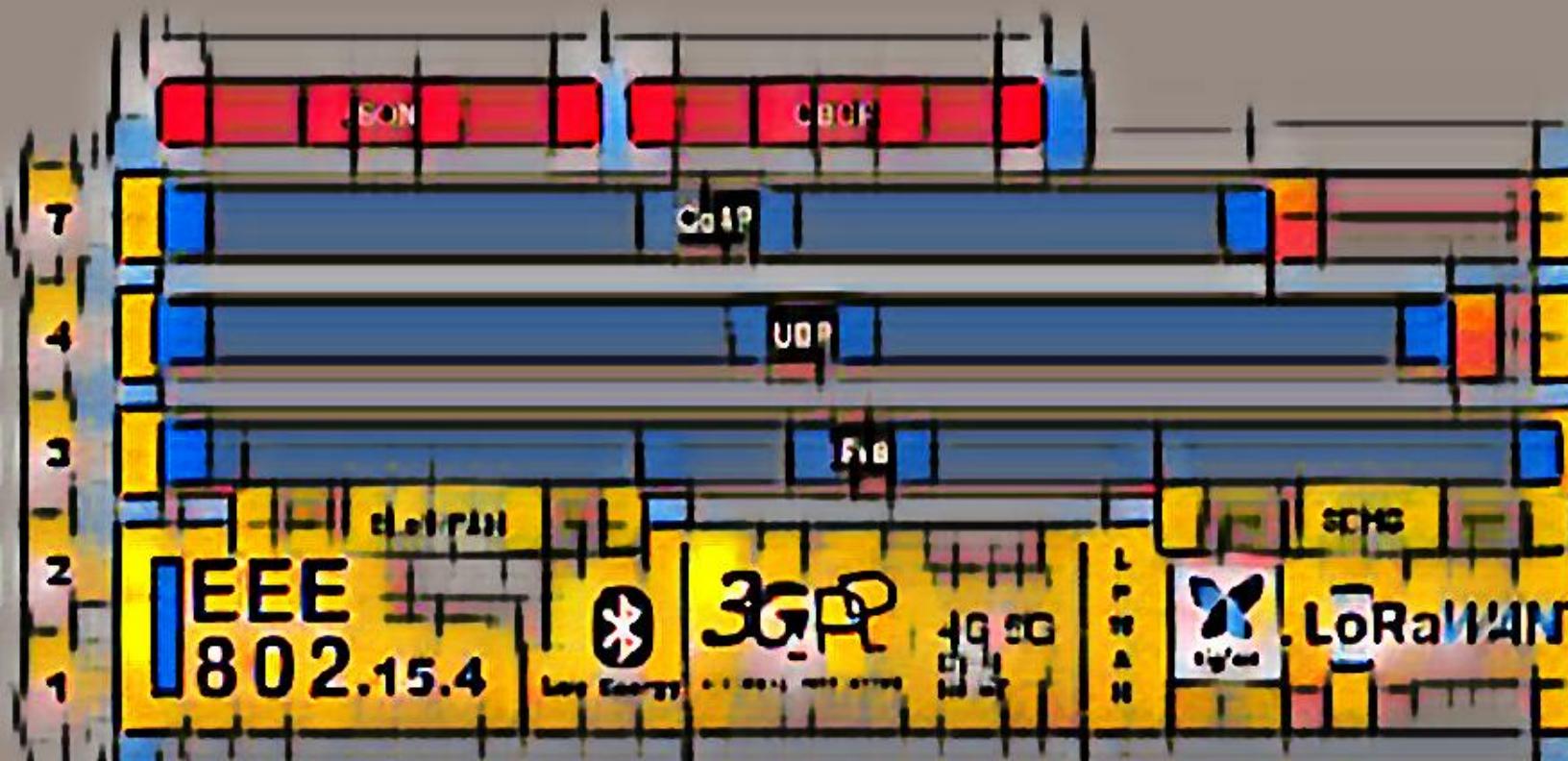


PROGRAMMING THE INTERNET OF THINGS

Laurent TOUTAIN



IMT ATLANTIQUE

Based on the PLIDO MOOC.

Published 16 avril 2022



Table of Contents

0.1	Available resources	13
0.2	Authors	13
1	THE BASIS OF THE INTERNET OF THINGS (IOT)	16
1.1	Introduction	16
1.1.1	Dedicated networks	16
1.1.2	3 technological phases	17
1.2	The Internet of Things	17
1.3	The problem	18
1.4	IoT evolution	20
1.5	Constrained objects	21
1.6	Interoperability	22
1.7	Le besoin de standardisation	24
1.8	Questions	24
2	ARCHITECTURE OF THE INTERNET	26
2.1	Protocols	26
2.2	Foundations of the Web	29
2.2.1	Resources	29
2.2.2	Identifiers	30
2.2.3	Interactions	32

2.3	Publish/Subscribe Model	33
2.3.1	MQTT	34
2.3.2	difference with REST	34
3	Wireshark	36
3.1	Installation	36
3.2	Startup	36
3.3	Capture	38
3.3.1	Web traffic analysis	40
3.3.2	Analysis of HTTP requests	42
3.3.3	Protocol stack analysis	43
3.4	Do it yourself	46
4	Modbus	48
4.1	Introduction	48
4.1.1	Registers	48
4.1.2	Protocol	49
4.1.3	Example : XY-MD02	49
4.1.4	IP Gateway	52
5	ARCHITECTURE FOR IOT	58
5.1	Introduction	58
5.2	Topologies	58
5.3	Layers 1 and 2	60
5.4	IP and adaptation layers	60
5.5	Implementation of REST	61
5.6	Data representation	62
5.7	Alternatives to REST	62
6	THE REPRESENTATION OF DATA	63
6.1	Introduction	63
6.2	The serialization	64
6.3	Base64	66
6.4	HTML	68
6.5	XML	68
6.6	JSON	69

6.7	CBOR	72
6.7.1	CBOR in Python	73
6.7.2	Type Positive Integer	73
6.7.3	Type Negative Integer	74
6.7.4	Type Binary sequence or Character string	75
6.7.5	Type Array	76
6.7.6	Type Tag	79
6.7.7	The floating type and particular values	79
6.8	Questions about CBOR	80
6.9	SenML	81
7	Implementing a Virtual Sensor	82
7.1	JSON	82
7.1.1	Minimal Server	82
7.1.2	Virtual sensor	83
7.2	CBOR	86
8	Time series	89
8.1	Sending an array	89
8.2	Differential coding	90
8.3	Architecture	91
8.4	Beebotte	92
8.4.1	Configuration	92
8.4.2	Enregistrement des ressources	92
8.4.3	Visualisation des ressources	95
8.5	Interopérabilité	96
8.6	et SenML ?	96
9	Découvrons le LoPy	102
9.1	Introduction	102
9.2	Installation d'Atom	103
9.2.1	Communiquez avec votre Pycom	104
9.2.2	Installez votre environnement de travail	105
9.3	Connexion au réseau Wi-Fi	106
9.4	Mise en place d'un client	107
9.5	BME 280	108
9.5.1	Le bus I2C	108
9.5.2	Mesure de la température	110
9.6	Thermomètre Wi-Fi	112

10	Sigfox	114
10.1	Récupération des identifiants	114
10.2	Enregistrement de l'objet	115
10.3	Visualisation des données émises par le Pycom	115
10.4	Que s'est-il passé coté radio	116
10.5	Récupération des données	116
10.5.1	Sur le serveur	118
10.5.2	Sur le LoPy	120
10.5.3	requête GET depuis le serveur	122
10.5.4	requête POST vers le serveur	124
10.6	Conclusion	128
11	LoRaWAN	129
11.1	Information sur le LoPy	130
11.2	The Things Network	130
11.3	Ajout d'une passerelle radio	145
11.3.1	Installation du Pygate	145
11.3.2	Configuration de The Things Network	146
11.4	Vue générale des échanges	146
11.5	Thermomètre LoRaWAN	147
12	CoAP	149
12.1	Introduction	149
12.2	Format d'une en-tête CoAP	150
12.2.1	Codage de code	151
12.2.2	Utilisation du champ Message ID	151
12.2.3	Les Tokens	154
12.2.4	Les options CoAP	156
12.2.5	Options CoAP	158
12.2.6	Représentation des URI	158
12.2.7	Représentation des données	161
12.3	Observe	162
13	Experimentons CoAP	165
13.1	Mise en œuvre du client/serveur	165
13.1.1	aiocoap	165
13.1.2	côté Objet	167

13.2 GET /time	169
13.3 POST	173
13.3.1 Ressource codée en ASCII	173
13.3.2 Ressource codée en CBOR	175
13.3.3 No Response	178
13.4 Chaîne complète de remonté de mesures	178
13.5 SCHC	179
13.5.1 Emission côté client	180
13.5.2 Réception côté serveur	181
13.5.3 serveur CoAP	182
13.6 Pistes d'améliorations	182
14 LwM2M	183
14.1 Introduction	183
14.2 Architecture	184
14.3 Enregistrement d'un Objet	185
14.3.1 Analyse de l'en-tête CoAP	185
14.3.2 Analyse du contenu du POST	187
14.3.3 Définition des ressources	188
14.4 Resource Directory	193
14.5 interrogation du client LwM2M	194
14.5.1 Lecture simple	196
14.5.2 Lecture d'une instance	197
14.5.3 Observe	197
15 Answers to the questions	199

Acronyms

3GPP	3rd Generation Partnership Project	IP	Internet Protocol
ABP	Authentication By Personalisation	IPv4	Internet Protocol version 4
ADSL	Asymmetric Digital Subscriber Line	IPv6	Internet Protocol version 6
AMQP	Advanced Message Queuing Protocol	IPSO	IP for Smart Objects
AS	Application Server	ITU	International Telecommunication Union
ASCII	American Standard Code for Information Interchange	IRI	International Resource Identifier
BLE	Bluetooth Low Energy	ISBN	International Standard Book Number
CBOR	Concise Binaire Object Representation	ISO	International Standardization Organization
CoAP	Constrained Application Protocol	JMS	Java Messaging Service
Cosem	Companion Specification for Energy Management	JSON	JavaScript Object Notation
CRC	Cyclic Redundancy Check	JSON-LD	JavaScript Object Notation for Linked Data
CSV	Comma Separated Values	LCIM	Levels of Conceptual Interoperability Model
DLMS	Device Language Message Specification	LPWAN	Low Power Wide Area Network
DTT	Digital Terrestrial Television	LwM2M	Lightweight Machine to Machine
DR	Data Rate	LNS	LoRaWAN Network Server
GSMA	GSM Association	MQTT	Message Queuing Telemetry Transport
HTML	HyperText Markup Language	NAT	Network Address Translation
HTTP	HyperText Transport Protocol	NGW	Network GateWay
HTTPS	HyperText Transport Protocol Secure	NIDD	Non IP Data Delivery
IANA	Internet Assigned Numbers Authority	OMA	Open Mobile Alliance
IBAN	International Bank Account Number	OTAA	Over The Air Authentication
IEEE	Institute of Electrical and Electronics Engineers	OVH	On Vous Herbègue
IETF	Internet Engineering Task Force	PAC	Porting Authorization Code
IoT	Internet of Things	REST	REpresentational State Transfer
		RFC	Request For Comments

RGW	Radio GateWay	TNT	Télévision Numérique Terrestre
RNIPP	Répertoire National d'Identification des Personnes Physiques	TTN	The Things Network
RSSI	Received Signal Strength Indicator	UDP	User Datagram Protocol
RTT	Round Trip Time	UIT	Union internationale des télécommunications
SCEF	Service Capability Exposure Function	UNB	(Ultra Narrow-Band
SenML	Sensor Measuring List	URI	Universal Resource Identifier
SCHC	Static Context Header Compression	URL	Universal Resource Locator
SF	Spreading Factor	URN	Universal Resource Name
SNR	Signal to Noise Ratio	VPS	Virtual Private Server
SSID	Service Set IDentifier	W3C	World Wide Web Consortium
STIC	Sciences et Technologies de l'Information et de la Communication	WWW	World Wide Web
TCP	Transmission Control Protocol	XML	Extensible Markup Language
TLV	Type Length Value	XMPP	Extensible Messaging Protocol et Presence



Introduction

The Internet of Things will not only add a new category of equipment to the network, it will also change the way protocols are implemented. So we need to be different but the same, disruptive but conservative. The Internet of Things is a major evolution of global protocols to meet two fundamental challenges : to be energy efficient and above all to be interoperable ; that is to say, to enable Objects to be easily integrated into existing information systems and ultimately to make the information produced widely available.

