eman ta zabal zazu

Universidad del País Vasco — Euskal Herriko Unibertsitatea

Department of Computer Architecture and Technology

A Context- and Template-Based Data Compression Approach to Improve Resource-Constrained IoT Systems Interoperability

Dissertation

for the degree of Doctor of Philosophy

Candidate

Jorge Berzosa

Supervisors

Roberto Cortiñas - Luis Gardeazabal

2019

Abstract

The Internet of Things (IoT) has emerged as a powerful paradigm with a huge range of possibilities, promoting its adoption across multiple technological domains. Roughly speaking, the IoT abstraction aims to interconnect every kind of *things*, like simple devices (a light bulb or a thermostat) or more complex and abstract systems, e.g., facility management. Behind these *things*, there are physical devices tasked with specific sensing or actuation roles. Similarly to the *things* themselves, these devices often have significant differences among them in terms of capabilities and the set of communication technologies they use. This heterogeneity leads to an integration challenge regarding interoperability at the connectivity level, including data representation.

A common approach to deal with interoperability in IoT systems is to re-use Internet mature technologies and approaches. At data level, this can be implemented by structuring data following a standard data model and using text-based data formats, e.g., XML. However, the type of devices usually deployed in IoT systems has limited capabilities as well as scarce processing and communication resources. Due to these restrictions, text-based data formats cannot be integrated into resource-constrained devices and networks in an easy and efficient way. Consequently, these limitations also apply to interoperable technologies that rely on text-based data formats such as Web Services.

In this Thesis, we present a novel data compression approach for text-based data formats, namely *Context- and Template-based Compression* (CTC), which is specially designed taking into account the limitations of resource-constrained devices and networks. CTC enhances data-level interoperability in IoT systems while keeping very low resource requirements (in terms of communication bandwidth, memory size and processing power). This Thesis also includes the specification of a set of complementary solutions which facilitate the deployment of CTC in IoT networks and its integration into applications targeted to resource-constrained devices.

Interestingly, CTC is designed with interoperability and extensibility in mind so that it can be applied to different data formats. This work also includes the evaluation of the proposed solution for two popular data formats, XML and JSON, both in real and simulated scenarios. Additionally, the results of the evaluations have been compared with some other current data compression approaches, showing that CTC is a suitable candidate for data compression in resource-constrained IoT deployments.

Laburpena

Gauzen Interneta (*the Internet of Things*, IoT) aukera anitz eskaintzen dituen paradigma bihurtu da, bere erabilera domeinu teknologiko desberdinetan sustatuz. IoT-ren helburu nagusia mota askotako *gauzak* elkar konektatzea da: bonbila edo termostatoa bezalako gailu sinpleetatik hasita, etxea edo makina moduko elementu abstraktu eta konplexuagoraino iritsiz . *Gauza* hauen atzean, sentsore eta aktuazio ardura espezifikok dituzten gailu fisikoak daude. *Gauzekin* gertatzen den bezala, gailu hauek elkarrengan oso ezberdinak dira, bereziki gaitasunei eta erabiltzen dituzten teknologiei dagokienez. Heterogeneotasun honek konektagarritasun mailako elkarreragingarritasunean integrazioa erronka handia aurkezten du, datuen errepresentazioa barne.

IoT sistemetan, elkarreragingarritasuna, Interneten denbora luzez erabili diren teknologiekin jorratzea ohiko joera da. Planteamendu honek, datuen errepresentazio mailan, datuak eredu estandar bat jarraituz egituratzea eskatzen du, baita testuan oinarritutako datu formatuak erabiltzea ere (esaterako, XML). IoT sistemetan erabili ohi diren gailuak, aldiz, gaitasun mugatuak eta prozesatzeko eta komunikatzeko baliabide urrikoak izan ohi dira. Murrizketa hauek medio, testuan oinarritutako datu formatuak, baliabide murriztuko gailu eta sareetan era erraz eta eraginkor batean integratzea ez da posible. Are gehiago, murrizketa hauek, testuan oinarritutako datu formatuak erabiltzen dituzten bestelako teknologia elkarreragileetara ere hedatzen da, besteak beste, Web Zerbitzuak.

Tesi honetan, testuan oinarritutako datu formatuen konpresiorako soluzio bat aurkezten da, baliabide murriztuko gailu eta sareen mugak aintzat hartuta diseinatua izan dena. Soluzio honek *Context- and Template-based Compression*, CTC, du izena. CTC-ek IoT sistemen datu mailako elkarreragingarritasuna hobetzea du helburu, beharrezko komunikazio banda zabalera, memoria tamaina eta prozesamendu ahalmen erabilera minimizatuz. Are gehiago, tesi honetan, CTC soluzioa IoT sareetan eta baliabide murriztuko gailuetan errazago integratzeko bidean, hainbat soluzio osagarri zehaztu dira.

CTC elkarreragingarritasuna eta hedagarritasuna ardatz izanik diseinatu da, datu formatu ezberdinetan aplikagarria izanik. Proposatutako soluzioa bi datu formaturekin ebaluatu da, XML eta JSON, ingurune simulatu eta errealetan. Ebaluazio hauen emaitzak, datu konpresioa jorratzen duten gaur egungo beste soluzio batzuen emaitzekin konparatu dira. Konparaketa honetan, CTC baliabide murriztuko IoT sistemen datu konpresiorako hautagai ona dela ikusi da.

Resumen

El Internet de las Cosas (*the Internet of Things*, IoT) ha surgido como un paradigma con un amplio número de posibilidades, promoviendo así su adopción en múltiples dominios tecnológicos. El objetivo de IoT es el de interconectar todo tipo de *cosas*, desde dispositivos simples, como una bombilla o un termostato, a elementos más complejos y abstractos como una máquina o una casa. Detrás de estas *cosas* se encuentran dispositivos físicos que desempeñan roles específicos de sensor o activador. De manera similar a las *cosas*, estos dispositivos varían enormemente entre sí, especialmente en las capacidades que poseen y el tipo de tecnologías que utilizan. Esta heterogeneidad genera una gran complejidad en los procesos integración en lo que a la interoperabilidad se refiere, incluyendo la representación de los datos.

Un enfoque habitual para abordar la interoperabilidad en sistemas IoT es el de reutilizar tecnologías de Internet ya consolidadas. Más concretamente, a nivel de representación de los datos, los datos se pueden representar siguiendo un modelo de datos estándar, así como formatos de datos basados en texto, e.g., XML. Sin embargo, normalmente el tipo de dispositivos que se se encuentra en sistemas IoT tiene capacidades limitadas, así como recursos de procesamiento y de comunicación escasos. Debido a estas limitaciones no es posible integrar formatos de datos basados en texto de manera sencilla y eficiente en dispositivos y redes con recursos restringidos. Además, esta situación también aplica a tecnologías interoperables que dependen de formatos de datos basados en texto como, por ejemplo, los Servicios Web.

En esta Tesis, presentamos una novedosa solución que permite una gestión eficiente de los datos mediante la compresión de datos para formatos de datos basados en texto y que está especialmente diseñada teniendo en cuenta las limitaciones de dispositivos y redes con recursos restringidos. Denominamos a esta solución *Context- and Template-based Compression* (CTC). CTC mejora la interoperabilidad a nivel de los datos de los sistemas IoT a la vez que requiere muy pocos recursos en cuanto a ancho de banda de las comunicaciones, tamaño de memoria y capacidad de procesamiento. Esta Tesis también especifica una serie de soluciones complementarias que facilitan el despliegue de CTC en redes IoT, así como su integración en aplicaciones orientadas a dispositivos con recursos restringidos.

Como aspecto destacable, CTC está diseñado con la interoperabilidad y extensibilidad en mente, por lo que puede ser aplicado a diferentes formatos de datos. En este trabajo se presenta la evaluación de la solución propuesta para dos formatos de datos populares, XML y JSON, tanto en entornos reales como en simulados. Además, los resultados de estas evaluaciones se han contrastado con soluciones de compresión de datos actuales. Los resultados muestran que CTC es un candidato válido para la compresión de datos en despliegues IoT con recursos restringidos.

Acknowledgements

A Maite, por su paciencia y apoyo cuando más lo necesitaba. Además, por decirme que desconectara de la tesis de vez en cuando.

A mis padres, porque me dieron el primer empujón que me ha traído hasta aquí y ahora.

A mis hermanos y hermana, por creer en mí aun sabiendo como soy.

Nire zuzendarientzat, dokumentu honi forma eman dioten finketak, hobekuntzak, eta, bereziki, zuzenketa guztiengatik.

Acronyms

6LowPAN	Internet Protocol (IPv6) and Low-power Wireless Personal Area Networks
ACK	Acknowledgement
API	Application Programming Interface
CBOR	Concise Binary Object Representation
CoAP	Constrained Application Protocol
CoRE	Constrained RESTful Environments
CPS	Cyber Physical System
CPU	Central processing unit
CTC	Context- and Template-based Compression
DNS	Domain Name System
DOM	Document Object Model
DPWS	Devices Profile for Web Services
DTD	Document Type Definition
DTLS	Datagram Transport Layer Security
EXI	Efficient XML Interchange
EXIP	EXIProcessor
HTML	Hyper Text Markup Language
HTTP	HyperText Transfer Protocol
IANA	Internet Assigned Numbers Authority
IDL	Interactive Data Language
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
INF	Infinity
IoT	Internet of Things
IP	Internet Protocol

IPSec Internet Protocol Security

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

JSON JavaScript Object Notation

LPWPAN Low-power Wireless Personal Area Networks

M2M Machine to Machine

MAC Medium Access Control

MCU MicroController Unit

MQTT Message Queuing Telemetry Transport)

NaN Not a Number

OGC Open Geospatial Consortium

OS Operating System

OSI Open Systems Interconnection

PC Personal Computer

RAM Random Access Memory

RDF Resource Description Framework

REST Representational State Transfer

RFC Request for Comments

ROM Read Only Memory

RPC Remote Procedure Call

SASL Simple Authentication and Security Layer

SOAP Simple Object Access Protocol

SOS Sensor Observation Service

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

URI Uniform Resource Identifier

URL Uniform Resource Locator

UTF-8 8-bit Unicode Transformation Format

W3C World Wide Web Consortium

WoT Web of Things

WS Web Service

WSDL Web Service Definition Language

WSN Wireless Sensor Network

XML eXtensible Markup Language

XMPP Extensible Messaging and Presence Protocol

XSF XMPP Standards Foundation

Contents

List of Figures

List of Tables

1 Introduction

This introductory chapter aims at setting the motivation of the Thesis and to establish its objectives. In this work, we present *Context- and Template-based Compression* (CTC), a compression approach for structured data represented in text-based data formats. CTC addresses the IoT interoperability problem at the data representation level and is specially targeted to resource-constrained devices and networks, typically deployed in IoT systems.

In this introductory chapter, we start in Section 1.1 with a brief explanation of the need of efficient data compression technologies for resource-constrained devices and networks in order to improve IoT interoperability. Next, in Section 1.2, we describe the core objectives of this Thesis as well as the main contributions. Finally, in Section 1.3, we outline the structure of the Thesis document.

1.1 Motivation

The Internet of Things (IoT) leads the trend to integrate almost any kind of devices (*Things*) into a global network (the Internet). In this scenario, interoperability has become one of the main integration challenges. On the one hand, different APIs and protocols are required at communication and application levels in order to interconnect a wide range of different devices and systems. On the other hand, data shows even more diversity regarding semantics and structure, resulting in complex data (pre)processing procedures (translation, filtering, aggregation, storing, etc.) and flows.

A promising solution is to leverage already available Internet technologies and standards, and apply them on top of IoT deployments. This involves the integration of Internet mature technologies directly on, or as close as possible to, the devices. At data level, interoperability is addressed by structuring the data following an agreed data model and representing it in platform-independent data formats which are usually text-based, such as XML or JSON.

However, this approach requires the technologies to be implemented either in the lower architectural layers (close to the physical devices) or in intermediary layers (such as gateways or middle-wares). In the latter case, the deployment of specific purpose intermediaries only moves the complexity related to the interoperability, without removing (or reducing) it while, in the former case, devices typically deployed in IoT systems may not have enough resources or capabilities to natively implement the technologies. These resource-constrained devices have very different characteristics of those commonly found through the Internet. The most important ones are the limited resources of the devices and the low bandwidth of the communication channel.

Due to these restrictions, the majority of the technologies that make the Internet a success cannot be directly integrated into resource-constrained devices and networks. In the specific case of data level interoperability, text-based data formats such as XML or JSON cannot be efficiently used in resource-constrained devices and networks due to limitations in processing power, memory size, transmission bandwidth and available energy. By extension, this is also the case for Internet technologies that rely on text-based data formats such as Web Services, which are one of the key technologies on today's interoperable Internet.

A viable solution for the use of verbose data formats on resource-constrained devices and networks is to compress structured data using a more efficient encoding. However, reducing data size effectively reduces the memory and communication resources (mainly bandwidth) needed but incurs in a processing overhead that may be beyond resource-constrained devices' capabilities. Furthermore, if the data format is modified to ad-hoc encodings the benefits of text-based formats and interoperability with the original format may be lost, if not handled properly. In order to apply data compression technologies to resource-constrained devices and networks to raise interoperability, technologies must be efficiently integrated within the limited resources and capabilities while keeping backwards and seamless compatibility with the original format.

Interoperability has been key in the extension and adoption of Internet technologies and it is expected to be equally (or more) important in the integration of IoT systems. the work presented in this Thesis provides a solution for data level interoperability targeted to resource-constrained devices and networks typically found in IoT deployments and represents a step towards an interoperable IoT.

1.2 Contributions

This thesis aims to address the paradigm of efficient compression technologies for structured data targeted to resource-constrained devices in the following context:

- The IoT suffers from a generalized interoperability lack problem.
- There are several standard and mature technologies used across the Internet but they are not directly usable on resource-constrained devices due to processing, storing and communication channel limitations.

- Specifically, the adaptation of standard data representation formats would significantly increase interoperability within the IoT as well as become an enabler for approaches such as Web Services.

- Data compression technologies can help in the use of text-based data formats in resource-constrained devices but current data compression technologies a) are designed for specific formats, b) are not easily integrable in a seamlessly way and c) impose constrains that may not be meet by resource-constrained devices and networks.

In this thesis, we present *Context- and Template-based Compression* (CTC), a compression approach for structured data represented in text-based data formats which allows resource-constrained devices to deal with high level, standard information data-formats, in pursuit of the seamless integration of information. Roughly speaking, templates are extracted from data model schemas so that their representation can be replaced in the data with a minimum number of references. Data are then compressed (by using lossless-compression) following an algorithm that takes into account the context(s) and templates derived from the original data format and schema.

This Thesis considers the representation of data, which can be done using many different formats, as well as the impact of the data format on processing time, memory usage and data transmission performance. First, by codifying the data in a more efficient format, the processing performance is increased. Second, the use of templates minimizes the memory needed to store schema information and the data structures. Third, data is codified in a more compact format, effectively reducing the message quantity needed to transmit the whole data.

In this work we focus on text-based data formats. More precisely, we use W3C's XML (Extensible Markup Language) and JSON (JavaScript Object Notation). XML and JSON are widely extended and are the basis for many application layer protocols, Web Services and related protocols. Although the work presented here uses XML and JSON as illustrative examples, CTC could be easily extended to other formats.

This document also specifies the communication model of CTC, including generic mechanisms for the management of the contextual information required by CTC. Basically this implies gathering, identifying and referencing schema related information in an efficient way. CTC builds these functionalities based on the concept of a *schema repository* which is a network component responsible of gathering, storing and managing schema information as well as assign and maintain references (i.e., links) and identification information. Although the *schema repository* was originally designed for CTC, it is generic enough to be applied to other data compression technologies based on contextual information. The *schema repository* concept is designed in a generic way, independent from any underlying communication protocol.

Table 1.1: Summary of Scientific and Technological contributions and related publications.

Scientific and Technological Contribution	Chapter(s)	Publication(s)
CTC Specification	4	[BGC17, MGC16]
CTC Communication Model	5	[BGC18]
CTC Software Framework	6	[BGC17]

The core work performed in this Thesis is based on the following hypothesis:

The proposed generic data model description structure and compression approach keep backward compatibility and interoperability while providing a more memory, processing and transmission efficient solution compared to the original data format.

In order to validate the hypothesis described above, this Thesis has the following main scientific and technological objectives:

Objective-1 Study and analyse that, through the use of templates and context information, it is possible to seamlessly use standard data models and data formats in an interoperable and backwards compatible way.

Objective-2 Design and develop a generic data model description structure not tied to a specific data model or data format.

Objective-3 Design and develop data compression mechanisms and algorithms that support the generic representation of structured data and that enable an efficient implementation in terms of processing time, memory usage and transmission bandwidth, tailored to resource-constrained devices and networks.

Objective-4 Develop generic context information transference mechanisms suitable to resource-constrained devices and networks.

The most relevant scientific and technological contributions of this Thesis are listed in Table 1.1, together with the chapters and publications in which they are addressed.

1.3 Document Structure

This introductory chapter is followed by a more in deep description in Chapter 2 of this Thesis' motivation and scope. The chapter includes a description of the scope of this Thesis within the IoT domain focused on interoperability as well as the role of structured data formats and data compression technologies within the context of resource-constrained devices and networks.

The motivation chapter is complemented with Chapter 3 which contains the related work for this Thesis. This related work chapter is divided in three main sections: a brief introduction to text-based data formats including structural concepts and terms, an in-deep state-of-the-art description of structured data compression technologies and an overview of the most relevant communication protocols within the IoT domain for this Thesis.

The core technical content of this Thesis starts in Chapter 4 with the specification of Context- and Template-based Compression approach. This chapter includes the description of the core components of CTC, the codification algorithm and two practical applications to XML and JSON data formats. The CTC specification is followed by Chapter 5 where the CTC communication model, how it fits within a typical IoT system, and the complementary mechanisms needed to be effectively used are explained. The technical content of this Thesis finishes with Chapter 6 and the description of the software framework that implements CTC and all the developed support tools.

Chapter 7 contains all the empirical tests performed during the development of the Thesis in order to evaluate and verify the work done and the objectives fulfilment. The evaluation chapter is followed by the conclusions in Chapter 8. This chapter includes a wrap-up of this document, the main conclusions and the future work.

2 Motivation Background

The purpose of this chapter is to establish in more detail the motivation and scope of this Thesis. First, in Section 2.1 we describe the scope of this Thesis within the IoT and CPS domains and why interoperability is one of the main challenges of IoT deployments. We continue in Section 2.2 by describing the current approaches in the IoT to reduce the interoperability gap and their implications. Next, in Section 2.3, we introduce the application of structured data formats in an interoperable IoT followed by a brief introduction in Section 2.4 to compression technologies for enabling structured data representation formats within the context of resource-constrained devices. Finally, Section 2.5, presents a summary and the conclusions of this chapter.

2.1 The Internet of Things and Cyber-Physical Systems

This section briefly describes an discuss the Internet of Things (IoT) and Cyber-Physical Systems (CPS) domains as well as outline the main paradigms relevant to this Thesis. The purpose is not to formulate a definite definition of the IoT and CPS concepts. However, there is a general confusion between the terms IoT and CPS, sometimes used interchangeably. The aim of this section is to explain what this Thesis considers as IoT and CPS, why they are relevant and what paradigms are addressed by this Thesis.

The IoT could be described as a network formed by interconnected "things" that are uniquely identifiable. The network can be formed by heterogeneous elements, from simple sensors to full featured computers. However, the term *Thing* has been adopted because the elements of the network can also represent more generic and/or complex concepts such as a car, a person or a place. Thus, each *Thing* can vary greatly in complexity, capabilities and purpose, forming cooperation patterns and clusters with other *Things*, and creating a final application from the sum of all the added functionalities.

The IoT is often associated with Machine-to-machine (M2M) communications. M2M is traditionally seen as the communication of two (or more) elements over an isolated network where applications do not interact with the outside. In contrast, the IoT is seen as distributed heterogeneous applications (sensing, actuation, etc.) producing and consuming data and being globally accessible to third party applications. The IoT relies on the idea that a globally interconnected network of information will provide more added value than isolated (sub-)networks.

From an infrastructure point of view, the IoT provides a communication architecture that links data producers and consumers of a highly heterogeneous nature, regarding various aspects such as communication bandwidth, processing capabilities, role, autonomy, smartness, etc.

This was not easily achievable before the upcoming of IPv6 [DH17] due to the complexity of addressing and escalating an increasing number of interconnected *Things*. IPv6 has brought a significant increment on the IP address range compared to IPv4. Specifically, the address range covers 2^{128} unique addresses, approximately 6.65×10^{23} per square meter of the Earth [Hag06]. These new address range provides more than enough addresses to uniquely identify each and every one of the *Things* that can be reasonably expected.

Cyber-physical Systems (CPS) are described as the integration of physical objects and environments together with their virtual counterparts (sometimes referred as "digital twin"). CPS are often build on complex architectures and synergies which are composed by physical devices (sensors, actuators, routers, etc.), a multi-layered communication infrastructure and middle-ware, and high level applications that model the cyber-representation as well as implement the management of the physical entities. Examples of CPS include Smart Cities and Smart Factories, which may be composed by (smart) buildings, (smart) homes, inhabitants, workers, (smart) waste collection systems, (smart) equipment, (smart) machinery, etc.

This Thesis sees the IoT as an integral part and an enabling technology of CPS. The IoT serves as the natural link between the physical and cyber/virtual world of CPS, as is represented in Figure 2.1.

The IoT approach, and hence CPS, imposes several challenges that derive from the distributed, heterogeneous and large-scale nature of typical IoT systems. These challenges range from the efficient management and scalability of a massive number of data points (processing, timing, storage, etc.) to interoperability in the communication protocols, application interfaces and data.

For instance, one of the domains that is gaining attention rapidly is the IoT security and privacy. The increased number of connected devices and infrastructure grow as well as its application to a wider range of application domains, has proportionally increased the appealing and risk of intentional attacks. Additionally, putting private data "on the net" and making it globally/remotely available, also increased the risk of stolen data as well as raised governmental and legal management concerns regarding data privacy.

Nevertheless, although the integration of added-value capabilities such as security is an acknowledged need, interoperability is raising as one of the mayor integration

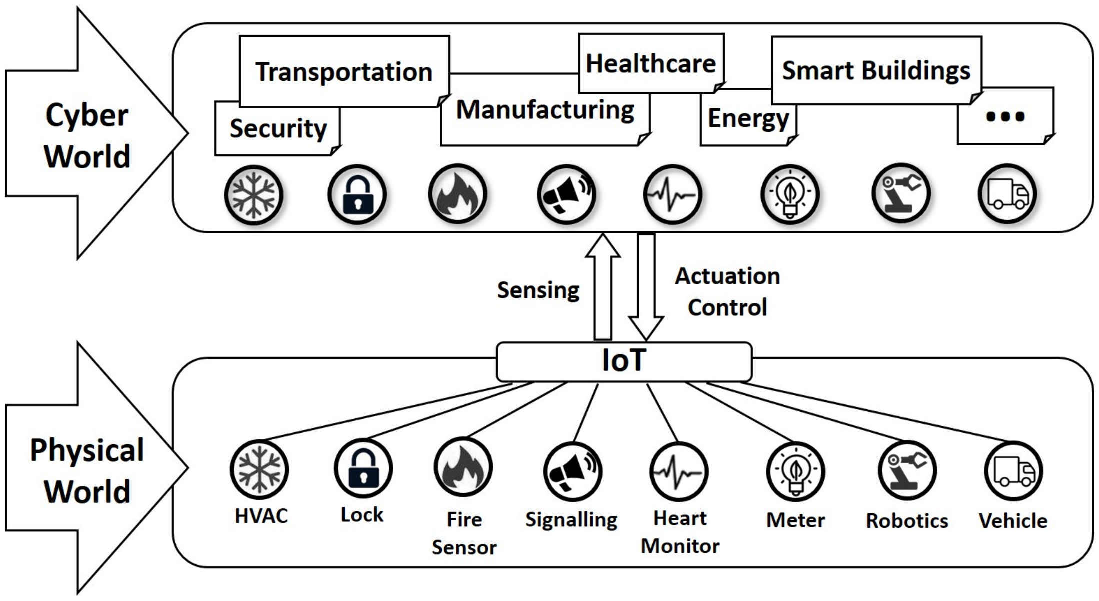


Figure 2.1: The IoT and CPS relationship.

challenges as the number of IoT enabled devices and applications grows rapidly. Interoperability challenges are mainly gathered within three domains: applications, communications and data. On the one hand, applications and communication channels require standard APIs and protocols in order to handle the increasingly heterogeneous devices and systems. On the other hand, data diversity regarding semantics and structure entail complex formatting procedures and complicate the creation of homogeneous and seamless data processing solutions.

This situation enforces the deployment of specific purpose gateways, proxies and middle-wares with multi-protocol interfaces and data preprocessing capabilities in order to provide the APIs and data formats (which sometimes do not even follow a standard) required by applications. Thus, although the IoT pretends to overcome the limitations associated to M2M systems, in practice, interoperability clashes often turn IoT networks into isolated networks leading to fragmented applications.

2.2 Towards IoT Interoperability

One of the main barriers to overcome for today's IoT is that there is no easy way to globally and seamlessly connect all the existing (and forthcoming) *Things*. In the current IoT landscape several incompatible technologies are being deployed. Actually, despite the *raison d'être* of the IoT, nowadays IoT systems often result in a group of isolated intra-nets that can not directly interact across the several heterogeneous networking interfaces that exist. In this scenario, the connection and integration of services, interfaces and data from several *Things* is very costly and complex. Thus, it is essential to remove the interoperability gap and provide universal standards and mechanisms for the *Things* to interact between them and the overall ecosystem.

There are various approaches to tackle interoperability issues in the IoT. An interesting approach is the one proposed by the Web of Things (WoT) [GTMW11, KKD18] which purpose is to overcome the IoT fragmentation and promote platform-independent standards. Compared to the IoT, the WoT is a more current trend which takes the IoT paradigm one step forward. The WoT concept was first develop by the WebofThings.org community [wc] which defines it as "... a refinement of the Internet of Things by integrating smart things not only into the Internet (network), but into the Web Architecture (application)". The main purpose of WoT is to (re-)use and leverage already available Web protocols and standards, and apply them on top of the IoT paradigm in order to remove the existing interoperability gap whether it applies to data, communication protocols, services or transversal technologies (such as security or discovery).

WoT is totally based on application layer protocols (from a network OSI layer point of view), in contrast to the IoT which is usually more focused on network and transport layers. The purpose of focusing on the application layer is to effectively abstract lower layers (e.g., physical or transport layers), which are the ones showing the highest degree of heterogeneity and main source of interoperability clashes in traditional IoT networks.

Although WoT only adds functionality at the application layer, it defines an architecture to organize the several existing Web technologies into a standard and functional structure. The WoT architecture stack is further divided into four layers. Each of these layers adds a higher level of functionality, briefly described in the list below.

- **Access Layer**: the first layer is the responsible of providing a standard Web interface, such as a HTTP-REST API.
- **Find Layer**: the second layer includes semantic Web standards to describe *Things* (and their services) in order to provide standard mechanisms to automatically (i.e. without human intervention) find them.
- **Share Layer**: the main purpose of the third layer is to secure the data interchanged between the *Things*.
- **Compose Layer**: the final layer includes the tools and frameworks to integrate and build applications on top of the heterogeneous data and services provided by the *Things*.

The WoT architecture promotes generic/standard Web interfaces in order to enhance overall interoperability and build loosely coupled services by providing mechanisms for highly configurable services/interactions. The WoT architecture includes Web scripting languages such as javascript, data encodings such as JSON [Bra14] and EXI [SKPK14], and Web protocols such as HTTP [FR14] and WebSockets [MF11]. In summary the aim of the

WoT architecture is to bring Web technologies to the IoT and add a standardization layer on top of the traditional IoT.

WoT has also attracted the interest of the W3C, which created the Web of Things Working Group [W3C]. The W3C's Web of Things Working Group focuses on platform independent APIs and discovery procedures for inter-platform operations in order to overcome the lack of interoperability across IoT systems. The group is mainly working on descriptive metadata and interaction models as well as communication and security requirements.

The W3C's Web of Things Working Group released the Web Thing Model [TGC17] which describes a common model and Web API for the "virtual counterpart of physical objects in the Web of Things". The aim is to provide an interoperable model and protocols to access and interact with *Things* using Web standards. The document contains three main sections:

- The first section (*Integration Patterns*) proposes three patterns to connect and integrate *Things* with the Web.
- The second section (*Web Things Requirements*) provides domain agnostic constrains and recommendations about protocol implementations for the WoT.
- The final section (*Web Things Model*) specifies a RESTful Web protocol together with the resource types, data models and payload syntax that *Things* should implement.

The *Integration Patterns* proposal contemplates three distinct patterns depending on the capabilities and architecture of the network. In the *Direct Connectivity Pattern*, clients send requests directly to the *Thing*'s API, whether they both are on the same or different network. This is the simplest pattern because it avoids the deployment of intermediaries. If the *Thing* is not able to provide a Web API, the *Gateway-Based Connectivity Pattern* is used. In this integration pattern an intermediate element (a gateway or proxy) provides the Web API on behalf of the *Thing* and makes the proper translations/adaptations. Sometimes it is more convenient to deploy a cloud service to act on behalf of the *Thing* (to provide the Web API), for instance, when a *Thing* has to be globally accessible. This is the third pattern and is called the *Cloud-Based Connectivity Pattern*.

As can be extrapolated from these integration patterns, one of the methods to overcome the lack of interoperability across networks is to directly implement Web technologies, tools and methods on the *Things*. However, the majority of the devices used to implement the core of the *Things* tend to have very scarce resources due to technical, economical and practical reasons. These resource-constrained devices are designed with low memory capacity (<256KBytes Flash/ROM and a few KBytes of RAM), limited processing capabilities (<48MHz, typically 8-16MHz) and an average consumption of a few

μA due to energy source limitations and autonomy requirements. Thus, it is usually not possible to integrate Web technologies directly on the resource-constrained devices. In these cases, an intermediary such as a physical gateway/proxy or software middle-wares, is used. The intermediary will provide an abstraction layer on top of the physical and transport layers as well as data encodings and formats of the underlying *Thing(s)*.

Regardless of the integration pattern used, ideally, Web technologies should be pushed as close to the *Thing* as possible. In this way, less specific translation layers and intermediaries would be necessary in order to achieve communication, data and application level interoperability. Failing that, using intermediaries to leverage domain specific protocols and translate them to Web technologies is a viable option but, in truth, it only moves the complexity to the intermediaries. Either domain/application/protocol specific intermediaries are deployed, or general intermediaries that must implement the myriads of protocols and standards should be used. In order to minimize the impact and complexity of the intermediaries, they should try to avoid specific technologies as much as possible and focus on general purpose technologies that are able to seamlessly adapt to the technologies used across network boundaries. This approach is not only applicable to the WoT philosophy, but to similar approaches (such as the one followed by Open Connectivity Foundation [OCF]) or IoT architectures in general if interoperability has to be supported.

2.3 Structured Data and Interoperability

When two (or more) distributed applications communicate between them, they need to use the same protocol and data “language”. This is mandatory so each application is able to decode and interpret the information sent by other applications. At data level this is achieved by structuring the data following an agreed data model. A data model defines the concepts used to describe the data and the semantic relationships between them (or concepts outside the data model). In turn, data models are implemented using data formats that capture the data model’s structure and vocabulary (the terms defined within the data model). A data format defines how the concepts are syntactically represented. Data models can be more or less generic (such as the one specified by SenML [JSA+18] standard which is used to describe the capabilities of generic sensors) or for specific applications (such as the model described by GeoJSON [BDD+16], for location information).

Structured data formats specify two levels of structural information. The first one is the structure and grammar of the format itself, i.e. how elements and their relationships are described. The second level is related to the data that is being formatted and contains the structure and terms of the data model, sometimes referred as vocabulary or language. Additionally, structured data formats are not only used to describe data that follows a given data model, but also to implement protocols (that, actually, also follow a particular data model).

Until the general adoption of text-based data representation formats, data were interchanged over the Internet following ad-hoc binary formats. These binary formats were hard to maintain and any change or update in the structure of the format could create parsing and interpretation mismatches between the communicating elements.

Nowadays, binary formats have been mainly dropped (except in the most resource-constrained scenarios) in favour of formats such as XML or JSON, which are text based. The main advantage of text-based data formats is that they allow the representation of information in an interoperable way by setting a clear separation between data itself and how it is presented. Data can be interchanged between different machines, systems and entities with all the necessary information for the processing and interpretation of the data contained in the data itself. Additionally, rules can be set for validation and pruning of wrongly structured data.

One of the Internet technologies that more relies on structured data formats is Web Services (WS). Interoperability is one of the core design principles of WS; they allow the direct interaction between different types of platforms, software and users (human or machine). They are based on the simple client/server model and are used in all kind of applications. Many widely used high level protocols, such as DPWS [CCK+06], SOS [BSE12] or SensorThingsAPI [LHK16], are built on top of WS.

Apart from enhancing overall interoperability, WS enable the application of high level services, such as the “self-*” services family (where “self-*” stands for self-discovery, self-configuring, etc.), directly on top of the devices.

WSs can be roughly divided into two types, depending on the implementation approach:

- **Based on services.** These type of WSs are mainly based on Remote Procedure Calls (RPCs) to access the services offered by a server. One of the most popular technologies used to implement RPCs is Simple Object Transfer Protocol (SOAP [12a]). The description and transport of SOAP is made by means of XML files.
- **Based on resources.** The data and services are represented as resources and they are accessed by means of Representational Transfer State (REST [Fie00]) approach. REST sees everything as an HTTP resource although the resource itself is typically described in HTML, XML or JSON.

As can be seen, WS need to rely on text-based data formats at some point, either for formatting a protocol, the data itself, or both. However, traditional text-based data format management technologies are unsuitable for their application in resource-constrained devices and networks. This is why binary formats are usually preferred or needed in resource-constrained networks. The price to pay is interoperability and the need for specific purpose intermediaries, though. These intermediaries are usually gateways or proxies that are needed in order to ensure interoperability across networks. This is, for instance, the strategy followed by WoT.

Interoperability has been a key enabler capability of the Internet and, as has been explained in the previous sections, it is key for a successful deployment and integration of the IoT and its associated technologies. Developing efficient and reliable technologies for data formats targeted to resource-constrained devices and networks would be a large step towards an interoperable IoT.

2.4 Compression of Structured Data

An effective technique used to reduce the size of the data is to compress the data itself using a more efficient encoding. Generic compressed data formats have been around for several years and are widely used in everyday technologies. These formats can be based on lossless compression algorithms (such as GZIP [lGA] which is based on DEFLATE [Deu96]) or lossy algorithms which are mainly used for audiovisual file formats (such as JPEG [Ham92] for images or MP3 [ISO93] for audio) where a controlled loss of information does not significantly affect the overall perceived result.

Generic compressed data formats can be applied to data without any assumption of structure or semantics. In contrast, compression techniques for structured data can directly take advantage from both, the structure of the original data format and the structure of the data model, and produce a more compact and efficient encoding of the data. These kind of technologies can also encapsulate and derive further context information from the original data format, such as data types or the semantics of a specific token.

Compression technologies for structured data are primarily based on information extracted from the structure itself and they make use of this knowledge to efficiently compress data streams. These technologies take advantage of the formal structure and grammar specification followed by the data format as well as the structure followed by the data model, usually described by means of a schema.

For instance, current compression technologies based on structured data are EXI, CBOR and Protocol Buffers. EXI [SKPK14], adopted as a recommendation by W3C, has emerged in the recent years as the most prominent XML compression algorithm. CBOR (Concise Binary Object Representation) [BH13] is a compact data format based on the JSON data model optimized for simplicity, processing speed, minimum resource usage and implementation compactness. Protocol Buffers [18e] is Google's proposal for structured data serialization. The structure of the data is described as an IDL document that acts as the data schema. This IDL document is pre-compiled in order to produce the code stubs that are used to marshal/un-marshal the coded streams to/from runtime objects. A more in deep description of EXI, CBOR and Protocol Buffers is provided in Section 3.2.

Compression technologies for structured data require the use of schema or structure information in order to achieve the most efficient compression. Although these technologies can compress data without schema information (i.e., solely relying on the data format structure), the most efficient compression is achieved when the schema is also available. However, how this information is shared and referenced, is usually not specified or it is assumed that an out of band method is used.

Additionally, the entities that interchange the compressed data require the identification of the schema (if any) that has been used as the base for the data compression. This identification has to be either advertised at runtime or agreed beforehand between all the parties.

However, compressing data only solves part of the problem. Reducing data size effectively reduces communication resources but the compression process itself produces an overhead that may be beyond resource-constrained devices' capabilities. Additionally, compressing data implies changing the data format in which the data is encoded. This can raise again an interoperability issue and loss all the benefits of text based formats, if not done properly. Ideally, the compression/decompression should be performed in a seamless and transparent way.

2.5 Conclusions

Currently, the IoT suffers from a generalized interoperability problem mainly produced by the increasingly number of heterogeneous applications, protocols, data and devices that are being deployed. This heterogeneity is specially notable in the semantics, formats and structures followed by the produced and consumed data.

Resource-constrained devices and networks usually found in IoT deployments have very different characteristics of those commonly found through the Internet. Notably, the capabilities of the devices are severely limited and communication channels are based on low bandwidth technologies. These restrictions complicate the integration of solutions and applications readily available on the Internet because they require significantly more resources than those offered by resource-constrained devices and networks.

This also applies to text-based data representation formats. Resource-constrained devices cannot process formats such as XML or JSON in an efficient manner (regarding processing time) and will put to test resources such as energy or memory. Additionally, using these formats has an impact on the network bandwidth needed which, again, has an impact on the devices' energy and network load.

Using compression technologies for structured data results in more compact encodings than text-based formats. However, compression technologies also produce a processing overhead that may be beyond resource-constrained devices' capabilities. Additionally, if the format is modified, the benefits of text-based formats and interoperability with the original format may be lost, if it is not handled properly. If compression technologies are to be used in resource-constrained devices and networks to raise interoperability, they must be efficiently executed within the limited resources of the devices while keeping backwards compatibility with the original format.

This thesis aims to address the paradigm of efficient compression technologies for structured data targeted to resource-constrained devices in order to leverage existing standard data formats and enhance IoT interoperability at data level.

3 Related Work

The objective of this chapter is to give the reader an insight of the technologies, researches and developments related to this Thesis. The chapter is divided into three main sections.

Section 3.1 gives a brief introduction to two text-based data formats that dominate the Internet, XML and JSON. These two data formats are used through this work as application examples. The section includes information about structural concepts and terms used across the document as well as relevant characteristics.

The next section, Section 3.2, describes state-of-the-art compression technologies or approaches targeted to structured data, including standards and open researches. For completeness reasons, Section 3.2 also includes protocols based on structured data and/or text-based data formats. The presented approaches are analysed within the context of resource-constrained devices and networks.

Finally, Section 3.3 contains an overview of the most relevant IoT communication protocols for this Thesis. This section introduces concepts and approaches of IoT communication protocols addressed in the technical chapters of this document (specially in Chapter 5) as well as briefly discuss their relevance and applicability.

3.1 Text-Based Data Formats

Text-based data formats are the preferred way to represent and interchange data over the Internet. Text-based data formats are mainly used for two purposes: to represent data in a structured way and to add metadata. By representing data in a structured and known way, a clear separation is set between data itself and how it is presented. The main advantage is that rules can be set for validation and wrongly structured data can be rejected. By the addition of metadata, the data can be enriched with capabilities such as self-description or high level validation.

The main advantage of Text-based data formats is that they allow the representation of information in an interoperable way. Data can be interchanged between different machines, systems and entities with all the necessary information for the processing and interpretation of the data contained in the data itself.

In this section we focus on the two text-based data formats used for data interchange that dominate the Internet: XML and JSON. These two data formats are also used through this work as relevant application examples. However, the principles and approaches developed within this thesis are applicable to other text-based data formats such as Hypertext Markup Language (HTML [FEL+17]) or the Resource Description Framework (RDF [WWWCc]) set of recommendations.

Nevertheless, this section does not pretend to be an exhaustive description of the XML and JSON specifications but its intent is to give the necessary information to understand the technical development shown in this thesis. Complementary information is available in Appendix A.1 and Appendix A.2.

3.1.1 eXtensible Markup Language (XML)

Among text-based data formats eXtensible Markup Language (XML [BPSM+08]) is one of the most famous ones. XML is a recommendation promoted by W3C and one of their leading technologies.

XML documents are formatted following a tree structure where the main nodes are XML elements. The tree starts with the *root element*, branching through *child elements* and finish in *leave elements*.

Elements represent the base information item of a XML document. There must be at least one element, the *root element*, and any additional item must be a child of this element. Thus all the items in a XML document form a hierarchical tree with the *root element* on the top. Elements can be defined in two ways: with a full start and end tags (<*element_name*> ... </*element_name*>) or with a self-closing tag (<*element_name* .../>).

The other main item of XML documents is attributes. Attributes are name/value pairs in the form "attribute_name=value". Every attribute is bound to an element and they are defined within the element's start or self-closing tag.

Figure 3.1 shows an example of a XML document containing four elements and one attribute. "pet" is the root element and it contains the child elements "name", "age" and "gender" with the values "Calcetines", "1" and "female" respectively. "species" is an attribute of the "pet" element with the value "feline". The tree structure is shown in Figure 3.2.

3.1.1.1 XML Structure Representation

After the appearance of XML, many technologies adopted it and developed their own model abstractions and terminology. This led to some confusion in the XML community and the need for a common model arose. In order to solve this issue, the W3C's released the recommendation XML Information Set [CT04], commonly known as XML Infoset. The

```
<pet species="Cat">
    <name>Calcetines</name>
    <age>1</age>
    <gender>Female</gender>
</pet>
```

Figure 3.1: XML element and attribute example.

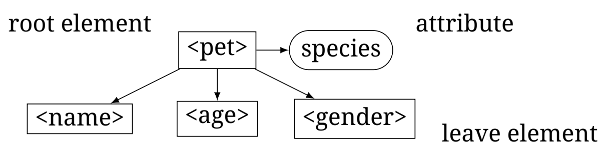


Figure 3.2: XML tree structure example.

main purposes of the XML Infoset is to provide a common terminology and to abstract a XML document into an idealized model.

The XML Infoset defines eleven components, denoted *information items*. Each of the information items contains a set of properties as well as links to the nested items.

- **Document Information Item:** as the name suggest, the Document Information Item provides general information about the XML document as well as contain all the other items in the document. This information includes, for instance, properties set in the *prolog* and the root element.

  The *prolog* is a XML construct gives contextual information to the XML parsers. For instance, the *prolog* shown in Figure 3.3 declares that the XML version used is “1.0” and that the text encoding is “UTF-8”. The *prolog* is optional but if a XML document contains one, it must a appear at the beginning of the document before any other item.

- **Element Information Item:** every element has one associated Element Information Item. The properties of this item contain information related to the element including the name, parent item, children (not only elements but also other items such as comments or processing instructions) and attributes.

- **Attribute Information Item:** every attribute has one associated Attribute Information Item. The properties of this item include the name and the parent element (the element in which this attribute is declared).

```
<?xml version="1.0" encoding="UTF-8"?>
```

Figure 3.3: XML prolog example.

```
<?app param1="123" param2="example"?>
```

Figure 3.4: XML processing instruction example.

- **Processing Instruction Information Item:** the model will contain one Processing Instruction Information Item per processing instruction (PI). The information contained in a PI is not intended for an XML Parser but for the application processing the XML document. This information is usually composed by instructions on how to process the document or to control different behaviours. Basically, a PI has a target that identifies the application as well as data that is used as parameters. For instance, the example in Figure 3.4 shows a PI for the application "app" with the content "param1="123" param2="example""

  Basically, the properties of a Processing Instruction Information Item will contain information about the PI's target and content.

- **Character Information Item:** although in theory the model should contain one Character Information Item per character in the document, in practice characters are grouped in strings and processed as such by applications. Nevertheless, the properties of the Character Information Item include the character code (following the ISO 10646 standard) and the parent item.

  CDATA sections are also included within Character Information Items. CDATA declarations are used to define sections with no XML markup. This sections will be escaped by parsers.

  CDATA sections are defined through a "<![CDATA[*escaped_text*]]>" declaration. The content of a CDATA section is considered simple text and it is not necessary to escape characters reserved by the XML Recommendation This is very useful to avoid repetitive escaping of characters.

