	**First Public Working Draft**	**Candidate Recommendation**	**W3C Recommendation**
WoT Architecture	14 September 2017	16 May 2019	1 August 2019
WoT Thing Description	14 September 2017	16 May 2019	1 August 2019

Table 2.4: WoT WG Normative deliverables.

2.3.2.2 **Normative Deliverables**

As summarized in Table 2.4, there are two normative deliverables, i.e., Web of Things (WoT) Architecture and Web of Things (WoT) Thing Description.

[1] https://www.w3.org/WoT/IG/
[2] https://www.w3.org/WoT/WG/

Document	First Public Note	Expected completion
WoT Scripting API	14 September 2017	Jun 2019
WoT Binding Templates	5 April 2018	Jun 2019
WoT Security and Privacy Considerations	14 December 2017	Jun 2019

Table 2.5: WoT WG Informative deliverables.

Web of Things (WoT) Architecture

The first normative deliverable of the WoT WG is the Web of Things (WoT) Architecture document[1].

This document provides some normative sections and other that are non normative, to provide further information, examples, and advice. The normative sections include the requirements for the WoT, a proposed architecture and the definition of the building blocks for that architecture. The blocks to build the WoT are briefly explained, as they are further expanded in the other documents of the WoT WG.

The non normative sections describe several use cases and offer advice for the implementation of servient (server + client) devices and their deployment. Finally, some security and privacy considerations are also provided.

Web of Things (WoT) Thing Description

The other normative deliverable of the WoT WG is the Web of Things (WoT) Thing Description[2], which describes a formal model and a common representation for describing *things*. The Thing Description

[1] `https://www.w3.org/TR/wot-architecture/`
[2] `https://www.w3.org/TR/wot-thing-description/`

(TD) describes the metadata and interfaces of *things*. The default encoding for a TD is JSON, but it also allows JSON-LD.

The TD includes the attributes of the servient, along with interaction affordances, which are the ways that a client or server can interact with the WoT servient. The interaction affordances can be properties, actions and events.

This document also provides some non normative examples, extensions, and security and privacy considerations, along with some transformations and ways to validate the generated TDs.

2.3.2.3 Informative Deliverables

Apart from the normative deliverables, the WoT WG is also working on the three informative deliverables presented in Table 2.5, i.e., WoT Scripting API, WoT Binding Templates, and WoT Security and Privacy Considerations.

WoT Scripting API

The first informative deliverable being developed by the WoT WG is the WoT Scripting API document[1]. This document describes a programming interface to allow scripts run on a *thing*. It includes the WoT object and two main interfaces, the ConsumedThing and ExposedThing interfaces. The former allows client interactions while the latter allows to define request handlers for *things*.

[1] https://www.w3.org/TR/wot-scripting-api/

WoT Binding Templates

The second informative deliverable is the WoT Protocol Binding Templates[1]. This document offers additional vocabulary and design patterns to use in the TD, with the objective of exposing *things* using different protocols. It describes how to bind the different interaction affordances to communication protocol methods. In addition, this document also provides some examples of TDs bound to different protocols.

WoT Security and Privacy Considerations

The last deliverable by the WoT WG is the WoT Security and Privacy Considerations document[2]. This document provides guidance for adding security and privacy on the WoT. In order to do so, general security requirements are described using a threat mode that defines possible attacks a WoT system might suffer. However, this document does not limit the security mechanisms a WoT network can use, it just provides information on possible attacks and mitigations.

2.4 Software Engineering

After explaining the base specifications that describe the models used in this thesis, this section presents some software engineering concepts that have been used for the work explained in Chapter 6. These concepts have been applied to the IoT with the purpose of improving the software generation process.

[1] `https://www.w3.org/TR/2018/NOTE-wot-binding-templates-20180405/`

[2] `https://www.w3.org/TR/wot-security/`

2.4.1 Software Product Lines

Software Product Lines (SPL) are one of the main tools for software engineering. SPLs allow to reuse software elements in different projects. In order to do that, SPLs include two differentiated parts: on the one hand, domain engineering, where reusable assets are developed and on the other hand, the application engineering, where different software products are generated selecting the reusable assets and resolving the variable parts [139].

2.4.2 Domain Specific Language

Domain Specific Languages (DSL) are programming languages that offer expressive power focused on particular problem domains, using appropriate notations and abstractions [214]. This way, a DSL can encapsulate the knowledge of the specific domain it applies to and contributes to the specification of more expressive models [135]. Well known examples of DSLs are *MatLab*, *SQL* or spreadsheet formulas, which with them, stakeholders can participate in the development of software solutions for their problems [44]. With this approach, DSLs can be developed to get the domain experts in the problem space to the programmers in the solution space [208]. In the case of this thesis, in Chapter 6, two DSLs are presented, one designed for use cases that implement the IEC 61850 standard and another one that is based on the WoT TD. This way, a system can be implemented using one of the described DSLs and improve the development of the software.

2.4.3 **Model Driven Engineering**

Model Driven Engineering (MDE) [178] is an approach that aims to alleviate the complexity of the platforms to be developed. To achieve that, MDE uses models with the with the goal of raising the abstraction level and be able to automate the software development processes. According to Selic [180], abstraction is one of the main techniques to cope with the complexity of the applications, while automation allows to boost the productivity and quality. To help express models at the different abstraction layers, modeling languages can be used, using modeling techniques [14]. These models are defined using the mentioned metamodels, which specify the semantics of a model used for specifying models, i.e., the metamodel. MDE uses DSLs to formalize metamodels, which is the base to create specific models including their structure, behavior and requirements of an application on a specific domain. The formalization results on a model that is then analyzed using transformation engines and generators to synthesize artifacts that include source code, simulation inputs, deployment descriptions, etc. MDE has already been successfully used for developing highly complex and critical embedded systems [175].

Getting information off the Internet is like taking a drink from a fire hydrant.

Mitch Kapor

I do not fear computers. I fear the lack of them.

Isaac Asimov

3

Lightweight Communication Protocols

After the review of the related literature, this chapter describes the first two subcontributions carried out during this PhD. These subcontributions are related with the analysis of lightweight communication protocols. The first one compares the performance of the MQTT and CoAP IoT protocols in a STM32 family board, while the second one focuses on a comparison of nine different CoAP implementations and their performance on Raspberry Pi boards.

Contents

3.1 Introduction 54
3.2 Related Work 58
3.3 Protocol Comparison 65

3.3.1 Analyzed Protocols 66
3.3.1.1 MQTT 66
3.3.1.2 CoAP 69
3.3.2 Experiment Setup 72
3.3.2.1 Scenario Definition 72
3.3.2.2 Architecture Design 75
3.3.2.3 Hardware and Software 76
3.3.3 Results 77
3.3.3.1 Overhead 77
3.3.3.2 Scalability 79
3.3.3.3 Latency 83
3.4 Comparison of CoAP Libraries 85
3.4.1 CoAP 86
3.4.2 CoAP implementations 91
3.4.3 Security in CoAP: DTLS 94
3.4.4 Feature Comparison 97
3.4.5 Security Feature Comparison 101
3.4.6 Experiment Setup 103
3.4.7 Results 105
3.4.7.1 Compatibility 105
3.4.7.2 Latency 106
3.4.7.3 Resource Consumption 107
3.5 Conclusion 111

3.1 Introduction

The next industrial revolution, generally referred to as Industry 4.0 is presently taking place and is expected to radically change the current industrial environment by the incorporation of CPSs that connect the

three main axis of an industrial plant: horizontal, vertical and end-to-end [96]. These CPSs represent a digital image of physical objects, not only on industrial domains, from clothing to cars as well as factories' machines. In order to do that, it is envisioned a rapid transformation in the design, operation and service of manufacturing systems, where machines, sensors and actuators are interconnected inside factories aiming to enable spontaneous collaboration, monitoring and control. The interconnection of CPSs, which can be one of the definitions of the IoT, presents a new challenge in terms of communication. Historically, CPSs use regular Internet protocols (e.g., SOAP or HTTP) for communication, but they have inherent disadvantages when implemented in resource constrained devices: large footprint, CPU usage, high memory and energy consumption, etc.

Several communication protocols exist for different domains. However, neither of the protocols originally designed for industrial environments nor those designed for the Internet offer the characteristics for the ever-growing amount of small devices that need to be connected. This is why in recent years, new IoT protocols have emerged, not only for IoT but also for the IIoT, which try to overcome the issues of heavyweight protocols, such as MQTT [12], CoAP [34], AMQP [5], DDS [53], JMS [103], XMPP [223], MQTT-SN [191], or Websockets [219]. Two of the main exponents of such protocols are CoAP and MQTT. Both of them have been designed to have a small footprint and overhead, but they present different features on communication model and Quality of Service (QoS) options, which makes each of them more suitable for a specific domain. Therefore, it is important to identify the right metrics to adopt the most appropriate IoT protocol for each project.

The first subcontribution in this chapter, presented in Section 3.3, aims to assist system designers to select between MQTT and CoAP for their system. MQTT has already acquired an important share of the IoT data-centric communications [58]. CoAP, on the other hand, falls behind, mainly because it was designed later. However, despite not being so mature, CoAP has been designed following well-established architectures and with the focus on reducing latency and bandwidth requirements. However, both protocols are getting some momentum on the industry and represent different communication paradigms (MQTT follows a push based publish-subscribe model over TCP while CoAP follows a pull based client-server model over UDP).

Being so lightweight, MQTT and CoAP can be used on resource constrained devices to reduce deployment and functional costs with cheaper and less energy consuming hardware. ARM is working towards this downwards scalability, aiming to lower costs both on production and operation for constrained devices, with platforms such as their Cortex-M series. This is why in order to help developers select the best IoT protocol for their system, the first contribution presents a theoretical and empirical comparison of MQTT and CoAP in terms of network overhead and latency running on an STM32 family board, which uses an ARM Cortex-M series processor. Also, a quantitative comparison is conducted with a demo implementation in constrained hardware. The achieved results provide useful metrics and guidelines to help designers to choose the appropriate schemes for their implementations.

Analyzing the results of the first subcontribution, a comparison of nine CoAP implementations is presented in the second subcontribution of this chapter. Aside from technical advantages of CoAP, a non-technical one is that it is very similar to HTTP, a popular Internet

protocol that many developers have the skills for, which makes the transition from HTTP to CoAP simple for said developers.

There are many implementations available for CoAP, each of these with its own particular features and requirements. Therefore, it is important to choose the CoAP implementation that suits better to the specific requirements of each application. In this chapter, on Section 3.4, a feature and empirical comparison of nine selected open source CoAP implementations is presented. To do so, first a survey is carried out on current CoAP implementations, and they are compared in terms of built-in core, extensions, target platform, programming language and compatibility. Then, CoAP server and clients are implemented using these libraries and their performance is tested in terms of latency, memory and CPU consumption in a real testbed deployed on an industrial scenario, in order to help in adopting a decision criterion for similar deployments. To do so, even though the selected implementations target different hardware, all of them can be run on the Raspberry Pi platform[1], hence, an industrial prototype that makes use of this platform has been selected to carry out the experiments.

On industrial applications, securing IoT devices is of pivotal importance due to the huge number of connected devices. Having a secure channel of communication is even more important for IoT environments than regular Internet, since IoT devices affect the physical world and exchange personal private data. No industrial company would accept a communication protocol that does not provide security features, hence, these cannot be left out of an implementation analysis. There are several kinds of attacks as explained by AlShahwan et al. [4], but two concepts arise: authentication and privacy. On the one hand, as demonstrated

[1] https://www.raspberrypi.org/

on the 2016 Dyn attack [69], a Distributed Denial of Service (DDoS) attack can be very effective using small, connected devices. This attack was carried out with the Mirai botnet, a malware that targets IoT devices [204]. To achieve this attack, the firmware of the devices has to be changed, which should be avoidable with proper authentication mechanisms. On the other hand, the importance of privacy is more straightforward to understand: from video streams from home cameras to bank account information for self-refilling fridges, private data is constantly being sent. As explained in Chapter 2, DTLS can be used to add security features to CoAP communications. As the existing DTLS libraries are still not mature enough to carry out a performance analysis on them, a preliminary theoretical analysis of the ongoing security development of these libraries is also presented in the second subcontribution.

The rest of the chapter is organized as follows. First, the related literature is analyzed in Section 3.2. Then, the two subcontributions are presented, on the one hand, a comparison of CoAP and MQTT on an ARM Cortex-M series platform on Section 3.3, and on the other hand a comparison of CoAP implementations on Section 3.4. Finally, some conclusions are presented in Section 3.5.

3.2 Related Work

As explained in the previous section, IoT protocols are growing in use, therefore, there are several pieces of research analyzing different aspects of their use, with diverse use case and application domains. The research carried out in this regard can be grouped in three wide groups: comparison of communication protocols, comparison of implementations of the same protocol, and the security aspects.

The first group, i.e., the comparison of the performance of different protocols allows to select the best lightweight protocol for each project. In the analyzed works, some aim to compare a bigger number of protocols than others. Talaminos-Barroso et al. [197] compare DDS, MQTT, CoAP, JMS, AMQP, and XMPP, implementing a tool for an eHealth application that allows to use all the mentioned protocols. Once implemented, they benchmarked the CPU and memory usage, bandwidth consumption, latency and jitter of each protocol, running the tool on virtual machines. Mun et al. [142] selected a different set of protocols, using CoAP, MQTT, MQTT-SN, Web Sockets, and TCP in their work. The main goal of their research is to ease the selection of the best protocol in each application. In order to do that, Mun et al., measure the performance, the energy efficiency, and the memory and CPU usage running on Amazon Web Services and Raspberry Pi's. Chen et al. [30] compare MQTT, CoAP, DDS and a custom protocol running over UDP using a network emulator. They configure different parameters for package loss and latency and measure the consumed bandwidth, latency, and package loss. Chen et al. also use Raspberry Pi-s, along with a laptop and an Arduino Uno for their measurements. In [109], the authors compare MQTT, CoAP, XMPP and MQTT over Web Sockets (MQTT-WS) on a smart parking system use case. In this work, the authors compare the mean response time of each of the protocols, using PCs as the servers. Çorak et al. [39] also compare CoAP, MQTT and XMPP, in this case, using an Intel Galileo Gen 2 board and a PC. In their analysis, the measured parameters are the packet creation time and the packet transmission time. Gündoğan et al. [65] analyze the performance of CoAP and MQTT, along with the new, lightweight protocols of the Named Data Networking (NDN) family.

The authors compare the overhead, ROM, RAM and CPU consumption, latency, and energy consumption. Another analysis that cover several IoT protocols is the one carried out by Anusha et al. [6], where the authors analyze AMQP, MQTT, CoAP, XMPP and DDS using Contiki, and measure the packet loss rate, message size, bandwidth consumption and latency.

Other works aim to compare just two protocols. Being two of the protocols with most momentum, several pieces of research compare CoAP and MQTT. De Caro et al. [42] use Smartphone implementations of CoAP and MQTT and compare latency, bandwidth usage and package loss ratio, configuring the network with different QoS and network loss ratios. Thangavel et al. [201] present a common middleware and measure latency and bandwidth consumption, using Beaglebone boards for their experiments. Ouakasse and Rakrak [153] also carry out a performance analysis of CoAP and MQTT, using the CORE network emulator. For their experiments, the authors defined two scenarios, i.e., low data traffic and high data traffic, and measured the throughput, latency, and inter-arrival time. Another work comparing CoAP and MQTT is the one carried out by Larmo et al. [123]. In this work, the authors use an internal network simulator at Ericson, and run the simulations using Bluetooth Low Energy (BLE) and low power Wi-Fi under CoAP and MQTT. Larmo et al. focus on the packet delay and the power consumption. In [124], Larmo et al. changed the underlying protocols and analyze the performance of CoAP and MQTT over NarrowBand IoT (NB-IoT) using once again the Ericson internal simulator. In this case, the authors focus on the throughput, service availability and coverage. In [11], the authors compare CoAP and MQTT using

PCs running the Ubuntu OS, and measure the bytes that are sent with different packet loss ratios and the consumed power.

Another logical combination is to compare CoAP with HTTP, as they share many similarities. Targeting different environments, they both have similar structure, and they even share many request and response codes, especially the most common ones. Ludovici et al. [134] present their own CoAP implementation for TinyOS called TinyCoAP, to then compare it against the original TinyOS CoAP implementation (CoAPBlip), along with HTTP over both TCP and UDP. They carry out the comparison on TelosB motes, measuring latency, and memory and energy consumption. Colitti et al. [37] also use motes, Tmote Sky and Zolertia Z1 and compare HTTP and CoAP in terms of response time and energy consumption. Kuladinithi et al. [120] present libcoap, their own implementation for CoAP, and they port it to Contiki and TinyOS, comparing its performance against HTTP. Elmangoush et al. [48] summarize CoAP, HTTP, MQTT, and AMQP and their features, but they only compare the first two. They measure bandwidth per request interval time, the response time per request interval time, and the response time for different payload sizes using the OpenMTC platform.

Table 3.1 summarizes the presented works, showing which protocols each one compares, what metrics are measured and which environment they are experimented on.

In this thesis, the work has been mainly carried out using CoAP, hence, for the second group of the analyzed related work, i.e., comparison of different implementations of the same protocol, different pieces of research comparing different implementations of CoAP have been analyzed.

Work	Analyzed Protocols												Environment	Metrics
	HTTP	HTTP/UDP	MQTT	DDS	JMS	AMQP	XMPP	MQTT-SN	C. UDP	TCP	MQTT-WS	NDN		
[134]	✓	✓											Telos B motes	Latency, memory, energy
[37]	✓												Tmote Sky and Zolertia Z1	Latency, energy
[120]	✓												Contiki and TinyOS	Performance
[48]	✓												OpenMTC	Bandwidth, latency, latency/payload
[42]			✓										Smartphone	Latency, bandwidth, package loss ratio
[201]			✓										Middleware	Latency, bandwidth
[153]			✓										CORE network emulator	Throughput, latency, inter-arrival time
[123]			✓										Simulator BLE & lpWiFi	Packet delay, energy
[124]			✓										Simulator NB-IoT	Throughput, availability, coverage
[11]			✓										PC	Bytes sent, energy
[197]			✓	✓	✓	✓	✓						eHealth	CPU, memory, bandwidth, latency, jitter
[39]			✓				✓						Intel Galileo 2 & PC	Packet creation & transmission time
[142]			✓					✓		✓			–	Performance, energy, memory, CPU
[30]			✓	✓					✓				–	Bandwidth, latency, package loss
[109]			✓				✓				✓		Smart Parking System	Mean response time
[65]			✓									✓	ARM Cortex-M3	Overhead, ROM, RAM, CPU, latency, energy
[6]			✓	✓		✓	✓						Contiki	Loss rate, size, bandwidth, latency

Table 3.1: CoAP and other protocols comparisons.

During the development of CoAP's specification, some work analyzed different implementations. Lerche et al. [127] present a list of available implementations and summarize the results of the first ETSI CoAP Plugtest [164]. The results include an interoperability report, but the authors do not present any performance analysis. Villaverde et al. [215] analyze CoAP theoretically, short after the first draft of CoAP's specification was presented. They analyze some of the available implementations, however, they do not carry out a performance analysis neither, they present conclusions form surveyed researches. Both these articles present a list of available implementations, but since they were published, CoAP's standardization was completed and new libraries have appeared, while others have not been updated to the final version, so they are not up to date.

Another relative survey is Kruger et al. [119]. They present a benchmark using different hardware, i.e., Raspberry Pi, BeagleBone and BeagleBone Black with the same CoAP implementation. Class 4 and class 10 SD cards were used, with running, off and uninstalled GUI. Kruger et al. measured latency, bandwidth and CPU and memory usage in the loopback interface. They also compare the latency on a TelosB mote server with a BeagleBone gateway and a laptop client.

After the work carried out in this chapter of the thesis, and published the comparison of CoAP implementations in [92] and [93], Wall et al. [217], implemented the CoAP observe extension in three different CoAP libraries, i.e., Californium, jCoAP, and libcoap. The authors tested and compared the performance the implementations on three different hardware platforms, i.e., Raspberry Pi 1, ZedBoard and Raspberry Pi 3. However, the test were carried out using the same library

for server and client, they were not mixed. The authors analyzed the packet build time, the socket time and the packet read time.

Table 3.2 summarizes the works comparing the performance of CoAP.

Work	Platform / Use Case	Metrics	Downsides
[127]	1st Interoperability plugtest and performance analysis	Performance based on other papers	Old implementations, no performance analysis
[215]	1st CoAP draft implementations	Interoperability	No performance analysis
[119]	Raspberry Pi, BeagleBone and BeagleBone Black	Latency, bandwidth, CPU and memory	Same implementation on different hardware
[217]	Californium, jCoAP, libcoap over Raspberry Pi 1 & 3, ZedBoard	Build time, the socket time, the packet read time	No mixed server/clients

Table 3.2: Previous CoAP benchmarks.

For the last group of research pieces, i.e., the security aspects of the protocols, the analysis has also been focused on CoAP. As previously mentioned, industrial applications have undeniable security requirements, due to the fact that they can affect the physical world though actuators and loosing data or publishing industrial secrets can have a huge economical impact on the companies. However, security on IoT is still in its early phases.

In this regard, discussion is still been carried out to analyze the feasibility of adding security to CoAP based communications, using DTLS [171] in resource constrained devices, due to the costs of the handshake and cryptographic calculations.

Kothmayr et al. [115] presented a preliminary work to later expand it in [116]. In this article, the authors point out the need to add security to information exchange to avoid attackers to capture the traffic between IoT devices. In this case, they propose to use DTLS and evaluate its performance in an OPAL sensor node. They conclude that it is possible

to add two-way authenticated asymmetric encryption through X.509 certificate exchange with less than 20 kB RAM usage and low power consumption.

Kwon et al. [121] analyze the challenge of using DTLS with the different available modes. To do that, the authors compare the RawPublicKey and PreSharedKey modes of DTLS both in a simulation and in a real test-bed to analyze the comparison in terms of code size, energy consumption, and processing and receiving time. Finally, they conclude that Class 1 devices can use PreSharedKey mode but even though RawPublicKey is mandatory to implement CoAP over DTLS, Class 1 and Class 2 devices can not implement it well due to resource limitations.

Based on the literature presented in previous paragraphs, it can be concluded that DTLS support for resource constrained devices is still a work in progress. Very resource constrained devices need to be secured in order to avoid attacks, but at the present day there is not yet an available complete solution.

3.3 **Protocol Comparison**

Analyzing the related literature presented in Section 3.2, it can be concluded that most of the research in comparing different communication protocols has been conducted on big platforms such as full PCs, or medium platforms, such as Raspberry Pi's or Beaglebones. However, smaller platforms are used in the IoT, so experiments including resource constrained devices such as ARM Cortex-M based boards can be very useful, and the surveyed literature lacks of such analysis. This subcontribution addresses that need, conducting an experiment to analyze and compare the performance of MQTT and CoAP on an ARM Cortex-M

based platform. Some of the pieces of research presented in the previous section also target small devices in sensor nodes, i.e., [109, 39, 65], but they were published after the work of this subcontribution was carried out and published in [92] and [93].

3.3.1 Analyzed Protocols

As previously stated, for the experimental analysis two protocols have been selected, i.e., MQTT and CoAP. These two protocols have been selected because they are the two main pure IoT protocols, which results on lighter resource requirements [67]. Both protocols are being used on IoT deployments, although systems using MQTT are more expanded, mainly due to being available some years before CoAP. However, CoAP is also gaining a lot of momentum [33]. This section presents the main features that these two protocols present.

3.3.1.1 MQTT

MQ Telemetry Transport (MQTT) was originally developed by IBM on 1999. The 3.1 version was released for public usage on 2010. Although the current version is v5.0 [149], it was presented in April 2019. Therefore, for this work, v3.1.1 [148] was used, released by OASIS on October 2014.

MQTT makes use of the publish/subscribe communication model. MQTT publishers, a type of client in MQTT, push messages using channels named topics and MQTT subscribers, the other type of MQTT client, subscribe to those topics to get their associated messages. A central server referred to as broker is required, the publishers pushes messages to the broker, and the broker redirects the received messages to the subscribers as can be seen in Figure 3.1.

Most MQTT implementations run on top of TCP, but the specification does not define the underlying transport protocol. It just specifies some requirements that the transport protocol need to fulfill, such as ordered and guaranteed messsage delivery. TLS can also be used to secure the message transmission and 8883 and 1883 are the registered default TCP ports registered with IANA, for TLS encrypted and non encrypted communications respectively.

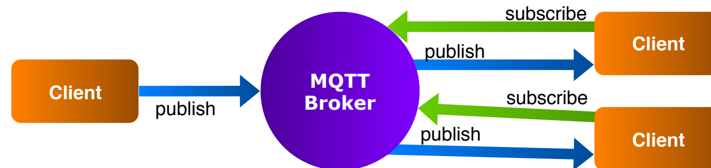


Figure 3.1: MQTT broker, publisher and subscribers.

There are 14 different message types in MQTT that manage the connection and subscription status (see Figure 3.3), pinging, and publishing, with their corresponding acknowledgments. The length of the message is variable as in depends on the type, the topic length and the payload length, but it ranges from the two bytes for ping messages to a maximum of 256 MB.

MQTT offers three levels of QoS, numbered from 0 to 2 (see Figure 3.2). QoS 0, also known as at-most-once works as best effort basis, it does not guarantee message reception. QoS 1, also called at-least-once, guarantees that the message is received, but it might be received more than once. QoS 2, the highest one, is known as exactly-once and sends further messages to avoid message duplication. MQTT clients ask for certain QoS when publishing or subscribing and the broker is expected to apply the appropriate QoS. However, if the broker does not support that level, the client and the broker must agree on the maximum

Message type	Code
CONNECT	0x10
CONNACK	0x20
PUBLISH	0x30
PUBACK	0x40
PUBREC	0x50
PUBREL	0x60
PUBCOMP	0x70
SUBSCRIBE	0x80
SUBACK	0x90
UNSUBSCRIBE	0xA0
UNSUBACK	0xB0
PINGREQ	0xC0
PINGRESP	0xD0
DISCONNECT	0xE0

Table 3.3: MQTT message types and their hexadecimal code.

supported QoS. A message can be published with a higher QoS than the subscribing client wants, in that case, the broker downgrades the QoS when pushing the message to the subscribed client.

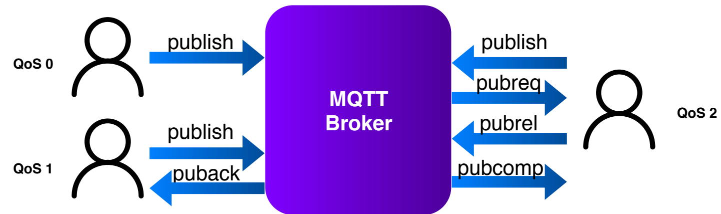


Figure 3.2: Message exchange with different QoS on MQTT.

MQTT also includes some additional features, i.e., last will (messages that are sent in case of unexpected disconnections), message buffering for recovery reasons, and retained messages, which allow to get the last message sent when a new client subscribes.

3.3.1.2 **CoAP**

The Constrained Application Protocol (CoAP) was developed by the IETF CoRE Working Group, following the HTTP protocol, but adapting it to resource constrained devices. The standardization process of the first version was finished when it was released as RFC 7252 [34] on June 2014.

CoAP follows the REST paradigm and the client-server communication pattern. A subset of the HTTP verbs is used in CoAP: GET, POST, PUT and DELETE (see Figure 3.4). The response codes are more numerous, many of which are also a subset of HTTP responses (see Figure 3.5). Even though it was originally designed for just pull requests, it is possible to use push notifications with a complementary extension.

Code	Name
0.00	EMPTY
0.01	GET
0.02	POST
0.03	PUT
0.04	DELETE

Table 3.4: CoAP request codes.

Being a protocol that targets resource constrained devices, it needs to be lightweight and less resource demanding. In this regard, the main difference with HTTP is that CoAP runs on top of UDP instead of TCP, hence it is unable to use TLS to secure the connections. To solve this, the IETF also proposed DTLS, which uses techniques based on TLS and adapt them for UDP. CoAP uses 5683 as the default port, and when including DTLS, 5684. The CoAP resources are addressed with URIS, in the form of `coap://host[:port]/[path][?query]`, where the parts in brackets are optional.

There are two different QoS in CoAP, CONfirmable (CON) and NON-Confirmable (NON), where the former requires acknowledgments while the latter does not. This is necessary in CoAP due to its underlying UDP protocol, which does not guarantee delivery. Apart from CON and NON mesages, there are other two types of messages, used for control: Reset messages and ACKs, which can deliver piggybacked responses. CoAP's header can be as small as 4 bytes, but this size is dynamic and depends on the type of message and the options that are being used.

Even though the CoAP specification is finished, the IETF is working on several extensions [78] to make CoAP more complete. For this subcontribution, the *Observing Resources in the Constrained Applica-*

Code	Name
2.01 (65)	Created
2.02 (66)	Deleted
2.03 (67)	Valid
2.04 (68)	Changed
2.05 (69)	Content
4.00 (128)	Bad Request
4.01 (129)	Unauthorized
4.02 (130)	Bad Option
4.03 (131)	Forbidden
4.04 (132)	Not Found
4.05 (133)	Method Not Allowed
4.06 (134)	Not Acceptable
4.12 (140)	Precondition Failed
4.13 (141)	Request Entity Too Large
4.15 (143)	Unsupported Content-Format
5.00 (160)	Internal Server Error
5.01 (161)	Not Implemented
5.02 (162)	Bad Gateway
5.03 (163)	Service Unavailable
5.04 (164)	Gateway Timeout
5.05 (165)	Proxying Not Supported

Table 3.5: CoAP response codes.

tion Protocol (CoAP)[68] has been used, which enables the publish-subscribe communication pattern to be used with CoAP, thus, enabling a CoAP server to send push notifications to registered clients.

3.3.2 **Experiment Setup**

After describing the protocols used in this subcontribution, this section presents the experimental setup.

3.3.2.1 **Scenario Definition**

Data transmissions on industrial environments have different requirements and characteristics, and can be classified according to their scopes. In [50], the authors split the scopes into three categories: non-real-time (e.g., configuration and parameterization), cyclical and acyclic communications. This contribution focuses on the two latter. Cyclical communications can be monitoring (upwards) and control (downwards) communications, while acyclic ones are mapped to event detection.

3.3.2.1.1 Monitoring

Upwards data transmission from sensors to controllers is one of the most used communication use case in industrial environments, in order to monitor the status of the system. Depending on how critical the data is, the reliability requirements change from no requirements, for situations such as ambient conditions monitoring, to stricter ones, like closed-loop axis positioning. Being a periodic task, reducing the overhead introduced by the communication protocol used for the transmission is desirable, lowering the network and energy usage of the sensing device. These are critical factors on the paradigm of IIoT.

MQTT and CoAP are both lightweight protocols, and as such, they introduce small overhead. They are flexible and with their characteristics, they allow to adapt to different requirements regarding reliability. MQTT's QoS is a direct example of that, and it adds higher levels of reliability with the cost of increasing the number of packages needed in a message exchange. Other features can also be used to control device disconnections on critical monitoring, e.g., retained messages or last will. CoAP, on the other hand implements CON and NON messages to enable different levels of reliability, also with the cost of increasing the number of packages needed for the exchange. Measuring the amount of the packages needed for different QoS configurations and their size on monitoring applications has been done in this contribution using the *tcpdump* network sniffer and adding the sizes of all related packages.

Industrial scenarios can not assume one-to-one communications, thus, scalability capabilities of both protocols should also be analyzed. MQTT's broker-based architecture can reduce the amount of transactions under some circumstances. A theoretical and experimental study has been carried out based on the previously mentioned *tcpdump* measurements. In some closed-loop scenarios, monitoring may imply some soft real-time requirements, which makes a latency analysis also to be important. This latency, the time since the data is known by the sensor until it reaches the monitor/controller has been measured with an oscilloscope in this subcontribution.

3.3.2.1.2 Control

Downwards action control is another common data communication scheme, sending data from controller to actuators. Same as monitoring communications, reliability depends on the criticality of the applica-

tions. For instance, sending an action to control an office lighting system is much less critical than an action to control a bridge crane lifting heavy weights. Even though energy consumption is not as important as in the previous scenario, mainly due to the fact that these scenarios are usually not battery powered, network traffic is still a major concern. Even more, the delay since an action is decided until it is executed depends directly on latencies, being a vital factor.

Same as the monitoring scenario, tests evaluating the overhead, scalability behavior, and latency (i.e., time since the control action has been decided by the controllers until it is received by the actuators), have been conducted.

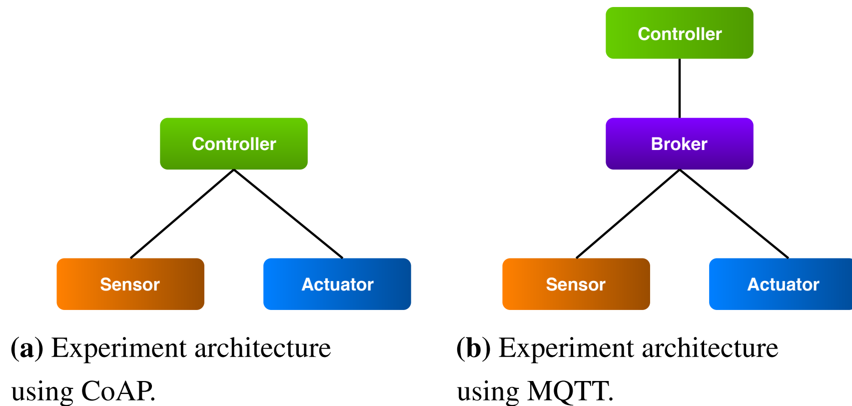
3.3.2.1.3 Event Detection

As event message transmission occur rarely comparing to control and monitoring, the overhead size and energy consumption is not as important in this scenario. It is an upwards communication, which requires the corresponding action to be executed as quickly as possible, therefore, having a low latency is the main requirement.

In this regard, a similar experiment to the latency test performed in the previous scenario has been carried out. An additional scenario has also been included, measuring MQTT clients that were not previously connected to the broker, where connect, publish and disconnect tasks are performed each time a sporadic event occurs. These are relevant in cases where holding open sessions for each device can be resource demanding for the broker, specially when those devices do not exchange regular messages (e.g., emergency-stop buttons).

3.3.2.2 Architecture Design

To be able to compare MQTT and CoAP, an architecture that fits the nature of both of them and allow a fair comparison has been designed. Figure 3.3 presents such architecture, with the three actors presented in the previous scenarios: sensors, actuators, and controllers. MQTT also requires the broker that connects all the actors.



(a) Experiment architecture using CoAP.

(b) Experiment architecture using MQTT.

Figure 3.3: Architecture scheme.

Sensors and actuators are both low level devices in charge of directly interacting with the physical world, either measuring some kind of variable and offering it to higher levels (sensors) or allowing higher level devices to affect the physical world (actuators). Controllers are above in the automation pyramid, communicating with sensors and actuators in order to fulfill their tasks and/or gather data.

CoAP has a Service Oriented Architecture (SOA) that maps directly into these actors, being the lower level ones (sensors and actuators) servers for the higher level clients (controller). On the other hand, MQTT's centralized architecture requires an additional actor, i.e., bro-

ker. The broker manages the distribution of the published messages to the subscribed clients.

3.3.2.3 Hardware and Software

As described above, there are several actors in this system. Figure 3.4 presents each actor on a different device and the connection between them using a FastEthernet switch.

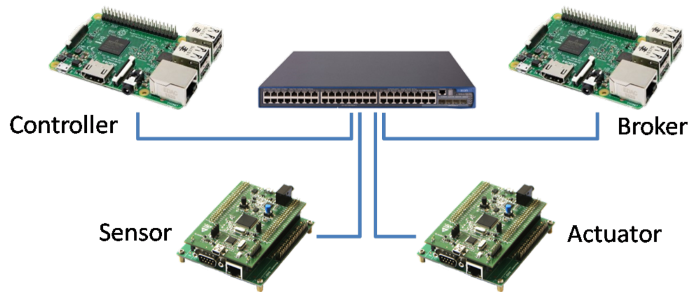


Figure 3.4: Platform scheme.

The actors of the system and their features are the following:

- The **sensor** is attached to a STM32F4 Discovery [193] board running FreeRTOS and lwIP. The MQTT *de facto* standard implementation for embedded devices has been used, i.e., Paho MQTT C/C++ for Embedded platforms [155]. For CoAP, a very lightweight implementation that integrates fine with lwIP has been selected: microcoap [1].

- The **actuator**'s setup is similar to the sensor, also being attached to a STM32F4 Discovery board with the same operative system and libraries.

- The **controller** is usually a more powerfull device, hence, a less constrained device has been selected, i.e., Raspberry Pi [169]. The Raspberry Pi runs the Raspbian operative system, and on top of that, Paho MQTT C [154] for MQTT and libcoap [131] for CoAP. Paho MQTT C [154] is the reference open source MQTT library for C implementation, while libcoap is a complete and versatile implementation of CoAP in C.

- As previously explained, MQTT needs a **broker**. In this experiment, the broker also runs on a Raspberry Pi running Raspbian. Mosquitto [140] has been selected as the broker implementation, which is supported by the Eclipse Foundation [47] and it is open source and very mature.

3.3.3 Results

After describing the setup of the experiment, in this section the obtained results are presented. Overhead, scalability and latency have been the measured parameters.

3.3.3.1 Overhead

Network overhead is the first measured metric. It represents how many bytes are sent for a successful payload transmission. It takes into account the messages and overheads related to not only MQTT or CoAP, but also the bytes required by the underlying protocols (i.e., TCP/UDP, IP, Ethernet).

3.3.3.1.1 Control

Figure 3.5 shows the sum of the bytes that are transmitted for a single byte payloads with different QoS and hierarchy levels. MQTT topics and CoAP resources allow to specify path hierarchies using "/" as the delimiter character. CoAP needs fewer bytes than MQTT for any QoS level. There are two main reasons for this: the underlying protocols (UDP for CoAP and TCP for CoAP) and the number of hops needed for end-to-end transmission. On the protocol transport layer, TCP requires additional messages to manage the connection while UDP does not. In terms of required hops, MQTT uses a broker as the intermediary device to send a message from a publisher to a subscriber. Hence, two complete MQTT communications are needed, one from the publisher to the broker and a second one from the broker to the subscriber. That results on MQTT's QoS 0 having double the overhead comparing to CoAP's CON transmission. In the case of MQTT, the higher the QoS level is, the more messages are required [148]. QoS 2 requires two and a half times the amount of bytes QoS 0 needs. On the other hand, CoAP's CON requests need double the bytes of NON ones, the required acknowledgment.

Adding hierarchical levels to the topic or path increases the amount of bytes, either by using a longer string for the topic in MQTT or by adding further path options in CoAP. This different behavior does not affect the number of bytes, as the overhead of using multiple options in CoAP is compensated due to the option header being only one byte, just as the "/" delimiter character, which is not included in the option.

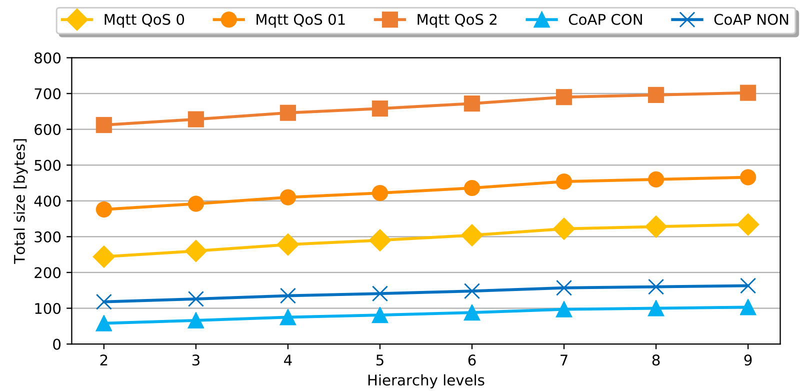


Figure 3.5: Overhead on control scenarios for CoAP and MQTT.

3.3.3.1.2 Monitoring

Monitoring and control scenarios are very similar on a functional level, hence, the results are also similar. MQTT results are exactly the same (see Figure 3.6), as the communication messages are equal but in the reverse direction, upwards. In the case of CoAP, in monitoring requires GET messages instead of PUT messages. A GET message requires a response, in both CON or NON transmissions. The answer in CON messages can be piggybacked, hence, both CON and NON monitoring have the exact same overhead as the codes in the header change, but not the size. This size is the same as a CON PUT request.

3.3.3.2 Scalability

The overhead analysis has been carried on one-to-one communication scenarios, but both protocols scale differently in one-to-many monitoring and control situations

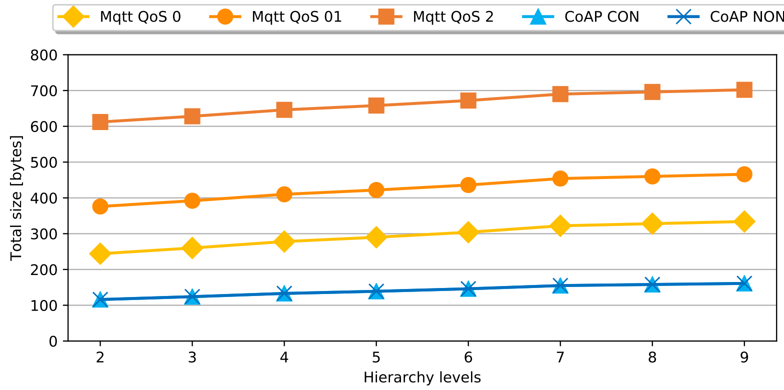


Figure 3.6: Overhead on monitoring scenarios for CoAP and MQTT.

3.3.3.2.1 Control

Thanks to MQTT's central broker architecture, scenarios where a single action has to be performed by multiple actuators require just one controller-to-broker transmission and the broker distributes the message to all the actuators. This results on requiring $N + 1$ transmissions, where N is the number of actuators. CoAP, in contrast, uses a client-server architecture. However, it allows multicast as improvement in one-to-many communications, but it is limited to local or IPv6 networks and excludes the usage of DTLS. These limitations are quite strict, so for this experiment, the test has been carried out without multicast communication. This results on a total number of transmission of N. Figure 3.7 presents the sum of the bytes needed for a different number of actuators.

Dividing the values from Figure 3.7 by a reference, the proportional size between different protocols can be obtained. CoAP has been selected as the reference, due its smaller overhead and CON messages as they are more versatile and can be used for both control and monitoring.

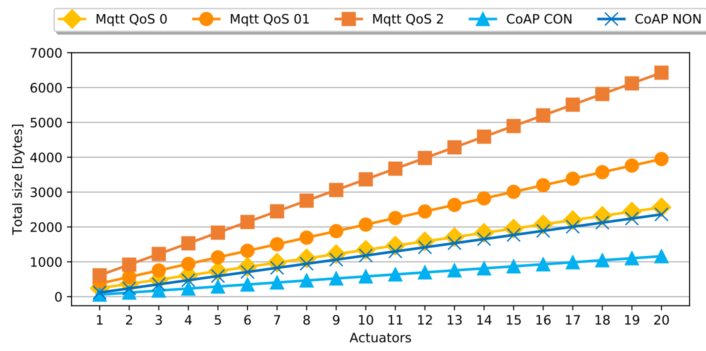


Figure 3.7: One-to-many control overhead with 1 to 20 actuators.

Fig. 3.8 shows horizontal lines for CoAP's messages, with values of 1 (CON) and 0.5 (NON). MQTT yields asymptotic functions tending to 1, 1.5 and 2.5, reducing the difference between MQTT's QoS 0 and CoAP by 80% for five actuators.

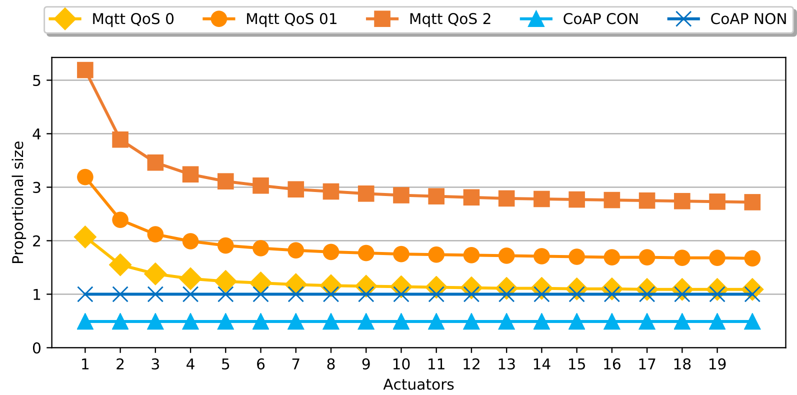


Figure 3.8: Proportional transmission size per actuator.

3.3.3.2.2 Monitoring

Once more, the scalability experiment yields analogous results for monitoring and control with the same exception. NON message exchange values in one-to-many scenarios are equal to CON exchanges, and therefore, so is the proportional size, as shown in Figure 3.9 and Figure 3.10.

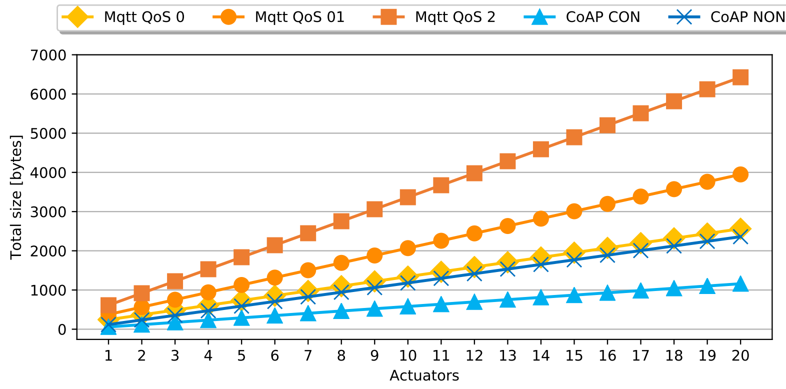


Figure 3.9: One-to-many monitoring overhead.

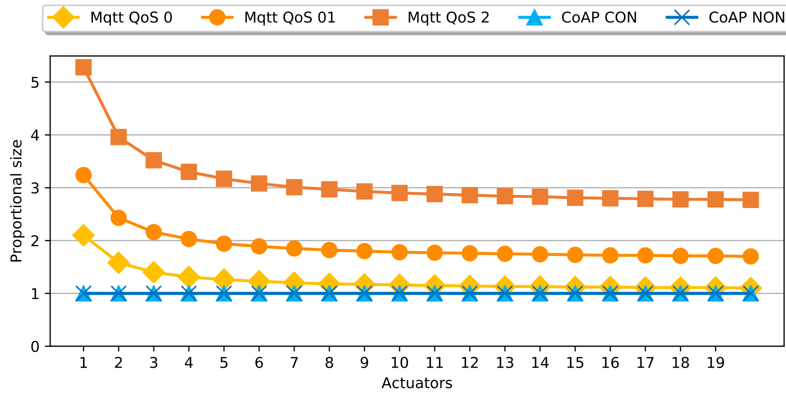


Figure 3.10: Proportional size per monitor.

3.3.3.3 Latency

The last metric is the latency, defined as the time between the control action is sent or the data measurement is ready until it reaches the destination device.