The IoT also changes the way we teach. Until now, this teaching was very stratified, with lectures showing how the protocols work and how they are organized. The practical implementation was more concerned with the configuration of interconnection equipment such as routers. The traditional vision consists in separating the functionalities. The protocols are stacked one on top of the other with the functionalities such as the coding of information, on which the routing of information is based, and at the top the applications. Each protocol in this stack is independent of the others and has very strict boundaries.

This vision has to be revised for the Internet of Things, the low memory capacities, the nature of the communication links make it difficult to specialize in only one field. Thus, I hoped to have forgotten forever the signal processing or electronic aspects when I left the university. However, in order to design an object, a multidisciplinary approach is essential, so you have to understand electronic concepts in terms of space as well as power consumption, signal processing because the signals are generally very weak, without forgetting traditional network problems such as routing, auto-configuration or security. It is also necessary to have an eye on the applications : how the data are represented and coded during their transmission and how they can interact with existing services.

The Thing itself is only a small part of the problem, it will send a more or less important flow of information to servers that will be responsible for analyzing, storing and finding trends. Since small streams make big rivers, these servers or the networks that lead to them must be correctly sized. But also that the cost of managing an object is very low, otherwise the cumulative cost of millions of objects can be a barrier to deployment..

This book is the result of the experiences we had with the FabLabs, and of an observation that the network is often the poor relation both of the applications created and of the support found on these platforms ; communications are seen from an application perspective, without taking into account a more global vision of interoperability leading to the concept of the Internet of Things. This is understandable because the protocol stacks of the Internet are relatively large and in a restricted environment, it is easier to get rid of them. Nevertheless, these protocol stacks have an advantage, they promote communication between the network components. Thus, it is possible to control an object from its laptop, data can be sent to servers to be processed,...

The variety of communication possibilities will allow the creation of innovative services. The protocol stacks must also adapt. This is what many organizations are doing, including the IETF, which standardizes the protocols used in the Internet.

This book is an adaptation of the MOOC Programming the Internet of Things on Coursera. It will cover the technologies, architectures and protocols needed for the end-to-end realization of information collection on networks dedicated to the IoT to the structuring of the data and its processing.

You're going to include :

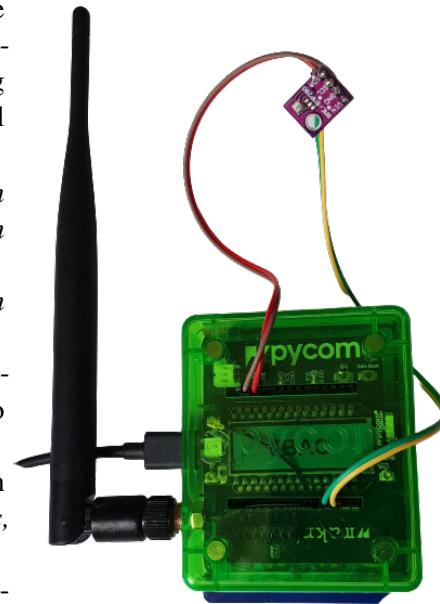
- discover a new category of networks called LPWAN of which Sigfox and LoRaWAN are the best known representatives ;
- see the evolution of the internet protocol stack from IPv4/TCP/HTTP to IPv6/UDP/CoAP while preserving the REST concept based on resources unambiguously identified by URI ;
- explain how CBOR can be used to structure complex data in addition to JSON ;
- Finally JSON-LD and the MongoDB database will allow us to easily manipulate the collected information. Thus, we will introduce the essential techniques to statistically validate the collected data.

Through this course, you will learn how to program an energy efficient Thing that is interoperable with other Things.

Hardware

You don't need hardware to do most of the exercises and experiment with the concepts. Just about everything can be done with Python scripts, but it's not as much fun as experimenting in real life. Especially when it comes to discovering the magic of long range radio networks. That's why we will use the following material :

- a LoPy4¹ - Pycom - *Quadruple Bearer MicroPython enabled Dev Board*; Caution : take the option *With headers*
- A Expansion Board 3.0² - Pycom - *Compatible with Pycom Multi-Network IoT*;
- une LoRa Antenna³ (868MHz/915MHz) & Sigfox Antenna Kit - Pycom with its small wire allowing to connect it to LoPy;
- a box⁴,but it is not necessary, if you are not careful with your device *Pycase Clear - Fits Pycom IoT Dev Boards, Expansion Board & Antenna kit*;
- A sensor BME280 3v3⁵ Capteur de pression température humidité BME280 - Boutique Semageek or BME280 3.3⁶;
- a USB cable (USB 2.0A to micro B 2.0 1.5 m);
- dupont male-femelle⁷ cable



LoPy offers a free one-year subscription to the Sigfox network, which is enough to experiment. An annual subscription costs about ten euros. On the other hand, accessing a LoRaWAN network is more problematic. The offers of the operators are not always adapted and the coverage of the community networks is not complete. But you can extend this coverage by installing your own LoRa antenna for less than a hundred euros. We will also use the solution proposed by Pycom.

To do this you need :

- an extension board pygate⁸;
- a LoPy as before or a slightly cheaper wi-py⁹;
- a pretty box¹⁰ to look professional ;

1. <https://pycom.io/product/lopy4/>
 2. <https://pycom.io/product/expansion-board-3-0/>
 3. <https://pycom.io/product/lora-868mhz-915mhz-sigfox-antenna-kit/>
 4. <https://pycom.io/product/pycase-clear/>
 5. <https://boutique.semageek.com/fr/704-capteur-de-pression-temperature-humidite-bme280-3009052078446.html>
 6. https://fr.aliexpress.com/item/1005002387867504.html?spm=a2g0o.productlist.0.0.580f7c3elwXr70&algo_pvid=bbd88dd7-92c5-4904-a4e7-6045b186dbd6&algo_expid=bbd88dd7-92c5-4904-a4e7-6045b186dbd6-1&btsid=0b0a182b16193744192742943e5955&ws_ab_test=searchweb0_0,searchweb201602_,searchweb201603_
 7. <https://www.amazon.fr/cable-dupont/s?k=cable+dupont>
 8. <https://pycom.io/product/pygate/>
 9. <https://pycom.io/product/wipy-3-0/>
 10. <https://pycom.io/product/pygate-case/>

- its antenna with its cable¹¹ ;
- and this time a USB-C cable.

0.1 Available resources

This Open Source book is available on https://github.com/ltn22/PLIDO_BOOK in French and English version. Remarks and comments can be sent back using Github tools like *Issue* and *Pull requests*.

The videos referenced in this book are also available on Youtube https://www.youtube.com/playlist?list=PLwy4KbYJoKLWeonJ8c5U6CLrxs_pZQi3q. And as the Youtubers say, don't forget to like the videos and subscribe to the channel.

Finally, a more interactive version, in the form of a MOOC, is available here : <https://bit.ly/3Ku0aL8>. The MOOC allows more interactivity with forums to directly ask questions and animations to better configure the system.

0.2 Authors

Laurent Toutain is a professor in the Network Systems, Cybersecurity and Digital Law department at IMT Atlantique, a school of the Mines-Telecom Institute. He is a member of the OCIF team (Communicating Objects - Internet of the Future) which focuses on protocol and architectural evolutions of the Internet related to the design of new services (Smart grid, smart clothes...). After having worked on the IPv6 protocol and transition mechanisms in different environments, he is currently interested in their integration in the Internet of Things. He also contributes to Fab Labs for the adoption of these protocols. He is the author of several reference books on networks.



Kamal Singh est Maitre de Conférences à Télécom Saint-Étienne où il dispense des cours de réseaux informatiques et de réseaux d'opérateurs. Il fait également partie de l'équipe de recherche Data Intelligence du Laboratoire Hubert Curien. Son travail porte sur l'internet des objets, les villes intelligentes, le Big Data, le Web sémantique, la qualité de l'expérience et le software defined networking.



11. <https://pycom.io/product/lora-868mhz-915mhz-sigfox-antenna-kit/>

Marc Girod Genet is an associate professor at Télécom Sud-Paris and a CNRS-SAMOVAR associate researcher (UMR 5157), where he leads the transverse theme on energy. His research interests include personal networks (including sensor networks and measurement architectures), M2M communications and IoT/WoT architectures, semantic data models and ontologies. Marc is also involved in standardization activities within AIOTI (Alliance for Internet of Things Innovation) and ETSI (rapporteur, TC Smart-BAN). In 2010, he received the special "Digital Green Growth" jury award for his work on smart grids and energy management (one of his two application areas along with eHealth).



Patrick Maillé is a professor in the Network Systems, Cybersecurity and Digital Law department of IMT Atlantique, a school of the Mines-Telecom Institute. A graduate of the Ecole Polytechnique and Télécom ParisTech, he defended his thesis at Télécom Bretagne (now IMT Atlantique) in 2005. His research work focuses on the economics of telecommunication networks using applied mathematics and economics tools (notably game theory).



Mireille Batton-Hubert is a Professor at the École Nationale Supérieure des Mines de Saint Étienne, at the teaching and research center, the Henri Fayol Institute. She is in charge of the Mathematical and Industrial Engineering team (GMI) and is attached to the UMR Limos. The Mathematical and Industrial Engineering department aims to propose quantitative methods for evaluating the overall performance of companies and their products. The Mathematical and Industrial Engineering department aims to propose quantitative methods for evaluating the overall performance of companies and their products. The Mathematical and Industrial Engineering department aims to propose quantitative methods for evaluating the overall performance of companies and their products. It brings together skills ranging from applied mathematics to industrial engineering.



Vincent Lerouvillois, Teaching Assistant



1. THE BASIS OF THE INTERNET OF THINGS (IOT)

1.1 Introduction

In this first part of the course, we will lay the foundations of what the Internet of Things (IoT) is. What do we mean by "Internet of Things" in the context of the book? What are the issues that the Internet of Things must address and its evolution today? What are the underlying technologies, architectures and protocols that will be used in this book?

To do this, we will draw a parallel between the way the Internet has integrated television and what we are currently experiencing with the Internet of Things.

Youtube



1.1.1 Dedicated networks

In the 1950s, television became very popular and almost every household bought a television set to watch their favorite programs. Dedicated transmission networks were deployed all over the world. In fact each type of communication had its own network one for radio one for telephone one for telex,...

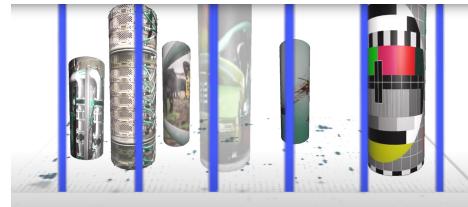
In the 80's, the internet was born, but the transmission speeds were low and the network was limited to file uploads. In the 90s, images and hypermedia with the World Wide Web (WWW) appeared, in connection with the increase of the speeds. Also in the 90s, the increase in power of microprocessors allowed the digitization of the TV signal. Televisions started to include microprocessors and the networks went from analog to digital transmission; but they remained dedicated to this single use broadcasting television. With the entry in the new millennium, the Internet gained in speed with the Asymmetric Digital Subscriber Line (ADSL) and optical fibers. It was possible to integrate images in web pages



but the quality was poor. At the same time, hundreds of television channels were broadcasting their programs in high resolution via satellite or Digital Terrestrial Television (DTT). Nowadays, Internet communications have gained in speed and quality and some countries have cut off DTT and chosen to transmit their programs only via Internet. In fact the use of the internet is not only a change in the distribution network but also a major change in the uses and applications. You can watch television on your cell phone or even watch your favorite series whenever you want, on demand.

1.1.2 3 technological phases

From this example, we can define three phases in the development of a technology. In the first phase, a specific network is built for a well-defined use. This is called a **vertical** approach; a technology is dedicated to a single use. It is difficult to exchange information between two verticals. We also refer to **silos** because they are isolated. In a second phase, the verticals start to integrate common technologies but not in a coordinated way. They still cannot communicate easily because they have not made the same choices.



In a final phase, verticals coordinate to converge on the same technologies by defining common rules and uses. This is done in order to reduce costs or to increase their impact in this case. We talk about **horizontal** because it covers several sectors. The Internet has become one of these horizontals for many services. The Internet of Things follows this same movement. Particular solutions have emerged to solve specific needs in agriculture, in the automotive industry, in health, in energy. When Internet networks allowed low cost communications, the architecture of the Internet was taken into account but without compatibility. The change we are currently experiencing is the definition of common functionalities for different domains. The goal is to reduce costs but also to cross information for a better management of the industrial process and a better use of resources.