- **Comment Information Item:** each item is linked to a comment in the XML document. The properties of this item contain the text of the comment and the parent item.

- **Namespace Information Item:** each item of the document will contain one Namespace Information Item per namespace in scope. XML Namespaces are explained in more detail in the following section.

- **Document Type Declaration Information Item, Unexpanded Entity Reference Information Item, Unparsed Entity Information Item and Notation Information Item:** a model will contain these items only if the source document defines a document type declaration. The properties of these items contain information to manage DTD's associated to the document such as the retrieval of the XML document's DTDs or notations for the inclusion of non-XML content.

XML Infoset provides a standard model to represent and refer to XML components. However, there are other technologies used to represent an in-memory instance of an XML document. The most popular data models for representing, storing, accessing and processing XML documents is W3C's Document Object Model (DOM [WWWCa]).

DOM represents XML documents as a tree-structure where everything is a node: the document itself, elements, attributes, etc. As with other tree-structures, DOM is represented as a graph and the relationships between nodes are described using terms like parent, child, sibling, etc. DOM also specifies a low-level Application Programming Interface (API) for accessing and processing XML documents. The API also allows to modify the nodes at run-time, which in effect, is equivalent to modifying the values of those nodes in the XML document itself.

3.1.1.2 Namespaces

In XML, namespaces are used to group elements and attributes. The name of an element or attribute within a namespace must be unique but it can be reused in another namespace because the namespace itself is used to solve the ambiguity.

Namespaces are specially useful when XML documents are shared or mixed with other XML documents developed by third parties. Namespaces ensure that the names of elements and attributes defined in each XML document format do not clash and XML applications parse then correctly.

Namespaces usually follow the URI [BLFM05] format. It is common practice to use registered domains as part of the namespace in order to assure the uniqueness (as well as provide an evident means for identification) and avoid clashes with other entities' choices.

Namespaces can be declared implicitly or explicitly. To declare a namespace implicitly, the namespace is declared within the element it will be associated with. This is known as declaring a default namespace and it is in scope of the element in which it is declared as well as all the child elements. In this context, being "in scope" means that the namespace is available to be used. As we will se later, a namespace may be in scope but not assigned. On the other hand, attributes are not covered by a default namespace.

Figure 3.5 shows an example of a default namespace declaration. The namespace "`http://example.com/namespaces/pets`" is in scope for the elements "pet", "name", "age" and "gender" and the four elements are associated to it.

In order to declare a namespace explicitly a prefix has to be assigned to it using the following form:

```
<pet xmlns="http://example.com/namespaces/pets" species="Cat">
    <name>Calcetines</name>
    <age>1</age>
    <gender>Female</gender>
</pet>
```

Figure 3.5: XML namespace declaration example.

```
<pe:pet xmlns:pe="http://example.com/namespaces/pets" species="Cat">
    <name>Calcetines</name>
    <age>1</age>
    <gender>Female</gender>
</pe:pet>
```

Figure 3.6: XML namespace prefix declaration example.

xmlns:*prefix*="*namespace_URI*"

The name of the prefix must follow the same naming rules as elements and must not start with the character string “xml”. In the example of Figure 3.6, the four elements “pet”, “name”, “age” and “gender” are within the scope of the namespace “pe” but only the “pet” element belongs to it. When an element or attribute is explicitly associated to a namespace, the prefix forms part of its name and is known as a *Qualified Name* (or QName). The original name (without the prefix) is known as the *Local Name*.

Usually attributes are not explicitly assigned to a namespace. This is because attributes are already associated with an element, thus, if the element belongs to a namespace, the attribute is already uniquely identifiable.

A XML document may contain more than one explicitly declared namespace and they can be mixed with a default namespace. For instance, Figure 3.7, shows a default namespace mixed with an explicit namespace. The element “pet” belongs to the default namespace (i.e. “`http://example.com/namespaces/pets`”) while elements “name”, “age” and “gender” belong to the namespace “`http://example.com/namespaces/info`”.

There are several XML namespaces already defined, the majority of them covering an standard or entity conventions. The following list contains the most common and representative namespaces.

```
<pet
        xmlns="http://example.com/namespaces/pets"
        xmlns:in="http://example.com/namespaces/info"
        species="Cat">
    <in:name>Calcetines</in:name>
    <in:age>1</in:age>
    <in:gender>Female</in:gender>
</pet>
```

Figure 3.7: Multiple XML namespace prefix declaration example.

- **XML Namespace:** the “xml” prefix is always bound to the “`http://www.w3.org/XML/1998/namespace`” URI and its contents are available by default in all XML parsers.
- **XMLNS Namespace:** the prefix “xmlns” is used to declare a prefix for an explicit namespace. The “xmlns” prefix is bound to the “`http://www.w3.org/2000/xmlns/`” URI and hard-coded into all XML parsers.
- **XML Schema Namespace:** the XML Schema Namespace covers the declarations used to define XML Schemas. This namespace is bound to the “`http://www.w3.org/2001/XMLSchema`” URI and is usually associated with prefixes “xs” or “xsd” (although it is not mandatory).
- **SOAP Namespace:** this namespace contains the declarations to define method calls following the client/server paradigm. Actually, there are two versions of the SOAP Namespace: “`http://schemas.xmlsoap.org/soap/envelope/`” URI for SOAP 1.1, usually used with the prefix “soap”, and “`http://www.w3.org/2003/05/soap-envelope`” for SOAP 1.2, usually bound to the prefix “soap12”.
- **WSDL Namespace:** WSDL stands for Web Services Description Language and is used to describe the methods and data structures used in web services. WSDL is closely related to SOAP and the SOAP Namespace is included in the declarations of the WSDL Namespace. The WSDL Namespace is bound to the “`http://www.w3.org/ns/wsdl`” URI and is usually associated with the prefix “wsdl”.

3.1.1.3 XML Schema

In general, a schema is a document that contains a specific model that describes a well-defined structure. XML Schema is the W3C specification [WWWCd] for describing the structure and vocabulary of XML documents. Although there are other specifications, such as Document Type Definition (DTD [WWWCb]) or RELAX NG [Rel], XML Schema is very mature and is widely used in many XML applications/domains as well as be the basis for other XML technologies (e.g., SOAP [12a]).

Usually, an XML document includes a reference to the XML Schema that describes its vocabulary. This XML document is denoted an “instance” of the schema document. The main use for XML Schemas is for document validation, which consist on verifying that the content of an XML document is in conformance with the model and structure described in the associated schema.

The *schema* element is the root element of any XML Schema document. This element provides several attributes to declare information about the overall schema including the namespace it is bound to and version information. The *targetNamespace* attribute

```
<schema
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:pe="http://example.com/namespaces/pets"
        targetNamespace="http://example.com/namespaces/pets"
        elementFormDefault="qualified">
    <element name="pet">
        <complexType>
            <sequence>
                <element name="name" type="string"/>
                <element name="age" type="integer"/>
                <element name="gender" type="string"/>
            </sequence>
            <attribute name="species" type="string"/>
        </complexType>
    </element>
</schema>
```

Figure 3.8: XML Schema simple example.

is used to declare the namespace URI that will identify this XML Schema. Other XML documents will use this URI to reference the schema as a namespace. Figure 3.8 shows an example of a simple XML Schema, containing four elements and one attribute. “pet” is the root element and it contains the child elements “name”, “age” and “gender” as well as an attribute of the “pet” element named “species”.

The Appendix A.1 gives a more detailed description of the XML Schema specification including the most important concepts and components relevant to this thesis.

3.1.2 JavaScript Object Notation (JSON)

JavaScript Object Notation [Bra14], better known as JSON, is a data format born from the JavaScript Programming Language. However, JSON is a text-based format as well as language independent and is not tailored to JavaScript Although being a text-based data format, JSON was designed to be relatively lightweight and easy to parse, at least, compared to XML.

The JSON format structure is based on two constructs: name/value pairs and arrays of values. JSON can also represent four basic types: strings, numbers, booleans and null. In JSON, objects are unordered sequences of name/value pairs separated by a coma (i.e. ‘,’), where the name is always represented as a string and the value is either a string, number, boolean, null, array or another nested object. JSON objects are enclosed between curly braces (i.e. ‘{’ and ‘}’). A JSON array is an ordered sequence of zero or more values separated by a coma (i.e. ‘,’), where values are strings, numbers, booleans, null, or nested arrays and objects. JSON arrays are enclosed between square brackets (i.e. ‘[’ and ‘]’). A simple example of a JSON document is shown in Figure 3.9.

JSON structure components are described in more detail in the following list:

```
{
  "name": "Calcetines",
  "age": 1,
  "gender": "Female",
  "owners": ["Maite","Jorge"]
}
```

Figure 3.9: JSON simple structure example.

- **Literals**: JSON specifies three literals: “true”, “false” and “null”. The first two (“true” and “false”) are used for boolean values while the third one (“null”) is used to represent null values.
- **Numbers**: JSON numbers are represented as a base 10 sequence of decimal digits. JSON numbers can be prefixed with a sign symbol (‘+’ or ‘-’) and may include a fractional part. The fractional part can be either represented with a dot (i.e., ‘.’) separated digits or with an exponent of ten prefixed by an ‘e’ or ‘E’. Notably, JSON does not support numbers that cannot be represented as sequences of digits (such as INF and NaN).
- **Strings**: a JSON string is a sequence of Unicode characters enclosed in quotation marks. A reverse solidus (i.e. ‘\’) is used to escape characters. Characters in the Basic Multilingual Plane (U+0000 through U+FFFF) may be optionally represented as a reverse solidus, followed by the letter ‘u’, followed by the four hexadecimal code point digits. For instance, the letter ‘A’ may be optionally encoded as “\u0041”. Extended characters that are not in the Basic Multilingual Plane, can be represented by combining characters within the Basic Multilingual Plane.
- **Objects**: objects are used to represent structured data composed by one or more fields. JSON objects are represented as a sequence of name/value pairs enclosed by curly braces. Names are represented as quoted strings and are separated by a colon (i.e., ‘:’) from the value. Consecutive value pairs are separated by a coma (i.e., ‘,’).

  The box below shows an example of a JSON object with two subschemas.

  ```
  { "name1": value1, "name2": value2 }
  ```

- **Arrays**: array represent a sequence of values. JSON arrays are represented as a coma separated sequence of zero or more values enclosed between square brackets. The values of an array can be of different types.

  The box below contains a JSON array with three subschemas.

  ```
  [ value1, value2, value3 ]
  ```

```
{
  "title": "root schema",
  "sub": {
    "title": "subschema"
  }
}
```

Figure 3.10: JSON Schema root schema and subschema example.

3.1.2.1 JSON Schema

A JSON Schema defines the structure of JSON data as well as provide information for validation, parsing and interaction. At the time of this writing Draft-04 version is still the more widely used JSON Schema version, compared to more recent ones (currently Draft-07 [WA18, WAL18]). Thus, this thesis focuses on JSON Schema Draft-04. However, the principles described in this thesis are easily extrapolated to more recent JSON Schema versions, such as Draft-07.

The JSON Schema Draft-4 specification is actually composed by three documents.

- JSON Schema core specification [GZC13] describes the core terminology, references to other JSON Schemas and vocabulary definition.
- JSON Schema Validation [ZC13] defines the vocabulary for validation assertions, link navigation and interaction constrains.
- JSON Hyper-Schema specification [LZC13] describe the hypertext structure and management of JSON documents such as resource link relations and multimedia vocabulary.

A JSON Schema is in itself a JSON document but it is used to define the data model followed by other JSON documents, known as “instances”. Thus, a JSON Schema defines the structure and constrains over the same structural components used by JSON documents and listed in the previous section, Section 3.1.2: null, boolean, number, string, object and array.

A JSON Schema document always starts from the root schema but it can contain any number of nested schemas, denoted subschemas. For instance, Figure 3.10 shows an example JSON Schema that is composed by a root schema (titled “root schema”) and one subschema (titled “subschema”).

The root schema and subschemas of a JSON document are either an object or a boolean. Schemas with boolean root elements are special schemas that either always pass validation of instances (“true”) or always fail (“false”). If the schema is an object, it contains the structure and constrains that must be followed by the JSON instances of the data model described in the schema. The properties of the JSON Schema contain the vocabulary of the data model and are refereed as “keywords”.

Appendix A.2 describes in more detail the structure and keywords defined by the JSON Schema Draft-04 specification.

3.2 Structured Data Compression

Integrating WSNs into larger networks is a mayor design challenge. The research community has tried to bring the technologies designed for the Internet to WSNs with the hope that they would become just another cluster of the cloud. However, Internet technologies do not usually take into account the limited resources and restrictions of a typical resource-constrained IoT device. Thus, these technologies are not easily adapted or are simply beyond the capabilities of a resource-constrained device.

Regarding lossless data compression, there are several approaches. In this thesis we focus on lossless data compression for structured data as opposed to general purpose approaches such as DEFLATE [Deu96] or MP3 [ISO93]. Compression techniques for structured data take advantage from the knowledge of the data format and the data model that describe the structured data. This knowledge is used to extract the grammar and vocabulary that describe the structured data and produce a more compact and efficient encoding of the data (compared to the original format).

There is a fairly hight amount of data compression technologies for structured data which had variable popularity over the years. Readers are encouraged to take a look at [Sak09], [Li10] and [BH13] for a survey on compression technologies targeted either to XML or JSON as well as ad-hoc binary data formats. A summary list is shown in Table 3.1.

In this section we will focus on standard technologies that are most widely used and have a high level of acceptance within the scientific and industrial communities at the time of this writing. Specifically, this section focuses on Efficient XML Interchange (EXI), Concise Binary Object Representation (CBOR) and Protocol Buffers. Despite the several data compression proposals targeted to XML, W3C’s EXI [SKPK14] has emerged in the recent years as the most prominent XML compression algorithm [MTSG10b, HB15]. CBOR [BH13] is a compact data format based on the JSON data model and it has gained increased popularity as a compact representation for JSON data streams. JSON is optimized for simplicity, processing speed, minimum resource usage and implementation compactness and has been ported to several programming languages. Protocol Buffers [18e] is Google’s proposal for structured data serialization. Protocol Buffers provide an ad-hoc schema format in IDL. Although Protocol Buffers define a JSON mapping it is mainly designed to be used on its own. Apart from the compression library, there are various tools

Compressor	**Target Data Format**	**Schema Aware**	**Compression Method**
ASN.1 [Uni15]	ad-hoc	YES	Binary Structure
AXECHOP [LDM05]	XML	NO	Context-Free Grammar
BSON [18a]	JSON	NO	Context-Free Grammar
CBOR [BH13]	ad-hoc, JSON	NO	Binary Structure
DTDPPM [Che05b, Che05a]	XML	YES	Dictionary
Exalt [Tom04, Tom03]	XML	NO	Context-Free Grammar
EXI [SKPK14]	XML, JSON	YES	Dictionary + Grammar
ISX [WLS07]	XML	NO	Labelled Tree
MessagePack [Fur13]	ad-hoc, JSON	NO	Binary Structure
Millau [GS00]	XML	YES	Dictionary
Protocol Buffers [18e]	ad-hoc, JSON	YES	Varints
QRFXFreeze [SNR15]	XML	NO	Structure Modelling + Dictionary
QXT [SS07]	XML	NO	Dictionary
RFXFreeze [SHK+08]	XML	NO	Structure Modelling + Dictionary
rngzip [LE07, Lea15]	XML	YES	Deterministic Tree Automata
SCMPPM [ANdlF03]	XML	NO	Structure Context Modelling
Smile [Fas17]	JSON	NO	Binary Structure
SXSI [ACM+15]	XML	YES	Labelled Tree
TinyT [MS10]	XML	YES	Grammar-Based Tree
TREECHOP [LMD05]	XML	NO	Dictionary
UBJSON [Kal18]	JSON	NO	Binary Structure
XAUST [SS05]	XML	YES	Deterministic Finite Automata
XBzip [FLMM06, FLMM09]	XML	NO	Labelled Tree
XCpaqs [WLLH04]	XML	NO	Dictionary
XCQ [NLWL06]	XML	YES	Dictionary
XGrind [TH02, Tol02]	XML	YES	Dictionary
XMill [LS00]	XML	NO	Dictionary
XMLPPM [Che00]	XML	NO	Multiplexed Hierarchical PPM
XPress [MPC03]	XML	NO	Dictionary
XQueC [ABMP07]	XML	YES	Structure Tree
XQzip [CN04]	XML	NO	Dictionary
XSeq [LZLY05]	XML	NO	Context-Free Grammar
XWRT [Ski16]	XML, HTML	NO	Dictionary
XXS [BCN14]	XML	YES	Dictionary

Table 3.1: Summary of compression technologies targeted to structured data.

to automatically create the code stubs used to marshal/un-marshal the coded streams to/from runtime objects.

Section 3.2.4 includes compression proposals found in the state of the art but did not result in formalized compression technologies or standards. These proposals are included in Section 3.2.4 because of their significance to this thesis, either because they make make explicit use of templates or they are specifically targeted to resource-constrained devices.

Finally, the last section describes compression approaches for protocols that are based on structured data and/or text-based data formats.

3.2.1 Efficient XML Interchange

Although there are several XML compression algorithms, currently the most promising one seems to be Efficient XML Interchange (EXI [SKPK14]), adopted as a recommendation by W3C. For a comprehensible comparative of XML compression algorithms readers are encouraged to consult [MTSG10b]. EXI relies on a binary representation of XML and it is designed to provide a considerable reduction on the size of the information in XML format (70-80 % as shown in [WKB+07]) and high performance when encoding/decoding (6.7 times faster decoding and 2.4 times faster encoding according to [Bou09]) as well as show better results when compared to other XML and JSON compression algorithms [MTSG10b, HB15].

In EXI, a XML document is represented by an EXI stream, which is composed of a header (containing encoding information) and a body (representing the data). The EXI header contains the options used to encode the EXI body. EXI bodies may carry whole EXI documents (i.e. documents that contain the root element followed by the reset of the document) or EXI fragments, which represent portions of a document.

EXI represents data according to three formal grammars. These three built-in grammars are the base of EXI and each one of them is applied to EXI documents (*Built-in Document Grammar*), EXI fragments (*Built-in Fragment Grammar*) and EXI elements (*Built-in Element Grammar*). The built-in grammars are dynamically created during the EXI stream processing. When a new element is found, a new matching built-in grammar is added. In this way, consecutive appearances of the same element will be codified more efficiently using the newly created grammar

If the XML Schema of the XML document is available, EXI can take advantage of the structure information inferred from the schema an use the set of schema-informed grammars instead of the built-in grammars. The set of schema-informed grammars is also composed by three grammars (equivalents in purpose to the three built-in grammars) for EXI documents (*Schema-informed Document Grammar*), EXI fragments (*Schema-informed Fragment Grammar*) and EXI elements (*Schema-informed Element Grammar*).

EXI Event Type	Grammar Notation	Information Items	
		Structure	Content
Start Document	SD		
End Document	ED		
Start Element	SE (qname) SE (uri:*) SE (*)	[prefix] local-name, [prefix] qname, [prefix]	
End Element	EE		
Attribute	AT (qname) AT (uri:*) AT (*)	[prefix] local-name, [prefix] qname, [prefix]	value
Characters	CH		value
Namespace Declaration	NS	uri, prefix, local-element-ns	
Comment	CM	text	
Processing Instruction	PI	name, text	
DOCTYPE	DT	name, public, system, text	
Entity Reference	ER	name	
Self Contained	SC		

Table 3.2: EXI Event types and codes.

Schema-informed and built-in grammars may be used together. The schema-informed grammars will be used as long as the processed XML data conforms to the schema. If there are any deviations, the built-in grammar will be used instead. EXI also defines a “strict” compression mode. In this mode no deviations from the schema-informed grammar are accepted. The strict mode is less flexible than the default mode but it is more efficient regarding compression rate and processing complexity.

The EXI grammars specify the events that are permitted during the data stream processing. The allowed events will depend on the currently used grammar and the information contained in the data stream. Table 3.2 lists the defined event types and codes. Events are codified using 1 to 3 non-negative integers. These codes are used to codify consecutive events within a data stream. For instance, Figure 3.11 shows the built-in document grammar as specified in the EXI Format [SKPK14], including the events that are accepted and the corresponding codes.

3.2.1.1 Built-in EXI Data Type Representations

EXI specifies various built-in data types. The EXI specification includes the representation (i.e. codification) of each of the built-in data types as well as the mapping to/from XML data types. Event codes are codified using n-bit unsigned integers (see Table 3.3) while the contents are represented using the corresponding data type. If no data type can be inferred, String data type is assumed.

The list of specified built-in data-types and their corresponding XML data types is summarized in Table 3.3 and listed below:

- **Unsigned Integer:** unsigned integers are encoded as a sequence of bytes with the least significant byte first. The most significant bit of every byte is set to ‘1’, except

```
Syntax                          Event Code

Document :
    SD DocContent                   0

DocContent :
    SE (*) DocEnd                   0
    DT DocContent                   1.0
    CM DocContent                   1.1.0
    PI DocContent                   1.1.1

DocEnd :
    ED                              0
    CM DocEnd                       1.0
    PI DocEnd                       1.1
```

Figure 3.11: Built-in Document Grammar.

Built-in EXI Datatype Representation	XML Schema Datatypes
Unsigned Integer	nonNegativeInteger or integer bounded to a value equal to or greater than 0 with minInclusive or minExclusive facets.
n-bit Unsigned Integer	integer bounded to a value equal to or smaller than 4096 with minInclusive, minExclusive, maxInclusive or maxExclusive facets.
Binary	base64Binary, hexBinary
Boolean	boolean
Integer	integers not coverred by the n-bit Unsigned Integer or Unsigned Integer data types
Decimal	decimal
Float	float, double
String	string, anySimpleType, anyURI, duration, QName, Notation, all types derived by union
QName	QName for values of xsi:type attribute
Date-Time	dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay, gMonth
List	All types derived by list, including IDREFS and ENTITIES

Table 3.3: Built-in EXI Data Type Representations and associated XML data types.

the last byte which is marked with a ‘0’ in its most significant bit. The actual value of the unsigned integer is stored in the 7 least significant bits of each byte.

- **n-bit Unsigned Integer:** this data type is encoded by representing the unsigned integer value within n bits. Complete bytes are concatenated with the least significant byte first.
- **Binary:** the Binary data type is simply encoded as a sequence of bytes. This sequence is preceded by a length field of the Unsigned Integer data type.
- **Boolean:** this data type is actually a n-bit Unsigned Integer of length 1. FALSE is represented as ‘0’ and TRUE as ‘1’.
- **Integer:** the Integer data type is encoded with a sign Boolean and an Unsigned Integer. A sign value of ‘0’ represents a positive number while the ‘1’ value is used for negative numbers. If the number is non-negative, the Integer value is stored in the Unsigned Integer. If the number is negative, the Unsigned Integer holds the magnitude of the value minus ‘1’.

  Optionally, if a schema definition of the data type is available and the value is bounded, the Integer can be encoded as a n-bit Unsigned Integer or Unsigned Integer.
- **Decimal:** the Decimal data type is composed of a sign field of Boolean data type and two Unsigned Integers. A sign value of ‘0’ represents a positive number while the ‘1’ value is used for negative numbers. The two following unsigned integers respectively represent the integral and fractional portions of the Decimal value. The fractional portion field is encoded with the digits in reverse order to preserve the leading zeros.
- **Float:** the Float data type is composed by two Integers representing the mantissa and the base-10 exponent, respectively. Special values on the exponent value are used to represent especial Float values such as INF or NaN.
- **String:** a String data type is encoded as a sequence of characters. The sequence is preceded by a length field of the Unsigned Integer data type. String data types can also be represented by their compact identifiers under certain conditions. This is explained in more detail in the next section, Section 3.2.1.2.
- **QName:** the QName data type is composed of an URI, local-name and prefix fields. The fields of the QName are encoded depending on the schema information (if available) and whether there is an assigned namespace. If the schema gives enough context information, the URI, local-name and prefix can be omitted. Otherwise URI and local-name fields are represented as String data types while prefixes are

encoded using the compact identifier assigned by the String Table, as explained in the next section, Section 3.2.1.2.

- **Date-Time:** the Date-Time data type is composed of various optional fields that will be present or not according to the specific represented value. The possible fields are listed below:

  - **Year:** an Integer containing the offset from the 2000 year.
  - **MonthDay:** a 9-bit Unsigned Integer that satisfies the equation $month * 32 + day$. day is bounded to the values [1,31] and $month$ is within the range [1,12].
  - **Time:** a 17-bit Unsigned Integer that satisfies the equation $((hour * 64) + minutes) * 64 + seconds$. $hour$ is within the range [0,24], $minutes$ [0,59] and $seconds$ [0,60].
  - **FractionalSecs:** an Unsigned Integer that contains the fractional portion of the seconds. In order to preserve the leading zeros, digits are encoded in reverse order.
  - **TimeZone:** a 11-bit Unsigned Integer that satisfies the equation $TZHours * 64 + TZMinutes + 896$. $TZHours$ is bounded to the values [-14, 14] and $TZMinutes$ is within the range [-59, 59].
  - **Presence:** a Boolean that indicates whether the presence fields are included in the Date-Time value.

  Which fields will be encoded depends on the specific XML Schema data type. For instance, the *date* XML data type will include the *Year*, *MonthDay* and *Presence* fields as well as the *TimeZone* field in case the presence is TRUE.

- **List:** a List data type is encoded as a sequence of items. The sequence is preceded by a length field of the Unsigned Integer data type. The items will be encoded according to their specific data type.

3.2.1.2 String Table

EXI uses a string table to assign "compact identifiers" to string tokens (such as qualified names and literals). The string table is dynamically expanded at run-time to include additional string values encountered in the document. When a string token is found (e.g. qualified names, literals, string values, etc.), it is checked against the string table. If the string is not found, the string is included in the string table and a new compact identifier is assigned. The index of the string token within the string table is used as the compact identifier. If there is a match, the string token is encoded using the associated compact identifier instead of using the string itself. When XML Schema information is available, the string table is initially pre-populated with the string tokens extracted from the schema allowing a much more efficient coding and compression.

The string table is structured into multiple partitions. These partitions are optimized for one of two purposes, either for the frequent use of compact identifiers or string literals. Which partition type will be used will depend on the nature of the tokens stored in it. For instance, URIs and prefix strings will be stored in partitions optimized for compact identifiers while value content items will be stored in partitions optimized for frequent use of string literals.

Splitting the string table in multiple partitions has a second advantage. The assigned compact identifiers are actually the index of the string tokens within the string table partition. By keeping multiple purpose-specific partitions, the indexes are keep relatively small, resulting in smaller compact identifiers which, in turn, produce more compact codes.

As has been explained above, the string table is populated with strings found in the currently processed EXI stream. However, after the EXI stream has been processed, the string table returns to its initial state (i.e. the state before the EXI stream processing started). This means that populated string tables cannot be reused between consecutive EXI streams

There is one notable exception to this rule. When schema-informed grammars are used (i.e. the XML Schema describing the data is available) the string table is already pre-populated with strings contained in the XML Schema. However, if new strings are added to the string table during the EXI stream processing (for instance, a string data value or because an element not conforming to the schema has been found), they will be removed from the string table after the EXI stream processing finishes.

3.2.1.3 EXI Profile

EXI Profile [FP14] recommendation proposes a series of configuration parameters and practices in order to reduce the memory needs of EXI implementations. EXI Profile is targeted to devices that are not allowed (either by design or convenience) to use arbitrary memory growth at runtime. The use of runtime memory is bounded by restricting the growth of string tables and the evolution of grammar(s), sacrificing some of the compression efficiency.

The EXI profile configuration is included as options in the EXI Header. EXI Profile bounds the consumed memory grow by allowing the management of the grammar learning mechanisms. Two options are provided to limit the number of grammars that can evolve at runtime as well as the number of newly inserted grammar productions Additionally, EXI Profile provides an option to disable the use of local value references. In this way the arbitrary grow of the string table is avoided.

3.2.1.4 EXI for JSON

Currently, the W3C is working on the EXI for JSON (EXI4JSON [PB18]) specification witch defines the use of EXI for JSON documents.

EXI4JSON is based on the use of an intermediate XML Schema that maps to the JSON structure. This XML Schema is used to perform an EXI schema-informed compression. First, JSON documents are transformed into an XML document following the schema and mapping specified by EXI4JSON recommendation. Then the schema-informed grammar derived from the schema is used to perform the encoding. EXI4JSON also mandates the use of the schema strict compression mode as well as define the schemaId "exi4json" to identify the EXI4JSON XML Schema.

Initial experiments performed with the *exificient-for-json* [18d] tool show that EXI4JSON outperforms other technologies for JSON document compression.

3.2.1.5 Conclusions

Despite its many configuration options, EXI may be too complex to be efficiently implemented in resource-constrained devices. On the one hand, the implementation may require too much code memory or processing time. On the other hand, EXI requires the use of runtime memory allocation in order to accommodate schema deviations and grammar learning.

EXI Profile recommendations may not cover the resource limitations of the most resource-constrained devices. In contrast, the compression approach proposed in this thesis is specifically targeted to resource-constrained devices and makes use of templates and schema context information for energy efficient management of standard data model representation formats.

EXI4JSON does not take direct advantage of the JSON Schema as it relies on an intermediate XML Schema. This means that EXI does not take advantage of the JSON Schema vocabulary. The proposed compression approach, on the other hand, directly exploits the JSON Schema information to perform the codification, removing unnecessary contextual information from the compressed stream.

3.2.2 Concise Binary Object Representation

Concise Binary Object Representation (CBOR [BH13]) is a compact data format based on the JSON data model. CBOR is optimized for simplicity, processing speed, minimum resource usage and implementation compactness.

Although it is designed to be used on its own, the CBOR specification defines a JSON mapping that can be used to directly transform data between JSON and CBOR formats.

JSON objects are converted to CBOR maps where the JSON property names are used as the keys of the CBOR map.

CBOR follows a very straightforward approach to encode data items. The first byte of each data item gives information about the data type of the data item. Thus, CBOR includes the information about the CBOR data types in-line, in the coded stream itself. The data type byte is further divided into two distinct fields, the *major type* (high order 3 bits) and *additional information* (remaining low-order 5 bits).

The *additional information* field provides further data-type specific information that is used to decode the data item's value. For instance, if the *additional information* value is less than 24 it represents a small unsigned integer (i.e. an integer between 0 and 23) while if the value is between 24 and 27, it indicates that the actual value of the *additional information* field is encoded in the following bytes and its length is 1, 2, 4 or 8 bytes long respectively.

The *additional information* field interpretation is done according to the *major type* field semantics. For instance, if the *major type* indicates and array, the *additional information* field provides the length of the array.

3.2.2.1 CBOR Data Type Codification

The following list summarizes the major types as well as the type-specific *additional information* and any complementary fields.

- **Major Type '0':** the *major type* '0' is used to represent an unsigned integer. The *additional information* field encodes the integer value either directly (if less than 24) or in the following bytes (if between 24 and 27).
- **Major Type '1':** this type represents a negative integer. The *additional information* field encodes the integer value in the same way as unsigned integers (*major type* '0').
- **Major Type '2':** *major type* '2' is used to represent byte strings. The *additional information* field is interpreted as an unsigned integer and specifies the length of the string. The actual byte string follows the data type encoding.
- **Major Type '3':** this type follows the same encoding specification as byte strings (*major type* '2') but it specifically represents a string of UTF-8 characters.
- **Major Type '4':** this type is used to represent arrays of data items. The *additional information* field is interpreted as an unsigned integer and specifies the number of items in the array.

- **Major Type '5':** the *major type* '5' represents a map. A map is composed of key/value data item pairs that are concatenated together to form the map. The first data item of each pair is the key, followed by the value. The *additional information* field is interpreted as an unsigned integer and specifies the number of data item pairs contained in the map. CBOR applications need to agree on what type(s) of keys are used.

- **Major Type '6':** *major type* '6' is a special type used to semantically tag data items. A tag data item is used to give semantic meaning to the following data item. The CBOR specification includes predefined values for the tag data item's *additional information* field. For example, dates and big numbers (bignums).

- **Major Type '7':** this type is used to encode floating-point numbers and special data types (such as *true* or *false*).

Finally, CBOR specification provides some hints on a CBOR-to-JSON transformation. Basically, base data types are directly mapped from CBOR to JSON (and vice-versa). For instance, CBOR integers (*major types* 0 or 1) are mapped to JSON numbers and CBOR arrays (*major type* 4) are mapped to JSON arrays. Notably, JSON objects are converted to CBOR maps where each key/value pair represents one of the properties of the object. The map the keys are CBOR string data items containing the name of the JSON property.

Some specifications based on CBOR may provide integer substitutes for the JSON property names. In these cases, property names are first transformed into the assigned integer substitutes and then used as keys for the CBOR map. Thus, the keys are integer data items instead of string data items achieving a more efficient encoding. For instance, this is the approach followed by Media Types for Sensor Measurement Lists (SenML) [JSA+18] specification.

3.2.2.2 Conclusions

CBOR does not rely on schema information and codifies the data types within the coded stream. This design decision simplifies the implementation as no cross refeferences to context information is needed in order to decode data values. However, every data value must be preceded by a data type description (in the form of one or more data type bytes) with the added overhead on the resulting compression size. This is the case even when a JSON mapping is used to define compact keys for the CBOR maps (which are transformed into JSON objects).

The compression approach proposed in this thesis takes full advantage of JSON Schema information to achieve good compression while resulting in simple enough implementations that fit the requirements of resource-constrained devices.

```
message Pet {
  required string name = 1;
  optional int32 age = 2;

  enum Species {
    CAT = 0;
    DOG = 1;
    TORTOISE = 2;
    RABBIT = 3;
  }

  required Species species = 3;

  message Event {
    required string date = 1;
    required string description = 2;
  }

  repeated Event events = 4;
}
```

Figure 3.12: Protocol Buffer ".proto" file example.

The most efficient CBOR compression is achieved by mapping the JSON property names to integers and using them as keys for the CBOR map, such as the approach followed SenML [JSA+18]. However, CBOR does not define any formal concept of schema and does not provide any mechanism to define and distribute the mapping structure.

3.2.3 Protocol Buffers

Protocol Buffers [18e] are Google's proposal for structured data serialization. The structure of the data is described as an IDL document that acts as the data schema. This IDL document is pre-compiled to produce the code stubs to marshal/un-marshal the coded streams to/from runtime objects. In a similar way as CBOR, Protocol Buffers also provide a JSON mapping that can be used to directly transform the structures defined in the IDL to JSON.

In Protocol Buffers, the structure of the data is specified in a IDL file (known as the ".proto" file). The Protocol Buffer community provides multiple tools in order to parse and process the .proto files and create the code stubs for various programming languages, such as java and C++. These stubs are included in the application code in order to serialize native data structures into the Protocol Buffers binary format and de-serialize Protocol Buffers into native structures.

Protocol Buffers use the concept of Protocol Buffer Messages which are defined within the .proto file. The format of the message definition in the .proto file contains uniquely numbered fields, a name and a data type. Messages can also be arranged into a hierarchy by defining messages within messages. The fields of a message can be tagged as optional, repeated or required. Figure 3.12 shows an example of a .proto file. In this example the "Pet" message is composed by the "name", "age", "species" and "events" fields, respectively numbered 1,2,3 and 4. The "events' field, in turn, is also a message of the "Event" type.

3.2.3.1 Data Type Codification

Protocol Buffers binary encoding is based on *varints*, specifically base 128 *varints*. *Varints* are a straightforward serialization method for integers where smaller integers are serialized in fewer bytes. In a base 128 *varints*, the most significant bit of each byte is set to '1' except in the last byte. The 7 less significant bits of each byte contain the corresponding portion of the value of the integer, in little endian order. For example, the number two is serialized as 0x02 (0000_0010) while the number 300 is serialized as 0xAC,0x02 (1010_1100, 0000_ 0010).

Protocol Buffer Messages are encoded as one data structure following a key/value format. Each element of the message is also encoded as a key/value pair, thus, the structure of an encoded Protocol Buffer Message is composed of concatenated key/value pairs.

The key is assigned according to the definition of the field as described in the .proto file, i.e. the number assigned to the field in the .proto file is used as the key. The data type of the field is not encoded in the protocol buffer message and can only be extracted from the .proto file. Thus, a decoding application needs to know the .proto file used by the encoding application.

However, the key also contains enough information to skip the encoded field so it can be skipped in case it is not recognized. This allows to mix different versions of the same .proto file in encoding and decoding applications. The information used to skip the encoded field is denoted a "wire type". The wire type is part of the key (in the lower three bits) together with the field number. The whole key is encoded as a *varint*.

Array types (defined as "repeated" in the .proto file) are encoded using a single key/value pair. A wire type is used to determine the length of the full array (in bytes) and the values are concatenated in order within the value. However, if multiple instances of the same array key are found within the same encoded stream, they are concatenated together. Optional fields are implicitly omitted by not including them in the encoded stream.

3.2.3.2 Conclusions

Protocol Buffers do not provide a direct mapping of JSON Schemas (or other schemas) and directly rely on the definition of the structures following the dedicated IDL format. Thus, Protocol Buffers rely on application specific and manually defined IDL files resulting in solutions that lack interoperability. Additionally, Protocol Buffers specification does not include any mechanism to distribute the IDL schemas as well as identify them.

In contrast, the solution proposed by this thesis is general enough to be mapped to multiple data model representation formats and produces the internal constructs needed for the compression from standard schemas (such as XML and JSON Schemas).

3.2.4 Other Proposals on Compression for Structured Data

This section describes other compression proposals found in the state of the art not covered in the previous sections. These works did not result in formalized compression technologies or standards, and could be considered as still being open researches. However, they are mentioned here because of their relation and significance to this thesis. Some of these compression proposals make explicit use of templates or pattern repetitions in order to represent structured data in a more compact encoding. Other proposals covered in this section are specifically targeted to resource-constrained devices.

Hoeller et al. [HRN+08, HRN+10a, HRN+10b] identified the inefficient management of XML formatted data in resource-constrained devices as a barrier to overcome in order to achieve full interoperability. They defined a series of mechanisms to efficiently manage XML in terms of processing, storing, and transmission.

In the solution proposed by Hoeller et al. identifiers (such as XML element names) are first separated from the original XML data and stored in program/flash memory. In a second step, the XML data is processed to extract repeating structures and templates. In this step the static and dynamic data of the XML data are separated and embedded into a structure denoted a XML Template Stream (XTS). The XTS is encoded in a memory efficient way. For instance, the repeated structures are substituted with references to the templates. Finally, the XTS is encoded using a binary format based on Huffmann encoding [Huf52].

Hoeller et al. also developed a pre-compiler tool called $XOBE_{SN}$ [HRN+10b]. This tool allows an easy integration of XML structures into C programs that are later translated into plain C (compilable with standard compilers). Additionally, it makes use of reused structures to efficiently store and process XML documents. The client/server communication model is based on XPath queries and optimized for this purpose.

The use of pseudo XML structures in the code makes the solution proposed by Hoeller et al. heavily tied to XML. This thesis proposes a more natural data binding technique by using native C structures, giving a convenient abstraction of the underlying original data representation format.

The work presented by Hoeller et al. does not provide a formal encoding or compression format for data transmission. The use of templates is suggested for the transmission of data but few details are given.

Käbisch et al. propose in [KPHK11] a solution to generate optimized XML-based Web services using EXI and targeted to resource-constrained devices. Basically, the solution uses a SOAP WSDL as input in order to generate the required EXI grammar, optimized EXI processor and binding code stubs. The paper also includes performance results that show the significant efficiency improvement regarding message size and code footprint.

Later, Käbisch et al. extended the core approach presented in [KPHK11] to propose a solution for efficient processing and storing of RDF documents in resource-constrained devices: µRDF [KPA15, CKK17]. This solution enables the efficient use of semantic data in resource-constrained devices in order to follow approaches such as WoT. The paper defines a XML Schema to describe the RDF document structure and uses it to generate the EXI grammar. Additionally, the paper presents µRDF, a semantic repository that efficiently represents and stores RDF data.

Käbisch et al. also explored other uses of EXI in order to optimize network traffic based on service filtering [KK14] . These filters are applied directly over EXI grammars and avoid the transmission of unwanted/useless data. The main contribution of this research line is to be an interesting use case of an EXI application.

TinyPack XML [SML12] is a XML compression method for WSN that takes advantage of the structured nature of XML and the similarity between data messages consecutively transmitted. Each data message to be transmitted is analysed and compared to previous messages. The common sections of the XML data are extracted and set as “format strings”. A compact identifier is assigned to the format strings and they are advertised to the (sub-)network. Dynamic data is encoded using techniques specific of the data type.

In TinyPack, the format string is refined with each every transmitted data message. This means that the data message has to be preprocessed every time in order to check for variations on the structure. The paper argues that there is indeed additional processing involved but that it is compensated by the savings in transmission time. Each time the format string is modified, it has to be advertised to the network. If the nature of the messages change often, this implies sending the format string very frequently.

They also propose other optional methods to extract the format string. These optional methods include the extraction of the format string from the XML Schema, although this would imply sending a lot of overhead data because, usually, all the elements within a XML Schema are hardly used all together. They also claim that the format string could be defined by hand on a message by message basis. Although this would result in an optimal assignment of format strings, it would rely on the end users skills and could be cumbersome for development processes. Nevertheless no data regarding the performance of these methods is provided and their benefits are qualitatively exposed.

Packedobjects [Moo09, Moo10] was first designed to implement network protocols in a compact format. The compression approach used by Packedobjects is based on the efficient encoding of the data types, which are previously extracted from a schema. Latter on, Packedobjects was applied to XML compression in [MKB13, MKB14, KMB13]. The data types used for the encoding are extracted from XML Schemas for which Packedobjects implement a subset of the XML Schema specification.

The experiments performed in [KMB13] show that Packedobjects outperforms XML compression technologies. However, the XML compression technologies used for the comparison are mainly not based on schema information (such as XMLPPM and XMILL), which are the ones showing the best performance. Additionally, they do not compare it to EXI which was the leading XML compression technology at the time.

A performance evaluation of Packedobjects against EXI is presented in [BMKR15]. The paper shows that Packedobjects and EXI have similar compression ratios but that Packedobjects shows a better processing performance. However, the comparison was made between a C Packedobjects implementation and a java EXI implementation (EXIProcessor) that is not intended for resource-constrained devices nor is as optimized as a C application. Thus, the comparison is not made under fair conditions.

3.2.5 Compression of Protocols Over Text-Based Data Formats

This section presents different approaches made by the research community to improve the efficiency of protocols built on top of structured data and text-based data formats. Notably, the efforts are targeted to SOAP or SOAP-based protocols. This makes sense as SOAP itself is implemented using XML. SOAP shares all the benefits of XML (self-describing, interoperable, etc) as well as its burdens, especially the verbosity.

Roşu proposes Adaptive SOAP (A-SOAP [Ros07]), a compression approach that takes advantage of the structure repetitions of messages transmitted between two nodes. The work presented is focused on Web Services implemented through SOAP and the final aim is to reduce the message processing time and message size. The A-SOAP stack incrementally builds a dictionary with the string tokens used in the messages and substitutes them with more compact identifiers. The dictionary is interchanged with the other endpoint according to a configurable policy.

A-SOAP is not specifically targeted to resource-constrained devices but it proposes simple enough mechanisms that could be adapted to resource-constrained devices. However, A-SOAP trades memory for processing and transmission efficiency because it does not provide any mechanism to control the memory used as the message types and endpoints number grows. Additionally, the achieved compression is suboptimal as it is simply based on the substitution of XML tags but does not take advantage of the structure and grammar of XML documents.

Devices Profile for Web Services (DPWS [CCK+06], a conjunction of ws-* services) defines a minimal set of implementation constraints to enable secure Web Service messaging, discovery, description and notification on resource-constrained devices. The main purpose of DPWS is to bring WSs to small embedded devices. That way, devices can communicate with each other or other DPWS enabled devices and applications using and standardized protocol.

DPWS is based on SOAP and makes extensive use of XML. Thus, DPWS also shares the verbosity of SOAP and XML. Additionally, DPWS sets demanding quality-of-service restrictions and require significant processing power and memory consumption. These qualities turn DPWS too complex and resource demanding to be adopted by resource-constrained devices, despite all advanced capabilities it provides.