3.3.3.3.1 Control

Figure 3.11 shows the latency for control scenarios. It shows that CoAP delivers messages five to ten times faster. These results are expected, as CoAP uses UDP and it does not require an intermediate actor.

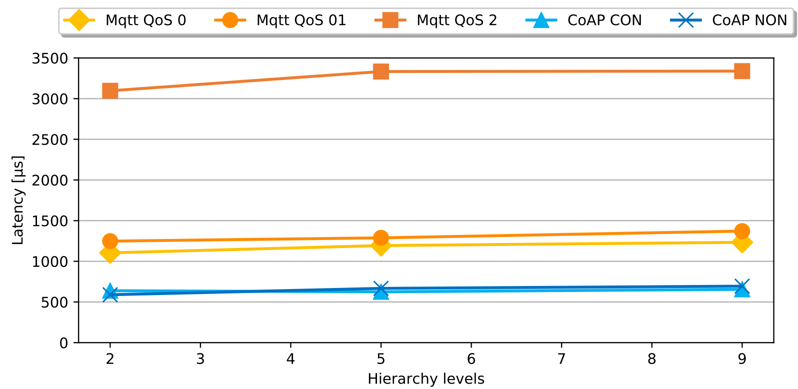


Figure 3.11: Control latency.

3.3.3.3.2 Monitoring

Like in the control scenario, CoAP is also faster than MQTT in monitoring as shown in Figure 3.12. However, in this case the difference is smaller, being the difference between two and five times. This is due to monitoring being an upwards communication. MQTT's architecture does not favor any direction on the communication, while in CoAP, the

initiative always stands in the controller, hence, downwards communication is faster than upwards. It is also worth noticing that the main difference between monitoring and control appears with MQTT using QoS 2, where handling the extra messages imposed by higher QoS is done faster in a higher resource device such as the Raspberry Pi.

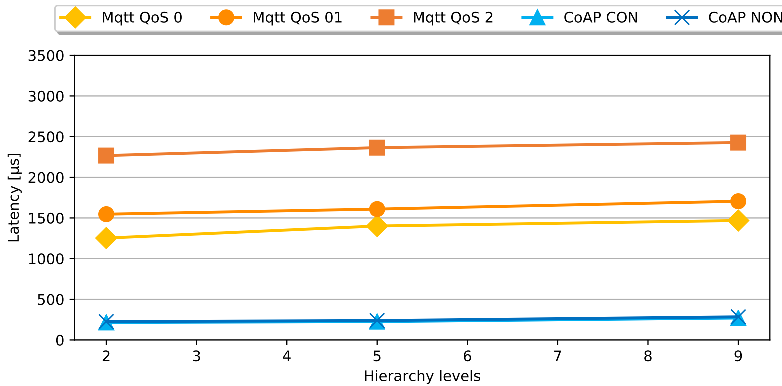


Figure 3.12: Monitoring latency.

3.3.3.3.3 Event detection

The last scenario for measuring the latency is event detection. In this case, the messages can be very spaced in time, hence, having an open connection may not be optimal. Taking that into account, the latency of MQTT has been measured with and without a previous connection established. CoAP runs on top of UDP, so it does not need an open session. Figure 3.13 shows the results of this test, where it can be seen that the latency of MQTT having to open the connection before sending the message increases by two or three compared to the case where the connection was already open. However, it is still on the milliseconds range, which can be enough to meet strict time requirements. If the

application imposes shorter latency requirements, CoAP would be a better alternative.

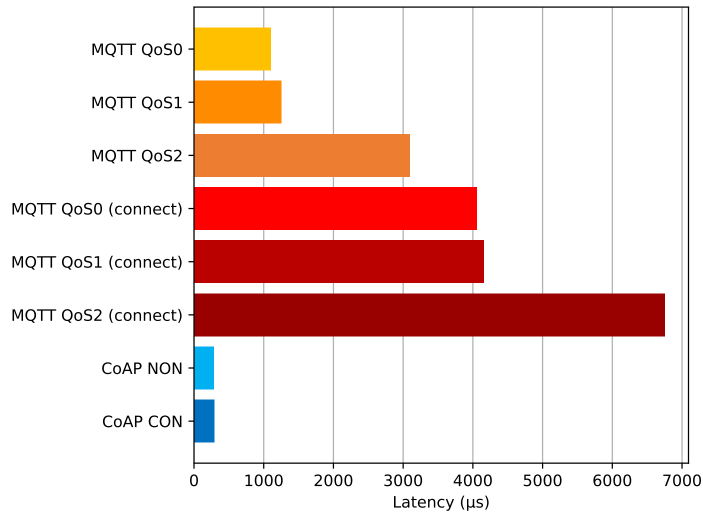


Figure 3.13: Event detection latency.

After the testing and comparing CoAP and MQTT on a lossless communication with no congestion, we can see that CoAP has advantages over MQTT. It requires fewer bytes to transfer the same message with shorter delays. In addition, thanks to the Observe extension, CoAP is more versatile for monitor and control uses, while MQTT and its publish-subscribe does not fit that good for control. This is why to continue the work of this thesis, CoAP has been the selected protocol.

3.4 Comparison of CoAP Libraries

After comparing MQTT and CoAP's performance on resource constrained devices, CoAP was selected to analyze it further. As it has been presented in Section 3.2 and grouped in Table 3.1, CoAP and

its performance has been analyzed and compared against other protocols, and in different hardware. However, the only previous research analyzing CoAP implementations is based on early drafts of the specification and is outdated. Besides, it does not analyze the performance of said implementations, it tests their compatibility and compare the main features. Thus, in this subcontribution an experiment with nine CoAP libraries has been done, comparing them theoretically, including the libraries they use for adding DTLS for encryption, along with a performance analysis. With this experiment, the main objective is to fill the gap offering an up to date comparison, both empirical and feature-based, to help system designers choose the implementation that better fits their requirements for industrial environments. For helping to choose the most adequate library, different aspects have been analyzed, such as the offered features or the performance.

3.4.1 **CoAP**

As previously explained, CoAP was published as RFC 7252 by the IETF CoRE Working Group [79] on June 2014. It is a web transfer protocol designed for resource constrained devices and networks that have already been used on eHealth systems [112] or Smart Lightning systems such as Smart Home environments [209] and Smart Cities [28]. It also has been used as the underlying protocol for frameworks or protocols higher in the application stack such as LWM2M [151] or W3C Web of Things [118].

Built on top of UDP and mimicking HTTP, CoAP follows the REST paradigm. The different technologies involved in IoT applications using CoAP and general use case stacks of the Internet are presented in Table 3.6. Instead of TLS, CoAP uses DTLS to encrypt messages.

Layer	IoT	General Applications
Application	CoAP(s)	HTTP(s)
Security	DTLS	TLS
Transport	UDP	TCP
Network	IPv6 6LowPAN	IPv4, IPv6
Link	IEEE 802.15.4	IEEE 802.3, 802.11

Table 3.6: IoT and general application stacks.

CoAP header is variable in size, with a minimum of 4 bytes. Optional options make the size variable and the structure a CoAP message is shown in Figure 3.14. The first two bits represent the CoAP version, followed by another two bits that represent the message's type. As previously explained, the underlying UDP does not guarantee message delivery, so CoAP optionally can do that in the application layer, using CON messages. The other types of messages are NON, ACK and Reset messages. As it also has been explained, the ACK can include piggy-backed payloads. The rest of the first header byte is for representing the length of the token, which can be included in the optional part of the header and it is limited to 8 bytes. The second byte of the header is for the message code, e.g., GET or 2.05 Content, etc. The other two bytes for the mandatory part of the header are reserved for the message ID. In the optional parts of the header are first the token and then options. Some of the most important possible options are presented in Table 3.7. Finally, after a separator byte of full set bytes (1s) goes the payload.

In order to categorize CoAP libraries, first, CoAP needs to be further analyzed, specially its extensions. As stated before, CoAP's specification is complete, however, some missing features have been

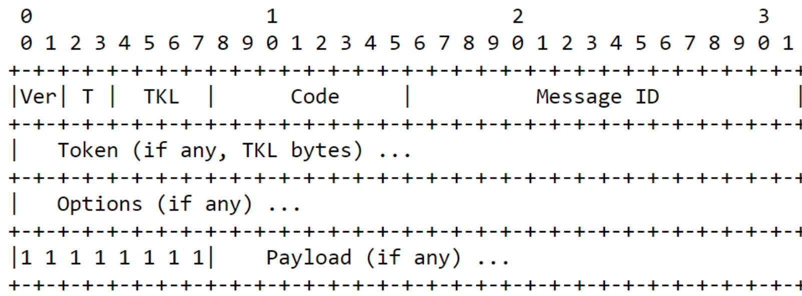


Figure 3.14: CoAP message [34].

No.	C	U	N	R	Name	Format	Length	Default
1	x			x	If-Match	opaque	0-8	(none)
3	x	x	-		Uri-Host	string	1-255	*
4				x	ETag	opaque	1-8	(none)
5	x				If-None-Match	empty	0	(none)
7	x	x	-		Uri-Port	uint	0-2	*
8				x	Location-Path	string	0-255	(none)
11	x	x	-	x	Uri-Path	string	0-255	(none)
12					Content-Format	uint	0-2	(none)
14		x	-		Max-Age	uint	0-4	60
15	x	x	-	x	Uri-Query	string	0-255	(none)
17	x				Accept	uint	0-2	(none)
20				x	Location-Query	string	0-255	(none)
28			x		Size2	uint	0-4	(none)
35	x	x	-		Proxy-Uri	string	1-1034	(none)
39	x	x	-		Proxy-Scheme	string	1-255	(none)
60			x		Size1	uint	0-4	(none)

C = Critical, U = Unsafe, N = No-Cache-Key, R = Repeatable.

* Taken from destination address/port of request message

Table 3.7: Most important options of CoAP.

described while others are being worked on by the IETF [78]. This missing features are described as CoAP extensions to make it more complete. The most important extensions that broaden the capabilities of the CoAP specifications are:

- **Constrained RESTful Environments (CoRE) Link Format** [181]. This extension is based on the HTTP Link Header field (RFC 5988) [146] and it is used to define the format for the links that constrained servers use to describe their resources, attributes, and the relationship between links. Link format defines a new Internet media-type, i.e.,"application/link-format", which is used to encode the payload. It also defines a well-known URI, i.e., `/.well-known/core` as a default entry point to get information about the resources of the server.

- **Block-Wise Transfers in the Constrained Application Protocol (CoAP)** [20]. CoAP aims to avoid IP fragmentation, hence, it uses small payloads. In most IoT scenarios this is not an issue since sensors do not usually send and actuators do not require large data. However, in some specific cases large payloads may be needed, for instance in firmware updates. To solve this, the Block-Wise transfer extension adds new options, allowing to send the payload in pieces in different CoAP messages following a Stop-and-Wait transmission with a flag to indicate whether the message has the last slice of the payload or more will follow. To do that, Block-Wise transfer adds new options to CoAP.

- **CoRE Resource Directory** [182]. An entity named Resource Directory (RD) is proposed by the IETF to store the information about the resources of different CoAP servers. This way battery

can be saved on the other servers when a client is searching for the available resources, also facilitating a centralized research discovery, with no need for scanning the entire network or using multicast addresses. This extension enables three ways for the RD to get the information about the resources of the servers. The servers can send their information themselves, the RD can ask the servers to send the information and in some cases the information can be requested in boot time from a third device, called the commissioning tool. The commissioning tool gets that information from a file, a database or similar.

- **Observing Resources in the Constrained Application Protocol (CoAP)** [68]. To enable a publish-subscribe communication pattern on CoAP, this extension is proposed, which allows to send push notifications to subscribed clients, similar to MQTT or XMPP. CoAP clients can subscribe to a resource, and the server sends the resource representation when a change happens on the resource. The server is the device that dictates what a change is, e.g., a value change, a value change out of a range, or periodic transmissions. A petition to subscribe is a GET request with an option enabled. To stop the subscription, a subscribed client can send another GET request to the same resource with the option disabled, or also a reset message.

- **Group Communication for the Constrained Application Protocol (CoAP)** [166]. CoAP is designed to use IPv6 in the internet layer and IPv6 supports multicast by default. However, IPv4 networks are still majorly deployed. This extension explains how to use CoAP on multicast environment and it also includes new fea-

tures: new CoAP processing functionalities (e.g., new rules for reuse of token values, request suppression and proxy operation), and a new management interface.

- **CoAP Simple Congestion Control/Advanced (CoCoA)** [17]. CoAP includes a basic behavior to minimize network congestion, but has many limitations. CoCoA adds more sophisticated methods to minimize even more the network congestion and allow a better throughput.

3.4.2 CoAP implementations

There are several CoAP implementations available. Some of them are more completed in terms of supporting the entire specification or adding support for more extensions. Also, some target more resource constrained devices and platforms than others. In this subcontribution, the aim is to compare some open source libraries, covering different programming languages and runtime environments. Nine implementations have been selected for the analysis. Diverse implementations have been selected, to offer an analysis with implementations that have different features, implement different CoAP versions, have been coded in different programming languages, etc. The selected libraries are presented below:

- **libcoap** [131] is a library written in C, designed to fit on a wide range of devices, from small embedded devices to big POSIX ones. It implements the final version of the RFC 7252 for both client and server side and it also provides support for several extensions, i.e., Observe, Block-Wise transfer and Resource Directory. The source code includes very complete examples and

it is designed to easily add DTLS with OpenSSL or tinydtls. Its github repository is up to date and the development continues with constant updates to enhance the features and integration with other DTLS libraries

- **smcp**[1] [188] is also written in C and aims to be deployed on devices from bate-metal sensors to Linux-based systems, including embedded ones. Similar to libcoap, it also supports client and server implementations over the final RFC 7252. It can be used on top of BSD sockets or μIP. The CoAP capabilities that smcp implements are extended with the Observe mode and Multicast groups. It also provides a useful command line client called smcpctl, and an at the time of the analysis experimental DTLS branch.

- **microcoap** [1] targets resource constrained microcontrollers, and hence, it is a limited CoAP library in C. The source code provides examples for POSIX and Arduino. It implements a part of the RFC 7252, just the server side with limited features. Some limitations are that it just supports some of the verbs (GET, PUT, and POST, DELETE is not supported), only piggybacked ACKs (non piggybacked ACKs are not allowed) and it does not support retries.

- **FreeCoAP** [56] is the last analyzed C implementation in this work. It targets GNU/Linux devices and it uses GnuTLS for

[1] Since this analysis was done, smcp evolved to something beyond CoAP. A spin off was created on March 2017 to disengage the CoAP part of the implementation, named libnyoci, `https://github.com/darconeous/libnyoci`

security, with a new tinydtls branch. It implements the final specification, i.e., RFC 7252.

- **Californium** [23] is a very complete implementation in Java for not so constrained devices. It targets back-ends running the JVM and it implements both client and server sides for CoAP. In addition to the final version of RFC 7252, it also implements several extensions, i.e., Observe, Block-Wise transfer and Resource Directory. The CoAP messages can be ciphered using the Scandium project for DTLS, which is integrated in the Californium project.

- **h5.coap** [66] is a JavaScript library that runs on top of the Node.js platform. It only implements the client side and follows the finished standard RFC 7252. It also implements the Observe and the Block-Wise transfer extensions.

- **node-coap** [145] is another JavaScript implementations targeting the Node.js platform. It provides support for generating both client and server applications and implements the stable version of the RFC 7252. In addition to the base CoAP, it also implements the Observe and Block-Wise transfer extensions.

- **CoAPthon** [35] is the first analyzed Python implementation for CoAP. It implements client and server sides, with the RFC 7252 specification. Some extensions are also implemented in this library, i.e., Observe, Core-Link format, Multicast and Block-Wise transfer.

- **CoAPy** [36] is the last analyzed library in this subcontribution. It implements an old CoAP draft (draft-ietf-core-coap-02) in Python, making it theoretically not compatible with the others.

In addition to the old CoAP draft, it also supports the Block-Wise transfer extension.

The above list covers implementations in different languages that target different hardware. However, there are some other libraries that are worth to mention. These libraries, e.g., Copper [117], Erbium [49] and TinyCoAP [207], have been discarded. Copper is a visual client implemented as a Firefox plugin. Erbium is a widely used C implementation that targets devices running ContikiOS and TinyCoAP targets tinyOS devices. The working platform for these experiments has been the Raspberry Pi, so these implementations do not fit.

3.4.3 Security in CoAP: DTLS

TLS [45] requires ordered and guaranteed message delivery, which UDP does not provide, hence, CoAP can not use TLS to cipher messages. Instead, DTLS [171] has been described by the IETF as RFC 6347 in order to secure UDP communications. DTLS is hugely based on TLS, but adds some features to overcome the issues related to delivery failure or disordered message arrival. However, it pays the cost of losing some of the benefits of UDP comparing to TCP, e.g., not requiring an open connection or the endpoints not needing to save session information. DTLS ciphers the messages, but not the routing information. This is important, as for example, in a mesh network, the nodes need to know the intended receiver to be able to route the datagrams through the network. Other security aspects such as Denial of Service (DoS) attacks, resource consumption and similars are out of the scope of this subcontribution as they are related to other levels of the TCP/IP stack.

The RFC 6347 specification defines 4 working modes, which depend on how the messages are encoded and decoded:

- NoSec: Cryptography not enabled.
- PreSharedKey: Symmetric cryptography. The keys for encoding and decoding are the same.
- RawPublicKey: Asymmetric cryptography. Raw keys are used, with different keys for encoding and decoding.
- Certificate: Asymmetric cryptography with keys provided through certificates. Same as the RawPublicKey, the encoding and decoding keys are different. However, the public keys are shared using certificates.

When using symmetric or asymmetric cryptography, the difference is the keys that need to be used to encode and decode the messages. On the one hand, symmetric cryptography uses the same key to encode and decode, which means that a secret common key needs to be held in both communications endpoints. Asymmetric, on the other hand, uses a pair of keys, public and private. Each of them only can only decode the message encoded with the opposite. The main disadvantage of the symmetric approach is key management, as it requires to be distributed to both devices, while the coding speed is its main advantage. On the contrary, with the asymmetric approach, a public key can be freely distributed without compromising security, although the secret key can not be shared. It is important to point out that as one of the keys is of public domain, messages ciphered with the private key do not prevent other listeners to decode it, but instead offers a way of authenticating the sender of a message as it is only owned by one endpoint. This means that messages may be ciphered twice, once with the desired receiver's public key to ensure its privacy and another with the sender's private key for authentication purposes.

In order to optimize the encoding and decoding resource and time consumption the asymmetric cryptography requires, usually it is only used for an initial process named handshake, where a symmetric temporal key is negotiated between client and server. This approach benefits from the lighter symmetric key encryption, while its main drawback is overcome as the generated key is temporal and different for each session.

There are three authentication options for the handshake process: no authentication; one-way authentication, where only one party of the communication is authenticated; and mutual or two-way authentication, where both the client and the server are authenticated. Each of these options use different messages for the handshake, and the three of them are represented on Figure 3.15.

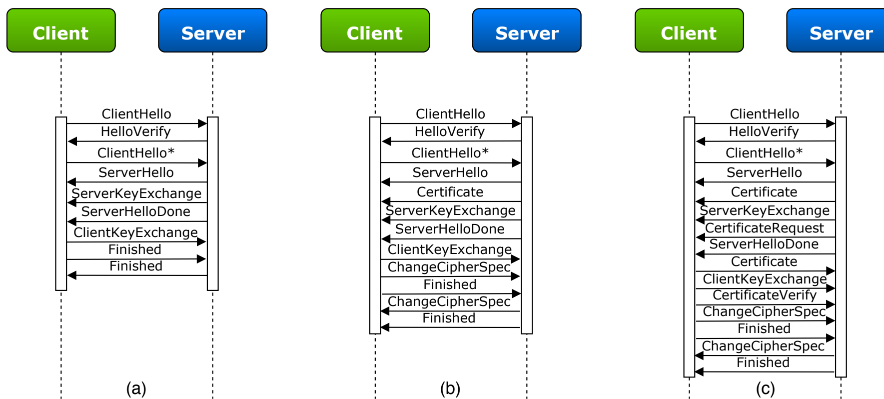


Figure 3.15: DTLS handshake with no authentication (a), server-side only (b) and mutual authentication (c).

From the modes specified by DTLS, CoAP requires NoSec to be implemented as messages do not always need to be ciphered. RFC 7252 also states that when using DTLS in PreSharedKey mode, the default and hence mandatory suite to support, is TLS_PSK_WITH_-

AES_128_CCM_8, while others are optional. When asymmetric modes are supported, TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 is the mandatory cipher suite. Both RawPublicKey and Certificate modes are set up using mutual authentication. All cipher suites are registered in the IANA registry[1].

There are several mathematical options to generate asymmetric key pairs. For resource constrained devices, elliptic curve algorithms are better optimized as they use shorter keys for the same level of security. The calculations are more complex, implying higher resource consumption when coding and decoding, but it is made up for when the keys are transmitted, which is much less consuming as they are shorter.

3.4.4 **Feature Comparison**

After the different CoAP implementations have been presented, the next step is to analyze them to assess their features and offer a guideline to select an appropriate one to assure that it fulfills the requirements of each project. For this, the first step is to compare the implementations from a theoretical viewpoint, analyzing their characteristics (language, target platform, extensions, etc.) and then to describe how easy is to use the libraries for the implementation of a project.

Table 3.8 summarizes the version of each of the analyzed libraries, the programming language they are implemented in and the target platform. There are four C implementations, one in Java and two in JavaScript and Python. The target platforms are diverse in the selected pool of implementations, but a Raspberry Pi running GNU/Linux can run all of them,

[1]http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml

Library	Analyzed Version	Language	Target platform
libcoap	Develop Sept. 24, 2016	C	POSIX, Contiki, lwIP, TinyOS
smcp	Master Sept. 24, 2016	C	Embedded devices, bare-metal sensors, Linux-based devices
microcoap	Master Sept. 24, 2016	C	Arduino, POSIX
FreeCoAP	Master Sept. 24, 2016	C	GNU/Linux
Californium	1.1.0-SNAPSHOT	Java	JVM supporting devices
h5.coap	0.0.0	JavaScript	Node.js supporting devices
node-coap	0.18.0	JavaScript	Node,js supporting devices
CoAPthon	Master Sept. 24, 2016	Python	Python supporting devices
CoAPy	0.0.3-DEV	Python	Python supporting device

Table 3.8: CoAP libraries' features - Part 1.

Table 3.9 presents more technical details, specifying the CoAP version the libraries implement, whether they provide support for client and servers, the supported extensions and DTLS libraries they integrate. CoAPy is implements an early CoAP draft (draft-02) and has not been active recently. All the other libraries implement RFC 7252, although microcoap does that just partially.

Library	Spec.	Client	Server	Extensions*				Security library
				Obs.	Block.	RD	Multi.	
libcoap	RFC 7252	✓	✓	✓	✓	✓		tinyDTLS, GnuTLS OpenSSL
smcp	RFC 7252	✓	✓	✓			✓	OpenSSL
microcoap	RFC 7252	–	✓					–
FreeCoAP	RFC 7252	✓	✓					tinyDTLS, GnuTLS
Californium	RFC 7252	✓	✓	✓	✓	✓		Scandium
h5.coap	RFC 7252	✓	–	✓	✓			–
node-coap	RFC 7252	✓	✓	✓	✓			node-mbed-dtls
CoAPthon	RFC 7252	✓	✓	✓	✓		✓	pyDTLS
CoAPy	Draft-2	✓	✓		✓			–

* Even if most of them do not explicitly say it, they support Core Link-Format extension.

Table 3.9: CoAP libraries' features - Part 2.

Microcoap only implements the server side and h.5coap only implements the client side. Al the others implement both client and server sides of the communication. Regarding the extensions, most of them implement the Observe and Block-Wise transfer. These are the most useful extensions, as they expand the features of CoAP itself. In addition, the most mature ones (i.e., libcoap and Californium) also support the Resource Discovery. CoRE link-format, despite not explicitly being mentioned in the description of the libraries, is supported by most of them.

Starting the experiment with a preliminary analysis, some preconceptions can be made. First, considering the compatibility between libraries, all the analyzed libraries but CoAPy should be compatible. CoAPy uses some deprecated options, such as Path-Uri, where code 9 is used. CoAP's draft-03 changed this option to code 11, thus, making implementations from previous drafts non compatible. And second, in terms of performance, the fist four libraries are expected to be faster and more lightweight. This is due to the fact that they are natively implemented C, instead of a non-native language that requires an additional software layer to be able to run: Java Virtual Machine (JVM) for Californium, Node.js and the V8 JavaScript engine for the JavaScript implementations and the Python interpreter for Python ones.

Some implementations are more mature than others, which makes that few of them have evolved not only to offer basic functionalities but also to provide advanced mechanisms to deal with resources, available request types and response codes. Due to this, the creation and management of resources varies between implementations, directlu affecting the application development processes, as it is described below:

- **libcoap** includes an interface to make adding resources very easy. The library itself manages the response codes, so the developers just need to add the name of the resource, which types if request it supports and link each request to a handler.

- **smcp** deals with resources similar to libcoap. The developers only need to create handlers and resources, and add them to the system through an interface. The library handles the response codes.

- **microcoap** uses a resource array. To add new resources to the server, they need to be defined, and add them to the array, along with their handlers. The library manages everything else by itself.

- **FreeCoAP** is a bit tricky for adding new resources and handling their response codes. They need to be defined by the developers, along with the resource paths. FreeCoAP does not include an interface to easy the resource generation.

- **Californium** is a very mature implementation and adding and managing resources is easy. To add a new resource, a Java class needs to be created, with the handlers for different types of supported request. Then, using an interface, the resources can be easily added to the server and the library maps the request to their corresponding handlers.

- **node-coap** does not provide tools for handling resources and response codes, the developers have to do it by hand. There is no interface for the creation of resources and management of response codes, hence, the handling of them has to be made by the application itself, not the library.

- **CoAPthon** manages by itself the response codes and resources. The developers need to create a Python class for each resource and include the methods that the application supports on the class.
- **CoAPy** needs a Python class for each resource. However, the application needs to handle the response codes, as the library does not provide with tools for that.

Summarizing, libcoap, smcp, microcoap, Californium and CoAPthon are the most developer friendly implementations to create CoAP servers, as they provide tools to create resources and handle the response codes. The developers just need to define resources and their handlers and link them to the server. The rest of libraries require the developers to handle the resources explicitly, adding unneeded complexity to the application.

3.4.5 Security Feature Comparison

Some analyzed CoAP libraries have the option to include DTLS capabilities using different DTLS implementations, as shown in Table 3.9. Six different DTLS libraries are present in the table: tinydtls, GnuTLS, OpenSSL, Scandium, PyDTLS and node-mbed-dtls. However, the last two are Python and JavaScript wrappers around OpenSSL and mbed TLS respectively, hence, in this section five libraries are analyzed, presented in Table 3.10. For the analysis, first, the evaluation version is specified, followed by the implementation language and the targeted device types. Finally, the last column indicates whether the libraries are general security libraries including TLS and DTLS or a more resource constrained approach, implementing just DTLS. The supported DTLS version has not been included in the table, as all the libraries implement the latest version, i.e., DTLS 1.2.

Library	Version	Language	Target	TLS/DTLS
OpenSSL	1.1.0f	C, assembly	UNIX and Windows	TLS+DTLS
GnuTLS	3.6.0	C	UNIX and Windows	TLS+DTLS
mbed TLS	2.6.0	C	Embedded devices; ports for ARM, PowerPC, MIPS, Motorola 68,000, ×86, ×64	TLS+DTLS
tinyDTLS	0.9	C	Embedded devices	DTLS
Scandium	Part of Californium Framework 1.0.0	Java	JVM supporting devices	DTLS

Table 3.10: DTLS libraries' features.

OpenSSL [152] is one of the most mature and complete cryptography libraries including all TLS and DTLS versions and certificate handling among other features. Some of the code base is implemented directly in assembly making it, in theory, one of the fastest libraries. Its main downside for IoT applications is that OpenSSL is conceived to be installed in desktop computers, so the device requirements are not optimized for resource constrained devices.

OpenSSL's licence is not compatible with GPL projects. Hence, **GnuTLS** [61] has been developed, which is a library that shares most of OpenSSL features. GnuTLS is not as optimized as OpenSSL so it is expected to be slightly slower, but it also supports all TLS and DTLS versions, certificates and targets desktop devices.

ARM bought PolarSSL and renamed it **mbed TLS** [7]. Being ARM a company that targets smaller devices, mbed TLS's footprint is smaller that the previous two, despite offering similar features, including TLS, DTLS and certificate support. It is not as optimized as the previous ones in terms of speed, but as an upside, mbed TLS offers an easy to use API to help developers.

Under the Eclipse Foundation's umbrella, two completely different DTLS implementations have been developed: **tinydtls** [202] and **Scandium** [203]. The former is specifically developed for resource con-

strained devices only implementing basic DTLS features to minimize the footprint. The latter is a Java library integrated in the Californium framework [23]. It implements the entire DTLS 1.2 specification and supports all CoAP requirements regarding security.

In this survey, the DTLS libraries used by the analyzed CoAP libraries are presented. However, there are also more DTLS libraries, such as WolfSSL (formerly CyaSSL) [221], Lithe [170] or tinyECC [122].

3.4.6 Experiment Setup

Current industry solutions rely on wired networks such as Profibus, Modbus, CAN and Profinet to cite some. However, these are not very flexible and modifying the network or their setup requires wiring changes. In contrast, wireless networks are becoming more reliable and their acceptance is also growing on industrial scenarios. This experiment has been deployed on an industrial prototype based on a Raspberry Pi, which is a widely used platform for gateways and Industry 4.0 scenarios, as explained in Chapter 1. This platform also allows to test a wide set of implementations comparing to more resource constrained devices. In this scenario, two Raspberry Pi 3 model B boards have been used, connected using WiFi to a local 56 Mbps router, as shown in Figure 3.16.

Raspberry Pi 3 model B features a Quad Core ARM Cortex A53 CPU, running at 1.2GHz and 1GB of RAM at 900MHz. Raspbian Wheezy has been the selected OS, installed on a 16 GB class 10 microSD card. The virtual used environments have been JRE 1.7 for Java, Python 2.7 and Node.js 4.6.1.

Figure 3.16: Experiment setup.

For testing, servers and clients have been generated with the libraries, using them as they come, unmodified. The only exception for this is FreeCoAP, that only allows IPv6, so it has been ported to IPv4. The compatibility between the servers and clients and their performance have been tested. All the combinations have been tested for their compatibility, except h5.coap server and microcoap client as the libraries do not provide support for generating them. The requests include a one byte payload, while the response payloads are 7 and 8 bytes alternatively. However, it is important to note that at the time of this experiment, CoAPthon does not allow to add any payload to 2.04 type responses [100], thus, it sends fewer bytes in the response, saving time and device resources. Regarding the measured metrics, the analysis includes memory usage, CPU consumption and latency. For measuring ROM usage, both executable and library file sizes have been summed. RAM and CPU consumption have been analyzed using a GNU/Linux tool named *time*, which shows statistics of an execution. To measure the latency, the *Tcpdump* network sniffer has been used in the client, to measure the Round Trip Time (RTT). 50 request have been sent for each client-server combination, with a one-second interval between sent messages.

3.4.7 Results

After the explanation of the experimental setup, in this section the results are presented. The analyzed metrics have been the compatibility, the latency and the resource usage.

3.4.7.1 Compatibility

Table 3.11 shows the compatibility test results. PUT request have been sent from clients to servers, in order to observe the responses.

Client \ Server	libcoap	smcp	microcoap	FreeCoAP	Californium	node-coap	CoAPthon	CoAPy
libcoap	✔	✔	✔	✔	✔	✔	✔	–
smcp	✔	✔	✔	✔	✔	✔	✔	–
FreeCoAP	✔	✔	✔	✔	✔	✔	✔	–
Californium	✔	✔	✔	✔	✔	✔	✔	–
h5.coap	✔	✔	✔	✔	✔	✔	✔	–
node-coap	✔	✔	✔	✔	✔	✔	✔	–
CoAPthon	✔	✔	✔	✔	✔	✔	✔	–
CoAPy	–	–	–	–	–	–	–	✔

Table 3.11: Implementation compatibility results.

As explained before, the preconception is that all by CoAPy are compatible, which can be corroborated with the experimental test. For this test, no tool has been used, but when adding a network packet analyzer for the performance test, an issue between the combination of h5.coap client and CoAPthon server has been found, not obvious on the compatibility test. CoAP messages use two kind of identifiers: message ID and token. Message ID pairs a message with its corresponding

acknowledgement, while tokens are used for more generic purposes and can be empty. The issue is that the CoAPthon server was generating a token when the client sent an empty one, contrary to RFC 7252, and h5.coap was checking the tokens to match in order to accept the acknowledgment. This results on the acknowledgement being rejected and the request being retransmitted until the maximum allowed number of retries are reached. The issue has been reported and later corrected in the CoAPthon's repository [101].

3.4.7.2 Latency

After the compatibility test, the latency has been analyzed. Figure 3.20, Figure 3.18, Figure 3.19, and Figure 3.17 present the measured maximum, median, mean, and minimum latency values in milliseconds for every server-client combination. The median and maximum values represent a normal and a worst case scenario values. The mean and minimum values are not as useful, but they represent another type of normal value and the best case scenario respectively.

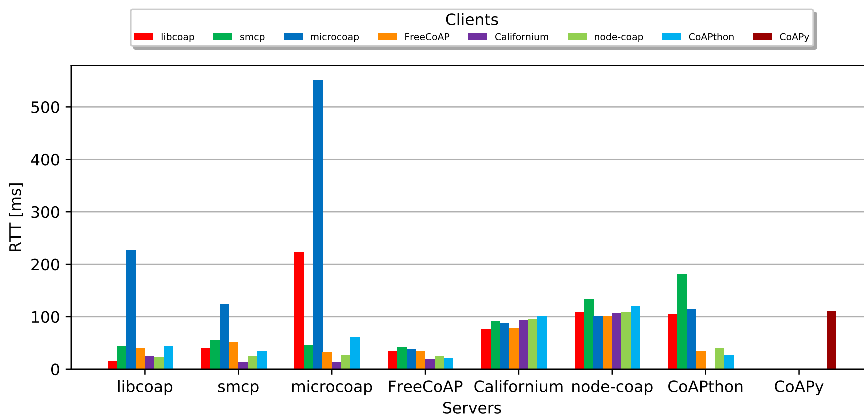


Figure 3.17: Maximum RTT in ms.

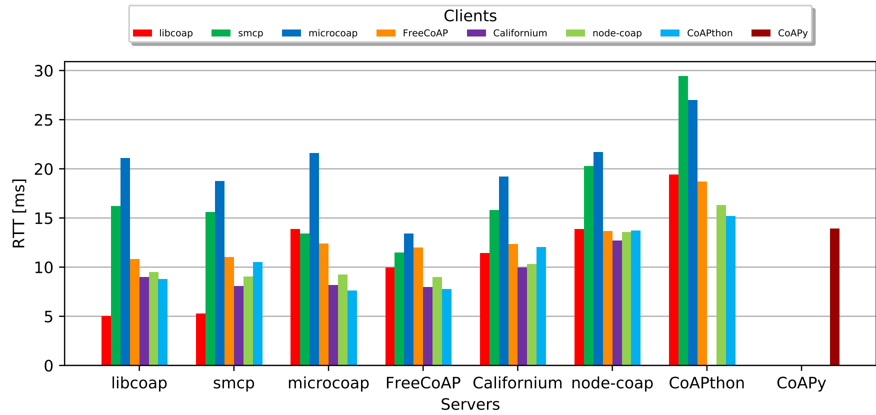


Figure 3.18: Median RTT in ms.

In these charts, the horizontal axis contains the different server implementations and the vertical bars represent the round-trip time for each of the clients in milliseconds.

As expected, libcoap, smcp and microcoap servers are the fastest, due to C not needing an additional abstraction layer and being a more optimized executable. Java (Californium), Node.js (h5.coap and node-coap) and Python (CoAPthon) implementations show surprisingly good results as clients, even in comparison to most C implementations, being libcoap the fastest library for most cases.

3.4.7.3 **Resource Consumption**

Once compared the latency times, the next test is to measure the system resource consumption. Table 3.12 shows how many bytes the applications need on ROM. Some libraries are installed (i.e., libcoap, smcp, h5.coap, node-coap, CoAPthon, and CoAPy) while others (i.e., microcoap, FreeCoAP and Californium) are included in the applications themselves. Regarding the installed libraries, the table shows either

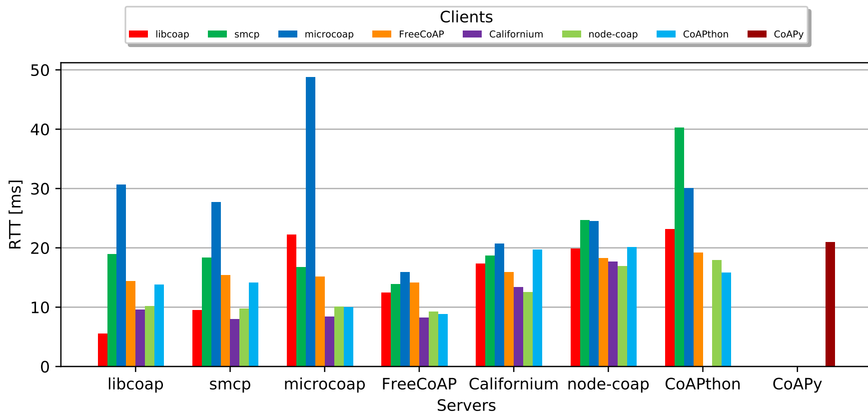


Figure 3.19: Mean RTT in ms.

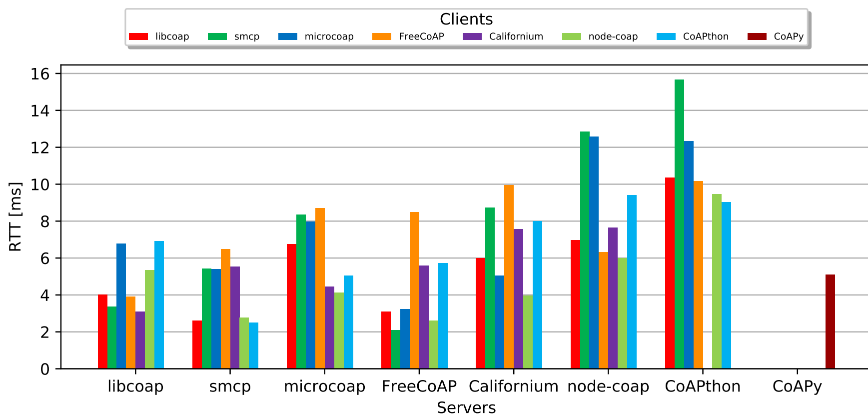


Figure 3.20: Minimum RTT in ms.

the *.a* files' or *lib* folder sizes in the library row and the sizes of the client and the server on their respective row. In the applications that do not need to be installed, the executable client and servers include the library.

	libcoap	smcp	microcoap	FreeCoAP	Californium	h5.coap	node-coap	CoAPthon	CoAPy
Library	296,150	383,366	-	-	-	106,097	47,341	277,095	76,074
Server	21,812	18,356	18,700	39,772	4,257,012	-	1,329	2,508	4,563
Client	33,328	22,684	-	31,616	4,257,329	3,621	883	1,672	1,794

Table 3.12: ROM usage in bytes.

C implementation have bigger executable and library sources than Python or JavaScript, while Java executables are considerably heavier. The size of I/O libraries has not been taken into account except for Californium and using non-native languages require adding a runtime environment (Node.js), an interpreter (Python) or a Virtual Machine (Java), which depends on the selected one, but in any case is heavier than the size difference.

Finally, the CPU and RAM usage of the implementations has been measured and presented in Table 3.13. For this experiment, the clients make 1000 requests. The Gnu/Linux tool *time* has been used for ollecting the data, executing the */usr/bin/time -v "server/client execution command"* command. This command outputs the peak usage of the RAM in KBs and the total CPU time in seconds for the application, both in user and system space.

As it can be seen in Table 3.13, C implementation are the fastest and most lightweight. All four tested C implementations show similar RAM consumption (about 3,3 MB), while Californium and both Node.js libraries require around 10 times more. Python libraries are more lightweight than these, but still far behind C ones. CPU usage results,

		libcoap	smcp	microcoap	FreeCoAP	Californium	h5.coap	node-coap	CoAPthon	CoAPy
Server	User Time	0.09	0.04	0.13	0.06	2.30	-	2.35	6.60	0.99
	System Time	0.13	0.08	0.09	0.13	0.27	-	0.24	1.23	0.16
	Peak RAM	3,252	3,232	3,236	3,240	24,492	-	31,008	14,348	8,332
Client	User Time	0.06	0.10	-	0.10	4.66	4.22	2.73	5.77	4.96
	System Time	0.08	0.31	-	0.21	0.41	0.62	0.21	1.25	0.13
	Peak RAM	3,240	3,312	-	3,256	29,676	37,732	34,484	12,924	10,644

Table 3.13: CPU and RAM usage in second and Kbytes.

especially measuring the User Time, also favor C implementations, as they are much faster due to being compiled and executed natively.

After testing the CoAP libraries, it can be confirmed that libcoap, smcp, microcoap, FreeCoAP, Californium, node-coap and CoAPthon are compatible, while CoAPy is not, because it is based on an outdated draft of CoAP's specification. Regarding server performance, C implementations prevail over the rest, specially libcoap and smcp being the fastest, while microcoap's and FreeCoAP's memory requirements are one order of magnitude lower. However, microcoap does not include all the specifications of the RFC 7252 and FreeCoAP does not handle response code generation tasks and URIs transparently. At the client side, where disk space requirements are not so critical, the Java, Node.js and Python implementations are surprisingly close to libcoap and smcp in terms of speed. Moreover, thanks to the higher abstraction level of their languages, Californium (Java), CoAPthon (Python), h5.coap and node-coap (Node.js) are also recommended for CoAP clients. In this subcontribution the strengths and weaknesses of each implementation are pointed out, but it is up to the system designers to choose the more adequate library for their application.

3.5 Conclusion

After presenting the comparison on different CoAP libraries, this section concludes this chapter, where two main subcontributions have been presented related to IoT communication protocols. These subcontributions aim to analyze IoT protocols and after selecting one, analyze the available implementations. To do so, first, a comparison of MQTT and CoAP in resource constrained devices has been carried out, and after determining that CoAP performs better, different CoAP implementations have been compared in terms of features and performance.

For the first subcontribution, an analysis comparing the overhead, scalability and latency of MQTT and CoAP using different QoS has been provided. Under the conditions of the experiment (i.e., lossless communication with no congestion on the network), CoAP performs better. It requires fewer bytes to transfer the same payload and has a shorter delay regardless of the QoS. However, the difference in overhead is not that big, specially on one-to-many communications, but the difference on latency is bigger. MQTT latencies are in the order of milliseconds, while CoAP latencies are as fast as hundreds of microseconds. Although the difference is strongly affected by the underlying transport protocols, they can not be taken out of consideration as they are intrinsic to the message exchange.

Under network traffic and/or strict latency requirements, CoAP has proven to be lighter and faster than MQTT. Nevertheless, if MQTT is capable of fulfilling the overhead and time specifications of an application, its maturity, easier configuration and publisher-subscriber decoupling, among other characteristics, have to be taken into account when designing an application.

For the second subcontribution, the analysis focuses on CoAP and its implementations. The features, behavior and performance of such implementations are analyzed and it can be confirmed that libcoap, smcp, microcoap, FreeCoAP, Californium, node-coap and CoAPthon are compatible, while CoAPy is not, due to implementing an outdated draft. As explained on the previous section, C implementation stand out on the server side performance. While libcoap and smcp are faster, microcoap and FreeCoAP are lighter. However, the last two have other downsides (microcoap does not implement all RFC 7252, and FreeCoAP does not provide tools for generating resources). On the client side, the devices are more powerful and Java, Node.js and Python implementations perform surprisingly close to libcoap and smcp in terms of speed. Moreover, thanks to the higher abstraction level of their languages, Californium (Java), CoAPthon (Python), h5.coap, and node-coap (Node.js) can be recommended for generating CoAP clients.

For the security perspective, there is still a lot of work to be done. Although being more mature, OpenSSL, GnuTLS and mbed TLS were not originally developed for resource constrained devices, hence, they can be resource demanding. On the other hand, mbed TLS and tinydtls were designed for resource constrained devices, but their implementation is still not completed.

However, the CoAP standard and its implementations are ready to be used in industrial applications, and to continue the work of this thesis, the next chapter makes use of CoAP on a Smart Grid domain.

During Stone Age, everyone knew everyone else in the same cave. There was no need for security. In the Middle Ages, they lived in castles or villages surrounded by town walls. History has shown that these security models don't work. The Internet has entered the Middle Ages.

Guy Leduc

4

Interoperability through IEC 61850

After the analysis of different IoT protocols and the available CoAP implementations, in this chapter the interoperability is analyzed using IEC 61850, a standard designed by the IEC for electrical substations. For that, a mapping of IEC 61850 to CoAP is proposed in this chapter, to then implement such mapping and evaluate its performance against HTTP and SOAP implementations, in terms of latency and the required bytes for the message exchange. Then, the IoT stack is completed, changing the resource representation from JSON to CBOR and a new evaluation of its performance is provided.

Contents

4.1 Introduction 115

4.2 Related Work 118
4.3 IEC 61850 122
4.3.1 Basic information model 123
4.3.2 Control blocks for additional functions 124
4.4 Mapping IEC 61850 to CoAP 126
4.4.1 Basic Services 127
4.4.2 Reporting Services 130
4.4.3 Logging Services 132
4.4.4 Setting Services 133
4.4.5 Eventing Services 134
4.4.6 Sampled Value Transmission Services 136
4.4.7 General Functionality Services 137
4.4.8 Other Services 139
4.4.8.1 Control Model Services 139
4.4.8.2 Time and Time Synchronization Services 139
4.5 Implementation of the mapping 139
4.6 Evaluation 145
4.6.1 Latency 147
4.6.2 Overhead 149
4.7 Lightweight Resource Representation 151
4.8 Validation 152
4.8.1 Latency 152
4.8.2 Overhead 155
4.9 Conclusion 158

4.1 Introduction

As explained in Section 1, one important application domain for Industry 4.0 is energy. In this regard, over the past few years, more active components such as photovoltaics, residential batteries, and home automation systems are being connected to electric power distribution grids. Some of them actively generate energy, leading to a paradigm change in power grids from distribution of energy to consumers to the need for active management of the resulting Smart Grids. Distribution grids used to be operated passively, as the centralized distribution does not need extensive control, but the inclusion of the Distributed Energy Resources (DERs) increases their complexity [46]. Thus, Smart Grids are being pushed for the integration of Supervisory Control and Data Acquisition (SCADA) functionalities to facilitate the management of power grids.