1.2 The Internet of Things

How do we define the Internet of Things ? Or rather, which Internet of Things are we going to study ? The ambiguity of the two terms "Internet" and "Things" requires a more precise definition ; or at least a classification to better understand what we are referring to.

The internet is now totally integrated in our lives, for work, for education, for entertainment. We use it at home or at work on our computers, and we carry it with us more and more with our smartphones.

Everyone has their own definition of what the internet is. For the general public, it can be very popular applications like Facebook, Tik-Tok, Netflix, Zoom. For some, a little more technophile, the Internet can be confused with the Web which is accessed via Chrome or Firefox. Technicians will talk about protocols like IP, TCP, HTTP, and addresses like IP addresses or URLs.

As this book is technology-oriented, our approach falls more into the latter category. We will see how protocols developed twenty years ago for computers can be applied to other devices that we have yet to define.

The goal of the Internet of Things is to further integrate the Internet to allow something other than computers to exchange data. The main goal is to optimize processes so that they are more efficient to save resources or increase productivity. It is therefore a buried internet, far from the fridge or the connected watch, which will bring back information with an infrastructure or other equipment. We can imagine sensors in a factory to control production, connected cars that will talk to each other to avoid collisions, measuring the filling rate of recycling bins in a city to optimize collection circuits, monitoring the degree of humidity in a field to reduce water consumption...

The Internet of Things can be summarized as follows : using protocols developed for computers and now cell phones (more powerful than the computers used by the Internet in its early days) but in more constrained environments. Indeed, Moore's laws, defining the processing power of processors as well as the continuous decrease of memory costs, have allowed us to provide small objects with resources comparable to those of computers of thirty years ago.

The Internet of Things means solving the following equation : continue to do the same thing because all current information systems use the same principles but do it differently because these principles are too costly in terms of energy, computing time and data exchange.

The Internet of Things (IoT) is a global architecture allowing objects (equipment of the same type or not) to interact autonomously via the Internet. This interaction :

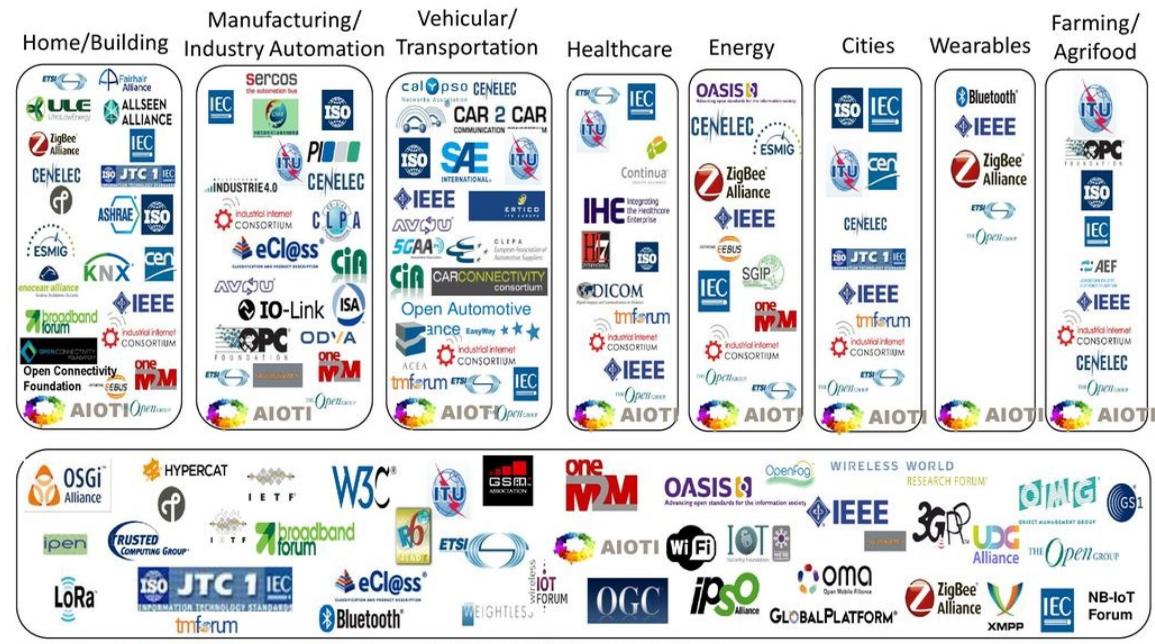
- is realized, by construction, through an Internet network, which generally implies that the objects/things are provided with an IP address ;
- is related to commands (control operations or function calls) or data or information exchanges.

The new IoT paradigm is a convergence of many application domains such as : smart homes or buildings, cities of the future, industry of the future (industry 4.0), energy, transportation systems, agriculture, eHealth, etc., towards a reduced, interoperable and secure protocol suite. The movement is underway and, given the number of players involved, will take several years. But the foundations are already well established and that's what you'll learn in this book.

1.3 The problem

One of the problems with the Internet of Things is that the IoT doesn't start ex-nihilo. Now that the technologies that made the Internet successful are mature, it's not just a matter of applying them to a new domain. Objects were able to communicate long before the Internet existed. Each sector has already developed its solutions, more or less standard, more or less proprietary.

La figure 1.1 on the facing page reprend un certain nombre de travaux et de groupes qui spécifient les protocoles pour l'internet des objets. The figure 1.1 on the next page lists a number of works and groups that specify protocols for the Internet of Things.



Source: AIOTI WG3 (IoT Standardisation) – Release 2.7

Horizontal/Telecommunication

FIGURE 1.1 – Some IoT standards

Without going into detail, we can see that some logos are found in several places, that there is a profusion of solutions for each sector that hinders interoperability and evolution. The Internet of Things, in its broadest sense, is about simplifying this architecture, just as the Internet did a few years ago in the telecoms sector, by simplifying this model and enabling these different players to converge on a common architecture and a smaller set of solutions.

This does not necessarily mean fewer players, but greater consistency in technological choices.

The figure 1.2 on the following page analyzes the IoT by application domain, focusing on the network component. IoT and connected objects are complex systems for which open source solutions, alliances between manufacturers, and standardization bodies are still fragmented; however, to a lesser extent, showing that structuring is underway.

This fragmentation of the ecosystem is paradoxical. If we summarize, the proposed solutions amount to interrogating a piece of equipment in the field to access a value, process it and send back a command to interact with the environment.

Why are there so many different solutions? It may come from the needs of reliability, security, data scope, but it also comes from history. Communication with Things is just as old as communication between computers (which are themselves Things). But at the time, each field went its own way, specializing solutions to meet its own needs. The result is solutions optimized for a particular domain. But every time a technology has to be modified, the work has to be adapted for each domain, introducing additional costs and delays.

Also, since each domain has its own data representation, it is relatively difficult to combine them to have a more global vision. We therefore end up with closed systems, expensive, not very scalable, but optimized for the tasks they have to perform.

Over time, the protocols of the Internet could be used, but it is mainly a matter of complementing existing technologies without it being possible to interconnect two domains.

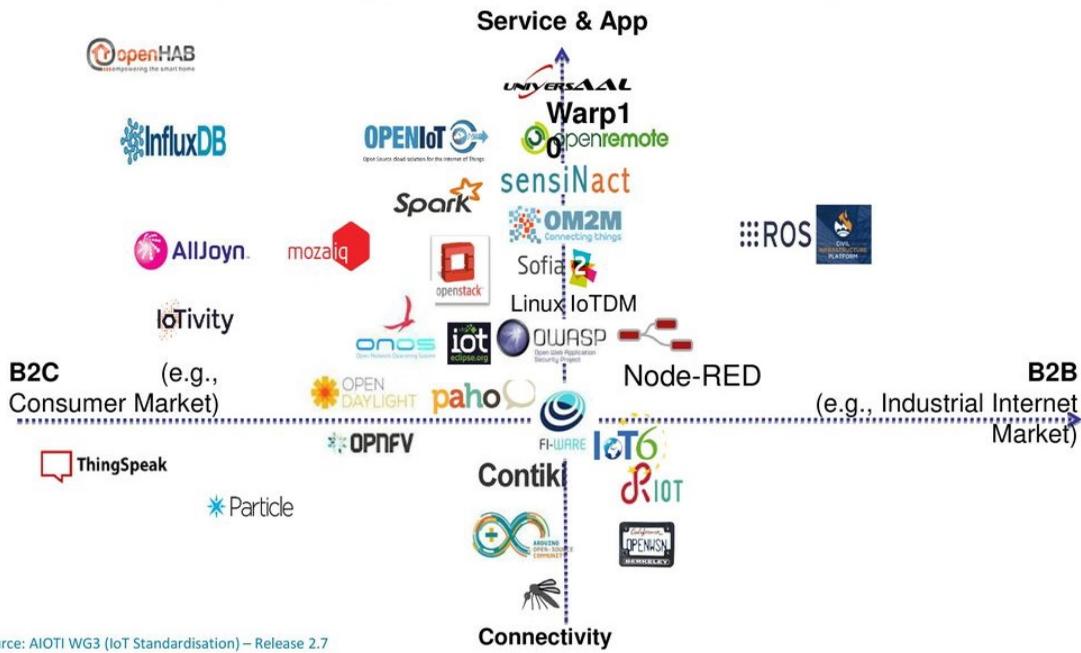


FIGURE 1.2 – Some open source applications for the IoT

1.4 IoT evolution

The example of the evolution of the television network speaks for itself. At the beginning, this network was analog and highly specialized to broadcast programs carried on analog signals on specialized equipment, the television sets.

With the progress of computer processors, it becomes possible to transport data using digital coding. But specialized networks are still needed, as the Internet does not offer the same quality.

The last step consists in integrating these flows into the traditional Internet. This becomes possible by increasing the speed of wired and radio networks (Wi-Fi, 4G...).

This pooling of accesses via the Internet allows not only a reduction of costs, but also the appearance of new uses such as television on cell phones or series on demand.

The Internet of Things follows the same path. In addition to specific technologies, the protocols of the Internet are integrated, but adapted to the contexts of the sector. We are currently experiencing the convergence towards a reduced set of protocols, a standardization of data representation, and its processing on more generic platforms.

The trigger is not the increase in speed as for television, but the possibility of having inexpensive equipment, with reduced capacities compared to traditional computing and energy autonomous, while having a better integration in the current information systems.

1.5 Constrained objects

With the progress of electronics, processors are becoming more and more powerful and the supercomputers of yesterday are now in a watch or a smart-phone.

For the Internet of Things, the logic is a little different. Moore's Law will lead to a reduction in manufacturing costs rather than an increase in processing power. The main criterion for a massive Internet of Things remains energy ; connecting a device to a power source or recharging a battery has a cost. Increasing the speed of the processor or the size of the memory induces a higher energy consumption of the object. We can therefore expect a certain stability in the performance of the objects because they will remain limited in performance.

Objects are generally limited in terms of processing power, memory and energy. According to the IETF standard [RFC 7228](#), devices can be divided into three classes that are also found in the segmentation of processors :

- Class 0, with less than 10 kB of volatile memory to store temporary data and 100 kB of Flash memory to store the object's computer code. It is the equivalent of an **Arduino UNO** (2 kB RAM and 32 kB Flash). It is almost impossible to install both the protocols used to communicate on the Internet (even in a restricted way) and the applications that run on it.
- Class 1 has about 10 KB of RAM and 100 KB of Flash. With an adaptation, it is possible to install an IP stack. For example, equipment like the Pycom Lopy4 that we will use later (and which is in the upper limit) on which the operating system is minimal. Thus, the Pycom uses a simplified version of the Python language (micro-Python) which allows it to be adapted to the system's limitations.
- Class 2 is less restricted with at least 50k RAM and 250k Flash (like a **Raspberry Pi**). The **Linux** operating system can run on these devices. Therefore, there are few limitations on the IP stack and the applications running on it. Class 1 devices have too many restrictions to use the protocols defined for larger objects. The Internet Engineering Task Force (IETF), the organization that standardizes Internet protocols, has proposed a revision of its protocol stack to adapt its protocol stack to a constrained environment.

Figure 1.3 on the next page summarizes the means of interconnection according to the class of the object :

- A class 0 device cannot directly use the internet to exchange information, hence the need to install a gateway to capture the traffic and send it to the internet. It does not have an IP address directly. The LoRaWAN gateways LoRaWAN Network Server (LNS) and 3GPP Service Capability Exposure Function (SCEF) act in this sense (we will come back to this). The data produced is encapsulated by these gateways in protocols such as HyperText Transport Protocol (HTTP) or Message Queuing Telemetry Transport (MQTT), which we will also see later in the course.
- Class 1 devices can also use a gateway to interconnect to the traditional internet, but rather than encapsulating the data produced in other protocols as Class 0 does, gateways for Class 1 devices will translate a constrained protocol into its equivalent in the unconstrained world. Class 2 devices can interact directly with other nodes on the Internet, without going through a gateway.