There are many attempts to bring DPWS to resource-constrained devices and networks, ranging from the compression of XML with EXI [AGGT10] to simplifications or translations of the protocol [MZP+09, MTSG10a, SHG13]. In some cases, the use of ad-hoc templates is also suggested [BZB+08].

There are also dedicated implementations, such as μDPWS. μDPWS [12b] is a specialized implementation of DPWS, designed to work on micro-controllers with small amounts of memory. Thus, it is well suited for memory-efficient networked embedded systems. Although this provides a useful implementation, it is still not suited for the most resource-constrained devices and networks.

Moritz et al. [MZP+09] leverage the necessary modifications needed to apply DPWS mechanisms in WSNs without a loss of interoperability. The major goal of all restrictions and enhancements is the minimization of exchanged messages inside the WSN and the reduction of memory usage of DPWS implementations. The enhancements are focused on the reduction of the necessary discovery messages based on a previous research made in [BZB+08]. Moritz et al. also define service and device templates for DPWS. These templates are used to create application specific DPWS clients in a more memory efficient way. The templates can also be used to extract application information and speed-up DPWS discovery processes.

Moritz et al. also detect some key points where improvements can be applied. The analysed points range from the message size consumed by namespace declarations, management (transmission and storing) of WSDL, or the major problems found while porting the event concept to WSNs. Finally, they make some recommendations. However, the paper is mainly focused on alternative options that DPWS provides in order to reduce the offered traffic and therefore innovative ideas are missing.

Later Moritz et al. define an ad-hoc compression mechanism for DPWS [MTSG10a] targeted to 6LowPAN networks, named encDPWS. encDPWS provides a binary more compact encoding than traditional XML based DPWS protocols. the binary encoding is designed to be used within the constrained network. The border router or gateway that links the constrained network with external networks is tasked with the required encoding/decoding processes to translate between the compact and original DPWS formats.

enDPWS is designed to be stateless and in order to meet this requirement HTTP related information is also included in the encoded DPWS messages. The paper also notices the need for a TCP to 6LoWPAN-UDP mapping but it is not covered in the paper.

All the proposals made by Moritz et al. are focused on specific solutions to enable DPWS on resource-constrained networks. However, these proposals can not be extended to other SOAP or XML based technologies.

A modified DPWS protocol stack that can be embedded in WSNs (Tiny SOA for wireless sensors, TinySOAWS) which support the 6LoWPAN architecture is proposed by Samara et al. [SHG13]. The proposed solution is based on the transformation of DPWS messages in a more compact XML structure and format. Basically XML tokens (element names, attributes, etc.) are substituted with more compact tags and the DPWS and SOAP namespaces are mapped to a single namespaces removing the need for explicit XML namespaces and prefixes. The proposed format preserves all the semantics of the DPWS and SOAP vocabularies. Although TinySOAWS results in a more compact XML document, it still uses XML format to encode data, resulting in a overhead for resource-constrained devices.

3.2.5.1 Conclusions

To date, the efforts to adapt DPWS to WSNs are based on a functionality subset, ad-hoc translations or use EXI for XML compression. All these solutions are either too specific or partially solve a specific problem. The mechanisms devised to reduce the overhead of the protocol logic (such as the reduction of discovery messages) are specific to DPWS and may not be applicable to other protocols or situations. All the elements in the network would need to support the DPWS functionality implemented with the ad-hoc solution.

3.3 IoT Communication Protocols

This section provides an overview of the most relevant communication protocols within the IoT domain for this Thesis. However, the descriptions provided here do not intent to be exhaustive but to provide the reader with the necessary information and background to understand the technical descriptions and relevance of the contribution of this Thesis.

The relevance of protocols targeted to constrained devices does not only takes into account message transmission but also transversal services such as service discovery, advertisement of network availability and seamless interoperability with other networks or protocols. Thus, the protocols included in this section gather the philosophy of more sophisticated Internet oriented protocols/mechanisms and redesign them to be applied to constrained environments.

On the other hand, compression technologies for structured data require the use of schema or structure information in order to achieve the most efficient compression.

However, they do not specify how this information is shared and referenced, or they assume that an out of band method is used. These drawbacks can be compensated by mechanisms provided by the underlying communication protocol.

There are several protocols targeted to IoT domains. These protocols successfully adapted or extended Internet standards to more constrained environments and provided more efficient implementations of their Internet-equivalent. For instance, we can find 6LoWPAN [SB10] as the IPv6 adaptation on top of 802.15.4 [GCB03]. CoAP [SHB14] brought the REST philosophy to constrained environments. In the same manner, MQTT provides a simple publish/subcribe implementation. On the other hand, DNS [Moc87] adaptations based on m-DNS [CK13b] and DNS-SD [CK13a] to constrained sensor networks [KK12, KK13].

These protocols offer optimized mechanisms to discover, register, retrieve, reference and, in summary, manage certain resources in an optimized way. Further information about protocols targeted to resource-constrained devices, their characterization and limitations can be found at [OA17].

This section focuses on a few IoT protocols that are relevant for the work presented in the technical chapters of this document. The following sections describe in more detail the three protocols listed below:

- **CoAP:** CoAP is a very popular IoT protocol that has successfully enabled native REST services in constrained environments. This thesis uses CoAP as one of the main building blocks to create a REST web service based on CTC that includes message delivery, template management and service/resource discovery. A binding of the CTC communication architecture to CoAP is provided in Section 5.3 and the evaluation is shown in Section 7.3.

- **MQTT:** MQTT is a lightweight publish/subscribe protocol targeted to M2M applications. It is also very popular within the IoT domain due to its simplicity and publish/subscribe approach. Furthermore, the MQTT community defined the MQTT-SN specification which further optimizes MQTT to resource constrained sensor networks. MQTT-SN specifies a runtime mechanism to map (potentially long) MQTT topics to a more compact topic identifier. This mechanism follows a similar approach to the method proposed by Thesis to assign compact identifiers to schemas.

- **XMPP:** XMPP is included because it is a simple, highly extensible IoT protocol based on XML. Additionally, XMPP includes an extension that specifies the application of EXI encoding to allow the use of XMPP in constrained environments. Thus, XMPP provides a relevant example of an IoT protocol that directly benefits from data model compression.

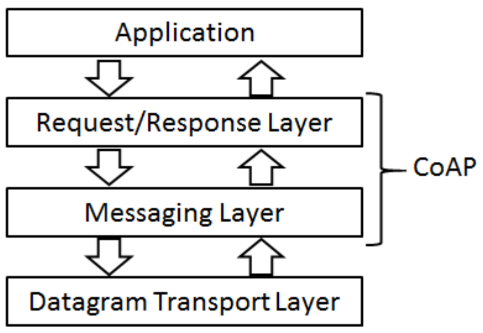


Figure 3.13: CoAP architecture.

3.3.1 Constrained Application Protocol, CoAP

The IETF CoRE group specifies the Constrained Application Protocol (CoAP [SHB14]) as a “specialized web transfer protocol for use with constrained networks”. CoAP follows the request/response interaction model and includes core concepts of the web such as URIs and media types. CoAP implements a limited subset of HTTP functionality, making it easy the cross-protocol proxying to HTTP. Thus, CoAP could be seen as the equivalent of HTTP within WSNs, allowing the integration with the Internet through a simple and nearly direct translation.

However, CoAP does not simply compress HTTP, it also provides a series of HTTP related mechanisms, commonly used to implement REST, but optimized to M2M constrained systems. These mechanisms include low header overhead, multicast and asynchronous messages, stateless mapping to HTTP, and built-in discovery. Basically, CoAP implements the REST approach by using CoAP requests to perform and action on a resource provided by a server which in turn sends a CoAP response. This is very similar to the approach followed by HTTP but instead of a connection oriented approach, CoAP uses and asynchronous one. Thus, unlike HTTP that is usually bind to TCP, CoAP runs over a datagram-oriented transport layer, such as UDP. Additionally, CoAP also supports multicast requests. On the other hand, CoAP defines several security modes and a binding to DTLS as well as the use of IPsec [Bor12].

Thanks to CoAP, RESTful mechanisms can be effectively used on WSNs while meeting the requirements of constrained systems. This leads to an easy development of simple applications oriented towards Web Services.

Although the CoAP specification defines a single protocol, it is usually represented with two layers (Figure 3.13). The first layer is composed by the messaging model which is meant to deal with the asynchronous interactions of the communications. The second layer defines the request/response methods on top of the messaging model.

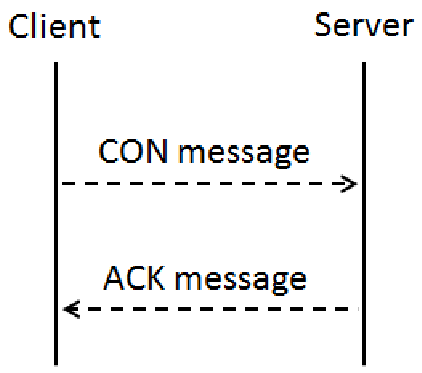


Figure 3.14: CoAP reliable message.

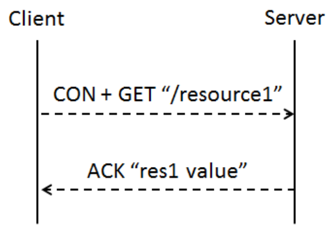


Figure 3.15: CoAP piggybacked response.

3.3.1.1 Messaging Layer

CoAP defines a compact header that is used in request and response messages. The messages can be marked either as Confirmable (CON) or Non-confirmable (NON). A CON message is used to implement reliable transmissions and indicates that an Acknowledgement (ACK) message is required as a response (Figure 3.14). If for some reason the recipient is not able to send an ACK response, a Reset (RST) message is sent instead. If reliability is not required, messages are marked as NON and, in this case, a response message is not required. However, as which CON messages, if the recipient is not able to process the NON message it may still reply with a RST message.

3.3.1.2 Request/Response Layer

CoAP messages are divided into requests, which are identified with Method Codes, and responses that carry Response Codes. If a request message is marked as CON, the response is included in the ACK message and is known as a piggybacked response (Figure 3.15). If the request cannot be satisfied immediately, the recipient still sends and empty ACK message in order to satisfy the reliability requirement. Once the response is available, it is send in a CON message that also requires to be confirmed with an ACK message (Figure 3.16). On the other hand, if the request is marked as NON, the recipient may respond with a NON response message. Optionally, the recipient may also respond with a CON message.

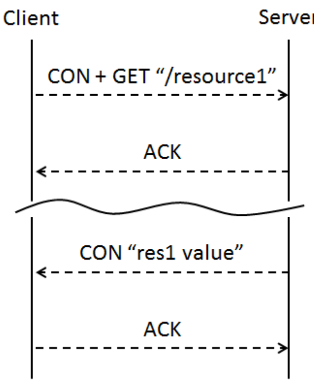


Figure 3.16: CoAP separate response.

CoAP defines the GET, PUT, POST, and DELETE methods. The semantics of these methods are similar to the equivalent methods specified by HTTP, although some differences apply.

- GET: the GET method is used to access and retrieve the resource identified by the requested URI. The GET method is safe and idempotent.
- POST: the POST method is used to request to the recipient to process the contents of the request message and apply it to the target resource. The POST method is usually used to create or update a resource. POST is neither safe nor idempotent.
- PUT: the PUT method is used to create or update the target resource with the contents of the request message. PUT is not safe but is idempotent.
- DELETE: this method is used to delete the target resource. DELETE is not safe but is idempotent.

Additionally, CoAP defines several Response Codes that are used to identify the resulting status after the request have been processed by the recipient. Response Codes are grouped into various categories: responses within the "Success" category are used to inform that the request was successfully received and processed. "Client Error" and "Server Error" categories' responses are respectively used to indicate that a client or a server has suffered from an error in the processing of the request.

3.3.1.3 Caching

Caching is a convenience mechanism of the CoAP protocol designed to improve the overall performance of the communications. The foundation behind CoAP caching is to store a response message with the hope that it can be used to satisfy a future request. When a client node makes a request, a server may decide to use the cached response in case a new (updated) value since the previous request is not available.

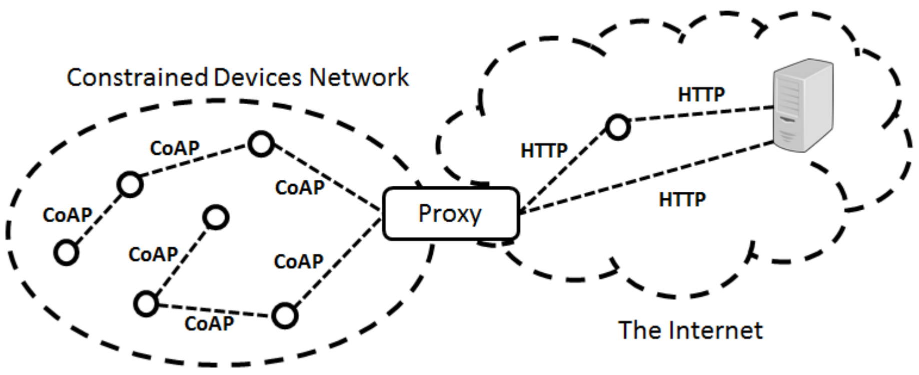


Figure 3.17: CoAP architecture.

There exist various parameters in order to decide whether a cached response can be used to satisfy a request. For instance, the cached value should be “fresh” enough to still be valid and the method used in the current request must match the method used to produce the cached value. Additionally, CoAP provides mechanisms for a client to specify when a message is cache-able. It also defines mechanisms to specify whether a message can be cached and check if it is still valid.

3.3.1.4 Proxying

CoAP supports proxying and relies on it to improve the overall performance of the network as well as to implement interoperability across networks. Given the similarities between HTTP and CoAP and that they both follow the REST [Fie00] architecture, the mapping between both protocols is quite straightforward. This is useful to deploy cross-protocol proxies that seamlessly map CoAP networks to HTTP networks.

Within CoAP, a proxy is an intermediary that forwards requests and relays responses in the name of node endpoints (Figure 3.17). An interesting property of CoAP proxies is that they do not assume or implement any application semantics, thus, they can be used to deploy application agnostic proxies between (sub-)network domains.

CoAP defines two types of proxies: forward-proxy and reverse-proxy. A Forward-Proxy is a proxy explicitly selected by a client node in order to relay messages on its behalf. On the other hand, a Reverse-Proxy is a proxy that relays messages and accepts requests in a transparent way, i.e. as if the messages would have been sent by the original node. Thus, the client node is not aware that it is communicating with a proxy instead of with the original node.

The role of a CoAP node is not fixed and it can switch from server, to client to forward-proxy and reverse-proxy during its lifetime, depending on the application behaviour.

Finally, CoAP proxies make use of caching whenever possible in order to speed-up responses and improve the overall network performance

3.3.1.5 Resource Discovery in CoAP

The CoRE group has defined many extensions for CoAP. One of the most interesting functionalities for this Thesis is the CoRE Link Format [She12] that specifies *Web Linking* to be used within CoAP. Within the HTTP domain [FGM+99], *Web Discovery* is defined as the ability to discover resources provided by a HTTP web server, while *Web Linking* [Not10] refers to the description of the relations between the resources. These two mechanisms together bring great flexibility to the system as they provide the necessary tools for discovery of hosted resources (by means of URIs), their description (by means of attributes) and progressive discovery of additional related resources (by means of link relations).

The CoRE Link Format is an extension of the HTTP Link Header format and *Web Linking*, used to describe links for CoRE Resource Discovery. This specification adapts *Web Linking* to the constrains of WSN systems but, unlike HTTP, these links are a resource on their own represented in the CoRE Link Format.

CoRE Resource Discovery is performed through a well-known relative URI "/.well-known/core" that is defined as a default entry point. The root of the well-known resource's path is "/.well-known/" as specified in [HLN10] and it is extended with "core". Thus, the full path specified by CoRE Resource Discovery is "/.well-known/core".

In order to perform discovery on a server, a GET request is made on the "/.well-known/core" path of the server. On reception, the server returns the set of links that conform the resources hosted in the server (or resources referenced by the server and stored elsewhere). These links are represented using the CoRE Link Format. However, the included links, their organization and relations are application specific.

The CoRE Resource Discovery is complemented with the the CoAP Resource Directory [SKB+18]. The Resource Directory is used to store information about generic web resources. A Resource Directory offers a REST interface designed for the registration and lookup of stored resources as well as provide discovery capabilities. CoAP Resource Directory allows the registration of generic resources and the creation of links to them. When a resource is registered, a path to the resource is returned as a means to reference it, for example, to delete it. However, the format of this path results in a string that can be unnecessarily large. This renders the path string as a suboptimal choice for the resource identifier.

3.3.2 MQTT

Message Queue Telemetry Transport (MQTT [BG14]) is a broker-based publish/subscribe messaging transport protocol. Clients subscribe and publish to topics which are created as a hierarchy of character strings. MQTT is designed to be light weight and easy to implement which make it suitable for constrained environments such as typical IoT

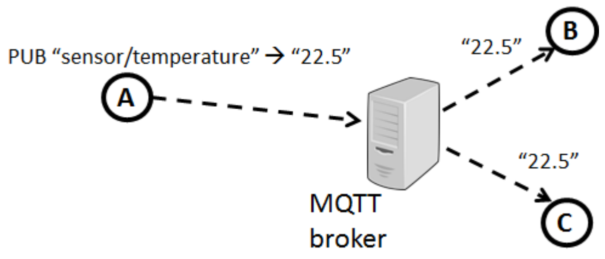


Figure 3.18: MQTT architecture.

```
building05/apartment10/room2/device03
```

Figure 3.19: MQTT topic example.

scenarios. For instance, MQTT adds minimal message size overhead and exchange of messages in order to reduce as much as possible network traffic.

MQTT runs on top of bidirectional and lossless connections (typically TCP/IP) and uses a central server or message broker to relay messages and implement the publish/subscribe behaviour. Figure 3.18 shows an example of three clients connected to a broker. In the example client A publishes a message to the topic "sensor/temperature". This message is received by clients B and C that were subscribed to that very same topic.

Topics are filtered by MQTT brokers and relayed to the subscribed clients. A topic is arranged following a hierarchy and multiple levels separated by a forward slash (i.e. '/'). Topics do not need to be pre-created and they can be used on the fly. Clients can subscribe directly to the desired topics or use wild-cards in order to cover a common set of topics in the hierarchy. Wild-cards can only by used in subscriptions. Figure 3.19 shows a four level topic ("building05", "apartment10", "room2" and "device03").

Finally, MQTT provides three levels of QoS that range from best effort unreliable transmission to reliable one time delivery.

3.3.2.1 MQTT-SN

MQTT-SN [SCT13] is a MQTT version tailored to the constrains and conditions typically found in sensor networks such as the particularities of wireless communication links, limited energy, low bandwidth, and short messages. In sort MQTT-SN is a publish/subscribe MQTT-SN protocol for constrained sensor networks.

MQTT-SN networks are composed by three distinct components (Figure 3.20): clients, gateways and forwarders. A MQTT-SN node connects to a MQTT node through a MQTT-SN gateway. the MQTT-SN node uses the MQTT-SN protocol to communicate with the MQTT-SN gateway and the gateway translates the messages to MQTT and relays them

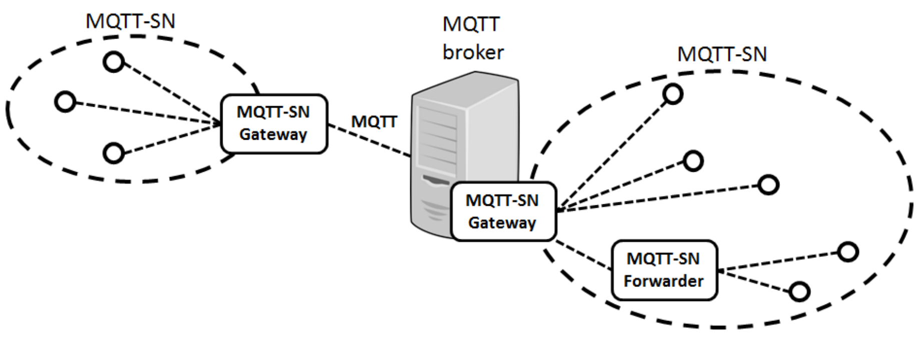


Figure 3.20: MQTT-SN architecture.

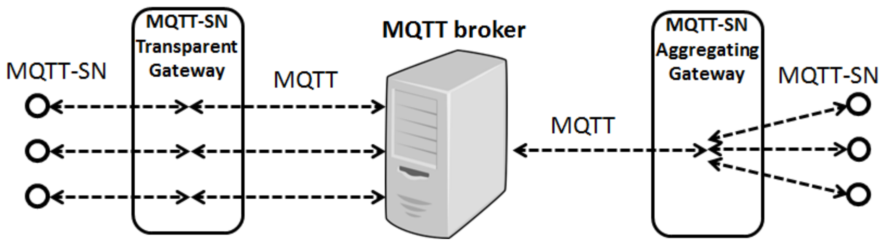


Figure 3.21: Transparent and Aggregating Gateways.

to the MQTT node. A forwarder is used in those cases where the gateway is not directly accessible by the MQTT-SN node (i.e. the gateways in a different network). The forwarder simply relays the node's messages to the gateway (or other forwarder) and vice-versa.

MQTT-SN gateways are further divided into two classes, *transparent* and *aggregating* gateways (Figure 3.21), depending on how the translation between MQTT and MQTT-SN is made. A *transparent gateway* will maintain one MQTT connection with the MQTT broker for each MQTT-SN node connected to the gateway. A *transparent gateway* will transparently translate between MQTT and MQTT-SN protocols keeping the contents of the messages unchanged and effectively maintaining an end-to-end connection. In contrast, an *aggregating gateway* will only maintain one MQTT connection with the broker, no matter the number of MQTT-SN nodes connected. In this case, there is no end-to-end connection and the gateway behaves more like a proxy, deciding which messages are relayed to the MQTT broker.

MQTT and MQTT-SN are very similar. The differences are motivated mainly by the more constrained environments MQTT-SN is designed for. On one hand, MQTT is designed to run on top of TCP-like transport layers while MQTT-SN only requires a bi-directional transportation layer. On the other hand, the mechanics of MQTT have been simplified. For instance, the MQTT CONNECT message (used to stablish a connection) is split into three messages in MQTT-SN. Only one of the messages is mandatory, reducing the complexity and bandwidth required for a regular connection.

Another aspect that has been optimized in MQTT-SN is topic management. In order to reduce the (usually relatively long) size of the MQTT topic, MQTT-SN replaces it with shorter "topic id". topic ids are two octect long. MQTT-SN defines a procedure to register topics at runtime and assign identifiers to them. Additionally, MQTT-SN defines "pre-defined" topic ids and "short" topic names. Pre-defined topic ids are topic ids that are statically mapped to a known set of topic names and are known to the clients and gateways rendering the registration process unnecessary. Short topic names are topic names that are already two octets short. These topic names are preserved and used without the need of a registration process.

MQTT-SN also implements a simplified discovery feature in order to look out for valid gateway addresses. MQTT-SN gateways can coexist in the same WSN and either co-operate sharing the network load, or behave in a standalone way. Furthermore, MQTT-SN supports sleeping clients through an off-line keep-alive procedure. Gateways store the messages for sleeping clients and relay all the messages to them once they awake.

3.3.3 XMPP

Extensible Messaging and Presence Protocol (XMPP [SA11a]) was born in the Jabber open-source community [17] as an alternative to closed instant messaging services. Nowadays, XMPP has significantly extended its initial capabilities and it covers several application domains beyond instant messaging (e.g. voice and video calls, multi-party chat, cloud computing, etc.) as well as provide a lightweight middleware for the routing of arbitrary XML data.

XMPP specification is open. It is promoted by the XMPP Standards Foundation and is standardized by the Internet Engineering Task Force (IETF). The XML specification is split into three parts XMPP Core [SA11a], XMPP IM [SA11b] and XMPP Address Format [SA15]. XMPP is easily extensible as it is based on XML. Custom functionalities can be implemented by defining XML extensions on top of the XMPP specifications. For instance, an extension for instant messaging and presence functionality is defined in [SA11b]. Other common extensions are listed in the XSF's XEP series [SAC16].

The XMPP core specifications define the XML streaming layer and communication primitives for messaging, network availability ("presence"), and request-response interactions. Additionally, XMPP specification has built-in security by means of channel encryption with TLS and authentication with SASL.

XMPP defines the vocabulary and application profile on top of XML for near-real-time transfer XML documents following a well defined streaming protocol. XMPP is designed to provide asynchronous end-to-end transmission of structured data.

3.3.3.1 XMPP streams and stanzas

XMPP makes use of two basic building blocks to interchange data between two entities: *XML streams* and *XML stanzas*.

```
<stream:stream
from="sensor1@example.com"
to="example.com"
version="1.0"
xmlns="jabber:client"
xmlns:stream="http://etherx.jabber.org/streams">
```

Figure 3.22: XMPP opening stream example.

```
</stream:stream>
```

Figure 3.23: XMPP closing stream example.

An XML stream is a message envelope that contains the XML elements that will be sent between two XMPP entities. An XML stream is established by the initiating entity (either a client or a server) and accepted by the receiving entity. An XML stream creates an unidirectional communication channel between the initiating and receiving entity. If a bidirectional communication is required, an additional XML stream must be created from the receiving entity to the initiating entity, which inverted roles.

Once an XML stream is established, any number of XML elements can be send within its scope. These elements include, for instance, security negotiation elements and XML stanzas.

An XML stream is bounded by the opening XML tag <stream> and its corresponding closing tag </stream>. Each tag is sent in a separate message. The first message starts the stream and contains the opening XML tag <stream> (Figure 3.22). The opening tag <stream> is considered as the "stream header' and contains the XMPP attributes and namespaces required to deliver and process the stream. A stream is finalized with a message containing the </stream> closing tag (Figure 3.23).

An XML stanza is XMPP's basic data unit and is used to enclose messages' payload. An XML stanza must be included as a first level XML element within the XML stream and be qualified with the namespaces "jabber:client" or "jabber:server".

XML stanzas are divided into three types: *message*, *presence* and *Info/Query* (IQ). *Message* stanzas are general purpose stanzas used to send data to a recipient while *IQ* stanzas are used as a request/response mechanism in order to request data from another XMPP entity. On the other hand, *presence* stanzas are used to broadcast network availability. Figure 3.24 shows an example of a *message* stanza.

The body of a XML stanza contains the payload information arranged and structured as required by the application and may be qualified by any XML namespace.

```
<message from="sensor1@example1.com/temperature"
to="client@example2.com">
<body>23</body>
</message>
```

Figure 3.24: XMPP *message* stanza example.

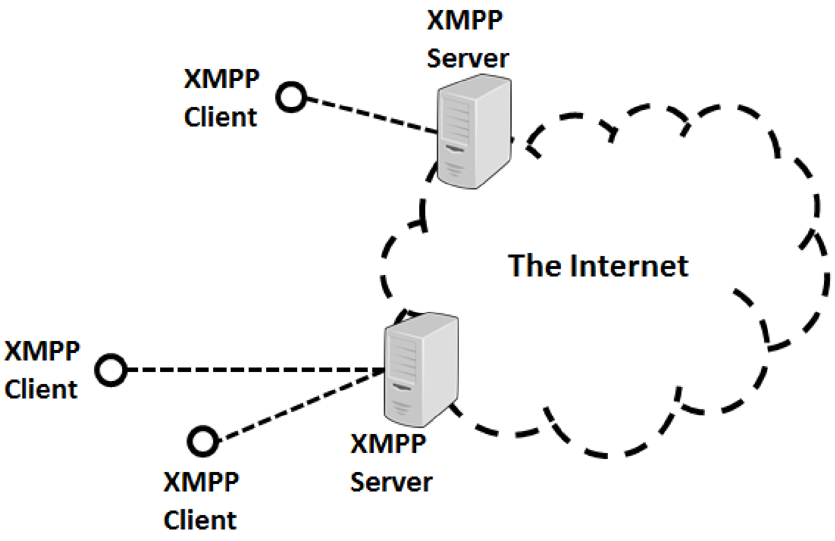


Figure 3.25: XMPP basic architecture.

3.3.3.2 XMPP Architecture

The architecture of XMPP is decentralized and is usually compared with the Internet Mail Architecture [Cro09] approach. XMPP architecture follows a distributed client/server approach (Figure 3.25). A XMPP client establishes a connection with a server in order to communicate with other XMPP entities. Two XMPP servers can connect to each other in order to provide a communication channel between two domains.

XMPP communications are connection oriented. XMPP uses persistent XML streams over long-lived TCP connections between XMPP entities (client-to-server and server-to-server) in order to provide a point-to-point transport layer. In this way XMPP entities are always ready to send, receive or relay XMPP messages. Additionally, these connections are characterized by a (typically) high number of concurrent messages between XMPP entities (clients and/or servers) that require a run time knowledge of network availability. The XMPP specification names this architecture approach as “Availability for Concurrent Transactions” (ACT).

XMPP messages are sent as XML "stanzas", each XML "stanza" representing a XML fragment within a stream. Each XML stanza includes routing attributes as well as the actual payload of the messages.

XMPP relies on globally unique addresses in order to route messages through the network. In order for an entity to be reachable within a XMPP network it must be address-

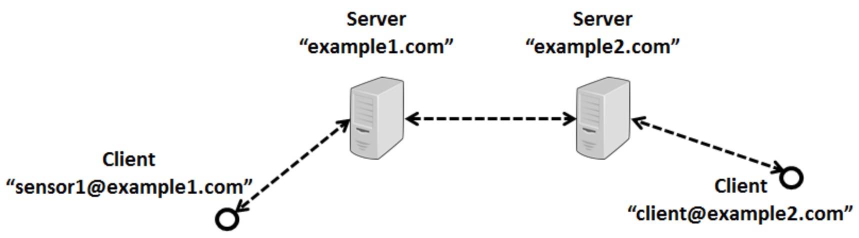


Figure 3.26: XMPP distributed architecture example.

able, i.e. be identified, by a globally unique address. This includes clients and servers but also other type of XMPP services. XMPP addresses follow the schema and format of email addresses. Server addresses are specified following the format <domainpart> (e.g., <example.com>). On the other hand, accounts follow the format <localpart@domainpart> (e.g., <sensor1@example.com>) and authorized resources assigned to an account <localpart@domainpart/resourcepart> (e.g., <sensor1@example.com/temperature>. The XMPP Address Format is defined in its own specification RFC 7622 [SA15].

The communication architecture of XMPP takes advantage of the global addresses in order to route messages through a distributed network of XMPP clients and servers. Clients send messages to other clients using intermediate servers as in a transparent and seamless way. Thus, message delivery in XMPP is end-to-end but physically client-to-server-to-server-to-client (Figure 3.26). This approach is similar to email delivery protocols such as SMTP [Kle08] and follows the general Internet Mail Architecture [Cro09].

Finally, one of the particularities of XMPP is that it has built-in advertisement of network availability also known as "presence". Presence is advertised end-to-end using dedicated primitives. This mechanism allows the detection at run-time of the presence (or absence) of an XMPP entity improving the overall network efficiency by avoiding sending messages to not-present entities. XMPP presence mechanisms are specified in [SA11b].

Figure 3.27 shows an example of a possible interaction between two XMPP entities, a client and a server. In this example, two XML streams are used during the session: one so the client can establish the connection and send messages to the server an another one for the server to send its responses.

The client initiates an XML stream and, in this example, so does the server in order to enable bidirectional communications. Then the client sends a presence stanza followed by a message stanza and an IQ request. Afterwards, the server sends the response to the clients IQ stanza. The client and server keep interchanging stanzas until, finally, the client closes the stream by sending a closing </stream> tag to the server in order to terminate the connection.

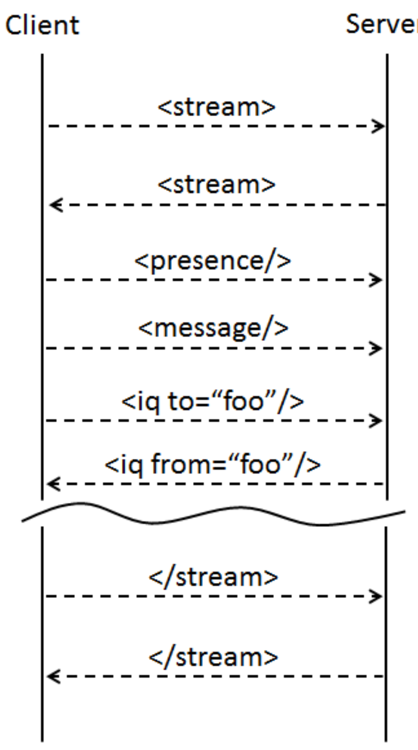


Figure 3.27: XMPP two streams example.

3.3.3.3 XMPP and EXI

XMPP defines an extension [WD16] to apply Efficient XML Interchange (EXI) Format to XMPP streams and stanzas. The intended use is for those application domains (such as sensor networks) where the constrains of the devices and networks do not allow for an efficient processing and transmission of XML.

Basically, the use of EXI together with XMPP requires a preliminary agreement on the set of parameters that will be used for the EXI codification. These parameters include practical encoding options such as data alignment. Among these parameters, the most important one is the schemas that will be used during the EXI processing because EXI can use schema information to drastically improve the compression ratio. The XMPP-EXI extension defines dedicated mechanisms to negotiate and inform the recipient entity of the XML schemas information that will be used in the EXI encoding.

During an XMPP-EXI connection negotiation, if the server informs the client that it does not have schema information, the client has two options: either the client performs schema-less compression or it uploads the missing schemas into the server. To upload the schema(s) a specific message is used, *uploadSchema*.

Optionally, the client can instruct the server to download the schemas from other sources with the *downloadSchema* message. An interesting property of this mechanism is that the schemas can be stored in dedicated servers and remove this responsibility from the client.

```
<exi:streamStart from='sensor1@example.com'
to='example.com'
version='1.0'
xml:lang='en'
xmlns:exi='http://jabber.org/protocol/compress/exi'>
<exi:xmlns prefix='' namespace='jabber:client'/>
<exi:xmlns prefix='streams' namespace='http://etherx.jabber.org/streams'/>
<exi:xmlns prefix='exi' namespace='http://jabber.org/protocol/compress/exi'/>
</exi:streamStart>
```

Figure 3.28: XMPP EXI streamStart example.

```
<exi:streamEnd xmlns:exi='http://jabber.org/protocol/compress/exi'/>
```

Figure 3.29: XMPP EXI streanEnd example.

The XMPP-EXI extension communications require specific opening and closing tags. EXI compression process requires for all XML elements to be closed before the compression takes place. Thus, XMPP streams cannot be started with the <stream:stream> tag if the closing tag </stream:stream> is left out, as in Figure 3.22. In order to make XMPP streams EXI compliant, in the XMPP-EXI extension the <stream> and </stream> opening and closing tags are replaced with the <exi:streamStart/> and <exi:streamEnd/> elements as shown in Figure 3.28 and Figure 3.29.

This change implies that XMPP-EXI and XMPP stream definitions are slightly different at the structure level although they share the same semantics. Additional preprocessing and post-processing is also required as stanzas and first level elements under <stream:stream> tags must be encoded as a standalone EXI body in XMPP-EXI streams. Furthermore, if namespaces are declared within the stream opening tag (<stream:stream> in XMPP and <exi:streamStart/> in XMPP-EXI) prefixes must be stored and preserved for their use in the stanzas.

Summarizing, the following modifications must be performed on the XML streams before performing the EXI compression. First, the <stream:stream> opening tag must be mapped to a standalone <exi:streamStart/> element, including attributes and namespaces. Next, all first level elements directly under the root <stream:stream> element must be mapped to standalone elements and missing namespace declarations must be added. Finally, </stream:stream> closing tag must be mapped to a standalone <exi:streamEnd/> element.

On the receiver side, the following steps must be performed on the EXI decoded XML streams before performing the XMPP parsing. First, the <exi:streamStart/> element must be mapped to a <stream:stream> opening tag including attributes and namespace declarations. Then, the EXI messages must be mapped to root elements within <stream:stream>

and namespaces must be mapped to the corresponding prefixes. Finally, <exi:streamEnd/> element must be mapped to a </stream:stream> closing tag.

3.3.4 Summary and Conclusions

Compression technologies for structured data require the use of schema or structure information in order to achieve the most efficient compression. However, these technologies do not deal with the transmission and interchange of the structural information. The transmission of this information is usually left to the underlying communication protocol or it is assumed that an out of band method is used.

Communication protocols targeted to constrained networks implement compact message formats and relatively simple mechanisms in order to cope with network restrictions. In some cases these protocols offer an optimized alternative to protocols usually used across the Internet such as 6LoWPAN [SB10] for IPv6, CoAP [SHB14] for HTTP or MQTT-SN [SCT13] for MQTT.

These protocols offer optimized mechanisms to discover, register, retrieve, reference and, in summary, manage certain resources in an optimized way. However, these mechanisms result in suboptimal solutions for context information sharing and for compact identifier assignment.

The Coap set of specifications include the defintion of the CoAP Resource Directory [SKB+18]. The specified approach offers mechanisms for the registration and lookup of stored web resources as well as provide discovery capabilities. The mechanisms provided by the Resource Directory could be used to design a schema management protocol. However, when a resource is registered, a path to the resource is returned as an identifier and a means to reference it. The format of this path results in a string that can produce unnecessarily large identifiers and negatively impact on the message length and, hence, network bandwidth.

MQTT-SN [SCT13] is a MQTT version tailored to the constrains and conditions typically found in sensor networks such as the particularities of wireless links, limited energy, low bandwidth, and short messages. In order to reduce the (potentially long) size of the MQTT topic, MQTT-SN replaces it with a shorter "topic id". MQTT-SN also defines a procedure to register topics at runtime and assign compact identifiers to them. However, the compact identifiers are limited to topics and cannot be used to identify resources or generic data.

XMPP (Extensible Messaging and Presence Protocol) [SA11a]) was born as an alternative to closed instant messaging services. Nowadays, XMPP has significantly extended its initial capabilities and it covers several application domains beyond instant messaging (e.g. voice and video calls, multi-party chat, cloud computing, etc.) as well as provide a lightweight middleware for the routing of arbitrary XML data. XMPP defines an extension [WD16] to apply Efficient XML Interchange (EXI) Format to XMPP streams.

The intended use is for those application domains (such as sensor networks) where the constrains of the devices and networks do not allow for an efficient processing and transmission of XML.

Basically, the use of EXI together with XMPP requires a preliminary agreement on the set of parameters that will be used for the EXI codification. These parameters include practical encoding options such as data alignment. The XMPP-EXI extension defines dedicated mechanisms to negotiate and inform the recipient entity of the XML schema information that will be used in the EXI encoding. XMPP-EXI provides convenient mechanisms for schema advertisement but it does not offer efficient schema identification and reference. XMPP-EXI uses the XML Schema namespaces in order to identify the schemas and their instances. XML namespaces tend to be verbose and negatively affect the data compression size and its transmission overhead.

There is a lack of standard and generic mechanisms to efficiently share schema information and assign compact identifiers. The communication model proposed in this Thesis provides all the required mechanisms and are generic enough to be applied to different compression technologies and underlying communication protocols.

4 Context- and Template-based Compression (CTC)

This chapter describes the core components of CTC, the codification algorithm and its application to two distinct data formats, XML and JSON, in order to show two practical and relevant examples.

The approach followed by CTC is to use a data representation encoding that is more efficient than standard data formats but that allows seamless transformation between the CTC format and the original format. CTC analyses the schemas of the data models in order to extract repeated structural portions, denoted templates. These templates are then used to perform a lossless-compression together with contextual information derived from the data, original data format and schema.

The main objective of CTC is to reduce the resources needed to transmit, store and process structured data compared to using standard text-based data formats. First, by compressing the structured data, the quantity of messages needed to transmit the whole data is effectively reduced. Second, the use of templates minimizes the memory needed to store the data models' schemas and the structures of the data. Finally, data is codified in a more efficient format, resulting in a reduction on the required processing time.

CTC is conceived as a part of a more complex distributed system. Figure 4.1 shows the simplified network architecture of such a system, which is similar to communication architectures found in traditional Low Power Wireless Personal Area Networks (LPWPAN) and CPS in general: resource-constrained devices are deployed in a dedicated network and an edge-router or gateway is used to access external networks (such as the Internet) and clients.

Devices interchange data with clients that either reside in the same local network or in external networks. Devices with constrained resources will be able to take advantage of CTC while more powerful devices use the original format at the same time. On the one hand, when both the resource-constrained device and the client implement CTC, the communication will be end-to-end, with the gateway acting as a mere router. On the other hand, if the client does not implement CTC and make use of the data models in their original format, the gateway will act as an application level gateway and translate the original format to CTC and vice-versa. CTC allows for the transformation between the two formats to be done in a transparent way so as not to break interoperability. The CTC communication model is described in detail in Chapter 5.

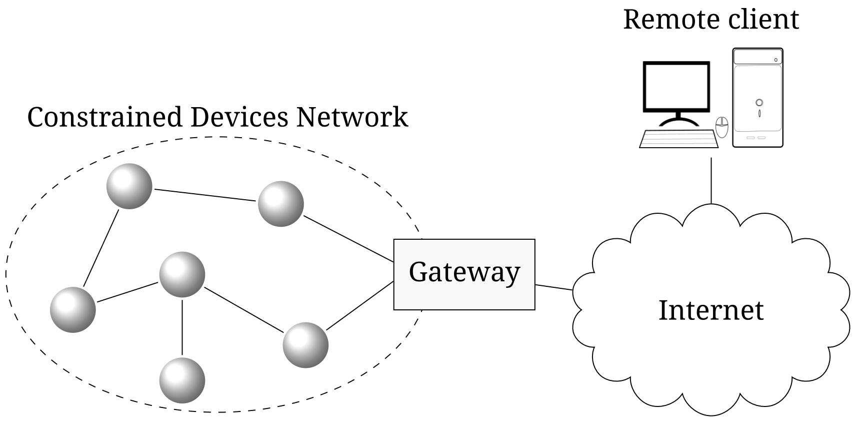


Figure 4.1: Simplified target network architecture of CTC.

4.1 CTC Components

CTC defines a data model structure representation that is able to describe the links between the items and templates that compose a data model. The proposed approach is intended to be generic and not tied to a specific data format. The data model's specific schema is used to extract a generic graph that is independent of the schema's original representation format as well as the templates used to build the schema instances. We denote this graph a *schema context*.

A *schema context* contains all the relevant schema information including individual nodes and links. This approach is similar to W3C Document Object Model (DOM [WWWCa]), which is one of the most popular data models for representing, storing, accessing and processing XML documents. DOM represents XML documents as a tree-structure where everything is a node: the document itself, elements, attributes, etc. DOM also specifies a low-level Application Programming Interface (API) for accessing, processing and modifying XML documents. In a *schema context*, data model schemas are also represented as graphs and the same terminology is used to refer to the relationships between nodes (parent, child, sibling, etc.).

However, unlike DOM, a *schema context* only considers two types of nodes: *Element*s and *eContext*s (short form of "Element Context"). An *Element* node encapsulates the properties of an item of the original schema and its associated template. For instance, an *Element* contains the cardinality and whether it is a basic type ("string", "integer", etc.). An *eContext* node basically groups child *Element* nodes. Depending on its type, an *Element* node may have an *eContext* which contains the list of child *Element* nodes. An *Element* with no *eContext* is a leaf of the *schema context* graph.

A simple *schema context* graph example is shown in Figure 4.2. The figure depicts the *eContext* and *Element* nodes, the links between them and associated templates. For

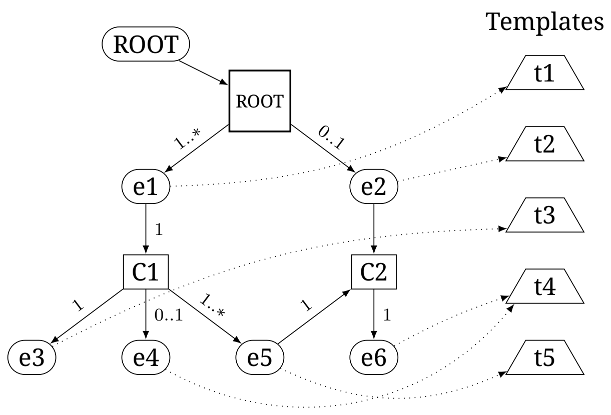


Figure 4.2: *Schema context* graph example. Rounded nodes denote *Element*s, square nodes *eContext*s and trapezium nodes templates. The numbers in the arrows indicate the cardinality: “1” one child, “1..*” one to many children, “0..1” none or one child (optional).

instance, *Element* “e1” has an *eContext* “C1” which in turn is the parent of child *Element*s “e3”, “e4” and “e5” with cardinalities “1”, “0..1” and “1..*” respectively. Additionally, “e3”, “e4” and “e5” *Element*s are linked to templates “t3”, “t4” and “t5” respectively. Note that *Element* “e4” shares its template (“t4”) with *Element* “e6”.