In order to enable SCADA functionalities in power grids, operators need to equip distribution substations that include relevant sensors and connect them to the back-end infrastructure. To guarantee the successful retrofitting on legacy systems, the deployment cost must be low, with minimal maintenance efforts, and the performance must be good with low latency, specially on low bandwidth networks. As the physical space on the electrical substation is limited, the industry is turning toward wireless communication technologies, especially in remote deployments. Wireless Personal-Area Networks (WPANs) are suggested to connect devices within substations, while Low-Power Wide-Area Networks (LPWANs) are expected to link substations to back-end systems. Counterintuitively, power line communication technologies are not suitable for electrical substation, as indeterminate impedance variations and cross-talk make them too unreliable [147].

In this context, the International Electrotechnical Commission (IEC) defined the IEC 61850 [73] standard, which was originally designed for modeling, controlling, and monitoring electrical substation automation systems and has been extended to support new power system domains such as Wind Power Plants or Hydroelectric Power Plants. To achieve that, the standard defines a data model, reporting schemes, events, settings, sample data transfers, commands and data storage functions.

The IEC 61850 specification defines a communication model and some mappings are included in the standard: Manufacturing Message Specification (MMS) [75], serial communication [76] and ISO/IEC 8802-3 Ethernet [77]. In addition to these protocols, others have been proposed (e.g., CORBA, HTTP-REST), which are analyzed in the next section. However, with the increasing adoption of the IoT, new lightweight network protocols have been defined, to make the system less resource demanding in terms of energy, RAM and CPU usage, and network bandwidth [59]. Even though the energy consumption is not a big issue for electrical substations, deploying lots of devices can be economically costly if their capabilities are high. Hence, besides allowing to build cheaper devices with less computing capability, the resource constrained IoT devices embrace the aforementioned lightweight approach where lots of small devices exchange information in real-time with minimal impact on the performance [91]. This is why the smart grid domain is adopting IoT protocols in all the layers of the networking protocol stack, as can be seen in examples like [225] or [16].

As stated on Chapter 3, MQTT and CoAP are some of the most widespread protocols [98], and are being adopted in many different environments and use cases outside of industry, such as IKEA's Trådfri Lights for CoAP [209] and Facebook's Messenger for MQTT [141].

However, the Industry is also adopting the IoT, as shown by [27] and [95]. For instance, at Ikerlan, MQTT has also been used for industrial machinery and equipment goods monitoring applications. Although MQTT was defined before CoAP and can be considered more mature, CoAP is also gaining popularity for certain scenarios. In control use cases, CoAP's client-server model is more appropriate while for monitoring, MQTT's publish-subscribe model is more straightforward. However, MQTT needs an intermediary broker which adds a hop for every message transmission. CoAP can also be used in conjunction with the Observe extension for publish-subscribe communication model, with no need for a broker. Taking all these arguments into consideration, for this contribution, CoAP has been selected as the communication protocol.

In the related literature, presented in the next section, different protocols have been proposed to map IEC 61850. However, there is no robust mapping to CoAP that covers all the functionalities. Hence, in this contribution, a full mapping to CoAP is proposed and implemented. After implementing it, a first evaluation is provided, where its performance is compared to HTTP and SOAP implementations. After this first evaluation, the CoAP implementation is improved, adding the Concise Binary Object Representation (CBOR) as the resource representation format, completing the IoT stack. Having this improvement, a new comparison is provided to validate the approach.

The rest of this chapter is organized as follows. First, the related literature is analyzed in Section 4.2. Then, the IEC 61850 standard is explained, to continue with the proposed mapping on Section 4.4. After describing the mapping proposal, the implementation is presented, along with a first evaluation. Section 4.7 presents CBOR and explains

how it has been included in the mapping, along with a new comparison, this time using the CBOR resource representation format. Finally, some conclusions are presented in Section 4.9.

4.2 Related Work

The IEC 61850 standard only includes the MMS mapping as the application layer protocol, but it does not exclude others. Some other derived standards, e.g., IEC 61400, provide other mappings, such as Web Services [71]. There are several studies in the scientific literature that provide a high level analysis of the features using different approaches in Smart Grid environments [162] [3]. Regarding the used protocols, although not part of standards, several works have been published proposing others, as shown in Table 4.1. In the table, the proposed mappings are listed, including the protocols, data representation formats, application domains, measured metrics, and the number of mapped functions out of the 61 functions defined in the standard.

Work	Protocol	Data format	Application Domain	Metrics	# functions
[177]	CORBA	IDL	-	-	-
[24]	DDS + CORBA	IDL	-	Jitter, latency	-
[25]	DDS	IDL	-	-	-
[15]	DDS	IDL	-	Reliability (Received/sent)	-
[159]	HTTP-REST	XML	μCHP & Electric Vehicle	-	16/61
[157]	HTTP-REST	JSON	Smart Home, Ambient Media & Health Management	-	18/61
[70]	XMPP	XML	Distribution STATic COMpensator (DSTATCOM)	-	11/61
[185]	CoAP	-	OPNET simulator	Packets/second, data size, traffic, delay	7/61

Table 4.1: IEC 61850 mappings: communication protocol, data format, application domain, metrics and number of mapped functions.

During the early development of the IEC 61850 standard, Sanz et al. [177] proposed to standardize the communication protocol for different manufacturer products. At that time, each company had its own proprietary protocol which was not interoperable with other companies' solutions. Their proposed solution was to use the Common Object Request Broker Architecture (CORBA). The Interface Definition Language (IDL) for the object interfaces. The work focuses on demonstrating the importance of the mapping, instead of the description of the mapping itself.

Calvo et al. [24] proposed combining CORBA and DDS in a middleware: CORBA for client-server functions and DDS for publish-subscribe ones. This way, the client request can be responded using CORBA and for periodic (sampled measured values) and aperiodic (events) notifications DDS can be used. In this proposal, the upper-level entities (*Servers*, *Logical Devices*, and *Logical Nodes*) are defined as CORBA objects, while lower-level entities (*Data*, *Data Attributes* and *Data Sets*) are mapped to internal objects. DDS is used to map *Generic Substation Events (GSEs)*, *Sampled Measured Values (SMVs)* and reporting and logging services.

Continuing this work, Calvo et al. also propose to use only DDS [25], configuring DDS mechanisms to accommodate the needs for client-server communications as well as publish-subscribe ones. For modeling the data, IDL is the proposed approach, inherited from CORBA. For the functions, they take into account the nature of the services: publish-subscribe model for periodic and aperiodic process messages and client-server for non-real time operations. IEC 61850 defines both types of communications, so a complete solution needs to use two different communication protocols with different commu-

nication paradigms or to map the client-server communication into a publish-subscribe mode, like the path followed in this proposal. Configuring DDS mechanisms in accordance, the authors achieve a similar behavior to a client-server communication.

Bi et al. [15] argue that MMS, CORBA and Web Services mappings of IEC 61850 run on TCP/IP protocols only over the Ethernet LANs inside the individual substation and are not capable of interconnecting substations. Thus, they are confined inside the substation and are not able to provide connectivity over large area networks. In order to overcome that issue, the authors present JMS and DDS as alternative and then focus their work on a DDS mapping. Not only do the authors provide a mapping proposal, but they also discuss a viability study for the presented solution. However, the mapping is not explained, the work focuses on providing a summary of the covered functionalities along with a simple reliability benchmark (ratio of received to sent packages).

The first proposal for using a more generic communication protocol is the one by Pedersen et al. [159]. In this proposal, the authors map IEC 61850 to REST, more specifically HTTP. A few of the functions are mapped and use XML as the data format. However, both HTTP and XML have been designed for devices with large resource capabilities, which is a downside for resource constrained environments, where more lightweight protocols and data representation formats are preferred. Along with the mapping, the authors present two case studies, i.e., interfacing with a *micro Combined Heat and Power* (μCHP) system and interfacing with an *Electric Vehicle*. However, the authors do not provide enough information to convincingly demonstrate the suitability of the proposed protocol.

Parra [157] also proposes to use a RESTful approach with HTTP, with JSON for the data representation, which is lighter than XML. In this proposal, the mapped services are numerous more than Pedersen et al., but some functionalities, such as settings, eventing, value transmission, logging and file transfer are not covered. The proposal is validated using three different application domains, i.e., a Smart Home prototype, an Ambient Media prototype and a Health Management prototype.

Another general communication protocol is proposed by Hussain et al. [70], which model a Distribution STATic COMpensator (DSTATCOM) and represent the data in XML to then send the XML via XMPP. They map some basic functions and eventing and sampled value transmission services, but there is no performance analysis in their paper.

Shin et al. [185] are the first to propose to map the IEC 61850 standard to an IoT protocol, concretely CoAP. They offer a simple and robust approach on the description of the methods, but they do not follow the RESTful principles [52] of CoAP. For instance, they use different URIs for requests to the same resource, instead of using different methods (e.g., GET, PUT or POST) with the same URI. Besides, Shin et al. map a reduced set of functions. The focus of Shin's et al. contribution is a set of simulations to provide a diagnose of the performance comparing against other SOAP and MQTT implementations, so the mapping is not extensively explained, nor do they map many functions. Another negative point of this approach is that a client requires a previous knowledge of the entire system's structure, as it is not possible to discover resources. In order to compare the performance, the authors measure the sent packets per second, the data size, the received traffic, and the network delay.

Analyzing the related literature, it can be seen that all the analyzed proposals have downsides: MMS is based on Ethernet and is designed to work only on LANs, HTTP is very verbose and adds a lot of overhead, CORBA requires complementary protocols and DDS is not designed for resource-constrained devices. The only previous approach using CoAP is not extensive and it does not follow the RESTfulness of CoAP. This is why in this contribution, a new, more complete and lightweight mapping of IEC 61850 to CoAP is proposed, following the RESTful architecture, along with a performance analysis. A direct comparison against other proposal is not entirely fair as each of them uses different hardware, so the experiments are carried out without the data from other contributions.

4.3 IEC 61850

After analyzing the related literature on using the IEC 61850 with different communication protocols, in this section the IEC 61850 standard is explained in more depth than in Chapter 2. As explained there, IEC 61850 is a standard by the IEC for modeling, controlling and monitoring electrical substations. However, it has evolved to also fit Smart Grid use cases. As described in Section 2.3, the main parts of the IEC 61850 standard are an Information model, the ACSI services and the mappings to specific communication protocols, more specifically to MMS.

The *Information Model* of the IEC 61850 standard includes two main classes for building the virtual view of real world elements, i.e., the Basic Information Model (BIM) and Control Blocks for Additional Functions (CB). The BIM defines how to organize the elements and their information, while the CBs are specialized classes for setting

and managing additional functionality. Figure 4.1 presents how the elements are related both in the BIM and CBs.

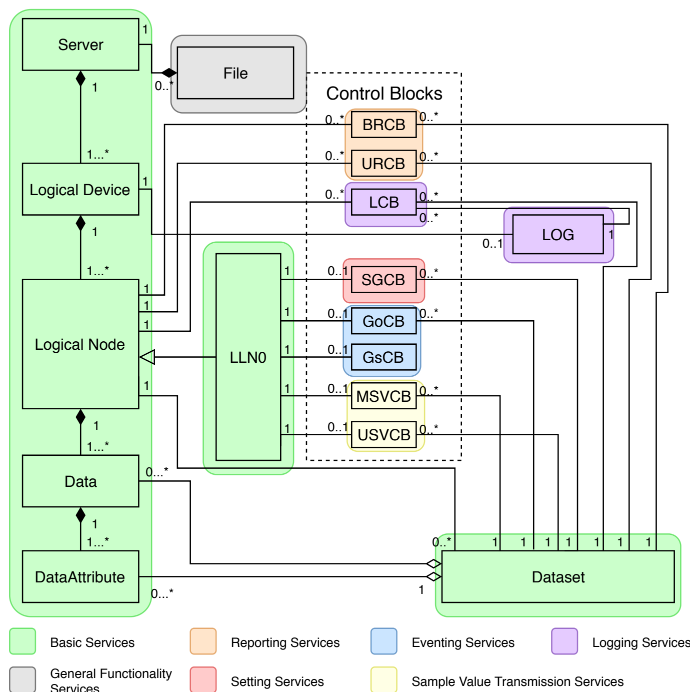


Figure 4.1: The Information Model of the IEC 61850 standard.

4.3.1 Basic information model

The BIM defines how to organize real world elements and their information in a simple and structured basis. The elements that conform the BIM are marked in green in Figure 4.1.

- **Servers** expose the IED to the outside and it is composed of a set of Logical Devices.

- **Logical Devices (LD)** are virtual representations of real devices. They are composed of sets of Logical Nodes.

- **Logical Nodes (LN)** virtually abstract each application functionality. All LDs include a special LN named **Logical Node Zero (LLN0)**, which represents common data of the LD.

- **Data** represent specific information associated to an LN about the physical world.

- **DataAttributes** are typed information pieces of a Data instance, e.g., value or timestamp. The values of a DataAttribute are defined by a type.

- **Datasets** group existing Data and DataAttributes.

4.3.2 Control blocks for additional functions

Outside the BIM, IEC 61850 also defines some CBs, represented in different colors in Figure 4.1, which are specialized classes for configuring and managing a set of additional functionalities related to the information model:

- **Reporting** (orange) blocks allow to define the conditions for generating reports with Datasets. There are two types of reporting CBs, i.e., *Buffered Report Control Blocks (BRCP)*, which use mechanisms to guarantee that the reports arrive to their destination, and *Unbuffered Report Control Blocks (URCB)*, which work on a best effort basis.

- **Logging** (purple) services use the *Log Control Block (LCB)* class to allow configuring Datasets to be logged and the conditions for including the information on the logs.

- **Setting** (red) has the *Setting Group Control Block (SGCB)* class, which allows defining groups of settings and switch the active group of settings.

- **Eventing** (blue) functions use the publish-subscribe communication model for delivering events generated in the device. There are two types of events, i.e., *Generic Object Oriented Substation Event (GOOSE)* managed by *GOOSE Control Block (GoCB)* to support the delivery of DataAttributes grouped in Datasets; and *Generic Substation State Event (GSSE)* managed by *GSSE Control Block (GsCB)* for delivering basic state change information.

- **Sampled Values** (yellow) CBs manage the transfer of sampled information in Datasets of DataAttributes in a time controlled way. It can be implemented in two ways: using multicast communication with a *Multicast Sample Value Control Block (MSVCB)* or unicast communication with an *Unicast Sample Value Control Block (USCVB)*.

All the elements in the model (including the elements both in the BIM and the CBs) have a unique identifier that identifies them within the parent object. Thus, they have an absolute reference concatenating parent objects' and the element name and can be addressed unequivocally in the entire system.

4.4 Mapping IEC 61850 to CoAP

After describing the IEC 61850 standard, the services of the IEC 61850 are described along with a proposal to map them to CoAP. As mentioned in the previous section, IEC 61850 defines a group of services, i.e., Abstract Communication Services for interacting with the device. Although there already exist some mappings of the IEC 61850 to different communication protocols presented in Section 4.2, these are based on legacy protocols that have two main drawbacks:

- They are protocols not designed for limited resource devices or low bandwidth data connections, that can have communication problems due to large overheads compared to the data.
- These legacy protocols present interoperability problems with existing IoT protocols, making it difficult to carry out large-scale deployments, especially if devices and services need to be integrated across vertical application domains.

As previously stated, CoAP has been the protocol selected for this contribution. CoAP is a good fit because it does not only use the client-server communication model, but it also offers a publish-subscribe mechanism using the Observe extension. This makes CoAP very aligned with the services described in the standard, since both communication models are used (e.g., basic services with client-server and publish-subscribe with reporting).

One first comment on using CoAP with the IEC 61850 is that CoAP defines the "/" path as an entry point for a server while IEC 61850 does not require such an entry point. Hence, it could be implemented as a server information or an element listing URI but this is out of the scope of this mapping.

Services can be grouped according to CB. A subsection for each group follows in the next lines, with the explanation of the services, along with the mapping. Some functions are quite straightforward to map, but others include a short explanation.

4.4.1 Basic Services

Basic services are the ones related to the classes of the BIM. They allow to interact with the elements of the BIM in order to manage their information. Table 4.2 presents the available functions along with the proposed mapping to CoAP URI and methods.

Function	URI	Method
	Server	
GetServerDirectory	coap://{host}/LDs	GET
GetServerDirectory	coap://{host}/Files	GET
	Logical Device	
GetLogicalDeviceDirectory	coap://{host}/LDs/{LD}	GET
	Logical Node	
GetLogicalNodeDirectory	coap://{host}/LDs/{LD}/{LN}/{ACSIClass}	GET
GetAllDataValues	coap://{host}/LDs/{LD}/{LN}/AllValues[?FC={fc}]	GET
	Data	
GetDataValues	coap://{host}/LDs/{LD}/{LN}/Datas/{Data}	GET
SetDataValues	coap://{host}/LDs/{LD}/{LN}/Datas/{Data}	PUT
GetDataDirectory	coap://{host}/LDs/{LD}/{LN}/Datas/{Data}/Directory	GET
GetDataDefinition	coap://{host}/LDs/{LD}/{LN}/Datas/{Data}/Definition	GET
	DataSet	
GetDataSetValues	coap://{host}/LDs/{LD}/{LN}/Datasets/{Dataset}	GET
SetDataSetValues	coap://{host}/LDs/{LD}/{LN}/Datasets/{Dataset}	PUT
CreateDataSet	coap://{host}/LDs/{LD}/{LN}/Datasets	POST
DeleteDataSet	coap://{host}/LDs/{LD}/{LN}/Datasets/{Dataset}	DELETE
GetDataSetDirectory	coap://{host}/LDs/{LD}/{LN}/Datasets/{Dataset}/Directory	GET

Table 4.2: Mapping of basic services to CoAP.

- The **Server** class provides:

GetServerDirectory to get a list of the names of all *LDs* or *Files*.

To map this service, two differentiated URIs are needed, one for *LDs* and another one for *Files*.

- **Logical Devices** class offers:

GetLogicalDeviceDirectory to retrieve the list of *LNs* of the referenced *LD*.

Similar to *GetServerDirectory*, only a GET request to the specific *LD*'s URI is needed, i.e., `coap://example.org/LDs/Windturbine01`.

- **Logical Node** class includes the following services:

GetLogicalNodeDirectory retrieves the list of *Objects* of the *LN*. The client selects the type of *Object* between *Data*, *Dataset*, *BRCB*, *URCB*, *LCB*, *LOG*, *SGCB*, *GoCB*, *GsCB*, *MSVCB* or *USVCB*.

GetAllDataValues enables to retrieve all *DataAttribute* values of all *Data* of the *LN*.

Adding a query in *GetAllDataValues* allows to filter the *Datas* according to their Functional Constraint (FC).

- **Data** class offers:

GetDataValues to allow to read a complete or a part of a *Data*.

SetDataValues makes it possible to set a complete or a part of a *Data*.

GetDataDirectory retrieves the list of all *DataAttributeNames* of the referenced *Data*.

GetDataDefinition enables the client to get the complete list of all *DataAttribute* definitions of the *Data*.

A new level of URI path is required in these services to differentiate between the three GET services. The GET requests for *GetDalaValues* uses the general URI of the *Data*. This way the GET and PUT requests can be sent to the same URI. For more concrete information of the data, “Directory” and “Definition” have been the selected URI path levels.

- **DataSet** class provides:

GetDataSetValues to return the values of the *DataAttributes* accessible by the referenced *Dataset*.

SetDataSetValues sets the values of the *DataAttributes* made visible by the referenced *Dataset*.

CreateDataSet enables the client to create a *Dataset* with a list of members defined with the *Functionally Constrained Data (FCD)* or *Functionally Constrained Data Attribute (FCDA)*.

DeleteDataSet deletes a *Dataset*.

GetDataSetDirectory retrieves the list of the *ObjectReferences* of all dataset members referenced by the *Dataset* made accessible to the client.

To map all these services, the same structure as in *Data* services has been followed, with a new level of URI for the *GetDataSetDirectory* service and different methods with the *Dataset*’s base URI to read, delete and update. For

creating a *Dataset*, one less URI level has been defined, as the *Dataset* that needs to be created does not exist yet, and thus, it does not have a URI.

4.4.2 Reporting Services

The first CBs described in this section are the CBs related to reporting services, i.e., BRCB and URCBs. These report internal information to the outside of the system. Table 4.3 lists BRCB and URCB services, along with their mapping to CoAP.

Function	URI	Method
	BRCB	
Report	coap://{host}/LDs/{LD}/{LN}/BRCBs/{BRCB}/Reports	Not.
GetBRCBValues	coap://{host}/LDs/{LD}/{LN}/BRCBs/{BRCB}	GET
SetBRCBValues	coap://{host}/LDs/{LD}/{LN}/BRCBs/{BRCB}	PUT
if enable	coap://{host}/LDs/{LD}/{LN}/BRCBs/{BRCB}/Reports	GET+Obs.
	URCB	
Report	coap://{host}/LDs/{LD}/{LN}/URCBs/{URCB}/Report	Not.
GetURCBValues	coap://{host}/LDs/{LD}/{LN}/URCBs/{URCB}	GET
SetURCBValues	coap://{host}/LDs/{LD}/{LN}/URCBs/{URCB}	PUT
if enable	coap://{host}/LDs/{LD}/{LN}/URCBs/{URCB}/Report	GET+Obs.

Table 4.3: Mapping of reporting services to CoAP.

- **BRCB**s offer:

  Report for sending reports generated in the server to the client.

  GetBRCBValues for allowing the clients to retrieve attribute values of a *BRCB*

  SetBRCBValues sets attribute values of a *BRCB*.

- **URCB**s provide:

Report sends reports from the server to the client.

GetURCBValues to retrieve attribute values of an *URCB*.

SetURCBValues for allowing clients to set attribute values of an *URCB*.

For the mapping of the functions of *BRCB* and *URCB*, two approaches have been considered. On the one hand, if CoAP were to be used as is, a new client and server would be needed with reverse roles compared to the IED. The new client (server on the current system) would send PUT messages to the server (client on the current system) when a report needed to be sent. On the other hand, this complexity can be avoided using the Observe extension of CoAP, which is the selected approach. This extension enables clients to subscribe to changes in the server, and in this case, the change is the generation of a report. To support this feature, when setting values on a reporting block, the reporting is enabled, an extended GET (GET+Obs. in Table 4.3) is required after setting it. This way, the client subscribes to the events and the *Report* service sends the reports when they are generated.

When selecting URIs, *BRCB*s keep several reports, while *URCB*s keep only one at a time, thus the differentiation on “Reports” or “Report”. In another matter, using a query with the FC is not necessary by CoAP means but the IEC 61850 standard defines it in its mappings. In early versions of the mapping, a query was included for consistency (i.e., `coap://example.org/LDs/Windturbine01/WROT/BRCBs/BRCB1?FC=BR`), but having added “BRCB” and “URCB” to the URI path, it was discarded (i.e., `coap://example.org/LDs/Windturbine01/WROT/BRCBs/BRCB1`). This also makes the URL more human friendly.

4.4.3 **Logging Services**

Using IEC 61850, the IEDs are able to log data. The logging services are presented in Table 4.4, along with the proposed mapping.

Function	URI	Method
	LCB	
GetLCBValues	coap://{host}/LDs/{LD}/{LN}/LCBs/{LCB}	GET
SetLCBValues	coap://{host}/LDs/{LD}/{LN}/LCBs/{LCB}	PUT
	Log	
QueryLogByTime	coap://{host}/LDs/{LD}/LLN0/LOG[?tstart={tstart}][&][?tstop={tstop}]	GET
QueryLogAfter	coap://{host}/LDs/{LD}/LLN0/LOG[?tstart={tstart}][&][?entry={entryid}]	GET
GetLogStatusValues	coap://{host}/LDs/{LD}/LLN0/LOG/Status	GET

Table 4.4: Mapping of logging services to CoAP.

- **LCB**s offer:

  GetLCBValues to retrieve attribute values of a *LCB*.

  SetLCBValues to allow clients to set attribute values of a *LCB*.

  The mapping of these services is straightforward, with GET and PUT methods to the base URI of the *LCB*.

- The **Log** block stores the logged data and has the following services:

  QueryLogByTime to retrieve a range of *Log* entries based on time ranges.

  QueryLogAfter to retrieve *Log* entries after a time and entry number;

  GetLogStatusValues to retrieve the attribute values of a *Log*.

There is only one *Log* for each *LD*, hence, the *LN* for common data of the *LD* (*LLN0*) is the one containing the *Log*. Regarding the functions, the difference between *QueryLogByTime* and *QueryLogAfter* is the variables used to define which log entry a client wants to retrieve. Thus, the same method and URI is used, with different optional queries. For *GetLogStatusValues* a new path level has been defined, “Status”.

4.4.4 Setting Services

Using setting services, a list of *Setting Group (SG)* can be stored, i.e., a set of values for configuration parameters. The **SGCB** is the CB that offers these services, listed on Table 4.5, along with the proposed mapping:

Function	URI	Method
SelectActiveSG	coap://{host}/LDs/{LD}/LLN0/SGCB?action=active	PUT
SelectEditSG	coap://{host}/LDs/{LD}/LLN0/SGCB?action=edit	PUT
SetSGValues	coap://{host}/LDs/{LD}/LLN0/SGCB	PUT
ConfirmEditSGValues	coap://{host}/LDs/{LD}/LLN0/SGCB?action=confirm	PUT
GetSGValues	coap://{host}/LDs/{LD}/LLN0/SGCB?buffer=[active/edit]	GET
GetSGCBValues	coap://{host}/LDs/{LD}/LLN0/SGCB/Status	GET

Table 4.5: Mapping of setting services to CoAP.

SelectActiveSG allows to load the values of the specified *SG* into the active buffer.

SelectEditSG allows to load the values of the specified *EditSG* into the edit buffer.

SetSGValues sets *Data* values to the *SG*.

ConfirmEditSGValues confirms that *SG* values set with *SetSGValues* shall overwrite old values.

GetSGValues enables clients to retrieve *Data* values of *SG*s.

GetSGCBValues retrieves the list of attribute values of the *SGCB*.

The *SGCB* is managed by a state machine and the actions defined in the queries (i.e., active, edit and confirm) are the transitions from one state to another. This is why they are defined as queries instead of new URI levels. In *SelectActiveSG* and *SelectEditSG* the selected *SG* is specified in the payload of the message. According to the IEC 61850 standard, only the *SG*s in the active or edit buffer can be accessed, so there is no need for different URIs, using the same one with different queries (i.e., buffer=active or buffer=edit) is enough.

4.4.5 Eventing Services

GoCB or GsCB are the CBs that control eventing services, allowing a fast distribution of internal events. Table 4.6 shows the proposed mapping of these services to CoAP URI and methods.

- **GoCB**s provide support for GOOSE changes of a Dataset.

  SendGOOSEMessage sends a *GOOSE* message over a *Multicast-Application-Association*.

  GetGoReference allows to retrieve the *MemberReferences* of specific members of the *Dataset*.

  GetGOOSEElementNumber allows to retrieve the member position of a selected *DataAttribute* in the *Dataset*.

Function	URI	Method
	GoCB	
SendGOOSEMessage	coap://{host}/LDs/{LD}/LLN0/GoCB/{event}	Not.
GetGoReference	coap://{host}/LDs/{LD}/LLN0/GoCB?offset={n}	GET
GetGOOSEElementNumber	coap://{host}/LDs/{LD}/LLN0/GoCB?ref={ref}	GET
GetGoCBValues	coap://{host}/LDs/{LD}/LLN0/GoCB	GET
SetGoCBValues	coap://{host}/LDs/{LD}/LLN0/GoCB	PUT
if enable	coap://{host}/LDs/{LD}/LLN0/GoCB/{event}	GET+Obs.
	GsCB	
SendGSSEMessage	coap://{host}/LDs/{LD}/LLN0/GsCB/{event}	Not.
GetGsReference	coap://{host}/LDs/{LD}/LLN0/GsCB?offset={n}	GET
GetGSSEDataOffset	coap://{host}/LDs/{LD}/LLN0/GsCB?label={label}	GET
GetGsCBValues	coap://{host}/LDs/{LD}/LLN0/GsCB	GET
SetGsCBValues	coap://{host}/LDs/{LD}/LLN0/GsCB	PUT
if enable	coap://{host}/LDs/{LD}/LLN0/GsCB/{event}	GET+Obs.

Table 4.6: Mapping of eventing services to CoAP.

GetGoCBValues allows to retrieve attribute values of the GoCB.

SetGoCBValues sets attribute values of the *GoCB*.

- **GsCB**s support GSSE basic state changes.

  SendGSSEMessage enables *GsCB*s to send a *GSSE* messages over a *Multicast-Application-Association*.

  GetGsReference retrieves the *DataLabels* of specific members of the *Collection* of the *GsCB*.

  GetGSSEDataOffset allows to retrieve the data position of a selected data in the Collection associated with the *GsCB*.

  GetGsCBValues retrieves attribute values of the *GsCB*.

  SetGsCBValues sets attribute values of the *GsCB*.

The mapping of getting and setting *GoCB* and *GsCB* values is straightforward. However, when setting values, an eventing configuration can be enabled or not. If it is, the client needs to subscribe to

the event (GET+Obs) and the same URI is going to send notifications when the event triggers. To retrieve references or indexes of events in both types of CBs, different queries are used.

4.4.6 Sampled Value Transmission Services

Sample values can be transmitted using MSVCB and USVCB CBs. Their services allow to distribute the sampled data of the IEDs. Table 4.7 lists said services along with the proposed mapping to CoAP:

Function	URI	Method
	Unicast	
SendUSVMessage	coap://{host}/LDs/{LD}/LLN0/USVCBs/{USVCB}	Not.
GetUSVCBValues	coap://{host}/LDs/{LD}/LLN0/USVCBs/{USVCB}	GET
SetUSVCBValues	coap://{host}/LDs/{LD}/LLN0/USVCBs/{USVCB}	PUT
if enable	coap://{host}/LDs/{LD}/LLN0/USVCBs/{USVCB}	GET+Obs.
	Multicast	
SendMSVMessage	coap://{host}/LDs/{LD}/LLN0/MSVCBs/{MSVCB}	Not.
GetMSVCBValues	coap://{host}/LDs/{LD}/LLN0/MSVCBs/{MSVCB}	GET
SetMSVCBValues	coap://{host}/LDs/{LD}/LLN0/MSVCBs/{MSVCB}	PUT
if enable	coap://{host}/LDs/{LD}/LLN0/MSVCBs/{MSVCB}	GET+Obs.

Table 4.7: Mapping of sample value transmission services to CoAP.

- **USVCB**s provide:

  SendUSVMessage sends sampled values from the server to the client.

  GetUSVCBValues retrieves attribute values of the *USVCB*.

  SetUSVCBValues sets attribute values of the *USVCB*.

- **MSVCB** blocks offer:

SendMSVMessage sends sampled values from the server to the client.

GetMSVCBValues retrieves attribute values of the *MSVC*.

SetMSVCBValues sets attribute values of the *MSVCB*.

All *USVM* and *MSVM* services use the CBs as the URIs. GET and PUT methods are used for getting and setting the values, and if when setting the transmission is enabled, the client needs to subscribe to the notifications using extended GET requests.

4.4.7 **General Functionality Services**

There are some additional services with general functionalities that are not related to any CB. Table 4.8 lists such services and their mapping to CoAP:

Function	URI	Method
	Access Control	
Associate	coap://{host}/Associations	POST
Abort	coap://{host}/Associations/{AssId}?action=abort	DELETE
Release	coap://{host}/Associations/{AssId}?action=release	DELETE
	File Transfer	
GetFile	coap://{host}/Files/{File}	GET
SetFile	coap://{host}/Files/{File}	POST
DeleteFile	coap://{host}/Files/{File}	DELETE
GetFileAttributeValues	coap://{host}/Files/{File}/Status	GET

Table 4.8: Mapping of additional services to CoAP.

- **Access Control** services manage sessions between the IEDs and external entities:

  Associate establishes an association with a specific server.

Abort abruptly disconnects a specific application association between a client and a server, discarding all pending service requests.

Release gracefully disconnects a specific application association between a client and a server, completing all pending service requests before termination, but without issuing new ones.

Associations can be created or deleted. To create an association, a POST request is used, while there are two options for deleting it. The same URI and method is used, but with a different query, which indicates how the association should be deleted. The query parameter is “abort” for an abrupt ending while “release” is used for graceful ending.

- **File Transfer** services allow the exchange of files between IEDs and external entities, and they include:

  GetFile transfers the contents of a file.

  SetFile allows to upload the contents of a file.

  DeleteFile deletes a file in the file storage of a server.

  GetFileAttributeValues retrieves the name and attributes of a specific file.

  The mapping of *Files* services is straightforward. Files can be read, created, and deleted with GET, POST, and DELETE requests. For retrieving the attributes of a file (i.e., size or last accessed time) the “Status” path is proposed.

4.4.8 Other Services

Outside of the CBs, there are some other general services: Control Model Services and Time and Time Synchronization Services. These services are out of scope of this work, but they are mentioned in this subsection.

4.4.8.1 Control Model Services

The services in this model allow to control devices in the IED. The services are: *Select*, *SelectWithValue*, *Cancel*, *Operate*, *CommandTermination* and *TimeActivatedOperate*.

4.4.8.2 Time and Time Synchronization Services

The IEC 61850 also describes a service for managing the time and its synchronization between devices. This service is *TimeSynchronization*.

4.5 Implementation of the mapping

Following the explanation of the functions and the proposed mapping, in this section the implementation of the mapping is presented. In order to facilitate the implementation and the comparison of the performance (showed in the next section), a proprietary tool by Ikerlan has been used that automatically generates the code using HTTP and WS-SOAP for the communication. The IEC 61850 standard defines 68 services, which have been mapped to CoAP in the previous section. However, this tool generates the code for just the BIM and the Reporting CBs (i.e., BRCBs and URCB) services, so for the CoAP implementation only these functions have been implemented. Nevertheless, with the implemented functions all the types of requests can be analyzed, i.e.,

retrieving, creating, updating, or deleting data or getting a notification when some event occurs. Figure 4.2 highlights the implemented blocks in black while the unimplemented ones in gray.

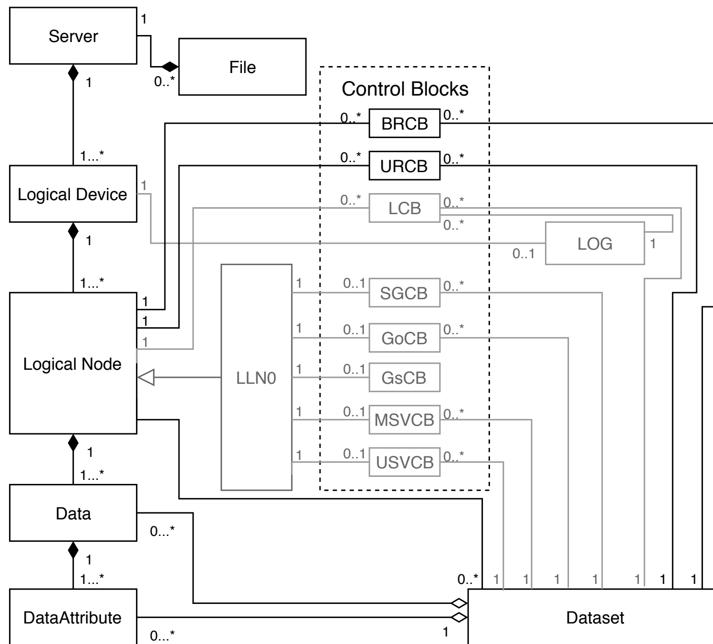


Figure 4.2: The Information Model of the IEC 61850 standard with the implemented blocks in black and the not implemented ones in grey.

Table 4.9 summarizes the implemented services from the previous section. A column has been added to include an identifier for each function in order to identify the function when presenting the results of the experiments. From the mapping presented in the previous section, two changes have been considered necessary:

- *SetFile* has been divided to differentiate if the File is new and needs to be created, or just needs to be updated. If it previously

existed, a PUT request is sent to the file's URI, while if it did not, a POST request is sent to */Files*.

- *GetDataValues* has been expanded to include a query with the functional constraint in order to integrate the implementation with the existing tool.

As previously explained, a tool that automatically generates the code has been used for the implementation. This tool is explained in more depth on Chapter 6, but for this chapter, some of its features need to be mentioned. The tool generates the code using a model that is represented with a tree view editor, as can be seen on the screenshot in Figure 4.3.

Prior to this work, the tool generated the code using HTTP and WS-SOAP for the communication, so the proposed CoAP mapping had to be included in the tool. The generated code is created in different layers as shown in Figure 4.4.

In the lower layer, there is a kernel with the core functions of the IEC 61850 standard. On top of that, there are two libraries (i.e., `lib-service-server-rest` and `lib-service-server-soap`) for providing REST and SOAP communication capabilities. The former uses the `microhttpd` [137] library for the HTTP implementation, whilst the latter uses `gsoap++` [64]. In the uppermost layer are the servers, which use the functions provided by the libraries (CoAP and HTTP use the REST library while the SOAP uses the SOAP library). As HTTP and SOAP do not support publish-subscribe communication, the reporting services require new server and clients with reversed roles. The service server stores the reports on a folder and a reporting client polls that folder.

Function	URI	Method	Id	Notes
Basic				
Server				
GetServerDirectory	coap://{host}/LDs	GET	1	Logical Devices
GetServerDirectory	coap://{host}/Files	GET	2	Files
Logical Device				
GetLogicalDeviceDirectory	coap://{host}/LDs/{LD}	GET	3	
Logical Node				
GetLogicalNodeDirectory	coap://{host}/LDs/{LDs}/{LN}/{ACSIClass}	GET	4 5 6 7	Datas Datasets BRCBs URCBs
GetAllDataValues	coap://{host}/LDs/{LDs}/{LN}/AllValues[?FC={fc}]	GET	8 9 10	Big payload Medium payload Small payload
Data				
GetDataValues	coap://{host}/LDs/{LDs}/{LN}/Datas/{Data}	GET	11	
SetDataValues	coap://{host}/LDs/{LDs}/{LN}/Datas/{Data}	PUT	12	
GetDataDirectory	coap://{host}/LDs/{LDs}/{LN}/Datas/{Data}/Directory	GET	13	
GetDataDefinition	coap://{host}/LDs/{LDs}/{LN}/Datas/{Data}/Definition	GET	14	
Dataset				
GetDataSetValues	coap://{host}/LDs/{LDs}/{LN}/Datasets/{Dataset}	GET	15	
GetDataSetDirectory	coap://{host}/LDs/{LDs}/{LN}/Datasets/{Dataset}/Directory	GET	16	
Reporting				
BRCB				
Report	coap://{host}/LDs/{LDs}/{LN}/{BRCBs}/BRCB/Reports	Not.	17 18	Big payload Small payload
GetBRCBValues	coap://{host}/LDs/{LDs}/{LN}/{BRCBs}/BRCB	GET	19	
SetBRCBValues	coap://{host}/LDs/{LDs}/{LN}/{BRCBs}/BRCB	PUT	20	
if enable	coap://{host}/LDs/{LDs}/{LN}/{BRCBs}/BRCB/Reports	GET+Obs	21	
URCB				
Report	coap://{host}/LDs/{LDs}/{LN}/{URCBs}/URCB/Report	Not.	22 23	Big payload Small payload
GetURCBValues	coap://{host}/LDs/{LDs}/{LN}/{URCBs}/URCB	GET	24	
SetURCBValues	coap://{host}/LDs/{LDs}/{LN}/{URCBs}/URCB	PUT	25	
if enable	coap://{host}/LDs/{LDs}/{LN}/{URCBs}/URCB/Report	GET+Obs	26	
Additional Services				
Access Control				
Associate	coap://{host}/Associations	POST	27	
Abort	coap://{host}/Associations/{AssID}?action=abort	DELETE	28	
Release	coap://{host}/Associations/{AssID}?action=release	DELETE	29	
File Transfer				
GetFile	coap://{host}/Files/{File}	GET	30 31	Big file Small file
SetFile (update existing)	coap://{host}/Files/{File}	PUT	32	
SetFile (new file)	coap://{host}/Files	POST	33	
DeleteFile	coap://{host}/Files/{File}	DELETE	34	
GetFileAttributeValules	coap://{host}/Files/{File}/Status	GET	35	

Table 4.9: The set of the implemented IEC 61850 services mapped to the CoAP communication protocol.

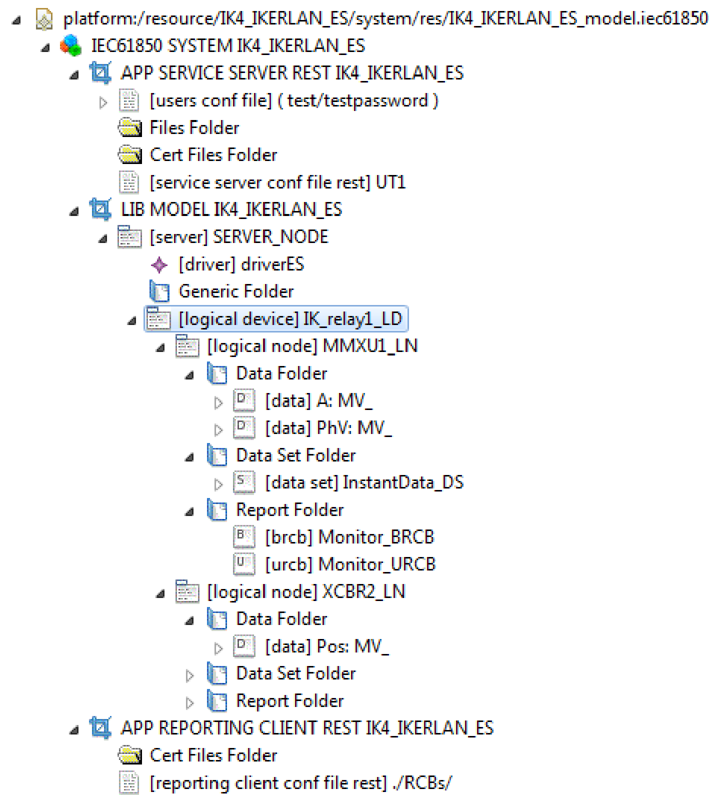


Figure 4.3: Screenshot of the tool that generates the IEC 61850 code automatically.

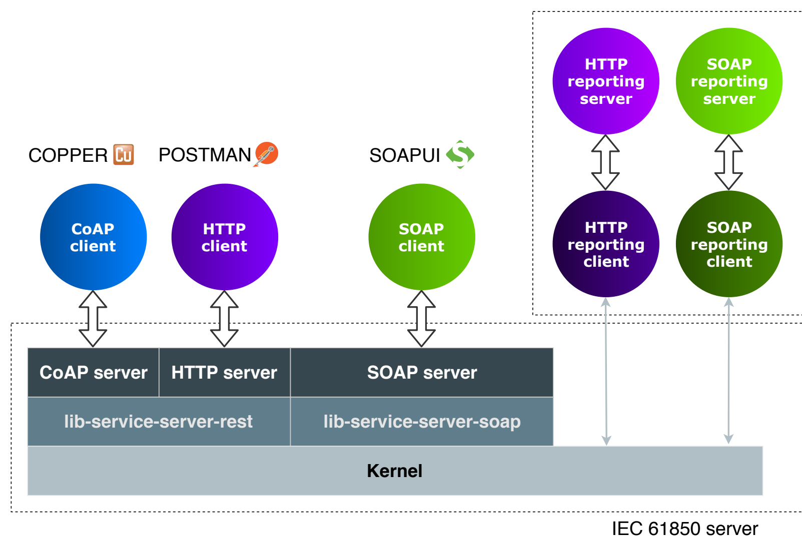


Figure 4.4: IEC 61850 tool layers.

For this contribution, CoAP communication capabilities have been added to the tool. It allows to build the entire CoAP resource tree using the model, same as HTTP and SOAP. A big difference with CoAP and the rest, is that CoAP does not need the complexity of creating new client and servers for the reporting services thanks to its Observe extension. As explained in the previous chapter, this extension allows to send notifications to subscribed clients. In the case of the reporting services, the client can subscribe to receive the reports. For that, the server checks whether new reports have been generated, and if so, it sends them to the subscribed clients using push notifications. As discussed in Chapter 3, there are many CoAP implementations for different environment. For this work, the develop branch version of Feb. 5, 2018 of libcoap [131] has been selected.

Regarding the data representation format, the HTTP implementation uses JSON, while SOAP uses XML. As a first step, the CoAP

implementation also uses JSON. Later in this chapter, this changes and the complete IoT stack is used, changing JSON for CBOR on the CoAP implementation.

4.6 Evaluation

After explaining the implementation, in this section the mapping is evaluated. The first step for evaluating the implementation is to generate the code of a concrete model. In this case, a model for a Smart Elevator has been used (see Figure 4.5). The only LD is represented in purple, the LNs in green and finally, the different CBs in orange. The individual CBs have been left out due to space constraints.

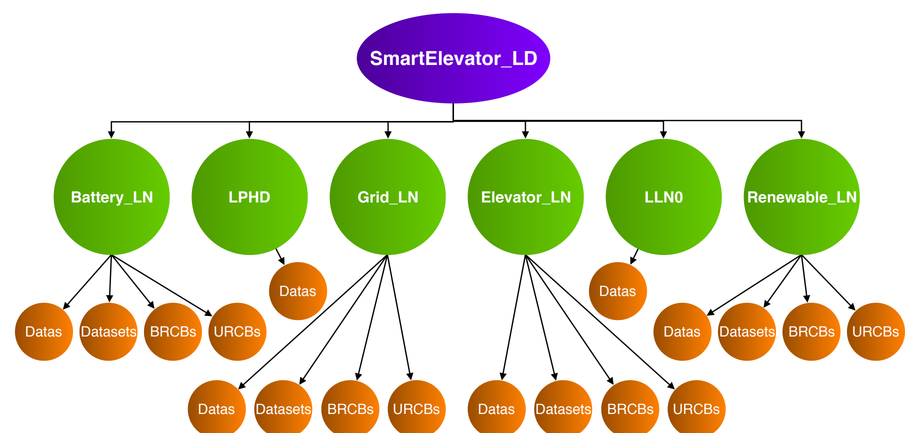


Figure 4.5: Model used in the experimentation.

For the evaluation, the generated code has been compiled and deployed on a Raspberry Pi 3 [168] board, a platform that is used more and more in industry scenarios, specially for prototyping. It has a Quad Core ARM Cortex A53 CPU that can run at 1.2 GHz and 1 GB of RAM at 900 MHz. Raspbian Stretch has been the selected OS, stored on a

16 GB class 10 microSD card. Even though the setup for a big scale deployment would not necessarily use the Raspberry Pi platform, the measured metrics would be proportional on the comparison of different protocols, making this setup appropriate for a laboratory evaluation. Even more, the Raspberry Pi platform is currently expanding to real-world industrial deployments as well[1].

Regarding the clients for the experiments, a Windows 7 PC has been used. Three different clients have been used: Postman 6.0.10[2] for HTTP, SoapUI 5.4.0[3] for SOAP, and Copper 1.0.1 [117] over Firefox 49.0.2 for CoAP. For the reporting functions in the case of HTTP and SOAP, the executables generated with the tool have been used. The network usage has been captured using the Wireshark[4] network packet analyzer on the client PC.