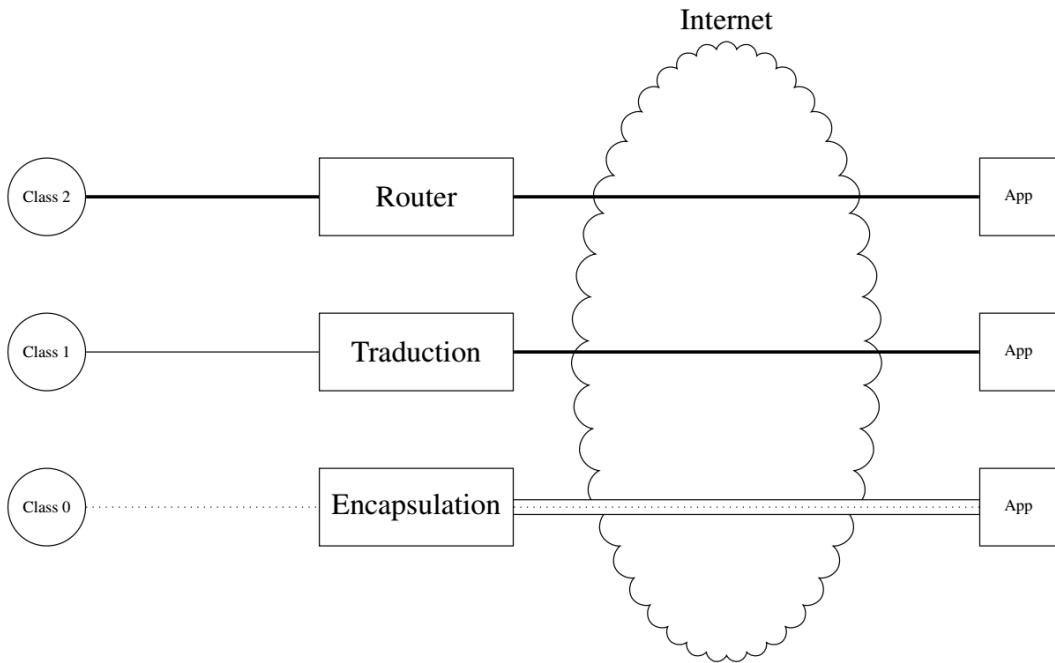


FIGURE 1.3 – Interconnection possibilities

1.6 Interoperability

Another challenge is the number of devices. Some studies predict 500 billion devices by the end of the decade.

With this massive Internet of Things, where almost every piece of equipment will include sensing or actionable elements, integration into an information system will become a real challenge. Because today's Internet of Things is designed for a vertical (i.e., for specific applications), devices are chosen and integrated at the time of system design. Engineers choose their sensors, know exactly what their characteristics are, and write their code based on what they have integrated.

The massive Internet of Things is a game changer. Devices or things cannot be integrated into a static information system from the start. The integration has to happen over time and has to handle device evolutions for a long time (device manufacturers may change, products will evolve with new features, etc.).

The Internet we know is a very good illustration of the need for interoperability. It has allowed, thanks to a standardization of the network and a strong decrease of the transmission costs, to develop new uses. You would never have invested in a network dedicated to videoconferencing and teleworking. But by pooling uses, it is possible ! The Internet of Things must follow the same path and also converge with the architecture of the current Internet. The key word is interoperability.

This interoperability issue has been formalized in the model Levels of Conceptual Interoperability Model (LCIM) [TM03] (see figure 1.4 on the facing page) can be represented by a counter to measure



INTEROPERABILITY

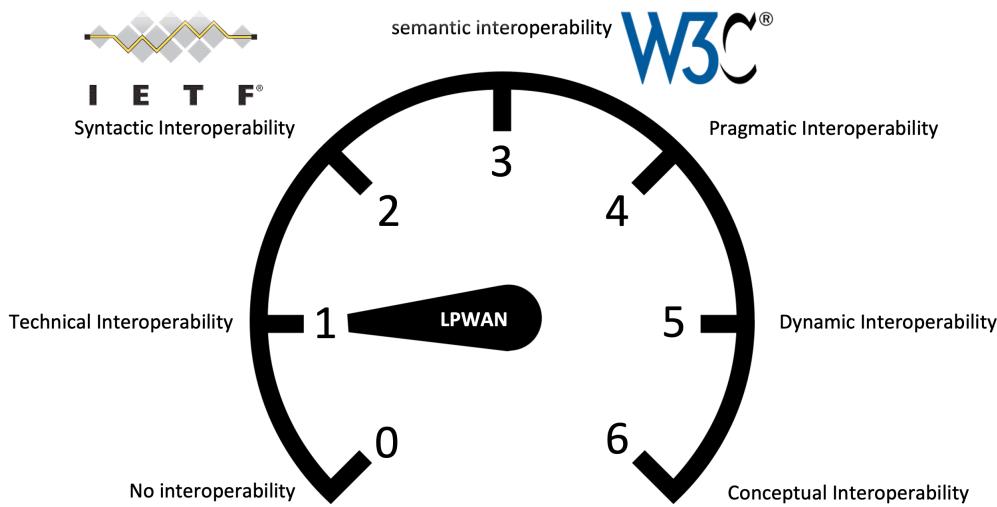


FIGURE 1.4 – Level of interoperability

the degree of interoperability.

It distinguishes six levels of interoperability, among which :

- At level zero, we are not connected ; we don't talk to anyone, so we don't have interoperability problems.
- At level 1, we are able to transmit information, but both sides must know the rules. We have an integrated system ; the applications must know precisely the specifications of the objects with which they communicate, because they define their own data exchange formats. We could take the example of an electrical board where a processor communicates with sensors via a printed circuit. The code running on the processor can be written in advance because there is little chance that a user will loosen the components to replace them with others. This corresponds to encapsulation, if we consider the networked objects in figure 1.3 on the preceding page, the interconnecting element must be configured to encapsulate the data to the right application at the right receiver depending on the sending object.
- Syntactic interoperability (level 2) where two nodes can exchange data without first being configured for this exchange. This is the case of the Internet. By using this suite of protocols, by having a valid address on the network, any application is able to exchange data with another. On the other hand, the data you will exchange is specific to an application. Video conferencing is a great example of Level 2 interoperability. You cannot use Zoom if your correspondent uses Teams because the formats are different. The IETF is the grouping of different actors (industrial, academic,...) who produce the standards related to this network. It is recognized by the acronym Request For Comments (RFC) followed by a number.
- Semantic interoperability (level 3) implies that the receiver is able to interpret the received data. The web is a very good example of level 3 interoperability. Whatever your browser, you can display the pages of a web site and follow the links. The meaning of the information

is understood in the same way on both sides. For the Web, the format HyperText Markup Language (HTML) makes it possible using tags (keywords) to structure a text by adding formatting information or links to other documents. The World Wide Web Consortium (W3C) defines the standards.

- The higher levels of interoperability will be linked to the accuracy of the model that will represent the system.

1.7 Le besoin de standardisation

Objects did not wait for the Internet to communicate. They have each evolved in their own vertical, developing solutions that are satisfactory but limited in terms of evolution and interoperability. This report on the dispersion of ecosystems highlights the need to :

- coordination among the alliances ;
- harmonization or alignment of standards ;
- to have reference implementations and reference models as soon as possible.

Otherwise, interoperability will not be properly addressed, new innovative multi-domain services will not be covered, and the development of IoT could be stalled or even aborted.

Aspects related to education and training of IoT actors, as well as those related to acceptance, uses and obviously the socio-economic aspect of IoT, are also essential points that must be considered.

Standards bodies such as the IETF or the W3C have designed protocols or data models capable of handling interoperability. In a way, this is the key to the success of the current Internet. It has solved the problem of interoperability at the syntactic and semantic level but at the cost of large messages.

The challenge for the Internet of Things is to integrate into this giant distributed system. As the Internet of Things is a newcomer, the evolution will have to be done on its side to take into account the existing rules, but adapting them. The next chapters will deal with these changes.

1.8 Questions

Question 1.8.1: New Area

Communicating objects are a brand new field, linked to the progress in miniaturization of electronic components :

- True
- False

Question 1.8.2: Protocoles

Which of these statements is true ?

- There are very few protocols to make objects communicate. As the Internet is a technology that has been very successful, its success will allow objects to communicate.
- There are many solutions to enable objects to communicate, the Internet of Things must enable them to be federated.

Question 1.8.3: Source of data

What is the primary source of data creation in the IoT ?

- sensors
- nanocomputers (Raspberry Pi type)
- Internet
- Web servers

Question 1.8.4: Challenges

What are the main technological challenges for the Internet of Things (3 answers) ?

- Have a low energy consumption.
- Have a simplified protocol architecture.
- Be able to run on open operating systems like Linux.
- Allow to secure data that may be sensitive.
- Continuously transmit their status and measured values.

2. ARCHITECTURE OF THE INTERNET

2.1 Protocols

You probably know the principle of protocol stacking in networks. Each protocol provides a service and relies on the lower layer to perform it. The original model defines seven layers to transport the data of an application, anywhere in the world. The network protocols are stacked on top of each other, with those above using the services offered by those below to carry the data. This gave rise to the reference model of the International Standardization Organization (ISO) which has been structuring networks since the 1970s. In theory, there are **7 layers**, but the Internet has made this model evolve and the numbers of the layers, associated with functionalities, have remained; this can lead to a strange numbering.

The Internet has simplified this architecture (see figure 2.2 on the facing page). That's why there are fewer layers and the numbers are not contiguous.

The first two layers from the bottom, grouped under the name of Interface, allow the transmission of binary data on a physical medium. Layer 1 deals with this modulation on a particular physical medium (optical fiber, copper pair, radio wave). Layer 2 groups together the mechanisms that allow this data to be structured in finite size blocks called frames, to define the access methods,

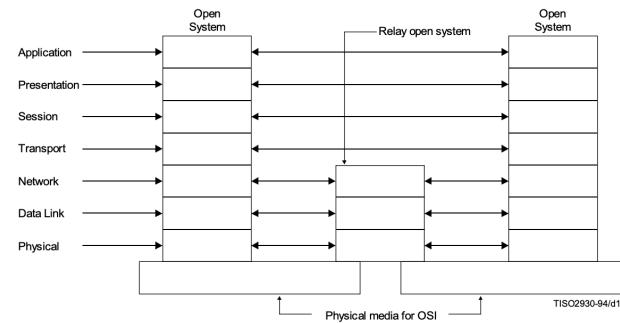


Figure 12 – Communication involving relay open systems

FIGURE 2.1 – Extract from ITU-T Rec. X.200 (1994 E)

Youtube



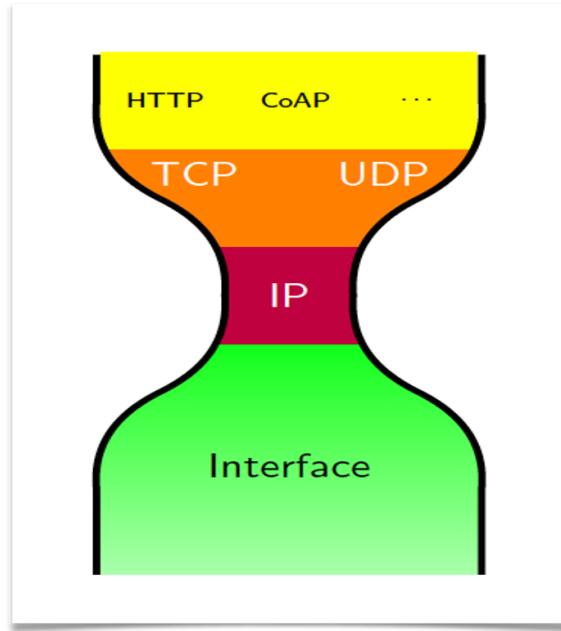


FIGURE 2.2 – Protocol layering of the Internet

i.e. when the equipment can transmit, and the address formats used to identify the equipment.

- the Institute of Electrical and Electronics Engineers (IEEE) which proposes standards like Ethernet for the wired networks or Bluetooth and Wifi for the radio networks,
- the 3rd Generation Partnership Project (3GPP) which operates at the same level and defines the protocols for cellular telephony (4G),
- ...