There are some other fundamental differences between DOM and *schema context*. DOM nodes only accept one parent (tree graph) while, in the *schema context*, a node may have multiple parents. Although DOM is conceived as a generic data model representation, it is specially targeted to XML and HTML formats while the *schema context* does not make any specific assumption regarding the original format. DOM is used to represent any type of XML document while *schema context* is only used to represent the data model schemas themselves, i.e., not the data. Additionally, DOM representation of XML documents consumes a lot of memory because the in-memory copy of a node keeps a lot of information and APIs tend to be heavy, producing verbose code. In contrast, the *schema context* is targeted to minimum memory footprint and the in-memory representation of a *schema context* only keeps the minimum information necessary to perform the codification.

CTC itself has two main components: the *context table* and the *template table*. The *context table* contains the *schema context*s while the *template table* is composed by the templates extracted from the schemas. Figure 4.3 shows a simplified representation of these two components. They are described with more detail in Section 4.1.1 and Section 4.1.2.

The encoding and decoding processes are executed following a specific algorithm, denoted *CTC Codification Algorithm*. In turn, the *CTC Codification Algorithm* uses the *context table* (or more specifically, the *schema context*s contained in the *context table*) as a reference in order to perform the encoding and decoding processes. The *CTC Codification Algorithm* is described in detail in Section 4.2.

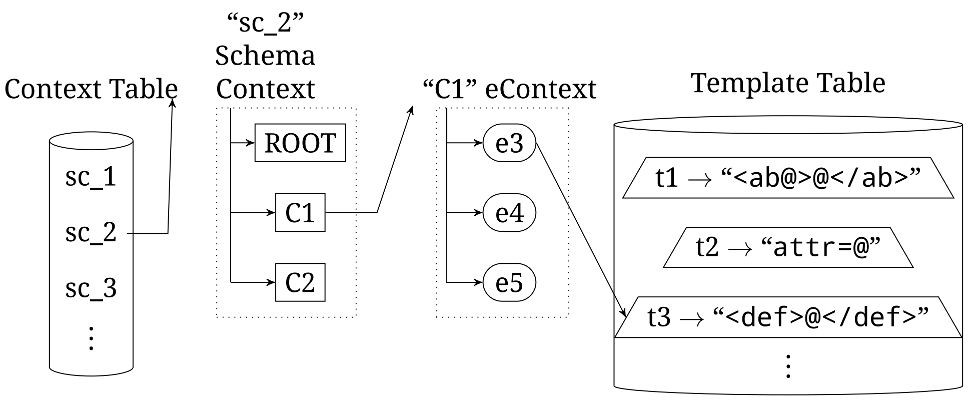


Figure 4.3: Example of representation of CTC components.

4.1.1 Context Table

The *context table* stores all the information of the data model schemas used by the device. Each entry of the *context table* is a *schema context* that contains the information related to the nodes in the schema, links between nodes, cardinality, links to templates and, in summary, all the information needed to process a data model instance described by the schema.

A *schema context* is identified by the *URI* (acronym for Uniform Resource Identifier) and *SchemaId* attributes. The *URI* attribute must be unique and it is used to globally identify the *schema context*. The *SchemaId* attribute is assigned at the device's bootstrapping phase (as described later) and must be unique within the (sub-)network the *schema context* is used (for example, within a wireless sensor local network).

A *schema context* is formally structured as a table where each entry is an *eContext*s node. In turn, each *eContext* entry contains a list with the child *Element* nodes. The first *eContext* of an *schema context* always belongs to the root *Element* node and indicates the entry point for the *CTC Codification Algorithm*.

Figure 4.3 shows a simplified representation of a *context table*, with the *template table* on the right side. The figure depicts a detail of a *schema context* with a *SchemaId* value of '2', represented as "sc_2", and *eContext*s "ROOT", "C1" and "C2". The figure also shows that *eContext* "C1" contains the child *Element*s "e3", "e4" and "e5" and that *Element* "e3" is linked to template "t3".

An *eContext* has the following attributes:

- `Id`: the unique identifier of the *eContext* node, which is denoted by the *eContext*'s entry index within the *schema context*.
- `MultipleParents`: TRUE if the *eContext* node is referenced by more than one *Element* nodes. FALSE otherwise.

- `Order`: the value of this attribute depends on the order the child nodes may appear in a data model instance document. If the order of the children is fixed and coincides with the order in which they are defined in the schema, the value is *fixed*. If the order is random, the value is *dynamic*. Finally, if only one single children can appear (among all the ones defined in the schema for that particular node), the value is *choice*.

- `Children`: it contains the list of child *Element* nodes.

The `MultipleParents` attribute allows the reuse of the same *eContext* node by more than one *Element* node. The result is a better leverage of the memory as it avoids unnecessary duplicities. On the other hand, the `Order` attribute is used to perform the most efficient encoding of the children, tailored to the schema's restrictions. This also avoids the parsing of irrelevant coded items improving the overall processing speed. The most efficient encoding is provided by *fixed* children, followed by *choice* and, finally, *dynamic* as the worst case. The *choice* order value is a special case of *dynamic* order, where the order is also unspecified but only one children can appear.

An *Element* node is composed of the following attributes:

- `Template`: a reference to the entry in the *template table* that contains the template for this *Element* node.

- `Type`: data type of the *Element* node. The data type can be either a basic type, a *constant*, a *complex* or a *schema*. For the basic type case, the following data types (inherited from the EXI [SKPK14] specification) are supported: *binary*, *boolean*, *decimal*, *float*, *integer*, *date-time* and *string*.

- `IsOptional`: TRUE in case the cardinality of the child *Element* is $0..m$, where $m > 0$, and FALSE otherwise.

- `IsArray`: TRUE in case the child *Element* can appear consecutively more than once, i.e., those children which have cardinality $n..m$, where $m > n$ and $m > 1$.

- `Context`: if the `Type` attribute is *complex*, `Context` attribute contains this child‘s *eContext*. In the case in which `Type` attribute is *schema*, the `Context` attribute is equal to the *schema context* that describes the schema. A special value is used to represent a “*any*” schema, i.e., an unspecified schema. Basic types and *constant* type do not make use of the `Context` attribute.

- `Separator`: separators are used to insert templates between lists of *Elements* or repeated *Elements*, such as the children of an *Element* node or *Elements* of type array. As in the `Template` attribute, Separator attribute contains a reference to the entry in the *template table* that represents the corresponding separator text string.

Table 4.1: *Schema context* table example. Each column represents an *eContext*. The content of the *Children* row represents the tuple (`Template`, `Type`, `IsOptional`, `IsArray`, `Context`). *x* denotes *complex*, *s string*, *f* FALSE and *t* TRUE.

Atribute	Id		
	ROOT(0)	C1(1)	C2(2)
MultipleParents	f	f	t
Order	fixed	dynamic	fixed

Children	e1(t1,x,f,t,C1)
	e2(t2,x,t,f,C2)
		e3(t3,s,f,f,-)
		e4(t4,s,t,f,-)
		e5(t5,x,f,t,C2)
			e6(t4,s,t,f,-)

The `Type` attribute is key in order to use the most efficient compression encoding for each data type. This also translates to average better processing performance as parsing/compressing with dedicated encoding is usually more efficient than processing plain string texts.

The `IsOptional` and `IsArray` attributes are used to codify the cardinality of the *Element* node. On one hand, `IsOptional` attribute identifies items that may not appear, removing the need to codify and process missing items. On the other hand, the `IsArray` attribute allows the codification of items' repetitions by reusing the same *Element* (and its template) without the need of in-memory duplications. Additionally, a template can be referenced by more than one *Element*. In this way, *template table* entries are reused when possible, reducing memory requirements.

Separators are useful for encoding templates related to *Element* lists that would otherwise need to be represented with nested *eContext*s and *Element*s. For example, in JSON the ',' character is used to separate the different items that compose the JSON document. This character is not part of the data model structure as such but it is needed to properly parse the JSON instance.

Table 4.1 shows the *schema context* table associated to the example data model *schema context* presented in Figure 4.2. For instance, as can be seen in Table 4.1, *Element* "e1" is linked to template "t1", is of *complex* type (thus, it has an *eContext*, "C1") and its a non-optional array (cardinality "1..*") with `IsOptional` to FALSE and `IsArray` to TRUE. As another example, *Element* "e4" is a leaf node (no *eContext*) of *string* basic type and it is optional (cardinality 0..1) with `IsOptional` to TRUE and `IsArray` to FALSE. Additionally, *Element* "e4" is linked to template "t4" together with *Element* "e6". The example assumes that no separator templates are used. Finally, note how *eContext* "C2" has `MultipleParents` attribute set to TRUE and its linked by *Elements* "e2" and "e5".

4.1.2 Template Table

The *template table* stores the list of templates of the schemas used by the device. Basically, templates are represented by using a character string format. The *template table* also contains the position of the place-holders that represent the extension/nesting points

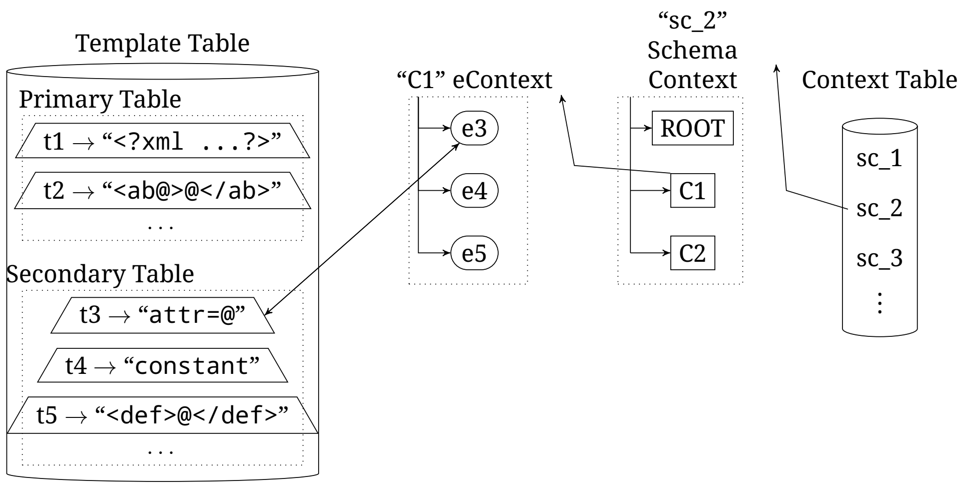


Figure 4.4: *Template table* structure detail with exemplary content.

for each template, i.e., where the templates of the child nodes or nested data models are inserted. Figure 4.4 shows the simplified *template table* structure with example content. In the figure, the place-holders of the example templates are represented with the character '@'.

The *template table* is structured and designed to provide efficient template searching and matching. As can be seen in Figure 4.4, the *template table* is divided into two sub-tables: *Primary Table* and *Secondary Table*. The *Primary Table* only contains the templates of the valid starting items of a data model, according to the structure described in the data model's schema. That is, the schema defines which items must appear first in a valid instance document and only the templates of these items are included in the *Primary Table*. For instance, in the XML case, the *Primary Table* will include the templates of the XML global elements. The *Secondary Table* contains the templates of all the remaining items.

In addition to the information related to the template representation, each table entry also contains information about the *Element* nodes that reference the template. This simplifies the matching between the original format, the templates, the *Element*s and their *eContext*s, thus, improving and optimizing the searching, matching and codification processes.

However, templates are only needed when data have to be transformed from/to the original format. As it will be explained later in Chapter 6, resource-constrained devices do not need to include the *template table*, reducing the memory needs. If the *template table* is needed by a device in order to transform from/to the original format, two distinct cases are considered: decoding and encoding.

When a coded stream is decoded to retrieve the data in the original format, the coded stream is parsed using the information in the *context table* and templates are merged together as the items are processed. In this case, the *template table* acts as a mere container of templates.

On the other case, when data in the original format have to be codified to CTC, the *template table* assumes a more active role. First, the *Primary Table* is used as the entry point for the encoding process and it is searched for a valid match. Once a match is found, the associated *Element*s and *eContext*s are recursively navigated until the full document is parsed. This searching strategy improves the search performance by reducing the searching range.

In summary, the *template table* has two main purposes. On the one hand, the *template table* is used during decoding phase to rebuild the codified data to its original format. On the other hand, it servers as a pattern matching reference during the codification process in order to search data in their original format, match the template patterns and, finally, extract the associated *Element* and *eContext* nodes.

4.1.3 Context Table and Template Table Creation

As explained in Section 4.1.1, the *context table* contains the list of all the *schema context*s used by the device. Each *schema context* is created by processing the individual data model schemas. As the schema is processed, the respective *eContext* and *Element* nodes are created for every item found in the schema.

In order to avoid confusion, this section uses the term *item* to refer to a node of the original schema (such as an “element” or “attribute” in XML or “property” in JSON) and the specific terms *Element* and *eContext* nodes to refer to the respective CTC nodes of the *schema context*.

There are five pieces of basic information per item that need to be extracted from the data model schema: (a) the links between the items, (b) the cardinality of those links, (c) the order in which the child items can appear, (d) the data type of the item and, finally, (e) the template and separator used to represent the structure of the item in the original data format. How this information is processed and gathered from the schema is described in detail in the following paragraphs, together with Algorithm 1.

If the cardinality of the item is 1, the `IsOptional` and `IsArray` attributes are set to FALSE. If the item has cardinality $n..m$, where $n = 0$, the `IsOptional` attribute is set to TRUE. In the case the cardinality is $m > 1$, the `IsArray` attribute is also set to TRUE. Then, the template is added to the *template table* and the `Template` attribute is set to the assigned index within the *template table*. If the item is an array (i.e., $IsArray =$ TRUE), the corresponding separator used by item lists is assigned to the `Separator` attribute.

Once cardinality attributes and templates have been processed, the item’s type is added to `Type` attribute. In case the item’s type is *complex*, the algorithm checks whether its *eContext* node already exists within the *schema context*. In case the *eContext* node already exists, its `MultipleParents` attribute is set to TRUE.

```
forall $Item\ in\ Schema$ do
    $Element$ = Create();
    if MinCardinality($Item$) $== 0$ then $Element.IsOptional = $ TRUE ;
    if MaxCardinality($Item$) $> 1$ then
        $Element.IsArray = $ TRUE;
        $Element.Separator$ = AddSeparator($TemplateTable, Item$);
    $Element.Template$ = AddTemplate($TemplateTable, Item$);
    $Element.Type$ = GetType($Item$);
    if $Element.Type == complex$ then
        $Element.Separator$ = AddSeparator($TemplateTable, Item$);
        $Element.Context$ = FindEContext($SchemaContext, Item$);
        if $Element.Context \neq NULL$ then
            $Element.Context.MultipleParents = $ TRUE;
        else
            $Element.Context$ = Create();
            $Element.Context$.Order = GetOrder($Item$);
            AddEContext($SchemaContext, Element.Context$);
    else if $Element.Type == schema$ then
        $Element.Context$ = FindSchemaContext($ContextTable, Item$)
    AddChild($Parent.Children, Element$)
```

Algorithm 1: *Schema context* creation

On the contrary, if the *eContext* node does not exist, a new *eContext* node is created and its `MultipleParents` attribute is set to FALSE. In case (a) the order of appearance of children is fixed and (b) the appearance matches the order defined in the schema, `Order` attribute is set to *fixed*. If the appearance order of the children can vary dynamically, `Order` attribute is set to *dynamic*. In case only one of the children can appear, `Order` attribute is set to *choice*. Additionally, the corresponding separator for the child *element* nodes is assigned to the `Separator` attribute. After the *eContext* node is composed, it is added to the *schema context*.

If the item's type is *schema*, the `Context` attribute is set to the associated *schema context*. For those cases where the nested schema is unknown a priory, the `Context` attribute is set to the special value "*any*".

Finally, the new child *element* node is added to the *eContext*'s `Children` attribute of the parent *Element* node.

Once the schema has been processed and the *schema context* has been created, a process called *Context Collapsing* is performed. This process reduces the number of *eContexts*, *Elements* and templates without any loss of information. If (1) the $Type$ attribute of an *eContext* node's and a child *Element* node's *eContext* are both *fixed*, (2) the child *Element* is neither optional nor array (i.e., $IsOptional =$ FALSE and $IsArray =$ FALSE), and (3) the child's *eContext* only has one parent (i.e., $MultipleParents =$ FALSE), then the *eContext* and template of the child *Element* are merged together with the *eContext* and template of the parent *Element*, including separators.

Context Collapsing is executed starting from the root node in a recursive way, for each *eContext* and its child *Element*s. During a *Context Collapsing*, nested *eContext* nodes of order *fixed* are merged together, including the associated templates and separators. This process optimizes the effective processing time by reducing the necessary iterations and accesses to the *schema context* and *template table*.

4.1.4 Schema Mapping

The previous section described the general algorithm and approach to create a *schema context* from a generic data model schema. Although the algorithm is generic, it has to be specifically implemented for each data format type, as the mapping of the schema to a *schema context* is data format specific. This section describes the specific case of the algorithm application to XML Schema and JSON Schemas.

Although a full detailed explication of the mapping of every single node type described in the XML Infoset and JSON Schema specification is out of the scope of this document, we give here an overview of the most relevant and representative cases.

4.1.4.1 XML Schema Mapping

XML complex and simple elements are transformed into CTC *eContext*s. The `Order` will vary depending on whether the containers' XML order indicator is "All" (*dynamic*), "Choice" (*choice*) or "Sequence" (*fixed*).

XML element particles are mapped as *Element*s. The cardinality of the *Element* will depend on the XML occurrence indicators "maxOccurs" and "minOccurs".

XML attributes are mapped in a similar way as XML elements, but they are grouped into a single child *Element* with a dynamically ordered *eContext*. The separator of this *Element* is set to a single space character (' ').

Templates are extracted from the XML elements and attributes, and formatted according to their nature. XML global elements (described in the Appendix A.1.5) are stored in the *Primary Table* while any other element is assigned to the *Secondary Table*.

An optional child *Element* containing the XML prolog is always added to the root *eContext*, followed by any relevant global definition (such as namespaces and prefixes). Global XML elements and attributes are also added as *Element*s to the root *eContext*.

Each XML namespace is transformed into a different *schema context*. If the XML type of an XML element or attribute belongs to a namespace other than the current one, the *Element*'s `Type` will be of *schema* type and the `Context` will be assigned to the *schema context* of the relevant namespace. If the XML element or attribute is of "<*any*>" type, the *Element*'s `Type` will also be of *schema* type but the `Context` is specially marked to represent the special value "*any*".

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                    elementFormDefault="qualified">
    <xs:element name="notebook">
        <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
                <xs:element name="note" type="Note"/>
            </xs:sequence>
            <xs:attribute ref="date"/>
        </xs:complexType>
    </xs:element>
      <xs:complexType name="Note">
        <xs:sequence>
            <xs:element name="subject" type="xs:string"/>
            <xs:element name="body" type="xs:string"/>
        </xs:sequence>
        <xs:attribute ref="date" use="required"/>
        <xs:attribute name="category" type="xs:string"/>
      </xs:complexType>
    <xs:attribute name="date" type="xs:date"/>
</xs:schema>
```

Figure 4.5: *Notebook* XML Schema document.

XML Built-in Data Types are mapped to the corresponding CTC basic types, i.e., *binary*, *boolean*, *decimal*, *float*, *integer*, *date-time* and *string*. As explained in Section 4.1.1, these types are inherited from the EXI [SKPK14] specification and are described in Section 3.2.1.1.

Context Table and Template Table Example. We present an example of a *schema context* and *template table* generated from an XML Schema. To this end, we use the *Notebook* XML document example proposed by Peintner et al. [PPG14]. Figure 4.5 shows the original *Notebook* XML Schema example. Figure 4.6 shows the *template table* generated before performing *Context Collapsing* (see Figure 4.6a) as described in Section 4.1.3, and after *Context Collapsing* (see Figure 4.6b).

As can be appreciated in Figure 4.6, templates are merged together after *Context Collapsing* is performed, eliminating in the process the unneeded *eContext* and *Element* nodes. The result is a more compact *schema context* and *template table*. Finally, Table 4.2 shows the *schema context* generated after *Context Collapsing*. This table is related to the *template table* shown in Figure 4.6b and contains the *eContext*s and *Element*s after pruning the unneeded items.

4.1.4.2 JSON Schema Mapping

This section describes the mapping of the *schema context* creation algorithm for the specific case of JSON Schema.

The mapping for the JSON Schema of the *schema context* creation algorithm is based on the Draft-04 version (i.e., “http://json-schema.org/draft-04/schema#”) of the JSON Schema

```
---------------------------------------- t1
<?xml version="1.0" encoding="UTF-8"?>
---------------------------------------- t2
<notebook @>
@
</notebook>
---------------------------------------- t3
date=@
---------------------------------------- t4
<note @>
@
@
</note>
---------------------------------------- t5
category=@
---------------------------------------- t6
<subject>@</subject>
---------------------------------------- t7
<body>@</body>
```

(a) No *Context Collapsing*.

```
---------------------------------------- t1
<?xml version="1.0" encoding="UTF-8"?>
---------------------------------------- t2
<notebook @>
@
</notebook>
---------------------------------------- t3
date=@
---------------------------------------- t4
<note @>
<subject>@</subject>
<body>@</body>
</note>
---------------------------------------- t5
category=@
```

(b) With *Context Collapsing*.

Figure 4.6: *Template table Notebook* example. Symbol '@' is used to represent the placeholders' positions. Templates 't1', 't2' and 't3' are stored in the *Primary Table*.

Table 4.2: *Schema context* Notebook example, after *Context Collapsing*. The content of the *Children* row represents the tuple (*Template*, *Type*, *IsOptional*, *IsArray*, *Context*). Cn represents the *eContext Id* and tn the template identifier, x denotes *complex*, s *string*, c *constant*, d *date-time*, t TRUE and f FALSE. No separators are used in this example.

Attribute	Id				
	C1 (ROOT)	C2 (CONTENT)	C3 (NOTEBOOK)	C4 (NOTE)	C5 (NOTE-ATT)
MultipleParents	f	f	f	f	f
Order	fixed	choice	fixed	fixed	dynamic
Children	(t1,c,t,f,-) (-,x,f,f,C2)	(t2,x,t,f,C3) (t3,d,t,f,-)	(t3,d,t,f,-) (t4,x,f,t,C4)	(-,x,f,f,C5) (-,s,f,f,-) (-,s,f,f,-)	(t3,d,f,f,-) (t5,s,t,f,-)

Specification [GZC13] and JSON Schema Validation vocabulary [ZC13]. We describe the mapping for the Draft-04 version as it is still widely used, in contrast to more recent ones (Draft-07 [WA18, WAL18] at the time of this writing). However, following the principles described here, it is straightforward to extend the mapping specification for the Draft-07 version.

The purpose of this section is not to provide a thorough technical description of the *schema context* creation from a JSON Schema, but to give an outline of the most relevant and representative application cases. Thus, all the possible keywords and item types may not be covered.

First, the JSON root schema is converted into a new *schema context* and the "id" JSON attribute is assigned as the unique identifier, i.e, the URI. The root *eContext* node is also added with the `Order` attribute set to *choice*.

The subsequent JSON sub-schemas are progressively converted to *eContext* nodes. The order of the *eContext* node will be *dynamic* by default as JSON specification does not enforce any specific ordering. JSON sub-schemas including the keywords “allOff” or “anyOff” are also converted to *eContext* nodes of order *dynamic*. However, if the keyword “oneOff” is included, the order assigned to the *eContext* node is *choice*.

If the JSON Schema contains a “definitions” sub-schema, a new *eContext* node is added as a direct child to the root *eContext* node. All the productions of the sub-schemas within the “definitions” sub-schema (i.e., its children) are added as children to the newly added “definitions” *eContext* node. The order of the “definitions” *eContext* node is set to *choice*.

The properties of JSON sub-schemas of type “object” are mapped to CTC *Element* nodes. The cardinality of the *Element* node will depend on the “type” keyword of the JSON property and the “required” keyword of the parent sub-schema. If the type is “array”, then the *Element* node’s `IsArray` attribute is set to TRUE. On the other hand, if the JSON property is listed within the “required” keyword of the parent sub-schema, then the *Element* node’s `IsOptional` attribute is set to FALSE.

If the sub-schema includes a “$ref” keyword pointing to a relative URI within the JSON schema, the CTC *Element* node’s `Type` attribute is assigned as *element*. On the contrary, if the scope of the “$ref” keyword URI is outside the current schema (i.e., it points to another JSON Schema), the *Element* node type is assigned to *schema* and the *eContext* node of the *Element* node will belong to a different *schema context*.

The remaining JSON base primitive types (boolean, integer, number and string) are mapped to the corresponding CTC basic types. As explained in Section 4.1.1, CTC basic types are *binary*, *boolean*, *decimal*, *float*, *integer*, *date-time* and *string*. These types are inherited from the EXI [SKPK14] specification and are described in Section 3.2.1.1.

The template of the root schema is included in the *Primary Table* of the *template table*. The templates of the other sub-schemas are stored in the *Secondary Table*.

Finally, in the case of JSON sub-schemas or properties of types “array” or “object”, the separator of the corresponding *Element* node will be set to the ‘,’ (coma) character.

JSON Context Table and Template Table Example. In order to clarify the JSON Schema mapping, we present here an example of a *schema context* and *template table* generated from a JSON Schema. The example JSON Schema used is derived from the *Notebook* XML document example proposed by Peintner et al. [PPG14]. Figure 4.7 shows the *Notebook* JSON Schema, which is semantically equivalent to the original *Notebook* XML Schema example proposed by Peintner et al.

Table 4.3 shows the *template table* generated after performing the process described in Section 4.1.3. As can be seen, the JSON schema has been split in the independent templates that conform a JSON document instance of the *Notebook* JSON Schema.

Table 4.4 shows the *schema context* generated from the *Notebook* JSON Schema. This table includes the *eContext*s and *Element*s as well as references to the templates shown in Table 4.3.

```
{
  "$schema": "http://json−schema.org/draft−04/schema#",
  "id": "notebook.schema04.json",
  "type": "object",
  "properties": {
    "notebook": {
      "type": "object",
      "properties": {
        "date": {
          "$ref": "#/definitions/date"
        },
        "Notes": {
          "type": "array",
          "minItems": 1,
          "items": {
            "$ref": "#/definitions/Note"
          }
        }
      },
      "required": ["Notes"],
      "additionalProperties": false
    }
  },
  "required": ["notebook"],
  "additionalProperties": false,
  "definitions": {
    "Note": {
      "type": "object",
      "properties": {
        "date": {
          "$ref": "#/definitions/date"
        },
        "category": {
          "type": "string"
        },
        "subject": {
          "type": "string"
        },
        "body": {
          "type": "string"
        }
      },
      "required": ["date", "subject", "body"],
      "additionalProperties": false
    },
    "date": {
      "type": "string",
      "format": "date−time"
    }
  }
}
```

Figure 4.7: *Notebook* JSON Schema document.

Table 4.3: CTC Templates generated from the *Notebook* JSON Schema. Symbol '@' is used to represent the place-holders' positions. Template 't1' is stored in the *Primary Table*.

t1	{"notebook":{@}}
t2	,
t3	"date":"@"
t4	"Notes":[@]
t5	{@}
t6	"category":"@"
t7	"subject":"@"
t8	"body":"@"

Table 4.4: *Schema context* generated from the *Notebook* JSON Schema. The content of the *Children* row represents the tuple (*Template*, *Separator*, *Type*, *IsOptional*, *IsArray*, *Context*). Cn represents the *eContext Id* and tn the template identifier, x denotes *complex*, s *string*, d *date-time*, t TRUE and f FALSE.

Attribute	Id					
	C1 (ROOT)	C2 (NOTEBOOK)	C3 (DEFINITIONS)	C4 (NOTES)	C5 (NOTE)	C6 (DATE)
Order	choice	dynamic	choice	dynamic	dynamic	dynamic
Children	(t1,t2,x,f,f,C2) (-,-,x,t,f,C3)	(t3,-,d,t,f,-) (t4,t2,x,f,f,C4)	(-,-,x,t,f,C5) (-,-,x,t,f,C6)	(t5,t2,x,f,t,C5)	(t3,-,d,f,f,-) (t6,-,s,t,f,-) (t7,-,s,f,f,-) (t8,-,s,f,f,-)	(-,-,d,f,f,-)

4.1.4.3 Other Data Model Representation Formats

In this Thesis, XML and JSON data formats are used as relevant application examples to show the capabilities and mechanisms that conform CTC. However, CTC can be used with any other data model representation format as long as the information regarding the structure of the data model can be extracted from a schema or by any other means. For instance, the principles and approaches developed within this Thesis could be applied to other text-based data formats such as Hypertext Markup Language (HTML [FEL+17]) or the Resource Description Framework (RDF [WWWCc]) set of recommendations. In a similar way as with the XML and JSON cases, the structure can be extracted and used by CTC to build the *context table* and *template table* that will be used for the CTC compression and management processes.

As the complexity of the format and related schema grows, so does the CTC schema mapping and encoding processes. However, this complexity is mostly concentrated in the mapping of the schema to the *context table* and *template table*. Additionally, different techniques can be applied (such as the *Context Collapsing* method) to the tables building in order to relieve the resource-constrained devices from the runtime overhead.

4.2 CTC Codification Algorithm

In this section we describe the generic rules that, applied together, form the *CTC Codification Algorithm*, used to perform the encoding and decoding processes. The rules define the actions to perform for each node, based on the information available in the *context table*. The rules are grouped and formalized using a set of equations that represent the different steps involved in the encoding/decoding process of each node.

4.2.1 Rules

We define the following terms:

- We denote $e_0 \ldots e_{n-1} \in C$ as the ordered list of child *Element*s of the *eContext* C where n is the total number of C's children.
- We denote $e'_0 \ldots e'_{m-1} \in C$ as the unordered list of child *Element*s of the *eContext* C, where m, $m \leqslant n$, is the number of C's children actually appearing in the data.
- The $Trim(x, y)$ function trims the representation of x to $\lceil \log_2 y \rceil$ bits. Optionally, the form x_y is also used to represent x with y bits. Thus, $Trim(x, y) = x_y$.
- The symbol $\oplus$ represents the concatenation of two bit arrays.
- The $Pos(C, e)$ function returns the position index of the child *Element* e within *eContext* C. Note that, for an ordered child e, $Pos(C, e_i) = i$ where $0 \leqslant i < n$. However, for an unordered child e', $Pos(C, e'_i) = i$ MAY not be TRUE.

Four set of rules are defined together with the corresponding equations: $Cod_S(s)$ for *schema context*s, $Cod_{EC}(C)$ for *eContext*s, $Cod_E(C, e)$ for children *Element*s and $Cod_T(e)$ for data types. $Cod_S(s)$ is always applied first.

Rule1: if the *SchemaId* of a (nested) schema is not known a priory (i.e., is of "*any*" type), the *SchemaId* must be codified before the root *eContext* of the schema is processed. Otherwise, the codification of the root *eContext* of the (nested) schema is processed directly.

$$Cod_S(s) = \begin{cases} Cod_{EC}(s) & SchemaId(s) \neq any \\ \\ SchemaId(s) \oplus Cod_{EC}(s) & SchemaId(s) = any \end{cases} \tag{4.1}$$

At the beginning of a coding/decoding process, Equation 4.1 is always used first. Thus, all CTC streams start with the *SchemaId* of the data model's schema, followed by the root *eContext*.

Rule2.a: if the order of the child *Element*s is fixed, the codification of the *eContext* is equal to the concatenation of the children's codification, following the same order the children are defined in the `Children` list attribute.

Rule2.b: if the order of the child *Element*s is independent of the order defined in the schema, a prefix equal to the child *Element*'s index plus 1 is added to the codification of each of the children. If not all the children are present, a prefix of 0_{n+1} is used to indicate the end of the children list, where n in the children quantity.

Rule2.c: if only one of the children can appear, a prefix equal to the child *Element*'s index is added to the codification of the children.

The following equation groups Rules 2a, 2.b and 2.c.

$$Cod_{EC}(C) = \begin{cases} Cod_E(C, e_0) \oplus \ldots \oplus Cod_E(C, e_{n-1}) & ContentType(C) = fixed \\ \\ Trim(Pos(C, e'_0) + 1, n + 1) \oplus Cod_E(C, e'_0) \oplus \ldots & \\ \ldots \oplus Trim(Pos(C, e'_i) + 1, n + 1) \oplus Cod_E(C, e'_i) & ContentType(C) = dynamic \\ \oplus Trim(0, n + 1) & \\ \\ Trim(Pos(C, e'_i), n) \oplus Cod_E(C, e'_i) & ContentType(C) = choice \end{cases} \tag{4.2}$$

Cod_{EC} is used to codify *eContext*s. As can be seen in Equation 4.2, the codification of an *eContext* depends mainly on the Type attribute. CTC defines a strict mode where the items of a schema are always codified strictly following the order defined in the schema. In this mode, all the *eContext* nodes where condition $Type = dynamic$ applies, are considered to be *fixed*. The strict mode provides a more compact compression at the cost of some of the flexibility of CTC. However, this mode is ideal for resource-constrained devices, as it is straightforward for the device to codify the data models respecting the items' definition order.

Rule3.a: if an *Element* is not an array nor optional, the codification is equal to the codification of the *Element*'s Type.

Rule3.b: if an *Element* is optional but not an array, a 1_1 prefix is added to the codification, followed by the *Element*'s type codification. In case the Type of the parent *eContext* is not *fixed*, the prefix is omitted. If the optional *Element* does not appear, a 0_1 will be added to the codification.

Rule3.c: if an *Element* is an array, a 1_1 prefix is added to each of the *Element* occurrences and a 0_1 is added when no more occurrences remain.

Equation 4.3 groups Rules 3.a, 3.b and 3.c.

$$Cod_E(C,e) = \begin{cases} 1_1 \oplus Cod_T(e) & (IsOptional(e) = \text{TRUE}) \;\&\; (IsArray(e) = \text{FALSE}) \\ & \;\&\; (e \neq null) \;\&\; ContentType(C) = fixed \\ 0_1 & (IsOptional(e) = \text{TRUE}) \;\&\; (e = null) \\ 1_1 \oplus Cod_T(e) \oplus \ldots & \\ \ldots \oplus 1_1 \oplus Cod_T(e) & (IsArray(e) = \text{TRUE}) \;\&\; (e \neq null) \\ \oplus\, 0_1 & \\ Cod_T(e) & otherwise \end{cases} \tag{4.3}$$

Finally, the following rules are used to process the *Element*'s type:

Rule4.a: if the *Element* is a basic type, the inherited built-in EXI data type representation is used to codify the *Element*'s value. These data types are described in Section 3.2.1.1.

Rule4.b: when the *Element* is of *complex* type, the equation Cod_{EC} is used to codify the *Element*'s context.

Rule4.c: if the *Element* is of *schema* type, the equation Cod_S is used to codify the *Element*'s context.

$$Cod_T(e) = \begin{cases} Cod_{EC}(Context(e)) & Type(e) = complex \\ Cod_S(Context(e)) & Type(e) = schema \\ EXI\_basic\_type(e) & otherwise \end{cases} \tag{4.4}$$

4.2.2 Example

In order to clarify the application of the rules and equations explained in the previous section, the step by step codification of the XML instance document shown in Figure 4.8 (which follows the "Notebook" schema of Figure 4.5) is described here. For simplicity reasons, the example below only expands the first occurrence of the XML element "*note*".

First, the *SchemaId* of the schema is codified, followed by the root *eContext*:

$$\boxed{Cod_S(s_{\mathit{NOTEBOOK}}) \Rightarrow SchemaId(s_{\mathit{NOTEBOOK}}) \oplus Cod_{EC}(C_{\mathit{ROOT}})}$$

Next, the prolog *Element* of the root *eContext* is processed, followed by the content of the data model instance:

```
<?xml version="1.0" encoding="UTF-8"?>
<notebook date="2007-09-12">
    <note category="EXI" date="2007-07-23">
        <subject>EXI</subject>
        <body>Do not forget it!</body>
    </note>
    <note date="2007-09-12">
        <subject>Shopping List</subject>
        <body>milk, honey</body>
    </note>
</notebook>
```

Figure 4.8: Schema instance example.

$$Cod_{EC}(C_{ROOT}) \Rightarrow Cod_E(C_{ROOT}, e_{PROLOG}) \oplus Cod_E(C_{ROOT}, e_{CONTENT})$$
$$Cod_E(C_{ROOT}, e_{CONTENT}) \Rightarrow Cod_T(e_{CONTENT}) \Rightarrow Cod_{EC}(C_{CONTENT})$$

The "*notebook*" XML element is codified into the stream taking into account that the *eContext*'s Type is *choice*:

$$Cod_{EC}(C_{CONTENT}) \Rightarrow 0_1 \oplus Cod_E(C_{CONTENT}, e_{notebook})$$
$$Cod_E(C_{CONTENT}, e_{notebook}) \Rightarrow Cod_T(e_{notebook}) \Rightarrow Cod_{EC}(C_{notebook})$$

The "*notebook*" *eContext* contains two *Elements*, one for the XML attributes and another for the "*note*" XML element:

$$Cod_{EC}(C_{notebook}) \Rightarrow Cod_E(C_{notebook}, e_{notebook\_att}) \oplus Cod_E(C_{notebook}, e_{note})$$

The "*note*" *Element* is an array with length two:

$$Cod_E(C_{notebook}, e_{note}) \Rightarrow 1_1 \oplus Cod_T(e_{note}) \oplus 1_1 \oplus Cod_T(e_{note}) \oplus 0_1$$

The "*note*" *eContext* contains three *Elements*, one for the XML attributes and another two for the "*subject*" and "*body*" XML elements:

$$Cod_T(e_{note}) \Rightarrow Cod_{EC}(C_{note}) \Rightarrow Cod_E(C_{note}, e_{note\_att}) \oplus Cod_E(C_{note}, e_{subject}) \oplus Cod_E(C_{note}, e_{body})$$

The "*note_att*" *eContext* contains the attributes of the "*note*" XML element. It is a *dynamic eContext* with two child *Elements*:

$$Cod_E(C_{note}, e_{note\_att}) \Rightarrow Cod_T(e_{note\_att}) \Rightarrow Cod_{EC}(C_{note\_att}) \Rightarrow$$
$$\Rightarrow 1_2 \oplus Cod_E(C_{note\_att}, e_{date}) \oplus 2_2 \oplus Cod_E(C_{note\_att}, e_{category})$$

Finally, basic type *Elements* are directly encoded using the EXI codification standard for built-in EXI data type representations. For instance, for the "*subject Element* of type *string*, the value is codified as:

$$Cod_E(C_{note}, e_{subject}) \Rightarrow Cod_T(e_{subject}) \Rightarrow 3_8 \oplus \text{``}EXI''$$

4.3 Summary and Conclusions

In this chapter, we presented *Context- and Template-based Compression* (CTC), a compression approach for standard data model representation formats. CTC provides a data model representation encoding targeted to resource-constrained devices that is more efficient than standard formats but that allows seamless transformation between the CTC format and the original format. The specification of the core components of CTC (*context table* and *template table*) is included as well as how these core components are created from standard data format schemas. We also provided two specific examples for XML and JSON Schema mappings. Finally, the chapter described in detail the CTC Algorithm used to encode/decode CTC streams based on the information stored in the *context table* and *template table*.

The verbosity of text-based data formats requires system resources that might be beyond the capabilities of the resource-constrained devices typically used into IoT networks. CTC tackles this problem by enabling the interoperable integration of heterogeneous devices at the data representation level while requiring very low resource needs in terms of communication bandwidth, memory usage and processing power.

Additionally, CTC supports interoperability-driven approaches such as the Web of Things. CTC eases the seamless use of Web Services by enabling the native use of standard data model representation formats Web Services are based on.

5 CTC Communication Model

The previous chapter focused on the description of the core CTC components and mechanisms. However, CTC coding/decoding components alone do not provide all the functionalities needed to be directly used together with a distributed application. This chapter describes the CTC communication model, how it fits within a distributed system, and the complementary mechanisms needed to be effectively used.

Although the CTC communication model is mainly designed to be used by resource-constrained devices, it is very flexible and simple. The CTC communication model is easily adapted to various scenarios and in conjunction with distinct technologies, targeted to resource-constrained domains or not.

Hence, although in this chapter we assume that the compression technology used is CTC, the proposed solution can be also applicable to other data compression technologies for structured data, such as EXI or CBOR.

The following sections describe the general communication architecture followed by CTC enabled systems, the complementary mechanisms needed to manage the interchange of schemas as well as a specific and practical implementation of CTC based on CoAP to show the applicability of the CTC communication model on a standard communication protocol.

5.1 Communication Architecture

CTC is conceived as a component within a distributed system such as the one shown in Figure 5.1: connected nodes (usually resource-constrained devices) are deployed in a local network and an edge-router or gateway is used to access external networks and nodes. This architecture is similar to communication architectures found in traditional Low Power Wireless Personal Area Networks (LPWPAN) and the IoT in general.

CTC communication architecture can be integrated into networks and architectures with other topologies such as clusters of local networks or two local networks connected by an Internet link. Nevertheless, this section considers the basic architecture depicted in Figure 5.1 because it is easily scalable and extrapolated to other, more complex, architectures.

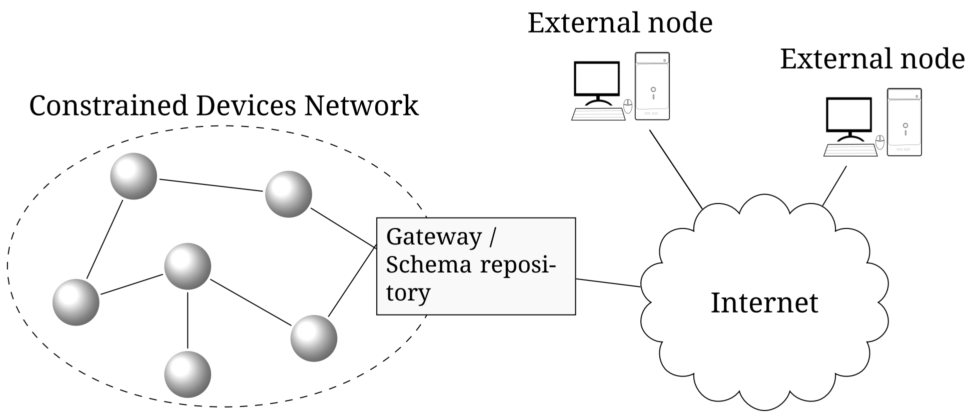


Figure 5.1: CTC communication model general architecture.

Depending on the application domain, nodes belonging to a (sub-)network interchange data with other nodes that may reside in the same (sub-)network or in an external network, i.e., a network accessed through an edge-router. If two connected nodes codify the transmitted data following the same encoding/format, no data transformation will be required in order for the two nodes to understand each other's data. If the two nodes are separated by a gateway, no application level transformation will be needed and the communication will be effectively end-to-end, with the gateway acting as a mere router. This is the simplest communication use case.

However, compression technologies targeted to resource-constrained systems (such as CTC) are especially conceived for those cases in which an (external) node uses a data format not suitable for resource-constrained nodes. Thus, data needs to be translated to CTC in order to be efficiently used within the constrained nodes' network. In this case, one of the connected nodes does not implement CTC (i.e., it makes use of data in their original format) and the gateway will act as an application level gateway, translating the original data format to CTC and vice-versa. In order for the gateway to fulfil this role, it needs to meet three requirements: 1) it must contain a CTC implementation, including schema management, 2) it must have access to the data (i.e., the payload of the messages) and 3) it must have access to the schema information of the interchanged data.

Regarding the third requirement, CTC enabled gateways and nodes need to know the *context table*s and *template table*s (and their identifiers) associated with the data models they are using. As explained in Chapter 4, this information is extracted from the schemas of the data models themselves. Thus, the schemas of the data models must be disseminated before CTC can be applied. Additionally, schemas must be uniquely identified within the CTC enabled (sub-)network. This requirement is because, in order to decode a CTC stream, the identifier of the schema against it has been encoded must be inserted in the stream itself. This identifier must be as compact as possible (as opposed to traditional URIs which tend to be verbose) in order to avoid unnecessary overhead.

In the CTC communication model, schema information is collected and made available by the *schema repository*. Nodes communicate with the *schema repository* in an initial dissemination phase, in which schema information is distributed and registered. Thus, a *schema repository* acts as a centralized resource information base (where the resources are schemas) and provides application agnostic mechanisms to register, store, request, update and, in summary, manage data model schemas.

When the gateway translates application data to/from CTC, the use of CTC is effectively hidden to the external nodes and, thus, the *schema repository* does not need to be externally accessible. On the other hand, when two nodes interchange data represented in CTC, they both need to have access to the same schema information stored in the same *schema repository*, regardless of the (sub-)network they are located in. This is a requirement in order to access consistent information and satisfy CTC's *schema context* dependencies during the bootstrapping phase.

Although it is not required in all cases, for convenience, the *schema repository* is depicted at the gateway itself in Figure 5.1. However, note that the gateway and the *schema repository* have two distinct functionalities: a gateway acts as a link between two networks, performing a data or protocol mapping/transformation if required, while a *schema repository* manages the data models' schema information that is needed by CTC. The *schema repository* is explained in more detail in the next section.

Nevertheless, CTC communication architecture allows the coexistence and interoperability between CTC enabled and not enabled devices. Devices with constrained resources will be able to take advantage of CTC while more powerful devices use the original data format at the same time.

5.2 Schema Repository

This section describes the role of the *schema repository*, the functionalities it provides and the procedure that must be followed to register, access, update and, in summary, to manage the schema information interchange process within a CTC enabled network. Although the *schema repository* approach was initially designed for CTC, it can be directly applied to other technologies for data compression based on schema information, such as EXI. Additionally, CTC uses the *schema repository* to explicitly manage schemas but the concept can be easily extended to more generic resources that need to be discovered at runtime and would benefit from a compact identifier assigned at runtime.

The main function of a *schema repository* is to provide a centralized location for storing information to access data model schemas (i.e., "links" to schema storing locations) as well as generate and assign compact identifiers, denoted *schemaId*s. Additionally, a *schema repository* may also be used as an intermediate schema storing place.

When a node joins the network for the first time, it can start a schema registration process with the *schema repository*. The purpose of registering a data model schema is to inform the *schema repository* of where the schema is located and how it can be retrieved. After this information is registered, a *schemaId* is assigned to the schema. Once the registration process is finished, other nodes are able to request schema information from the *schema repository*.

Upon the reception of a schema registration request, the *schema repository* first checks whether that particular schema has been already registered. In that case, the associated *schemaId* is returned to the node. If the schema is not registered yet, the *schema repository* generates a new *schemaId*.

When a schema is registered, an associated globally unique identifier is provided in order to unambiguously identify the schema. Additional parameters are also included in order to give relevant information needed to access the schema, such as the URL where the schema is located or the data format. This set of parameters, which conform all the information required to locate and access the schema, is denoted a *schema link register*.

Initially, schemas may be stored at the node itself or at another convenient location such as an external server. Note that resource-constrained nodes only need to store the schemas of the data models they actually use. Moreover, if the schemas are stored in an external server and are accessible by the interested (client) nodes, they can be totally stripped from the source node.

A *schema repository* may download the schemas directly from a node or from an external server as depicted in Figure 5.2, a) and b) respectively. Optionally, the *schema repository* may also became an intermediate container of the downloaded schemas. In this case, schemas will be also accessible and downloadable directly from the *schema repository*. Storing the schemas directly in the *schema repository* gives some advantages. On one hand, nodes are relieved from the burden of serving the schemas each time a schema is requested, reducing energy consumption. On the other hand, the traffic of the constrained network is reduced in case the schema is requested by an external node. Additionally, problems related to sleeping nodes (such as non-availability) are avoided.

Once the registration process is finished and the schemas are downloaded and stored in the *schema repository*, they are processed in order to generate the *context table* and *template table*. As an efficiency improvement, the *schema repository* could also pre-load a set of standard schemas or download already pre-compiled *context table*s and *template table*s.

CTC registration process aims to be generic and, consequently, does not assume any underlying protocol. The next section provides a generic definition of the structure of a *schema link register* and Section 5.2.2 describes the abstract methods used in a schema registration process. An illustrative example can be found in Section 5.3. The example

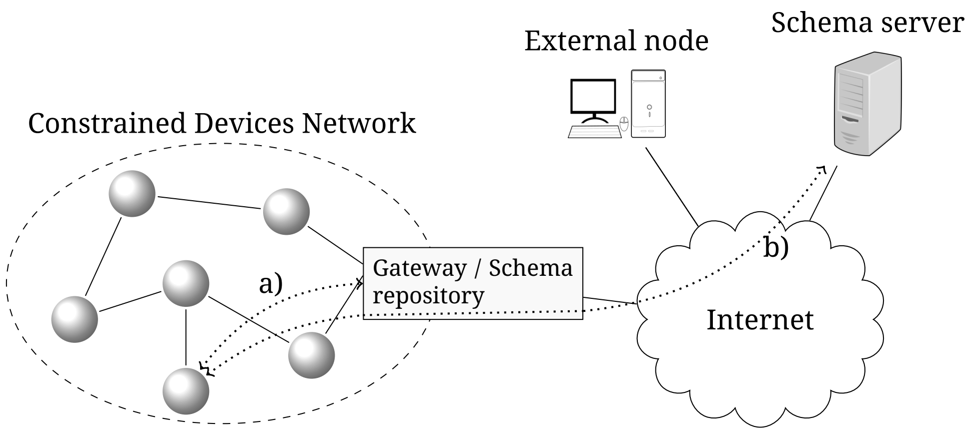


Figure 5.2: Template location. a) at the Node, b) at an external Server.

shows a binding of the registration process to CoAP [SHB14]. This binding specifies a registration API using CoAP methods to gather/register the schemas using CoRE Link Format [She12], together with an extension of CoAP *resource directory* [SKB+18].

5.2.1 Schema Link Register Structure

A *schema link register* is used to represent all the information needed to access, download and process a datamodel schema.

A *schema link register* entry must contain, at least, the following information:

- `unique identifier`: it must identify the schema in a globally unique way so it usually takes the form of an URI. For instance, in the case of XML or JSON schemas, the associated namespace could be directly used as unique identifier under the assumption that they unambiguously identify the schema.
- `schemaId`: the locally unique (compact) identifier assigned by the *schema repository* to the schema.
- `lifetime`: the registration of a schema must be refreshed periodically or the schema registration will be removed from the *schema repository*. The lifetime attribute specifies the period of time a *schema link register* is considered valid before a refresh is needed.
- `location`: this attribute contains the location where the schema can be accessed. It will usually take the form of an URL.
- `size`: the size of the schema. This attribute is necessary to manage the schema download as well as arrange the necessary resources.

- `hash`: the computed hash of the schema document, such as a md5hash, in order to verify that the version of the schema is the proper one as well as to detect corrupted schemas.
- `format`: the data representation format used by the schema, e.g., XML, JSON, EXI and CTC.

Depending on the specific case, additional information may be needed. For instance, security configuration options may be specified in order to overcome security measures set by a server.

More than one *schema link register* may share the same `unique identifier` and `schemaId`. These different instances of the same schema may be used to declare multiple optional locations (for example, in the node and in mirror servers) or schemas stored in different formats.

5.2.2 Schema Registration Management Abstract Methods

This section describes the abstract methods used to manage the schema registration process of a *schema repository*. These methods are generic enough so they can be mapped to a variety of protocols and architecture approaches.

The management of the schema registration process can be logically divided into five abstract methods: REGISTER, ASSIGN, LOOKUP, DELETE and DOWNLOAD.

- **REGISTER**: this method is used to register a schema into a *schema repository*. The REGISTER method must include the `unique identifier` parameter as well as the location from where the schema is accessible. The `lifetime` parameter may be included but it is not mandatory if a default value is defined for all the stored *schema link registers*. The `format` parameter is mandatory if it cannot be inferred by other means. Finally, `size` and `hash` parameters should be included in order to verify the schema.
- **ASSIGN**: a *schema repository* uses this method to assign a `schemaId` to a schema. The ASSIGN method is the response to a REGISTER method.

  The ASSIGN method includes the parameters `unique identifier` and `schemaId`, and can optionally specify a `lifetime` parameter in case the value used in the REGISTER method has not been accepted (for instance, if the *schema repository* demands shorter refresh cycles than the one requested by the node).
- **LOOKUP**: a node uses the LOOKUP method to request a *schema link register*. The LOOKUP method has a single mandatory parameter, the `unique identifier`. A LOOKUP method may be used to retrieve information to access the schema or just

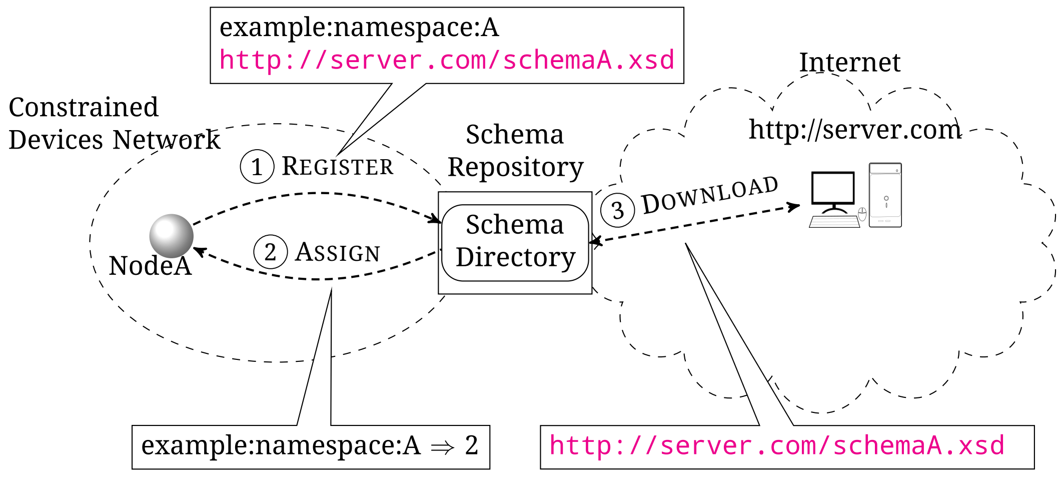


Figure 5.3: CTC abstract methods REGISTER, ASSIGN and DOWNLOAD.

to get the `schemaId`. Filtering functionalities may be implemented by the *schema repository* in order to retrieve multiple *schema link registers* that meet the same criteria.

- **DELETE**: this method is used by a node to explicitly erase a *schema link register* entry.

  The DELETE method has two mandatory parameters, the `unique identifier` and `schemaId`. The *schema link register* will only be deleted if both parameters are consistent with the information stored in the *schema repository*. Note that only the *schema link register* will be deleted, the `schemaId` will not be necessarily freed. The `schemaId` will be freed when all the *schema link registers* of the associated `unique identifier` are deleted from the *schema repository*.

- **DOWNLOAD**: this method is used by a node or gateway to download a schema based on the information stored in a specific *schema link register*.

Figures 5.3 and 5.4 present an example that summarizes the different processes of the registration management methods during the life-cycle of a schema. More precisely, Figure 5.3 shows the REGISTER, ASSIGN and DOWNLOAD methods. First, *NodeA* sends a REGISTER method to the *schema repository* (step 1). In this step *NodeA* specifies the `unique identifier` (“`example:namespace:A`”) and `location` (“`http://server.com/schemaA.xsd`”) parameters. In this case, the `format` parameter is not included since it can be inferred from the schema document file extension (“.xsd”). Then, the *schema repository* answers with an ASSIGN method (step 2) and assigns the `schemaId` ‘2’ to the data model schema. Additionally, the *schema repository* downloads the schema (step 3) and stores it locally.

Figure 5.4 shows the LOOKUP and DELETE methods. *NodeB* sends a LOOKUP method (step 4) and the *schema repository* answers sending two *schema link registers*: one rep-

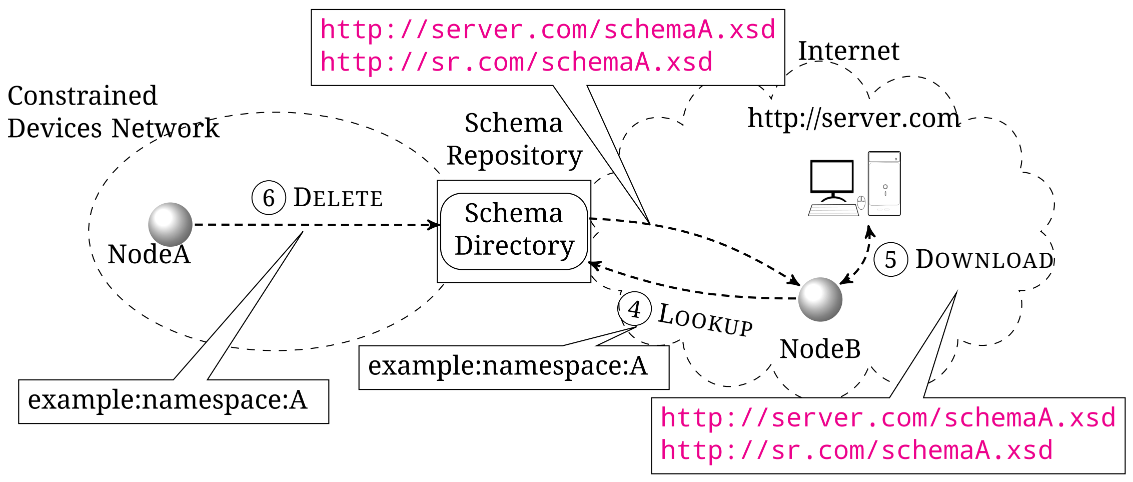

http://server.com/schemaA.xsd
http://sr.com/schemaA.xsd
Internet
Constrained Devices Network
Schema Repository
http://server.com
6 DELETE
Schema Directory
NodeA
5 DOWNLOAD
4 LOOKUP
NodeB
example:namespace:A
example:namespace:A
http://server.com/schemaA.xsd
http://sr.com/schemaA.xsd

Figure 5.4: CTC abstract methods LOOKUP and DELETE.

resenting the original *schema link register* registered by *NodeA* (“`http://server.com/schemaA.xsd`”) and the second one created by the *schema repository* itself, containing the link (“`http://sr.com/schemaA.xsd`”) to the schema locally stored in the *schema repository*. The *NodeB* decides to download the schema from the *schema repository* (step 5) so it uses the information in the second *schema link register* to execute a DOWNLOAD method. After downloading the schema, *NodeB* processes it and generates the *context table* and *template table* in order to use CTC in the forthcoming communications. Finally, *NodeA* deletes the *schema link register* (step 6) and the *schema repository* releases the `schemaId`, also deleting the additionally created *schema link register* for the locally stored schema copy.

5.3 CoAP Binding

As has been explained in previous sections, the purpose of the *schema repository* is to serve as a repository of *schema link registers* and schemas as well as to manage the generation and assignment of *schemaIds*. This section describes how the functionalities of a *schema repository* are implemented using CoAP as the underlying protocol.

5.3.1 Schema Directory

The *schema repository* implementation based on CoAP is built on top of the concept of a *schema directory*. In turn, the concept of a *schema directory* is largely based on the *resource directory* [SKB+18] specification for CoAP.

A CoAP *resource directory* is used to store information about generic web resources. Additionally, a *resource directory* provides a REST interface designed for the registration and lookup of stored resources. A *schema directory*, in contrast, is specifically designed

```
Req: GET coap://[ff02::1]/.well−known/core?rt=core.sd*

Res: 2.05 Content
</sd>;rt="core.sd";ct=40,
</sd−sch>;rt="core.sd−lookup−sch";ct=40,
</sd−sid>;rt="core.sd−lookup−sid";ct=40
```

Figure 5.5: Example discovery on a *schema directory* example.

for one resource type (data model schemas) and implements a dedicated CoAP interface in order to provide the methods and functionalities for a *schema repository*.

A *schema directory* uses the same discovery approach as a *resource directory*, i.e., a GET request is performed on a well-known path ("/.well-known/core") on the server. For the *schema directory* case, the server is always a *schema repository*.

On a *resource directory*, the GET request indicates the "Resource Type" query parameter. The specified parameter values are "core.rd", "core.rd-lookup-res", "core.rd-lookup-ep","core.rd-lookup-gp" or "core.rd-group". These parameters are used for querying the location of the different interfaces for registration, lookup and groups. However, a *schema directory* uses a different set of "Resource Type" values than the ones defined by the CoAP *resource directory*. The *schema directory* registration path is requested by specifying a "Resource Type" parameter with a value of "core.sd" in the query string. In order to request the lookup paths for *schema link register* entries, a request must be performed with the value "core.sd-lookup-sch". Finally, a "Resource Type" value of "core.sd-lookup-sid" is used to query the *schemaId*s lookup paths. These new "Resource Type" parameter values are not IANA (Internet Assigned Numbers Authority) official registered identifiers and they are defined here in order to be able to differentiate them from the CoAP *resource directory*'s counterparts.

In the same way as in a *resource directory*, a successful request will return a response containing a list of link entries (following the CoRE Link Format) that satisfy the request's parameters.

The following example (Figure 5.5) shows the CoAP request issued by a node performing a discovery operation as well as the response sent by a *schema repository*. The request queries all the interfaces of a *schema directory*, i.e., "Resource Type" parameters that start with the value "core.sd". In the example, the schema registration interface path is "/sd", the schema lookup interface path is "/sd-sch" and the *schemaId* lookup interface path is "/sd-sid". Additionally, the content-format for all the interfaces is "application/link-format" (ct=40). Note that the paths are application dependant and the ones shown in Figure 5.5 are just examples.

5.3.2 Schema Directory Registration Interface

In order to perform a REGISTER request to register a data model schema (or a list of data model schemas), a node issues a POST method to the registration interface. This POST method must contain the list of *schema link register* entries of the data model schema to be registered.

Each *schema link register* must include the globally unique identifier required by the *schema repository* and, optionally, an anchor attribute (as explained in the CoAP Resource Directory specification [SKB+18], section "5.3.Registration") in order to indicate schemas stored outside the origin node. The `uid` attribute is used to specify the globally unique identifier of a *schema link register*. Although it is not mandatory, it is recommended that the `uid` attribute follows the URI format. There is no upper bound to the length of the `uid` attribute. Although it would be recommended to set a maximum length limit, unique identifiers will be probably mapped to namespaces used in the schemas (XML and JSON namespaces), which are known to be verbose.

On a successful registration, the *schema repository* creates a new registration resource in the *schema directory* for each unique identifier not already registered. Therefore, for each new data model schema (i.e., `uid`s not already registered in the *schema directory*) contained in the *schema link register* list, a new resource is created and a new *schemaId* is generated and assigned to the data model schema. In case a data model schema with the same unique identifier has already been registered, the *schema repository* reuses the already assigned *schemaId*. Due to security reasons, some checking may be necessary in order to ensure that the unique identifier and links really identify and point to the same data model schemas.

Once the *schema repository* finishes the internal registration process (i.e., resource creation and *schemaId* assignment), it returns the location of the registration resource to the origin node. The location is included in the "Location" header of the response sent by the *schema repository*. As in the *resource directory*, a node that just registered its data model schemas, should remember the location of the registration resource. This location is used in management operations such as the refreshing of the registration lifetime as well as inspect, update or remove the resource.

Schema registrations must be refreshed within a given period. This period can be specified in the registration request using the `lifetime` parameter. If a `lifetime` parameter is not specified, a default value is assigned. Lifetime values are assigned to the whole registration resource. If the lifetime of a registration resource expires, all the links registered by the origin node are removed from the *schema directory*. If all the links of a given unique identifier (and hence, of a data model schema) are removed, the schema registration resource is deleted and the *schemaId* is freed.

The registration request may also include a `context` parameter that will be applied to all the *schema link register*s contained in the request. However, if a *schema link register* already has an `anchor` attribute, the `context` parameter is ignored. The purpose of the `context` parameter is to reduce the size of each *schema link register*. If the `context` parameter is not included, it is assumed that the origin node is the context. Nevertheless, links that upon registration did not contain an `anchor` attribute, which indicates that the data model schema is stored outside the origin node, are assigned an anchor equal to the context URI of the registration.

The registration request interface of the *schema directory* is an adaptation of the registration interface specified for the *resource directory* (specified in [SKB+18], Section 5.3).

Interaction: CoAP Endpoint -> Schema Directory
Method: POST
URI Template: +sd?lt,con

The parameters `lt` and `con` are specified as defined in [SKB+18], i.e, they respectively indicate the lifetime of the registration in seconds and the default base URI for the request' *schema link register* entries.

Following with the example shown in Figure 5.5, Figure 5.6 shows an example of a request sent and response received by a node registering two data model schemas using the registration interface path "/sd". The node attempts to register two data model schemas with unique identifier values of "`example:uri:schema1`" and "`example:uri:schema2`". For the data model schema identified as "`example:uri:schema1`", two links are provided, one located in the node itself (on the path "/sch/schema1") and the other in an external server in the URL "`http://example.com//schema1/schema`". For the data model schema identified as "`example:uri:schema2`" one single link located at the node in the path "/sch/schema2" is provided. The content format for all the links is "application/xml" ('41' in CoAP identifier code format). The `size` and `hash` parameters are provided in order to verify the schema document.

Upon registration, the *schema repository* will, optionally, download the schema, store it locally and make it available through a CoAP resource interface. In this case, the *schema repository* adds the link to the *schema directory* so it is available on the lookups interfaces.

5.3.3 Schema Directory Lookup Interfaces

The *schema directory* provides lookup interfaces in a similar way as a *resource directory*. These interfaces are necessary in order to discover and retrieve the links to the data model

```
Req: POST coap://sd.example.com/sd
Content−Format: 40
Payload:
</sch/schema1>;ct=41;rt="schema";uid=example:uri:schema1;sz=512;hash="0123456789↩
    abcdef0123456789abcdef",
</schema1/schema>;ct=41;rt="schema";uid=example:uri:schema1;anchor="http://example.↩
    com";sz=512;hash="0123456789abcdef0123456789abcdef",
</sch/schema2>;ct=41;rt="schema";uid=example:uri:schema2;sz=322;hash="↩
    abcdef0123456789abcdef0123456789"

Res: 2.01 Created
Location: /sd/1245
```

Figure 5.6: Example registration request on a *schema directory*.

schema locations and to the *schemaIds* assigned to the schemas. The *schema directory* provides two types of lookup interfaces, the *schema lookup* interface and *schemaId lookup* interface. The *schema lookup* interface is used to retrieve the list of *schema link registers* to data model schema locations while the *schemaId lookup* interface is used to get the *schemaIds* assigned to the data model schemas.

A request to the *schema lookup* interface will return the *schema link register* to the location where data model schemas are stored and from where they can be accessed and downloaded. Note that the *schema link registers* of the schemas locally stored in the *schema repository* will be also included.

The resource type for the *schema lookup* interface is "core.sd-lookup-sch" and is specified as follows:

Interaction: CoAP Endpoint -> Schema Directory
Method: GET
URI Template: +lookup-location?uid

The `lookup-location` parameter is the lookup URI of the *schema lookup* interface as returned by the well-known path. The result is the list of *schema link registers* to the data model schema identified by the `uid` parameter. It is convenient to apply filters (by specifying relevant parameters, in this case `uid`) to schema lookup requests. This avoids the need to explicitly search the required resource in the (potentially large) returned resource list.

In the example shown in Figure 5.7, a node issues a schema lookup request of the data model with the unique identifier "`example:uri:schema1`". The *schema directory* returns three *schema link registers*, the first one pointing to the schema stored in the origin node ("`coap://[2001:db8:3::123]:61616/sch/schema1`"), the second to the schema stored in an external server ("`http://example.com/schema1/schema`") and

```
Req: GET coap://sd.example.com/sd-sch?uid=example:uri:schema1

Res: 2.05 Content
</sch/schema1>;ct=41;rt="schema";uid=example:uri:schema1;anchor="coap://[2001:db8:3::123↩
    ]:61616";sz=512;hash="0123456789abcdef0123456789abcdef",
</schema1/schema>;ct=41;rt="schema";uid=example:uri:schema1;anchor="http://example.↩
    com";sz=512;hash="0123456789abcdef0123456789abcdef",
</sd/sch/1>;ct=41;rt="schema";uid=example:uri:schema1;sz=512;hash="0123456789↩
    abcdef0123456789abcdef"
```

Figure 5.7: Example schema lookup request on a *schema directory*.

the last one pointing to the schema stored in the *schema repository* itself ("`coap://sd.example.com/sd/sch/1`").

A request to the *schemaId lookup* interface will return the list of *schemaId*s assigned to the queried unique identifiers. The resource type for the *schemaId lookup* interface is "core.sd-lookup-sid" and it is specified as follows:

Interaction: CoAP Endpoint -> Schema Directory
Method: GET
URI Template: +lookup-location?uid,con

The `lookup-location` parameter is the lookup URI of the *schemaId lookup* interface as returned by the well-known path.

The result is the *schemaId* resource assigned to the registered data model schema identified by the `uid` and/or registered by the origin node identified with the context URI `con`. Many unique identifiers can be included in the query parameter `uid` separated with a space. In this case, the result may be a list of *schemaId* resources.

Figure 5.8 shows an example of a node requesting a *schemaId* lookup on a *schema directory*. In this example, the node only specifies one unique identifier ("`example:uri:schema1`") and, consequently, the *schema repository* only returns one resource with the assigned *schemaId* of "1". In contrast, in the example depicted in Figure 5.9 the node requests a *schemaId* lookup for data model schemas registered by a particular node, in this case "`coap://[2001:db8:3::123]:61616`". In the example, the queried node had registered two data model schemas so the *schema repository* returns two resources, "/sd/sid/1" and "/sd/sid/2" with assigned *schemaId*s of "1" and "2" respectively.

schemaId resources can also be queried in order to get the list of registration resources associated to a specific data model schema. Although this functionality is not expected to be specially useful for applications, it may be needed by management applications.

For instance, Figure 5.10 shows an example of a node querying a *schemaId* resource which, in this case, is linked to two distinct registration resources (produced by two nodes registering the same data model schema).

```
Req: GET coap://sd.example.com/sd-sid?uid=example:uri:schema1

Res: 2.05 Content
</sd/sid/1>;uid="example:uri:schema1";sid=1
```

Figure 5.8: Example *schemaId* lookup request on a *schema directory* with a single “uid” value.

```
Req: GET coap://sd.example.com/sd-sid?con="coap://[2001:db8:3::123]:61616"

Res: 2.05 Content
</sd/sid/1>;uid="example:uri:schema1";sid=1
</sd/sid/2>;uid="example:uri:schema2";sid=2
```

Figure 5.9: Example *schemaId* lookup request on a *schema directory* with a context parameter.

```
Req: GET coap://sd.example.com/sd/sid/1

Res: 2.05 Content

</sd/1245>;con="coap://[2001:db8:3::123]:61616"
</sd/4567>;con="coap://[2001:db8:3::124]:61616"
```

Figure 5.10: Example *schemaId* resource query on a *schema directory*.

```
Req: DELETE /sd/1245

Res: 2.02 Deleted
```

Figure 5.11: Example *schemaId* resource deletion on a *schema directory*.

5.3.4 Schema Directory Registration Deletion

Like CoAP *resource directory* entries, *schema directory* registrations have soft state and will be erased if they are not refreshed within the lifetime specified in the registration (see Section 5.3.2). Nevertheless, the schema directory provides a registration removal interface in order for the origin node to be able to explicitly remove its registration.

The registration removal interface is specified as follows:

```
Interaction: CoAP Endpoint -> Schema Directory
Method: DELETE
URI Template: +location
```

The `location` parameter is the registration resource path returned in the “Location” header of the response to the successful registration request.

If the *schema directory* receives a DELETE request, the registration resource is treated in the same way as if the lifetime expires: all the *schema link register*s contained in the registration resource are removed from the *schema directory*. If all the links of a given unique identifier (and hence, of a data model schema) are removed, the schema registration resource is deleted and the *schemaId* is freed.

Figure 5.11 shows an example of a node deleting the registration resource created on the registration example shown in Figure 5.6.

5.3.5 CTC Link Format

Apart from the formats supported by the CoAP *resource directory*, CoRE Link Format [She12] (“application/link-format”), JSON CoRE Link Format (“application/link-format+json”), and CBOR CoRE Link Format [LRB17] (“application/link-format+cbor”), a *schema directory* also accepts the CTC Link Format (“application/link-format+ctc”).

The CTC Link Format is a complementary and convenient way of formatting link contents in a compressed link format. In order to be able to make use of the CTC Link Format, a globally unique identifier must be assigned to the CTC Link Format schema and the *schema register* must register it into the *schema directory* before any other registration process takes place. By registering the CTC Link Format schema, the *schema directory* is pre-populated with the *schema link registers* and a *schemaId* is assigned to the schema.

The data model schema for the CTC Link Format is specified based on the JSON Schema for the Content Format "application/link-format+json" and defined in [LRB17]. This schema is extended with the attributes `uid` and `sid` in order to support the interfaces of a *schema directory*. The schema is shown in Figure 5.12.

5.4 Summary and Conclusions

The IoT relies on the deployment of interconnected heterogeneous devices and systems. This demands interoperable communications and data formats which are typically addressed through the adoption of standard text-based data formats. However, the verbosity of these text-based data formats demands processing and communication resources that might be beyond the capabilities of the resource-constrained devices and networks typically used in IoT networks.

A common technique used to handle structured data more efficiently is to compress the data itself using a more efficient encoding. Compression technologies targeted to structured data take advantage of the contextual information extracted from the data format and model (formal structure, grammar, schema, etc.) and use this information to efficiently compress data.

In this section we have presented a generic communication model for the efficient management of the contextual information required by compression techniques for structured data. The section describes the *schema registration* and *schema repository* approaches in a generic, independent way from any underlying communication protocol. The *schema registration* and *schema repository* mechanisms allow to dynamically assign and distribute schema information at run time. These mechanisms are very flexible and have been designed taking into account the multiple restrictions of resource-constrained devices.

The proposed solution provides a flexible and interoperable communication architecture. Devices using standard data representation formats can coexist and communicate with devices using compressed formats. Devices can be located within the same (sub-)network or reside in separated networks. A CTC gateway can be deployed to behave as an application level gateway and seamlessly translate between the compressed format and the original data format.

We used CTC as the compression technology for the *schema registration* and *schema repository* approaches. In Chapter 4 we showed that CTC is specially targeted to resource-constrained devices and it is able to provide a generic transformation from one format to other in a seamless, interoperable and efficient way. Together with the communication

```
{
  "$schema": "http://json−schema.org/draft−06/schema#",
  "type": "object",
  "required": [
    "href"
  ],
  "properties": {
    "href": {
      "type": "string"
    },
    "rel": {
      "type": "string"
    },
    "anchor": {
      "type": "string"
    },
    "rev": {
      "type": "string"
    },
    "hreflang": {
      "type": "string"
    },
    "media": {
      "type": "string"
    },
    "title": {
      "type": "string"
    },
    "type": {
      "type": "string"
    },
    "rt": {
      "type": "string"
    },
    "if": {
      "type": "string"
    },
    "sz": {
      "type": "integer"
    },
    "ct": {
      "type": "integer"
    },
    "obs": {
      "type": "boolean"
    },
    "con": {
      "type": "string"
    },
    "uid": {
      "type": "string"
    },
    "sid": {
      "type": "integer"
    }
  },
  "additionalItems": false
}
```

Figure 5.12: JSON schema for “application/link-format+ctc”.

model presented in this chapter, CTC proves to be a very good candidate for generic data model representation and compression in resource-constrained devices and networks. Nevertheless, the CTC Communication Model is generic enough to be applied to other structured data compression technologies based on contextual information, e.g., EXI.

As will be shown later in Section 7.3, CTC and CTC Communication Model have a positive impact on the reduction of transmitted messages. Different configurations of the proposed solution can fit different needs and scenarios depending on the available resources and schemas used. For instance, the flexibility in the location of schemas' storing place notably reduces the message transmission overhead and removes the need to store the schemas on the devices themselves, saving memory resources.

6 CTC Library

The functionalities provided by CTC are encapsulated in a software library, denoted as *CTC Library*. These functionalities include the management of the *context table* and *template table*, the execution of the *CTC Codification Algorithm* and the mechanisms of the CTC Communication Model. This library is embedded and used by resource-constrained devices, external clients, CTC Gateway and, in summary, any entity that requires access the functionalities offered by CTC and to encode/decode data streams. In this chapter we will use the term "device" to refer to any HW/SW entity that makes use of the *CTC Library*.

The *CTC Library* follows a modular approach in order to tailor its capabilities to the needs of the application and resources of the device. A support tool for the *CTC Library*, called *CTC Compiler,* is used to process the data model schemas and automatically create the *context table* and *template table* as well as the necessary native code for the data model bindings. This code is embedded in the devices's application code at programming time and is referenced by the *CTC Library* in order to provide the CTC capabilities.

6.1 Architecture and Components

The architecture of the *CTC Library* is designed to be modular. In this way unneeded components can be removed at compile time and effectively reduce code size. The generic architecture of the *CTC Library* is shown in Figure 6.1.

The core of the *CTC Library* is composed by the *Codifier* and *Manager* components. The *Codifier* component implements the *CTC Compression Algorithm* and is in charge of decoding/codifying the data streams while the *Manager* component is the responsible of the management of the *context table*, including external accesses. On the other hand, the *Compiler* component is used to process data model schemas at runtime and create the corresponding *schema context* and templates. The *CTC Library* also provides a *Generic Binder* component which is used to either decode the coded data stream and rebuild the data in their original data format or to parse data in the original data format and transform they to a CTC coded data stream.

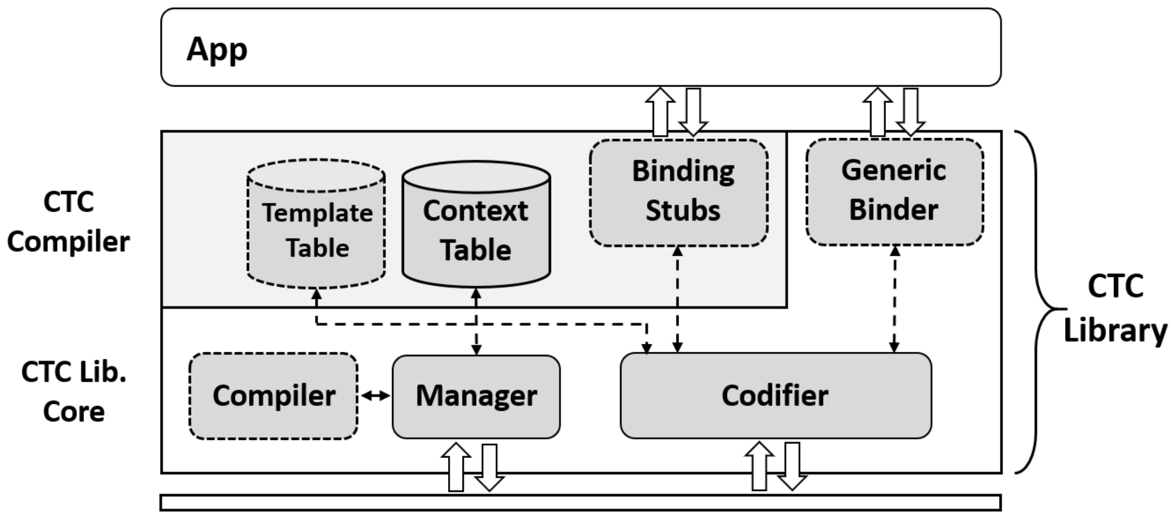


Figure 6.1: Architecture of the *CTC Library*. Components in the white area are provided by the *CTC Library* while components in the grey area are created by the *CTC Compiler*. Dotted components are optional.

The *Context Table* and *Template Table* components represent the implementation of the *context table* and *Template Table* concepts described in Chapter 4. Thus, the *Context Table* component holds all the *schema context*s used by the device, whereas the *Template Table* component contains the schema templates.

Finally, the *Binding Stubs* component groups all the data model binding code used to directly map the data stream decoded by the *Codifier* component into native structures and to transform native structures into coded data streams through the *Codifier* component. In other words, *Binding Stubs* component acts as the interface between the application's native structures and the CTC implementation.

The *Context Table*, *Template Table* and *Binding Stubs* components (depicted in the grey area in Figure 6.1) are not provided by default by the *CTC Library*. These components are created using the *CTC compiler*.

Each component of the *CTC Library* architecture is described in more detail in the following list.

- **Codifier**: the *Codifier* component implements the *CTC Compression Algorithm* and is the decoding/codifying engine of the *CTC Library*. The *Codifier* component provides mechanisms for the *Binding Stubs* and *Generic Binder* components. The *Binding Stubs* component uses these mechanisms to map native structures to CTC coded streams while the *Generic Binder* component does the same with data model instances in their original format. On the other hand, The *Codifier* component will use the *Context Table* and *Template Table* components, or more specifically, the information provided by these components, in order to perform the decoding and codifying processes.

The *Codifier* component is mandatory and is always included as part of the *CTC Library*.

- **Manager**: the *Manager* component is the responsible of the management of the external accesses (either write or read) to the *context table*. These accesses are usually produced by interactions with other nodes or the CTC *schema repository* as part of the schema registration process. The *Manager* makes use of the *Compiler* component to process schemas and create the *schema context*s and templates for the *Context Table* and *Template Table* components.

  The *Manager* component is considered as a mandatory component and is practically always included as part of the *CTC Library*. However, as a special case and if no schema registration is needed (for instance, if an application uses statically assigned *schemaId*s), this component could be removed.

- **Generic Binder**: the *Generic Binder* component is used to rebuild the data coded in a data stream to its original format (by processing the data stream and merging the templates) and vice-versa. The *Generic Binder* uses the mechanisms provided by the *Codifier* component to recreate the data model instance to its original format, usually by simply merging the templates as the data stream is decoded. This component is also able to codify data in their original data format to their CTC coded version by making use of the *Codifier* and the information in the *Context Table* and *Template Table* components.

  The *Generic Binder* is an optional component and is only used by applications that need to process data in their original data format. This component is provided by the *CTC Library*.

- **Compiler**: the *Compiler* component is the embedded implementation of the *CTC Compiler* tool. It is used by devices to parse schemas and create *schema context*s and templates to populate the *context table* and *template table*. The *Compiler* component is used by devices that need to populate the *context table* and *template table* at runtime, for instance, CTC gateways that need to include a new registered schema.

  This component should not be confused with the *CTC Compiler*. The *CTC Compiler* is a standalone tool used at code compile time (as it will be described in the following section) to generate the *Context Table*, *Template Table* and *Binding Stubs* components. The *Compiler* component, on the other hand, is part of the *CTC Library* and used at runtime to feed newly created *schema context*s and templates to the *context table* and *template table*.

  The *Compiler* is provided by the *CTC Library* but it is an optional component as it is only needed by devices that need to update the *context table* and *template table* at runtime.

- **Context Table**: the *Context Table* component holds all the *schema context*s of the application. This component also contains the schemas of the *schema context*s. The schemas are included so they can be retrieved by the *Manager* component. However, the schemas can be stripped if another suitable storing place (such as an external server) is provided.

  The *Context Table* is a mandatory component but the inclusion of the schemas in the *Context Table* is optional. The *Context Table* component is created by the *CTC Compiler*.

- **Template Table**: the *Template Table* component contains the templates of all the *schema context*s stored in the *Context Table* component. These templates are mainly used by the *Generic Binder* to transform data coded in CTC to their original data format and vice-versa.

  The *Template Table* is an optional component created by the *CTC Compiler*. It is only needed if the *Generic Binder* component is also included as part of the *CTC Library*.

- **Binding Stubs**: the *Binding Stubs* component groups all the data model binding code automatically generated by the *CTC compiler*. These code stubs are used to directly map the data stream decoded by the *Codifier* component into native structures and to transform native structures into coded streams through the *Codifier* component. This results in a much more efficient processing of data because the parsing of the original data format is completely avoided. Note that a device that uses the *Binding Stubs* component does not need to include the *Template Table* and *Generic Serializer* components.

  The *Binding Stubs* component is an optional component created by the *CTC Compiler*. It is only needed if the application uses native structures to represent data and does not require an intermediate representation of the data in their original data format.

In practice, there are two basic *CTC Library* component combinations that will be used in real applications: a component combination tailored to resource-constrained devices and a full-featured *CTC Library* for applications that need to access data in their original data format.

A resource-constrained device will directly use native structures to represent data instead of using the original data format, which will be more verbose and more CPU consuming to process. Thus, a resource-constrained device will only include the *Codifier* and *Manager* core components. The *CTC Compiler* will be used to create the *Context Table* and *Binding Stubs* components from the schemas of the data models used by the device's application. Note that the *Template Table* is not included while the *Binding Stubs* component is used to efficiently process data as native structures. Additionally, if the schemas of the data models are reachable from an external source (such as a dedicated

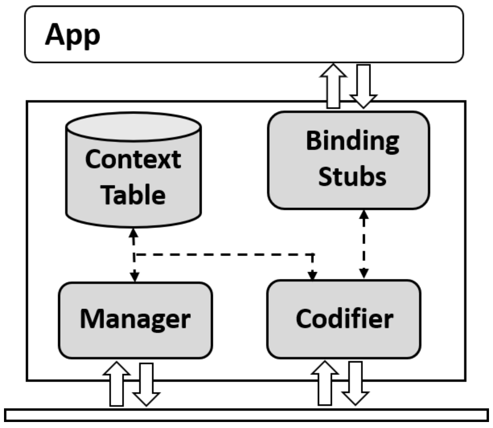


Figure 6.2: *CTC Library* configuration for resource-constrained devices.

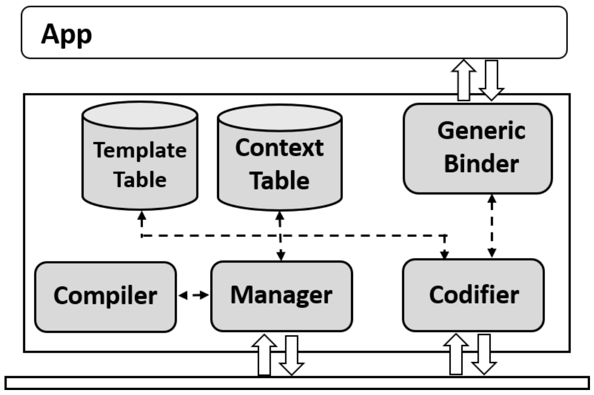


Figure 6.3: Full-featured *CTC Library* configuration.

server), they can be stripped from the *Context Table* component. This combination of components is depicted in Figure 6.2.

On the other hand, devices that need to act as translators or, in any case, translate from the CTC format to the original data format (and vice-versa), will use a full-featured *CTC Library*. Examples of devices that will use a full-featured *CTC Library* include CTC gateways or external applications that communicate with a CTC enabled network but nevertheless need to process data in their original data format (for instance, for compatibility or legacy reasons). In this case, the *CTC Library* will include all the core components (*Codifier*, *Manager*, *Compiler* and *Generic Binder*) as well as the *Context Table* and *Template Table* components. Usually, the *CTC Compiler* is not used at compile time as the schemas used by the devices may not be known in advance. Thus, *Context Table* and *Template Table* components may be initially empty and they will be populated as the schemas are discovered and registered. This component combination is shown in Figure 6.3.

6.2 CTC Compiler

The *CTC Compiler* is a complementary tool for the *CTC Library*. This compiler is used to process data model schemas and create the corresponding *schema contexts* and templates

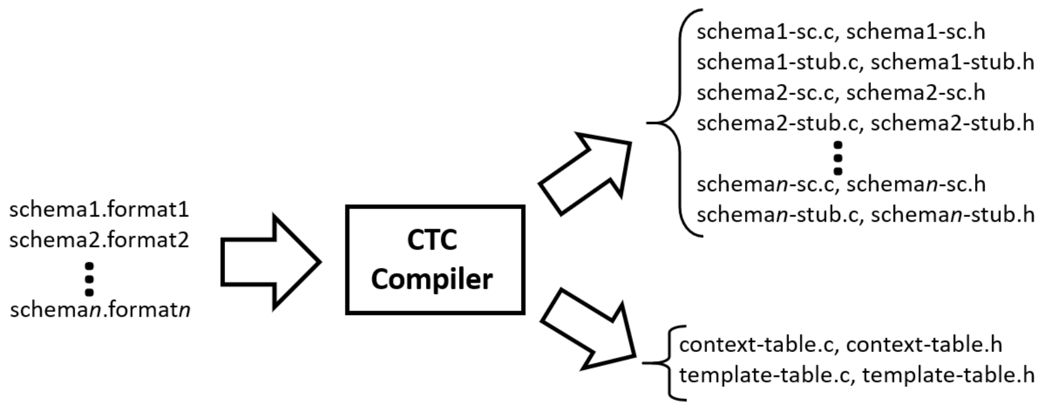

schema1.format1
schema2.format2
scheman.formatn
CTC Compiler
schema1-sc.c, schema1-sc.h
schema1-stub.c, schema1-stub.h
schema2-sc.c, schema2-sc.h
schema2-stub.c, schema2-stub.h
scheman-sc.c, scheman-sc.h
scheman-stub.c, scheman-stub.h
context-table.c, context-table.h
template-table.c, template-table.h

Figure 6.4: Inputs and outputs of the *CTC Compiler*.

to be included in the *Context Table* and *Template Table* components. Additionally, the *CTC Compiler* also generates the necessary C code for the data model bindings that is included as part of the *Binding Stubs* component of the *CTC Library*. In summary, the *CTC Compiler* generates all the data model(s) specific code needed by the *CTC Library* components.

Figure 6.4 shows a simplified model of the *CTC Compiler*'s inputs and outputs. All the schemas that are to be supported by the application have to be included as inputs for the *CTC Compiler*.

For each schema, two pairs of code and header files will be created. One of these file pairs corresponds to the *schema context* of the schema (depicted as "schema#-sc.{c,h}" in Figure 6.4) and the other pair contains the code binding stubs (depicted as "schema#-stub.{c,h}" in Figure 6.4). The objective of the code binding stubs is twofold. In one hand, it provides a managing wrapper on top of the C structures representing the elements of the schema. On the other hand, it acts as an interface between the application and the *CTC Library*.

Additionally, another two pairs of code and header files will be created for the *Context Table* and *Template Table* components, respectively depicted as "context-table.{c,h}" and "template-table.{c,h}" in Figure 6.4. The *Context Table* files contain the references to the *schema context* files created from the input schemas, while the *Template Table* files include all the templates of all those *schema context*s.

All these files are included into the application and compiled together with the core implementation of the *CTC Library*. Currently, the *CTC Compiler* supports two schema types: XML Schemas and JSON Schemas.

The different steps followed by the *CTC Compiler* to process the schemas and create the native code is shown in Figure 6.5. The *CTC Compiler* follows four steps to create the native code from the schemas.

First, each schema is parsed separately and an internal representation of the data model structure and the relationships between the elements is created. Next, the internal

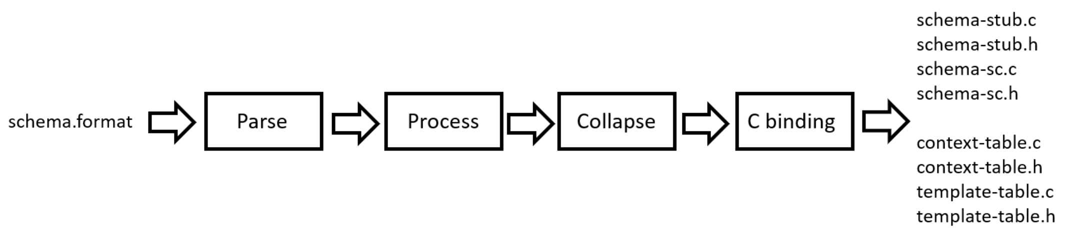


Figure 6.5: Internal process of the *CTC Compiler*.

representation is processed and a first version of the *Context Table* and *Template Table* components is created.

Then, the *Context Collapsing* process is applied to the previously created *Context Table* and *Template Table* components. As described in Section 4.1.3, this process removes unnecessary and redundant *eContext*s, *Element*s and templates, resulting in a more compact representation and faster processing. Once the definite version of the *Context Table* and *Template Table* components is created, the binding C code for the schema is generated. As explained before, two sets of files are created for every schema: one set contains the *schema context* code (depicted as “schema-sc.{c,h}” in Figure 6.5) and the other set (depicted as “schema-stub.{c,h}” in Figure 6.5) implements the code stubs to access and process the schema data model as native C structures.

Finally, once all the schemas are processed, the code for the *Context Table* and *Template Table* components is produced (depicted as “context-table.{c,h}” and “template-table.{c,h}” in Figure 6.5)

6.2.1 CTC Compiler Example

In this section we show an example for the *CTC Compiler*. The same *Notebook* schema example used in Section 4.1.4.1 will be used here. For convenience reasons, the *Notebook* XML Schema is shown again here in Figure 6.6.

The *schema context* and templates produced after parsing, processing and collapsing the schema are shown in Table 6.1 and Table 6.2, also copied from Section 4.1.4.1 for convenience reasons.

The *CTC Compiler* converts the *schema context* shown in Table 6.1 to C code and creates the C files “notebook-sc.c” and “notebook-sc.h”. In the same manner, the C code stubs to access the notebook data model *schema context* is stored in the files “notebook-stub.c” and “notebook-stub.h”.

Once the *Notebook* schema is processed and all the specific code files are created, the *CTC Compiler* creates the files for the *Context Table* component and includes the necessary references to the *Notebook* schema’s *schema context*. Finally, the *CTC Compiler*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                    elementFormDefault="qualified">
    <xs:element name="notebook">
        <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
                <xs:element name="note" type="Note"/>
            </xs:sequence>
            <xs:attribute ref="date"/>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="Note">
        <xs:sequence>
            <xs:element name="subject" type="xs:string"/>
            <xs:element name="body" type="xs:string"/>
        </xs:sequence>
        <xs:attribute ref="date" use="required"/>
        <xs:attribute name="category" type="xs:string"/>
    </xs:complexType>
    <xs:attribute name="date" type="xs:date"/>
</xs:schema>
```

Figure 6.6: *Notebook* XML Schema document.