The usage of wireless technologies (including cellular technologies) is spreading into industrial deployments [107], especially in remote locations (e.g., electrical substations in remote environments). Using wireless technologies, installation costs are lower, data rates sufficient, they have good reliability, and they are easy to deploy. Although most electrical substations are deployed using Ethernet or power line networks [60], deploying wired network infrastructure increases the deployment time and economic cost. Overall, IEEE 802.11 protocols are suitable for ensuring high security and QoS for less critical smart distribution network applications [156]. Hence, a transition to Wireless-based substations or mixed approaches is expected, where in-

[1] https://www.rs-online.com/designspark/raspberry-pi-and-arduino-in-industrial-environments

[2] https://www.getpostman.com/

[3] https://www.soapui.org/

[4] https://www.wireshark.org/

teroperability is ensured through standards such as IEC 61850. Taking all this into account, the server and the client are connected though a WiFi network in this experiment.

The functions that have been tested are presented in Table 4.9. As it can be seen in the table, some of the functions have been tested more than once using different parameters, with response or requests that include different sizes of payloads. For the first evaluation, three aspects have been taken into account: the suitability of the proposal, the latency improvements and the needed bytes for the message exchange.

Firstly, with this experiment, it is demonstrated that the proposed mapping is suitable. The notification of reports is simpler and natural with CoAP, because the Observe extension can be used to communicate through the publish-subscribe model. The reporting system in HTTP and SOAP is more complex, due to needing to add extra components and is not able to use the publish-subscribe model for the communication, which is much more suitable for this use case.

4.6.1 **Latency**

Secondly, the latency has been measured for the three protocols. Table 4.10 presents the maximum and the median latencies for the analyzed functions. Report subscription (functions id 21 and 26) and notifications (functions id 17, 18, 22 and 23) have not been included in the latency analysis. SOAP and HTTP implementations do not include these functions, so they can not be compared.

The results indicate that CoAP's median value is the lowest in general, with values between 6 and 8 ms for most of the functions. Some peaks are shown when the block-wise transfer is required. This is specially obvious in functions 8 and 9, where the median latency

Id	Maximum latency (ms)			Median latency (ms)		
	CoAP	**HTTP**	**SOAP**	**CoAP**	**HTTP**	**SOAP**
1	22.2	32.3	12.7	5.9	5.4	5.1
2	93.7	69.7	38.2	6.2	7.6	5.8
3	67.4	31.7	212.4	7.0	7.7	8.0
4	38.3	76.8	20.7	12.0	8.2	6.9
5	26.8	226.6	65.4	6.5	5.8	9.1
6	44.9	36.1	103.2	6.8	7.1	8.8
7	67.3	82.0	97.7	6.4	5.2	7.0
8	8354.2	484.9	297.4	**4003.6**	64.2	75.5
9	5291.4	312.5	304.1	**3460.1**	256.2	73.5
10	231.8	27.7	25.0	14.4	7.2	8.8
11	25.3	155.5	42.3	**6.5**	10.0	11.6
12	105.9	299.5	77.6	**6.2**	209.6	8.7
13	57.6	93.0	28.3	7.4	10.5	7.7
14	597.6	102.9	31.6	**390.6**	13.1	13.7
15	31.0	46.7	220.8	6.4	6.3	9.1
16	73.7	94.6	23.8	10.5	7.4	6.8
19	29.7	42.4	25.2	9.6	8.5	8.6
20	21.0	677.7	76.4	**6.2**	209.3	9.5
24	32.7	245.0	24.3	8.2	209.5	9.9
25	22.1	103.4	32.9	**5.9**	7.7	7.8
27	83.0	103.1	65.0	12.2	8.3	5.4
28	37.7	97.8	23.5	8.8	6.2	5.1
29	30.7	14.5	23.5	10.7	5.5	5.1
30	1196.4	272.9	83.9	**956.4**	219.3	19.7
31	60.5	26.5	308.3	6.4	6.5	9.2
32	62.9	262.8	66.8	7.2	209.9	8.9
33	58.4	227.4	32.4	7.7	207.6	8.6
34	27.3	157.3	32.8	6.1	8.8	8.4
35	38.4	40.6	17.6	6.4	9.1	7.1

Table 4.10: Maximum and median latencies in ms with CoAP, HTTP and SOAP.

jumps to 4003.6 and 3260.1 ms respectively. As stated in Chapter 3, the block-wise transfer extension uses a “Stop & Wait” approach to send large payloads, compared to TCP’s sliding window protocol, where more than one segment can be sent without waiting for the previous ACKs. SOAP also performs well in some cases, specially when CoAP does not. HTTP, on the other hand, is in the middle for most functions, although it has some negative peaks.

4.6.2 Overhead

Lastly, the bytes exchanged are presented in Figure 4.6, including the payload bytes and the overheads. TCP requires an open channel in order to exchange messages and for this analysis, it is assumed that the channel is already open. This means that TCP handshake, keep-alive and session closing messages are not counted. Reporting subscription messages have not been included because, same as before, they are not part of the IEC 61850 specification. The SOAP implementation of the tool includes a Logoff function instead of Release and Abort, so the measurements are the same in the chart for both functions.

In the chart in Figure 4.6, it is clearly shown that the XML representation the SOAP implementation uses is larger than the JSON representation used by HTTP and CoAP. At the same time, SOAP is very verbose, which adds even more overhead than the rest. Both HTTP and CoAP use the same JSON representation of the resources, hence, the data parts of the message are exactly the same. The overhead is much smaller with CoAP in most cases, but when the block-wise transfer is needed, the difference decreases. When many blocks are required, CoAP has more overhead than HTTP (functions 8 and 9). The reason for this is that with CoAP, when the block-wise transfer is used, each

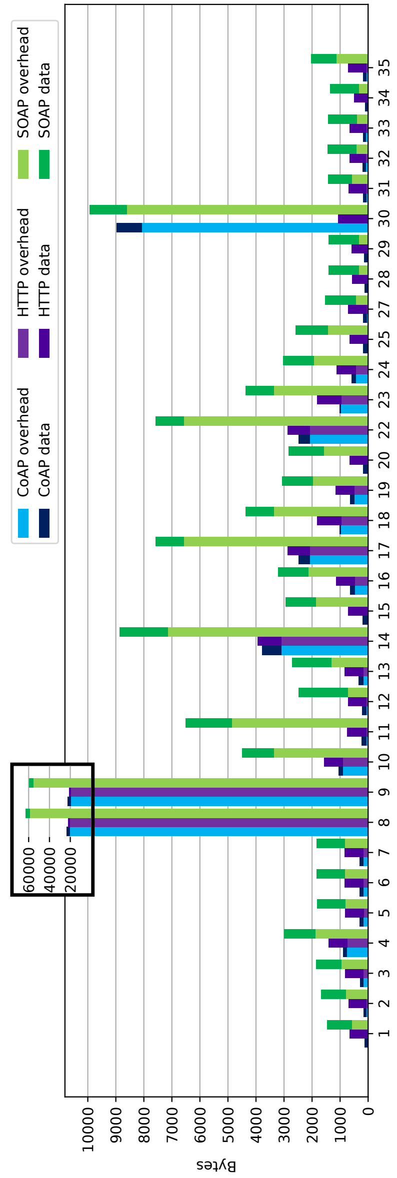


Figure 4.6: Data and overhead bytes.

message containing a part of the payload includes the entire header, while HTTP divides the entire message, header included.

4.7 Lightweight Resource Representation

After the first implementation and its evaluation, a lightweight resource representation has been added to the CoAP implementation, in order to have a complete IoT stack. This way, an even more lightweight communication is achieved, which makes the communication more suitable for the newly used wireless low bandwidth networks that are being deployed in these scenarios, as stated in Section 4.1.

The selected lightweight resource representation format is the Concise Binary Object Representation (CBOR) [19], a lightweight representation format proposed by the IETF for IoT application in the RFC 7049. It is a binary serialization data format that aims to encode data with small code footprint, small message size and extensibility with no version negotiation, making it more lightweight than other resource representations, such as JSON or XML. These features make CBOR a good option for IoT domains, completing a lightweight stack along with UDP, CoAP and CBOR [97], enabling an even more lightweight communication.

In [163], the authors compare several data serialization formats in Smart Grid environments and show that CBOR needs fewer bytes than JSON and XML. The IETF also uses CBOR to represent its Link Format [128], which is used in CoAP. Adding CBOR to the implementation of the IEC 61850 mapping reduces the total number of transmitted bytes.

In order to include CBOR in the CoAP implementation, the `lib-service-server-rest` presented in Figure 4.4 had to be forked for the CoAP implementation. Figure 4.7 shows the new library. In the

second layer, `lib-service-server-rest` has been substituted by `lib-service-server-http` and `lib-service-server-coap`. Also, libraries for the resource representations are shown. The SOAP implementation uses gsoap++ [64], the HTTP uses JSON with jsoncpp [104], and finally, the libcbor [130] has been included to use CBOR in CoAP.

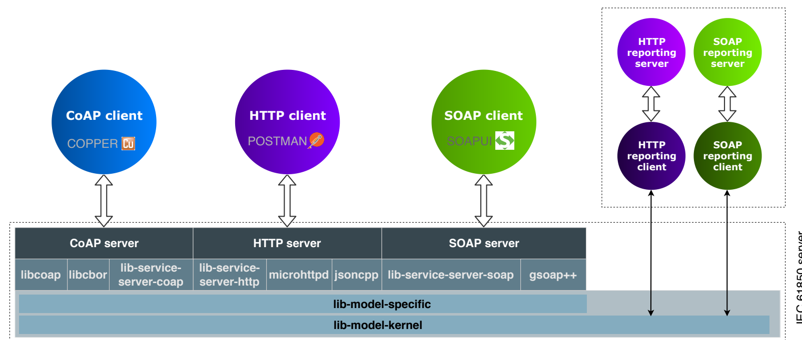


Figure 4.7: Layered overview of the reference design of the IEC 61850 software tool.

4.8 Validation

Having added CBOR, a new comparison has been carried out to compare the performance and overhead of using the entire IoT stack (CBOR on top of CoAP on top of UDP). The experimental scenario is the same as the one from the previous section. This comparison aims to validate the mapping and its implementation using CBOR.

4.8.1 Latency

Same as the first evaluation, the first measured metric is the latency, the time from a request is sent to the response arrives. To analyze he latency,

500 requests have been made for each function and the maximum and median values have been calculated. Table 4.11 compares the CoAP metrics using the previous implementation with JSON and the new one with CBOR. For this experiment, CON messages have been used in CoAP request. However, the performance results with NON messages would be similar, as the responses are piggybacked on ACKs in CON messages.

In Table 4.11, the maximum latency values are showed in the left columns and the median values are in the right. The maximum time represent the worst case scenario, while the median represents a value that can be expected, which is more important. The median typically shows better values than the average, as it mitigates outliers.

Analyzing the median values, it can be seen that using CBOR lowers the latencies in all the functions. The improvements are bigger with large payloads, because the payload is smaller using CBOR and if the block-wise transfer is required, less blocks are needed, which is a huge improvement. Most of the values in the CBOR column range between 5 and 8 ms. The average response time of all the tested functions, CoAP's response time is 71% of HTTP's and 85% of SOAP's when using CBOR if the functions using the block-wise transfer are not counted (functions 8, 9, 14 and 30). These functions are excluded from this calculation because the block-wise transfer in CoAP works following a Stop & Wait model, which increases the response times by a large amount.

Figure 4.8 plots the ratio of the median response time values. CoAP is used as the baseline (at 100%) and HTTP and SOAP are represented in relation to that baseline ($percentage = \frac{CoAP\ median}{Protocol\ median} \times 100$). Values

Id	Max. latency (ms)		Median latency (ms)	
	CBOR	JSON	CBOR	JSON
1	21.8	22.2	5.7	5.9
2	20.8	93.7	5.8	6.2
3	37.3	67.4	5.9	7.0
4	36.6	38.3	6.6	12.0
5	11.9	26.8	6.2	6.5
6	29.9	44.9	6.1	6.8
7	62.6	67.3	6.0	6.4
8	5119.1	8354.2	**3127.9**	4003.6
9	6495.6	5291.4	**2722.7**	3460.1
10	41.6	231.8	7.4	14.4
11	10.0	25.3	5.9	6.5
12	18.6	105.9	6.1	6.2
13	14.5	57.6	6.0	7.4
14	325.4	597.6	**232.5**	390.6
15	92.0	31.0	5.9	6.4
16	16.7	73.7	6.4	10.5
19	37.7	29.7	6.2	9.6
20	81.3	21.0	5.5	6.2
24	7.6	32.7	6.2	8.2
25	14.0	22.1	5.4	5.9
27	19.6	83.0	9.7	12.2
28	67.7	37.7	8.0	8.8
29	61.7	30.7	7.1	10.7
30	1018.6	1196.4	**682.3**	956.4
31	14.8	60.5	6.0	6.4
32	76.5	62.9	6.0	7.2
33	15.4	58.4	6.1	7.7
34	18.1	27.3	5.6	6.1
35	13.4	38.4	5.8	6.4

Table 4.11: Maximum and median of the response times in ms using JSON and CBOR resource representation in the CoAP implementation.

below 100% mean that CoAP is faster than the compared protocol, which is the case for most functions.

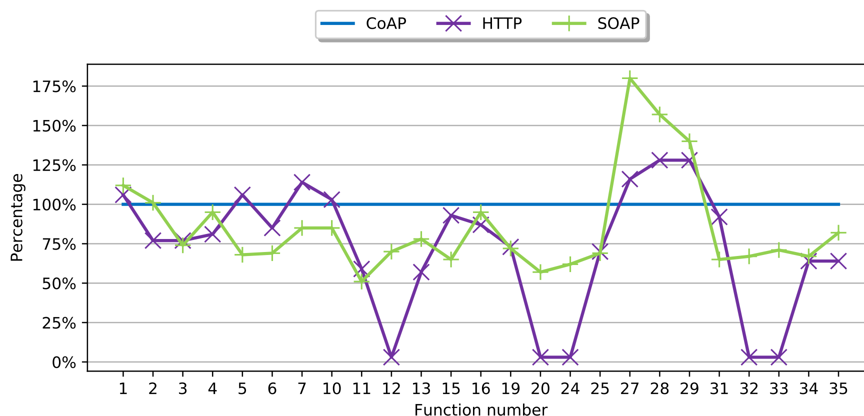


Figure 4.8: Percentage of CoAP's median response time compared to HTTP and SOAP in the functions without block-wise transfer. Values below 100% mean that the protocol is slower than CoAP.

As downside for CoAP, there are some negative peaks, with median values up to 2 or 3 seconds (functions 8 and 9). As previously explained, the block-wise transfer extension is the responsable for this because of its Stop&Wait pattern. TCP's sliding window approach used in HTTP and SOAP makes these last to outperform CoAP in such cases.

4.8.2 **Overhead**

The last analyzed metric for the evaluation is the overhead of the messages. Figure 4.9 compares the payloads and overheads of the different functions and protocols. Bars represent the number of bytes for each protocol and resource representation (payload in CBOR, JSON or XML for CoAP, HTTP and SOAP respectively) that are needed to send data

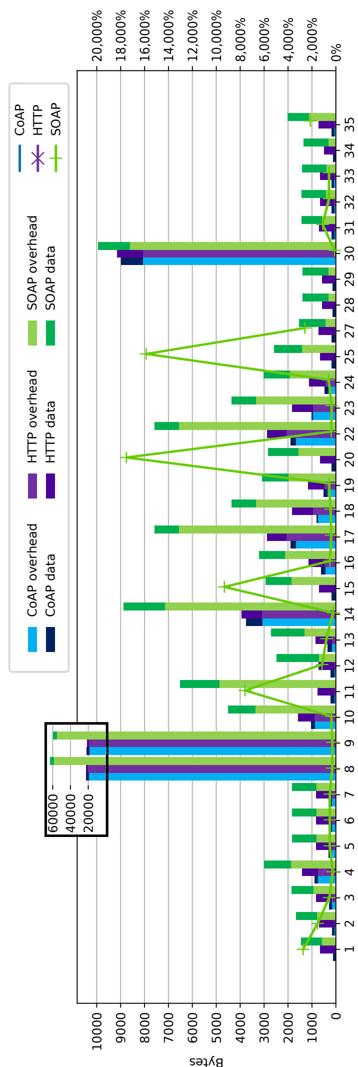


Figure 4.9: The bars show the number of bytes required for the resource representation and overhead in CoAP, HTTP and WS-SOAP implementations. The lines represent the overheads of HTTP and SOAP as fractions of the CoAP baseline.

from the server to the client. For calculating the overhead, the bytes added by the different protocols are counted, i.e., the bytes on the wire minus the payloads. TCP's management messages (i.e., channel open and close handshakes and keep-alive messages) are not counted. Same as the first evaluation, the functions for subscribing to the reports (functions 21 and 26) are not included due to not being part of the standard and HTTP and SOAP do not include them. In the SOAP implementation, a single function called Logoff substitutes Release and Abort functions, thus, the data is the same for both.

In this comparison the differences between data representation formats is clear, especially in the case of XML (used in SOAP), which is the largest by far. The lines on Figure 4.9 represent the relation of the resource representation compared to CoAP. Figure 4.10 zooms in to show the HTTP (JSON) and CoAP (CBOR) relationship better, as in Figure 4.9 these values are in the lower part of the chart. The XML representation goes as high as 17500% in respect of CBOR, while JSON is just slightly worse than CBOR. The average difference of all the functions is that CBOR requires the 89% of bytes compared to JSON and 19% compared to XML.

Regarding the overhead, SOAP has the largest one because it is a very verbose protocol. HTTP has much lower overhead than SOAP, but CoAP has a even lower overhead for most functions. However, when big payloads are sent using the block-wise transfer, CoAP's overhead increases to be more than the HTTP's. This happens because when chunking the payload, the block-wise transfer extension adds the entire header to each message, while HTTP chunks the entire message, header included. In Figure 4.11 a comparison of all the sent bytes is presented, payload plus overhead, using CoAP's implementation as the

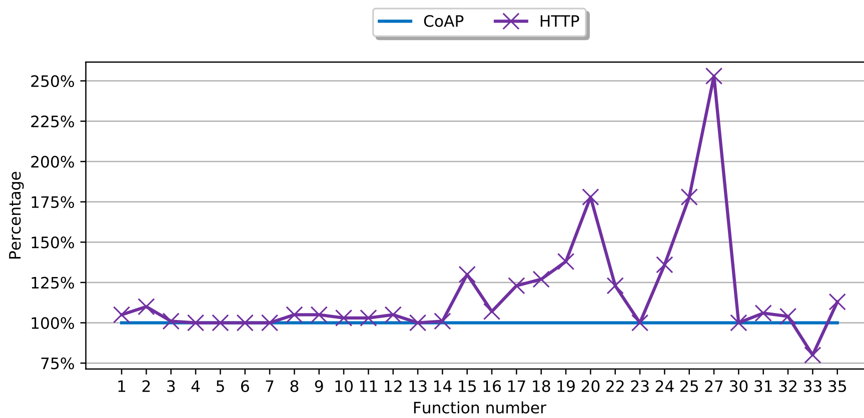


Figure 4.10: Percentage of required bytes for the resource representation of HTTP's (JSON) implementation with CoAP's (CBOR) as baseline.

baseline. CoAP+CBOR uses on average 44% of the bytes comparing to HTTP+JSON and 18% comparing to WS-SOAP+XML.

4.9 Conclusion

Having presented a mapping of IEC 61850 to CoAP, implemented it and compared its performance against HTTP and SOAP, some conclusions can be provided.

First, a literature review has been carried out, surveying different existing mappings of the IEC 61850 standard to various communication protocols. There are several approaches using different protocols, however, all of them except the one presented by Shin et al. [185] use heavy protocols designed for powerful resource devices in terms of RAM, CPU, network capabilities, etc. The work from Shin et al. does not include many functions and does not follow the RESTful paradigm of CoAP, so a new proposal, also using CoAP has been presented in

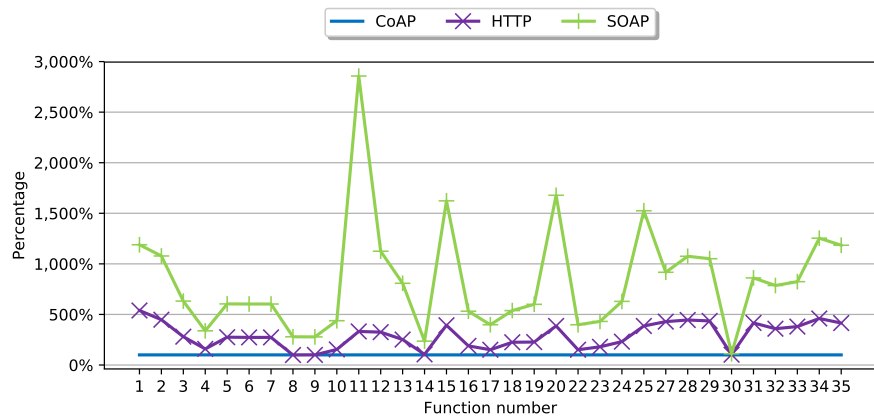


Figure 4.11: Percentage of total number of required bytes (payload + protocol overhead) for sending data in HTTP's and WS-SOAP's implementation with CoAP's as baseline.

this chapter, aiming for a deployment on resource constrained device and networks. In addition to mapping more functions than the previous works and following a RESTful approach with CoAP, the proposed mapping in this chapter has also been evaluated using real hardware instead of using a network simulator tool.

In order to do that, a new extensive mapping has been proposed, covering all ACSI services from the IEC 61850 standard and following the RESTful architecture. Although some of the ACSI services can be easily mapped to CoAP (e.g., basic services), others are more complex (e.g., reporting and eventing services). Using the client-server model for communication, the functions to retrieve, change, create and delete data are straightforward to be mapped to the GET, PUT, POST and DELETE verbs of CoAP. On the other hand, notification services can be mapped to CoAP with the help of the Observe extension.

After describing the proposed mapping, a part of the mapping has been implemented. The implemented functions are the ones that a previous tool at Ikerlan supports with HTTP and SOAP mappings. With the implementation, it is demonstrated that it is possible to integrate the IEC 61850 with the IoT, though CoAP and CBOR. Thanks to the Observe extension, all the ACSI services can be implemented using CoAP in a natural way.

Thanks to the experiments carried out, it is clear that the CBOR representation requires fewer bytes than JSON and XML, and that CoAP adds less overhead than HTTP and WS-SOAP. This results on a lower response time for the CoAP implementation, at least when the block-wise transfer is not required. The block-wise transfer penalizes a lot, especially when many blocks are needed. Reducing the overhead is of vital importance for some Smart Grid systems, where devices can be in very remote locations with very limited connectivity. This connectivity issues can be as extreme as having a 1500bytes/s bandwidth in some parts of the network, making big messages to timeout and retransmissions start, amplifying the problem.

Although the Observe extension adds a natural way of mapping the publish-subscribe functions, it still has some limitations. When setting a BRCB or a URCB, the client can not directly subscribe and needs an extra extended GET request, which includes the Observe option. A contribution expanding the capabilities of the Observe extension to support advanced subscription mechanisms is presented in the next chapter.

Copy from one, it's plagiarism; copy from two, it's research.

Wilson Mizner

We are like dwarfs [the moderns] sitting on the shoulders of giants [the ancients]. Our glance can thus take in more things and reach farther than theirs. It is not because our sight is sharper nor our height greater than theirs; it is that we are carried and elevated by the high stature of the giants.

Bernard of Chartres

5

Advanced Subscription Mechanisms

Chapter 4 points out some limitations on the Observe extension of CoAP. In this chapter, an enhancement for the Observe extension is proposed, aiming to overcome those limitations. In order to do that, some new CoAP option and response codes are proposed, and the integrated in the IEC 61850 mapping presented in Chapter 4.

Contents

5.1 Introduction 162
5.2 Related Work 164
5.2.1 IETF RFCs 164
5.2.2 IETF working drafts 165
5.2.3 Other proposals 166

5.3 Enhancement Proposal 170
5.3.1 Subscribe through PUT/POST requests 172
5.3.2 Lightweight responses 173
5.3.3 Subscribe through third resources 175
5.3.4 New CoAP options and response codes 178
5.4 Evaluation: exploratory example 180
5.5 Implementation in the IEC 61850 Mapping 183
5.6 Validation 184
5.7 Conclusion 186

5.1 Introduction

With the advent of the IoT, a wide variety of devices are being connected to the Internet. These devices can connect and disconnect to different networks for different reasons, such as going to sleep due to battery constraints or being mobile devices that physically move out of the network's reach.

One of the most important protocols for this type of devices is CoAP, along with its several extensions, which has been compared to MQTT (see Chapter 3) and then used in an Industrial domain (see Chapter 4). By default, CoAP uses a client-server communication model, but with its Observe extension [68], it can also communicate using a publish-subscribe paradigm. The Observe extension defines an option to be included in GET requests, indicating that the client wants to receive notifications when the resource state changes. The server is in charge of determining what a change constitutes in a resource state, e.g., value change, value change out of a defined bound, timeout, etc. The notifications are CoAP response messages. When used in a request, the Observe option is used to register or deregister to a resource on

the server, and when it is used in a response, it indicates a sequence number for reordering notifications. Figure 5.1 presents the exchanged messages for a successful subscription process and the delivery of two notifications.

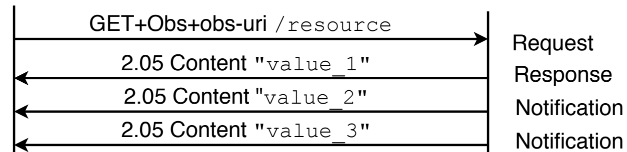


Figure 5.1: Subscribing to a resource and getting notifications using CoAP and the Observe extension.

However, as concluded in Chapter 4, the Observe extension has some limitations. More specifically, the subscription process is very simple and it does not allow many alternatives to a basic subscription process. In order to enhance the IEC 61850 mapping to CoAP presented in Chapter 4, the subscription process needs to be improved, enhancing the Observe extension. More concretely, some new ways to request a subscription, in order to be able to subscribe when editing or creating a resource, get lightweight responses to a subscription request, and subscribe using proxy resources.

This contribution first analyzes the existing approaches to improve the Observe extension of CoAP to later propose new CoAP option and response codes. Then, an exploratory example is presented along with a theoretical analysis. The next part of the contribution consists on the integration of the new proposed options with the mapping presented in Chapter 4, with a comparison of some of the mapped functions with and without the new options. Finally, some conclusions are presented.

5.2 Related Work

The CoRE [80] group has defined several standards and some working drafts for resource constrained devices and networks. One of such standards is the *Observing Resources in the Constrained Application Protocol* (RFC 7641), which enables CoAP devices to support the publish-subscribe communication model. RFC 7641 can be divided in two parts, one for the subscription mechanisms and another one for the notification generation and transmission management. Since CoAP was published, research has been carried out to enhance CoAP notifications, but the enhancements are mostly limited to the management of the notification, not the subscription process. In the following, the related work is surveyed, analyzing already finished standards, working drafts and proposals from outside the IETF.

5.2.1 IETF RFCs

Several RFCs have been proposed by the IETF to improve CoAP's capabilities, including some that have already been mentioned in previous chapters. This includes *Observing Resources in the Constrained Application Protocol (CoAP)* [68], *Constrained RESTful Environments (CoRE) Link Format* (RFC 6690) [181], *Group Communication for the Constrained Application Protocol (CoAP)* (RFC 7390) [166], *Block-Wise Transfers in the Constrained Application Protocol (CoAP)* (RFC 7959) [20], and *PATCH and FETCH Methods for the Constrained Application Protocol (CoAP)* (RFC 8132) [195]. The first one is the only one that is related to the publish-subscribe communication model in CoAP. However, it is the basic observe, which is what this contribution extends.

5.2.2 IETF working drafts

Apart from the already finished specifications, the IETF is also working on several drafts that are being actively discussed within the community. For this contribution, the most relevant ones are *Dynamic Resource Linking for Constrained RESTful Environments* (Dynlink) [183] and *Publish-Subscribe Broker for the Constrained Application Protocol (CoAP)* (CoAP-PubSub) [114], since they expand the features of the Observe extension of CoAP, focusing on the notification management.

Dynlink defines some parameters to configure attributes for a conditional observation of the resources. In addition to the notifications, the defined parameters also are used in push or poll requests. The attributes are stored in the server on a table named *binding table* and when an event triggers, the server must check the binding table to decide if a notification needs to be sent to the clients. The parameters defined in Dynlink are *pmin* – minimum time between notifications, *pmax* – maximum time between notifications, *st* – step in the value; *gt* – upper limit value, *lt* – lower limit value, and *band* – a bounded or unbounded value range.

CoAP-PubSub broker proposes to use an intermediate device as a broker, mimicking other publish-subscribe communication protocols. With a broker, the client and servers can be decoupled, which can make the system to support limited availability or sleepy nodes. Publishing clients can push data to the broker using PUT or POST requests and the subscribed clients receive the notifications or poll the broker to get the information. In this working draft, different functions are described to interact with the broker: *Discovery*, *Create*, *Publish*, *Subscribe*, *Unsubscribe*, *Read*, and *Remove*.

The IETF is also working on other drafts that are not directly related to the Observe extension but can be used in combination with it, such as *CoAP Simple Congestion Control/Advanced (CoCoA)* [17], *Representing Constrained RESTful Environments (CoRE) Link Format in JSON and CBOR* [128] and *CoRE Resource Directory* [182] among others. However, none of the finished RFCs nor the active working drafts propose more advanced subscription mechanisms.

5.2.3 **Other proposals**

Outside of the IETF proposals, there are other works that propose ways to enhance the observation of CoAP resources. These works focus on optimizing the notification generation and delivery processes, optimizing the number of notifications using different approaches, e.g., semantic meaning for the resources, using conditions through queries or options, or using intermediary devices.

Ketema et al. [111] propose *conditional observation*, where a client specifies the notification criteria in conjunction with the Observe option. Instead of using queries like [184], the authors propose to use new CoAP options, drafted in [129]. The mentioned options are *Minimum-Interval* and *Maximum-Interval*. The options use 5 bits for the option value, hence 32 different observation types can be coded, including *time series*, *maximum response time*, *minimum response time*, *step*, *All Values Less/Greater/Equal*, and *Periodic*. In their paper, the authors also demonstrate that the options perform well, however, the draft has expired.

Going further the conditional observe, Mietz et al. [138] propose to use a new option named *High-Level State Option*. This option gives semantic meaning to values exposed by a resource. To do that, a POST

request is sent to the resource including this option and the server creates a subresource with an arbitrary URI. This URI can be queried by a client. In this paper, the authors give the example of a temperature resource. The subresource can be the meaning of the temperature, i.e., cold, warm or hot. This approach has three main advantages: it gives semantic meaning to raw values of the sensors, it needs fewer notifications, and it simplifies the management of the subscribed clients compared to the conditional observe, as it does not need to manage them one by one.

Tanganelli et al. [199] propose a different mechanism for minimizing the number of notifications using proxies. These proxies support and optimize notification periods, similar to brokers. In this proposal, the proxies subscribe to the servers and the subscribing clients specify the conditions in which they want to subscribe. This way, the proxy is the only node that subscribes to the server and uses a new proposed algorithm to calculate the optimal period for the proxy to receive the notifications from the server and comply with the notification receiving criteria specified by the clients. Using this approach, resource constrained clients that are not able to manage all the generated notifications can subscribe to resources that otherwise could not.

Ludovici et al. [133] also focus on reducing the number of notifications, using QoS parameters to support timeliness. In order to do that, the authors propose four priority levels used to determine the delivery order of the notifications. Following this approach, the client can specify which notification can be dropped if the server can not manage all of them. An observing client demands a priority level when subscribing and the server can accept the level or negotiate a lower one. Using this approach, observers with different roles can define

different requirements, leading to a reduction on energy consumption, and delivery delay and ratio improvements.

In [174], Sacramento et al. propose a new framework for planning registration states, and notification aggregation and scheduling using proxies to maximize energy saving, bandwidth efficiency and delay reductions. When there are multiple notification requests in a network optimization, issues are arisen, including needed registration steps, energy saving, consistency, and caching. CoAP and the Observe extension are able to use caches and proxies in order to improve the scalability and efficiency. Optimizing them can improve the performance of an entire system, and according to the authors, planning proxies to aggregate and schedule notification achieves huge improvements on the performance. In [41], the authors continue the work accounting for timeliness fairness. Correia et al. argue that considering the network's lifetime, i.e., the time while all nodes are up, the timeliness of messages should be evenly distributed to maximize it. Evenly distributed notifications between nodes make battery consumption on the nodes balanced, resulting on a longer network lifetime. With this approach, the fairness is taken into account on the registration steps, planning data forwarding through proxies in a way that registration steps impose equity among the nodes. Correia et al. continue the work in [40] adding a heuristic approach for developing an algorithm to make the decision on the registration steps.

According to Ishaq et al. [99], the Observe and group extensions of CoAP can not be gracefully combined and observing a group of resources is not straightforward. In order to solve this, the authors propose a new entity called *Entity Manager (EM)*, which manages the observation of groups. The EM pushes changes in a group after aggregating them to observing clients, instead of using individual no-

tifications for each client, leading to a reduced amount of message exchanges. The authors propose to use a new resource named "/e" to create entities (resource groups), which supports POST request. The created entity acts as proxy for the entire group and can be accessed like any other resource. It can be polled, push requests can be sent to it, and it can be observed.

In Choi et al. [31], the authors propose to cluster the nodes to aggregate the data. A node acts as the head of the cluster, and aggregates and transmits the data behaving as a proxy. This way, the performance in terms of bandwidth consumption and transmission time is improved and it allows many-to-many message exchanges in CoAP. Topics are used to represent the sensor attributes and the cluster head is the intermediary to obtain the sensor information. The client nodes register to the cluster head using the topic, and the cluster then receives the messages and redirects them to the sensors. In order to be able to do all this, the authors define three different options: *NORMAL* to send a normal condition of the server, *DIRECT* for messages that need to be sent in immediately, and *LIMIT* point out that the message has a deadline.

The last analyzed piece of research in this context is the one by Kome et al. [113]. In their work, the authors propose a new CoAP based protocol for IoT, CoAP 2.0. Among other features, they focus on the publish/subscribe communication paradigm, aiming to offer more advanced notifications. In these regard, Kome et al. propose to enable the choice of adding rules on a GET request with the observe option, with the objective of receiving the notifications that fulfill the included rule. This way, the observer only receives the notifications that are of its interest. In addition, this work also allows to be subscribed to

the same resource with different rules, using a database to manage the active subscriptions.

Table 5.1 summarizes the analyzed related work presented in the previous paragraphs.

Work	Approach
[183]	New CoAP options for conditional observation
[114]	Broker for decoupling devices
[111]	New CoAP options for conditional observation
[138]	New CoAP options for semantic meaning of values
[199]	Proxies for optimizing notification periods
[133]	Priority levels to drop notifications
[174]	Proxies for aggregating and scheduling notifications
[41]	Timeliness fairness to the previous work
[40]	Heuristic algorithm for registration steps of the previous work
[99]	EM for managing observation groups
[31]	Cluster nodes, the head of the cluster aggregates and notifies subscribers
[113]	CoAP 2.0, including notification rules

Table 5.1: Summary of the related work on CoAP's publish/subscribe.

However, all the analyzed proposals focus on the generation and the delivery of notifications, none of them tackles the subscription mechanism. As presented in the introduction, the contribution from Chapter 4 have some requirements that can improve the subscription process.

5.3 Enhancement Proposal

After reviewing the works to improve CoAP, in this section a proposal to improve the subscription mechanisms of the Observe extension is presented. The goal for this contribution is to reduce the overhead and

the latency when subscribing to a CoAP resource. In order to make a proposal, the first step is to clearly define the requirements.

- Subscribe though PUT and POST requests. In the current state of the RFC 7641, a client is able to subscribe to a resource only using extended GET requests, it is not possible to update or create a resource and subscribe to that same resource in a single step. Expanding the use of the observe option from GET request to also other CoAP verbs enables to create or update a resource and subscribe to its notifications at once.

- Lightweight responses. A client might want to subscribe to get notifications from a resource, but also not be interested in the current state of the said resource (especially on low bandwidth data connections). Sometimes, a "Subscribed" message might be enough for the client, and get the information when something happens with the resource. Even more, the resource may not have valid data values or may even not exist yet.

- Subscribe through third resources. Some resources may be related to other resources. A client may want to update, create or read such resources while subscribing to the related one. A clear example of this use case is a configuration resource that manages the notification generation of another resource. Reading, updating or creating the configuration resource may be of interest to a client, and also subscribe to the notifications of the main resource without sending a request to it. With the current state of the RFC 7641, two requests are needed, one for requesting the configuration resource and a second one to request to subscribe.

In order to fulfil these requirements, new CoAP options and new CoAP response codes are proposed. These options and codes are for subscribing requests and responses, they do not affect in any way the notifications or their generation. Hence, dynamic environments or networks would especially benefit from using them, as these environments are more likely to have connecting and disconnecting devices that subscribe and unsubscribe more often.

5.3.1 Subscribe through PUT/POST requests

With the current state of the Observe extension, a client only can subscribe to a resource using an extended GET request. PUT and POST requests can not include the Observe option, so they can not be used to subscribe to a resource. Hence, the first proposal is to enable the Observe option to be used with PUT and POST requests, mimicking GET requests. This way, the interaction to subscribe is similar with PUT and POST request, where the server responds with "2.01 Created" or a "2.04 Changed" codes along with the Observe option with the identifier for the notification number. With this message exchange, a client can create or update a resource while subscribing to its notifications in a single step. After the first response, the notifications will be sent exactly like the notifications described in RFC 7641. An example of a resource update and subscription, the response, and two notifications are presented in Figure 5.2.

If the subscription to a resource fails, the response will not include the Observe option, just the same as the regular GET subscription described in RFC 7641.

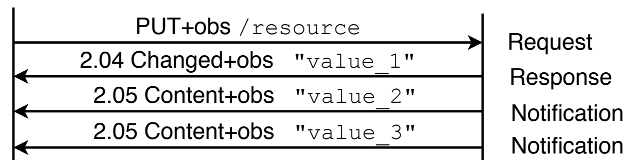


Figure 5.2: Update the values of a resource, subscribe to it and get notifications.

5.3.2 **Lightweight responses**

In order to be able to subscribe to a resource and get a "Subscribed" type of response message without the resource representation, a new CoAP option is proposed, namely "no-payload". When including this option, the observe option must also be present in the request, and it is compatible with GET, PUT and POST requests. The "Subscribed" type of response messages require new response codes, and the proposed codes are "2.10 Subscribed", "2.11 Created and Subscribed" and "2.14 Changed and Subscribed". The last two response codes imply that the resource has been correctly created or changed, and all of them indicate a successful subscription, but the response does not include the resource representation. Response codes have been selected following a parallelism with previous response codes, from 2.0X codes to 2.1X codes:

- "Subscribed: 2.10"
- "Created and Subscribed: 2.11", parallel to 2.01 Created
- "Changed and Subscribed: 2.14", parallel to 2.04 Changed

An interaction using a GET request is shown in Figure 5.3. The client sends a GET request with the observe and no-payload options

activated, indicating that it wants to subscribe but it is not interested in the current state of the resource. The figure represents a successful interaction, with a “2.10 Subscribed” response and then the notifications when they are generated. If the GET request can not be satisfied for some reason, regular CoAP error codes are to be used. If what fails is the subscription step, “5.00 Internal Server Error” or a “5.03 Service Unavailable” are the responses, depending on whether a MAX-AGE option is included.

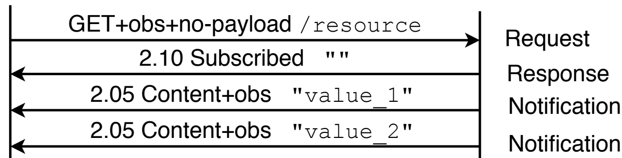


Figure 5.3: Subscription to a resource without getting the current state and get notifications.

If a client wants to create or update a resource using a POST or PUT request, the adequate response is “2.01 Created” or “2.04 Changed” depending on whether the resource previously existed or not. With this proposal, “observe” and “no-payload” option can be used with PUT and POST requests. Figure 5.4 presents an example using an extended PUT response with both options present, and a “2.14 Changed and Subscribed”, indicating that the resource has been successfully updated and the client has been subscribed, but the response does not include the resource representation. If the resource did not previously exist, the correct response would be “2.11 Created and Subscribed”. POST request work in the same way. If the resource is correctly created or updated but the subscription fails, the server responds with a “2.01

Created" or "2.04 Changed" codes. If there is any other error, regular CoAP codes are to be used.

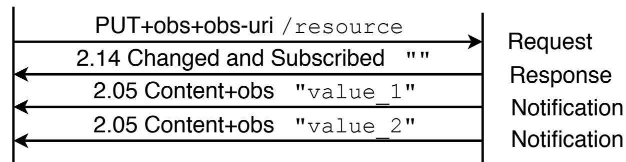


Figure 5.4: Change a resource, subscribe to it but do not receive a payload, then get notifications.

5.3.3 Subscribe through third resources

For subscribing to a resource through a related resource, e.g., using a configuration resource, a new option is proposed, i.e., "observe-uri". This option is interesting for instance when a resource is reachable and is able to change or create the generation of notifications of a third resource. A client might be interested in retrieving, creating or updating the configuration resource and subscribe to the main resource at once.

With this use case in mind, a new CoAP response code is proposed, i.e., "2.15 Content and Subscribed". This response code indicates that the resource representation that the server is including in the server is the one of the resource the client is sending the request to, but the subscription is for the related resource. This way, the client gets the representation of the resource it sends the request to and subscribe to a related resource:

- "Content and Subscribed: 2.15", parallel to 2.05 Content

Figure 5.5 shows an example of a request. The client sends a request for getting a resource named "resource", while subscribing to another

resource. The server responds with a “2.15 Content and Subscribed” code and the representation of the “resource” resource, but it subscribes to a different resource, the one indicated in the “observe-uri” option. The notifications that will arrive after the response have the resource representation of the resource the client has requested to subscribe to, which is different from the first response. If the subscription fails, the response code falls back to “2.05 Content”, also including the configuration resource’s representation. If there are other issues with the request handling, regular CoAP error codes apply. If the related resource does not exist, the response will be a “4.02 Bad-Option” and the response should include the observe-uri option.

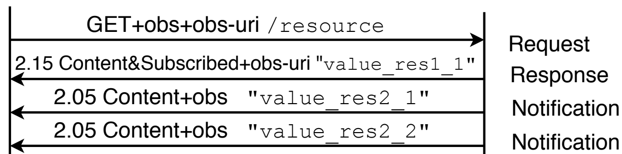


Figure 5.5: Get the representation of a resource, subscribe to a different resource and get notifications.

For POST and PUT request the behaviour is similar, with “2.01 Created” responses if the resource did not previously exist or “2.04 Changed” if it did. Figure 5.6 represents a PUT request to an already existing resource and the server responds with a “2.04 Changed” response message. Then, the notifications of the related resources arrive with the normal notification behaviour. In the event that the resource has been correctly created or updated but the subscription fails, the response does not include the “observe” option. If some other type of error occurs, regular CoAP error codes are to be used, depending on the error. If the related resource does not exist, the error handling is the same as the previous case (GET + “observe” + “observe-uri”), the

response code to be used is “4.02 Bad-Option” with the “observe-uri” option included.

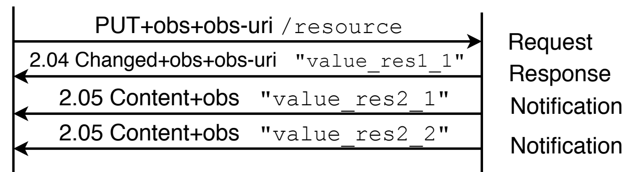


Figure 5.6: Update a resource, subscribe to a different resource and get notifications.

The “observe-uri” option can also be used combined with “no-payload”. Figure 5.7 shows a message exchange between a client that requests a resource and subscribes to a related resource, with no need for getting the current representation of the requested resource. At first glance, this use case seems to not make a lot of sense, as the client could subscribe using a GET request directly to the resource pointed out in the “observe-uri” option and the result would be the same. However, with this message exchange, the client subscribes to the related resource while checking the existence of the requested resource. In order to do that, the client sends a GET request, with “observe”, “no-payload”, and “observe-uri”. The server responds with a “2.10 Subscribed” message if everything goes correctly. If there is an error, the handling mechanisms that have been explained in the previous paragraphs are followed, using regular error codes for general errors, “4.02 Bad-Option” with observe-uri option if the related uri does not exist, and “5.00 Internal Server Error” or “5.03 Service Unavailable” if the client can not be subscribed.

For creating or updating a resource with a PUT or POST request, the interaction is similar as the one needed with GET requests and no payload. The responses will have “2.11 Created and Subscribed”

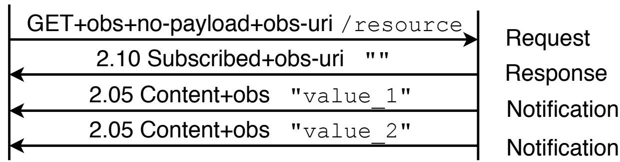


Figure 5.7: Poll a resource without getting the representation, subscribe to a different resource and get notifications.

or "2.14 Changed and Subscribed" code and the observe-uri option. Figure 5.8 shows a successful subscription while changing a resource. If the subscription fails, the response will have a "2.01 Created" or "2.04 Changed" code. Same as a GET request, if the resource in the "observe-uri" option does not exist, the server responds with a "4.02 Bad Option" code and the observe-uri option. If any other error occurs, regular option and response codes are used.

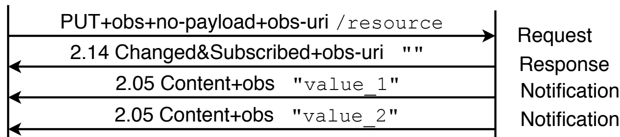


Figure 5.8: Update a resource without getting the representation, subscribe to a different resource and get notifications.

5.3.4 New CoAP options and response codes

Analyzing all the interaction patterns, it can be seen that the new option and response codes do not affect the notifications, they only enhance the subscription mechanisms. The notification generation and delivery works exactly the same way as described on RFC 7641. If an error occurs during the subscription process, the errors must be handled the

way they are described in the current standard. The proposed options and their characteristics are presented in Table 5.2.

No.	C	U	N	R	Name	Format	Length	Default
24					No-payload	empty	0	(none)
43	x	x	-	x	Observe-uri	string	0-255	(none)

C=Critical, U=Unsafe, N=No-Cache-Key, R=Repeatable.

Table 5.2: New proposed options for CoAP.

In order to select the code number, the rules from RFC 7252 have been followed. 24 is the code for "no-payload" and 43 for "observe-uri". "No-payload" is elective (non-critical) and can be safely ignored. If ignored, the server sends the resource representation. It is safe to forward for a proxy for the same reason and it is not repeatable. The meaning of the option is to indicate whether a payload is required in the response, hence, it has no length nor format. "Observe-uri" share similarities with the "uri-path" option. It indicates to which resource the client wants to subscribe. It is critical, unsafe to forward and repeatable, with one option per resource path level. The indicated resource path is represented in string format and has a maximum length of 255. The uri is relative to the resource of the uri-path of the option, hence, contrary to other options related to paths, ".." is a valid value. To be able to subscribe to the resource URI represented with this option, it must be observable.

The new proposed response codes are shown in Table 5.3. These codes indicate that the subscription has been successful, but the resource representation is not included in the response because the client requested not to receive it.