Above, we have the IETF standardized protocol. The Internet Protocol (IP) protocol adapts simply to any medium of communication. IP thus proposes an abstraction of the means of communication for the application layers, making the access to the network and the addressing universal. The treatment in the **routers** (equipment in charge of routing the information in the network) must be as fast as possible to treat a maximum of packets per second. Moreover, IP does not specialize for one service or another; it only routes the packets to the right destination. The Internet is a worldwide network built around this protocol, potentially reaching all the equipment connected to it.

Internet experts like this representation in **hourglass** where IP appears in a central position but is smaller compared to the other protocols. By design, IP is very simple ; both to be easily ported to many Layer 2s and to be easily used by higher layers, but also to process data very quickly in the interconnection nodes.

IP is implemented everywhere on the Internet as well in the equipment at the end of the network as in the routers in charge of sending the data to the right destination.

Above that are two protocols that are only implemented in the end devices. If level 3 allows to reach a machine, level 4 will allow to identify the application which must process the data. The

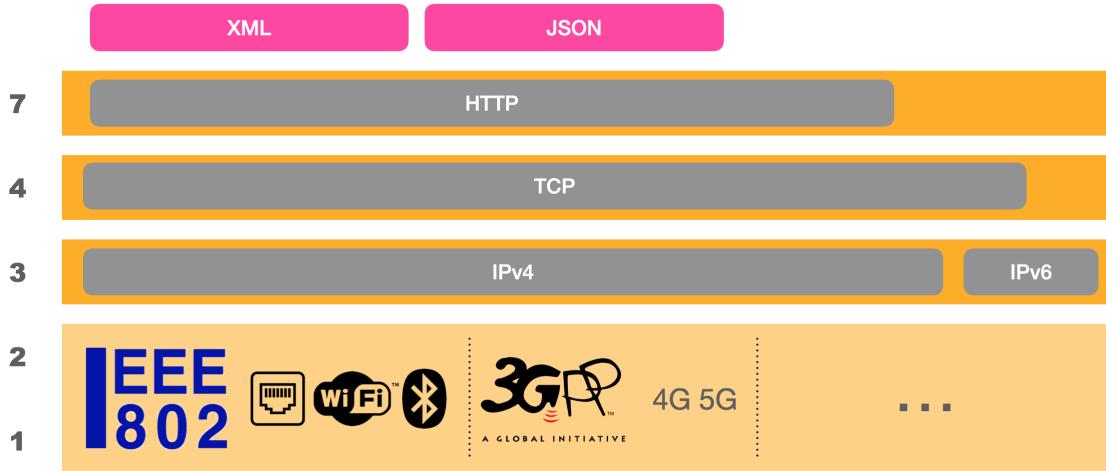


FIGURE 2.3 – Main Internet protocols

"addresses" of these applications are numbers between 1 and 65535 called **ports**. For example, Web servers use port number 80 or 443.

The protocol Transmission Control Protocol (TCP) will monitor the transferred data and will be able to retransmit lost data, slow down or accelerate the data transfer if it detects a network saturation. On the other hand, its implementation is complex and costly in memory. In the simple cases, User Datagram Protocol (UDP) is preferred ; it does not bring additional treatment UDP, it is a minimal protocol which is satisfied to direct the data towards the good application without any other control.

Above, we find the applications that historically are classified in layer 7. The applications are very numerous but the most widespread is HyperText Transport Protocol (HTTP) which is used to transport the Web pages, but also it allows direct communications between computers.

For the general public, the Internet designates above all the totality of this protocol assembly and is often confused with the application that democratized its use : the Web. This is also true for technicians, the traffic produced by the Web is largely present in the Internet. This diagram, figure 2.3 shows the protocol stack that is mostly used in the Internet. We see that at level 3 we have two versions of the protocol ; version 4 is the version historically deployed and it has been so successful that it is more and more difficult to have addresses for machines. To keep the network running, a new version has been developed. Internet Protocol version 6 (IPv6) makes the addressing almost infinite with addresses on 128 bits. IPv6 gains little by little ground in the traditional uses and it is especially an essential brick for the Internet of the objects.

The Web uses mainly the protocol HTTP. And as HTTP relies on TCP, these two protocols are dominant on the network.

Finally this graph adds an additional layer, above layer 7, to indicate how the transported data is structured with formats like Extensible Markup Language (XML) or JavaScript Object Notation (JSON) that we will see in the following.

Question 2.1.1: Protocol Stack

In the internet protocol stack, which protocols are responsible for routing packets to their destination (2 answers)

- Ethernet
 - IEEE
 - 802.15.5
 - Wi-Fi
 - IPv4
 - IPv6
 - UDP
 - TCP
 - MQTT
 - HTTP
 - CoAP
 - XML
 - JSON

2.2 Foundations of the Web

One of the most important success stories based on the Internet is the architecture that led to the Web since it is a great source of inspiration for the development of new services. The Web forms large distributed systems and is based on several principles that make it universal and scalable. Surfing with a browser is only the visible part of the traffic ; the principles of the web are also used for video streaming, exchanges between computers.

The Web and its extensions are based on a client-server model. Servers own resources and clients can access or modify them through a protocol such as HTTP. The client-server model is something common in computer networks, but the Web follows certain design guidelines known as REpresentational State Transfer (REST).

According to Roy Fielding, who defined this model, REST is a set of principles, properties and constraints. REST uses the client-server communication model and generally uses the HTTP protocol.

The REST principle allows to design scalable servers. A server must be stateless, which means that it does not retain any information after responding to a client request. This simplifies the processing in the server that has to handle requests from a large number of clients.

This requires that the report be located on the client side. This state is fed from the structured data that the client receives from the server. Thus, when a client requests a Web page, it can contain other Universal Resource Identifier (URI) to complete it, for example images, style sheets, scripts, etc.

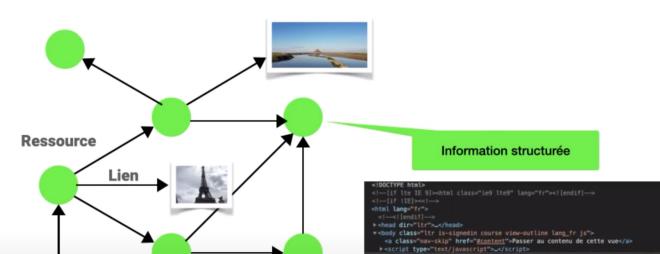
The client must therefore understand the data that the server sends him and therefore know the representation format of the resource that he receives in order to find the URI. In addition to the resource itself, the server adds additional information, called metadata. Among other things, this information includes the format of the content (content format). It can be pure text, an image or a structured text format such as HyperText Markup Language (HTML) or JSON.

Youtube



2.2.1 Resources

The basic element is the resource, which can be defined as a data block of



finite size. Resources can themselves contain references to other resources which in turn will refer to other resources etc. This forms a mesh between resources which is compared to a spider's web. The resource can be, for example, an image in which case it will not refer to anything else. To refer to another resource, its content must be structured and therefore defined in a format where it is easy to understand that part of the content is a reference to another resource. HTML is one such language that allows web pages to reference each other through links.

2.2.2 Identifiers

Each Web resource is identified by a unique value called URI. If the URI contains international characters, (like accented letters, ...) it is called International Resource Identifier (IRI).

For example, to identify an image, we can name it

image

but there is little chance that this name is unique, other people on Earth have surely had the same idea. On the other hand, if I preface it with my phone number

33667789078 image

will be unique if I name only one resource "image". Another user on the same principle can name his resource :

33667239018 image

without any possible ambiguity. However, because the phone number is unique in the phone number space, other unique numbers could conflict in other numbering spaces.

To avoid conflicts, it is interesting to give the numbering space at the beginning, for example :

tel : 33667789078 image

and

ss : 33667789078 image

the two identifiers will be unique, even if by chance this phone number and this social security number coincide.

The URI formalize this principle. The [RFC 3986](#) explains how they can be constructed. A URI starts with a scheme indicating the naming authority, followed by an authority value and then a path in the authority space. Characters such as ":" or "/" are used to improve the readability of the URI.

For instance :

mailto : mduerstifi.unizh.chssh://utilisateurexample.com

ftp : //ftp.is.co.za/rfc/rfc1808.txt



FIGURE 2.4 – Structure of a URI

`http://example.com/ma_ressource`

No one else in the universe will be able to identify their resources with this string since `example.com` belongs to me. I therefore have an infinite namespace that allows me to designate the infinite set of resources without anyone else being able to take the same names. An URI is an administrative construct that allows you to assign a globally unique identifier to a specific resource.

The URI (cf.figure 2.4) is intended to easily name a resource, to be able to link resources together to form this global spider web. The schema defines both the namespace of the authority and its format. An address or domain name as an authority is both a way to ensure global uniqueness, but also to know how to access the resource.

For example, **spotify** has defined its own schema and then it no longer needs authority but structures the path to reference a playlist.

All the books have a number International Standard Book Number (ISBN) which allows to identify. It can also be integrated in a URI. These two types of identifiers make it possible to refer to a single object but only by reading it one cannot reach the resource. This sub-family of URI identifiers is called Univeral Resource Name (URN).

A subset of URI can be used directly to locate the resource, i.e. find out on which server the resource is located and how to access it. This is a Univeral Resource Locator (URL) that is well known to the general public and used by Web browsers.

The `http` schema is very convenient because it can also be read as an URL. This scheme gives :

- le protocole à utiliser pour accéder à la ressource (`http`),
- the authority that indicates the server address (and its port),
- and finally, the access path of what we are going to ask to the server and which can sometimes correspond to a tree of files on a server.

But we must see that the initial goal is to make a unique identifier. The `https` schema gives the way the suite will be built and in a second time only will be seen as the protocol to use to access the resource. The authority is unique and in a second time will be used to locate the server. And finally the path will indicate how to access the resource on the server. So the resources of our global spider web are present on servers and each resource has a unique identifier. First, the client knows the URI of a resource. If it is a URL, he can contact the server. The server returns the resource to him. The client analyses it and discovers the URLs it contains. It can then question the other server(s) to reconstruct locally a part of the web necessary for the treatment that the client wants to carry out.

```

▼ Hypertext Transfer Protocol
  ▶ GET /site/kamalsingh25/ HTTP/1.1\r\n
    Host: sites.google.com\r\n
    User-Agent: Mozilla/5.0 (X11; Linux i686; rv:45.0) Gecko/20100101 Firefox/45.0\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: en-US,en;q=0.5\r\n
    Accept-Encoding: gzip, deflate\r\n
    Connection: keep-alive\r\n
  \r\n
\[Full request URI: http://sites.google.com/site/kamalsingh25/\]
  [HTTP request 1/1]

```

FIGURE 2.5 – Content of an HTTP GET request

Question 2.2.1: Unicity

What is unique in the world (6 answers) ?

- A first name.
- A family name.
- a social security number used in France.
- a passport number.
- a cell phone number with its international prefix.
- a full bank account number (International Bank Account Number (IBAN)).
- the IP address of my machine in my private network.
- the IP address of a Coursera server (13.225.34.28).
- the domain name plido.net.
- the name of a city.

2.2.3 Interactions

The interactions between clients and servers are very simple. The client will manage the interactions with the resources on a server. It can, for example, retrieve a resource using a **GET** method. It can also write data in an existing resource thanks to a method **PUT**.

The number of interactions is very limited. HTTP or HyperText Transport Protocol Secure (HTTPS) is a way to implement these methods.

HTTP is a protocol which can be used to implement a Web server the principles of REST. (qualified in English of **RESTfull**). HTTP defines different methods for the client to interact with resources on the server :

- **GET** is used to retrieve the representation of a resource (e.g. Web page, temperature value of a sensor, etc.). For example, the figure 2.5 gives the header format HTTP GET to retrieve a Web page ;
- **HEAD** est utilisée pour récupérer uniquement les métadonnées présentes dans les en-têtes de réponse sans le corps de réponse ;
- **POST** is used to inform the server of a new resource ;
- **PUT** is used to store a resource at the location identified by the URI in the request. If the resource already exists, it will be modified ;
- **PATCH** allows the client to modify only a part of the resource ;

— **DELETE** is used to delete the specified resource.

Question 2.2.2: State

The server keeps track of previous requests ?

- True
- False

Question 2.2.3: Wold Wide Web

The World Wide Web is based on this principle of states for :

- work on both computers and smartphones,
- be able to serve a large number of requests,
- encrypt communications.