Table 6.1: *Schema context* Notebook example, after *Context Collapsing*. The content of the *Children* row represents the tuple (*Template*, *Type*, *IsOptional*, *IsArray*, *Context*). Cn represents the *eContext Id* and tn the template identifier, x denotes *complex*, s *string*, c *constant*, d *date-time*, t TRUE and f FALSE.

Attribute	Id				
	C1 (ROOT)	C2 (CONTENT)	C3 (NOTEBOOK)	C4 (NOTE)	C5 (NOTE-ATT)
MultipleParents	f	f	f	f	f
Order	fixed	choice	fixed	fixed	dynamic
Children	(t1,c,t,f,-) (-,x,f,f,C2)	(t2,x,t,f,C3) (t3,d,t,f,-)	(t3,d,t,f,-) (t4,x,f,t,C4)	(-,x,f,f,C5) (-,s,f,f,-) (-,s,f,f,-)	(t3,d,f,f,-) (t5,s,t,f,-)

Table 6.2: *Notebook* XML Schema's templates, after *Context Collapsing*. Symbol '@' is used to represent the place-holders' positions

t1	<?xml version="1.0" encoding="UTF-8"?>
t2	<notebook @> @ </notebook>
t3	date=@
t4	<note category=@ @> <subject>@</subject> <body>@</body> </note>
t5	category=@

```
#include "notebook−sc.h"
#include "template−table.h"

const schema_context_s sc_ = {
  "sc_",      // uri
  6,          // element_number
  &ec__root  // element_contexts
};


static const element_s _children__root[] = {
  /*0*/ {24, &template_table_notebook[0], NULL},    //_prolog, "<?xml version=\"1.0\" encoding↩
     =\"UTF−8\"?>"
  /*1*/ {9, &template_table_notebook[1], &ec__body} //_body, "@"
};

const element_context_s ec__root = {
  content_fixed, 2, _children__root
};


static const element_s _children__body[] = {
  /*0*/ {89, &template_table_notebook[2], &ec_notebook}, //notebook, "<notebook@>@</↩
     notebook>"
  /*1*/ {86, &template_table_notebook[3], NULL}      //date, " date=\"@\""
};

const element_context_s ec__body = {
  content_choice, 2, _children__body
};
```

Figure 6.7: Code snippet of the “notebook-sc.c” file.

includes the templates of the *Notebook* schema into the *Template Table* and creates the files “template-table.c” and “template-table.h”.

A code fragment of each file is shown in the following figures. Figure 6.7 and Figure 6.8 respectively contain a code snippet of the *schema context* and *Template Table* source code files for the *Notebook* XML Schema example (i.e. “notebook-sc.c” and “template-table.c” files). Two different code fragments of the “notebook-stub.c” are shown in Figure 6.9 and Figure 6.10, one of the C structure bindings and the other of the C function wrappers.

6.3 Summary and Conclusions

In this section we described the *CTC Library* and its main components. The *CTC Library* provides an implementation of the CTC capabilities, including the management of the *context table* and *template table*, the execution of the *CTC Codification Algorithm* and the mechanisms of the CTC Communication Model. The *CTC Library* is embedded in applications in order to access the CTC capabilities as well as encode and decode data streams.

```
const template_s template_table_notebook[] = {
  /*0*/ {42, "<?xml version=\"1.0\" encoding=\"UTF−8\"?>"},
  /*1*/ {23, "<notebook@>@</notebook>"},
  /*2*/ {11, " date=\"@\""},
  /*3*/ {15, " category=\"@\""},
  /*4*/ {48, "<note@><subject>@</subject><body>@</body></note>"},
};
```

Figure 6.8: Code snippet of the “template-table.c” file.

```
struct Note {
  char *subject;
  char *body;
  char *date;
  char *category;
};

struct notebook {
  int __notebook_sequence_size;
  struct __notebook_sequence {
    struct Note *note;
  } *__notebook_sequence;
  char *date;
};
```

Figure 6.9: Code snippet of the C structure binding of the *Notebook* XML Schema document.

```
Note* Note_create(void);

int Note_destroy(Note* Note);

int Note_decode(const char* in_stream, int is_size, Note* Note);

int Note_encode(const Note* Note, char* out_stream, int os_size);
```

Figure 6.10: Code snippet of the basic C function wrappers for the *Note* structure binding of the *Notebook* XML Schema document.

The *CTC Library* provides a modular approach that allows to tailor the library capabilities to the needs and resources of the devices. This is specially important for resource-constrained devices in order to reduce the size of the library as much as possible (and make the most of the available memory) while implementing all the required functionalities. As will be shown later in Section 7.1, the *CTC Library* has a very small memory footprint and provides an efficient CTC implementation suitable for the most resource-constrained devices.

The *CTC Library* is complemented by the *CTC Compiler* tool, easing and automating the implementation of the *context table*, *template table* and data model bindings to native code.

7 Evaluation

This chapter contains all the empirical tests performed during the development of the work presented in this document. The purpose of these tests is to evaluate and verify the performed work in terms of compression size, processing time and memory usage, as well as the sustained hypothesis.

Section 7.1 contains the performance evaluation of a prototype implementation of CTC, using XML as the specific data format. This section also includes a comparison of CTC to the leading XML compressor: EXI. Section 7.2 extends the compression efficiency evaluation of CTC for the JSON case. This section also contains a comparison of CTC applied to XML and JSON documents Finally, Section 7.3 shows the CTC evaluation within a typical REST architecture deployment and focused on the impact of CTC on the transmission efficiency and communication load.

7.1 XML Compression Performance Evaluation

In this section, two performance tests are presented in order to compare CTC and EXI implementations. The results are grouped into three sections, each one focused on one performance metric: compression size, processing time and memory usage.

In the first test, a set of XML instances are compressed by using (a) EXIficient [16b], an EXI implementation, and (b) a prototype implementation of the CTC approach. The purpose of this test is to evaluate and compare the compression rate of both implementations.

In the second test, the compressed data streams obtained from the previous test are decompressed to analyse the required processing time and memory usage. For the decodification of the EXI streams, another EXI implementation more suited to resource-constrained systems is used, Embeddable EXI Processor (EXIP) [KPED14]. EXIP is considered here because, to the author's knowledge, it is the best suited to resource-constrained systems. Other implementations targeted to resource-constrained systems, such as WS4D-uEXI [WS4], only implement a subset of the EXI specification and/or are somewhat outdated.

The tests were performed in a CC2650 MCU [CC2] running at 48MHz. The test applications as well as the code under test were compiled with the optimizations turned on.

The set of XML documents used in the tests is composed of the XML Schema instances for Network Configuration Protocol (*netconf*) [net], Media Types for Sensor Markup Language (*SenML*) [JSA+18], Data Types definitions for OPC-UA (*OPC-UA Types*) [fou] and Zigbee Smart Energy Profile 2.0 (*SEP2*) [SEP], presented in the EXIP evaluation paper [KPED14]. As in [KPED14], three different documents per XML Schema are considered. Additionally, the *notebook* XML document used as an example in the EXI Primer web page [PPG14] is also included.

7.1.1 First comparison: compression size

For this comparison, each XML document was compressed using the EXIficient [16b] EXI processor implementation and a prototype implementation of the CTC approach. Then the resulting compressed streams were compared between them to evaluate the overall and relative compression size efficiency.

In order to ensure fairness, the EXI compression options were carefully configured. First, the EXI "schema strict" compression mode was selected. This mode takes into account the XML Schema(s) that describe a XML document in order to achieve the most compact compression. Additionally, all the EXI *preserve* options were set to FALSE, reducing the overhead that may be produced by compressing meta-data, such as comments. Finally, the EXI *schemaId* option was set to the constant string "1". This ensures that the *schemaId* option is included in the EXI header but removes the arbitrary overhead of long identifiers. Although, EXI *schemaId*s are rarely one character long and it benefits EXI regarding compression size, this configuration sets a best case EXI benchmark for the comparison.

For each XML document, four cases are considered: with/without EXI Profile parameters and with/without including the EXI options in the EXI header. In CTC, the *context table*s and *template table*s were created from the XML Schemas and the compression was performed using the CTC approach. Results in terms of size in bytes and relative compression are shown in Table 7.1 and Figure 7.1.

In Table 7.1, the *Document* column lists the XML document instances while the column *Size* contains the respective original sizes in bytes. The *EXIP* case contains results for schema strict mode with all *preserve* options to FALSE and *schemaId* to constant string "1". The *EXIP-EP* lists the results for the same configuration as the *EXIP* case plus the EXI Profile option enabled. *CTC* and *CTC-S* cases respectively contain the results for the CTC normal and strict modes. For each case in the table, two columns are included to show the compressed size in bytes (B) and proportional compared to the original XML

Table 7.1: XML document compression comparative in bytes.

Document	Size	EXIP		EXIP-EP		CTC		CTC-S	
		B	%	B	%	B	%	B	%
notebook	297	(3+)59	20.9	(13+)59	24.2	62	20.9	61	20.5
netconf-01	395	(3+)21	6.1	(13+)21	8.6	21	5.3	21	5.3
netconf-02	660	(3+)51	8.2	(13+)51	9.7	50	7.6	50	7.6
netconf-03	423	(3+)3	1.4	(13+)3	1.7	3	0.7	3	0.7
SenML-01	448	(3+)97	22.3	(13+)98	24.8	138	30.8	130	29.0
SenML-02	219	(3+)61	29.2	(13+)61	33.8	64	29.2	60	27.4
SenML-03	173	(3+)45	27.7	(13+)45	33.5	46	26.6	45	26.0
SEP2-01	406	(3+)64	16.5	(13+)64	19.0	65	16.0	64	15.8
SEP2-02	92	(3+)19	23.9	(12+)19	33.7	19	20.7	19	20.7
SEP2-03	522	(3+)27	5.7	(13+)27	7.7	24	4.6	24	4.6
OPC-UA-01	936	(3+)61	6.8	(12+)62	7.9	73	7.8	73	7.8
OPC-UA-02	278	(3+)4	2.5	(12+)4	5.8	4	1.4	4	1.4
OPC-UA-03	300	(3+)4	2.5	(13+)4	5.7	4	1.3	4	1.3

document instance (%). In both *EXIP* and *EXIP-EP* cases, numbers inside brackets indicate the extra overhead in bytes due to EXI options embedded in the EXI header.

The cases depicted in Figure 7.1 are the same ones included in Table 7.1. Stacked bars indicate the extra overhead due to EXI options embedded in the EXI header.

For instance, in the "notebook" row of Table 7.1, the original XML document has a size of 297 bytes and, for the *EXIP* case, EXI compresses it to 59 bytes plus a header overhead of 3 bytes, achieving a relative size of 20.9%. For the *EXIP-EP* case, the compressed size is 59 bytes plus a header overhead of 13 bytes which result in a relative size of 24.2%. In the *CTC* case, the achieved compression size is equal to the *EXIP* case, i.e., 62 bytes but the *CTC-S* case has slightly better results with a compressed size of 61 bytes and a relative size of 20.5%.

Results show that CTC has a very similar compression size performance compared to EXI and an average better performance if we take into account the EXI header. The overhead produced by the EXI header could be overcome by providing the EXI options out of band but it would imply a loss of flexibility, which is one of the strong points of EXI. However, it may be necessary in order to reduce communication bandwidth, specially for the EXI Profile case. On the other hand, it is interesting to note that in the case of the *SenML-01* document, EXI shows better compression results. The reason lies in the fact that, in its current version, CTC is not able to compress occurrences of strings outside the schema, while EXI does not differentiate between strings belonging to data or to the schema vocabulary. CTC sacrifices this capability for the benefit of simplicity and still achieves similar or better results than EXI, as can be seen in Table 7.1.

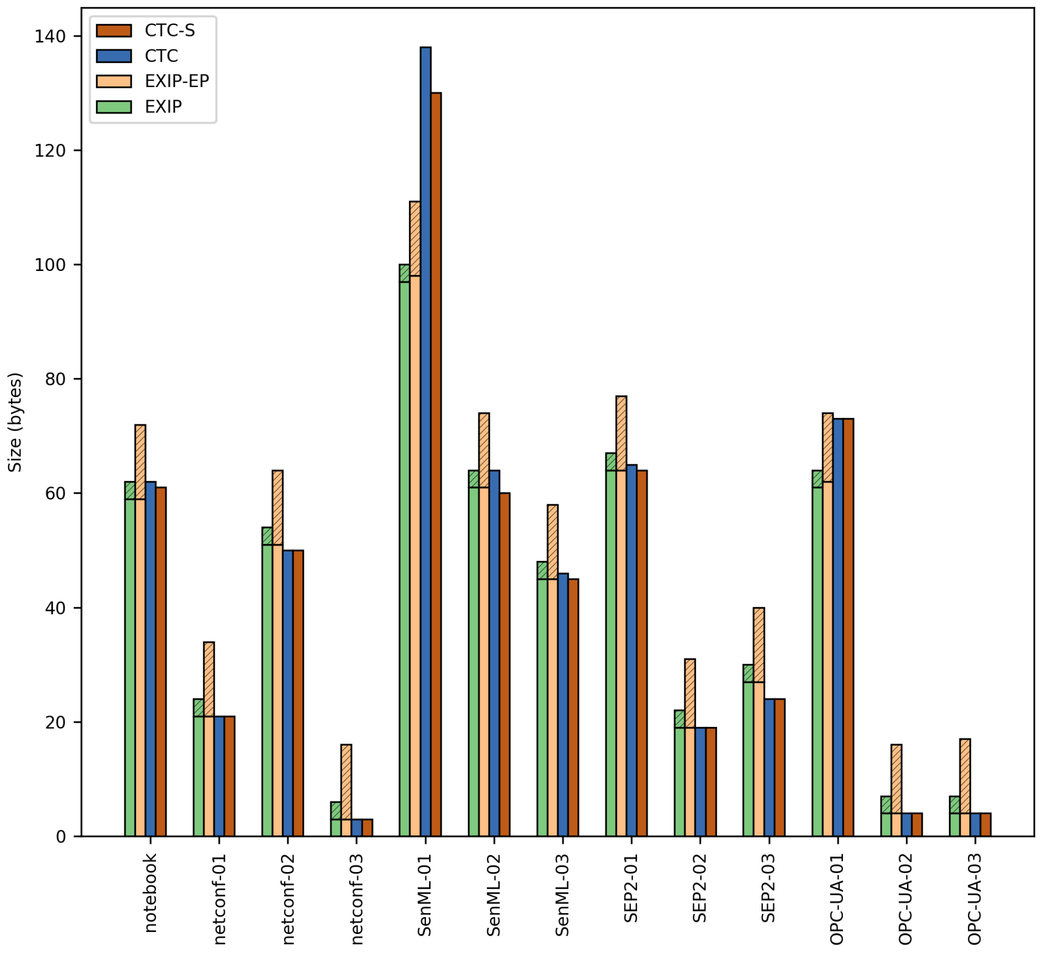


Figure 7.1: XML document compression comparative in bytes.

7.1.2 Second comparison: processing time

In the second test, EXI streams produced in the previous compression test (described in Section 7.1.1) were decoded using the EXIP v5.4 [16a] EXI implementation and a prototype implementation of the CTC approach. The time needed to process each coded stream was recorded in order to evaluate the overall and relative processing time efficiency.

In the EXIP case, the EXI grammars were statically created from the set of XML test schemas using the tools provided by EXIP and they were included in the EXIP test code. These EXI grammars were then used at runtime by EXIP to perform the decoding of the EXI streams. In the case of the CTC approach, the *context table*s and *template table*s were created using the *CTC Compiler* and added to the CTC test code as described in Section 6.2. A prototype implementation of CTC was used to decode the CTC streams.

100 test runs were performed for each EXI and CTC compressed stream with no additional system load. As in the previous test, we considered four EXI cases for each XML instance document decoding: with/without EXI Profile parameters and with/without including the EXI options in the EXI header. For the EXI Profile cases, EXI Profile options had to be stripped from the EXI header of the EXI compressed streams because they are not supported by EXIP. This means that the results shown for the EXI Profile cases are actually slightly better than they would be in a full featured implementation, benefiting EXI in the comparison of this section.. Nevertheless, these results are used as a benchmark for the processing time evaluation as we consider them as a worst case scenario for CTC.

Table 7.2 and Figure 7.2 show the test results. The *EXIP* column list the results for EXIP configured in schema strict mode, all *preserve* options to FALSE and *schemaId* to constant string “1”. The *EXIP-H* contains the results for the same configuration as the *EXIP* column but with the EXI options embedded in the EXI header. The *EXIP-EP* column shows the results for the same configuration as *EXIP* case but also including the EXI Profile option. The results of the *EXIP-EP-H* column have the same configuration as the *EXIP-EP* column but with the EXI options embedded in the EXI header, excluding EXI Profile parameters as explained in the previous paragraph. The *CTC* column shows the processing time for the CTC case in normal and strict modes. Only one column is presented for both modes because there is no notable difference between the results yield by CTC in normal and strict modes. The *without-H* column contains the average processing time improvement (%) in the CTC case compared to the EXIP cases with EXI options not included in the EXI header, i.e., *EXIP* and *EXIP-EP*. The *with-H* column lists the average processing time improvement (%) compared to the EXIP cases with the EXI options embedded in the EXI header, i.e., *EXIP-H* and *EXIP-EP-H*.

For example, the “netconf-01” row of Table 7.2 shows that EXIP needs 290μs, 531μs, 314μs and 629μs to process the “netconf-01.xml” XML document respectively for the *EXIP*, *EXIP-H*, *EXIP-EP* and *EXIP-EP-H* test configurations. In contrast, CTC only requires 152μs

Table 7.2: XML document decoding time comparative. Numbers are in microseconds.

Document	EXIP	EXIP-H	EXIP-EP	EXIP-EP-H	CTC	Improvement (%)	
	Time (μs)					without-H	with-H
notebook	684	929	574	808	625	0.6	28.0
netconf-01	290	531	314	629	152	49.6	73.8
netconf-02	814	1049	946	1251	518	41.1	54.9
netconf-03	286	518	320	621	183	39.6	67.8
SenML-01	1576	1817	1409	1635	1493	0.0	13.5
SenML-02	726	966	615	840	618	7.8	31.5
SenML-03	465	705	591	511	377	28.6	38.0
SEP2-01	1186	1429	770	1004	910	6.9	25.2
SEP2-02	553	787	213	437	230	39.9	62.4
SEP2-03	1210	1446	860	1079	763	26.3	39.5
OPC-UA-01	1804	1935	1251	1385	788	48.4	52.5
OPC-UA-02	500	642	98	244	81	72.9	81.7
OPC-UA-03	540	684	142	285	101	70.4	79.1

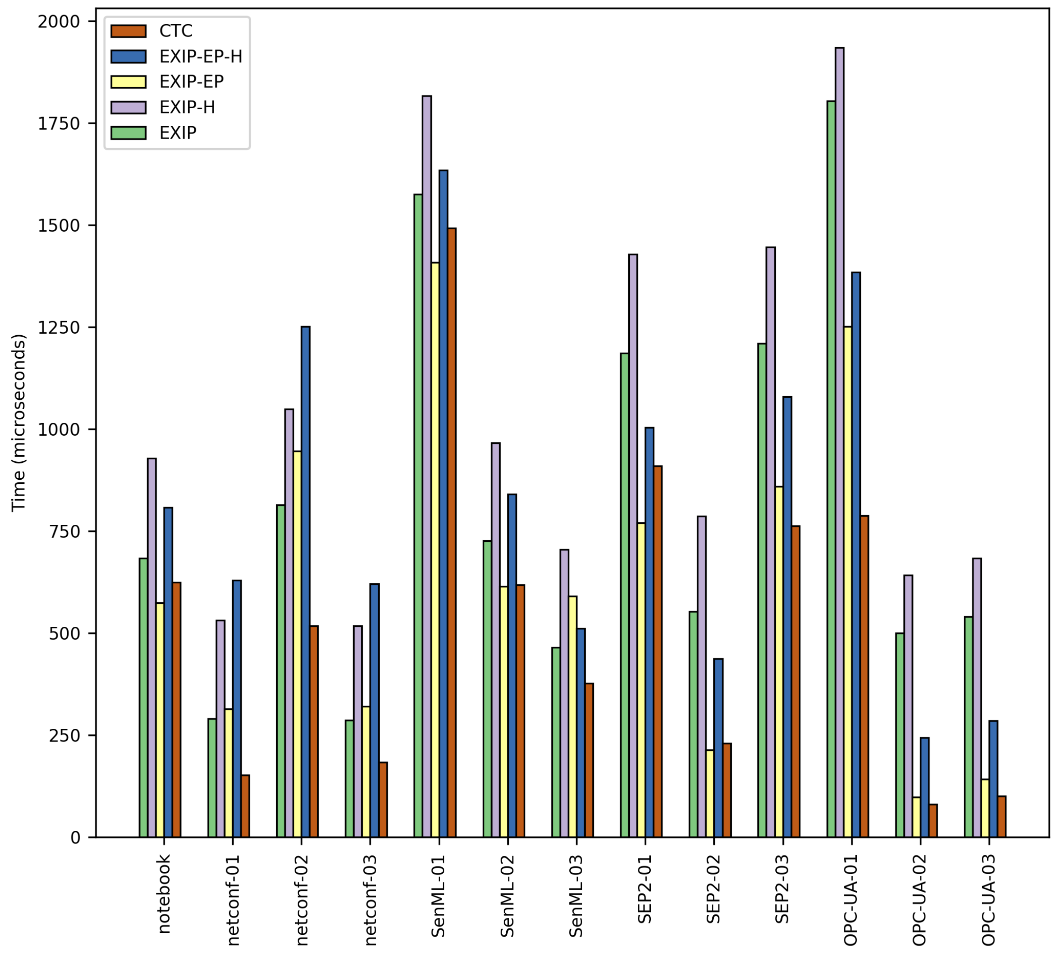


Figure 7.2: XML document decoding time comparative.

which implies a processing time improvement on average of 49.6% compared to the *EXIP* and *EXIP-EP* cases, and of 73.8% compared to the *EXIP-H* and *EXIP-EP-H* cases.

Results from the tests show that CTC performed generally better in terms of processing time compared to any EXIP case: an average of 36.6% time reduction and up to 87.4% time reduction. This is a direct result from using the simpler CTC approach and structure of the *context table*s and *template table*s compared to the EXI specification and EXIP implementation. This difference in processing time will be more pronounced in devices with slower CPUs such as the popular TelosB [Tel] , which runs at 8MHz. For those cases, times shown in Table 7.2 will be considerably incremented. Reducing processing time is key in resource-constrained devices in order to reduce the energy consumption as much as possible and make the most of the available energy.

7.1.3 Third comparison: memory usage

In the last comparison, memory usage of the EXIP library and CTC prototype implementations were compared in terms of required code and runtime data memories (i.e. code size, data, heap and stack consumption).

The EXIP library supports a dedicated compilation configuration for EXI Profile. In this configuration, the EXIP library code and EXI grammars are more compact and the RAM usage is notably reduced. Both compilation configurations, normal and EXI Profile, were taken into account in the comparison. For CTC, the memory consumption for the *context table*s and *template table*s are separately considered. Measures were taken from the test applications used in the second test (described in Section 7.1.2). Results are listed in Table 7.3 and Table 7.4.

Table 7.3 shows the flash code memory used by the EXIP and CTC implementations. Memory usage is listed for the EXI and CTC base libraries in the *Library* row, and for each XML schema in the rows labelled as “*.xsd”. The *EXIP* column contains the flash memory usage in bytes of the EXIP library in normal mode while column *EXIP-EP* lists the results in the EXI Profile configuration. Two columns are shown for the *CTC* case. The *Core* column contains the memory required when the *template table* is not included in the application code while the *Template* column indicates the additional memory space in bytes needed by the *template table* in case it is included. Finally, the *Comparison* column shows the relative size in % of the *CTC* case compared to the *EXIP* and *EXIP-EP* cases, without and with the *template table* included in the application code.

Figure 7.3 shows the Flash memory usage listed in Table 7.3. Stacked bars in the *CTC* columns indicate the extra overhead of the *template table*.

For example, in the “netconf.xsd” row, the EXIP library requires 8226 bytes to include the schema information (i.e., the EXI grammars) in the normal mode, while it needs 8979 bytes with the EXI Profile capabilities enabled. The CTC implementation uses 1224 bytes

Table 7.3: Flash code memory usage comparative in bytes.

Component	EXIP	EXIP-EP	CTC		Comparison (%)			
			Core	Template	EXIP		EXIP-EP	
Library	21493	21794	1722	0	8.0	8.0	7.9	7.9
notebook.xsd	4786	5242	208	196	4.3	8.4	4.0	7.7
netconf.xsd	8226	8979	1224	1812	14.9	36.9	13.6	33.8
SenML.xsd	5550	6064	320	300	5.8	11.2	5.3	10.2
SEP2.xsd	85776	94500	25560	27188	29.8	61.5	27.0	55.8
OPC-UA.xsd	133528	130823	21172	42396	15.9	47.6	16.2	48.6

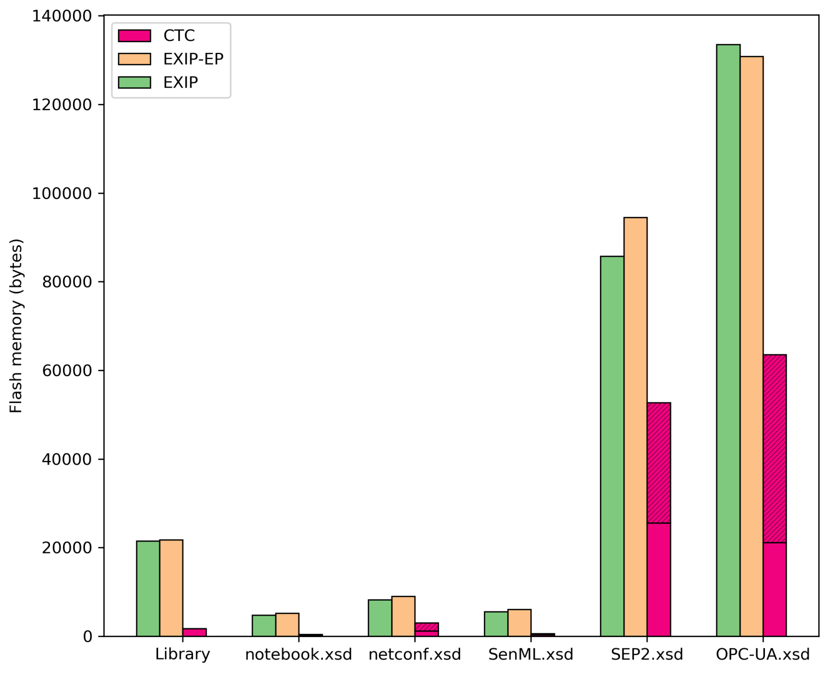


Figure 7.3: Flash code memory usage comparative in bytes.

for the same schema and an additional 1812 bytes in case the *template table* is included. For the “netconf.xsd” schema, the CTC implementation consumes the 36.9% memory size of the *EXIP* case, and the 14.9% if the *template table* is not included.

Table 7.4 shows the consumption of data memory (RAM). The rows and columns have the same meaning as in Table 7.3, with the notable exception that the used memory refers to data memory rather than code memory. Apart from the data memory usage shown in Table 7.4, the maximum heap and stack used for the *EXIP* case is 1734 and 904 bytes respectively. For the *EXIP-EP* case, maximum heap and stack usage amounts to 1294 and 792 bytes respectively. Finally, CTC uses no heap and the maximum stack size used for the tests is 692 bytes.

Results show that CTC requires significant less code and data memory than EXIP. In the case of the base library, CTC takes 8.0% the size of the EXIP implementation and the

Table 7.4: Data memory (RAM) usage comparative in bytes.

Component	EXIP	EXIP-EP	CTC
Library	292	292	12
notebook.xsd	1196	252	0
netconf.xsd	1748	292	0
SenML.xsd	1348	292	0
SEP2.xsd	11860	292	0
OPC-UA.xsd	13765	292	0

7.9% of the EXIP EXI Profile implementation. For the XML test schemas, the comparative size ranges from 7.7% to 61.5%. As has been explained in Chapter 6, the *template table* can be stripped from devices that do not make use of it, thus, reducing even further CTC memory requirements for the XML test schemas. In this case, the comparative memory usage will be reduced to 4.0% to 29.8%.

Additionally, CTC uses nearly no data RAM while EXIP uses 252 bytes in its best case and up to 13765 bytes in the worst one. The maximum heap used for the *EXIP* case is 1734 and 1294 bytes for the *EXIP-EP* case. In contrast, CTC uses no heap memory at all.

Devices need to share the memory between multiple functionalities (application, sensor drivers, communication stacks, etc.) and will likely need to accommodate more than one schema. Thus, it is of most importance to assign memory resources as efficiently as possible and make the most of the available memory. Although the CTC Library used in these tests is still a prototype, results show that CTC memory requirements are much more suited to resource-constrained devices than EXI implementations due to the significantly smaller memory footprint and runtime usage requirements. On the one hand, the CTC core library requires 92% less memory than the EXIP library configured for EXI Profile, and the schema information is reduced to a 4.0% in the best case and to a 61.5% in the worst. On the other hand, CTC core library uses no RAM, no heap and a maximum stack size of 692 bytes, while EXIP uses 252 RAM bytes, 1294 heap bytes and 792 stack size in its best case.

7.2 JSON Compression Evaluation

In this section the compression efficiency of CTC for JSON document instances is compared against EXI4JSON [PB18] and CBOR [BH13]. A set of JSON instances are compressed using (a) a prototype implementation of CTC, (b) EXIficient [16b], an EXI implementation, and (c) CBOR diagnostic utilities [18b], a command line set of utilities provided by the CBOR community.

This section also includes a comparison of the efficiency in terms of size for two different data formats, XML and JSON, and their respective CTC codification.

The set of JSON instance documents used in the tests is composed of a series of JSON Schema instances for Media Types for Sensor Measurement Lists (SenML) [JSA+18], the JSON implementation of OGC Observations and Measurements (O&M) specification [SJDT15] and the GeoJSON Format [BDD+16]. For the GeoJSON case, the schema used is the geometry schema defined within the JSON OGC O&M specification [SJDT15].

Three different instance documents per JSON Schema are considered. For the SenML case, the three examples included within sections 5.1.2 and 5.1.3 of the SenML specification [JSA+18] are used. In the O&M case, the three instances used are the examples included in the JSON OGC O&M specification [SJDT15] within sections 7.6 (Specimen data) and 7.8 (Sampling feature collection) as well as the first example of section 7.9 (Observation data). Finally, the GeoJSON instance used are taken from the examples included in the GeoJSON Format [BDD+16] specification in sections A.2 (LineStrings), A.3 (Polygons with no holes) and A.6 (MultiPolygons).

For this comparison, the JSON instance documents were first compressed using the EXIficient [16b] EXI processor implementation and the CBOR diagnostic utilities [18b]. EXIficient includes a mode to compress JSON schemas using the EXI4JSON [PB18] approach. This mode automatically sets the EXI options to the ones specified in the EXI4JSON recommendation: the EXI "schema strict" compression mode is set to TRUE and the EXI *schemaId* option is set to the constant string "exi4json". The alignment option is set to "bit-packed". On the other hand, CBOR diagnostic utilities includes a tool to automatically convert from JSON instances to CBOR. No specific label or tag mappings have been used in the codification. Finally, the corresponding CTC *schema contexts* and *template tables* were created from the JSON Schemas and the compression was performed using the CTC approach.

Results in terms of size in bytes are shown in Table 7.5. The Size column lists the original size of the respective JSON instance. The table shows the results for the EXI4JSON, CBOR, CTC in normal mode and CTC in strict mode (column *CTC-S*) in their corresponding columns. Each case includes two (sub-)columns in order to show the absolute size in bytes (column *B*) and proportional to the original size (column *%*).

For example, the original size of the "GeoJSON-01" JSON instance is 78 bytes. EXI4JSON is able to compress the "GeoJSON-01" instance to 47 bytes resulting in a 60.3% of the original size while CBOR achieves a compression to 44 bytes for a 56.4% size compared to the original size. In contrast, the size of the CTC compressed data is 13 and 12 bytes respectively for the CTC normal and strict modes, with a relative size of 16.7% and 15.4%.

Results show that CTC outperforms both EXI4JSON and CBOR codifications. EXI4JSON and CBOR achieve an average relative size of 69.8% and 73.6% respectively while CTC shows an average relative size of 39.2% (38.2% in the CTC *Schema Strict* mode). EXI4JSON does not take advantage of the JSON Schema directly as it depends on a mapping to a

Table 7.5: JSON instance document compression comparative in bytes (B) and proportion (%).

Document	Size	EXI4JSON		CBOR		CTC		CTC-S	
		B	%	B	%	B	%	B	%
SenML-01	103	83	80.6	86	83.5	61	59.2	60	58.3
SenML-02	291	172	59.1	223	76.6	142	48.8	140	48.1
SenML-03	406	233	57.4	287	70.7	184	45.3	185	45.6
GeoJSON-01	78	47	60.3	44	56.4	13	16.7	12	15.4
GeoJSON-02	116	63	54.3	63	54.3	29	25.0	27	23.3
GeoJSON-03	259	136	52.5	202	78.0	86	33.2	83	32.0
O&M-01	520	453	87.1	449	86.3	270	51.9	264	50.8
O&M-02	335	294	87.8	274	81.8	142	42.4	137	40.9
O&M-03	486	454	93.4	446	91.8	274	56.4	269	55.3

Table 7.6: senML instances' size comparison in bytes for XML, JSON and CTC cases.

Document	XML	JSON	CTC		CTC-S	
			CTC-X	CTC-J	CTC-X	CTC-J
SenML-01	171	103	61	61	60	60
SenML-02	414	291	143	143	140	141
SenML-03	605	406	184	184	185	185
SenML-04	118	56	39	39	39	39

dedicated XML Schema, as explained in Section 3.2.1.4. Although the resulting compression is undoubtedly better than the raw JSON, it is outperformed by CTC, which takes advantage of the JSON schema knowledge for the compression. On the other hand, CBOR does not take into account schema information and suffers from the overhead of in-lining the data types into the coded stream.

7.2.1 XML, JSON and CTC comparison

In this section we will compare the size of data formatted in two different data formats, XML and JSON, as well as the resulting CTC compression efficiency. The selected data model for this evaluation is JSON Schema instances for Media Types for Sensor Measurement Lists (SenML) [JSA+18]. The SenML specification describes several examples in different data formats and encodings, including XML and JSON. The example documents used in this evaluation are the four examples found within sections 5.1.1 (SenML-04), 5.1.2 (SenML-01 and SenML-02) and 5.1.3 (SenML-03) of the SenML specification [JSA+18].

Table 7.6 summarizes the original and compressed sizes of the selected instance documents. The *XML* column contains the size of the data in XML format and the *JSON* column the size of the data in JSON format. The *CTC* and *CTC-S* columns show the compressed size in bytes for CTC in normal and strict modes, respectively for the XML (columns *CTC-X*) and JSON (columns *CTC-J*) cases.

As can be seen in Table 7.6, JSON format is consistently more compact than XML. This is due to the less verbose nature of JSON that uses less grammar constructs and

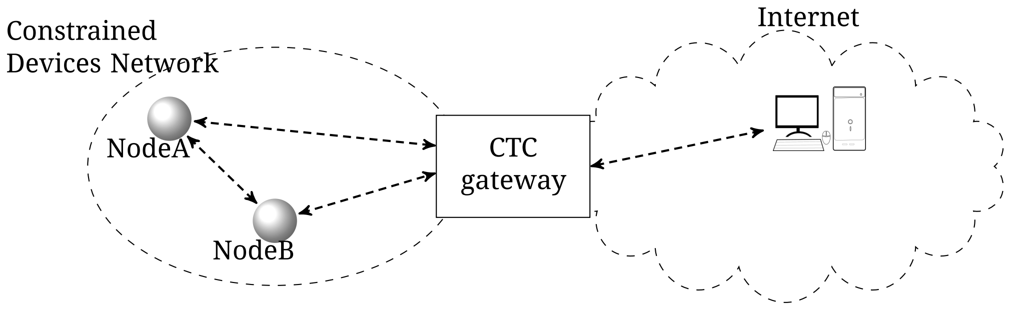


Figure 7.4: REST validation deployment.

string tokens to describe relationships between items of the structure. Nevertheless, CTC codification yields similar results for both XML and JSON cases. This is because CTC mapping captures the core structure of the data model and segregates data format specific artefacts. The codified stream only includes the minimum needed information in order to properly decode and parse the stream while the data format specific information (mainly stored in the *template table*) is only used if the original format must be reconstructed.

7.3 CTC Impact on Communication Performance

In this section we will show the CTC performance within a REST communication architecture as well as evaluate the impact on the communication load. We consider the following deployment (shown in Figure 7.4): two resource-constrained devices (NodeA and NodeB) are deployed in a local network and are connected to an external network through a gateway. For all the tests performed in this evaluation, the roles of the nodes are the same.

NodeA produces data periodically and NodeB gathers it at a regular time rate. Both devices use the CoAP protocol to implement the REST interface. CoAP Block-Wise Transfers [BS16] are used to send fragmented data streams that do not fit into a single IP Frame.

Data are accessible on NodeA through the path “/out/data”. CoAP requests and subscriptions are allowed on that path. The layout of NodeA’s CoAP API is shown in Figure 7.5a. On the other hand, NodeB consumes the data produced by NodeA. To do so, NodeB subscribes to the path “/out/data” provided by the CoAP API of NodeA. After the subscription, NodeA periodically sends data to NodeB. The subscription message is not taken into account in the tests results and evaluation. The layout of NodeB’s CoAP API is shown in Figure 7.5b.

The role of the gateway is to act as an interface between the local network and the external network, as well as host the CTC *schema repository*. Additionally, the gateway has

```
/.well–known/core
/sch/schema1
/out/data
```

(a) NodeA CoAP API layout.

```
/.well–known/core
/sch/schema1
```

(b) NodeB CoAP API layout.

Figure 7.5: Layouts of NodeA and NodeB CoAP APIs.

to act as a CTC Gateway i.e., transform between CTC and the original format, whenever required.

The purpose of this test is to evaluate the impact on the communication load of using CTC and CTC communication model mechanisms. In order to make a fair comparison, we also take into account the overhead produced by the CTC discovery, schema register and schema download processes. We consider two cases: (a) data formatted in the original data format and (b) compressed with CTC. For the second case, the CTC communication model is used.