Code	Description
2.10	Subscribed
2.11	Created and Subscribed
2.14	Changed and Subscribed
2.15	Content and Subscribed

Table 5.3: New proposed response codes for CoAP.

5.4 Evaluation: exploratory example

After explaining the new proposed option and response codes, in this section a theoretical evaluation is carried out as a first step. This theoretical evaluation focuses on the exchanged messages in order to validate the proposal. An exploratory use case is presented for the analysis: an alarm clock that counts the time that an industrial process has been running, where the alarm represents the expected end time for the process. The alarm clock consists on two resources: `time` and `time/alarm`. This scenario allows to compare the size of the request and response messages from different actions with the current state of CoAP and its extensions and the proposal from this contribution.

Table 5.4 summarizes the actions that such alarm clock can have, and the message exchange that the actions require both in the current state of CoAP and the new proposal. In the first column there are numbers to identify each action in the chart presented later.

In order to measure the size of the exchanged messages, the size of each section of the header and the payload must be taken into account. Table 5.5 shows the sizes in the alarm clock example. Even though it is not necessary, in the "2.04 Changed" response the payload is also included to ascertain that the resource has been successfully updated.

	Action	New Proposal	Current CoAP
1	Subscribe to an alarm, without needing to know when	GET+obs+no-payload /TIME/ALARM	GET+obs /TIME/ALARM
2	Get the current time and subscribe to an alarm	GET+obs+observe-uri /TIME subscribe to /TIME/ALARM	GET /TIME GET+obs /TIME/ALARM
3	Check that the time exists and subscribe to the alarm	GET+obs+no-payload+observe-uri /TIME subscribe to /TIME/ALARM	GET /TIME GET+obs /TIME/ALARM
4	Change the alarm and subscribe	PUT+obs /TIME/ALARM	PUT /TIME/ALARM GET+obs /TIME/ALARM
5	Change the alarm and without needing a response and subscribe	PUT+obs+no-payload /TIME/ALARM	PUT /TIME/ALARM GET+obs /TIME/ALARM
6	Change the time and subscribe to the alarm	PUT+obs+observe-uri /TIME subscribe to /TIME/ALARM	PUT /TIME GET+obs /TIME/ALARM
7	Change the time and subscribe without needing the alarm	PUT+obs+no-payload+observe-uri /TIME subscribe to /TIME/ALARM	PUT /TIME GET+obs /TIME/ALARM

Table 5.4: Action for a clock use cases with current mechanisms and with the new approach.

Section		**Size** (bytes)
Basic header		4
Observe option		2
Uri-Path option	`time`	5
	`alarm`	6
Observe-uri option	Request `alarm`	6
	Response	1
No-payload option		1
Header separation byte		1
Payload		16

Table 5.5: The sizes of the different parts of a CoAP message in the alarm clock use case.

Other options (e.g., Uri-Host, Uri-Port, Content-format, or Max-Age) have been left out of this comparison as they are exactly the same for the old and new approaches, hence they do not affect the analysis.

Figure 5.9 compares the bytes exchanged to get, update and subscribe to the clock and the alarm, using the current CoAP specification, and the new option and response codes. The different actions are presented in Table 5.4. The first bar for each pair represents the used bytes with the current CoAP options, divided in the request and response (purple) and the subscription request and response (blue). The second bar represents the usage with the new approach in green.

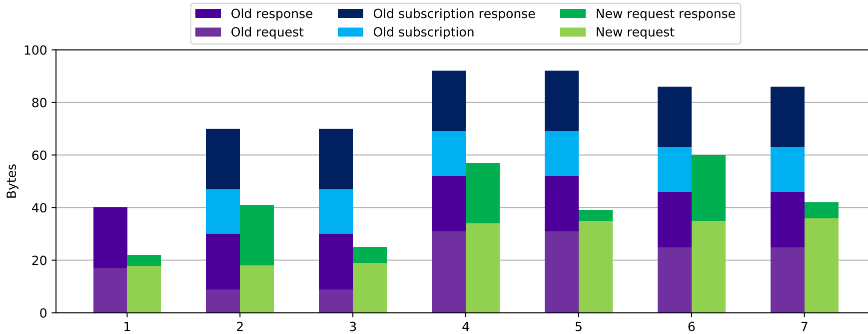


Figure 5.9: Overhead in the old and new approaches in different usages of a clock and alarm use case.

The chart clearly shows that the bytes requires for a successful subscription can be reduced using the new CoAP option and response codes presented in this contribution. Network usage is reduced significantly in each of the actions, due to two main reason. On the one hand, not requiring a payload in the response significantly reduces the response size, as the payload is usually the biggest part of a CoAP message. On the other hand, reducing the two requests (action and subscription) to just one has an even bigger impact on the overhead.

5.5 Implementation in the IEC 61850 Mapping

After evaluating the new enhanced subscription proposal of the CoAP's observe using an exploratory example, in this section the newly proposed CoAP option and response codes are included in an industrial domain, i.e., the IEC 61850 mapping presented in Chapter 4. In that chapter, it was assessed that the reporting services could be improved using more advanced subscription techniques. The main limitation of the CoAP mapping, lies in the *SetURCBValues* and *SetBRCBValues* functions, where additional requests are needed if the report generation is enabled. Even more, when subscribing to reports, the reports might not even be generated, so the request would fail. Adding the new enhanced subscription mechanisms allow to interact with the server in a lighter way.

With the current CoAP specification, if a client wants to subscribe to reports when updating a BRCB or URCB it needs to do it in two steps. First, update the values and then send another message requesting the subscription. With the options presented in this contribution this can be done in a single step. If the client wants to just update the values without subscribing, the request is the same with the new options. The new options allow to support the following requirements in the IEC 61850 to CoAP mapping:

- Subscribe to Reports with the no-payload option activated. This way the report is not requested before it is generated.
- Subscription to the reports through the [UB]RCB resource, instead needing a second request directly to the report.
- Subscription when updating the [UB]RCB resource, i.e., with a PUT request.

The mapping of the *SetBRCBValues* and *SetURCBValues* functions of the IEC 61850 standard are updated as presented in Table 5.6.

	Current CoAP	
SetBRCBValues	coap://{host}/LDs/{LDs}/{LN}/{BRCBs}/BRCB	PUT
Subscribe	coap://{host}/LDs/{LDs}/{LN}/{BRCBs}/BRCB/Reports	GET+Obs
SetURCBValues	coap://{host}/LDs/{LDs}/{LN}/{URCBs}/URCB	PUT
Subscribe	coap://{host}/LDs/{LDs}/{LN}/{URCBs}/URCB/Report	GET+Obs
	New CoAP option and response codes	
SetBRCBValues	coap://{host}/LDs/{LDs}/{LN}/{BRCBs}/BRCB observe-uri: Reports	PUT+obs+no-payload+ observe-uri
SetURCBValues	coap://{host}/LDs/{LDs}/{LN}/{URCBs}/URCB observe-uri: Report	PUT+obs+no-payload+ observe-uri

Table 5.6: Updating [UB]RCBs and subscribing to Reports with the current CoAP specification and with the new enhanced subscription mechanism.

5.6 Validation

Having updated the mapping to include the new enhanced subscription mechanisms, the next step is to compare the exchanged bytes in this specific use case. Figure 5.10 shows the difference in the number of bytes needed to update an URCB or a BRCB using the *SetURCBValues* or *SetBRCBValues* functions in the current CoAP and with the new mechanisms. For the experiments, two different cases have been considered, large and small reports. The left bar of each pair represent the bytes needed with the current message exchange, including the resource update (in purple) and the subscription messages (in blue). The right bar represents the exchanged bytes making use of the new option and response codes (in green).

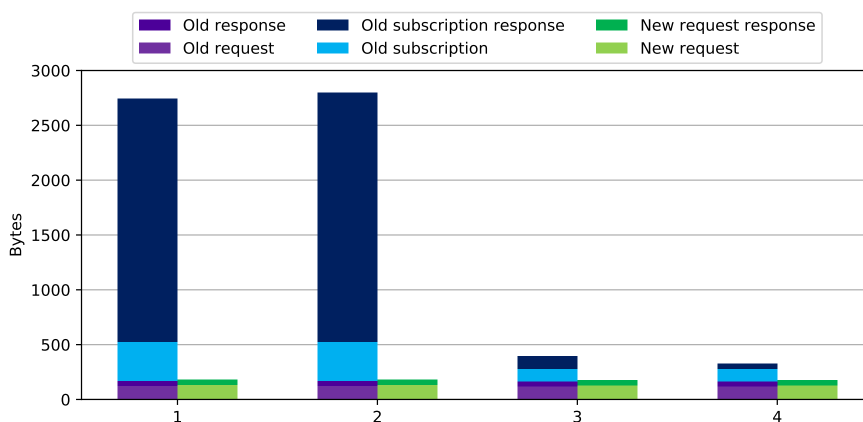


Figure 5.10: Overhead in the old and new approaches: SetBRCBValues (1) and SetURCBValues (2) with a big payload; and SetBRCBValues (3) and SetURCBValues with small payload (4). The left bar of each pair represents the current message exchange with CoAP. The right bar, the new approach with new option and response codes.

The two left pairs show the *Set[UB]RCBValues* with large reports, while the right pairs show the same services with small reports. This example makes it very clear the improvements that can be achieved using the new options, as in this case, the reports can be quite large and not sending them can improve the network usage. A comparison on latency times of the new and old approaches has been considered, but as different number of messages are necessary for both approaches, it has been considered that it is not fair, hence, it has not been conducted.

5.7 Conclusion

After validating the new proposed CoAP option and response codes integrating them in the IEC 61850 mapping, some conclusions can be provided.

First, after reviewing the CoAP specification, its extensions, drafts, and other work expanding the capabilities of the notification processes in CoAP, it can be concluded that the enhancement of the subscription mechanisms has not been extensively analyzed. Chapter 4 points out some weak points of the subscription process that have been addressed in this chapter.

In order to try to solve the aforementioned weaknesses, two new CoAP options and four new response codes are proposed in this contribution. With them, a client is able to subscribe to resources with no need to get the current state of the resource, and can also subscribe while changing or creating a resource in one step. Thus, the needed messages are reduced from four (GET, PUT or POST request, response, subscription request, and subscription response) messages to two (request and response).

Second, in some cases, getting the current representation of a resource a client wants to subscribe to might not be necessary when subscribing. It can even be harmful if the resource has no valid value yet, or the representation is large and the network has limitations. This is the case presented in the previous chapter, where the generated reports can be large, needing several blocks using the block-wise transfer. With the new subscription mechanisms this is heavily improved as it has been proven on the previous section.

Last, the possibility of subscribing to a resource while sending a request to a related one might be interesting to a client. This is

especially useful if a resource includes a subresource that includes the information about the notification generation. With the new options, a client can retrieve, update or create the configuration resource and subscribe to the main resource at once.

In essence, the number and the size of the exchanged messages are decreased. This has been proved theoretically in the test use case of the alarm clock and also practically, integrating it on the IEC 61850 mapping to CoAP, which clearly benefits from the advanced subscription mechanisms in the case of the reporting services. In that case, sending the entire report can be avoided, which in the case presented in this contribution can be of more than 2 KB, which using the block-wise transfer, can need a few seconds to be sent, as presented in Chapter 4.

Considering Table 5.1, the mechanisms presented in this contribution only affect the subscription process, not the notification generation and delivery. Hence, they are more optimal in dynamic network or mobile devices, where devices subscribe and unsubscribe more frequently, for example, in Smart Vehicles, where the IEC 61850 has been applied as explained in Section 4.2, and it has also been used in a Catenary-free Tram use case during this thesis, as explained in the next chapter. Another example for a dynamic environment is the case of Smart Factories, where manufacturing machines may move around the warehouse or factories and collaborate with each other to perform tasks.

The Internet is the world's largest library. It's just that all the books are on the floor.

John Allen Paulos

When wireless is perfectly applied the whole earth will be converted into a huge brain, which in fact it is, all things being particles of a real and rhythmic whole. We shall be able to communicate with one another instantly, irrespective of distance. [...] A man will be able to carry one [instrument for communication] in his vest pocket.

Nikola Tesla

6 Software Engineering Techniques in the IoT

Having presented the mapping of IEC 61850 to CoAP in Chapter 4 and overcoming the detected issues describing more advanced subscription mechanisms for CoAP on Chapter 5, this chapter describes how software engineering techniques can be used to accelerate the process of developing software for ICPSs that use IEC 61850 and CoAP, and more generally, Industry 4.0 devices. With these techniques, the development of the systems is faster and less error prone, as they automate the process of generating the source code for deploying in the devices. Software engineering techniques require the use of models and in this contribution two different models have been used, one for each subcontribution. The first subcontribution presents TRILATERAL (sofTware pRoduct lIne based muLtidomain iot ArTifact gEneration

for industRiAL cps), a tool that allows to use a tree view editor to enter the model of the system to be developed and automatically generate IEC 61850 compliant artifacts to be deployed on the hardware. The second subcontribution uses the model defined by the WoT, changing the model from a niche one to a general use one.

Contents

6.1 Introduction 191
6.2 Related Work 194
6.3 Technological Background 199
6.3.1 Model-Driven Engineering and Domain Specific Language 199
6.3.2 Automated code generators 200
6.4 IEC 61850 based model 201
6.4.1 Problem Statement 201
6.4.1.1 Use cases 203
6.4.2 Solution Design: TRILATERAL 206
6.4.3 Implementation 210
6.4.4 Evaluation 216
6.4.5 Validation 218
6.5 WoT based model 224
6.5.1 First Iteration 225
6.5.1.1 Implementation 225
6.5.1.2 Generic Framework Implementation . 226
6.5.1.3 WoT system implementation 230
6.5.1.4 Evaluation 233
6.5.2 Second iteration 236
6.5.2.1 Abstract syntax 236

6.5.2.2 Concrete syntax 242
6.5.2.3 From Concrete Syntax to Abstract Syntax 244
6.5.2.4 Developing a WoT servient using the WoT Toolkit 246
6.6 Conclusion 250

6.1 Introduction

In the context of Industry 4.0, IoT devices are advanced embedded system within Cyber-Physical Systems (CPS) that require monitorization and control capabilities [200]. These devices are playing an important role in many industrial domains, such as Smart Grids, Smart Manufacturing and Smart Logistics [125, 196]. Combining CPSs with the potential of IoT in industrial processes is an important aspect of Industry 4.0 [106], as it allows to control and monitor an Industrial CPS (ICPS) from the outside, enabling automated analysis, decision making, and advanced anomaly detection. As stated in Chapter 1, interoperability is of vital importance in this kind of scenarios to avoid siloed solutions.

A recurring issue in this type of developments is that existing and new projects do not reuse work and each new project is developed from scratch [213]. This makes the projects costly and more error prone, which is very negative, especially on industrial environments, where safety is of critical importance. There is previous research in this domain, using models and semantics to enable interoperability, e.g., [2], [158], or [126]. A promising approach to overcome these issues is to use Model-Driven Engineering (MDE) methodology and the Software Product Line (SPL) paradigm, which allow to ease the implementation process along with code reuse, accelerating the development time while

reducing bugs, errors and costs. SPL can be beneficial to improve productivity and reduce costs [26], while Domain Specific Languages (DSL) allow to simplify complex codes [54, 70], following an MDE approach. Thanks to DSLs, the systems are more flexible as well as able to provide more immediate responses [110]. SPL on the other hand facilitates the reuse of common elements in different systems. In the first subcontribution of this contribution we applied MDE and SPL to the IEC 61850 implementation presented in Chapter 4.

Connected objects are siloed if they use proprietary or ad-hoc standards that are not interoperable. Hence, this chapter presents two subcontributions, where MDE and SPL are applied to develop two tools that generate source code automatically, using standards with the objective to eliminate siloed systems. The first one uses IEC 61850 to create the metamodel for generating code for ICPSs, while the second one uses the WoT TD as the basis for its metamodel, to then generate WoT servients (a Servient is a software stack that implements the WoT building blocks which allow the implementation of systems that conform with the abstract WoT Architecture). The first subcontribution is TRILATERAL (sofTware pRoduct lIne based muLtidomain iot ArTifact gEneration for industRiAL cps), which uses the IEC 61850 standard as the basis for creating a metamodel to generate an artifact to be deployed on a ICPS, based on a model inserted in the tool using a tree view graphical editor. This work has been carried out in collaboration with another PhD student[1]. Considering the number of devices an ICPS can contain, capturing their data and knowing what is happening in the ICPS is important [84]. The devices that conform an ICPS can be classified in three categories, i.e., sensors, actuators and displays [82], and

[1] `https://orcid.org/0000-0001-9220-047X`

in order to exchange their information, communication protocols are needed, which can be a different one depending on the characteristics of the diverse use cases. To allow different communication protocols that fit best for the different use cases, a modular approach has been followed in the development of TRILATERAL, which allows to select between HTTP, SOAP and CoAP communications, but it is open to include additional protocols if needed.

However, TRILATERAL has two main downsides. On the one hand, it uses a standard that although having been applied on different domains and having other derived standards such as IEC 61400 [71] for wind turbines, it was specifically designed for electrical substations. On the other hand, IEC 61850 specifies a model that it is not very flexible and can be large. This is why the second subcontribution presented in this chapter uses the WoT TD, which aims to support IoT interoperability using already existing standards, following a similar approach to the World Wide Web (WWW). Regarding the development costs, MDE and SPL can help accelerate the development process and increase code quality while decreasing costs, as it is explained in Section 6.4.

This second subcontribution, connects MDE with the WoT in order to increase the speed of IoT system development and deployments, maintaining interoperability in an easy, fast and reliable way, using a general use specification instead of a niche one as IEC 61850. To achieve this goal, a metamodel based on the WoT Thing Description (TD) [216] and a code generator that uses the TD metamodel to create C++ code have been developed. With those two pieces, the code for a WoT servient can be generated using a tree view tool.

In summary, this chapter describes two tools that apply software engineering techniques, each one defining a DSL (based on IEC 61850 and WoT TD) to then use SPL to reuse the DSL metamodels in different domain and use cases. In addition, the generated tools provide communication capabilities in order to enable data exchange between different devices (i.e., ICPSs for the IEC 61850 based model and servients for the WoT TD based model).

The remainder of this chapter is organized as it follows. First, Section 6.2 surveys the related work regarding the automatic code generation approaches for IoT or CPS domains focusing on MDE. Next, the technology used in this contribution is summarized. Section 6.4 describes the first subcontribution, which uses an IEC 61850 based metamodel. Section 6.5 uses a metamodel based on the WoT TD. Finally, some conclusions are provided.

6.2 Related Work

Following the introduction, this section reviews the related literature. In this regard, some research has been carried out to automate the code generation for IoT systems, as well as using MDE techniques in IoT domains. This section analyzes the related literature, describing the available proposals and the downsides they present.

First, there are several works that address the issue of development and maintenance costs using reuse techniques. SPL and MDE are two paradigms that allow code and element reuse, and are becoming increasingly common in industry [26, 224]. Research has also been carried out to use SPL and MDE when developing ICPSs, by managing the variability between devices inside an ICPS [82, 198, 187, 9]. A web-based DSL has also been proposed to model IoT systems in [189].

Another proposal using DSL in the resource constrained devices of the IoT is presented by Negash et al. [143], where the behavior of client nodes can be changed dynamically through a gateway.

Based on a model, a model-to-text transformation can be done, where the generated text is source code. In this regard, Riedel et al. [173] use a Model Driven Software Development (MDSD) approach in order to generate code for IoT systems. They present a tool that automatically generates C, C# and Java code for Windows, Linux, Symbian, Particle, and Contiki, using SOAP Web Services as the communication protocol. The authors use Essential Meta-Object Facility (EMOF) for data meta-models and Eclipse Modeling Framework (EMF) to generate the code, translating the messages of the IoT subsystems. With this tool, developers can configure what part of the generated code runs on the gateway and what part on the IoT node. The gateways are able to transform the XML representation of the resources to other more lightweight formats to fit in more resource constrained devices. In the scenario Riedel et al. present, the gateway and IoT node run on the same mobile device with hardware interfaces. However, the authors claim that using different physical devices for the gateway and other parts should not be an issue. Yet, the performance is rather low when the number of nodes grows to bigger that 50, as SOAP can not cope with the overhead produced by the underlying TCP stack. Thus, according to the authors, a RESTful approach using HTTP is neither a solution when many nodes participate in the networks as HTTP also uses TCP.

Another proposal by Ciccozzi and Spalazzese [32] also proposes the use of MDE for generating code to deploy on IoT systems. In this work, a Smart Street Lights scenario is presented, where the lights include sensors to detect cars, bikes, emergency vehicles, infrastructures, and

people. This way, the street lights can be adapted to their presence and speed. The authors present the MDE4IoT framework to generate the code, for which they design a model for an IoT adaptive model. The individual parts of the framework are also validated, but the main downside of this approach is that the model used is their own, instead of using a standard one such as the WoT.

Calcina-Ccori et al. [22] propose to generate the source code with a different approach. They work on top of an organic network of devices that collaborate together named *Swarm*. To be able to work on a Swarm based framework, the authors present SwarmBroker, which is a broker that handles the communication and cooperation among the devices present in the Swarm network. This approach is more aligned with the WoT than the previous one, as it also defines the resources as *servients* (server + client) and uses a RESTful communication over HTTP. In order to address the distributed nature of Swarm, Calcina-Ccori et al. use SwarmBroker on each device, with lightweight brokers for resource constrained devices. The local brokers act as directories and each device registers its services in them. The objective is to generate the source code based on high level service descriptions, and they use Swagger for generating the ready-to-run code and documentation. According to the authors, the achieved efficiency is of 500% in the development time for the automatic code generation against of a manual coding approach. However, this work is also based on TCP communication protocols, hence, it shares the same problems as the previous ones.

In [176], Salihbegovic et al. present a high level DSL for designing IoT devices, i.e., DSL-4-IoT. It includes a visual editor and generates code for OpenHAB[1]. OpenHAB allows to interface software compo-

[1] `https://www.openhab.org/`

nents and middlewares with hardware and software, and to map the device code to software components. The code generated by the DSL-4-IoT is Java code for servers, and the authors test the solution in Smart Homes and remote patient monitoring use cases.

Another analyzed approach that automatically generates source code from a model is the one by Delicato et al. [43]. In this work, the authors propose a middleware to add nodes to a wireless sensor network. As this approach is built from scratch, the authors were able to design their solution using the WoT paradigm and use HTTP as the communication protocol. This project was tested on a parking lot use case, where sensors inform about free parking spaces. However, the main focus of this work is to connect SmartSensors using a mashup network, hence, it is limited to this kind of devices.

In [205], Thramboulidis et al. present their approach for integrating CPSs and IoT with a tool named UML4IoT. They use a framework to address the challenges of using IoT in the development process of manufacturing systems. The authors aim to automate the process of generating CPSs with their framework. To do that, the CPSs are modeled using SysML and implemented on a conventional Object Oriented API, to then transform that API into a RESTful API. In this approach, LWM2M is used to enable communication capabilities.

The last analyzed piece of research is the one conducted by Usman et al. [212], where the authors propose to use SPL along with MDE in order to ease the maintenance of mobile applications for different operative systems. Their solution is a tool named MOPPET, which implements a feature model to model the application and then automatically generate the different variations. However, the main drawback of

MOPPET is that it does not generate the user interface, it focuses on the application logic.

Table 6.1 summarizes the analyzed related work presented in the previous paragraphs.

Work	Proposal	Downside
[173]	Generate C, C# and Java source code for Windows, Linux, Symbian, Particle, and Contiki with SOAP communication	Heavyweight languages and protocols
[32]	MDE4IoT framework, with their own model	Not standard model
[22]	Generate code from high level service description for a broker in Swarm over HTTP	Heavyweight protocol
[176]	Visual editor to generate code for OpenHAB in Java	Heavyweight language
[43]	Generate WoT code using HTTP	Specific use case
[205]	UML4IoT, framework for integrating CPSs and IoT based on SysML, using Object Oriented API	
[212]	Maintain different OS mobile applications, generating variations automatically.	Manual user interface

Table 6.1: Previous CoAP benchmarks.

In summary, the works analyzed in the previous paragraphs leave open issues that are addressed with this contribution. Some of the approaches generate heavyweight languages like Java [176] or use heavyweight communication protocols [173, 22], which makes them hard to use in resource constrained devices. This is why they generate software for gateways and not the actual IoT devices [173]. The other contributions have the downside that they focus on a single type of nodes (e.g., [43]). The approach presented in this contribution is twofold. On the one hand, in the first subcontribution TRILATERAL is presented, which based on the analyzed literature, is the first tool that aims to ease the development of ICPSs following SPL and MDE techniques. It uses a DSL to generate a metamodel based on the IEC 61850

standard, and is able to select the adequate communication protocol (i.e., HTTP, SOAP, or CoAP) for each use case. On the other hand, the second subcontribution is based on the WoT, which aims to be the linking path to achieve interoperability between different IoT systems and overcome the problem of siloed systems, while being able to run on resource constrained devices.

6.3 Technological Background

After reviewing the related literature, this section summarizes the technologies used for this contribution.

6.3.1 Model-Driven Engineering and Domain Specific Language

One of the main technologies used in this contribution is the Model-Driven Engineering (MDE) approach. MDE is a discipline that proposes to increase the abstraction level to automate the development of software using models. Abstraction allows to cope with complexity while automation enables to boost productivity and quality.

A Domain Specific Language (DSL) is a language that is designed to perform tasks in a specific domain. DSLs simplify complex codes in order to enable effective communications with stakeholders [54]. However, the concepts and notation of a DSL should be as close as possible to the concepts of the domain and the representation end users use in their daily practice. Usually, the concepts of a DSL (*abstract syntax*) can be described using models such as class diagrams, while different forms are used to specify the notation (*concrete syntax*), depending on the desired representation.

Since concepts of a Domain Specific Language (DSL) are typically described using models, there is a perfect synergy between MDE and DSL standards and tools. One example is the Meta-Object Facility (MOF)[1] standard. MOF is a subset of UML class diagrams, a vocabulary that it is used to define the abstract syntax of a language. Thus, IEC 61850 elements and WoT concepts can be expressed using MOF, as presented in the following sections.

6.3.2 Automated code generators

The Eclipse[2] ecosystems includes several projects for different tools. One of them is the Eclipse Modeling Framework (EMF)[3], which aims to enable model definition and their instance using the Ecore language and generate software artifacts based on a model description. Eclipse and EMF have become a de facto standard for creating model-based tools with a common base for different purposes, e.g., model transformation, reverse engineering, code generation, or document generation for instance. Some additional tools offered by Eclipse are Xtext[4] and the Xtend[5]. The former allows to develop programming languages and DSLs with code generation capabilities, while the latter is a flexible and expressive Java dialect that compiles into readable Java compatible source code. In short, Xtend provides code generation support to Xtext. Due to the high integration level of Xtext, Xtend has been used to implement the code generators presented in this contribution.

[1] https://www.omg.org/mof/
[2] https://www.eclipse.org/
[3] https://www.eclipse.org/modeling/emf/
[4] https://www.eclipse.org/Xtext/
[5] https://www.eclipse.org/xtend/

6.4 IEC 61850 based model

After describing the main technologies for this contribution, in this section the first subcontribution is presented. This subcontributions makes use of the information model of the IEC 61850 standard which has been explained in Chapter 2 and 4. This section first states some problems of the development of ICPSs. Then, the proposed solution is presented, i.e., TRILATERAL. After that, the implementation is explained, following an evaluation and a validation.

6.4.1 Problem Statement

Considering the work carried out in [82], the fulfilled domain analysis shows that different domains can share requirements in control and monitoring use cases. And these use cases are pivotal in Industry 4.0.

Every ICPS is different and it includes different devices [83], so when a data capturing or monitoring system needs to be implemented, an ad-hoc solution is created. The ad-hoc solution is manual, with an error prone and time consuming implementation process [198, 21]. This makes the costs for the development costly in terms of time and resources. Figure 6.1 shows different levels of automation, which can also be considered generic solutions.

On the left part of the figure, three individual industrial domains are shown, each of them with their own individual development process represented in different colors. Taking into account that the industrial domains, although being different, share some common requirements that make the solution for these requirements generic, which improves the code quality and decreases development time. In order to do that, first it has to be analyzed whether the different domains can use the same

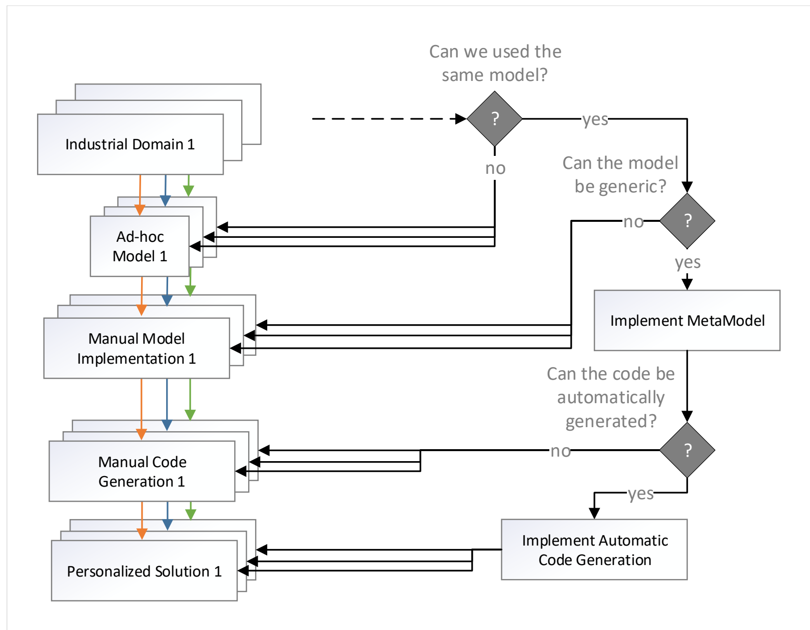


Figure 6.1: Manual vs Automated processes.

base model. If not, each project needs its own model. Otherwise, the next step is to analyze if the model can be generic. If so, a metamodel can be generated, using a DSL. Finally, it has to be analyzed whether the code to implement the metamodel can be automatically generated using MDE techniques. If it can, an automatic code generator needs to be implemented.

Following this process, in this subcontribution the IEC 61850 has been selected as the generic model. Based on that, a DSL has been defined to describe the metamodel, and finally an automatic code generator has been developed, which creates the artifact to be deployed on an ICPs. With this approach, a model can be implemented using the IEC 61850 metamodel, using a tree view editor to input the specific model of the system and selecting the communication protocol (HTTP, SOAP and CoAP). This allows to have a solution that is interoperable and intuitive regarding the device and data modeling and naming, as it has a hierarchical structure, and has lower cost for development, deployment, and maintenance.

6.4.1.1 **Use cases**

Thanks to the relation of Ikerlan with industrial companies, three use cases or application domains have been considered, i.e., Wind Farm, Smart Elevator and Catenary-free Tram. These use cases have different needs on connectivity, due to their application environment having very different characteristics, as can be seen in Table 6.2. IEC 61850 has also shown that it can be applied on other domains such as Press Machines or Automated Warehouses [82], but for this subcontribution, they have not been considered as they are too similar to the Smart Elevator use

case, where the ICPSs are in a well connected, controlled physical space.

Wind farm	Smart elevator	Catenary-free tram
• Remote location • Connectivity issues	• Controlled environment • No connectivity/power issues	Station • Controlled environment • Smaller header with big payloads On route • Mobile • Connectivity/Power issues

Table 6.2: Characteristics of the different domains regarding the connectivity.

Wind Farm

A Wind Farm has wind turbines generating energy using wind power. The turbines need to be monitored to know how much energy they generate in real time. They are critical infrastructure that can create huge problems if they fail or malfunction, so some parameters need to be monitored to ensure safety. In order to monitor them, the particular features of each wind turbine on a farm have to be represented. As each Wind Farm can have a different configuration, it is very variable, and hence, writing the source code for the monitoring of the Wind Farms can be time consuming.

Regarding the connectivity, Wind Farms tend to be located on remote locations, where the properties of the wind are more optimal for generating power, i.e., stronger or more constant wind. Hence, the farms are located far from cities, in locations where connectivity can be limited, such as the top of mountains or offshore. This environment

is not mobile, but heavyweight protocols can be a huge handicap if the connectivity is limited.

Smart Elevator

A big building may have several elevators with a control and monitoring system to make a more efficient use of them. Currently, some elevators include batteries that store energy when the elevator is moving down and later use that energy to improve the energetic efficiency of the building, for instance, to use the energy of the batteries when the energy cost is higher. In this use case, monitoring and controlling the batteries of each elevator can help to make decisions on their use.

Elevators are located inside buildings, hence, they are in a location that is controlled with not many problems of connectivity or energy supply. This results on a scenario where the communication protocol constraints are not as severe as in others.

Catenary-free Tram

All transportation systems have parameters to monitor, from critical parameters (i.e., speed, directions or maintenance related information) to non critical systems such as information or multimedia features, or climate systems. A tram is no different, and in the case of a catenary-free one, an efficient energy system needs to be developed in order to control the power so the tram arrives to the next charging point.

In this scenario, two types of communications occur, i.e., when the tram is on route and when the tram is on a stop. The former may have power and connectivity limitation, while the latter does not, hence, much more information can be bulked. However, since a tram has to follow a schedule, the time for bulking the data may also be limited,

so the time needed to transfer the data has also need to be taken into consideration.

6.4.2 Solution Design: TRILATERAL

For providing a common solution for all the use cases presented in the previous section, a tool named TRILATERAL has been developed. With TRILATERAL a system designer is able to create an IEC 61850 compatible data model graphically, and TRILATERAL generates the source code that is used in the artifact to be deployed on the ICPS. The artifact is the framework responsible for establishing communications between the devices of the ICPS and the monitoring system using HTTP, SOAP or CoAP (the one or ones selected by the system designer).

TRILATERAL is a SPL and MDE tool that uses a DSL based on IEC 61850 and offers monitoring and controlling capabilities using one or more of the aforementioned protocols (i.e., HTTP, SOAP and CoAP). The DSL is used to generate a IEC 61850 based model. A tool with this features allows to reduce costs in terms of development, testing and deployment, as it reuses parts that have already been developed and validated.

There are two main parts on TRILATERAL as shown in Figure 6.2: the *server model* and the *information model* definitions. The user creates both of them using the tree view editor. On the one hand, the server model definition is used to select the device logic and which communication protocol the system will use to communicate. On the other hand, the information model is used to describe the ICPS structure, defining all the devices and attributes.

For the information model, a model based on the IEC 61850 meta-model has been created using the DSL. The BIM and the CBs of the

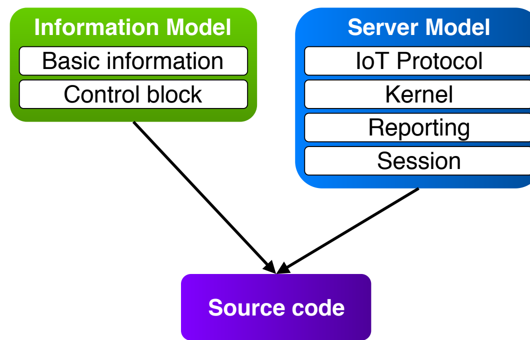


Figure 6.2: TRILATERAL components.

IEC 61850 are modeled, where the BIM defines the elements of the physical world (Logical Device, Logical Node, Data, DataAttribute) and the CBs add additional functionalities to the information model, in this case, the reporting functions.

The server model is also configured using TRILATERAL. In this model, the user selects the communication protocol or protocols, and configures some parameters of the kernel, the reporting, and the user session to manage the ICPS. Thanks to the DSL TRILATERAL implements, the user can automatically configure the server to communicate the artefact with an external system. Figure 6.3 presents the workflow to use TRILATERAL, with the several steps that have to be followed to create an artifact:

1. First, the user configures the server, selecting the communication protocol between SOAP, HTTP and CoAP. The user chooses the protocol that is best for the specific use cases and if needed, more that one can be selected. The session, kernel and reporting parameters (e.g., reporting buffer size, location of configuration

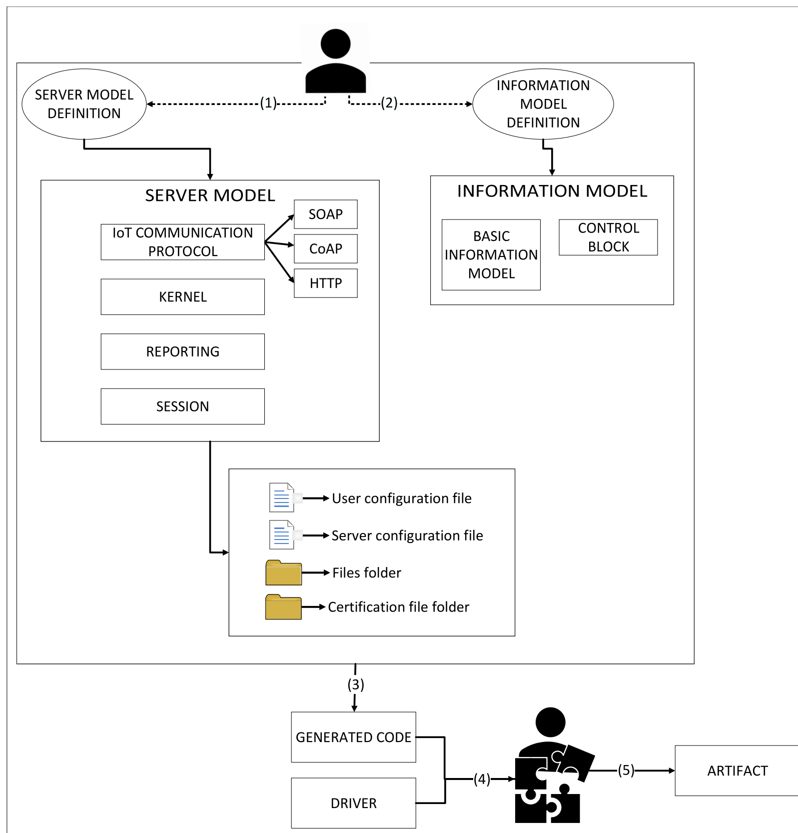


Figure 6.3: TRILATERAL workflow.

files, refresh times, ports, etc.) are also defined in this step. Figure 6.4 shows a tree with all the parameters that can be specified.

Once the communication protocol and the parameters have been defined, TRILATERAL automatically creates the needed files for the user to enter the necessary information. For instance, TRILATERAL creates the users file, where the user can enter user/password pairs. Finally, TRILATERAL also creates two directories, one to keep the certificates and another one for the files that the IEC 61850 standards needs to offer file management functionalities.

2. In this step, the user describes the structure of the information model, following IEC 61850. To do that, the user creates the tree with the BIM (Logical Device, Logical Node, Data, DataAttribute, and DataSets) and the CBs (Report), using TRILATERAL.

3. After configuring the server model and the information model, in this step TRILATERAL automatically generates the source code using Model-to-Text (M2T) transformation.

4. In order to complete the artifact, the user needs to introduce the needed driver (the connector that links the physical devices to the information model). Then, the generated code from the previous step is compiled along with the specific driver inserted by the user.

5. In the final step, after compiling the driver and the device logic, the libraries and executables to be deployed in the ICPS are generated, i.e., the artifact.

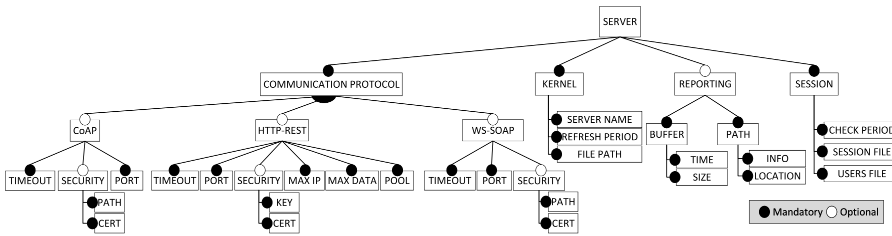


Figure 6.4: Server definition feature model.

to be able to start monitoring and controling the ICPS, the artifact needs to be deployed in a device within the ICPS. When the deployment is completed, outer systems can communicate with the artifact using CRUD (Create, Read, Update, and Delete) functions. Thus, a middleware between an ICPS and monitoring system can be created using TRILATERAL, which allows to monitor and control all the devices within an ICPS.

6.4.3 **Implementation**

Followint the description of the design of TRILATERAL, in this section the implementation is explained. Figure 6.5 shows the steps that have been followed to develop TRILATERAL, a graphical Eclipse plugin developed using the Eclipse Modeling Framework (EMF)[1] [192].

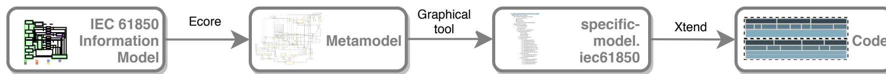


Figure 6.5: TRILATERAL implementation steps.

The first step has been to create a common metamodel to be used in all the different use cases. In order to do that, EMF provides the

[1] https://www.eclipse.org/modeling/emf/

Ecore metamodel, which allows to create other metamodels. In this case, Ecore has been used to create a new metamodel for the IEC 61850 standard. Figure 6.6 shows a simplified version of the metamodel generated with Ecore, that models IEC 61850.

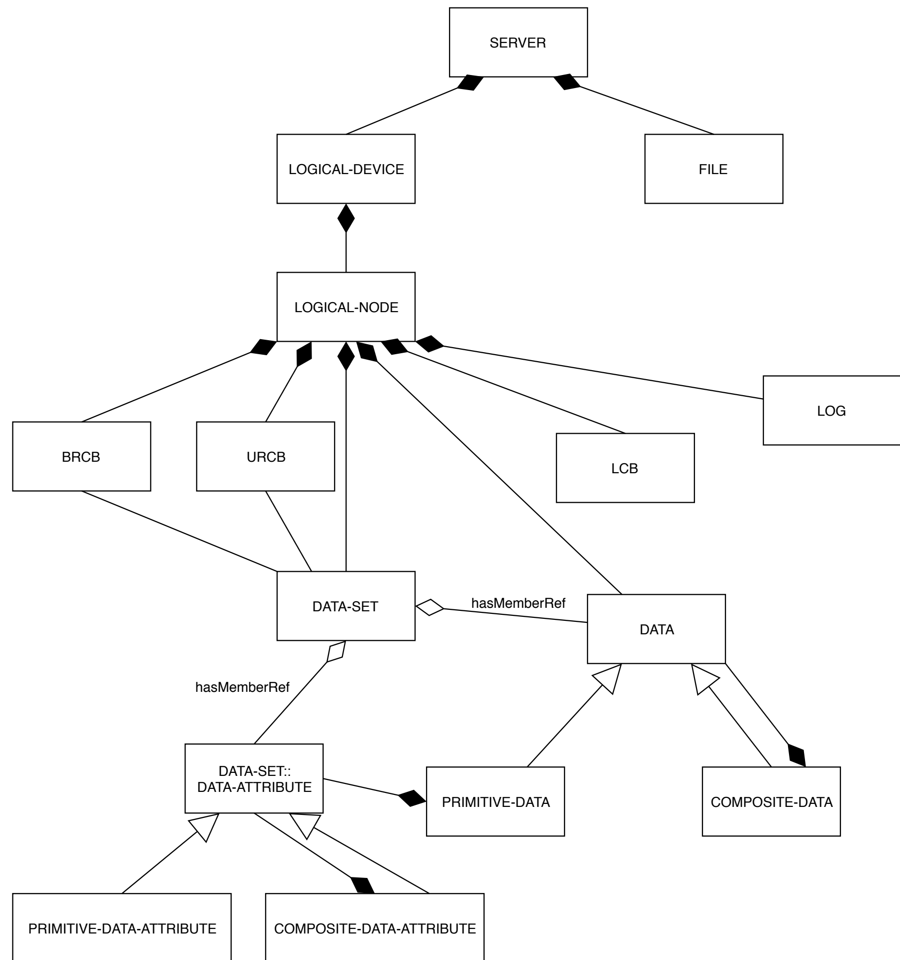


Figure 6.6: Simplified version of the metamodel used in TRILATERAL.

Once the metamodel has been defined with Ecore, the EMF has generated a tree view editor that allows to describe specific models.

The specific models are files with the .iec61850 extension, which is consistent with the metamodel's name. Users describe the model of their system using the tree view editor, which creates the .iec61850 file. Figure 6.7 shows a screenshot of the tree view editor, where a Smart Elevator has been modeled. The model shows a Server named *SERVER_NODE*, which has a LD named *SmartElevator_LD*. *SmartElevator_LD* has four LN, and each of them has its own Datas, Datasets and Reporting CBs. This file can also be opened as a XML formatted text file.

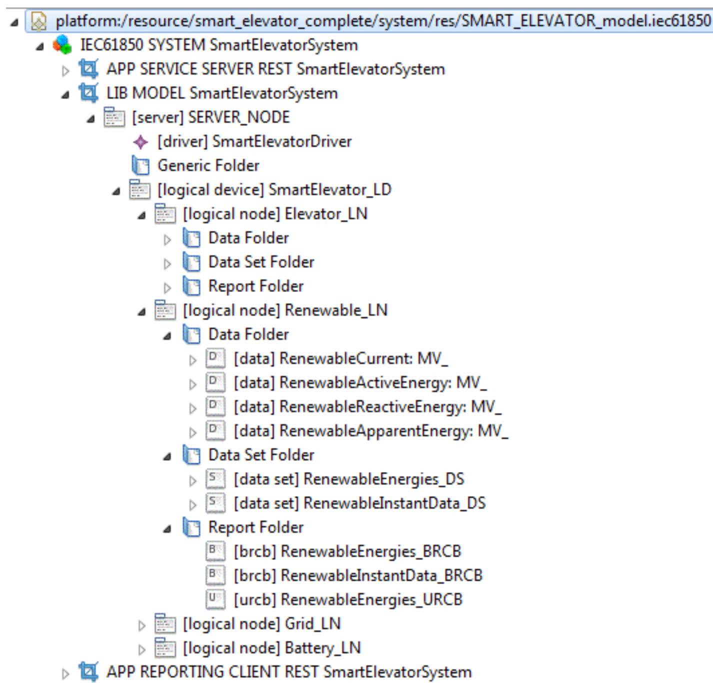


Figure 6.7: Screenshot of TRILATERAL's tree view editor.

The next step to generate the source code is to take the .iec61850 file and use a code generator to convert the model defined in the .iec61850

file into source code. In order to do that, EMF offers a tool named Xtend, which makes M2T transformation. A code generator has been implemented using Xtend, to create C++ code based on the .iec61850 file. The code generator creates header (.h) and source (.cpp) files. Some of the files are generated based on the model, while others insert the needed libraries to the generated code.

On the tree view editor the communication protocol is also chosen. SOAP, HTTP and CoAP are the current available options, and the generated code depends on this selection. The generated code consists on several Eclipse projects organized in layers, as shown in Figure 6.8.