Question 2.2.4: Presentation of Information

What formats are used to represent structured information (2 answers) :

- | | | | |
|-----------------------------------|-------------------------------|-------------------------------|-------------------------------|
| <input type="checkbox"/> Ethernet | <input type="checkbox"/> IPv4 | <input type="checkbox"/> MQTT | <input type="checkbox"/> JSON |
| <input type="checkbox"/> IEEE | <input type="checkbox"/> IPv6 | <input type="checkbox"/> HTTP | |
| <input type="checkbox"/> 802.15.5 | <input type="checkbox"/> UDP | <input type="checkbox"/> CoAP | |
| <input type="checkbox"/> Wi-Fi | <input type="checkbox"/> TCP | <input type="checkbox"/> XML | |

Question 2.2.5: Scheme

In the URI <https://plido.net/unit/definition.html>, where is the scheme ?

Question 2.2.6: Authority

In the URI <https://plido.net:8080/unit/definition.html>, where is the authority ?

2.3 Publish/Subscribe Model

There are other formalisms than REST. Another formalism, very popular, is broadcast oriented by using the "publish/subscribe" principle. As we will show in the following, even if the functionalities between these two modes may seem similar to HTTP, the design philosophy is very different : publish/subscribe aims at integrated applications while REST aims at global interoperability.

The publish/subscribe model makes the decoupling between the sender of a message and its recipient. In this paradigm (see figure 2.6 on the following page), there are "Publishers" who produce data or messages and send the message to an entity generally called "Broker". In addition, messages can be classified into "Topics", contents or types, etc. Then, there are subscribers who subscribe to the broker, for example to a given topic, in order to receive the messages that interest them, as shown in the diagram.

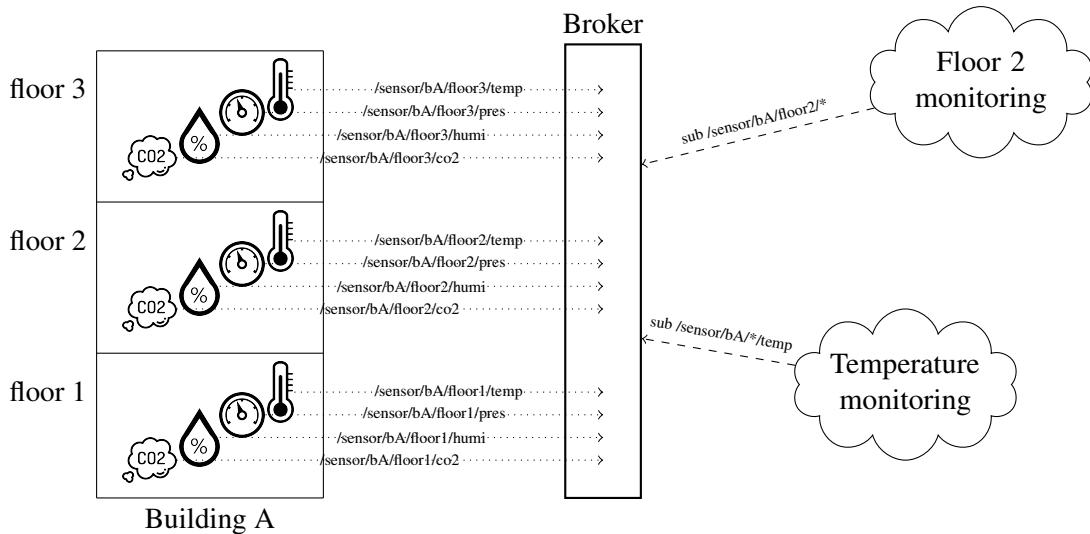


FIGURE 2.6 – Example of MQTT topics.

The broker can then use filters to send only these messages to the subscribers of the concerned topic. There are several Publish-Subscribe protocols such as Message Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP), Java Messaging Service (JMS) or Extensible Messaging Protocol et Presence (XMPP).

2.3.1 MQTT

MQTT is detailed in the rest of the course because it is very popular for communication between processes, but also in the Internet of Things.

Originally developed by IBM in 1999, for the MQSeries middleware, it became an OASIS standard in 2013, and in 2016 an ISO standard¹.

For example, let's imagine that several sensors are installed, on several floors, in two buildings A and B. Some sensors collect temperature information and others collect humidity information. These sensors can send the data regularly to a central broker.

The data can be classified into different topics which can also be organized hierarchically. For example, the topic /sensor means "all sensor data", /sensor/buildingA/ means "sensor data only installed in building A". In addition, /sensor/buildingA/floor3/temperature could mean "temperature sensor data installed only on the third floor in building A".

Some subscribers may subscribe to messages based on their interest. For example, a subscriber interested only in the humidity data of the whole building B can subscribe to the topic /sensor/buildingB/*/humidity and the broker will send only this data to this subscriber.

2.3.2 difference with REST

The main advantages of the publish-subscribe paradigm over the client-server paradigm, as included in REST, are the following :

1. <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.pdf>

- weak coupling between sender and receiver, the broker acts as an intermediary and stores the information ;
- scaling. Data from a source is sent out only once by the source. The broker copies it to all subscribers. In a client/server mode, the data must be sent by the server as many times as the clients request.

The lack of coupling between the sender and the receiver is done in terms of space, time and synchronization. The one who publishes the data has a simplified task. He doesn't have to manage or know who is consuming it, he only has to send it to the broker.

MQTT is very lightweight and designed for low-power devices. It has a very small software footprint and is optimized to work in low-bandwidth environments. This makes MQTT ideal for IoT applications. Even so, the use of TCP and the many, many acknowledgements can be cumbersome for highly constrained devices or networks. A lighter version based on UDP exists for these use cases, but it is not widely used.

Although they are similar, the naming principles of MQTT topics and REST URIs are completely different. Compared to MQTT, the path in the URI has no semantics. It is just meant to be unique. It cannot be used to aggregate multiple information sources. If two sensors publish respectively on the topics /sensor/buildingA/temperature and /sensor/buildingB/temperature, a subscriber can subscribe to the topic /sensor/*/temperature in order to receive all the measurements ; this is impossible with REST : it will be necessary to make as many requests as sensors to get all the measurements.

URIs are simply unique to the world in their construction, whereas MQTT topics are application specific. An MQTT topic can be interpreted differently by two different applications. This does not allow for semantic interoperability. Subscribers must be constructed with knowledge of the topics used by the publishers.

3. Wireshark

Wireshark is going to be our friend in the rest of this book. It will help us to understand the protocols and to analyze the data which will circulate. Unfortunately, in certain cases, we must resort to more rustic tools such as traces in **hexadecimal**¹ (base 16). It is thus necessary to become familiar with these tools. We will do straight away by analyzing simple HTTP requests. If you have access to a computer that can run Wireshark, we recommend that you try to do the manipulations described below and answer the questions.

3.1 Installation

The installation of Wireshark is done by going on the eponymous site <https://www.wireshark.org/>, or under Linux by installing the package `wireshark`. This program requires particular rights to access messages coming from the network, you must grant them at the time of the installation.

3.2 Startup

If you launch Wireshark with the right privileges, the welcome window will display the available interfaces, as shown in the figure 3.1 on the next page on Windows. Compared to the reference model of the ISO, these are all the level 2 interfaces present on the computer. It can be a physical card like Ethernet or Wi-Fi or a virtual interface used to communicate internally on the computer.

It is a question of determining which interface to choose. This is not always easy because their names are not always very explicit. The small curves on the left of the name indicate the instantaneous traffic that Wireshark measures. On the diagram, 3 interfaces are active : Ethernet, communication with a virtual machine and an interface called *Indexloopback*. The first one allows to have the communication with the outside and the last one will be very useful during the exchanges between two processes in this machine.