A set of three different data model schemas and instances are used in the evaluation: OPC-UA, SEP2 and senML. Each of these data models represents a different benchmark that can be found in a realistic scenario. On the one hand, senML instance and schema are relatively compact and set a reasonable benchmark for a lower bound. On the other hand, the OPC-UA instance and schema are much more verbose and set a good example of a heavy data model. Finally, SEP2 schema is very heavy but the SEP2 instance is very compact. This last example represents a reasonable worst case for the CTC validation, because there is a big overhead due to the schema size but CTC compression ratio is low due to the small size and compactness of the instance.

The specific data model instances used in the tests are shown in Figure 7.6, Figure 7.7 and Figure 7.8. The respective schemas are available in [fou], [SEP] and [JSA+18].

These three data model instance and schema sets allow us to set reasonable low and high benchmark bounds for the evaluation in order to compare the impact of the data model instance and schema size in the overall performance. Thus, for each of the data formats (original and CTC), we run three sets of tests, one for each data model.

The discovery process needed to discover the location of the *schema directory* is taken into account in the evaluation. However, no other discovery steps (such as the discovery of the CoAP gateway) are considered in order to remove any overhead not specifically related to CTC. Thus, the parameters usually configured at runtime through the CoAP discovery process are hard-coded in the nodes. For example, the resource paths used by the test applications (i.e., used by NodeB to request data to NodeA) are statically assigned at programming time.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:Node xmlns:tns="http://opcfoundation.org/UA/2008/02/Types.xsd" ↩
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ↩
    xsi:schemaLocation="http://opcfoundation.org/UA/2008/02/Types.xsd ↩
    ../Schemas/OPC-UA-Types.xsd " xsi:type="tns:Node">
    <tns:NodeId>
        <tns:Identifier>tns:Identifier</tns:Identifier>
    </tns:NodeId>
    <tns:NodeClass>Unspecified_0</tns:NodeClass>
    <tns:BrowseName>
        <tns:NamespaceIndex>0</tns:NamespaceIndex>
        <tns:Name>tns:Name</tns:Name>
    </tns:BrowseName>
    <tns:DisplayName>
        <tns:Locale>tns:Locale</tns:Locale>
        <tns:Text>tns:Text</tns:Text>
    </tns:DisplayName>
    <tns:Description>
        <tns:Locale>tns:Locale</tns:Locale>
        <tns:Text>tns:Text</tns:Text>
    </tns:Description>
    <tns:WriteMask>0</tns:WriteMask>
    <tns:UserWriteMask>0</tns:UserWriteMask>
    <tns:References>
        <tns:ReferenceNode/>
    </tns:References>
</tns:Node>
```

Figure 7.6: OPC-UA data model instance.

```
<Reading href="/upt/0/mr/4/r" xmlns="http://zigbee.org/sep">
    <value>14</value>
</Reading>
```

Figure 7.7: SEP2 data model instance.

```
<?xml version="1.0" encoding="UTF-8"?>
<senml xmlns="urn:ietf:params:xml:ns:senml" bn="urn:dev:ow:10e2073a01080063" >
    <e n="voltage" t="0" v="120.1" u="V" />
    <e n="current" t="0" v="1.2" u="A" />
</senml>
```

Figure 7.8: SenML data model instance.

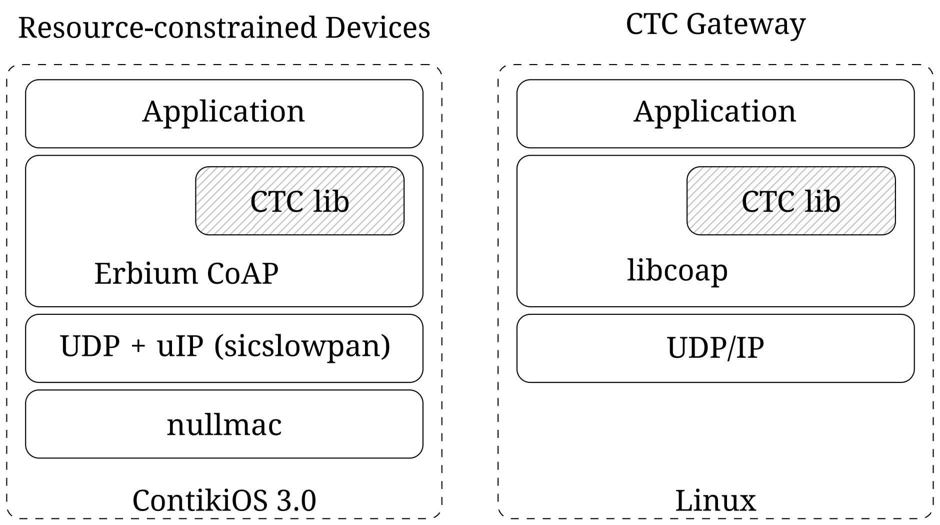


Figure 7.9: Software architecture of the resource-constrained devices and CTC Gateway.

The simplified software architecture, including the communication stack, of the two resource-constrained devices and the gateway is shown in Figure 7.9. As can be seen, all the resource constrained devices use Contiki OS 3.0 [DGV04, 18c] as an operating system and the communication stack is composed by Contiki's nullmac, uIP and CoAP components. The platform used is Zolertia's Z1 [Lig16] and the tests were run within the Cooja [ÖDE+06, 16c] simulator. The CTC gateway is implemented on a PC running an Ubuntu 14.04 distribution. The CTC gateway and *schema repository* applications are implemented on top of the libCoAPv4.1 [Ber17] CoAP library and is connected to the simulated network through a virtual socket provided by Cooja. All the nodes use a prototype implementation of the *CTC Library*. The specific binding of the *schema repository* approach to CoAP is explained in more detail in Section 5.3.

7.3.1 Message Fragmentation

Contiki allows the configuration of the maximum IP frame size in order to fit it to the available resources on the device. The selection of this configuration parameter also bounds the maximum allowed size of the CoAP message and CoAP block size. The CoAP block size indicates the maximum data payload that can be sent in a single CoAP message. These two parameters greatly influence the results of this evaluation.

On one hand, it is always desirable to fit a data unit in the payload of a single message frame (Case a) in Figure 7.10) in order to avoid fragmentation. In case the data is larger than the allowed payload, it has to be split into multiple payload fragments and a corresponding IP and COAP headers is attached to each of them to compose an IP frame, as can be seen in Case b) in Figure 7.10. In turn, if the IP frames do not fit within a MAC frame payload, each of them has to be further divided into smaller fragments with their own MAC-header, as shown in Case c) in Figure 7.10. Fragmentation increases the

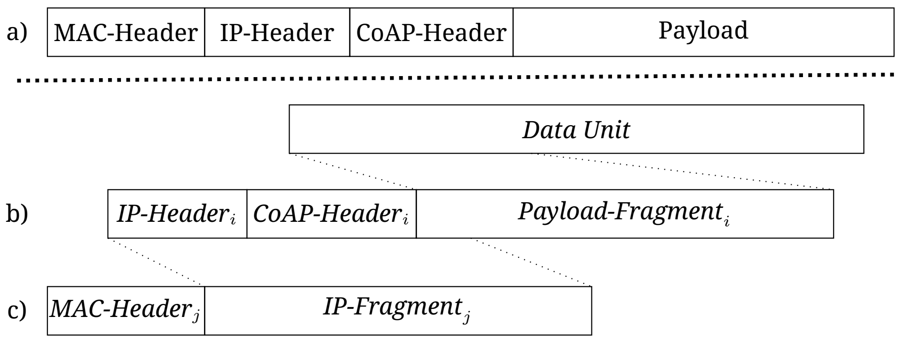


Figure 7.10: Message fragmentation.

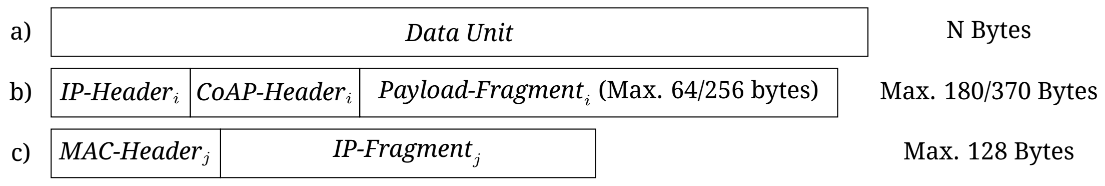


Figure 7.11: Message structure fo the evaluation.

number of total bytes sent because the headers of the underlying communication layers have to be re-sent together with the fragmented payloads.

Increasing the IP frame size, reduces IP level fragmentation and the overhead of additional IP headers. However, a buffer has to be keep in RAM in order to fit a whole IP frame. Thus, there is a physical limit to the size of the IP frame, which tends to be rather low in resource-constrained devices. For instance, for the Z1 platform [Lig16] (8KB RAM and 92KB Flash) with Contiki case, the transmission buffer tends to be in the order of 140 to 256 bytes and, for the test applications used in this evaluation (which are fairly simple), using a value larger than 400 bytes yields a memory over-usage error.

Nevertheless, using a low IP frame size would increase the number of messages needed to send a data unit, which may unfairly benefit the CTC case. In this evaluation, a data unit corresponds to the document instances used in the tests which will vary in size, as depicted in Case a) in Figure 7.11.

For the evaluation tests performed in this section, we used two message frame configurations (shown in Case b) in Figure 7.11). In the first configuration, the IP frame size selected is 180 bytes, enough to fit a CoAP block size of 64 bytes. In the second configuration, the selected IP Frame size is 370 bytes and the CoAP block size is 256 bytes. These two configurations will show the relative impact of CTC in two realistic data fragmentation situations while keeping a reasonable (yet somewhat optimistic for the second configuration) use of available RAM. In all cases, the maximum number of bytes that fit in a single MAC layer message is 128 (Case c) in Figure 7.10).

```
</.well−known/core>;ct=40,
</sd>;rt="core.sd";ct=40,
</sd−sch>;rt="core.sd−lookup−sch";ct=40,
</sd−sid>;rt="core.sd−lookup−sid";ct=40
```

Figure 7.12: Link format information returned by the *schema repository*.

```
</sch/schema>;ct=41;rt="schema";uid=http://opcfoundation.org/UA/2008/02/Types.xsd;sz↩
    =170892;hash="92CE25B721D41DDCB1CA021292D552A1",
</schema1/schema>;ct=41;rt="schema";uid=http://opcfoundation.org/UA/2008/02/Types.xsd;↩
    anchor="http://example.com";sz=170892;hash="92CE25B721D41DDCB1CA021292D552A1"
```

Figure 7.13: *Schema link register* for the OPC-UA case.

7.3.2 Overhead of the schema register and download processes

In a first step, we separately evaluate the communication resources needed in terms of sent messages for the schema registration and schema download processes. This information will be used latter to evaluate the impact of CTC in the communication performance.

The registration process is performed as explained in Section 5.2.2 and the specific binding to CoAP is described in Section 5.3. Basically, it is composed of two steps: a) the discovery of the *schema repository* resources and b) the actual registration of the *schema link register*. The resource links returned by the *schema repository* discovery are shown in Figure 7.12. The *schema link registers* used for the registration of the OPC-UA, SEP2 and senML data models are shown in Figure 7.13, Figure 7.14 and Figure 7.15.

Table 7.7 summarizes the message quantity needed to perform the registration and schema download processes of each of the data models. The *size* column of the "register" rows contain the size in bytes of the resource links returned in the discovery phase, as shown in Figure 7.12, plus the size of the respective *schema link registers*, i.e., Figure 7.13, Figure 7.14 and Figure 7.15. For the "schema" rows, the *size* column shows the size in bytes of the respective schema. The message quantity for each CoAP block size configuration, i.e., 64 and 256, is respectively shown in columns *Block:64* and *Block:256*, together

```
</sch/schema>;ct=41;rt="schema";uid=http://zigbee.org/sep;sz=287778;hash="↩
    CEBC11A7EF98E589D624D47B3B4E3935",
</schema1/schema>;ct=41;rt="schema";uid=http://zigbee.org/sep;anchor="http://example.↩
    com";sz=287778;hash="CEBC11A7EF98E589D624D47B3B4E3935"
```

Figure 7.14: *Schema link register* for the SEP2 case.

```
</sch/schema>;ct=41;rt="schema";uid=urn:ietf:params:xml:ns:senml;sz=965;hash="27↩
    F18B40A6037DC7CBA8CE00A3F4DD3A",
</schema1/schema>;ct=41;rt="schema";uid=urn:ietf:params:xml:ns:senml;anchor="http://↩
    example.com";sz=965;hash="27F18B40A6037DC7CBA8CE00A3F4DD3A"
```

Figure 7.15: *Schema link register* for the senML case.

Table 7.7: Transmitted messages (max. 128 bytes) and raw bytes quantity for the registration and schema download processes for two cases: 64 and 256 bytes per CoAP block.

Process	Size	Messages / Bytes	
		Block:64	Block:256
OPC-UA register	130+295	18 / 1527	13 / 1267
OPC-UA schema	170892	5342 / 421932	2671 / 27370
SEP2 register	130+247	16 / 1355	11 / 1095
SEP2 schema	287778	8994 / 711264	4498 / 460934
senML register	130+255	16 / 1363	11 / 1103
senML schema	936	30 / 2314	15 / 1514

with the total number of bytes (separated by a "/").

For instance, in the case of the "OPC-UA register" process, the size of the resource links is 130 bytes and the size of the *schema link registers* is 295 bytes. With a CoAP block configuration of 64 bytes, 18 messages are needed to transmit the data for a total of 1527 bytes including headers' overheads. In the case of the CoAP block size configuration of 256 bytes, the message quantity is reduced to 13 and the number of transmitted bytes to 1267. Note that, for all cases, the maximum number of bytes per message is 128 as explained in the introduction to Section 7.3.

As can be seen, when the size of the CoAP block is 64, the number of total bytes transmitted is higher. This is a direct result of the overhead produced by fragmentation and the need to send additional IP headers.

7.3.3 Direct impact on transmitted message quantity

In the second evaluation we compare the communication resources needed, in terms of sent messages of a maximum of 128 bytes, to transmit a data unit in the original data format and using CTC. As in the previous section (Section 7.3.2), we considered two CoAP block sizes: 64 and 256. The data model instances used in the tests were previously shown in this section, in Figure 7.6, Figure 7.7 and Figure 7.8.

Table 7.8 contains the message quantity (and bytes) sent in order to transmit a data model instance in the original and CTC formats. The *Size* column shows the size in bytes of the data unit in the original data format (column *Orig.*), coded in CTC (column *CTC*) and proportional size between CTC and original formats. The message quantity used to

Table 7.8: Messages per data unit transmission.

Instance	Size			Messages / Bytes					
				Block:64			Block:256		
	Orig.	CTC	%	Orig.	CTC	%	Orig.	CTC	%
OPC-UA	936	73	7.8	26 / 2038	4 / 291	15 / 14	14 / 1408	2 / 201	14 / 14
SEP2	92	19	20.7	4 / 305	2 / 147	50 / 48	3 / 247	2 / 147	67 / 60
senML	219	60	27.4	8 / 597	2 / 188	25 / 31	4 / 387	2 / 188	50 / 49

transmit each data model instance is shown for the two CoAP block size configuration cases, i.e., 64 and 256 bytes. The results for these configurations are respectively contained within columns *Block:64* and *Block:256*, with the total number of transmitted bytes separated by a "/". For each block configuration, Table 7.8 includes the message and byte quantities for the original format (column *Orig.*), coded in CTC (column *CTC*) and relative between the original and CTC format (column *%*).

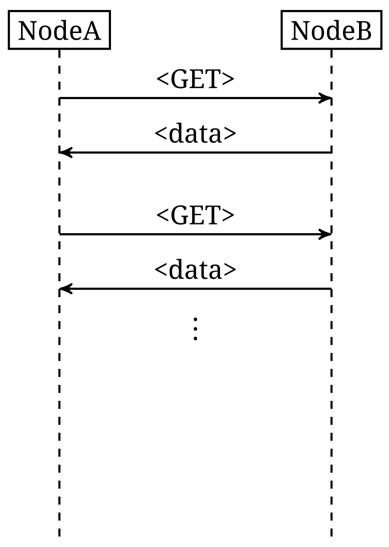
For example, the XML document instance "OPC-UA" has an original size of 936 bytes that is reduced to 73 bytes after being compressed with CTC. In the CoAP block configuration of 64 bytes, 26 messages or 2038 bytes need to be transmitted to deliver the original uncompressed document while the compressed data only requires 4 messages or 291 bytes to be transmitted, which amounts to the 15% of messages of the original document. In the case of the CoAP block configuration of 256 bytes, 14 messages or 1408 bytes are needed to transmit the original document. In contrast, 2 messages or 201 bytes suffice to send the compressed document which is the 14% of messages needed to transmit the original document.

As can be seen, the number of required message transmissions is significantly reduced for the CTC case. For a CoAP block size of 64, the number of messages sent in the case of CTC compared to the original data format ranges from 15% to 50%. When the CoAP block size is 256, the number of messages sent ranges from 14% to 67%. The results show that, regardless of data fragmentation, using CTC yields significant better use of the communication channels due to the reduction in transmitted messages and bytes for the same data unit.

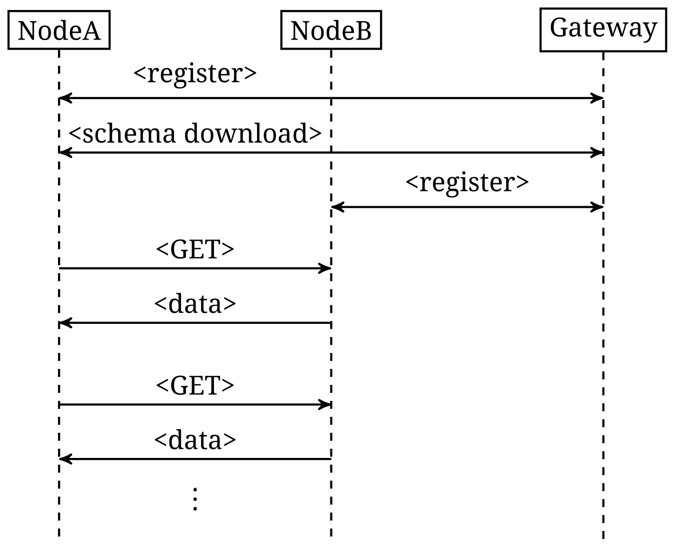
7.3.4 Long-term impact on transmitted message quantity

For the third evaluation we compare the message quantity needed to transmit increasing numbers of data units. The purpose of this test is to show the benefits of using CTC for a number of data unit transmissions, even taking into account the overhead produced by the schema register and schema download processes. We also discuss here the benefits and drawbacks of storing the schema in the node, depending on the schema size.

As has been explained before, two nodes are deployed in a local network connected to a gateway. NodeA produces a new data unit periodically and sends it to NodeB. All



(a) Original Format



(b) CTC Format

Figure 7.16: Logical sequence of the schema register, schema download and data transmission processes followed in the two cases: a) the original format and b) the CTC format.

the elements have a direct communication link and no multi-hop communications are considered. In the cases where data is compressed with CTC, the schema register process takes place before any data transmission is performed. Additionally, for the CTC case where the schemas are downloaded directly from the node, the download process takes place just after the schema register process, before any data transmission takes place. Figure 7.16 shows the logical sequence followed in the original data format (Figure 7.16a) and CTC (Figure 7.16b) cases.

Table 7.9 presents the accumulated number of messages as consecutive data units are incrementally sent. Different quantities of data unit transmissions are considered to show the benefits of CTC as the number of transmitted data units increases. The table shows the accumulated number of messages per data unit for the original and CTC formats. The quantity of messages considered for each row is indicated in the column *Data Units Sent*. Each data model column is further divided into three columns which respectively show the number of messages needed to transmit the data units in the original format (column *Orig*), in the CTC format (column *CTC*) and the proportional number of messages in the CTC case compared to the original format (column %). For the original data format cases, only the transmission of the data model instance is accounted for (see Table 7.8). In the CTC cases, the messages needed for the registration and schema download processes (see Table 7.7) are also taken into account. The top and bottom halves of the table respectively show the results for a CoAP block size of 64 bytes and 256 bytes.

As an example, lets consider the *OPC-UA* column and the row labelled as “10” of the *Block:64* case. 260 messages are needed to transmit 10 data units in the original data format as opposed to the 5400 messages required for the data units sent in the CTC format. Thus, almost 21 times more messages are sent in the CTC case compared to the original

Table 7.9: Accumulated messages transmitted including schema download process.

Data Units Sent	OPC-UA			SEP2			senML		
	Orig	CTC	%	Orig	CTC	%	Orig	CTC	%
Block: 64									
10	260	5400	2077	40	9030	22575	80	66	83
100	2600	5760	222	400	9210	2303	800	246	31
1000	26000	9360	36	4000	11010	275	8000	2046	26
10000	260000	45360	17	40000	29010	73	80000	20046	25
100000	2600000	405360	16	400000	209010	52	800000	200046	25
Block: 256									
10	140	2704	1931	30	4529	15097	40	46	115
100	1400	2884	206	300	4709	1570	400	226	57
1000	14000	4684	33	3000	6509	217	4000	2026	51
10000	140000	22684	16	30000	24509	82	40000	20026	50
100000	1400000	202684	14	300000	204509	68	400000	200026	50

case. The main reason lies in the CTC need to perform preliminary processes, i.e., schema register and schema download processes, before being able to transmit compressed data.

However, the tendency changes as the number of sent data units increases. For instance, 26000 messages are needed to transmit 1000 *OPC-UA* data units in the original format. In the CTC case, the message quantity is reduced to 9360 which is a 36% of the messages needed in the original format.

As can be seen in Table 7.9, the proportional number of messages needed to transmit the data units is reduced after a number of data units is sent. This means that the message quantity reduction only pays of once the number of saved messages is greater than the combined messages needed for the schema registration and schema download processes. For example, in the case of the SEP2 data model instance and with a CoAP block size of 64, this only happens when the number of transmitted data units is greater than 4505 (between upper rows “1000” and “10000” in Table 7.9), while for the OPC-UA case 244 transmitted data units are necessary (between upper rows “100” and “1000” in Table 7.9). In contrast, for the senML case only 8 data unit transmissions suffice to benefit from CTC, i.e., with 10 data units sent the message quantity is already reduced to a 83%, as can be seen in the row “10” of Table 7.9.

As can be deduced from the results shown in Table 7.7 and Table 7.9, the size of the schema is the factor with more weight for determining the message reduction efficiency. This is due to the fact that the schema download process has a much bigger impact than the schema registration process (as shown in Section 7.3.2).

Table 7.10 shows the accumulated message savings for a CoAP block size of 64 and 256 bytes, without taking into account the schema download process, i.e., it is assumed that the schemas are not downloaded directly from the nodes. The table follows the same structure as Table 7.9.

If we consider again the *OPC-UA* column and the row labelled as “10” of the *Block:64* case, we will see that the number of messages needed to transmit 10 data units in the

Table 7.10: Accumulated messages transmitted not taking into account schema download process.

Data Units Sent	OPC-UA			SEP2			senML		
	Orig	CTC	%	Orig	CTC	%	Orig	CTC	%
Block: 64									
10	260	58	22	40	36	90	80	36	45
100	2600	418	16	400	216	54	800	216	27
1000	26000	4018	15	4000	2016	50	8000	2016	25
Block: 256									
10	140	33	24	30	31	103	40	31	78
100	1400	213	15	300	211	70	400	211	53
1000	14000	2013	14	3000	2011	67	4000	2011	50

original data format is still the same: 260 messages. In contrast, only 58 messages are required for the CTC case which is a 22% of the messages needed in the original format.

As another example of the improvement, the SEP2 data model instance with a CoAP block size of 64, shows a better performance for the CTC case after only 8 data unit transmissions, which is a significant improvement to the results shown in Table 7.9 where 4505 data unit transmissions were necessary.

Thus, it is clear that for the OPC-UA and SEP2 cases, where the schema size is considerable, the overhead of downloading the schema is very high and it takes longer to compensate it. In this cases, it is more reasonable to avoid the overhead produced by the schema download by storing and downloading the schemas from an external server.

Although this test only takes into account one hop transmissions, the number of transmitted messages impact on multi-hop topologies will be even bigger. The benefits of reducing the number of messages sent by a node will be extended to relaying nodes as the number of retransmissions will be reduced proportionally, saving resources (such as energy and bandwidth) not only in the origin node, but also in the adjacent ones.

7.4 Summary and Conclusions

This chapter showed the empirical evaluation (including results) of the work performed during the Thesis. The evaluation covered the performance tests of a prototype implementation of CTC for the XML and JSON data formats compared to other standard compression technologies and encodings targeted to resource-constrained devices. This chapter also included the evaluation of the impact of CTC on a typical REST architecture deployment following the CTC communication model.

First, we presented the performance of CTC for the XML data format (Section 7.1). The results show that CTC provides good performance results compared to EXI implementations. CTC achieves better performance than EXI implementations in terms of processing time and memory usage, while keeping a similar efficiency in terms of compression for

EXI's ideal case. These results support CTC as a good candidate for resource-constrained devices as it produces very efficient implementations in terms of memory usage and energy consumption.

Although the *CTC Library* used in Section 7.1 is still a prototype, results show that CTC memory requirements are much more suited to resource-constrained devices than EXI implementations due to the significantly smaller memory footprint and runtime usage requirements. The modular approach of the *CTC Library* allows to tailor the capabilities to the needs and resources of the devices. This is important because resource-constrained devices need to share the limited memory between multiple functionalities (application, sensor drivers, communication stacks, etc.) and assigning memory resources as efficiently as possible is key in order to make the most of the available memory.

In Section 7.2 we extended the evaluation performed in Section 7.1 to the JSON data format as well as compare the XML and JSON cases. Section 7.2 showed that CTC provides good performance results compared to EXI4JSON and CBOR implementations. These results demonstrate that CTC is also suitable for both XML and JSON Schemas, and that CTC can handle seamless transformation to various data model representation formats in a resource efficient way. Thus, CTC is a good candidate for generic data model representation in resource-constrained devices.

The final section (Section 7.3) showed the impact of CTC on the communication load when it is integrated in a typical REST architecture deployment. The results demonstrate the positive impact and the reduction on transmitted messages when CTC is used. The evaluation also showed the ability of the CTC communication model mechanisms to adapt to different configurations and needs.

In summary, the evaluation and results included in this chapter demonstrate that CTC is suitable for generic data model representation in resource-constrained devices because it provides an encoding format that produces efficient implementations in terms of processing time, memory usage and compression ratio, while enabling interoperable applications.

8 Conclusions

In this Thesis, we presented *Context- and Template-based Compression* (CTC), a compression approach for structured data. CTC provides a more efficient encoding than text-based data formats while being especially suited to resource-constrained devices and networks. Although CTC is an alternative to text-based data formats, it keeps backwards compatibility, resulting in an practical solution for addressing IoT interoperability at data representation level.

The number of devices targeted at IoT domains is rapidly growing. These devices are highly-heterogeneous and tailored to the needs of the (also heterogeneous) IoT applications. At data representation level, the diversity in semantics and structures hinder the efforts to provide seamless and interoperable data processing solutions. This diversity results in very costly and complex procedures for the connection and integration of services, interfaces and data as well as set a barrier to overcome for today's IoT systems to enable the global and seamless connection of all the existing and forthcoming technologies. This is the reason why universal standards and mechanisms are essential for removing the interoperability gap across the IoT ecosystem.

Various initiatives are working in increasing the IoT interoperability, such as Web of Things and Open Connectivity Foundation communities. These initiatives are based on the promotion of platform-independent standards (either already available or specifically developed) to be used by IoT devices and systems. However, this approach requires for the standards to be implemented either in the lower architectural layers (as close to the device as possible) or in intermediary layers (such as gateways or middle-wares). In the former case, devices may not have enough resources or capabilities to natively implement the standards while, in the latter case, the deployment of specific purpose intermediaries only moves the interoperability-related complexity, it does not remove or reduce it.

At data level, interoperability is achieved by structuring the data following a well-defined data model and data format. Text-based data formats, such as XML and JSON, have been one of the key interoperability enablers across the Internet and they are the basis for other high-level technologies such as Web Services. It is expected that text-based data formats will also play an important role in the IoT integration. However, text-based data formats are hardly suitable for the resource-constrained devices typically deployed in IoT networks.

CTC addresses all these problems by easing the interoperable integration of data represented in text-based data formats while requiring very few resources regarding processing power, memory size and communication bandwidth. Compressing data using a more efficient encoding is a common approach to deal with the verbosity of text-based data formats. However, compressing the data may raise some drawbacks. On the one hand, the compression process produces an overhead that may be beyond the capabilities of resource-constrained devices or cancel the benefits of the size reduction. CTC provides a structured data representation encoding targeted at resource-constrained devices and networks. On the other hand, changing the data format to another encoding may break interoperability if not handled properly. CTC is more efficient than standard data formats and allows seamless transformation between the CTC format and the original format while keeping backwards interoperability.

This Thesis also describes complementary technologies build around CTC. Firstly, the CTC communication model (Chapter 5) specifies the communication architectures CTC is targeted at, as well as the *schema registration* and *schema repository* mechanisms, which aim to provide flexibility and interoperability by dynamically assigning and distributing schema information at running time. Secondly, the *CTC Library* (Chapter 6) describes the modular approach that allows tailoring the CTC capabilities to the needs and resources of the devices. Finally, the *CTC Compiler* tool (Section 6.2) is designed to ease and automate the implementation of the *context table*, *template table* and data model bindings to native code. All these complementary technologies facilitate the integration of CTC in general IoT deployments.

8.1 Summary of the contributions

In this work, we have shown that CTC provides good performance results for XML and JSON data formats, compared with current compression technologies. For the XML case (Section 7.1), CTC achieves better performance than EXI implementations in terms of processing time and memory usage, while keeping a similar efficiency in terms of compression for EXI's better case (*Schema Strict*). In the JSON case (Section 7.2), we have shown that CTC outperforms EXI4JSON and CBOR implementations in terms of compression size by taking advantage of the JSON Schema information.

These results show that CTC is in-line with the objectives described in Chapter 1.2 and that it can handle seamless compression of various data formats in an efficient way and suitable for resource-constrained devices and networks.

The solution offered by CTC is agnostic of any specific data format and are able to rebuild the original data format based on the information extracted from the schema and stored in the *context table* and *template table*, thus, meeting **Objective-1**. The *context table* and *template table* themselves fulfil **Objective-2** by providing a structure description mechanism for generic data models and text-based data formats. Furthermore, CTC also meets the **Objective-3** because it offers a solution for structured data compression suitable for resource-constrained devices and networks, and efficient regarding processing time, memory size and transmission bandwidth as shown in Chapter 7.

The last objective, **Objective-4**, is addressed in the CTC communication model. The CTC communication model provides flexible and interoperable mechanisms for typical IoT communication architectures. In the proposed CTC communication model, devices using standard data representation formats can coexist and communicate with devices using compressed formats whether they reside in the same (sub-)network or not. The communication can be end-to-end if both devices implement CTC or a CTC gateway can be deployed to seamlessly translate between the compressed format and the original data format.

The *schema registration* and *schema repository* mechanisms defined within the CTC communication model enable the dynamic assignment and distribution of schema information at run time. Following the same approach as CTC, these mechanisms are very flexible and tailored to the multiple restrictions of resource-constrained devices and networks. In this Thesis we use CTC as the compression solution for the *schema registration* and *schema repository* mechanisms. A specific underlying binding protocol, CoAP, is also used as an illustrative and relevant example. However, the proposed CTC communication model is generic enough to be applied to other structured data compression approaches based on contextual information (such as EXI) or binding protocols (such as MQTT).

In Section 7.3 we showed the positive impact and the reduction on the quantity of exchanged messages when the CTC communication model is used. The section also shows that different configurations meet the restrictions of the available resources and application needs.

In summary, all the objectives of this work have been successfully met and has been proven that the hypothesis this Thesis is based on (described in Chapter 1.2) holds. CTC is a good candidate for generic structured data representation targeted to resource-constrained devices and networks as it produces very efficient implementations in terms of memory usage and energy consumption while maintaining interoperability with the original data format.

Additionally, the modular approach followed by the *CTC Library* allows to tailor the CTC capabilities to the needs of the application and further optimize the used resources. This is complemented by the *CTC Compiler* tool, which eases the adoption of CTC and its integration on IoT application developments.

8.2 Future Work

In this work, a preliminary implementation of the CTC Library has been developed. This prototype will be further developed to improve and to fully implement the features presented in this document. For instance, the parsing of the formatted data is based on a straightforward algorithm for text search on a list of elements, i.e., the *template table*. Further research on text search and text matching would improve this process. As another example, the CTC *Compiler* would need further development to produce the full set of data binding code stubs.

As a future work, we are planning on extending the CTC mapping to additional data model representation formats as well as define further bindings of the *schema registration* mechanisms to other typical IoT communication protocols. Specifically, it is of great interest to extend the *CTC Compiler* capabilities to process data model based protocols, such as SOAP, in order to support the automatic generation of Web Service bindings. The extension of CTC to more technologies would raise its usability for more IoT scenarios and make CTC more appealing to IoT system developers and integrators.

Another interesting improvement of CTC would be to include mechanisms to take into account constraints described in the schema, e.g., the maximum value of a number. Leveraging the constrains would improve the compression rate by producing more compact representations of the data as well as enable partial validations to the coded streams

We are also exploring the possibility of embedding the data model related information stored in the *schema context* directly in the code stubs generated by the *CTC Compiler*, in contrast to keeping the information in a separated and dedicated structure. The purpose of the research is to assess if such a change would bring improvements regarding memory usage and processing performance, while keeping software modularity.

Finally, we are considering another line of research focused on the application of approaches used in CTC to EXI. Some of the mechanisms designed for CTC could be applied to EXI grammar implementations in order to enhance its efficiency, from in-memory representation to grammar processing. By improving the implementation efficiency of EXI processors, the use of EXI would open to a wider range of resource-constrained devices.

A Data formats: technical aspects

This appendix contains further details of the XML Schema and JSON Schema specifications. This information is complementary to the descriptions found in Section 3.1 but it is not mandatory by any means to follow the work described in this document. However, this information has been gathered here for convenience as it may help in understanding the mapping processes described in Section 4.1.4 as well as some of the design decisions behind the *Context Table* and *Template Table* specifications.

A.1 XML Schema

This section gives a more detailed overview of the XML Schema specification [WWWCd]. The XML Schema specification describes the structure and vocabulary of XML documents.

Usually, an XML document includes a reference to the XML Schema that describes its vocabulary and the XML document is denoted an “instance” of the schema document. The main use for XML Schemas is for document validation, which consist on verifying that the content of an XML document is in conformance with the model and structure described in the associated schema.

The following subsections describe the most important concepts and components of the XML Schema specification relevant to this thesis. However, these subsections do not aim to provide exhaustive information but to give enough information to understand the principles proposed in this thesis.

A.1.1 Basic Structure

The *schema* element is the root element of any XML Schema document. This element provides several attributes to declare information about the overall schema including the namespace it is bound to and version information.

The general format of a *schema* element starting tag is shown in Figure A.1. The *targetNamespace* attribute is used to declare the namespace URI that will identify this XML Schema. Other XML documents will use this URI to reference the schema as a namespace. The *attributeFormDefault* and *elementFormDefault* attributes specify how

```
<schema
    targetNamespace="URI"
    attributeFormDefault="qualified | unqualified"
    elementFormDefault="qualified | unqualified"
    version="version_number">
```

Figure A.1: Schema element general format.

```
<schema
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:pe="http://example.com/namespaces/pets"
        targetNamespace="http://example.com/namespaces/pets"
        elementFormDefault="qualified">
    <element name="pet">
        <complexType>
            <sequence>
                <element name="name" type="string"/>
                <element name="age" type="integer"/>
                <element name="gender" type="string"/>
            </sequence>
            <attribute name="species" type="string"/>
        </complexType>
    </element>
</schema>
```

Figure A.2: XML Schema example.

elements and attributes should be qualified in the instance documents. An element or attribute is qualified if it is associated to a namespace, as explained in section 3.1.1.2.

As has been already said, XML elements are one of the basic building blocks of an XML document. To declare an element three main properties must be specified: the name, the type and the cardinality. The XML Schema Recommendation specifies the use of two kinds of data types: built-in data types and user-defined data types. User-defined data types are specified in the the schema through the *simpleType* and *complexType* declarations. Built-in data types, on the other hand, are simple data types already defined in the XML Schema namespace and they are available to be reused for schema definitions.

Attribute declarations are very similar to element declarations. However, attribute declarations can only be of simple types.

Figure A.2 shows an example of a simple XML Schema, containing four elements and one attribute. “pet” is the root element and it contains the child elements “name”, “age” and “gender” as well as an attribute of the “pet” element named “species”.

A.1.2 Built-in Data Types

The following list summarizes the most common data types defined in the XML Schema namespace:

- **string:** a chain of characters.

- **Name:** a string that contains a valid XML name.
- **QName:** a string that contans a qualified XML name.
- **anyURI:** an Uniform Resource Identifier (URI).
- **byte:** a numeric value in the range [-128, 127].
- **unsignedByte:** a numeric value in the range [0, 255].
- **hexBinary:** binary information encoded in hexadecimal.
- **base64Binary:** binary information encoded in Base64.
- **integer:** a whole number (no fractional part).
- **positiveInteger:** an integer greater than 0.
- **negativeInteger:** an integer lower than 0.
- **nonNegativeInteger:** an integer greater or equal to 0.
- **nonPositiveInteger:** an integer lower or equal to 0.
- **int:** a signed 32-bit integer.
- **unsignedInt:** an unsigned 32-bit integer.
- **long:** a signed 64-bit integer.
- **unsignedLong:** an unsigned 64-bit integer.
- **short:** a signed 16-bit integer.
- **unsignedShort:** an unsigned 16-bit integer.
- **decimal:** a decimal value. It may or may not include a fractional part.
- **float:** a IEEE single-precision 32-bit floating-point value. INF and NaN values are accepted.
- **double:** a IEEE double-precision 64-bit floating-point value. INF and NaN values are accepted.
- **boolean:** a logical value. *true*, *false*, 0, and 1 values are accepted.
- **time:** a time value in the format "*hour*:*minutes*:*seconds*".
- **dateTime:** a date and time value in the format "*year-month-day*T*hour*:*minutes*:*seconds*".
- **date:** a date value in the format "*year-month-day*".

```
<complexType name="id_or_info_type">
    <choice>
        <element name="identifier" type="string"/>
        <sequence>
            <element name="name" type="string"/>
            <element name="age" type="integer"/>
            <element name="gender" type="string"/>
        </sequence>
    </choice>
</complexType>
```

Figure A.3: XML Schema nested content models example.

A.1.3 Complex Data Types

Complex data types are used to create user-defined XML elements that contain other elements and/or attributes. Complex data types are declared with the *complexType* declaration and define the content model of the complex element. In the context of XML Schemas, a content model specifies how elements are grouped together. The XML Schema Recommendation defines three content models:

- **sequence:** elements must appear in the order defined in the schema.
- **choice:** only one of the elements defined in the schema may appear.
- **all:** elements may appear in any order, and any of them may be omitted.

sequence and *choice* content models can be used within other content models and not just individual elements. For instance, Figure A.3 shows an example of a *choice* content model containing an element and a *sequence* content model. Specifically, the example declares that a “id_or_info_type” type element is either an “identifier” element or an ordered list of “name”, “age” and “gender” elements.

The *all* content model is used when the elements are known, but not the order. However, there are some restrictions on the use of the *all* content model. On one hand, the *all* declaration must be the only content model of a *complexType* element definition and it cannot contain *sequence* or *choice* declarations, only elements. On the other hand, children of an *all* declaration may only appear once in the instance document, i.e. the cardinality of the children is bounded to 0 or 1.

Complex data types can be extended by deriving a new data type from a base or original data type. Complex data types can be extended either through restriction or extension.

When deriving a complex data type through extension, the content model of the new data type is the combination of the content model of the original data type and the content model specified in the derived data type. In the example shown in Figure A.4 a new

```
<complexType name="pet">
    <sequence>
        <element name="name" type="string"/>
        <element name="age" type="integer"/>
        <element name="gender" type="string"/>
    </sequence>
    <attribute name="species" type="string"/>
</complexType>

<complexType name="pet_ext">
    <complexContent>
        <extension base="pet">
            <sequence>
                <element name="owner" type="string" />
            </sequence>
        </extension>
    </complexContent>
</complexType>
```

Figure A.4: XML Schema complex data type extension example.

```
<complexType name="pet">
    <sequence>
        <element name="name" type="string"/>
        <element name="age" type="integer"/>
        <element name="gender" type="string"/>
    </sequence>
    <attribute name="species" type="string"/>
</complexType>

<complexType name="pet_rest">
    <complexContent>
        <restriction base="pet">
            <sequence>
                <element name="name" type="string"/>
                <element name="age" type="integer"/>
            </sequence>
        </restriction>
    </complexContent>
</complexType>
```

Figure A.5: XML Schema complex data type restriction example.

complex type named “pet_ext” is created by adding a new element with the name “owner” to the user-defined data type “pet”.

When deriving complex types through restriction, the content model of the new data type is a subset of the original data type. The difference of extension by restriction applied to simple types and complex types is that in simple types a more restricted range of values is specified while in complex types the types content model declarations are restricted. In the example shown in Figure A.5 a new complex type named “pet_rest” is created by removing the element named “gender” from the user-defined data type “pet”.

A.1.4 Simple Data Types

The *simpleType* declaration allows to define simple data types based on already defined data types, which may be built-in or custom data types. As simple types are always

```
<attribute name="species">
    <simpleType>
        <restriction base="string">
            <enumeration value="Cat"/>
            <enumeration value="Dog"/>
            <enumeration value="Tortoise"/>
            <enumeration value="Rabbit"/>
        </restriction>
    </simpleType>
</attribute>
```

Figure A.6: XML Schema restriction declaration example.

derived from other types, they are also known as *derived types*. There are three kinds of derived types: restriction, list and union.

A restriction is a simple type that defines a subset of the type it is derived from. To specify the type from which the simple type is derived, the *base* attribute is used. The subset of the base type can be bounded using a series of elements defined within the *simpleType* declaration known as facets. For instance, *totalDigits* facet specifies the number of digits of a numeric type, *minLength* facet sets the minimum number of characters in a string type and *pattern* facet allows to define regular expressions that string types must follow. Not all the facets have to be used at the same time and each type only supports a subset of the facets.

Figure A.6 shows an example of a restricted attribute "species" derived from the built-in type "string" and that only allows the use of four values: "Cat", "Dog", "Tortoise" and "Rabbit". In this case, the *enumeration* facet is used, which specifies allowed values through an enumerated list.

The *list* declaration allows to define a list of items in which each item is a *simpleType*. The base simple type can be a global simple data type (including built-in data types) or a locally defined simple type.

In the example shown in Figure A.7 a global simple type is defined,"species_type", using the restriction declaration and the enumeration facet. This global simple type is then used as an item of the "species_list_type" derived type.

union declarations enable the combination of multiple simple types. The simple types to be combined are listed in the *memberTypes* attribute. The types listed can be a global simple data type (including built-in data types) or a locally defined *simpleType*.

The Figure A.8 shows an example where we define a simple type named "unknown_or_age_type" from the union of the "integer" built-in data type and a globally defined data type, "string_unknown_type". The "unknown_or_age_type" data type will allow float values as well as the string "Unknown".

```
<simpleType name="species_type">
    <restriction base="string">
        <enumeration value="Cat"/>
        <enumeration value="Dog"/>
        <enumeration value="Tortoise"/>
        <enumeration value="Rabbit"/>
    </restriction>
</simpleType>

<simpleType name="species_list_type">
    <list itemType="species_type"/>
</simpleType>
```

Figure A.7: XML Schema list declaration example.