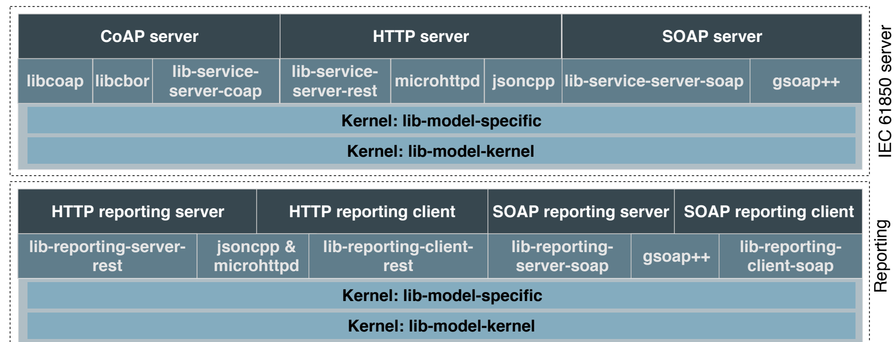


Figure 6.8: Layers of the IEC tool.

The generated code is organized in three layers of Eclipse projects, i.e., kernel, libraries and applications.

- The **Kernel** layer includes two projects, the `lib-model-kernel` for the generic classes of the model, and the `lib-model-`*`specific`*, which includes the specific model generated with the tree view editor, unique for each use case.

- On top of that, the **Libraries** layer enclosures the auxiliary libraries that are used for generating the server applications, de-

pending on the selected communication protocol. These libraries can be grouped in two groups, the libraries that help to use the resources for each communication protocol and the auxiliary libraries that help with the other aspects, such as resource representation formats, server generation, etc.

  - For **CoAP** communication, `lib-service-server-coap` links the model to the communication protocol, `lib-coap` is used as the CoAP library and `libcbor` allows to represent the resources using CBOR.
  - In the case of **HTTP**, `lib-service-server-rest` is the library to link the model to the HTTP protocol, `json-cpp` for representing the resources using the JSON format and `microhttpd` to implement the HTTP server.
  - **SOAP** communications are implemented using the `lib-service-server-soap` library. In addition, `gsoap++` offers all the needed functions to use SOAP communication, including server functions and the XML resource representation parsing.

- Last, the specific application servers are on the **Applications** layer. One server is generated per protocol, and only the needed stack is present in each of the executables. This means that if only a CoAP server is modeled in TRILATERAL, the executable will not include any of the HTTP or SOAP libraries.

As previously explained in Chapter 4, the reporting services using SOAP and HTTP work separately from the main applications. These communication protocols need separate server and clients because the

communication paradigm changes due to them following the client-server communication model. CoAP has the Observe extension to enable publish-subscribe communication, so it does not have this issue. The separate client and servers for reporting services have the same three layers. The lowest layer is the same, with the kernel libraries being common to the main applications'. In the second layer, `jsoncpp`, `microhttpd`, and `gsoap++` are also present as auxiliary libraries, but there are also auxiliary libraries that are not present in the main application. These new libraries are `lib-reporting-server-rest`, `lib-reporting-client-rest`, `lib-reporting-server-soap`, and `lib-reporting-client-soap`, which help to create the application on the uppermost layer. The applications on the uppermost layers include servers and clients for exchanging the reports generated in the main application in HTTP and SOAP. The working way is similar in both cases regarding the reporting functions. The service server stores the reports in a folder defined when configuring the system. The reporting client periodically polls that folder to check whether a report has been generated. In the case of CoAP, this is not necessary, as a client just subscribes to the reports using the observe extension, and when the server generates a report, it sends it to the client using a notification message.

Creating new artifacts for different domains with TRILATERAL is quite trivial, as Figure 6.9 shows. A user just needs to enter the models (Information Model and Server Model) on the tree view editor and TRILATERAL takes the generated .iec61850 file and automatically generates the source code. This source code then needs to be compiled along with the driver so the artifact can be generated.

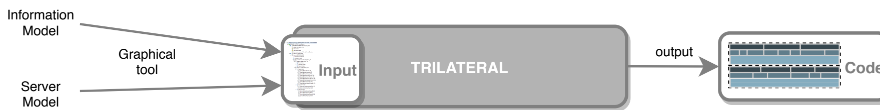


Figure 6.9: Creating a ICPS artifact in TRILATERAL.

6.4.4 Evaluation

After explaining the implementation, this section evaluates TRILATERAL in a Catenary-free Tram use case. As explained in Section 6.4.3, the generated code is divided in projects in different layers. Some of the projects are common for every communication protocol, while others are specific. Table 6.3 presents the sizes that each compiled project needs in KBs, be it a library file or an executable. It also summarizes which included communication protocol uses which project. The code generated by TRILATERAL is based on an already existing implementation of the system, and that implementation has been used as a template. Hence, the generated code and the previously existing code are the same, which makes the performance of both automatically generated implementation and manual implementation the same.

The projects at the uppermost layer named `app-service-server-*` are the final applications of the server for each of the protocols. The rest of the projects of that layer, i.e., `app-reporting-*` are the final reporting server and clients. Below that, in the middle layer, are the auxiliary external libraries and the libraries to generate the application servers and reporting client and servers. The last two libraries are the ones that specify the model. On the one hand, `libmodel-`*`specific`* is very use case dependant, as it describes the unique model for each case, so its size can not be determined without the actual model. On

Library/Executable	KB	CoAP	HTTP	SOAP
app-reporting-client-rest	675.1		X	
app-reporting-client-soap	1600			X
app-reporting-server-rest	1300		X	
app-reporting-server-soap	1600			X
app-service-server-coap	9600	X		
app-service-server-rest	8200		X	
app-service-server-soap	9700			X
libenv.a	1200			X
libgsoap++.a	1600			X
libjsoncpp.a	2000		X	
libcoap.a	814.5	X		
libcbor.a	381.6	X		
libreporting-client-soap.a	2800			X
libreporting-server-soap.a	3000			X
libreporting-client-rest.a	2700		X	
libreporting-server-rest.a	2900		X	
libservice-server-rest.a	6900		X	
libservice-server-soap.a	7100			X
libservice-server-coap.a	6900	X		
libmodel-specific.a	?????	X	X	X
libmodel-kernel.a	25 700	X	X	X
Total	96 671.2	43 396.1	50 375.1	54 300
		45%	52%	56%

Table 6.3: Libraries and executables need for introducing communication protocols to an artifact by TRILATERAL.

the other hand, `libmodel-kernel` is the kernel with the common abstract classes of the model.

Thanks to the modular design of TRILATERAL, selecting the communication protocol in the tree view editor makes the created system optimized, as only the needed projects are included. This leads to a reduction on the range of 50% and 30%, depending on the protocol as shown in Figure 6.10.

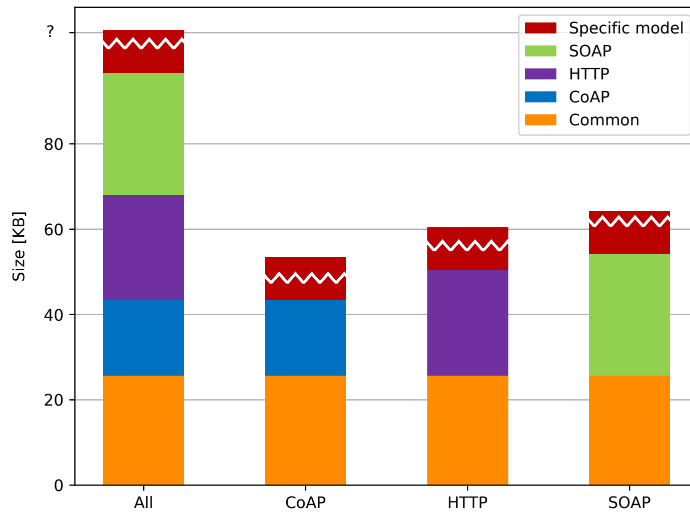


Figure 6.10: Different protocol sizes.

6.4.5 Validation

One of the three analyzed domains has been selected to validate TRILATERAL, i.e., the Catenary-free Tram use case. Trams are a passenger transportation method that run on rails in urban areas. The Catenary-free Trams are being increasingly used in Europe because it has the advantage of simplifying the infrastructure as it does not need a cate-

nary. In order to offer better safety features in trams, it is important to monitor the different systems it has, e.g., engine and brake status, batteries, etc. Depending on the company and route, a tram can have a different configuration of wagons, with different features and devices to monitor. Figure 6.11 shows an example.

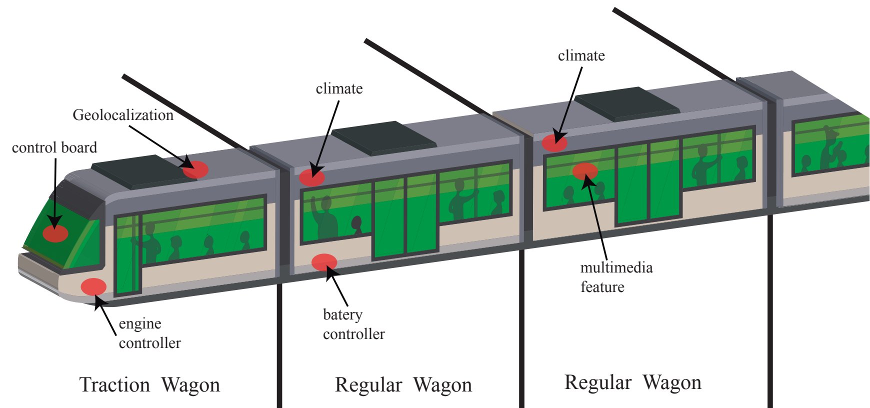


Figure 6.11: Catenary-free tram and its systems.

Hence, Catenary-free Tram systems have different parameters to be monitored. Some of them are critical, e.g., speed, direction or maintenance related information such as battery state, brake wear, etc. Other parameters are not so critical, e.g., multimedia features or HVAC systems. Regular wagons usually include a battery controller to control the entire energy system for the movement of the tram. Other components are also present, which include speed and resistance profiles, engine controlers, braking motors, auxiliary loads, etc.

Apart from the high number of devices, two different working modes have to be taken into account. On the one hand, when the tram is on route, the environment is mobile, which can affect power and

connectivity capabilities. On the other hand, a tram can take advantage of being on a stop or station to bulk big pieces of data. In this case, the time might also be limited as the tram needs to follow a schedule. This working modes make this scenario very complete to analyze, as two communication protocols might be needed, one for each working mode.

The Catenary-free Trams are adapted to each city, line, mechanism, route, etc. That is why the system of each train might have different requirements. As this type of tram does not use any catenary, the available energy is limited, as it needs to arrive to the next charging point. In order to solve that issue, different techniques are used. Some trams use fast change accumulators while others get energy from the rails. Thus, even if the different mechanism requirements are related to energy, different systems are used in the trams. Non critical systems can also be installed in order to collect more information about the tram, but this depends on the stakeholders of the tram system.

In addition, the criticality of elements can change over time, due to changes in the legislation or advances in the systems. This may provoke to update the devices to have a better control on them. Not only that, the tram system can also evolve, introducing more devices to increase the number of monitored devices and even new wagons may need to be added if the tram line has a lot of passengers. This means that the tram structure can evolve over time. Furthermore, the devices may also evolve, in the case they get damaged or obsolete.

Figure 6.12 shows a partial model of a tram in TRILATERAL's tree view editor, i.e., the climate system. This model has a server named *SERVER_NODE*, which is composed of an LD named *ClimateSystem_-*

LD. *ClimateSystem_LD* has two LNs, and they have several Datas, Datasets and Reporting CBs.

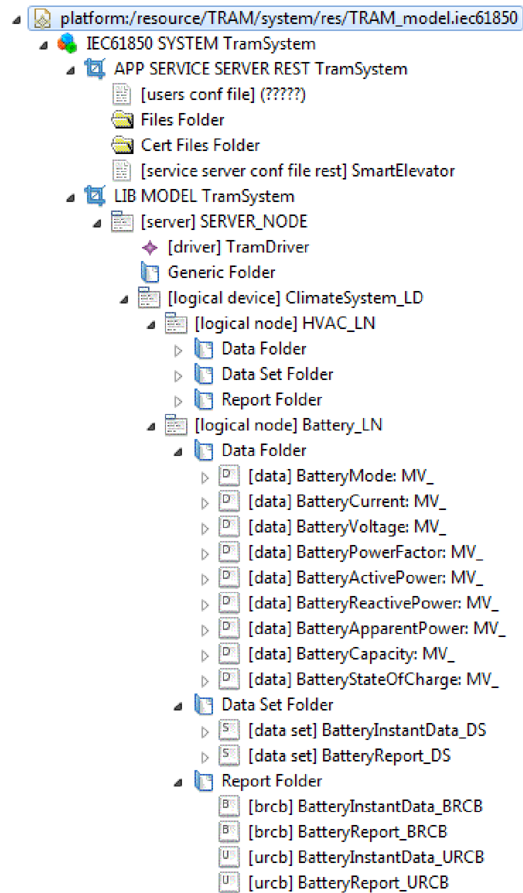


Figure 6.12: Screenshot of TRILATERAL's tree view editor describing a part of the climate system of a catenary-free tram.

In this use case, two scenarios can be differentiated, one for the running tram and the other one for the tram on a station or stop. Depending on the scenario, the environment is different, which results in different communication requirements, hence, different selected communication

protocols. For the first case, i.e., when the tram that is moving, CoAP has been the selected protocol, as it is a mobile scenario where connectivity or power may be limited. For the second case, HTTP has been selected, because the scenario is a controlled environment, with no connectivity issues.

As explained in subsection 6.4.3, the generated `libmodel-`*`specific`* is unique for each model and is variable in size. In this use case, the library has been named `libmodel-tram`, which compiled has a size of 11,700 KBs.

Once TRILATERAL has been developed and validated with a Catenary-free Tram use case, it can be concluded that the proposed solution can be used in a transport domain, even if IEC 61850 was designed for electrical substations. Hence, it also can be used in other IoT domains, such as transport, elevation or energy, that have remote monitoring and control requirements.

In addition, the benefits of SPL and MDE are proven. Event though there exist diverse domains with different characteristics, many of them share similarities and have common requirements. Although the development of the DSL to create the metamodel of the IEC 61850 has been complex, once completed, generating an artifact for a monitoring and controlling IoT system has become much simpler, mainly due to the tree view editor.

This has been proved internally with three use cases from different domains. The other use cases are Smart Elevators for big buildings that include several elevators and Wind Farms. Table 6.4 lists the use cases and the used protocol in each of them, along with an explanation of their characteristics.

Protocol	Use case		
	Wind farm	Smart elevator	Catenary-free tram
WS-SOAP		- Controlled environment - No connectivity/power issues	
HTTP-REST	- Remote location - Connectivity issues		Station - Controlled environment - Smaller header with big payloads
CoAP			On route - Mobile - Connectivity/Power issues

Table 6.4: Characteristics of the use cases and the selected communication protocol.

TRILATERAL has been used to model the three use cases and generate the source code to complete the artifacts deployed on the ICPSs. The artifacts allow to remotely monitor and control IEC 61850 compliant devices, using the adequate communication protocol for each use case.

The Smart Elevator use case is a controlled environment with no power or connectivity problems. SOAP has been the selected communication protocol for this scenario, which performs satisfactorily. Each elevator has been modeled with its own LD.

After the Smart Elevator use case, the second deployed use case has been the Catenary-free Tram. As previously explained, two different scenarios have been considered, a moving tram and a tram on a stop. Three LDs have been used to model the Catenary-free Tram, one for batteries and electrical components, another one for the active demand management system and the last one for the climate system. In the deployment of this use case, an issue has been detected, the system was too heavyweight. That is why a change has been made to the initial version of TRILATERAL, making the system more modular. This

way, each part of the kernel includes only its corresponding part of the system, leading to a lighter weight of each part.

The last use case, i.e., the Wind Farm, has proven this to be a positive change. In this use case, a LD has been used for each wind turbine. The first use case has also been rebuild to update the system and the new build is more lightweight.

Summarizing, TRILATERAL has allowed Ikerlan to improve the process of developing ICPSs, reducing development time in addition to improving the validation and maintenance tasks. The validation and maintenance tasks are improved thanks to the reusability of the code, meaning that the already validated parts do not need to undergo though another testing phase for new projects. All this results on an increased code quality and hence, higher added value. It has also opened new opportunities to extend the application of the IEC 61850 standard to additional domains such as Automated Warehouses or Press Machines.

6.5 WoT based model

The second subcontribution in this chapter changes the base specification from the IEC 61850 standard to the W3C's WoT for creating a new tool. The objective of the WoT is to support IoT interoperability using already existing and open standards, and research is being done on its usability on industrial environments, e.g., [206, 179]. In this subcontribution, mostly the TD document presented in Section 2.3 has been used. This document defines the information model and a JSON-LD serialization to describe *Thing*s and their metadata. For that, it uses a formal interaction model and domain specific vocabularies in order to describe Thing interactions and to enable semantic interoperability. This way, it links different machine-understandable definitions and

defines the protocol binding and serialization formats. The specification of the TD is still a work in progress and the work carried out in this subcontribution has two iterations. The first one is based on the Working Draft of October 21, 2018[1], while for the second one, the TD version has been updated to the Candidate Recommendation[2]. Note that this contribution does not focus on security, hence, the security parts of the model have been left out.

6.5.1 First Iteration

The tool presented in this subcontribution is still a work in progress, and two iterations have been made so far. The first iteration has been to generate an abstract syntax, based on the Working Draft of October 21, 2018 of the WoT TD. This subsection presents the first iteration, with the explanation of the implementation and how to implement a WoT system using the tool. Also, an evaluation is provided.

6.5.1.1 Implementation

For the implementation, CoAP has been the selected IoT protocol, but the system is able to add additional protocols with minimal changes. After the review of the available implementations in Chapter 3, libcoap has been selected due to its completeness and versatility.

Different tools have been used in this contribution, and each of them generates an intermediate result that allows to generate the source code

[1] `https://www.w3.org/TR/2018/WD-wot-thing-description-20181021/`

[2] `https://www.w3.org/TR/2019/CR-wot-thing-description-20191106/`

of a WoT servient. The workflow with the different tools and results is presented in Figure 6.13

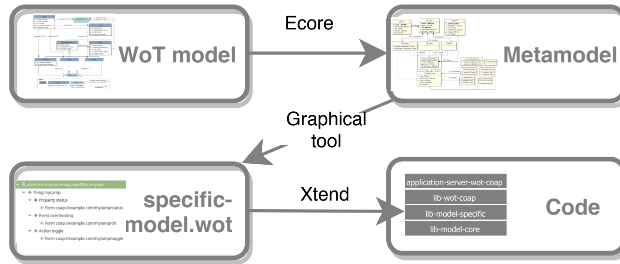


Figure 6.13: Workflow and the used tools, with intermediate results before the final code.

The steps that have been followed are similar as the ones to create TRILATERAL. The first step has been to create a generic metamodel. This metamodel is based on the WoT TD and using a graphical tree view editor, a user can enter a specific model. Finally, based on the specific model the source code of the servient is generated.

6.5.1.2 Generic Framework Implementation

In order to create a generic framework that allows to describe any model using the TD, the first step is to define a metamodel. To do that, Ecore has been used, which allows to create a TD based metamodel that specifies a DSL that can then be used to describe a specific model for a concrete use case. The user can describe the specific model using a tree view editor, which generates a file with the .wot extension. It is unique for each use case and depends on the device that is being modeled. A

subgroup of the classes of the TD draft available on October 21st, 2018[1] has been used, without including the security parts.

However, some minor modifications have been made to adapt the TD to the features offered by Ecore. First, the optional *label* attribute of the InteractionPattern class was made mandatory, and second, mandatory identifying attributes have been added to DataSchema (*did*), Form (*fid*) and Link (*lid*). These changes have been considered necessary in order to have identifiers for each instance of the classes so they can be used in the code. A hashed or random string could have been used for that, but this decision has been taken for easier code readability and increased code maintainability. Additionally, some attribute names had to be changed. On the one hand, *href* had to be changed to *_href* in the Form and Link classes as *href* is a reserved word in EMF. On the other hand, the *description* attribute of the InteractionPattern and DataSchema classes have been renamed to *idescription* and *sdescription* because Properties and Events are subclasses of both, and a class can not have different attributes with the same name.

The designed metamodel for this contribution is very similar to the WoT model. Figure 6.14 shows the classes and their attributes of the used model.

- **Thing** is the root element of the metamodel that describes a physical or virtual Thing. A Thing may have several Links and Interaction Patterns.

- **Link**s define relationships between resources.

[1] https://www.w3.org/TR/2018/WD-wot-thing-description-20181021/

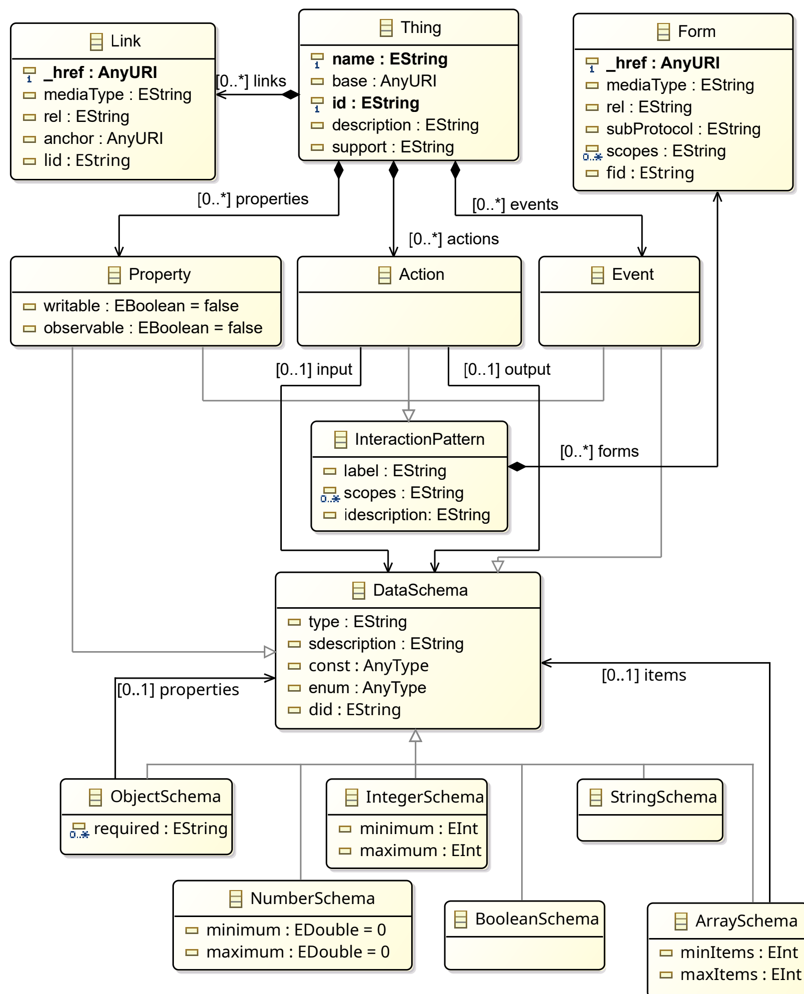


Figure 6.14: Metamodel based on the WoT specification on the Working Draft of October 21. DataSchema has several subclasses.

- **Interaction Pattern** groups the three subclasses that define the interaction methods of the Thing:

  - The **Property** class exposes the internal state of the Thing, in order to retrieve or change it.
  - **Action**s allow to manipulate the internal state of a Thing, but it enables more advanced options compared to a Property. It also allows to call functions of the Thing.
  - **Event**s communicate state transitions that are triggered by internal state changes.

  An Interaction Pattern may have several Forms.

- The **Form** class is used to indicate where to access a service.

- **DataSchema** class is a superclass that has several subclasses that define the data types. Property and Event are subclasses of DataSchema and Actions have input and output attributes that are also DataSchemas. The subclasses of DataSchema are **ObjectSchema**, **BooleanSchema**, **NumberSchema**, **IntegerSchema**, **ArraySchema** and **StringSchema**. The data types of the *const* and *enum* must be consistent with the subclass.

Using the DSL that this metamodel represents, a specific WoT object can be modeled using a graphical tree view editor that EMF offers. Figure 6.15 shows an example of a TD for a motor modeled to use with CoAP. The created model is saved in a .wot file.

After the specific model is defined, the next step is to generate the code. In order to do that, a code generator has been implemented using

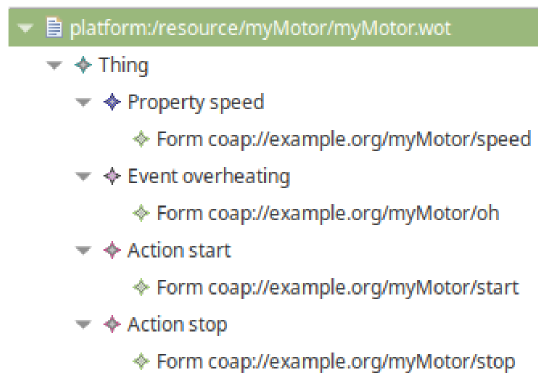


Figure 6.15: Graphical tool to generate the tree structure that allows to generate the .wot file.

Xtend. The code generator takes the .wot file as input, to get the instances of the specific classes in order to create the code implementation in C++. However, this part of the code just represents the model, it lacks of communication capabilities to the outside. A communication library (libcoap for CoAP communication in this case) has to be included and with that, the code generator also generates the code for the resources and functions, in order to complete the code to implement the WoT servient.

6.5.1.3 WoT system implementation

The final code for the servient is organized in four Eclipse projects, organized on a stack as shown in Figure 6.16.

The project on the base of the stack is `lib-model-core`. It is the implementation of the metamodel and includes its abstract classes, which are the classes represented in the TD. This way, it allows to upper layers to access the generic classes of the metamodel. This layer is common for all the use cases, i.e., it is reusable for every WoT servient.

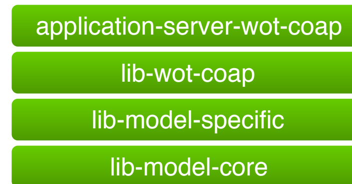


Figure 6.16: Different layers of the code projects.

The layer on top of `lib-model-core` is `lib-model-`*`specific`*. This project includes the specific classes of the TD of the modeled device, which are created based on the .wot file. As previously explained, a user creates the .wot file using the tree view editor, and with that, the Xtend code generator creates the classes and the code to generate the `lib-model-`*`specific`* project.

The next layer is `lib-wot-coap`, which integrates the specific model with the communication protocol library, in this case the lib-coap library for CoAP. The W3C defines the functions a WoT servient should have in the WoT Scripting API document[1]. More concretely, the WoT Scripting API describes a programming interface to represent the WoT servient. This document defines three interfaces, but in this contribution, only the *ConsumedThing* interface has been implemented. This interface describes a client API that allows to send requests to servers, invoke Actions, retrieve and update Properties, and observe Properties, Actions and Events. The other two interfaces are *WoT* and *ExposedThing*, where the former provides methods to discover, consume and expose a Thing and the latter allows to define the handlers, Properties, Actions, and Events of a Thing. For this contribution, a static system has been used as a first step, and these two interfaces allow to model dynamic system, thus, they have not been included.

[1]`https://www.w3.org/TR/wot-scripting-api/`

Table 6.5 shows a proposal to map the functions of the *ConsumedThing* interface to CoAP, indicating what CoAP verbs must be used for each of the functions of the interface. The request URI is defined in the InteractionPattern of each Action, Event or Property.

Function	Method
invokeAction	PUT
setProperty	PUT
getProperty	GET
addListener	GET + Obs
removeListener	GET
removeAllListeners	GET
observe	GET + Obs

Table 6.5: Functions for sending request to servers in order to retrieve or update Properties, invoke Actions, and observe Properties, Actions and Events.

The layered stack used for the projects allows to easily include additional communication protocols. In order to enable the use of different protocols, another library has to be implemented at the same layer as `lib-wot-coap`. Then, using the graphical tree view editor the desired protocol or protocols could be selected.

Finally, the higher project of the stack is the layer that interacts with the layers below and generates the executable file. This layer is not a library, it is the final executable that can be added to the device and expose the WoT servient to the outside, so other Things can interact with it.

With the workflow presented in this section, using the created tools, the source code for WoT servients can be generated in order to expose

them to WoT applications. However, the application and its logic needs to be implemented separately.

6.5.1.4 Evaluation

After describing the implementation process and how the code is generated based on a model, this section evaluates the implementation using the example of a motor of a machine on a factory. This example describes a motor with the TD, which is adapted in Listing 6.1 to the metamodel created in this contribution and to the CoAP protocol for the communication. The motor includes a property to represent its *speed*, two actions to *start* and *stop* the motor, and an *overheating* alarm using an event if the temperature of the motor increases to not safe values.

As described in Section 6.5.1.1, a graphic tool like the example in Figure 6.15 is used to model the Thing. The motor has been modeled and saved in a file called myMotor.wot, and the Xtend code generator has automatically created the source code with the required eclipse projects. In this example, the second lowermost layer of the stack in Figure 6.16 is called `lib-model-myMotor`. One small part of the source code that has been generated is shown in Figure 6.17, which includes the constructor methods of the `speed_PROPERTY`, `start_ACTION`, `stop_ACTION`, and `overheating_EVENT` classes, which extend their own generic class. The source code from the rest of the layers is common for every modeled Thing if the communication protocol does not change. But adding other communication protocols is possible thanks to the modular design of this solution.

Listing 6.1: Example TD used for the evaluation.

```
{
  "id": "urn:dev:wot:com:example:servient:motor",
  "name": "MyMotor",
  "properties": {
    "speed" : {
      "type": "dataschemaType_integer",
      "sdescription": "Status of myMotor",
      "did": "ds_speed",
      "writable": true
      "forms": [{
        "href": "coap://example.com/myMotor/speed",
        "fid": "speed"}]
    }
  },
  "actions": {
    "start" : {
      "idescription": "Start the motor",
      "forms": [{
        "href": "coap://example.com/myMotor/start",
        "fid": "start"}]
    }
    "stop" : {
      "idescription": "Stop the motor",
      "forms": [{
        "href": "coap://example.com/myMotor/stop",
        "fid": "stop"}]
    }
  },
  "events":{
    "overheating":{
      "type": "dataschemaType_boolean",
      "idescription": "Overheating alert!",
      "did": "ds_oh",
      "forms": [{
        "href": "coap://example.com/myMotor/oh",
        "fid": "oh"}]
    }
  }
}
```

```
namespace PROPERTY {
    class speed_PROPERTY : public core_model::PROPERTY::Property{

        public:
        speed_PROPERTY(string label, string scopes, string idescription, Form* forms, dataschemaType type,
                string sdescription, bool dconst, bool* denum, bool writable, bool observable):
            PROPERTY("speed", "[]", "", forms, dataschemaType_integer, "Speed of myMotor",
                    NULL, NULL , true, false){};
    };
}

namespace EVENT {
    class overheating_EVENT : public core_model::EVENT::Event{

        public:

        overheating_EVENT(string label, string scopes, string idescription, Form* forms, dataschemaType type,
                string sdescription, bool dconst, bool* denum):
            EVENT("overheating", "[]", "Overheating alert!", forms, dataschemaType_boolean, "", NULL, NULL){};
    };
}

namespace ACTION {
    class start_ACTION : public core_model::EVENT::Event{

        public:

        start_ACTION(string label, string scopes, string idescription, Form* forms,
                DataSchema<T> input, DataSchema<T> output):
            ACTION("start", "[]", "Start the motor", forms, input, output){};
    };
}

namespace ACTION {
    class stop_ACTION : public core_model::EVENT::Event{

        public:

        stop_ACTION(string label, string scopes, string idescription, Form* forms,
                DataSchema<T> input, DataSchema<T> output):
            ACTION("stop", "[]", "Stop the motor", forms, input, output){};

    };
}
```

Figure 6.17: Small part of the generated code as an example.

6.5.2 Second iteration

Building on the work of the first iteration, the second iteration has two main improvements. On the one hand, the base WoT TD version has been updated to the newest one, although still not being the final version, it is the Candidate Recommendation[1]. On the other hand the tool has been updated to support a concrete syntax so a regular JSON-LD with a WoT TD can be supported. To be able to integrate the abstract syntax defined in the previous subsections and the concrete abstract, a Model-to-Model (M2M) transformation has also been defined, which is able to transform the concrete syntax into the abstract syntax. This way, the tool now supports two ways of inputting a model, i.e., using the previously presented tree view tool and a text based JSON-LD TD.

In this subsection, the abstract and concrete syntax are explained, along with the process of generating the source code. The process of developing a WoT servient with the second version of the tool is also presented.

6.5.2.1 Abstract syntax

The first step for the second iteration of the WoT code generator tool has been to update the Ecore metamodel presented in Subsection 6.5.1 to the Candidate Recommendation of November 6th, 2019. As explained in that subsection, instantiating the concepts of the metamodel, a specific model for a system can be created. Both versions are similar, but they have some differences. In addition, the security features are also modeled, although their code is not generated in this version. In this subsection, the model is represented in four figures. Figure. 6.18 shows

[1] `https://www.w3.org/TR/2019/CR-wot-thing-description-20191106/`

the core classes of the WoT TD, Figure. 6.19 the classes related to the JSON schema, Figure. 6.20 the classes for security, and finally, Figure. 6.21 presents the Hypermedia controls vocabulary.

Regarding core classes, see Figure. 6.18, some of them have been changed. The main differences from the previous version are:

- **SecuritySchemes** class has been included.
- **VersionInfo** and **Multilanguage** classes have been added.
- InteractionPatterns have been renamed to **InteractionAffordances**, including its subclasses, i.e., **PropertyAffordance**, **ActionAffordance**, and **EventAffordance**.

In addition, there are also some small changes in the attributes or the relationships between the classes. Furthermore, some minor modifications have been introduced due to some technical limitation of the underlying technology:

- The *href* attributes in Forms and Links had to be renamed to *_href* because href is a reserved word in XML (the serialization format of EMF models). However, this renaming does not have any effect since specific *getter* and *setter* methods have been defined maintaining the public API of the metamodel compliant with the WoT TD specification.
- Mandatory *id* attributes have been specified for the InteractionAffordance and SecurityScheme classes, since they are indeed specified in the JSON tuple to define such elements.

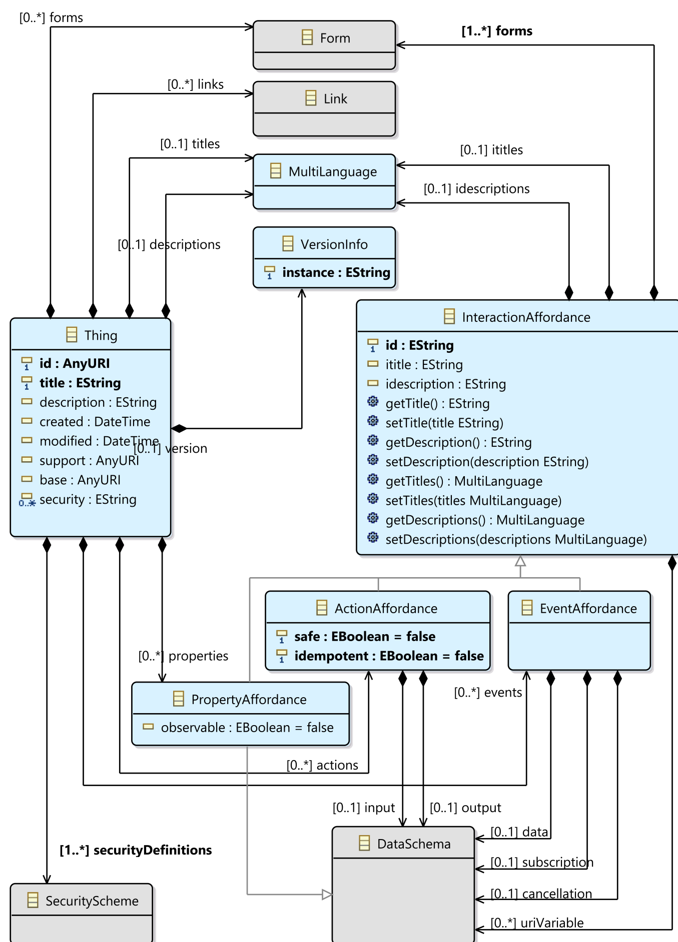


Figure 6.18: EMF implementation of the TD Core vocabulary model.

- The *title* and *description* attributes on InteractionAffordance and DataSchema classes have been renamed to *ititle*, *dtitle*, *idescription*, and *ddescription* to avoid collisions, since EMF – although it supports multiple inheritance – does not allow attributes redefinition nor collision. Similarly to the *href* case, specific *getter* and *setter* methods have been defined so that the public API of the metamodel complies with the WoT TD specification.

In the classes related to the DataSchema (see Figure. 6.19), a new relationship appears with the MultiLanguage class. Apart from that, a new subclass has also been included, i.e., **NullSchema**. This way, the different types of data represented with DatasSchemas are **ArraySchema**, **ObjectSchema**, **StringSchema**, **BooleanSchema**, **NumberSchema**, **IntegerSchema**, and **NullSchema**.

The classes related to the security (see Figure. 6.20) are grouped on the **SecurityScheme** superclass, which has several subclasses and it also may have a description in multiple languages with the **MultiLanguage** class. The subclasses of SecurityScheme enable different security features and include the following subclasses: **BasicSecurityScheme**, **NoSecurityScheme**, **CertSecurityScheme**, **BearerSecurityScheme**, **PoPSecurityScheme**, **DigestSecurityScheme**, **APIKeySecurityScheme**, **PSKSecurityScheme**, **PublicSecurityScheme**, and **OAuth2SecurityScheme**.

Finally, some additional classes are shown in Figure. 6.21, which define the classes for the Hypermedia control of the WoT TD. These classes are the already described **Link** and **Form**, and the new **ExpectedResponse**, which describes the expected response message.

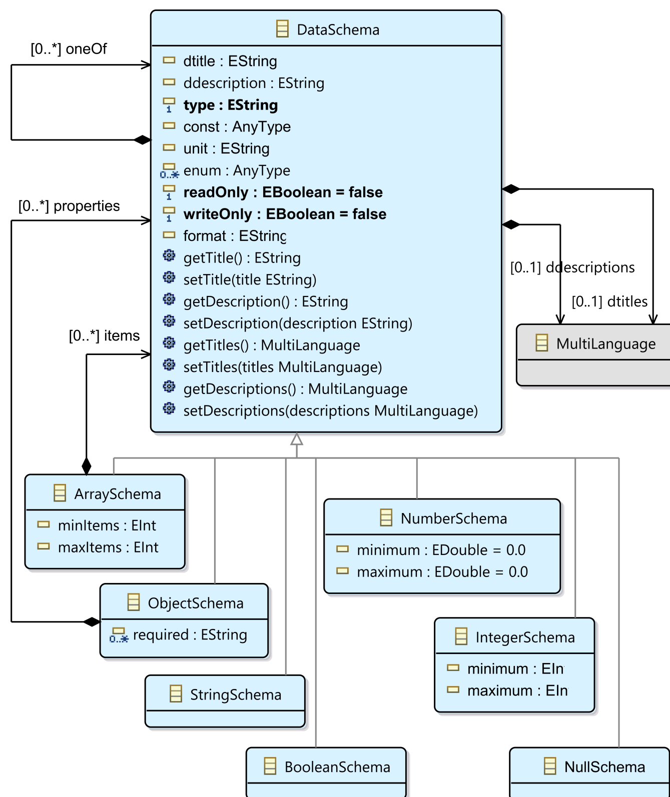


Figure 6.19: EMF implementation of the TD Data Schema vocabulary model.

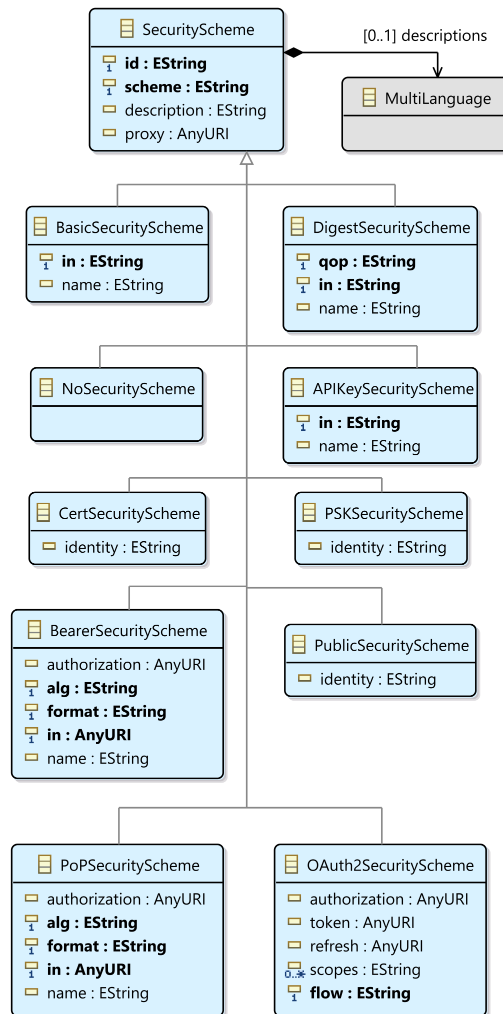


Figure 6.20: EMF implementation of the TD WoT security vocabulary model.

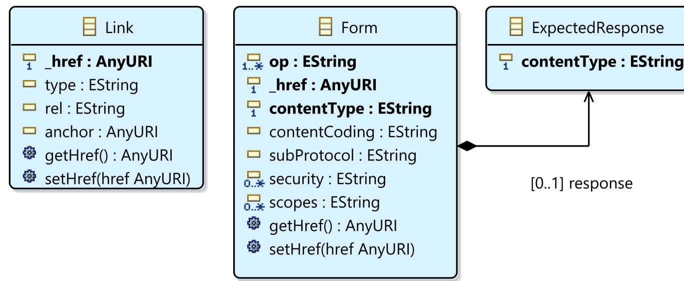


Figure 6.21: EMF implementation of the TD Hypermedia controls vocabulary model.

6.5.2.2 Concrete syntax

The next step for the second iteration of the WoT tool has been to create a concrete syntax, extending the tool from the first iteration. The concrete syntax allows to model the WoT servient using a JSON-based representation, as specified by the TD. This way, a TD of a WoT servient described in JSON that complies with the TD document can be supported by the tool. To be able to support a concrete syntax, Xtext has been used also, which allows to develop textual domain-specific languages. Xtext takes a BNF-like grammar describing the domain-specific language as input and generates an EMF-based metamodel. Not only that, but Xtext also generates the necessary tools to validate, parse, and reify as instances of the equivalent metamodel textual instances conformant to the grammar, thus allowing to exploit all the modeling tools available in the Eclipse ecosystem.

Listing 6.2 shows an extract of the grammar that has been defined to support the WoT TD definitions described in JSON. It shows the most relevant parts of the rules to define *ThingDescription* objects, *VersionInfo* objects, and *Form* objects in JSON.

Listing 6.2: Extract of the Xtext grammar.

```
JsonThingDescription:
  {JsonThingDescription}
  '{'
  (
    // Example of JSON-LD property
    // "@context" omitted from brevity
      ( '"@type"' ':' ( ld_type+=AnyString | '[' ld_type+=AnyString (
          ',' ld_type+=AnyString)* ']' ) ','? )?
    // Example of single-valued, primitive properties
    // "description", "created", "modified", "support" and "base"
        omitted for brevity
    & ( '"id"' ':' id=AnyString ','? )?
    & ( '"title"' ':' title=AnyString ','? )?
    // Example of single-valued or multivalued property
    & ( '"security"' ':' ( security+=AnyString | '[' security+=
        AnyString (',' security+=AnyString)* ']' ) ','? )?
    // Example of single-valued, complex property
    & ( '"version"' ':' version=JsonVersionInfo ','? )?
    // Examples of multivalued, complex property
    // "properties", "actions", "events", "securityDefinitions",
    // "titles" and "descriptions" omitted for brevity
    & ( '"forms"' ':' '{' forms+=JsonForm (',' forms+=JsonForm)* '}'
        ','? )?
    & ( '"links"' ':' '{' links+=JsonLink (',' links+=JsonLink)* '}'
        ','? )?
  )
  '}';
JsonVersionInfo:
  {JsonVersionInfo}
  '{'
    '"instance"' ':' instance=AnyString
  '}';
JsonForm:
  {JsonForm}
  '{' (
      ( '"op"' ':' ( op+=AnyString | '[' op+=AnyString (',' op+=
          AnyString)* ']' ) ','? )?
    & ( '"contentType"' ':' contentType=AnyString ','? )?
    // ...
  ) '}';
// ...
```

Figure 6.22 shows the Ecore metamodel generated from the grammar. The instances that conform the metamodel are automatically generated based on the text based JSON descriptions thanks to the tools generated by Xtext. It can be observed that the classes on the metamodel are the classes in the grammar, and each class contains the attributes specified in the rule. That is, the elements in Figure 6.22 are parallel to the rules from Listing 6.2. Thanks to this automatically generated metamodel, other tools based on EMF can be used, e.g., model transformation engines. This way, the concrete syntax can be transformed to the abstract syntax presented in the previous subsection, thus, the source code can automatically generated from the concrete syntax, using a Model-to-Model (M2M) transformation as a middle step.

Analyzing Figure 6.22, it can be seen that the concepts are similar to the ones in Figures 6.18–6.21. Thus, transforming the instances of the concrete syntax metamodel to the abstract syntax metamodel is simple. This M2M transformation has two main advantages. On the one hand, the abstract syntax metamodel perfectly matches the concepts of the WoT TD, abstracting it from any details regarding implementation of serialization. On the other hand, it allows to use the parts of the first iteration of the tool to generate the code once the abstract syntax model is specified, without the need of considering the concrete syntax details.

6.5.2.3 **From Concrete Syntax to Abstract Syntax**

With both the concrete and abstract syntax specified, the code can be generated based on a WoT TD JSON or using the graphical tool presented in the first iteration section. To be able to use the concrete syntax, the first step is to transform it to the abstract syntax. This

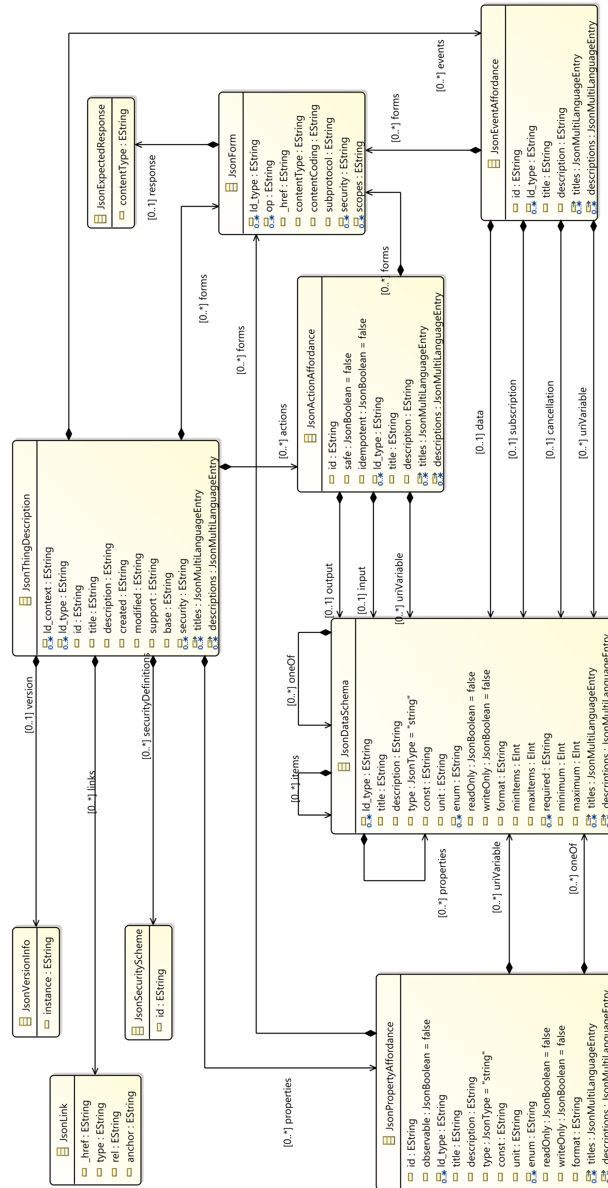


Figure 6.22: Metamodel derived from the concrete syntax for the WoT TD specification.

way, it does not matter how the servient has been inputted in the tool. Listing 6.3 shows an extract of the M2M transformation. It can be noted that the mapping between the two metamodels is almost one-to-one.