1. <https://en.wikipedia.org/wiki/Hexadecimal>

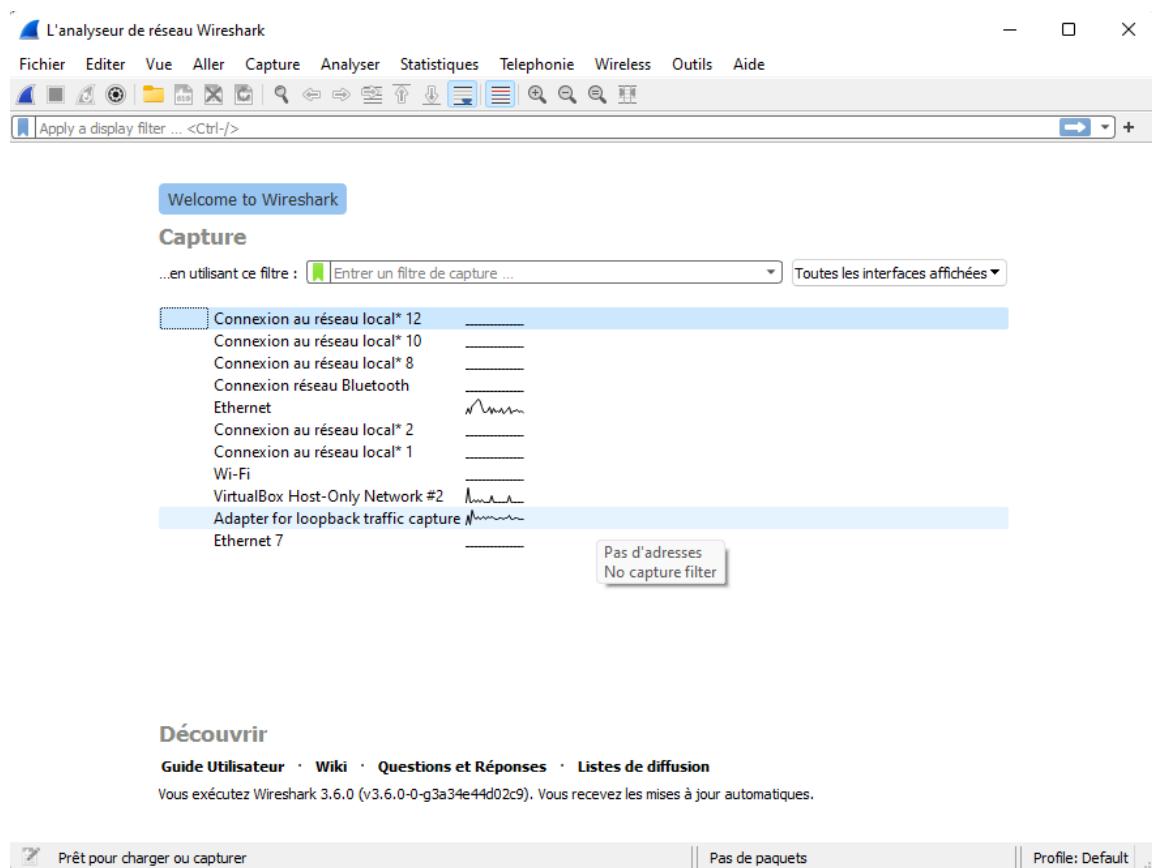


FIGURE 3.1 – Wireshark opening

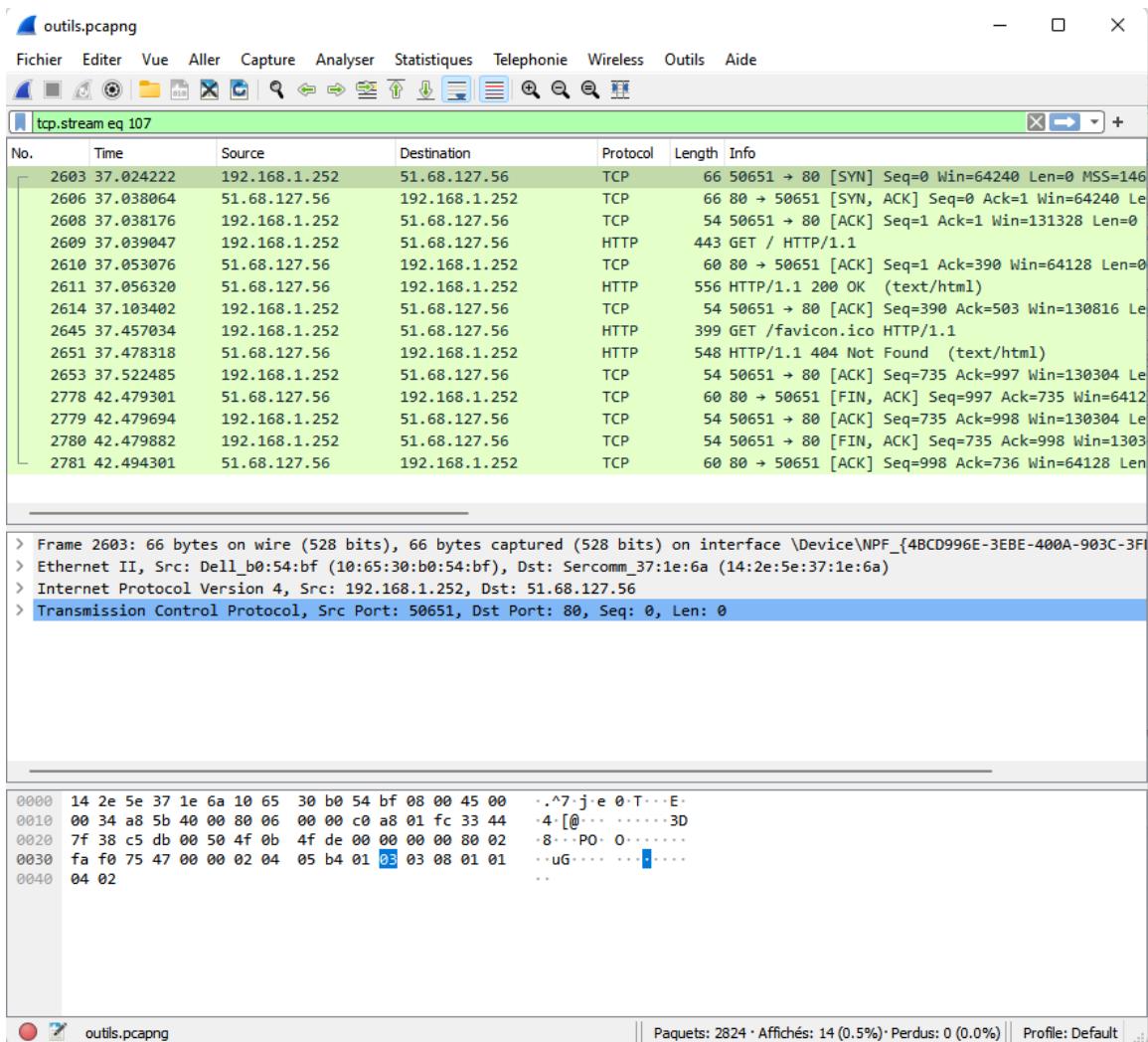


FIGURE 3.2 – Traffic Capture

3.3 Capture

By clicking on the name of the interface giving access to the external network (**Ethernet** in our case), the window splits into 3 parts, as shown in the figure 3.2.

The Wireshark screen is divided into 3 parts :

- at the top, scrolls the frames that are captured on the network, each protocol has a dedicated color for easy identification :
 - the captured frame number, this is an information added by Wireshark,
 - the time of capture of the frame. This information is also added by Wireshark,
 - the IP address (IPv4 or IPv6) of the machine originating the packet,
 - the IP address (IPv4 or IPv6) of the machine receiving the packet,
 - the highest level protocol contained in the frame. In our case, this can be TCP if the TCP message does not contain data, as in the case of connection opening, or certain acknowledgements. We also see the messages HTTP which are of course encapsulated

- in TCP,
- the size in bytes of the frame captured by Wireshark,
 - Finally Wireshark provides a summary of the content of the frame, to understand what is happening on the network. In the screenshot, we can see for the messages, the GET requests or the notifications ;
 - if a frame is selected in the list, it appears in the middle area with the protocol stack. The content of each of these protocols can be detailed by clicking on the small triangle on the left ;
 - the bottom window gives the equivalent in hexadecimal. The highlighted parts correspond to the fields selected in the middle window. Note that the information is found both in hexadecimal and in the character American Standard Code for Information Interchange (ASCII), which helps in reading when looking for a specific value.

Question 3.3.1: Column one

In the first column :

- The frame number assigned by Wireshark upon reception
- The frame number is read directly in the Ethernet frame

Question 3.3.2: 2nd column

In the second column :

- The time of reception by Wireshark
- The sending time of the frame

Question 3.3.3: The third and fourth columns

In the third and fourth columns :

- The Ethernet addresses of the hosts.
- Only the IPv4 addresses of the machines.
- IPv4 or IPv6 addresses of machines.

Question 3.3.4: The fifth column

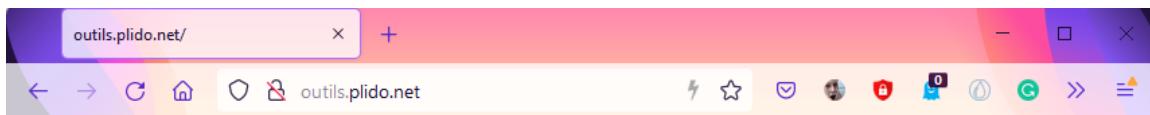
In the fifth column :

- The application protocol (level 7).
- The last (higher level) recognized protocol.
- The level 4 protocol (here TCP or UDP).

Question 3.3.5: The sixth column

In the sixth column :

- The size in bits of the frame.
- The size in bytes of the frame.



Hello!

This is the default web page for this server.

It just display a simple page. You will find a more sophisticated page [here](#)

Question 3.3.6: The seventh column

In the seventh column :

- A summary of the information carried by the higher-level protocol.
- IPv4 options.
- The ASCII content of the highest level message.

That's a lot of traffic, so we'll limit what is displayed by adding a filter to a particular recipient. The site tools.plido.net at IPv4 address 51.68.127.56. In the window currently showing *Apply a display filter*. type the following instructions :

```
ip.addr==51.68.127.56
```

don't forget the double ==. The window should turn green when everything is typed indicating that the filter syntax is correct. When you press enter, the window should be empty.

3.3.1 Web traffic analysis

In the address bar of your favorite browser, type the following URL :

```
http://outils.plido.net
```

and the Web page indicated figure 3.3.1 must appear.

Wireshark allowed to visualize the traffic exchanged between the computer and the Web server. The traffic should be similar to the one in the figure 3.2 on page 38. The figure is obtained by selecting the menu *Statistics/Flow Graph* and checking *Limit to Display Filter*. It is a little more readable because it represents the exchanges in the form of time diagrams.

Three phases can be distinguished :

- TCP connection opening with the emission of three TCP messages ;
- the data transfer phase :
 - the client sends an HTTP GET request to the server to request the resource from the root (/),
 - the server acknowledges the message at the TCP level to indicate that it has been received.
 - the server sends the answer to the previous request and specifying the status (200 : OK) and the content is formatted in HTML.

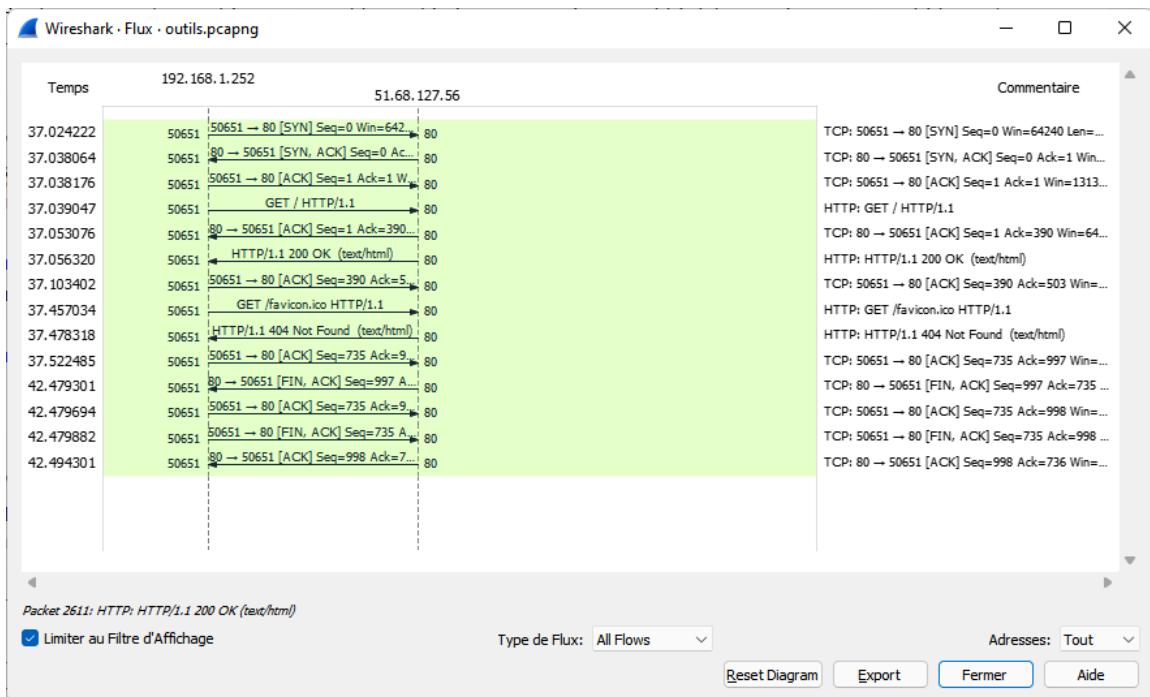


FIGURE 3.3 – Time diagram of exchanges.

- the client acknowledges this message at the TCP level,
 - the client sends a new HTTP GET request to obtain the resource /favicon.ico
 - the server answers that the resource does not exist (404 : Not Found). This request implicitly acknowledges the previous message.
 - the client acknowledges the server's response at the TCP level.
- the server ends the connection after 5 seconds of inactivity. The closing is done by exchanging 4 TCP messages.

Question 3.3.7: HTTP notification code

In the following trace, we saw that the server responded to client requests with a 3-digit number. Using the [RFC 7231](#), can you assign the left digit to a category of notifications :

- 0
1
2
3
4
5

- Redirection
 Error on the server side
 Error on the client side
 Unassigned
 Success
 Information

3.3.2 Analysis of HTTP requests

The version 1.1 of the protocol HTTP is specified by the [RFC 7230](#). We will look at a small description in English of the architecture and the formats of the messages.

2. Architecture

HTTP was created for the World Wide Web (WWW) architecture and has evolved over time to support the scalability needs of a worldwide hypertext system. Much of that architecture is reflected in the terminology and syntax productions used to define HTTP.

2.1. Client/Server Messaging

HTTP is a stateless request/response protocol that operates by exchanging messages across a reliable transport- or session-layer "connection". An HTTP "client" is a program that establishes a connection to a server for the purpose of sending one or more HTTP requests. An HTTP "server" is a program that accepts connections in order to service HTTP requests by sending HTTP responses.

Question 3.3.8: Standardization body.

Which standards organization published this document ?

- Microsoft
- ISO
- IEEE
- IETF

The terms "client" and "server" refer only to the roles that these programs perform for a particular connection. The same program might act as a client on some connections and a server on others. [...]

Most HTTP communication consists of a retrieval request (GET) for a representation of some resource identified by a URI. In the simplest case, this might be accomplished via a single bidirectional connection (==>) between the user agent (UA) and the origin server (O).

```
request    >
UA ===== 0
          < response
```

A client sends an HTTP request to a server in the form of a request message, beginning with a request-line that includes a method, URI, and protocol version, followed by header fields containing request modifiers, client information, and representation metadata, an empty line to indicate the end of the header section, and finally a message body containing the payload body.

A server responds to a client's request by sending one or more HTTP response messages, each beginning with a status line that includes

the protocol version, a success or error code, and textual reason phrase possibly followed by header fields containing server information, resource metadata, and representation metadata, an empty line to indicate the end of the header section, and finally a message body containing the payload body.

The following example illustrates a typical message exchange for a GET request on the URI "http://www.example.com/hello.txt":

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

Question 3.3.9: Formatting HTTP messages

Do HTTP headers have a fixed size (you can check the [RFC 7231](#) which gives indications on the protocol) ?

- the header is one line of 80 characters.
- a blank line separates the header from the content. The header can contain as many lines as necessary.

Question 3.3.10: HTTP Header Options

How are the optional lines in the header constructed ?

- keyword : values
- unformatted text
- keyword : data length : values

3.3.3 Protocol stack analysis

The frame containing the HTTP GET request allows to visualize the protocol encapsulation defined by the reference model of the ISO. In Wireshark, by clicking on the frame, we can see it disassembled and in hexadecimal in both windows as shown in the figure 3.4 on the next page.