```
<simpleType name="string_unknown_type">
    <restriction base="string">
        <enumeration value="Unknown"/>
    </restriction>
</simpleType>

<simpleType name="unknown_or_age_type">
    <union memberTypes="integer string_unknown_type"/>
</simpleType>
```

Figure A.8: XML Schema union declaration example.

A.1.5 Global and Local Declarations

User-defined data types can be declared globally or locally. To define a global element type, it must be declared as a direct child of the root element (i.e. the *schema* element) of the schema document. This makes the type available to be reused through the whole schema document. Local declarations are children of elements other than the root element and can be used only within the element in which it is declared or its child elements.

The main reason to declare global element types is re-usability. In the example of Figure A.9 we can see a globally declared element type "pet_type" which is then assigned to the type of the root element "pet". Actually, the elements "name", "age", "gender" and attribute "species" are also assigned to globally available types: "string" and "integer". These global types are part of the declaration of the XML Schema namespace (http://www.w3.org/2001/XMLSchema) which in the example appears included as the default namespace.

It is possible to reuse a whole global element and not just the type trough the *ref* attribute. In the example shown in Figure A.10 the elements contained within the type "pet_type" are declared using the references of globally declared elements.

To create a local type, the type has to be declared as a direct child of the corresponding element. Unlike global declarations, local *complexType* definitions are not named (i.e. do

```
<schema
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:pe="http://example.com/namespaces/pets"
        targetNamespace="http://example.com/namespaces/pets"
        elementFormDefault="qualified">
    <complexType name="pet_type">
        <sequence>
            <element name="name" type="string"/>
            <element name="age" type="integer"/>
            <element name="gender" type="string"/>
        </sequence>
        <attribute name="species" type="string"/>
    </complexType>
<element name="pet" type="pe:pet_type"/>
</schema>
```

Figure A.9: XML Schema global complex element declaration example.

```
<schema
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:pe="http://example.com/namespaces/pets"
        targetNamespace="http://example.com/namespaces/pets"
        elementFormDefault="qualified">
    <element name="name" type="string"/>
    <element name="age" type="integer"/>
    <element name="gender" type="string"/>
    <complexType name="pet_type">
        <sequence>
            <element ref="pe:name"/>
            <element ref="pe:age"/>
            <element ref="pe:gender""/>
        </sequence>
        <attribute name="species" type="string"/>
    </complexType>
    <element name="pet" type="pe:pet_type"/>
</schema>
```

Figure A.10: XML Schema global element declaration by reference example.

```
<element name="pet">
    <complexType>
        <sequence>
            <element name="name" type="string"/>
            <element name="age" type="integer"/>
            <element name="gender" type="string"/>
        </sequence>
        <attribute name="species" type="string"/>
    </complexType>
</element>
```

Figure A.11: XML Schema local element declaration example.

```
<element ref="owner" maxOccurs="5"/>

<element name="owner" type="string" minOccurs="2" maxOccurs="3"/>

<element name="owner" minOccurs="0" maxOccurs="unbounded"/>
```

Figure A.12: XML Schema element cardinality example.

not include a *name* attribute) and are known as *anonymous complex types*. In the example of Figure A.11 the element “pet” is bound to a local type with three child elements (“name”, “age” and “gender”) and an attribute “species”.

Attributes can also be created using local types or global types (including reference) in the same way as with elements.

A.1.6 Element Cardinality

The cardinality of an element specifies how many occurrences of the element can appear in the instance document. Cardinality is specified by declaring the *minOccurs* and *maxOccurs* attributes.

minOccurs and *maxOccurs* attributes are optional and they may be omitted. If any of them is not included, the default value in both cases is 1. The *maxOccurs* attribute allows the use of the value “unbounded” in order to indicate no upper limit to the number of occurrences.

In the first example shown in Figure A.12, the “owner” element is a global element and states that the element instance must appear at least one time and a maximum of 5 times. The second example declares an element named “owner” that must appear in the instance document a minimum of two times and a maximum of three. The last example declares an optional element (it may appear 0 times) but with no upper limit to the number of occurrences.

```
<complexType name="pet_type">
    <sequence>
        <element name="name" type="string"/>
        <element name="age" type="integer"/>
        <element name="gender" type="string"/>
        <any namespace="##any"
            processContents="lax"
            minOccurs="0"
            maxOccurs="unbounded"/>
    </sequence>
    <attribute name="species" type="string"/>
</complexType>
```

Figure A.13: XML Schema *any* declaration example.

A.1.7 The *any* Declaration

The *any* declaration is used to declare elements in a non explicit way. This type of element declarations are known as *element wildcards* and enable, for instance, to indicate that an element can be any element declared within a given namespace.

It is not possible to create *any* global declarations and they must always appear as children of content model declarations. *any* declarations are quite flexible and allow to specify the range of elements allowed through the *namespace* attribute. From instance, the “##any” value allows elements from all namespaces and the “##targetNamespace” value allows elements from only the targetNamespace to be included.

In the example in Figure A.13, we declare that the “pet_type” element type accepts elements from any namespace as its children. Thus, an instance document containing an element of “pet_type” type must include the “name”, “age” and “gender” elements and, optionally, any number of elements from any namespace.

A.1.8 Schema Reuse

The XML Schema Recommendation provides two declarations to reuse definitions made in multiple XML Schema documents: *import* and *include*. The *import* declaration allows to use global declarations from XML Schemas with a different *targetNamespace*. *import* declarations must be globally declared. On the other hand, the *include* declaration allows to combine XML Schemas that share the same *targetNamespace* or that do not have a *targetNamespace* at all.

In the example shown in Figure A.14 the “http://example.com/namespaces/info” schema is imported into the “http://example.com/namespaces/pets” schema with the prefix “in”. The child elements within the “pet_type” type are specified through a reference to global elements in the “in” schema. In contrast, the type of the “species” attribute is assigned to a global simple type declaration of the“in” schema.

```
<schema
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:in="http://example.com/namespaces/info"
        targetNamespace="http://example.com/namespaces/info"
        elementFormDefault="qualified">
    <element name="name" type="string"/>
    <element name="age" type="integer"/>
    <element name="gender" type="string"/>
    <simpleType name="species_type">
        <restriction base="string">
            <enumeration value="Cat"/>
            <enumeration value="Dog"/>
            <enumeration value="Tortoise"/>
            <enumeration value="Rabbit"/>
        </restriction>
    </simpleType>
</schema>



<schema
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:pe="http://example.com/namespaces/pets"
        xmlns:in="http://example.com/namespaces/info"
        targetNamespace="http://example.com/namespaces/pets"
        elementFormDefault="qualified">
    <import namespace="http://example.com/namespaces/info"/>
    <complexType name="pet_type">
        <sequence>
            <element ref="in:name"/>
            <element ref="in:age"/>
            <element ref="in:gender""/>
        </sequence>
        <attribute name="species" type="in:species_type"/>
    </complexType>
    <element name="pet" type="pe:pet_type"/>
</schema>
```

Figure A.14: XML Schema import declaration example.

```xml
<schema
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:pe="http://example.com/namespaces/pets"
        targetNamespace="http://example.com/namespaces/pets"
        elementFormDefault="qualified">
    <element name="name" type="string"/>
    <element name="age" type="integer"/>
    <element name="gender" type="string"/>
    <simpleType name="species_type">
        <restriction base="string">
            <enumeration value="Cat"/>
            <enumeration value="Dog"/>
            <enumeration value="Tortoise"/>
            <enumeration value="Rabbit"/>
        </restriction>
    </simpleType>
</schema>



<schema
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:pe="http://example.com/namespaces/pets"
        targetNamespace="http://example.com/namespaces/pets"
        elementFormDefault="qualified">
    <include schemaLocation="pets_info.xsd"/>
    <complexType name="pet_type">
        <sequence>
            <element ref="pe:name"/>
            <element ref="pe:age"/>
            <element ref="pe:gender"/>
        </sequence>
        <attribute name="species" type="pe:species_type"/>
    </complexType>
    <element name="pet" type="pe:pet_type"/>
</schema>
```

Figure A.15: XML Schema include declaration example.

Global declarations of schemas included through an *include* declaration are treated as if they were defined within the same schema. This is mainly useful when a complex vocabulary is being defined and it is practical to split it in multiple sections (denoted *modules*) across schema documents.

In the example shown in Figure A.15 the "http://example.com/namespaces/pets" is split into two modules. The definitions made in the first schema document are used in the second one but all the declarations are made within the same namespace and they are treated as if the would have been defined within the same document.

A.2 JSON Schema

In this section we describe in more detail the JSON Schema specification. This section covers the Draft-04 version (i.e.: "http://json-schema.org/draft-04/schema#") of the JSON Schema [GZC13] specification and JSON Schema Validation [ZC13] vocabulary. At the time of this writing Draft-04 version is still the more widely used JSON Schema version, compared to more recent ones (currently Draft-07 [WA18, WAL18]).

```
{
  "title": "root schema",
  "sub": {
    "title": "subschema"
  }
}
```

Figure A.16: JSON Schema root schema and subschema example.

The JSON Schema Draft-4 specification is actually composed by three documents.

- JSON Schema core specification [GZC13] describes the core terminology, references to other JSON Schemas and vocabulary definition.
- JSON Schema Validation [ZC13] defines the vocabulary for validation assertions, link navigation and interaction constrains.
- JSON Hyper-Schema specification [LZC13] describe the hypertext structure and management of JSON documents such as resource link relations and multimedia vocabulary.

A JSON Schema is in itself a JSON document and is composed by the same structural components used by JSON documents: null, boolean, number, string, object and array. This primitives are described in more detail in Section 3.1.2.

A JSON document or JSON component that follows the structure and constrains specified by a JSON Schema is denoted an “instance”.

A.2.1 JSON Schema Structure

A JSON Schema document always starts from the root schema but it can contain any number of nested schemas, denoted subschemas. For instance, Figure A.16 shows an example JSON Schema that is composed by a root schema (titled “root schema”) and one subschema (titled “subschema”).

The root schema and subschemas of a JSON document are either an object or a boolean. Schemas with boolean root elements are special schemas that either always pass validation of instances (“true”) or always fail (“false”). If the schema is an object, it contains the structure and constrains that must be followed by the JSON instances that follow the data model described in the schema. The properties of the JSON Schema contain the vocabulary of the data model and are refereed as “keywords”.

The “$schema”keyword accepts an URI value and it indicates the JSON Schema version this particular schema conforms to. For the JSON Schema Dratf-04 version the value “http://json-schema.org/draft-04/schema#” is predefined. At the time of this writing, the last version (Draft-07) is defined as “http://json-schema.org/draft-07/schema#”.

The “$ref” keyword is used to reference either an internal or external JSON schema. In the case of internal references, they can be used to include subschemas that are defined in another location of the JSON Schema document instead of directly embedding the schema itself. This is useful, for instance, to define a schema once and reference it from multiple locations within the JSON Schema document. External references are used to include schemas that are defined in another JSON Schema documents. In this case, it also allows the reuse of schema definitions across JSON Schema document boundaries. An important remark is that the URI value of a “$ref” keyword is not a network locator, but an identifier (described in the next paragraph).

The “$id” keyword is used to assign an explicit URI to an schema. This URI also defines the base URI for the subschemas of the schema. The identifiers of nested subschemas are resolved recursively against the base URI of the parent schema. Any schema can be referenced (with the “$ref” keyword) by any other schema within the same JSON Schema using the assigned “$id” identifier. External schemas (defined in another JSON Schema document) can be referenced by means of the “$id” URI value of the root schema.

A.2.2 JSON Schema Validation Keywords

The main purpose of the JSON Schema is to provide the means to specify the structure and restrictions applicable to a specific data model. This information can be used to validate JSON documents. The JSON Schema Validation specification [ZC13] defines the vocabulary and keywords to assert and validate JSON documents. The version of the JSON Schema and, hence, the used vocabulary, is specified by means of the “$schema” keyword. The following vocabulary corresponds to the JSON Schema Dratf-04 version, identified with the predefined “$schema” keyword value of “http://json-schema.org/draft-04/schema#”.

A.2.3 keywords for numbers

The keyword “multipleOf” is used together with JSON numbers to specify that a number must be a multiple of the specified value, i.e. the value of the instance number divided by the value of the “multipleOf” keyword must be an integer.

The value of the “maximum” keyword is a number that specifies the maximum value allowed for the instance. If the boolean keyword “exclusiveMaximum” is used together with the “maximum” keyword and its value is true, then the value of the instance must be lower than the “maximum” keyword value.

In a similar manner, the value of the “minimum” keyword is a number that specifies the minimum value allowed for the instance. If the boolean keyword “exclusiveMinimum” is used together with the “minimum” keyword and its value is true, then the value of the instance must be greater than the “minimum” keyword value.

A.2.4 keywords for strings

The values of the “maxLength” and “minLength” keywords are unsigned integers that respectively specify the maximum and minimum number of characters hold by a string instance.

The “pattern” keyword is used to define a string regular expression. A string instance is valid if it matches the regular expression.

A.2.5 keywords for arrays

The “items” keyword is used to specify the object schema that must be meet by all the elements of the instance array. Optionally the “items” keyword can hold an array, in which case, it specifies the schema for the instance array elements one by one. The “items” keyword can be followed by the “additionalItems” keyword. The “additionalItems” keyword is used to specify whether more elements than the ones defined in the “items” keyword are allowed, in case the value of the “items” keyword is an array.

The values of the “maxItems” and “minItems” keywords are unsigned integers that respectively specify the maximum and mimimum number of elements that can be held in an array instance.

The “uniqueItems” keyword is used to declare if the elements of the array must be different between them. If the value of the “uniqueItems” keyword is true, all the elements of the array must be unique.

A.2.6 keywords for objects

The values of the “maxProperties” and “minProperties” keywords are unsigned integers that respectively indicate the maximum and minimum number of properties that can be defined by an object instance.

The “required” keyword specifies an array of strings that must match the names of the properties defined in the object.

The “properties” keyword determines the structure of the object instance and its children objects. The value of the “properties” keyword is an object in itself and each value of the object contains a JSON Schema identified by the keyword of the value. Instances of the object have to match the child names and corresponding schemas.

The “patternProperties” keyword is very similar to “properties”, the only difference is that instead of validating against keywords in the object that perfectly match the name of the instance child, it validates against the regular expression and schema of the keywords.

The “additionalProperties” keyword is used in conjunction with “patternProperties” and “properties”. The “additionalProperties” keyword specifies the schema of the object instance's children that are not covered by the "patternProperties" and "properties" keywords.

The "dependencies" keyword defines validation rules that must be met by an object instance in case a specific child match one of the keywords defined within the dependency. The value of the "dependencies" keyword is and object in which each child can be either an object or an array of unique strings. In case the child is an object, if the object instance has a property that matches the keyword, then the object instance must validate against the schema of the dependency. If the child is an array and the object instance has a property that matches the keyword, then the object instance must also contain properties that match the strings in the array.

A.2.7 keywords for any instance type

The "enum" keyword defines an array of unique elements. An instance is valid if its value matches one of the elements listed in the "enum" keyword array.

The value of "type" keyword is either a string or an array of unique strings. The strings' value must be one of the primitive types defined in the JSON Schema specification and described in Section 3.1.2. The instance validates successfully if the type matches one the strings of the "type" keyword.

The "allOf" keyword value is an array of objects. A valid instance must match the schemas of all the objects within the "allOf" keyword. The value of the "anyOf" keyword is also an array of objects. However, in this case an instance successfully validates if it matches at least the schema of one of the objects within the "anyOf" keyword. In a similar way, the value of the "oneOf" keyword is also an array of objects. However, an instance successfully validates if it matches only one of the schemas of the objects within the "oneOf" keyword. The "not" keyword value is an object for which valid instances must fail to validate.

Finally, the "definitions' keyword is used to define a standardized area to locate schemas, instead of directly nesting them in the root schema. This is useful, for example, to define schemas that are referenced from more than one parent schema.

A.2.8 Metadata keywords

The "title" and "description" keywords are not used to validate instances. However, they are useful to annotate schemas and subschemas and give contextual information to the schema readers or to visualization tools.

The "default" keyword is used to define the default value of an schema, in case one is not provided in an instance.

The “format” keyword is used to provide semantic validation of an instance. The value of the “format” keyword is a string that contains the identifier of a “format attribute”. The JSON Schema validation specification predefines a set of format attributes such as “date-time”, “email”, “hostname”, “ipv4”, “ipv6” and “uri”. Valid instances must match the restrictions defined in the schema as well as meet the restrictions associated to the format attribute.

Bibliography

[12a] W3C Simple Object Access Protocol (SOAP). http://www.w3.org/TR/soap, 2012. Last visited on October of 2018.

[12b] Ws4d-udpws - the devices profile for web services (dpws) for highly resource-constrained devices. https://gitlab.amd.e-technik.uni-rostock.de/michael.rethfeldt/ws4d-udpws, 2012. Last visited on October of 2018.

[16a] Embeddable EXI Processor in C. http://exip.sourceforge.net, 2016. Last visited on October of 2018.

[16b] EXIficient EXI procesor, java version. http://exificient.github.io/java/, 2016. Last visited on October of 2018.

[16c] An introduction to Cooja. https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja, 2016. Last visited on April 2018.

[17] Jabber.org community. https://www.jabber.org, 2017.

[18a] Binary JSON (BSON). http://bsonspec.org/, 2018. Last visited on July of 2018.

[18b] CBOR diagnostic utilities. https://github.com/cabo/cbor-diag, 2018. Last visited on March of 2018.

[18c] Contiki: The Open Source OS for the Internet of Things. http://www.contiki-os.org, 2018. Last visited on April 2018.

[18d] EXI for JSON - How EXI can be used to represent JSON data efficiently. https://github.com/EXIficient/exificient-for-json, 2018. Last visited on April of 2018.

[18e] Protocol Buffers. https://developers.google.com/protocol-buffers/, 2018. Last visited on March of 2018.

[ABMP07] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. Xquec: A query-conscious compressed XML database. *ACM Trans. Internet Techn.*, 7(2):10, 2007.

[ACM+15] Diego Arroyuelo, Francisco Claude, Sebastian Maneth, Veli Mäkinen, Gonzalo Navarro, Kim Nguyen, Jouni Sirén, and Niko Välimäki. Fast in-memory xpath search using compressed indexes. *Softw., Pract. Exper.*, 45(3):399–434, 2015.

[AGGT10] H. Abangar, M. Ghader, A. Gluhak, and R. Tafazolli. Improving the performance of web services in wireless sensor networks. In *Future Network and Mobile Summit, 2010*, pages 1–8, 2010.

[ANdlF03] Joaquín Adiego, Gonzalo Navarro, and Pablo de la Fuente. SCM: Structural Contexts Model for Improving Compression in Semistructured Text Databases . Technical Report IT-DI-2003-0004, Dep. Informatica, Universidad de Valladolid, 2003.

[BCN14] Nieves R. Brisaboa, Ana Cerdeira-Pena, and Gonzalo Navarro. XXS: efficient xpath evaluation on compressed XML documents. *ACM Trans. Inf. Syst.*, 32(3):13:1–13:37, 2014.

[BDD+16] H. Butler, M. Daly, A. Doyle, Sean Gillies, T. Schaub, and T. Schaub. The GeoJSON format. RFC 7946, August 2016.

[Ber17] Olaf Bergmann. libcoap: C-implementation of CoAP. https://libcoap.net, 2017. Last visited on April 2018.

[BG14] Andrew Banks and Rahul Gupta. MQTT version 3.1.1. Technical report, OASIS, 10 2014. OASIS Standard.

[BGC17] Jorge Berzosa, Luis Gardeazabal, and Roberto Cortiñas. Context- and Template-Based Compression for Efficient Management of Data Models in Resource-Constrained Systems. *Sensors*, 17(8):1755, 2017.

[BGC18] Jorge Berzosa, Luis Gardeazabal, and Roberto Cortiñas. A Communication Model for Efficient Management of Standard Data Formats in Resource-Constrained IoT Networks. *Computer Standards & Interfaces*, 2018. Submitted.

[BH13] Carsten Bormann and Paul E. Hoffman. Concise binary object representation (CBOR). RFC 7049, October 2013.

[BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. Uniform Resource Identifier (URI): Generic Syntax. https://rfc-editor.org/rfc/rfc3986.txt, January 2005. RFC 3986.

[BMKR15] Jiva N. Bagale, John P. T. Moore, Antonio D. Kheirkhahzadeh, and Yasmine Z. Rosunally. Energy consumption trade-offs for XML compression on embedded devices. In *2015 Sustainable Internet and ICT for Sustainability, SustainIT*

2015, Madrid, Spain, April 14-15, 2015, pages 1–3. IEEE Computer Society, 2015.

[Bor12] Carsten Bormann. Using CoAP with IPsec. Internet-Draft draft-bormann-core-ipsec-for-coap-00, Internet Engineering Task Force, December 2012. Work in Progress.

[Bou09] C. Bournez. Efficient XML Interchange Evaluation. Technical report, W3C, 4 2009. Last visited on October of 2018.

[BPSM$^+$08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Technical report, W3C, 11 2008. Last visited on June of 2018.

[Bra14] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, March 2014.

[BS16] Carsten Bormann and Zach Shelby. Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959, August 2016.

[BSE12] Arne Bröring, Christoph Stasch, and Johannes Echterhoff. OGC® Sensor Observation Service Interface Standard. http://www.opengeospatial.org/standards/sos, 2012. Last visited on June of 2018.

[BZB$^+$08] A. Bobek, E. Zeeb, H. Bohn, F. Golatowski, and D. Timmermann. Device and service templates for the devices profile for web services. In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 797–801, 2008.

[CC2] CC2650, simplelink multi-standard 2.4 ghz ultra-low power wireless MCU. http://www.ti.com/product/CC2650. Last visited on October of 2018.

[CCK$^+$06] Shannon Chan, Dan Conti, Chris Kaler, Thomas Kuehnel, Alain Regnier, Bryan Roe, Dale Sather, Jeffrey Schlimmer, Hitoshi Sekine, Jorgen Thelin, Doug Walter, Jack Weast, Dave Whitehead, Don Wright, and Yevgeniy Yarmosh. Devices profile for web services specification. http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf, February 2006.

[Che00] James Cheney. XMLPPM: XML-Conscious PPM Compression. http://xmlppm.sourceforge.net/, 2000. Last visited on July of 2018.

[Che05a] James Cheney. DTDPPM: DTD-Conscious Compression. http://xmlppm.sourceforge.net/dtdppm/index.html, 2005. Last visited on July of 2018.

[Che05b] James Cheney. An empirical evaluation of simple DTD-conscious compression techniques. In *Eighth International Workshop on the Web and Databases (WebDB)*, pages 43–48. Citeseer, 2005.

[CK13a] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. RFC 6763 (Proposed Standard), February 2013.

[CK13b] S. Cheshire and M. Krochmal. Multicast DNS. RFC 6762 (Proposed Standard), February 2013.

[CKK17] Victor Charpenay, Sebastian Käbisch, and Harald Kosch. \mu μ RDF store: Towards extending the semantic web to embedded devices. In Eva Blomqvist, Katja Hose, Heiko Paulheim, Agnieszka Lawrynowicz, Fabio Ciravegna, and Olaf Hartig, editors, *The Semantic Web: ESWC 2017 Satellite Events - ESWC 2017 Satellite Events, Portorož, Slovenia, May 28 - June 1, 2017, Revised Selected Papers*, volume 10577 of *Lecture Notes in Computer Science*, pages 76–80. Springer, 2017.

[CN04] James Cheng and Wilfred Ng. Xqzip: Querying compressed XML using structural indexing. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, volume 2992 of *Lecture Notes in Computer Science*, pages 219–236. Springer, 2004.

[Cro09] Dave Crocker. Internet Mail Architecture. `https://rfc-editor.org/rfc/rfc5598.txt`, July 2009. RFC 5598.

[CT04] John Cowan and Richard Tobin. XML Information Set (Second Edition). Technical report, W3C, 2 2004. Last visited on October of 2018.

[Deu96] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, May 1996.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE Conference on Local Computer Networks (LCN 2004), 16-18 November 2004, Tampa, FL, USA, Proceedings*, pages 455–462. IEEE Computer Society, 2004.

[DH17] Dr. Steve E. Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, July 2017.

[Fas17] LLC FasterXML. Smile Data Format. `https://github.com/FasterXML/smile-format-specification`, 2017. Last visited on July of 2018.

[FEL$^+$17] Steve Faulkner, Arron Eicholz, Travis Leithead, Alex Danilo, and Sangwhan Moon. HTML 5.2. Technical report, W3C, 12 2017. Last visited on June of 2018.

[FGM$^+$99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[FLMM06] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006*, pages 751–760. ACM, 2006.

[FLMM09] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):4:1–4:33, 2009.

[fou] OPC foundation. Opc-ua types schema. `https://opcfoundation.org/UA/2008/02/Types.xsd`. Last visited on October of 2018.

[FP14] Youenn Fablet and Daniel Peintner. Efficient XML Interchange (EXI) Profile for limiting usage of dynamic memory. Technical report, W3C, 09 2014. Last visited on October of 2018.

[FR14] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, June 2014.

[Fur13] Sadayuki Furuhashi. MessagePack. `https://msgpack.org/`, 2013. Last visited on July of 2018.

[GCB03] Jose A. Gutierrez, Edgar H. Callaway, and Raymond Barrett. *IEEE 802.15.4 Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensor Networks*. IEEE Standards Office, New York, NY, USA, 2003.

[GS00] Marc Girardot and Neel Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the web. *Computer Networks*, 33(1-6):747–765, 2000.

[GTMW11] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. *From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices*, pages 97–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[GZC13] Francis Galiegue, Kris Zyp, and Gary Court. JSON schema: core definitions and terminology. Internet-Draft draft-zyp-json-schema-04, Internet Engineering Task Force, January 2013. Work in Progress.

[Hag06] S. Hagen. *IPv6 Essentials*. O'Reilly Media, 2006.

[Ham92] Eric Hamilton. Jpeg file interchange format. Technical report, C-Cube Microsystems, Milpitas, CA, USA, 9 1992.

[HB15] Hill and W. Bruce. Evaluation of efficient XML interchange (EXI) for large datasets and as an alternative to binary JSON encodings. https://calhoun.nps.edu/handle/10945/45196, 2015.

[HLN10] Eran Hammer-Lahav and Mark Nottingham. Defining Well-Known Uniform Resource Identifiers (URIs). https://rfc-editor.org/rfc/rfc5785.txt, April 2010. RFC 5785.

[HRN+08] Nils Hoeller, Christoph Reinke, Jana Neumann, Sven Groppe, Daniel Boeckmann, and Volker Linnemann. Efficient XML usage within wireless sensor networks. In Xudong Wang and Ness B. Shroff, editors, *WICON*, ACM International Conference Proceeding Series, page 74. ICST, 2008.

[HRN+10a] Nils Hoeller, Christoph Reinke, Jana Neumann, Sven Groppe, Martin Lipphardt, B. Schuett, and Volker Linnemann. Stream-Based XML Template Compression for Wireless Sensor Network Data Management. In *MUE*, pages 1–9. IEEE, 2010.

[HRN+10b] Nils Hoeller, Christoph Reinke, Jana Neumann, Sven Groppe, Christian Werner, and Volker Linnemann. Efficient XML data and query integration in the wireless sensor network engineering process. *IJWIS*, 6(4):319–358, 2010.

[Huf52] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.

[ISO93] ISO/IEC. ISO/IEC 11172-3:1993 - Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 3: Audio. Standard, International Organization for Standardization, Geneva, CH, August 1993.

[JSA+18] Cullen Jennings, Zach Shelby, Jari Arkko, Ari Keränen, and Carsten Bormann. Media types for sensor measurement lists (SenML). Internet-Draft draft-ietf-core-senml-13, Internet Engineering Task Force, March 2018. Work in Progress.

[Kal18] Riyad Kalla. Universal Binary JSON Specification. http://ubjson.org/, 2018. Last visited on July of 2018.

[KK12] Ronny Klauck and Michael Kirsche. Bonjour contiki: A case study of a dns-based discovery service for the internet of things. In Xiang-Yang Li, Symeon Papavassiliou, and Stefan Rührup, editors, *Ad-hoc, Mobile, and Wireless Networks - 11th International Conference, ADHOC-NOW 2012, Belgrade, Serbia, July 9-11, 2012. Proceedings*, pages 316–329. Springer, 2012.

[KK13] Ronny Klauck and Michael Kirsche. Enhanced DNS message compression - optimizing mdns/dns-sd for the use in 6lowpans. In *2013 IEEE International Conference on Pervasive Computing and Communications Workshops, PERCOM 2013 Workshops, San Diego, CA, USA, March 18-22, 2013*, pages 596–601. IEEE, 2013.

[KK14] Sebastian Käbisch and Richard Kuntschke. Leveraging efficient XML interchange (EXI) for filter-enabled data dissemination in embedded networks. In Valérie Monfort and Karl-Heinz Krempels, editors, *Web Information Systems and Technologies - 10th International Conference, WEBIST 2014, Barcelona, Spain, April 3-5, 2014, Revised Selected Papers*, volume 226 of *Lecture Notes in Business Information Processing*, pages 79–95. Springer, 2014.

[KKD18] Kazuo Kajimoto, Matthias Kovatsch, and Uday Davuluru. Web of things (wot) architecture. Technical report, W3C, 09 2018. Last visited on October of 2018.

[Kle08] Dr. John C. Klensin. Simple Mail Transfer Protocol. `https://rfc-editor.org/rfc/rfc5321.txt`, October 2008. RFC 5321.

[KMB13] Antonio D. Kheirkhahzadeh, John P. T. Moore, and Jiva N. Bagale. Xml-compression techniques for efficient network management. In *Workshops Proceedings of the Global Communications Conference, GLOBECOM 2013, Atlanta, GA, USA, December 9-13, 2013*, pages 996–1000. IEEE, 2013.

[KPA15] Sebastian Käbisch, Daniel Peintner, and Darko Anicic. Standardized and efficient RDF encoding for constrained embedded networks. In Fabien Gandon, Marta Sabou, Harald Sack, Claudia d'Amato, Philippe Cudré-Mauroux, and Antoine Zimmermann, editors, *The Semantic Web. Latest Advances and New Domains - 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings*, volume 9088 of *Lecture Notes in Computer Science*, pages 437–452. Springer, 2015.

[KPED14] Rumen Kyusakov, Pablo Punal Pereira, Jens Eliasson, and Jerker Delsing. EXIP: A framework for embedded web development. *TWEB*, 8(4):23:1–23:29, 2014.

[KPHK11] Sebastian Käbisch, Daniel Peintner, Jörg Heuer, and Harald Kosch. Optimized xml-based web service generation for service communication in restricted embedded environments. In Zoubir Mammeri, editor, *IEEE 16th Conference on Emerging Technologies & Factory Automation, ETFA 2011, Toulouse, France, September 5-9, 2011*, pages 1–8. IEEE, 2011.

[LDM05] Gregory Leighton, Jim Diamond, and Tomasz Müldner. AXECHOP: A grammar-based compressor for XML. In *2005 Data Compression Conference (DCC 2005), 29-31 March 2005, Snowbird, UT, USA*, page 467. IEEE Computer Society, 2005.

[LE07] Christopher League and Kenjone Eng. Type-based compression of XML data. In *2007 Data Compression Conference (DCC 2007), 27-29 March 2007, Snowbird, UT, USA*, pages 273–282. IEEE Computer Society, 2007.

[Lea15] Christopher League. RNGzip — type-based XML compression. https://contrapunctus.net/league/haques/rngzip/, 2015. Last visited on July of 2018.

[lGA] Jean loup Gailly and Mark Adler. The GZIP home page. http://www.gzip.org/. Last visited on June of 2018.

[LHK16] Steve Liang, Chih-Yuan Huang, and Tania Khalafbeigi. OGC SensorThings API Part 1: Sensing. http://docs.opengeospatial.org/is/15-078r6/15-078r6.html, 2016. Last visited on June of 2018.

[Li10] C. Li. *Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization and Data Query Technologies: Label Streams, Semantics Utilization and Data Query Technologies*. IGI Global research collection. IGI Global, 2010.

[Lig16] Antonio Lignan. Zolertia Z1 mote. https://github.com/Zolertia/Resources/wiki/The-Z1-mote, 2016. Last visited on April 2018.

[LMD05] Gregory Leighton, Tomasz Müldner, and James Diamond. Treechop: A tree-based query-able compressor for xml. Technical report, In Proceedings of the Ninth Canadian Workshop on Information Theory (CWIT, 2005.

[LRB17] 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-09, Internet Engineering Task Force, July 2017. Work in Progress.

[LS00] Hartmut Liefke and Dan Suciu. XMILL: an efficient compressor for XML data. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 153–164. ACM, 2000.

[LZC13] Geraint Luff, Kris Zyp, and Gary Court. JSON Hyper-Schema: Hypertext definitions for JSON Schema. Internet-Draft draft-luff-json-hyper-schema-00, Internet Engineering Task Force, January 2013. Work in Progress.

[LZLY05] Yongjing Lin, Youtao Zhang, Quanzhong Li, and Jun Yang. Supporting efficient query processing on compressed XML files. In Hisham Haddad, Lorie M. Liebrock, Andrea Omicini, and Roger L. Wainwright, editors, *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*, pages 660–665. ACM, 2005.

[MF11] Alexey Melnikov and Ian Fette. The WebSocket Protocol. RFC 6455, December 2011.

[MGC16] Jorge Berzosa Macho, Luis Gardeazabal, and Roberto Cortiñas. Efficient Management of Data Models in Constrained Systems by Using Templates and Context Based Compression. In Carmelo R. García, Pino Caballero-Gil, Mike Burmester, and Alexis Quesada-Arencibia, editors, *Ubiquitous Computing and Ambient Intelligence - 10th International Conference, UCAmI 2016, San Bartolomé de Tirajana, Gran Canaria, Spain, November 29 - December 2, 2016, Proceedings, Part II*, volume 10070 of *Lecture Notes in Computer Science*, pages 332–343, 2016.

[MKB13] John P. T. Moore, Antonio D. Kheirkhahzadeh, and Jiva N. Bagale. Domain-specific XML compression. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2013 Data Compression Conference, DCC 2013, Snowbird, UT, USA, March 20-22, 2013*, page 510. IEEE, 2013.

[MKB14] John P. T. Moore, Antonio D. Kheirkhahzadeh, and Jiva N. Bagale. Towards markup-aware text compression. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014*, page 417. IEEE, 2014.

[Moc87] P. Mockapetris. Domain names - implementation and specification. `https://rfc-editor.org/rfc/rfc1035.txt`, November 1987. RFC 1035.

[Moo09] John Moore. Get stuffed: tightly packed abstract protocols in Scheme. In *10th Scheme and Functional Programming Workshop, Boston, USA, 22 Aug 2009*, pages 111–115, 2009.

[Moo10] John Moore. Everything counts in small amounts. In *Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS Darmstadt (Germany) November 15-16, 2010*, pages 273–283, 2010.

[MPC03] Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. XPRESS: A queriable compression for XML data. In Alon Y. Halevy, Zachary G. Ives, and AnHai

Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 122–133. ACM, 2003.

[MS10] Sebastian Maneth and Tom Sebastian. Fast and tiny structural self-indexes for XML. *CoRR*, abs/1012.5696, 2010.

[MTSG10a] Guido Moritz, Dirk Timmermann, Regina Stoll, and Frank Golatowski. encD-PWS - message encoding of SOAP web services. In *PerCom Workshops*, pages 784–787. IEEE, 2010.

[MTSG10b] Guido Moritz, Dirk Timmermann, Regina Stoll, and Frank Golatowski. Encoding and compression for the devices profile for web services. In *AINA Workshops*, pages 514–519. IEEE Computer Society, 2010.

[MZP$^+$09] Guido Moritz, Elmar Zeeb, Steffen Prüter, Frank Golatowski, Dirk Timmermann, and Regina Stoll. Devices profile for web services in wireless sensor networks: Adaptations and enhancements. In *ETFA*, pages 1–8. IEEE, 2009.

[net] Network configuration protocol (netconf) schema. https://www.iana.org/assignments/xml-registry/schema/netconf.xsd. Last visited on October of 2018.

[NLWL06] Wilfred Ng, Wai Yeung Lam, Peter T. Wood, and Mark Levene. XCQ: A queriable XML compression system. *Knowl. Inf. Syst.*, 10(4):421–452, 2006.

[Not10] M. Nottingham. Web Linking. RFC 5988 (Proposed Standard), October 2010.

[OA17] Olayinka O. Ogundile and Attahiru S. Alfa. A survey on an energy-efficient and energy-balanced routing protocol for wireless sensor networks. *Sensors*, 17(5), 2017.

[OCF] OCF. Open Connectivity Foundation. https://openconnectivity.org/. Last visited on June of 2018.

[ÖDE$^+$06] Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with COOJA. In *LCN 2006, The 31st Annual IEEE Conference on Local Computer Networks, Tampa, Florida, USA, 14-16 November 2006*, pages 641–648. IEEE Computer Society, 2006.

[PB18] Daniel Peintner and Don Brutzman. EXI for JSON (EXI4JSON). W3C working draft, W3C, 7 2018. Last visited on October of 2018.

[PPG14] Daniel Peintner and Santiago Pericas-Geertsen. Efficient XML Interchange (EXI) Primer. Technical report, W3C, 4 2014. Last visited on October of 2018.

[Rel] Relax ng specification. Technical report.

[Ros07] Marcel-Catalin Rosu. A-SOAP: adaptive SOAP message processing and compression. In *2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA*, pages 200–207. IEEE Computer Society, 2007.

[SA11a] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, March 2011.

[SA11b] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121, March 2011.

[SA15] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Address Format. RFC 7622, September 2015.

[SAC16] Peter Saint-Andre and Dave Cridland. Xep-0001: Xmpp extension protocols. Technical report, XMPP Standards Foundation (XSF), 11 2016.

[Sak09] Sherif Sakr. XML compression techniques: A survey and comparison. *J. Comput. Syst. Sci.*, 75(5):303–322, 2009.

[SB10] Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley Publishing, 2010.

[SCT13] Andy Stanford-Clark and Hong Linh Truong. MQTT for sensor networks (MQTT-SN) protocol specification, version 1.2. Technical report, International Business Machines Corporation (IBM), 11 2013.

[SEP] Zigbee smart energy profile 2.0. `http://www.zigbee.org/zigbee-for-developers/applicationstandards/zigbeesmartenergy/`. Last visited on October of 2018.

[SHB14] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014.

[She12] Z. Shelby. Constrained RESTful Environments (CoRE) Link Format. RFC 6690 (Proposed Standard), August 2012.

[SHG13] Ioakeim K. Samaras, George Hassapis, and John V. Gialelis. A modified dpws protocol stack for 6lowpan-based wireless sensor networks. *IEEE Trans. Industrial Informatics*, 9(1):209–217, 2013.

[SHK$^+$08] R. Senthilkumar, S. Daphne Hannah, A. Y. Raj Kumar, R. Joyson, and A. Kannan. Rfxfreeze: A non-queriable compressor for rfx storage structure. In *2008 International Conference on Computing, Communication and Networking*, pages 1–5, Dec 2008.

[SJDT15] Cox Simon J D and Peter Taylor. OGC observations and measurements – JSON implementation. https://portal.opengeospatial.org/files/64910, 2015. Last visited on March of 2018.

[SKB+18] Zach Shelby, Michael Koster, Carsten Bormann, Peter Van der Stok, and Christian Amsüss. CoRE resource directory. Internet-Draft draft-ietf-core-resource-directory-13, Internet Engineering Task Force, March 2018. Work in Progress.

[Ski16] Przemyslaw Skibinski. XWRT (XML-WRT). https://github.com/inikep/XWRT, 2016. Last visited on July of 2018.

[SKPK14] John Schneider, Takuki Kamiya, Daniel Peintner, and Rumen Kyusakov. Efficient XML Interchange (EXI) Format 1.0 (Second Edition). Technical report, W3C, 02 2014. Last visited on October of 2018.

[SML12] Tommy Szalapski, Sanjay Madria, and Mark Linderman. Tinypack XML: real time XML compression for wireless sensor networks. In *2012 IEEE Wireless Communications and Networking Conference, WCNC 2012, Paris, France, April 1-4, 2012*, pages 3165–3170. IEEE, 2012.

[SNR15] Radha Senthilkumar, Gomathi Nandagopal, and Daphne Ronald. Qrfxfreeze: queryable compressor for rfx. *The Scientific World Journal*, 2015, 2015.

[SS05] Hariharan Subramanian and Priti Shankar. Compressing XML documents using recursive finite state automata. In Jacques Farré, Igor Litovsky, and Sylvain Schmitz, editors, *Implementation and Application of Automata, 10th International Conference, CIAA 2005, Sophia Antipolis, France, June 27-29, 2005, Revised Selected Papers*, volume 3845 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2005.

[SS07] Przemyslaw Skibinski and Jakub Swacha. Combining efficient XML compression with query processing. In Yannis E. Ioannidis, Boris Novikov, and Boris Rachev, editors, *Advances in Databases and Information Systems, 11th East European Conference, ADBIS 2007, Varna, Bulgaria, September 29-October 3, 2007, Proceedings*, volume 4690 of *Lecture Notes in Computer Science*, pages 330–342. Springer, 2007.

[Tel] Cm5000 telosb. https://www.advanticsys.com/shop/mtmcm5000msp-p-14.html. Last visited on October of 2018.

[TGC17] Vlad Trifa, Dominique Guinard, and David Carrera. Web thing model. Technical report, W3C, 04 2017. Last visited on June of 2018.

[TH02] Pankaj M. Tolani and Jayant R. Haritsa. XGRIND: A query-friendly XML compressor. In Rakesh Agrawal and Klaus R. Dittrich, editors, *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 225–234. IEEE Computer Society, 2002.

[Tol02] Pankaj M. Tolani. XGrind. http://xgrind.sourceforge.net/, 2002. Last visited on July of 2018.

[Tom03] Vojtech Toman. Exalt (An Experimental XML Archiving Library/Toolkit). http://exalt.sourceforge.net/, 2003. Last visited on July of 2018.

[Tom04] Vojtěch Toman. Syntactical compression of xml data. In *Presented at 16th Intl. Conf. on Advanced Information Systems Engineering (CAiSE'04*, 2004.

[Uni15] International Telecommunication Union. ITU-T Recommendation X.690 (08/2015) - Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) . Technical report, 2015.

[W3C] W3C. WEB OF THINGS AT W3C. https://www.w3.org/WoT/. Last visited on June of 2018.

[WA18] Austin Wright and Henry Andrews. JSON schema: A media type for describing json documents. Internet-Draft draft-handrews-json-schema-01, Internet Engineering Task Force, March 2018. Work in Progress.

[WAL18] Austin Wright, Henry Andrews, and Geraint Luff. JSON schema validation: A vocabulary for structural validation of json. Internet-Draft draft-handrews-json-schema-validation-01, Internet Engineering Task Force, March 2018. Work in Progress.

[wc] webofthings.org community. WebOfThings.org. https://webofthings.org/. Last visited on June of 2018.

[WD16] Peter Waher and Yusuke DOI. XEP-0322: Efficient xml interchange (EXI) format. Technical report, XMPP Standards Foundation (XSF), 11 2016.

[WKB+07] G. White, J. Kangasharju, D. Brutzman, , and S. Williams. Efficient XML Interchange Measurements Note. Technical report, W3C, 02 2007. Last visited on October of 2018.

[WLLH04] Hongzhi Wang, Jianzhong Li, Jizhou Luo, and Zhenying He. Xcpaqs: Compression of XML document with xpath query support. In *International Conference on Information Technology: Coding and Computing (ITCC'04), Volume 1, April 5-7, 2004, Las Vegas, Nevada, USA*, page 354. IEEE Computer Society, 2004.

[WLS07] Raymond K. Wong, Franky Lam, and William M. Shui. Querying and maintaining a compact XML storage. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 1073–1082. ACM, 2007.

[WS4] WS4D-uEXI. `http://ws4d.org/2012/ws4d-uexi-available-as-open-source/`. Last visited on October of 2018.

[WWWCa] W3C World Wide Web Consortium. Document Object Model (DOM). `https://www.w3.org/DOM/`. Last visited on October of 2018.

[WWWCb] W3C World Wide Web Consortium. Document type definition (DTD). `https://www.w3.org/TR/xml/`. Last visited on October of 2018.

[WWWCc] W3C World Wide Web Consortium. Resource Description Framework (RDF). `https://www.w3.org/RDF/`. Last visited on June of 2018.

[WWWCd] W3C World Wide Web Consortium. XML schema. `https://www.w3.org/XML/Schema`. Last visited on October of 2018.

[ZC13] Kris Zyp and Gary Court. JSON Schema: interactive and non interactive validation. Internet-Draft draft-fge-json-schema-validation-00, Internet Engineering Task Force, January 2013. Work in Progress.