Once the model has been transformed, the abstract model can be transformed into text using M2T transformation. This text is the source code of a servient. The M2T transformation is the same as the one explained in subsection 6.5.1, but it has been updated with the changes applied to the abstract syntax. CoAP has been selected has the communication protocol with libcoap as the CoAP implementation, but other communications protocols could be added with minimal changes. Listing 6.4 shows a small part of the template for generating the C++ source code based on the abstract syntax.

6.5.2.4 **Developing a WoT servient using the WoT Toolkit**

The different elements that have been explained in the previous subsections can be combined to developing WoT servients. This subsection describes the process of developing the source code to using the WoT tool, which has been named *WoT Toolkit*.

Figure 6.23 presents the different steps that to develop a WoT servient, with the main involved artifacts (models or files), represented with the shape of a paper sheet.

The artifacts that have a solid line are the ones that have been manually created, while the ones with discontinuous are automatically generated thanks to different types of transformations. In addition, there are two layers in the artifacts, i.e., M1 and M2. The former groups the artifacts that are defined or created for each time a new development process is enacted, while the latter groups the ones that have been developed to create the toolkit.

Listing 6.3: Extract of the M2M transformation from the concrete syntax metamodel to the abstract syntax metamodel.

```
static def Thing toThing(JsonThingDescription jtd) {
  return TdFactory.eINSTANCE.createThing => [
    id = jtd.id
    title = jtd.title
    description = jtd.description
    created = jtd.created?.toXMLGregorianCalendar
    modified = jtd.modified?.toXMLGregorianCalendar
    support = jtd.support
    base = jtd.base
    titles = jtd.titles?.toMultiLanguage
    descriptions = jtd.descriptions?.toMultiLanguage
    version = jtd.version?.toVersionInfo
    security.addAll(jtd.security)
    forms.addAll(jtd.forms.map[toForm])
    links.addAll(jtd.links.map[toLink])
    securityDefinitions.addAll(
      jtd.securityDefinitions.map[toSecurityScheme])
    properties.addAll(
      jtd.properties.map[toPropertyAffordance])
    actions.addAll(
      jtd.actions.map[toActionAffordance])
    events.addAll(
      jtd.events.map[toEventAffordance])
  ]
}

static def Link toLink(JsonLink jl) {
  return TdFactory.eINSTANCE.createLink => [
    href = jl._href
    type = jl.type
    rel = jl.rel
    anchor = jl.anchor
  ]
}
```

Listing 6.4: Extract of the code generation template.

```
1 static def property_h(PropertyAffordance p) '''
2   #ifndef «p.c_id»_PROPERTY_DEFINED
3   #define «p.c_id»_PROPERTY_DEFINED
4
5     // Includes omitted for brevity purposes
6     // Using namespaces omitted for brevity purposes
7
8   namespace PROPERTY_AFFORDANCE {
9     class «p.c_id»_PROPERTY_AFFORDANCE : public core_model::
          PROPERTY_AFFORDANCE::PropertyAffordance {
10
11      public:
12
13      «IF p.type == "dataschemaType_boolean"»
14            // Code omitted for brevity purposes...
15      «ELSEIF p.type == "dataschemaType_string"»
16        // StringSchema
17        «p.c_id»_PROPERTY_AFFORDANCE(string id, string title, string
            description, Form* forms, dataschemaType type, string
            d_const, string* d_enum, bool writable, bool observable):
18        PROPERTY_AFFORDANCE(id, title, description, forms, type,
            d_const, d_enum, writable, observable)
19        {};
20            // Code omitted for brevity purposes...
21            «ELSE»
22            // Code omitted for brevity purposes...
23      «ENDIF»
24            // Code omitted for brevity purposes...
25
26      virtual ~«p.c_id»_PROPERTY_AFFORDANCE();
27    };
28  }
29  #endif // «p.c_id»_PROPERTY_AFFORDANCE_DEFINED
30 '''
31
32 static def c_id(PropertyAffordance pa) {
33   return pa.id.sanitize()
34 }
```

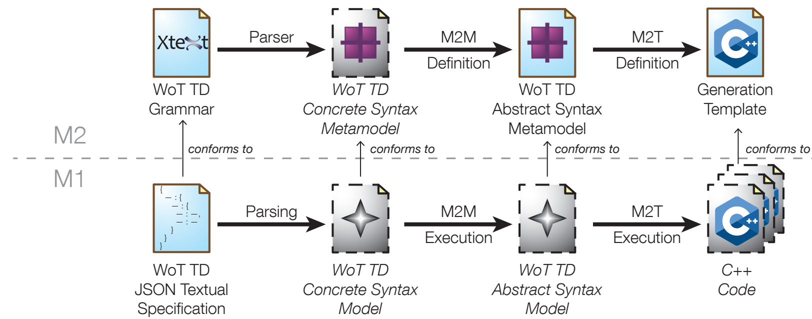


Figure 6.23: Development process of a WoT Servient using the WoT tool.

The development process happens at the M1 layer and the M2 layer is used to support the development process. The toolkit includes a JSON editor, which can be used to describe a servient TD (M1 layer, leftmost element). This editor is derived from the WoT TD Grammar (M2 layer, leftmost element), which is defined using Xtext, as described in subsection 6.5.2.2. The JSON editor provides code completion and validation to assist the JSON description process.

The inputted JSON, i.e., the *WoT TD JSON Textual Specification*, is automatically parsed and a *WoT TD Concrete Syntax Model* (second element on M1) is generated automatically, which complies with the *WoT TD Concrete Syntax Metamodel*.

When the JSON is saved on disk, the M2M transformation is automatically executed to transform the *WoT TD Concrete Syntax Model* into the corresponding *WoT TD Abstract Syntax Model* (M1, third element). The generated abstract model also conforms to the *WoT TD Abstract Syntax Metamodel* described in subsection 6.5.2.1. The *WoT TD Abstract Syntax Metamodel* is the starting point of the first iteration of the WoT tool described in subsection 6.5.1. The second iteration

also supports to create the *WoT TD Abstract Syntax Model* using the tree view editor. From this point on, the two iterations follow the same workflow.

The next step is to transform the *WoT TD Abstract Syntax Model* into the *C++* (last element on M1 layer) source code. This process launches when saving the inputted model, either when the JSON editor or the tree view editor is used. The M2T transformation takes the *WoT TD Abstract Syntax Model* as input and transforms it into text, i.e., C++ source code. As explained in subsection 6.5.1, the source code is the second lowermost layer in Figure 6.16, i.e., *lib-model-specific*.

As it has been previously mentioned, the elements with discontinuous lines in Figure 6.23 are automatically created. This means that in the workflow for developing the source code for a servient, from a developer point of view (M1 layer), only the *JSON Textual Specification* has to be defined when using a text based TD, or the *WoT TD Abstract Syntax Model* if the tree view editor is used. The *WoT TD Grammar*, the *WoT TD Abstract Syntax Metamodel*, and the *Generation Template* had to be manually created, but they are common for all projects, hence, they are not needed to be developed for every project. This way, the implementation effort is greatly reduced.

6.6 Conclusion

After describing the two subcontributions of this chapter, this section provides some conclusions. To do that, first some lessons that have been learned during the development and evaluation of TRILATERAL are presented.

- Decreased development and deployment time.

The time to develop and deploy an artifact has decreased drastically from previous similar projects. One of the reasons for this is the use of a generic model, which makes the design of an adhoc model not necessary, as it reuses the IEC 61850 metamodel. The other main reason is the automatic source code generation, which leads to a reduction of time and costs of developments as well as to an increase on the code quality. Using reusable parts also accelerates the validation process, due to the parts that have already been validated not needing to go through the testing process again. In addition, the tree view editor for inserting the specific model also ease the development process, reducing time and errors. The use case has to be modelled in any case, using TRILATERAL or developing the code manually. However, once the system is modelled, using TRILATERAL, the model can be implemented from scratch, which can take days or weeks depending on the model, while with TRILATERAL, the model can be described using the tree view editor in hours or days, and TRILATERAL generates the source code automatically.

- Usefulness in different domains.

  Although the IEC 61850 standard was designed for electrical substations, in this contribution it is also used in other domains, even though being energy related. However, the variability of the use cases is high, representing very diverse characteristics and communication patterns (device-to-cloud systems and device-to-device systems).

- IEC 61850 outside the electrical substations.

The validation process of the developed systems has also been improved because common blocks have been used in the different use cases. The same code that implements the IEC 61850 metamodel in the kernel has been used in all scenarios. Client and server projects for the main and reporting applications are also almost the same, they have minimal changes. Hence, the already tested and validated parts do not need to undergo a new process. This also helps maintain the code, because updates, bug corrections, etc. just need to be developed once and deploy the same patches on all systems. Furthermore, the usefulness of IEC 61850 outside of electrical substations has been proved, and using it in further projects with domains not related to energy, such as manufacturing and transport, can also be expected to be successful.

- SPL and MDE benefits on Industry 4.0.

  Finally, the last learned lesson is that TRILATERAL proves that SPL and MDE are beneficial for the development of the Industry 4.0. Different industrial domains share many commonalities, such as requirements for monitoring and controlling the systems. Even though the development of a tool like TRILATERAL is complex and costly for a single project, the benefits come on the long term, when it is used to develop more use cases. Thanks to the DSL and the tree view editor, joining SPL, MDE techniques and a DSL the development times are decreased considerably, helping the adoption of Industry 4.0 solutions.

Some of the presented learned lessons have established the basis to continue the work using the WoT TD instead of IEC 61850. Taking

both subcontributions into consideration, some common conclusions can be provided.

The first conclusion is that creating a tool that joins MDE and SPL techniques with a DSL to allow to develop the source code for artifact in the case of TRILATERAL or servients for the WoT metamodel tool is possible. With these tools, a tree view graphical editor can be used to model the required system using already established standards, i.e., IEC 61850 in the first subcontribution and the WoT on the second. Thanks to these tools, the development and implementation time and hence, costs are decreased. In addition, the use of standards has many benefits, such as cost reduction from design to operation and maintenance, and greater flexibility by enabling interoperability. Figure 6.24 describes the steps that have been taken in order to create both tools, which now can be used to create the code of an ICPS or a WoT servient automatically.

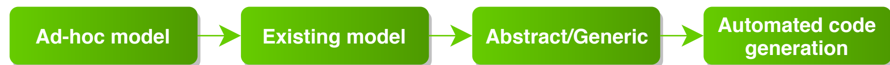


Figure 6.24: Different phases on the development of projects.

Before creating TRILATERAL and the WoT model based framework, CPSs had to be modeled using a specific model created ad-hoc just for a specific use case or domain. This approach requires to create a model from scratch every time, and then implemented that model in order to deploy the devices.

Creating a model for every single scenario is costly, hence reusing a model for different environment is beneficial. In this regard, a model for a concrete domain can be used, such as the IEC 61850 model, or a generic one that has been designed for diverse kinds of domains such as the WoT TD. This approach allows to transfer the gained knowledge

from one project to another, no matter the specific domain in the case of the WoT, and for the IEC 61850 model, in this contribution it has also been used for different domains from its original target.

Yet, using a common model for each scenario, the only transferred element from one project to the next is the knowledge of the model. Creating an abstract generic metamodel and using MDE techniques has the advantage of reusing a big part of the code, reducing development and testing costs. In the case of this contribution, the bottom and the two upper layers of the presented stacks for both tools can be reused. Only the specific model has to be implemented manually, which has been named `lib-model-`*`specific`* in both cases. The lower layer, the one that includes the abstract classes of the metamodel is also common for every project that uses each tool.

This issue of having to manually create the `lib-model-`*`specific`* layer for each project has been solved using the automated code generation thanks to the EMF and Xtend. They allow to input the specific model using a tree view graphical tool. In addition, the case of the WoT toolkit also supports to input the TD in a textual manner, more specifically, using a JSON. After the framework is completed, creating the code for an artifact or a WoT servient is much faster. The artifact or the WoT servient has to be designed and modeled with a manual approach, but the implementation time has proven to be much less. However, further analysis requires more implementations in different projects and use cases, in order to have more data on development and time costs.

In the case of TRILATERAL, in addition to creating the model, adding a different communication protocol for each use case is also easy. As previously explained, TRILATERAL has been evaluated in three

industrial domains, i.e., Wind Farm, Smart Elevator and Catenary-free Tram, proving that it can be used on diverse scenarios with different requirements. Depending on the industrial needs, the most adequate communication protocol has to be selected, e.g., CoAP is lighter than the others, but it does not perform so well with large payloads as showed in Chapter 4. Additionally, some use cases might have more than one scenario (e.g., Catenary-free Tram) that make the use of more than one communication protocol fitting, which can benefit even further from an automated solution. The WoT toolkit is still in an earlier development phase and only supports CoAP communication, but the toolkit is prepared to include more protocols in the future.

After carrying out the evaluation of the Catenary-free Tram use case, it has been proved that the development time has been reduced. The main disadvantage of TRILATERAL is that the drivers are not generated automatically and the deployment is manual. This makes the updates easy to be generated, but the deployment is not that optimized yet. A similar issue is presented also in the WoT TD tool, where the application logic has to also be manually implemented for each use case.

Summarizing, TRILATERAL has allowed Ikerlan to improve the engineering process of ICPS development, reducing time and monetary costs while improving the code quality and validation and maintenance processes. With these advantages in mind, a similar approach has been followed for the second subcontribution, using the WoT instead of IEC 61850 for applying a more general use specification.

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

Winston Churchill

To infinity and beyond!

Buzz Lightyear

Conclusions and Future Work

After explaining all the contributions carried out during this PhD thesis, this sections concludes this dissertation. In order to do that, on the one hand, some conclusions are provided, both general conclusions for the thesis and some individual conclusions of each contribution. Then, on the other hand, some lines for future work are provided, also general ones for the thesis and individuals for each contribution.

Contents

7.1 Conclusions 258

7.1.1 C1: Analysis of IoT 259

7.1.1.1 C1.1: Analysis of IoT communication protocols 259

7.1.1.2 C1.2: Analysis of CoAP 260
7.1.2 C2: Interoperability though IEC 61850 262
7.1.2.1 C2.1: Mapping IEC 61850 to CoAP . 262
7.1.2.2 C2.2: Implementation and comparison of the mapping 263
7.1.3 C3: CoAP enhancement proposal 264
7.1.4 C4: Apply Software Engineering techniques in IoT environments 265
7.1.4.1 C4.1: IEC 61850 based metamodel . 266
7.1.4.2 C4.2: WoT based metamodel 266
7.1.5 General conclusions of the thesis 268
7.2 Future Work 270
7.2.1 C1: Analysis of IoT 271
7.2.1.1 C1.1: Comparison of IoT protocols . 271
7.2.1.2 C1.2: Comparison of CoAP implementations 272
7.2.2 C2: Interoperability though IEC 61850 273
7.2.3 C3: CoAP enhancement proposal 275
7.2.4 C4: Apply Software Engineering techniques in IoT environments 275
7.2.4.1 C4.1: IEC 61850 based metamodel . 275
7.2.4.2 C4.2: Interoperability through WoT . 276
7.2.5 Future work of the thesis 278

7.1 Conclusions

In order to present the conclusions of the work carried out during this thesis, first the conclusions of each contribution are provided. Table 7.1 summarizes the contributions, indicating the chapter where it is

explained, the research question it addresses and the achieved publications.

Contribution	Chapter	RQ	Publication(s)
C1.1	Chapter 3	RQ1.1, RQ1.2	[91]
C1.2	Chapter 3	RQ1.3	[92], [93]
C2.1	Chapter 4	RQ2.1	[94], [85]
C2.2	Chapter 4	RQ2.2	[87], [85]
C3	Chapter 5	RQ3.1, RQ3.2	[86], [85]
C4.1	Chapter 6	RQ4.1	[81], [90]
C4.2	Chapter 6	RQ4.2	[89], [88]

Table 7.1: Summary of the contributions, on which chapter they are explained, which RQ they address and the related published articles.

7.1.1 C1: Analysis of IoT

The first contribution defined in Chapter 1 is the analysis of the state of the IoT. In this regard, the analysis has been divided in two subcontributions, one to analyze two IoT communication protocols and another one to analyze the available libraries for the selected protocol.

7.1.1.1 C1.1: Analysis of IoT communication protocols

The first subcontribution is the analysis of different IoT communication protocols. In this subcontribution, the overhead, scalability and latency of MQTT and CoAP have been analyzed, using different QoS and running on a STM32F407VGT6 Discovery board, which includes an ARM Cortex-M4 microprocessor, which, as explained in Section 2.2, is widely used in Industry. The experiments have been carried out with a lossless network with no congestion, and according to the experiments,

CoAP performs better, requiring fewer bytes to transmit the same payload and has a lower delay, for all QoS cases.

Even though the difference in terms of overhead is not big, specially in one-to-many scenarios, the contrast in latency is bigger. MQTT transmissions have a latency of the millisecond order, while CoAP needs hundreds of microseconds. The advantage of CoAP is because of the underlying UDP, but this can not be ignored because the use of TCP and UDP is part of the entire available protocol stack. Hence, the selected protocol to continue the work has been CoAP.

Although CoAP has proven to be more lightweight and faster than MQTT, the latter may also be a good choice if the overhead and latency requirements are met in a project. MQTT is more mature, it has easier configuration and its publish-subscribe communication paradigms offers decoupling between clients, which makes it a good choice for some domains, especially monitoring use cases.

7.1.1.2 **C1.2: Analysis of CoAP**

Based on the conclusions of the first subcontribution, CoAP has been selected as the communication protocol for continuing the work of this thesis. The second subcontribution compares nine implementations with their features, behavior and performance. Four C (i.e., libcoap, smcp, microcoap, and FreeCoAP), one Java (Californium), two Python (CoAPthon and CoAPy), and another two Node.js (node-coap, h5.coap) libraries are compared. Security features have also been analyzed theoretically, as the cryptographic features using DTLS are still under development.

All the analyzed implementations follow the final specification of the CoAP standard with the exception of CoAPy, which is based on an

outdated draft. This make all the libraries but CoAPy compatible with each other.

Analyzing the performance of the servers, the four C implementations prevail. However, they can be split in two groups, faster libraries (libcoap and smcp) and the ones that leverage lower memory requirements (microcoap and FreeCoAP), which need one order of magnitude less memory. In addition, microcoap and FreeCoAP have other downsides, as microcoap only implements parts of the RFC 7252 and FreeCoAP does not provide response code generation and URIs handling transparently.

Regarding the clients, they can have lower disk space requirements, Java, Node.js and Python libraries perform surprisingly close to libcoap and smcp in terms of latency. Moreover, thanks to the higher abstraction level of their languages, Californium (Java), CoAPthon (Python), h5.coap and node-coap (Node.js) may also have their place in some projects that implement CoAP clients.

Considering the security libraries combined with the analyzed CoAP implementations, OpenSSL, GnuTLS, and mbed TLS are more mature than the rest as they are more general libraries that also include TLS support. However, OpenSSL and GnuTLS target bigger devices, so for resource constrained devices, tinydtls and mbed TLS might be a better choice. Still, in order to provide a good comparison between DTLS libraries, the implementations have to be more mature, as they still are not complete, or they are not easy to integrate with the CoAP libraries.

Summarizing, it can be concluded that all the analyzed CoAP libraries but CoAPy might be appropriate depending on the use case. More heavy implementations can be used in backbone systems as client because they are fast and the backbone systems usually do not have

a lack of resources, while the lightweight libraries can be deployed on smaller devices. The main goal of this subcontribution has been to provide information so a system designer can choose the library that adapts best to their system, reviewing the strengths and weaknesses of each of them.

7.1.2 C2: Interoperability though IEC 61850

After analyzing the CoAP libraries, the next contribution has been to achieve interoperable communications. To do that, the IEC 61850 standard has been selected, which provides an information model and an interface for communicating with the information model. This contribution is divided in two subcontributions, where the first one proposes a mapping of the IEC 61850 functions to CoAP and the second one implements the mapping and compares the performance to other approaches using HTTP and SOAP.

7.1.2.1 C2.1: Mapping IEC 61850 to CoAP

With the objective to provide a mapping of IEC 61850 to CoAP, different approaches that use diverse communication protocols have been surveyed. However, as explained in Chapter 4, they have some downsides, such as using legacy or heavyweight protocols, using powerful devices in terms of RAM, CPU, etc, not mapping many functions or in the case of the previous CoAP approach, not following the RESTful architectural style.

With this subcontribution, it has been proved that it is possible to map the functions of the IEC 61850 standard to a lightweight IoT protocol such as CoAP. The mapping of some of the ACSI services are

straightforward (e.g., basic services), but others are more complex (e.g., reporting and eventing services)

On the one hand, most of the services from IEC 61850 fit best with a client-server communication, and they can be used directly with CoAP, using URIs and different methods (GET, PUT, POST and DELETE) in order to map the functions. On the other hand, the rest of the services fit better with a publish-subscribe model, which can be achieved with the Observe extension for CoAP. However, additional steps are needed as a client needs to subscribe to the resources due to some limitations in the subscription mechanisms of the Observe extension. This issue is addressed in contribution C4.

7.1.2.2 **C2.2: Implementation and comparison of the mapping**

After proposing a mapping, the next subcontribution has been to implement that mapping, in order to validate and compare its performance to other approaches using HTTP and SOAP as the communication protocols.

The first thing this contribution demonstrates is that it is possible to integrate the IEC 61850 in the IoT using CoAP. With the second iteration, the resource representation to use in CoAP's implementation has been changed from JSON to CBOR, which allows to complete the IoT stack, using UDP, CoAP and CBOR.

With the first iteration, the CoAP mapping has been compared to the HTTP and SOAP implementations in terms of overhead and latency. CoAP requires fewer bytes in the network than HTTP and SOAP when the block-wise transfer is not used. Besides, it also has lower latency times. However, the block-wise transfer penalizes CoAP's performance

when sending big payloads, which can be even slower than HTTP and SOAP when using many blocks.

In addition, the second iteration proves that changing the resource representation to CBOR makes the resource representation to have fewer bytes than JSON and XML, benefiting the CoAP implementation even more.

Summarizing, it can be concluded that CoAP's mapping is more effective for most functions. Thanks to the Observe extension, CoAP proves to fit better than the other analyzed client-server protocols, as it provides ways to notify clients the way some functions need. However, the subscription mechanism of the Observe extension has some limitations which are addressed in C4, and the block-wise transfer extension can be very penalizing with big payloads.

7.1.3 **C3: CoAP enhancement proposal**

The mapping of the IEC 61850 has surfaced some limitations of the subscription mechanisms of CoAP's Observe. In order to overcome those limitations, this contribution proposes new CoAP options and response codes. Three requirements that describe different use cases have been considered for the new option and response codes.

The first requirement has been to be able to subscribe to a resource while creating or updating it, reducing the needed messages from four (PUT or POST, response, subscription request and subscription response) to just two (request and response).

The second considered requirements has been to receive a lightweight response to a subscription request. Sometimes, the current state of a resource might not be of interest to a client when it subscribes to a resource. In the case of the IEC 61850 mapping, this is applied to

the reports, which might not even exist when a client configures their creation and notification. Even more, the reports can be big, which as explained in the previous contribution, it can affect negatively because of the block-wise transfer extension.

The last requirement is the ability to create, update or get a resource while subscribing to a related one. One example for this is a configuration resource that can be polled, updated or created while subscribing to the main resource.

The proposed CoAP option and response codes have been first evaluated in an industrial alarm clock example, providing an analysis of the exchanged bytes. After that, they have been included in the IEC 61850 mapping, achieving a huge reduction of the exchanged bytes. In addition to achieving a reduction on the bandwidth usage, in the case of the IEC 61850 standard, the new option and response codes allow a more natural interaction between servers and clients.

The proposed enhancements are focused for the subscription stages, hence, they are more optimal in dynamic environments or mobile devices, where devices come in and out of the network frequently, such as sleeping nodes in small, resource constrained devices, or moving devices.

7.1.4 **C4: Apply Software Engineering techniques in IoT environments**

After proposing new subscription mechanisms for CoAP, the next contribution has been to accelerate the development of systems using software engineering techniques. To be able to do that, two different specifications have been selected, on the one hand, the IEC 61850 used in previous contributions, and on the other hand, a general use

specification, the WoT. Two subcontributions have been made in this contribution, one for each selected specification.

7.1.4.1 **C4.1: IEC 61850 based metamodel**

For the C4.1 subcontribution, a tool named TRILATERAL has been used, which uses Software Engineering techniques in order to generate the source code for ICPSs based on a model. This subcontribution explains TRILATERAL.

TRILATERAL is an MDE SPL solution that uses an IEC 61850 based DSL to create a metamodel. It allows to select between three communication protocols, i.e., HTTP, SOAP and CoAP. With all this parts, users can enter the model for an ICPS they need to implement using a tree view graphical editor. The tree view editor also allows to configure different parameters and the communication protocols, and automatically generates the source code for the artifact to be deployed on a ICPS.

In addition to explaining TRILATERAL, this contribution also describes the evaluation that has been carried out in different industrial domains, i.e., Wind Farm, Smart Elevator and Catenary-free Tram. During the development and deployment processes for the evaluation, several lessons have been learned, which are also explained in this subcontribution.

7.1.4.2 **C4.2: WoT based metamodel**

The second subcontribution of the last contribution of this thesis has been to change the model used for the code generation from the IEC 61850 standard to the WoT.

In order to do that, a new toolkit has been created. This new toolkit is a first approach so is still not as complete as TRILATERAL, but using a similar approach, a DSL has been created based on the WoT TD. Two iterations of development have been done, where the second expands the work of the first one. In the first one, a tree view editor that uses that DSL can be used to input the desired model based on the TD, using MDE techniques, the defined model is transformed into source code to be included on a WoT servient.

Since the first iteration was developed, the TD specification has been updated, so for the second one, the first step has been to update the TD version to the Candidate Recommendation version. In addition, a new concrete syntax has been defined for the WoT TD, along with a M2M transformation that transforms the concrete syntax into the updated abstract syntax. With this approach, the tools generated in the two toolkits are compatible, which allows two different ways to develop a WoT servient. On the one hand, the tree view editor can be used to describe the abstract syntax, and then the toolkit is able to transform the abstract syntax into the source code. On the other hand, a JSON with a TD compatible format can be inputted on the text based JSON editor. The toolkit transforms the JSON into a concrete syntax, and then the concrete syntax into the abstract syntax. This abstract syntax is the same as creating it using the tree view editor, so it can also be transformed into source code.

Although the development of a tool with these characteristics is costly at first, the code quality is increased, reducing the errors and detecting them earlier, leading to easier maintenance. In addition, new projects to develop WoT servients can be implemented using this tool, which accelerates the development process.

7.1.5 General conclusions of the thesis

After providing the conclusions for each contribution, this subsection provides the general conclusions of the thesis. The hypothesis formulated in Chapter 1 has three main pillars to be based on, i.e., lightweight communication protocols, industrial standards and software engineering techniques.

Regarding the lightweight communication protocols, as explained before, MQTT and CoAP have been compared in order to select one of them, answering RQ1.1. In this case, CoAP has been selected, as it fits better for controlling domains thanks to its client-server communication model, while MQTT would have been a better choice for monitoring use cases (RQ1.2). To answer RQ1.3, nine available CoAP libraries have been compared and the upsides and downsides of them have been explained in Chapter 3. The security libraries that use DTLS have also been analyzed theoretically, since some of them are still not fully implemented, and they are not easily integrated with the CoAP libraries.

One of the CoAP libraries has been selected for continuing the work of this thesis, which has been libcoap, as it has a good balance on offered features, hardware requirements and latency. On Chapter 4, CoAP has been used along an industrial standard, which has brought to light that CoAP has a very simple mechanism for a client to subscribe to get notifications. This simple mechanism has some limitations in specific use cases (RQ3.1). In order to overcome those limitations, some new CoAP response codes and options have been proposed, answering RQ3.2. With this solution, more advanced subscription features are available, expanding the overall capabilities of the subscription process in general and more concretely, it leads to a better fit for the industrial standard used in this thesis.

The second pillar of the work in this thesis is the use of industrial standards. In this case, two specifications have been used, the IEC 61850 standard and the WoT. IEC 61850 was designed for electrical substations but as explained in Chapter 4, it has also successfully been used in other domains. To answer RQ2.1, IEC 61850 has been mapped to CoAP and to later implement it. Then, the implementation has been compared to other approaches that use HTTP and SOAP as the communication protocols, with the result that CoAP is usually faster and more lightweight, with the exception on the cases that the payloads are big and the block-wise transfer extension is needed with many blocks (RQ2.2). Limitations have also been found in the subscription process, which have been solved as explained in the previous paragraph.

The other used specification is the Web of Things. It is not specifically an industrial standard, but it can be applied to industrial domains. The WoT has been used in Chapter 6, as the basis of a metamodel that allows to automatically generate the source code for WoT servients.

The last pillar of the hypothesis is software engineering, which provides some tools that help to achieve the adoption of Industry 4.0 accelerating the development and deployment process.

In this thesis, software engineering techniques have been first used with IEC 61850, allowing to create a tool that automatically generates source code, answering RQ4.1 as explained in Chapter 6. In order to build the tool, a DSL has been designed based on the IEC 61850 information model, which allows to generate a metamodel. Thanks to this metamodel, a visual tree view editor can be used to input a specific model for a project, and an MDE and SPL tool automatically generates the source code that includes the model and the communication func-

tions. The communication functions can use CoAP, HTTP or SOAP, which a user can select in the tree view editor.

To answer RQ4.2, and as explained in Chapter 6, the same approach has been followed. In this case, the designed DSL is based on the WoT TD and it has allowed to create a metamodel. Once again, a tree view editor can be used to input the specific model for each project, and the tool generates the source code automatically, using CoAP for the communication functions. Additionally, this toolkit is also capable of automatically transform a WoT TD JSON into the source code, using different steps and transformations.

Applying software engineering techniques makes the code quality to increase, as automatically generated code reduces manual errors. In addition, being the generated code modular, some parts are reused for every project. This makes the testing and validation process faster, because the reused parts are already tested and validated.

In summary, during this thesis the hypothesis has been proved using the RQ and the contributions, with lightweight communication protocols (e.g., CoAP), industrial standards (e.g., IEC 61850 and the WoT applied to industry) and software engineering can be used to accelerate the adoption of Industry 4.0.

7.2 Future Work

From the contributions and the conclusions, several lines for future work arise. Some of these new lines of work have not been followed because they do not affect the continuity of the work carried out during this thesis, while others had to be discarded because of limited time and resources. Hence, the future lines of future works can be divided in

the lines to expand the individual projects for each contribution and the general ideas to continue the work of the entire thesis.

7.2.1 C1: Analysis of IoT

From the first contribution, the analysis of IoT communication protocols several lines for future work arise, for both subcontributions.

7.2.1.1 C1.1: Comparison of IoT protocols

In the case of the first subcontribution and its conclusions, the first line of work is to make the analysis on a stressed network. In this scenario, TCP is expected to be an important asset to overcome a congested or a less reliable network, which may result on a better MQTT performance.

Being security an important issue in industrial scenarios, testing the protocols and libraries with the cryptography features enabled and configured with different cipher suites may help to extend the analysis of the MQTT and CoAP and their libraries further. Adding TLS and DTLS for MQTT and CoAP respectively is an interesting approach for future work.

Another line for future work is to enlarge the set of analyzed IoT protocols, with protocols such as HTTP, XMPP, AMQP, or DDS and their performance on resource constrained platforms. Comparing not only the performance of the protocols but also the behavior of concrete implementations might give system designers the tools they need to choose both the protocol and the implementation for their applications.

Last, MQTT-SN is another interesting IoT protocol in development. This protocol is heavily based on MQTT but runs usually on top of UDP, which can be a more fair comparison against CoAP. Existing

MQTT brokers and clients are starting to implement MQTT-SN, but there is not yet a mature reference implementation to base a study on.

7.2.1.2 C1.2: Comparison of CoAP implementations

In the subcontribution that compares CoAP protocols, some future lines of work also can be pointed out. The comparison of CoAP implementation has been carried out on a Raspberry Pi platform. This platform is quite powerful compared to resource constrained ones, hence, evaluating the libraries on more constrained devices, e.g., ARM Cortex-M architecture, would be valuable. For such devices, Real Time Operative System or baremetal implementations would be required, so Python, Java and Node.js implementations would probably need to be discarded. However, this analysis might be very useful as Industry 4.0 points to small devices.

Another line for future work is the analysis of the performance of CoAP implementations in backbone systems that may need to serve thousands of devices. Comparing the libraries in terms of scalability might be relevant, since Java, Python or Node.js libraries may overcome the performance of C implementations in larger scenarios. For some projects, the selected implementation for a backbone system may need to satisfy the demands of large systems with a huge number of connected devices.

In this subcontribution, the comparison has been carried out between CoAP libraries. A similar analysis for other protocols such as MQTT, AMQP and DDS, may also provide useful insights for their use in resource constrained platforms.

As explained in the previous subcontribution, security is important in industrial domains. Hence, including the DTLS libraries is the

last proposed line for future work. However, the DTLS implementations are still not mature enough, but when they are more developed and integrated with CoAP libraries, comparing the results of secured transactions and against non-secured tests should also be worthy.

7.2.2 **C2: Interoperability though IEC 61850**

Regarding the C2 contribution, subcontribution C2.2 is a continuation of C2.1. In this contribution, the IEC 61850 has been mapped to CoAP to then be implemented and compared to approaches using HTTP and SOAP. However, there are open lines for continuing the work carried out during this contribution.

RESTful system can have different levels [105]. Using resources is the first level, a way to divide the complexity of a system simplifying the services. The second level adds verbs or methods to perform different actions while using the same URI. The highest level adds discoverability to make the protocol more self-documenting. For clients to be able to discover resources interactively, Hypertext As The Engine Of Application State (HATEOAS) provides ways to add hypermedia indicating what can be done next to the payload of server responses [172, 218]. This contribution fits in the second level in the current state, but as future work, an improvement to be compatible with the third level is possible.

Another important CoAP extension is the Resource Directory (RD) [182], which serves a directory with the list of the resources available not only on a server, but the entire system. In this contribution, the mapping is implemented on a single host, therefore, the RD has not been used. However, as a future line of work, this extension can be useful for multiserver systems (i.e., Wind Farms), and build systems

with greater added value (e.g., creating high level applications that involve resource matchmaking and composition technologies).

The block-wise transfer extension is another important CoAP extension that has been used in this contribution. It allows to split the data when it is large. Analyzing the latency of the messages sent, it can be concluded that the block-wise transfer penalizes a lot the performance of the CoAP mapping, especially when many blocks are needed. A path for future work is to search for more efficient ways to send large payloads with CoAP over UDP.

The analysis in this contribution has been carried out in a one on one laboratory setting and a dedicated WiFi network. To go further with the analysis, testing the performance on a system with more nodes and with a not ideal network is required. This can be carried out using a simulation tool (e.g., Omnet++, Ns-2, etc.) or real hardware in an electrical substation.

Another limitation of this contribution is that the analysis has been carried out in a single scenario. Using different application domains which require different models such as electric vehicles, microgrids, photovoltaics, or microturbines would improve the analysis. In these scenarios, advanced properties such as real-time data analysis, proactive control, and automated outage mitigation mechanisms [13] could be enabled. The applicable domain can also be expanded to outside of energy domains to Smart Homes, Ambient Media and Health management for instance.

Finally, a last proposed line of work could be to propose a mapping to other IoT protocols such as MQTT or AMQP, and compare their performance. However, the suitability of these protocols in this use case

needs to be studied to check whether the features fit the requirements for the IEC 61850.

7.2.3 C3: CoAP enhancement proposal

The C3 contribution presents new CoAP options and response codes in order to have more advanced subscription mechanisms for observing resources. To continue the work, the first step is to experiment with more use cases and to carry further analysis, to later present a document to the IETF with the goal of creating an RFC.

7.2.4 C4: Apply Software Engineering techniques in IoT environments

This contribution has two distinguished subcontributions that also have their own future lines for continuing the work.

7.2.4.1 C4.1: IEC 61850 based metamodel

This subcontribution explains the tool for generating artifacts for ICPSs based on the IEC 61850 standard named TRILATERAL. However, there is still room for improvement with this tool and its evaluation.

The first line for future work is to apply TRILATERAL to more diverse domains, such as Automated Warehouses or Press Machines. These use cases would allow to validate its suitability for more diverse ICPSs, even outside of energy related use cases.

Another point for improvement is to enhance TRILATERAL to be a Dynamic SPL, which would allow to have physical element changes (update, add or remove physical nodes) in the designed ICPS. Currently, the model is static and no change can be done without a system update.

At this moment, TRILATERAL can not be updated remotely, the model has to be updated using the tree view editor and the generated source code has to be compiled in order to create the artifact. A new possible functionality for TRILATERAL would be the ability for remote updates, which would allow an ICPS to easily adapt to the surrounding context and the changes on the model and hardware, also reducing maintenance costs.

As explained in Chapter 6, one of the parts that generates the artifact, i.e., the driver, has to be manually implemented. A huge possible improvement for TRILATERAL would be to automatically generate the drivers, allowing for an easy link between the data model and the physical devices (sensors, displays and actuators). However, this task is very dependant on the specific hardware for each project.

Finally, creating TRILATERAL has been costly in terms of workforce compared to created from scratch. The benefits of these type of tools are expected in longer term, as shown in Figure 7.1. It is expected that using it in future projects will help to reduce the costs of those projects. A Return Of Investment (ROI) analysis is expected as a future work, when the data about the costs of future projects is available.

7.2.4.2 **C4.2: Interoperability through WoT**

The last subcontribution in this thesis has been to create a new tool that uses the WoT TD instead of the IEC 61850 to generate the source code automatically. However, this work may still be improved and completed, taking into account that the WoT's definition has not been finished yet, it is still in progress.

The main continuation for this work is to support the complete TD. In order to do that, the first step will be to update both the concrete and

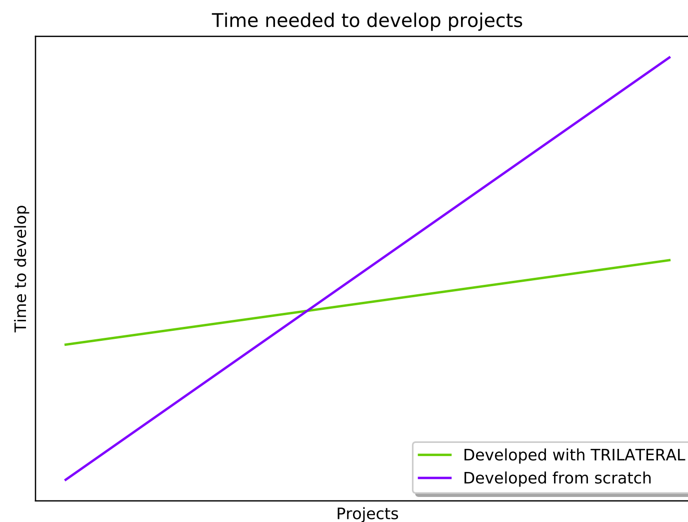


Figure 7.1: General view of the time to develop projects from scratch versus projects implemented with TRILATERAL.

abstrac syntax metamodels when the WoT TD definition is finalized. Also, the parts that have been left out (i.e., the security classes) will be included. The JSON in the second iteration supports the security classes as input, but it does not do anything with them at this stage.

Same as the previous subcontribution, the system in this one is static, so only the *ConsumedThing* interface has been implemented. In order to support a dynamic system, the *ExposedThing* and *WoT* interfaces will also have to be implemented.

Finally, the work carried out for this subcontribution has used CoAP as the communication protocol. However, the WoT is open to use more of them, such as HTTP or MQTT, so to add them is also in the roadmap of this tool. Thanks to the modular design of the generated code, adding new protocols is not very time consuming, as just the two uppermost layers of projects need to be coded.

7.2.5 Future work of the thesis

In addition to the future lines of work for each contribution, the work of the thesis in its completeness can also be expanded.

First, one of the main limitations of the work carried out during this thesis is that the security aspects of the work have been considered out of scope. However, as previously explained, for industrial scenarios' security is essential. Hence, the first line for continuing the work carried out during this thesis is to add security features to the initial analysis and then to the rest of the contributions.

Second, the contributions of this thesis focus on easing and accelerating the development processes for adopting the Industry 4.0. Another important aspect before deploying the systems is the testing and validation of them. In this aspect, generating tools for automating the testing and validation of the generated systems may help to accelerate even further the adoption of Industry 4.0.

Finally, the last limitation of this thesis is that most of the parts of the work have been deployed on lab environments. In this regard, the last line for continuing the work is to overcome this issue, testing the presented contribution on real industrial deployments instead of laboratory set-ups.

Bibliography

[1] 1248. *microcoap*. URL: https://github.com/1248/microcoap.

[2] Mohammad Aazam, Sherali Zeadally, and Khaled A. Harras. "Deploying Fog Computing in Industrial Internet of Things and Industry 4.0". In: *IEEE Transactions on Industrial Informatics* 14.10 (2018), pp. 4674–4682. DOI: 10.1109/TII.2018.2855198.

[3] Michele Albano, Luis Lino Ferreira, Luís Miguel Pinho, and Abdel Rahman Alkhawaja. "Message-oriented middleware for smart grids". In: *Computer Standards & Interfaces* 38 (Feb. 2015), pp. 133–143. ISSN: 09205489. DOI: 10.1016/j.csi.2014.08.002.

[4] Feda AlShahwan, Maha Faisal, and Godwin Ansa. "Security framework for RESTful mobile cloud computing Web services". In: *Journal of Ambient Intelligence and Humanized Computing* 7.5 (2016), pp. 649–659. DOI: 10.1007/s12652-015-0308-5.

[5] *AMQP*. URL: https://www.amqp.org/.

[6] M Anusha, ES Babu, LS Mahesh Reddy, AV Krishna, and B Bhagyasree. "Performance analysis of data protocols of internet of things: a qualitative review". In: *International Journal of Pure and Applied Mathematics* 115.6 (2017), pp. 37–47.

[7] ARM. *mbed TLS*. https://tls.mbed.org/. 2016.

[8] Kevin Ashton. "That 'internet of things' thing". In: *RFID journal* 22.7 (2009), pp. 97–114.

[9] Inmaculada Ayala, Mercedes Amor, Lidia Fuentes, and José M. Troya. "A Software Product Line Process to Develop Agents for the IoT". In: *Sensors* 15.7 (2015), pp. 15640–15660. DOI: 10.3390/s150715640.

[10] Iman Azimi, Amir M. Rahmani, Pasi Liljeberg, and Hannu Tenhunen. "Internet of things for remote elderly monitoring: a study from user-cen tered perspective". In: *Journal of Ambient Intelligence and Humanized Computing* 8.2 (2017), pp. 273–289. DOI: 10.1007/s12652-016-0387-y.

[11] Soma Bandyopadhyay and Abhijan Bhattacharyya. "Lightweight Internet protocols for web enablement of sensors using constrained gateway devices". In: 2013, pp. 334–340. DOI: 10.1109/ICCNC.2013.6504105.

[12] Andrew Banks and Rahul Gupta. "MQTT Version 3.1. 1". In: *OASIS standard* (2014).

[13] Guneet Bedi, Ganesh Kumar Venayagamoorthy, Rajendra Singh, Richard R. Brooks, and Kuang-Ching Wang. "Review of Internet of Things (IoT) in Electric Power and Energy Systems". In: *IEEE Internet of Things Journal* 5.2 (Apr. 2018), pp. 847–870. ISSN: 2327-4662. DOI: 10.1109/JIOT.2018.2802704.

[14] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. "Neo4EMF, A Scalable Persistence Layer for EMF Models". In: *Modelling Foundations and Applications*. Ed. by Jordi Cabot and Julia Rubin. Cham: Springer International Publishing, 2014, pp. 230–241. ISBN: 978-3-319-09195-2.

[15] Yanbing Bi, Lin Jiang, Xin Jun Wang, and Li Zhen Cui. "Mapping of IEC 61850 to Data Distribute Service for digital substation communication". In: *IEEE Power and Energy Society General Meeting*. 2013, pp. 1–5. ISBN: 9781479913039. DOI: 10.1109/PESMG.2013.6672970.

[16] Francesco Bonavolonta, Chiara Caputi, Annalisa Liccardo, and Alessandro Teotino. "Protection of MV smart grid based on IoT technology". In: 2019, pp. 112–116. DOI: 10.1109/METROI4.2019.8792881.

[17] Carsten Bormann, August Betzler, Carles Gomez, and Ilker Demirkol. *CoAP Simple Congestion Control/Advanced*. Internet-Draft draft-ietf-core-cocoa-03. Work in Progress. Internet Engineering Task Force, Feb. 2018. 18 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-core-cocoa-03`.

[18] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks*. RFC 7228. May 2014. DOI: `10.17487/RFC7228`. URL: `https://rfc-editor.org/rfc/rfc7228.txt`.

[19] Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. Oct. 2013. DOI: `10.17487/RFC7049`. URL: `https://rfc-editor.org/rfc/rfc7049.txt`.

[20] Carsten Bormann and Zach Shelby. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. RFC 7959. Aug. 2016. DOI: `10.17487/RFC7959`. URL: `https://rfc-editor.org/rfc/rfc7959.txt`.

[21] Safa Bougouffa, Kilian Meßzmer, Suhyun Cha, Emanuel Trunzer, and Birgit Vogel-Heuser. "Industry 4.0 interface for dynamic reconfiguration of an open lab size automated production system to allow remote community experiments". In: *2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*. Dec. 2017, pp. 2058–2062. DOI: `10.1109/IEEM.2017.8290254`.

[22] Pablo Calcina-Ccori, Laisa Costa, Geovane Fedrecheski, John Esquiagola, Marcelo Zuffo, and Flávio Corrêa da Silva. "Agile Servient Integration with the Swarm: Automatic Code Generation for Nodes in the Internet of Things". In: *Proceedings of the International Conference on Future Networks and Distributed Systems*. ICFNDS '17. Cambridge, United Kingdom: ACM, 2017, 30:1–30:6. ISBN: 978-1-4503-4844-7. DOI: `10.1145/3102304.3102334`.

[23] The Eclipse Foundation. *Californium*. `https://www.eclipse.org/californium/`. 2017.

[24] Isidro Calvo, Oier García De Albéniz, Adrián Noguero, and Federico Pérez. "Towards a modular and scalable design for the communications of electrical protection relays". In: *IECON Proceedings (Industrial Electronics Confer-*

ence) (2009), pp. 2511–2516. ISSN: 1553-572X. DOI: 10.1109/IECON.2009.5415221.

[25] Isidro Calvo, Oier Garcia de Albéniz, and Federico Pérez. "A communication backbone for Substation Automation Systems based on the OMG DDS standard". In: *Przeglad Elektrotechniczny*. Vol. 88. 2012, pp. 146–150.

[26] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. "An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry". In: *Journal of Systems and Software* 91 (2014), pp. 3–23. ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.12.038.

[27] Rich Castagna. *How Smart Grid Technology Is Driving Renewable Energy*. https://www.iotworldtoday.com/2019/08/29/how-smart-grid-technology-is-driving-renewable-energy/. 2019.

[28] Miguel Castro, Antonio J. Jara, and Antonio F.G. Skarmeta. "Smart lighting solutions for smart cities". In: *Proceedings - 27th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2013*. 2013, pp. 1374–1379. DOI: 10.1109/WAINA.2013.254.

[29] Jim Chase. "The evolution of the internet of things". In: *Texas Instruments* 1 (2013), pp. 1–7.

[30] Yuang Chen and Thomas Kunz. "Performance evaluation of IoT protocols under a constrained wireless access network". In: *2016 International Conference on Selected Topics in Mobile and Wireless Networking, MoWNeT 2016* (2016). DOI: 10.1109/MoWNet.2016.7496622.

[31] Dong-Kyu Choi, Joong-Hwa Jung, Hyung-Woo Kang, and Seok-Joo Koh. "Cluster-based CoAP for Message Queueing in Intemet-of-Things Networks". In: *International Conference on Advanced Communication Technology, ICACT*. IEEE, 2017, pp. 584–588. DOI: 10.23919/ICACT.2017.7890157.

[32] Federico Ciccozzi and Romina Spalazzese. "MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering". In: *Intelligent Distributed Computing X*. Ed. by Costin Badica, Amal El Fallah Seghrouchni, Aurélie Beynier, David Camacho, Cédric Herpson, Koen Hindriks, and Paulo Novais.

Cham: Springer International Publishing, 2017, pp. 67–76. ISBN: 978-3-319-48829-5. DOI: 10.1007/978-3-319-48829-5_7.

[33] Catalin Cimpanu. *The CoAP protocol is the next big thing for DDoS attacks*. https://www.zdnet.com/article/the-coap-protocol-is-the-next-big-thing-for-ddos-attacks/. 2018.

[34] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. https://tools.ietf.org/html/rfc7252. 2014.

[35] Giacomo Tanganelli. *CoAPthon*. https://github.com/Tanganelli/CoAPthon. 2019.

[36] People Power Co. *CoAPy: Constrained Application Protocol in Python*. http://coapy.sourceforge.net/. 2010.

[37] Walter Colitti, Kris Steenhaut, Niccolò De Caro, Bogdan Buta, and Virgil Dobrota. "Evaluation of constrained application protocol for wireless sensor networks". In: *IEEE Workshop on Local and Metropolitan Area Networks*. 2011. DOI: 10.1109/LANMAN.2011.6076934.

[38] Louis Columbus. *Roundup Of Internet Of Things Forecasts And Market Estimates, 2016*. https://www.forbes.com/sites/louiscolumbus/2016/11/27/roundup-of-internet-of-things-forecasts-and-market-estimates-2016/#2cc4123d292d. 2016.

[39] Burak H. Çorak, Feyza Yildirim Okay, Metehan Güzel, Sahin Murt, and Suat Ozdemir. "Comparative Analysis of IoT Communication Protocols". In: 2018. DOI: 10.1109/ISNCC.2018.8530963.

[40] Noélia Correia, David Sacramento, and Gabriela Schutz. "Dynamic Aggregation and Scheduling in CoAP/Observe-Based Wireless Sensor Networks". In: *IEEE Internet of Things Journal* 3.6 (Dec. 2016), pp. 923–936. DOI: 10.1109/JIOT.2016.2517120.

[41] Noélia Correia, Gabriela Schutz, and Alvaro Barradas. "Fairness for CoAP/Observe based wireless sensor networks with aggregation deployment". In: *IEEE World Forum on Internet of Things, WF-IoT 2015 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 2015, pp. 110–115. DOI: 10.1109/WF-IoT.2015.7389036.

[42] Niccolo De Caro, Walter Colitti, Kris Steenhaut, Giuseppe Mangino, and Gianluca Reali. "Comparison of two lightweight protocols for smartphone-based sensing". In: *IEEE SCVT 2013 - Proceedings of 20th IEEE Symposium on Communications and Vehicular Technology in the BeNeLux* (2013). DOI: 10.1109/SCVT.2013.6735994.

[43] Flávia C. Delicato, Paulo F. Pires, and Thais Batista. *Middleware Solutions for the Internet of Things*. London: Springer Publishing Company, Incorporated, 2013. ISBN: 978-1-4471-5481-5. DOI: 10.1007/978-1-4471-5481-5.

[44] Óscar Díaz and Cristóbal Arellano. "Integrating microblogging into domain specific language editors". In: 2013, pp. 219–225. DOI: 10.1109/CGC.2013.42.

[45] Tim Dierks and Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://rfc-editor.org/rfc/rfc5246.txt.

[46] Konrad Diwold, Simon Mayer, Alfred Einfalt, Josiane Xavier Parreira, Jack Hodges, Darko Anicic, and Ralf Mosshammer. "Grid Watch Dog: A Stream Reasoning Approach for Lightweight SCADA Functionality in Low-Voltage Grids". In: *Proceedings of the Eighth International Conference on the Internet of Things (IoT 2018)*. 2018. DOI: 10.1145/3277593.3277601.

[47] *Eclipse Foundation*. URL: https://eclipse.org/org/foundation/.

[48] Asma Elmangoush, Ronald Steinke, Thomas Magedanz, Andreea Ancuta Corici, Alex Bourreau, and Adel Al-Hezmi. "Application-derived communication protocol selection in M2M platforms for smart cities". In: *2015 18th International Conference on Intelligence in Next Generation Networks, ICIN 2015*. Institute of Electrical and Electronics Engineers Inc., 2015, pp. 76–82. DOI: 10.1109/ICIN.2015.7073810.

[49] Matthias Kovatsch. *Erbium*. http://people.inf.ethz.ch/mkovatsc/erbium.php. 2014.

[50] Ismael Etxeberria-Agiriano, Isidro Calvo, Federico Pérez, and Oier García de Albéniz. "Mapping different communication traffic over DDS in industrial environments". In: *6th Iberian Conference on Information Systems and Technologies (CISTI 2011)*. 2011.

[51] Dave Evans. "The internet of things: How the next evolution of the internet is changing everything". In: *CISCO white paper* 1.2011 (2011), pp. 1–11.

[52] Roy Thomas Fielding. "REST: Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000.

[53] DDS Foundation. *DDS*. URL: http://portals.omg.org/dds/.

[54] Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley signature series. Reading, Massachusetts, USA: Addison-Wesley, 2011. ISBN: 978-0-321-71294-3. URL: http://vig.pearsoned.com/store/product/1,1207,store-12521%5C_isbn-0321712943,00.html.

[55] Free Software Foundation. *A Lightweight TCP/IP stack*. http://savannah.nongnu.org/projects/lwip/. 2002.

[56] Keith Cullen. *FreeCoAP*. https://github.com/keith-cullen/FreeCoAP. 2017.

[57] FreeRTOS. *FreeRTOS*. http://www.freertos.org/. 2016.

[58] Paul Fremantle. "A reference architecture for the internet of things". In: *WSO2 White paper* (2015).

[59] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys Tutorials* 17.4 (2015), pp. 2347–2376. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2444095.

[60] Davide Della Giustina, Paolo Ferrari, Alessandra Flammini, Stefano Rinaldi, and Emiliano Sisinni. "Automation of Distribution Grids With IEC 61850: A First Approach Using Broadband Power Line Communication". In: *IEEE Transactions on Instrumentation and Measurement* 62.9 (Sept. 2013), pp. 2372–2383. ISSN: 0018-9456. DOI: 10.1109/TIM.2013.2270922.

[61] GnuTLS. *The GnuTLS Transport Layer Security Library*. https://www.gnutls.org/. 2017.

[62] Abel Gómez, Iker Fernandez de Larrea, Markel Iglesias-Urkia, Beatriz Lopez-Davalillo, Aitor Urbieta, and Jordi Cabot. "Una Aproximación Basada en Modelos para la Definición de Arquitecturas Asíncronas". In: *JISBD*. 2019. DOI: 11705/JISBD/2019/035.

[63] David Greenfield. *Is Raspberry Pi Ready for Industry?* https://www.automationworld.com/raspberry-pi-ready-industry. 2019.

[64] Robert van Engelen. *gSOAP 2.8.66 User Guide*. https://www.genivia.com/doc/soapdoc2.html. 2018.

[65] Cenk Gündoğan, Peter Kietzmann, Martine Lenders, Hauke Petersen, Thomas C. Schmidt, and Matthias Wählisch. "NDN, COAP, and MQTT: A comparative measurement study in the IoT". In: 2018, pp. 159–171. DOI: 10.1145/3267955.3267967.

[66] Łukasz Walukiewicz. *h5.coap*. https://github.com/morkai/h5.coap. 2014.

[67] Samer Hamdani and Hassan Sbeyti. "A comparative study of COAP and MQTT communication protocols". In: 2019. DOI: 10.1109/ISDFS.2019.8757486.

[68] Klaus Hartke. *Observing Resources in the Constrained Application Protocol (CoAP)*. RFC 7641. Sept. 2015. DOI: 10.17487/RFC7641.

[69] Scott Hilton. *Dyn Analysis Summary Of Friday October 21 Attack*. https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/. 2016.

[70] S. M. Suhail Hussain, Mohd Asim Aftab, and Ikbal Ali. "IEC 61850 Modeling of DSTATCOM and XMPP Communication for Reactive Power Management in Microgrids". In: *IEEE Systems Journal* (2018), pp. 1–11. ISSN: 1932-8184. DOI: 10.1109/JSYST.2017.2769706.

[71] IEC TC-88. *Wind energy generation systems - Part 25-4: Communications for monitoring and control of wind power plants - Mapping to communication profile*. 2016.

[72] IEC TC-57. *Communication networks and systems in substations – Part 1: Introduction and overview*. 2003.

[73] IEC TC-57. *Communication networks and systems in substations – Part 7-1: Basic communication structure for substation and feeder equipment – Principles and models*. 2003.

[74] IEC TC-57. *Communication networks and systems in substations – Part 7-2: Basic communication structure for substation and feeder equipment – Abstract communication service interface (ACSI)*. 2003.

[75] IEC TC-57. *Communication networks and systems in substations – Part 8-1: Specific Communication Service Mapping (SCSM) – Mappings to MMS (ISO 9506-1 and ISO 9506-2) and to ISO/IEC 8802-3*. 2003.

[76] IEC TC-57. *Communication networks and systems in substations – Part 9-1: Specific Communication Service Mapping (SCSM) – Sampled values over serial unidirectional multidrop point to point link*. 2003.

[77] IEC TC-57. *Communication networks and systems in substations – Part 9-2: Specific Communication Service Mapping (SCSM) – Sampled values over ISO/IEC 8802-3*. 2003.

[78] IETF. *Constrained RESTful Environments (core)*. https://datatracker.ietf.org/wg/core/documents. 2017.

[79] IETF. *Constrained RESTful Environments (core)*. https://datatracker.ietf.org/wg/core/charter. 2019.

[80] IETF. *Constrained RESTful Environments (core)*. https://datatracker.ietf.org/wg/core/documents. 2018.

[81] Aitziber Iglesias., Markel Iglesias-Urkia., Beatriz López-Davalillo., Santiago Charramendieta., and Aitor Urbieta. "TRILATERAL: Software Product Line based Multidomain IoT Artifact Generation for Industrial CPS". In: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,* INSTICC. Prague, Czech Republic: SciTePress, 2019, pp. 64–73. ISBN: 978-989-758-358-2. DOI: 10.5220/0007343500640073.

[82] Aitziber Iglesias, Hong Lu, Cristóbal Arellano, Tao Yue, Shaukat Ali, and Goiuria Sagardui. "Product Line Engineering of Monitoring Functionality in Industrial Cyber-Physical Systems: A Domain Analysis". In: *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A*. 2017, pp. 195–204. DOI: 10.1145/3106195.3106223.

[83] Aitziber Iglesias, Goiuria Sagardui, and Cristobal Arellano. "Industrial Cyber-Physical System Evolution Detection and Alert Generation". In: *Applied Sciences* 9.8 (2019). ISSN: 2076-3417. DOI: 10.3390/app9081586.

[84] Aitziber Iglesias, Tao Yue, Cristóbal Arellano, Shaukat Ali, and Goiuria Sagardui. "Model- Based Personalized Visualization System for Monitoring Evolving Industrial Cyber-Physical System". In: vol. 2018-December. 2018, pp. 532–541. DOI: 10.1109/APSEC.2018.00068.

[85] Markel Iglesias-Urkia, Diego Casado-Mansilla, Simon Mayer, Josu Bilbao, and Aitor Urbieta. "Integrating Electrical Substations within the IoT using IEC 61850, CoAP and CBOR". In: *IEEE Internet of Things Journal* 6.5 (Oct. 2019), pp. 7437–7449. ISSN: 2327-4662. DOI: 10.1109/JIOT.2019.2903344.

[86] Markel Iglesias-Urkia, Diego Casado-Mansilla, Simon Mayer, and Aitor Urbieta. "Enhanced publish/subscribe in CoAP: Describing advanced subscription mechanisms for the Observe extension". In: *ACM International Conference Proceeding Series*. Association for Computing Machinery, 2018. DOI: 10.1145/3277593.3277594.

[87] Markel Iglesias-Urkia, Diego Casado-Mansilla, Simon Mayer, and Aitor Urbieta. "Validation of a CoAP to IEC 61850 Mapping and Benchmarking vs HTTP-REST and WS-SOAP". In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. Vol. 2018-September. 2018, pp. 1015–1022. DOI: 10.1109/ETFA.2018.8502624.

[88] Markel Iglesias-Urkia, Abel Gómez, Diego Casado-Mansilla, and Aitor Urbieta. "Automatic generation of Web of Things servients using Thing Descriptions". In: *Personal and Ubiquitous Computing, Submitted* (2019).

[89] Markel Iglesias-Urkia, Abel Gómez, Diego Casado-Mansilla, and Aitor Urbieta. "Enabling Easy Web of Things Compatible Device Generation Using a Model-Driven Engineering Approach". In: *Proceedings of the 9th International Conference on the Internet of Things*. IoT 2019. Bilbao, Spain: ACM, 2019, 25:1–25:8. ISBN: 978-1-4503-7207-7. DOI: 10.1145/3365871.3365898.

[90] Markel Iglesias-Urkia, Aitziber Iglesias, Beatriz López-Davalillo, Santiago Charramendieta, Diego Casado-Mansilla, Goiuria Sagardui, and Aitor Urbieta. "TRILATERAL: A Model-Based Approach for Industrial CPS – Monitoring and Control". In: *Model-Driven Engineering and Software Development*. Ed. by Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić. Cham: Springer International Publishing, 2020, pp. 376–398. ISBN: 978-3-030-37873-8.

[91] Markel Iglesias-Urkia, Adrián Orive, Marc Barcelo, Adrian Moran, Josu Bilbao, and Aitor Urbieta. "Towards a lightweight protocol for Industry 4.0: An implementation based benchmark". In: *Proceedings of the 2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics, ECMSM 2017*. Institute of Electrical and Electronics Engineers Inc., 2017. DOI: 10.1109/ECMSM.2017.7945894.

[92] Markel Iglesias-Urkia, Adrián Orive, and Aitor Urbieta. "Analysis of CoAP Implementations for Industrial Internet of Things: A Survey". In: *Procedia Computer Science*. Ed. by Shakshuki E. Vol. 109. Elsevier B.V., 2017, pp. 188–195. DOI: 10.1016/j.procs.2017.05.323.

[93] Markel Iglesias-Urkia, Adrián Orive, Aitor Urbieta, and Diego Casado-Mansilla. "Analysis of CoAP implementations for industrial Internet of Things: a survey". In: *Journal of Ambient Intelligence and Humanized Computing* 10.7 (July 2019), pp. 2505–2518. ISSN: 1868-5145. DOI: 10.1007/s12652-018-0729-z.

[94] Markel Iglesias-Urkia, Aitor Urbieta, Jorge Parra, and Diego Casado-Mansilla. "IEC 61850 meets CoAP: Towards the integration of Smart Grids and IoT standards". In: *ACM International Conference Proceeding Series*. Association for Computing Machinery, 2017. DOI: 10.1145/3131542.3131545.

[95] Forbes Insights. *How IoT Is Impacting 7 Key Industries Today*. https://www.forbes.com/sites/insights-inteliot/2018/08/24/how-iot-is-impacting-7-key-industries-today/. 2018.

[96] International Controller Association (ICA). *Controlling in the Age of Intelligent Networks Dream Car of the Dream Factory of the ICV 2015*. https://www.icv-controlling.com/en/work-groups/think-tank/industrie-40.html. 2015.

[97] IoT One. *Emerging Open And Standard Protocol Stack For IoT*. https://www.iotone.com/guide/emerging-open-and-standard-protocol-stack-for-iot/g494. 2016.

[98] Ian Skerrett. *IoT Developer Survey 2017*. https://www.slideshare.net/IanSkerrett/iot-developer-survey-2017. 2017.

[99] Isam Ishaq, Jeroen Hoebeke, Ingrid Moerman, and Piet Demeester. "Observing CoAP Groups Efficiently". In: *Ad Hoc Networks* 37 (Feb. 2016), pp. 368–388. DOI: 10.1016/j.adhoc.2015.08.030.

[100] Giacomo Tanganelli. *PUT response with payload*. https://github.com/Tanganelli/CoAPthon/issues/45. 2016.

[101] Giacomo Tanganelli. *Server auto-token generation breaks RFC 7252*. https://github.com/Tanganelli/CoAPthon/issues/48. 2017.

[102] Jonathan Jeon. *Web Browser as Universal client for IoT*. https://www.slideshare.net/hollobit/web-browser-as-universal-client-for-iot. 2016.

[103] *JMS*. URL: http://www.oracle.com/technetwork/java/jms/index.html.

[104] open-source-parsers. *JsonCpp*. https://github.com/open-source-parsers/jsoncpp. 2018.

[105] Leonard Richardson. *Justice Will Take Us Millions Of Intricate Moves*. https://www.crummy.com/writing/speaking/2008-QCon/act3.html. 2009.

[106] Henning Kagermann, Johannes Helbig, Ariane Hellinger, and Wolfgang Wahlster. *Recommendations for Implementing the Strategic Initiative INDUSTRIE 4.0: Securing the Future of German Manufacturing Industry; Final Report of the Industrie 4.0 Working Group*. Forschungsunion, 2013. URL: https://books.google.es/books?id=AsfOoAEACAAJ.

[107] Charalampos Kalalas. "Cellular networks for smart grid communication". Doctoral dissertation. Universitat Politècnica de Catalunya, 2018.

[108] Isak Karabegović, Edina Karabegović, Mehmed Mahmić, and Ermin Husak. "Implementation of Industry 4.0 and Industrial Robots in the Manufacturing Processes". In: *Lecture Notes in Networks and Systems* 76 (2020), pp. 3–14. DOI: 10.1007/978-3-030-18072-0_1.

[109] Paridhika Kayal and Harry Perros. "A comparison of IoT application layer protocols through a smart parking implementation". In: 2017, pp. 331–336. DOI: 10.1109/ICIN.2017.7899436.

[110] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008. ISBN: 978-0-470-03666-2. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470036664.html.

[111] Girum Ketema, Jeroen Hoebeke, Ingrid Moerman, Piet Demeester, Li Shi Tao, and Antonio J. Jara. "Efficiently Observing Internet of Things Resources". In: *Proceedings - 2012 IEEE Int. Conf. on Green Computing and Communications, GreenCom 2012, Conf. on Internet of Things, iThings 2012 and Conf. on Cyber, Physical and Social Computing, CPSCom 2012*. IEEE, 2012, pp. 446–449. DOI: 10.1109/GreenCom.2012.70.

[112] Hasan Ali Khattak, Michele Ruta, and Eugenio Eugenio Di Sciascio. "CoAP-based healthcare sensor networks: A survey". In: *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences and Technology, IBCAST 2014*. IEEE Computer Society, 2014, pp. 499–503. DOI: 10.1109/IBCAST.2014.6778196.

[113] Marco Lobe Kome, Frederic Cuppens, Nora Cuppens-Boulahia, and Vincent Frey. "CoAP Enhancement for a better IoT centric protocol: CoAP 2.0". In: 2018, pp. 139–146. DOI: 10.1109/IoTSMS.2018.8554494.

[114] Michael Koster, Ari Keränen, and Jaime Jimenez. *Publish-Subscribe Broker for the Constrained Application Protocol (CoAP)*. Internet-Draft draft-ietf-core-coap-pubsub-04. Work in Progress. Internet Engineering Task Force, Mar. 2018. 23 pp.

[115] Thomas Kothmayr, Corinna Schmitt, Wen Hu, Michael Brünig, and Georg Carle. "A DTLS based end-to-end security architecture for the Internet of Things with two-way authentication". In: *Proceedings - Conference on Local Computer Networks, LCN*. 2012, pp. 956–963. DOI: 10.1109/LCNW.2012.6424088.

[116] Thomas Kothmayr, Corinna Schmitt, Wen Hu, Michael Brünig, and Georg Carle. "DTLS based security and two-way authentication for the Internet of Things". In: *Ad Hoc Networks* 11.8 (2013), pp. 2710–2723. DOI: 10.1016/j.adhoc.2013.05.003.

[117] Matthias Kovatsch. "Demo Abstract: Human–CoAP Interaction with Copper". In: *Proceedings of the 7th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2011)*. Barcelona, Spain, June 2011. DOI: 10.1109/DCOSS.2011.5982145.

[118] Matthias Kovatsch, Ryuichi Matsukura, Michael Lagally, Toru Kawaguchi, Kunihiko Toumura, and Kazuo Kajimoto. *Web of Things (WoT) Architecture*. https://www.w3.org/TR/wot-architecture/. 2017.

[119] Carel P. Kruger and Gerhard Petrus Hancke. "Benchmarking Internet of things devices". In: *Proceedings - 2014 12th IEEE International Conference on Industrial In formatics, INDIN 2014*. Institute of Electrical and Electronics Engineers Inc., 2014, pp. 611–616. DOI: 10.1109/INDIN.2014.6945583.

[120] Koojana Kuladinithi, Olaf Bergmann, Thomas Pötsch, Markus Becker, and Carmelita Görg. "Implementation of CoAP and its Application in Transport Logistics". In: *Proc. IP+ SN, Chicago, IL, USA* (2011).

[121] Hyeokjin Kwon, Jiye Park, and Namhi Kang. "Challenges in deploying CoAP over DTLS in resource constrained environments". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9503 (2016). Ed. by Kim H.-W. Choi D., pp. 269–280. DOI: 10.1007/978-3-319-31875-2_22.

[122] Cyber Defense Laboratory. *TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks*. http://discovery.csc.ncsu.edu/software/TinyECC/. 2011.

[123] Anna Larmo, Felipe Del Carpio, Pontus Arvidson, and Roman Chirikov. "Comparison of CoAP and MQTT performance over capillary radios". In: 2018. DOI: 10.1109/GIOTS.2018.8534576.

[124] Anna Larmo, Antti Ratilainen, and Juha Saarinen. "Impact of CoAP and MQTT on NB-IoT system performance". In: *Sensors (Switzerland)* 19.1 (2019). DOI: 10.3390/s19010007.

[125] Paulo Leitão, Armando Walter Colombo, and Stamatis Karnouskos. "Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges". In: *Computers in Industry* 81 (2016), pp. 11–25. DOI: 10.1016/j.compind.2015.08.004.

[126] Francesco Lelli. "Interoperability of the time of Industry 4.0 and the Internet of Things". In: *Future Internet* 11.2 (2019). DOI: 10.3390/fi11020036.

[127] Christian Lerche, Klaus Hartke, and Matthias Kovatsch. "Industry adoption of the Internet of Things: A constrained application protocol survey". In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. 2012. DOI: 10.1109/ETFA.2012.6489787.

[128] Kepeng Li, Akbar Rahman, and Carsten Bormann. *Representing Constrained RESTful Environments (CoRE) Link Format in JSON and CBOR*. Internet-Draft draft-ietf-core-links-json-10. Work in Progress. Internet Engineering Task Force, Feb. 2018. 20 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-core-links-json-10.

[129] Shitao Li, Kepeng Li, Jeroen Hoebeke, Floris Van den Abeele, and Antonio J. Jara. *Conditional observe in CoAP*. Internet-Draft draft-li-core-conditional-observe-05. Work in Progress. Internet Engineering Task Force, Oct. 2014. 14 pp. URL: https://datatracker.ietf.org/doc/html/draft-li-core-conditional-observe-05.

[130] Pavel Kalvoda. *CBOR format implementation for C & others*. http://libcbor.org/. 2017.

[131] Olaf Bergmann. *libcoap: C-Implementation of CoAP*. https://libcoap.net. 2017.

[132] Yang Lu. “Industry 4.0: A survey on technologies, applications and open research issues”. In: *Journal of Industrial Information Integration* 6 (2017), pp. 1–10. ISSN: 2452-414X. DOI: https://doi.org/10.1016/j.jii.2017.04.005.

[133] Alessandro Ludovici, Eduardo Garcia, X. Gimeno, and Anna Calveras Augé. “Adding QoS Support for Timeliness to the Observe Extension of CoAP”. In: *International Conference on Wireless and Mobile Computing, Networking and Communications*. IEEE, 2012, pp. 195–202. DOI: 10.1109/WiMOB.2012.6379074.

[134] Alessandro Ludovici, Pol Moreno, and Anna Calveras. “TinyCoAP: A novel constrained application protocol (CoAP) implementation for embedding RESTful web services in wireless sensor networks based on tinyOS”. In: *Journal of Sensor and Actuator Networks* 2.2 (2013), pp. 288–315. DOI: 10.3390/jsan2020288.

[135] Ana Patrícia Fontes Magalhaes, Aline Maria Santos Andrade, and Rita Suzana Pitangueira Maciel. “Model driven transformation development (MDTD): An approach for developing model to model transformation”. In: *Information and Software Technology* 114 (2019), pp. 55–76. DOI: 10.1016/j.infsof.2019.06.004.

[136] Charles McLellan. *The Industrial Internet of Things: A guide to deployments, vendors and platforms*. https://www.zdnet.com/article/the-industrial-internet-of-things-a-guide-to-deployments-vendors-and-platforms/. 2019.

[137] Free Software Foundation. *GNU Libmicrohttpd*. https://www.gnu.org/software/libmicrohttpd/. 2018.

[138] Richard Mietz, Philipp Abraham, and Kay Römer. “High-level States with CoAP: Giving Meaning to Raw Sensor Values to Support IoT Applications”. In: *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. IEEE, Apr. 2014, pp. 1–6. DOI: 10.1109/ISSNIP.2014.6827637.

[139] Leticia Montalvillo, Óscar Díaz, and Thomas Fogdal. “Reducing coordination overhead in SPLs: Peering in on peers”. In: vol. 1. 2018, pp. 110–120. DOI: 10.1145/3233027.3233041.

[140] *Mosquitto*. URL: https://mosquitto.org/.

[141] MQTT.org. *MQTT used by Facebook Messenger*. https://mqtt.org/2011/08/mqtt-used-by-facebook-messenger. 2011.

[142] Dae-hyeok Mun, Minh Le Dinh, and Young-woo Kwon. “An Assessment of Internet of Things Protocols for Resource-Constrained Applications”. In: vol. 1. 2016, pp. 555–560. DOI: 10.1109/COMPSAC.2016.51.

[143] Behailu Negash, Tomi Westerlund, Amir M. Rahmani, Pasi Liljeberg, and Hannu Tenhunen. “DoS-IL: A Domain Specific Internet of Things Language for Resource Constrained Devices”. In: *Procedia Computer Science*. Ed. by Shakshuki E. Vol. 109. Elsevier B.V., 2017, pp. 416–423. DOI: 10.1016/j.procs.2017.05.411.

[144] Henrik Frystyk Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://rfc-editor.org/rfc/rfc2616.txt.

[145] Matteo Collina. *node-coap*. https://github.com/mcollina/node-coap. 2019.

[146] Mark Nottingham. *Web Linking*. RFC 5988. Oct. 2010. DOI: 10.17487/RFC5988. URL: https://rfc-editor.org/rfc/rfc5988.txt.

[147] Stanley Nwabuona, Markus Schuss, Simon Mayer, Konrad Diwold, Lukas Krammer, and Alfred Einfalt. “Time-synchronized Data Collection in Smart Grids through IPv6 over BLE”. In: *Proceedings of the Eighth International Conference on the Internet of Things (IoT 2018)*. 2018. DOI: 10.1145/3277593.3277632.

[148] OASIS. *OASIS Message Queuing Telemetry Transport (MQTT) TC*. 2017. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt.

[149] OASIS. *MQTT Version 5.0*. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[150] Object Management Group (OMG). *CORBA*. https://www.corba.org/. 2019.

[151] Open Mobile Alliance (OMA). *Lightweight M2M*. http://openmobilealliance.org/iot/lightweight-m2m-lwm2m. 2017.

[152] OpenSSL Software Foundation. *OpenSSL Cryptography and SSL/TLS Toolkit*. https://www.openssl.org/. 2016.

[153] Fathia Ouakasse and Said Rakrak. "A Comparative Study of MQTT and CoAP Application Layer Protocols via. Performances Evaluation". In: *Journal of Engineering and Applied Sciences* 13.15 (2018), pp. 6053–6061. DOI: 10.3923/jeasci.2018.6053.6061.

[154] *Paho C*. URL: https://eclipse.org/paho/clients/c/.

[155] *Paho MQTT C/C++ for Embedded platforms*. URL: https://eclipse.org/paho/clients/c/embedded/.

[156] Palak P. Parikh, Tarlochan Singh Sidhu, and Abdallah Shami. "A Comprehensive Investigation of Wireless LAN for IEC 61850–Based Smart Distribution Substation Applications". In: *IEEE Transactions on Industrial Informatics* 9.3 (Aug. 2013), pp. 1466–1476. ISSN: 1551-3203. DOI: 10.1109/TII.2012.2223225.

[157] Jorge Parra. "Restful Framework for Collaborative Internet of Things Based on IEC 61850". Doctoral dissertation. Universidad del País Vasco - Euskal Herriko Unibertsitatea (UPV/EHU), 2016, pp. 669–675. ISBN: 0780388674.

[158] Pankesh Patel, Muhammad Intizar Ali, and Amit Sheth. "From raw data to smart manufacturing: AI and semantic web of things for industry 4.0". In: *IEEE Intelligent Systems* 33.4 (2018), pp. 79–86. DOI: 10.1109/MIS.2018.043741325.

[159] Anders Bro Pedersen, Einar Bragi Hauksson, Peter Bach Andersen, Bjarne Poulsen, and Dieter Træholt Chresten and Gantenbein. "Facilitating a Generic Communication Interface to Distributed Energy Resources: Mapping IEC 61850 to RESTful Services". In: *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on* (2010), pp. 61–66. DOI: 10.1109/SMARTGRID.2010.5622020.

[160] Dragan Peraković, Marko Periša, and Petra Zorić. "Challenges and issues of ICT in industry 4.0". In: *Lecture Notes in Mechanical Engineering* (2020), pp. 259–269. DOI: 10.1007/978-3-030-22365-6_26.

[161] Goiuri Peralta, Markel Iglesias-Urkia, Marc Barcelo, Raul Gomez, Adrian Moran, and Josu Bilbao. “Fog computing based efficient IoT scheme for the Industry 4.0”. In: *2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM)*. Institute of Electrical and Electronics Engineers Inc., 2017, pp. 1–6. DOI: 10.1109/ECMSM.2017.7945879.

[162] Bo Petersen, Henrik Bindner, Bjarne Poulsen, and Shi You. “Smart Grid communication middleware comparison distributed control comparison for the internet of things”. In: *SMARTGREENS 2017 - Proceedings of the 6th International Conference on Smart Cities and Green ICT Systems*. 2017, pp. 219–226.

[163] Bo Petersen, Henrik Bindner, Shi You, and Bjarne Poulsen. “Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the Internet of Things”. In: *Proceedings of Computing Conference 2017*. Vol. 2018-January. 2018, pp. 1339–1346. DOI: 10.1109/SAI.2017.8252264.

[164] ETSI. *Plugtests*. https://www.etsi.org/events/plugtests. 2019.

[165] Nissanka B. Priyantha, Aaman Kansal, Michel Goraczko, and Feng Zhao. “Tiny web services: Design and implementation of interoperable and evolvable sensor networks”. In: *SenSys’08 - Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems*. 2008, pp. 253–266. DOI: 10.1145/1460412.1460438.

[166] Akbar Rahman and Esko Dijk. *Group Communication for the Constrained Application Protocol (CoAP)*. RFC 7390. Oct. 2014. DOI: 10.17487/RFC7390. URL: https://rfc-editor.org/rfc/rfc7390.txt.

[167] R. S. Raji. “Smart networks for control”. In: *IEEE Spectrum* 31.6 (June 1994), pp. 49–55. DOI: 10.1109/6.284793.

[168] Raspberry Pi Foundation. *Raspberry Pi*. https://www.raspberrypi.org/. 2018.

[169] *Raspberry Pi*. URL: https://www.raspberrypi.org/.

[170] Shahid Raza, Hossein Shafagh, Kasun Hewage, René Hummen, and Thiemo Voigt. "Lithe: Lightweight secure CoAP for the internet of things". In: *IEEE Sensors Journal* 13.10 (2013), pp. 3711–3720. DOI: 10.1109/JSEN.2013.2277656.

[171] Eric Rescorla and Nagendra Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Jan. 2012. DOI: 10.17487/RFC6347. URL: https://rfc-editor.org/rfc/rfc6347.txt.

[172] Martin Fowler. *Richardson Maturity Model: steps toward the glory of REST*. https://martinfowler.com/articles/richardsonMaturityModel.html. 2010.

[173] Till Riedel, Nicolaie Fantana, Adrian Genaid, Dimitar Yordanov, Hedda R. Schmidtke, and Michael Beigl. "Using web service gateways and code generation for sustainable IoT system development". In: *2010 Internet of Things (IOT)*. Tokyo, Japan, Nov. 2010, pp. 1–8. DOI: 10.1109/IOT.2010.5678449.

[174] David Sacramento, Gabriela Schutz, and Noélia Correia. "Aggregation and Scheduling in CoAP/Observe Based Wireless Sensor Networks". In: *IEEE International Conference on Communications*. Vol. 2015-September. IEEE, 2015, pp. 654–660. DOI: 10.1109/ICC.2015.7248396.

[175] Andrey Sadovykh, Wasif Afzal, Dragos Truscan, Pierluigi Pierini, Hugo Bruneliere, Alessandra Bagnato, Abel Gómez, Jordi Cabot, and Orlando Avila-García. "On a tool-supported model-based approach for building architectures and roadmaps: The MegaM@Rt2 project experience". In: *Microprocessors and Microsystems* 71 (2019), p. 102848. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2019.102848.

[176] Adnan Salihbegovic, Teo Eterovic, Enio Kaljic, and Samir Ribic. "Design of a domain specific language and IDE for Internet of things applications". In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. May 2015, pp. 996–1001. DOI: 10.1109/MIPRO.2015.7160420.

[177] Ricardo Sanz, Jose Antonio Clavijo, Miguel J. Segarra, Angélica de Antonio, and Mariano Alonso. "CORBA-Based Substation Automation Systems". In: *Proceedings of IEEE Conference on Control Applications*. 2001.

[178] Douglas C. Schmidt. “Guest Editor’s Introduction: Model-Driven Engineering”. In: *Computer* 39.2 (Feb. 2006), pp. 25–31. ISSN: 0018-9162. DOI: 10.1109/MC.2006.58.

[179] Luca Sciullo, Angelo Trotta, Lorenzo Gigli, and Marco Di Felice. “Deploying W3C Web of Things-Based Interoperable Mash-up Applications for Industry 4.0: A Testbed”. In: *Wired/Wireless Internet Communications*. Vol. 11618 LNCS. 2019, pp. 3–14. DOI: 10.1007/978-3-030-30523-9_1.

[180] Bran Selic. “The pragmatics of model-driven development”. In: *IEEE Software* 20.5 (Sept. 2003), pp. 19–25. DOI: 10.1109/MS.2003.1231146.

[181] Zach Shelby. *Constrained RESTful Environments (CoRE) Link Format*. RFC 6690. Aug. 2012. URL: https://rfc-editor.org/rfc/rfc6690.txt.

[182] Zach Shelby, Michael Koster, Carsten Bormann, Peter Van der Stok, and Christian Amsüss. *CoRE Resource Directory*. Internet-Draft draft-ietf-core-resource-directory-13. Work in Progress. Internet Engineering Task Force, Mar. 2018. 70 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-core-resource-directory-13.

[183] Zach Shelby, Matthieu Vial, Michael Koster, Christian Groves, Julian Zhu, and Bill Silverajan. *Dynamic Resource Linking for Constrained RESTful Environments*. Internet-Draft draft-ietf-core-dynlink-05. Work in Progress. Internet Engineering Task Force, Mar. 2018. 19 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-core-dynlink-05.

[184] Zach Shelby, Matthieu Vial, Michael Koster, Christian Groves, Julian Zhu, and Bill Silverajan. *Reusable Interface Definitions for Constrained RESTful Environments*. Internet-Draft draft-ietf-core-interfaces-11. Work in Progress. Internet Engineering Task Force, Mar. 2018. 27 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-core-interfaces-11.

[185] In-Jae Shin, Byung-Kwen Song, and Doo-Seop Eom. “International Electronical Committee (IEC) 61850 Mapping with Constrained Application Protocol (CoAP) in Smart Grids Based European Telecommunications Standard In-

stitute Machine-to-Machine (M2M) Environment”. In: *Energies* 10.3 (Mar. 2017), p. 393. ISSN: 1996-1073. DOI: 10.3390/en10030393.

[186] Fadi Shrouf, Joaquín Ordieres, and Giovanni Miragliotta. “Smart factories in Industry 4.0: A review of the concept and of energy management approached in production based on the Internet of Things paradigm”. In: *2014 IEEE International Conference on Industrial Engineering and Engineering Management*. Dec. 2014, pp. 697–701. DOI: 10.1109/IEEM.2014.7058728.

[187] Andreas Daniel Sinnhofer, Peter Pühringer, Felix Jonathan Oppermann, Klaus Potzmader, Clemens Orthacker, Christian Steger, and Christian Kreiner. “Combining Business Process Variability and Software Variability Using Traceable Links”. In: *Business Modeling and Software Design - 7th International Symposium, BMSD 2017, Revised Selected Papers*. 2017, pp. 67–86. DOI: 10.1007/978-3-319-78428-1_4.

[188] Robert Quattlebaum. *SMCP - a full-featured embedded CoAP stack*. https://github.com/darconeous/old-coap-archived/. 2017.

[189] Manfred Sneps-Sneppe and Dmitry Namiot. “On web-based domain-specific language for Internet of Things”. In: *International Congress on Ultra Modern Telecommunications and Control Systems and Workshops*. Vol. 2016-January. IEEE Computer Society, 2016, pp. 287–292. DOI: 10.1109/ICUMT.2015.7382444.

[190] Don Box, Gopal Kavivaya, Andrew Layman, Satish Thatte, and Dave Winer. *SOAP: Simple Object Access Protocol*. https://www.ietf.org/archive/id/draft-box-http-soap-01.txt. 1999.

[191] Andy Stanford-Clark and Hong Linh Truong. “Mqtt for sensor networks (mqtt-sn) protocol specification”. In: *International business machines (IBM) Corporation version* 1 (2013), p. 2.

[192] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[193] *STM32F4DISCOVERY*. URL: http://www.st.com/en/evaluation-tools/stm32f4discovery.html.

[194] STMicroelectronics. *Discovery kit with STM32F407VG MCU*. http://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-discovery-kits/stm32f4discovery.html. 2016.

[195] Peter Van der Stok, Carsten Bormann, and Anuj Sehgal. *PATCH and FETCH Methods for the Constrained Application Protocol (CoAP)*. RFC 8132. Apr. 2017. URL: https://rfc-editor.org/rfc/rfc8132.txt.

[196] Kunal Suri, Arnaud Cuccuru, Juan Cadavid, Sebastien Gerard, Walid Gaaloul, and Samir Tata. "Model-based Development of Modular Complex Systems for Accomplishing System Integration for Industry 4.0". In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - MODELSWARD,* INSTICC. SciTePress, 2017, pp. 487–495. ISBN: 978-989-758-210-3. DOI: 10.5220/0006210504870495.

[197] Alejandro Talaminos-Barroso, Miguel A. Estudillo-Valderrama, Laura M. Roa, Javier Reina-Tosina, and Francisco Ortega-Ruiz. "A Machine-to-Machine protocol benchmark for eHealth applications - Use case: Respiratory rehabilitation". In: *Computer Methods and Programs in Biomedicine* (2016). DOI: 10.1016/j.cmpb.2016.03.004.

[198] Hao Tang, Di Li, Shiyong Wang, and Zhijie Dong. "CASOA: An Architecture for Agent-Based Manufacturing System in the Context of Industry 4.0". In: *IEEE Access* 6 (2018), pp. 12746–12754. DOI: 10.1109/ACCESS.2017.2758160.

[199] Giacomo Tanganelli, Carlo Vallati, Enzo Mingozzi, and Matthias Kovatsch. "Efficient Proxying of CoAP Observe with Quality of Service Support". In: *2016 IEEE 3rd World Forum on Internet of Things, WF-IoT 2016*. IEEE, 2017, pp. 401–406. DOI: 10.1109/WF-IoT.2016.7845444.

[200] Fei Tao, Ying Zuo, Li Da Xu, and Lin Zhang. "IoT-Based Intelligent Perception and Access of Manufacturing Resource Toward Cloud Manufacturing". In: *IEEE Trans. Industrial Informatics* 10.2 (2014), pp. 1547–1557. DOI: 10.1109/TII.2014.2306397.

[201] Dinesh Thangavel, Xiaoping Ma, Alvin Valera, Hwee Xian Tan, and Colin Keng Yan Tan. "Performance evaluation of MQTT and CoAP via a common middleware". In: *IEEE ISSNIP 2014 - 2014 IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing, Conference Proceedings* (2014). DOI: 10.1109/ISSNIP.2014.6827678.

[202] The Eclipse Foundation. *Eclipse tinydtls*. https://projects.eclipse.org/projects/iot.tinydtls. 2017.

[203] The Eclipse Foundation. *Scandium (Sc) - Security for Californium*. https://github.com/eclipse/californium/tree/master/scandium-core. 2017.

[204] The New Jersey Cybersecurity & Communications Integration Cell. *Mirai Botnet*. https://www.cyber.nj.gov/threat-profiles/botnet-variants/mirai-botnet. 2016.

[205] Kleanthis Thramboulidis and Foivos Christoulakis. "UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems". In: *Computers in Industry* 82 (2016), pp. 259–272. DOI: 10.1016/j.compind.2016.05.010.

[206] Aparna Saisree Thuluva, Darko Anicic, and Sebastian Rudolph. "Semantic web of things for industry 4.0". In: *CEUR Workshop Proceedings*. Vol. 1875. 2017.

[207] Alessandro Ludovici. *TinyCoAP*. https://github.com/AleLudovici/TinyCoAP. 2013.

[208] Juha-Pekka Tolvanen. "Industrial experiences on using dsls in embedded software development". In: *Proceedings of the Embedded Software Engineering Kongress*. 2011.

[209] Pimoroni. *Controlling IKEA Trådfri Lights from your Pi*. https://learn.pimoroni.com/tutorial/sandyj/controlling-ikea-tradfri-lights-from-your-pi. 2018.

[210] *Transmission Control Protocol*. RFC 793. Sept. 1981. URL: https://rfc-editor.org/rfc/rfc793.txt.

[211] *User Datagram Protocol*. RFC 768. Aug. 1980. URL: `https://rfc-editor.org/rfc/rfc768.txt`.

[212] Muhammad Usman, Muhammad Zohaib Iqbal, and Muhammad Uzair Khan. "A product-line model-driven engineering approach for generating feature-based mobile applications". In: *Journal of Systems and Software* 123 (2017), pp. 1–32. DOI: `10.1016/j.jss.2016.09.049`.

[213] Frank J Van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 3540714367.

[214] Arie Van Deursen, Paul Klint, and Joost Visser. "Domain-specific languages: An annotated bibliography". In: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36. DOI: `10.1145/352029.352035`.

[215] Berta Carballido Villaverde, Dirk Pesch, Rodolfo De Paz Alberola, Szymon Fedor, and Menouer Boubekeur. "Constrained application protocol for low power embedded networks: A survey". In: *Proceedings - 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2012*. 2012, pp. 702–707. DOI: `10.1109/IMIS.2012.93`.

[216] W3C. *Web of Things at W3C*. `https://www.w3.org/WoT/`. 2019.

[217] Arne Wall, Hannes Raddatz, Bala Vikram Reddy Gopu, and Dirk Timmermann. "Evaluation of CoAP Implementations for Live Streaming using CoAP-Observe". In: 2019, pp. 468–473. DOI: `10.1109/WF-IoT.2019.8767189`.

[218] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, 2010. ISBN: 9350231166.

[219] *Websockets*. URL: `https://www.websocket.org/`.

[220] Mark Weiser. "The computer for the 21st century". In: Scientific American Ubicomp, 1991.

[221] WolfSSL. *WolfSSL*. `https://www.wolfssl.com/`. 2017.

[222] Martin Wollschlaeger, Thilo Sauter, and Juergen Jasperneite. "The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0". In: *IEEE industrial electronics magazine* 11.1 (2017), pp. 17–27. DOI: `10.1109/MIE.2017.2649104`.

[223] *XMPP*. URL: https://xmpp.org/.

[224] Bobbi Young, Judd Cheatwood, Todd Peterson, Rick Flores, and Paul C. Clements. "Product Line Engineering Meets Model Based Engineering in the Defense and Automotive Industries". In: *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A*. 2017, pp. 175–179. DOI: 10.1145/3106195.3106220.

[225] Yiying Zhang, Hui Li, Dequan Gao, and Jinping Cao. "The applications of sensor network for smart substation". In: *Advanced Materials Research* 712-715 (2013), pp. 1872–1875. DOI: 10.4028/www.scientific.net/AMR.712-715.1872.

Declaration

I herewith declare that I have produced this work without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This work has not previously been presented in identical or similar form to any examination board.

The dissertation work was conducted from 2016 to 2019 under the supervision of Dr. Aitor Urbieta Arteche and Dr. Diego Casado Mansilla at Ikerlan and the University of Deusto.

Vitoria-Gasteiz,

Markel Iglesias Urkia

This dissertation was finished writing in
Vitoria-Gasteiz on Thursday 9th February, 2020