The second window shows the protocol stack, inverted with respect to the classical representations

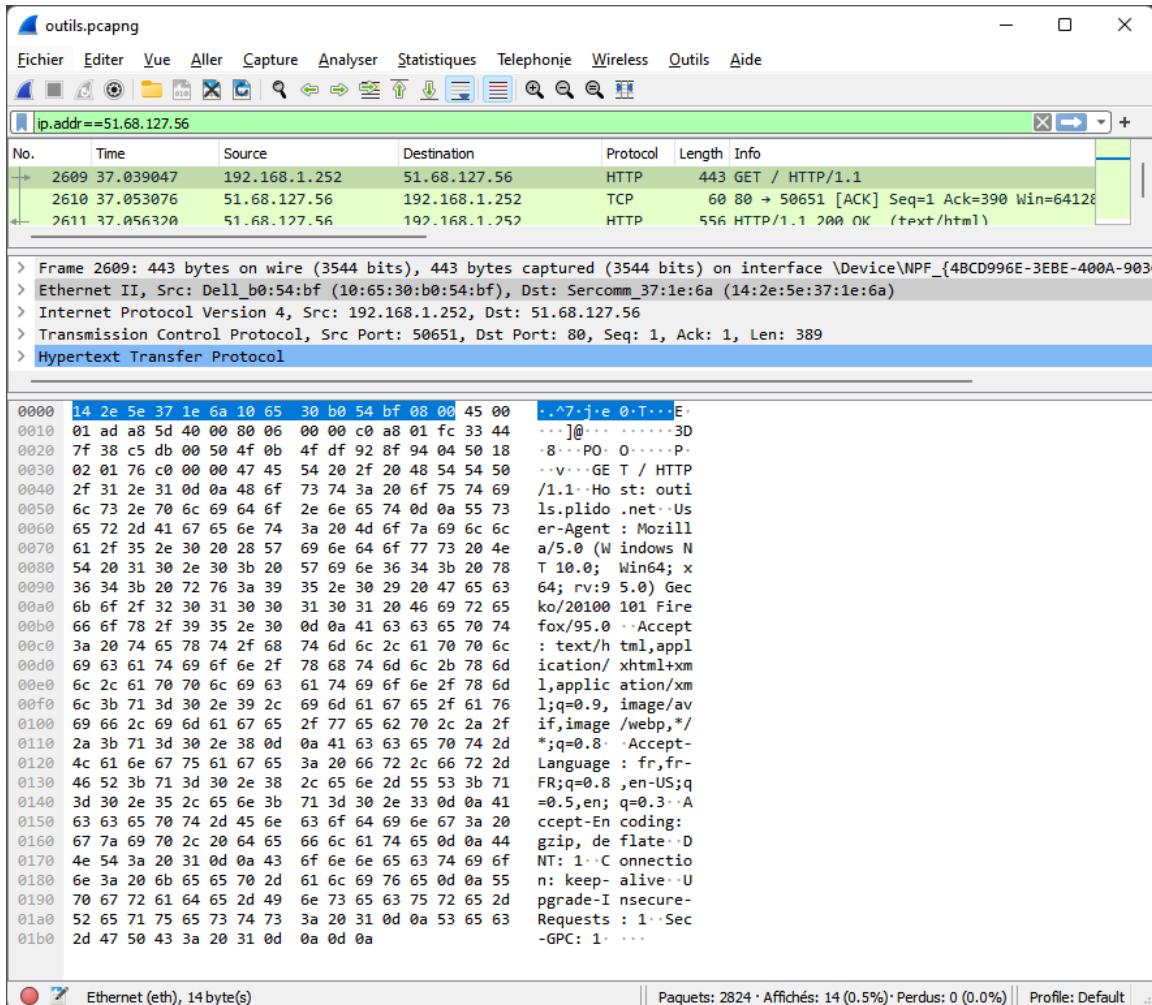


FIGURE 3.4 – Content of the frame carrying the HTTP GET request

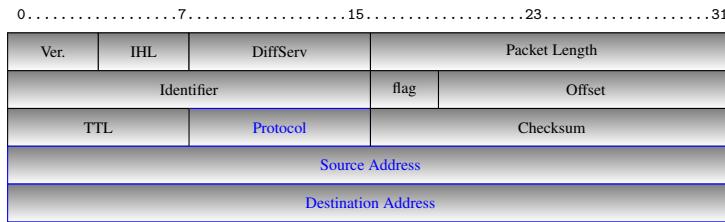


FIGURE 3.5 – Format of an IPv4 header

(cf. figure 2.3 on page 28), but corresponding to the order of encapsulations in the frame. How Wireshark could arrive at such a result.

Ethernet

Wireshark receives a frame from the network **Ethernet** or **Wi-Fi**². The format of an Ethernet frame is defined by the standard **IEEE 802.3**. The header contains three fields :

- 6 bytes for the MAC address of the destination,
- 6 bytes for the source address,
- 2 bytes for the higher level protocol. Thus the value 0x0800 indicates IPv4 and 0x86dd IPv6.

Following the principle of the ISO reference model, the addresses are those of adjacent nodes, i.e. connected to the same Ethernet or Wi-Fi network.

In our case, the top level protocol is therefore an IPv4 packet and Wireshark can continue to analyze this. Formally it is data from the Ethernet frame, but it can be understood as an IPv4 packet.

Question 3.3.11: My address

In the example, figure 3.4 on the preceding page, what is the Ethernet address of the machine sending the frame ?

IPv4

The IPv4 packet format defined in the [RFC 791](#) has changed very little since its publication in 1981. Figure 3.5 shows this format. Without going into detail, the fields :

- Addresses **source** and **destination** will contain the IPv4 addresses on 32 bits of the end equipment. Intermediary equipments, called routers are in charge of copying the packet to its destination.
- The field **protocol** designates the upper layer, the value 6 corresponds to **TCP** and 17 to **UDP**.

Question 3.3.12: hop by hop

Does the Ethernet address 14:2e:5e:37:1e:6a found in the packet 3.4 on the preceding page correspond to the Ethernet address of the recipient of the packet ? What does it correspond to ?

2. For the Wi-Fi network, it transforms the format into that of an Ethernet frame for a more compact display

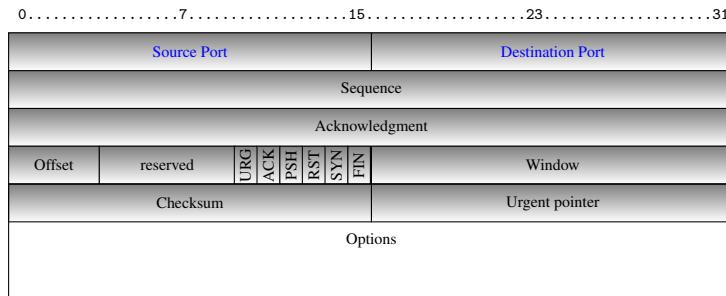


FIGURE 3.6 – Format of a TCP header

TCP

Wireshark, from the protocol field 0x06, determines that the IP data following the header is a TCP message, so it can continue disassembling the frame. The format of the TCP header is given in figure 3.6. The port numbers determine which application is used. If a client is going to use any number (50651 in the figure 3.4 on page 44), the servers will use numbers known by all. Thus, the Web servers will be assigned the value 80. They can choose others, as we saw when building the URL.

Wireshark knows this list of well-known port numbers and can continue to parse the frame as HTTP.

Without going into details, we can also notice a series of binary values which are used for example to open or close a TCP connection. If we go back to the opening phase of the connection (cf. figure 3.3 on page 41), the connection is opened by :

- the transmission by the client of a TCP message with the SYN bit set,
- the transmission by the client of a TCP message with the SYN bit set,
- the client responds by returning a message with the ACK bit set.

These three messages, which do not contain any data, are used to synchronize the initial value of the sequence field at each end of the connection.

Question 3.3.13: Closing the connection

Using the figure 3.4 on page 44 or your Wireshark captures, what are the messages involved in closing the connection ?

3.4 Do it yourself

A Web server can be written in Python thanks to the module **Flask**. The program `simple_server.py` allows to create a Web server on its computer.

Listing 3.1 – simple_server.py

```

1 from flask import Flask
2 app = Flask("MyFirstWebServer")
3
4 @app.route('/', methods=['GET'])
5 def hello_world():
6     return "HelloWorld"
7

```

```
app.run(host="0.0.0.0", port=8080)
```

This script requires some explanation :

- The import line 1 includes the Flask object from the flask module.
- On line 2 an instance of a Flask object, i.e. a web server, is created. A name is associated to it for debugging purposes.
- Line 4 contains the most delicate part of the script. @ is a decorator that is used in python to add properties to a function. Here we associate a URI path and a REST method to the function which is then defined. This way, when the Flash server receives a GET request on this URI path, it will call the function hello_world.
- The function hello_world simply returns a text that the browser will display.
- the server is launched, line 8, by calling the method run. It will wait on all interfaces (wildcard address 0.0.0.0) and on port 8080.

To launch the server, you must first install the Flask module with Indexipip.

```
# pip3 install Flask
Collecting Flask
  Downloading Flask-2.0.2-py3-none-any.whl (95 kB)
    |
    |  95 kB 4.3 MB/s
Collecting Jinja2>=3.0
  Downloading Jinja2-3.0.3-py3-none-any.whl (133 kB)
...
...
```

Once the package is installed, simply run the program :

```
# python3.9 simple_server.py
* Serving Flask app 'My First Web Server' (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production
deployment.
Use a production WSGI server instead.
* Debug mode: off
* Running on all addresses.
WARNING: This is a development server. Do not use it in a production
deployment.
* Running on http://192.168.1.53:8080/ (Press CTRL+C to quit)
127.0.0.1 - - [14/Dec/2021 21:06:55] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2021 21:06:59] "GET /favicon.ico HTTP/1.1" 404 -
```

Question 3.4.1: loopback

What URI should you enter in your browser to access this server locally.

Question 3.4.2: Server name

Using Wireshark, you can determine in the response the values of the HTTP options IndexContent-Type and Server.

4. Modbus

4.1 Introduction

Modbus appeared in 1979 at a time when the Internet did not exist yet! It is still very popular in the industry. Originally Modbus was built on a serial bus **RS-485** which connected different equipments called (see figure 4.1) :

- secondary or slaves and
- a primary, also called master, which manages communications.

Each secondary has a unique number or address. The addresses are between 1 and 247. The primary does not need an address since all communication is with it.

Youtube



The primary sends a request to a secondary and the secondary replies to the primary. Direct communication between two secondaries is not possible.

4.1.1 Registers

A Modbus device can take two types of data through registers :

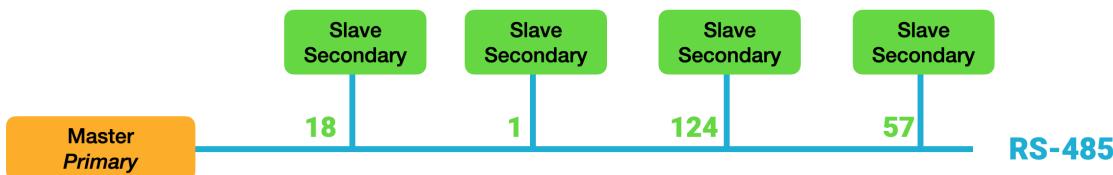


FIGURE 4.1 – Modbus wired architecture



FIGURE 4.2 – Modbus frame

- relays that can take a binary value "on" or "off". If the primary can change the state and, of course, read it, it is called a *coil*. If the binary value can only be read it is a *discrete input*.
- 16-bit registers. They are used to represent a value such as an electric current, a temperature, a rotation speed,... Similarly, if one can only read the value at is called an *input register* otherwise, if it can also be modified by the primary, it is called an *holding register*.

A Modbus device can have up to 10 000 registers of these four categories.

4.1.2 Protocol

Modbus is a query/response protocol. The primary sends a request to the address of a device to read or write one of its registers.

A Modbus frame is a sequence of characters starting with a byte with the address of the secondary followed by a command or function code specific to each register category :

- 1 to read a coil,
- 2 to read a discrete input,
- 3 to read a holding register,
- 4 to read an input register,
- 5 to write a coil,
- 6 to write a holding register.

The rest of the frame contains the data and then a Cyclic Redundancy Check (CRC) to validate that there is no transmission error in the frame. The data part can be different in the request and the response. For example, to read a holding register, the request contains the address of the first register to be read and the number of registers to be read and the response contains the number of data transmitted followed by their values. To write to a register, the data in the frame will be the address of the register and the data to write.

4.1.3 Example : XY-MD02

Let's take a closer look at a concrete example. We will use a temperature and humidity sensor, the **XY-MD02** (see figure 4.3 on the next page) whose specifications are easily accessible via an Internet search.

The green part is composed of four connectors whose meaning is indicated on the label. The two left connectors constitute the bus named A+ and B- and the two right connectors allow to supply the equipment with a voltage between 5V and 30V.

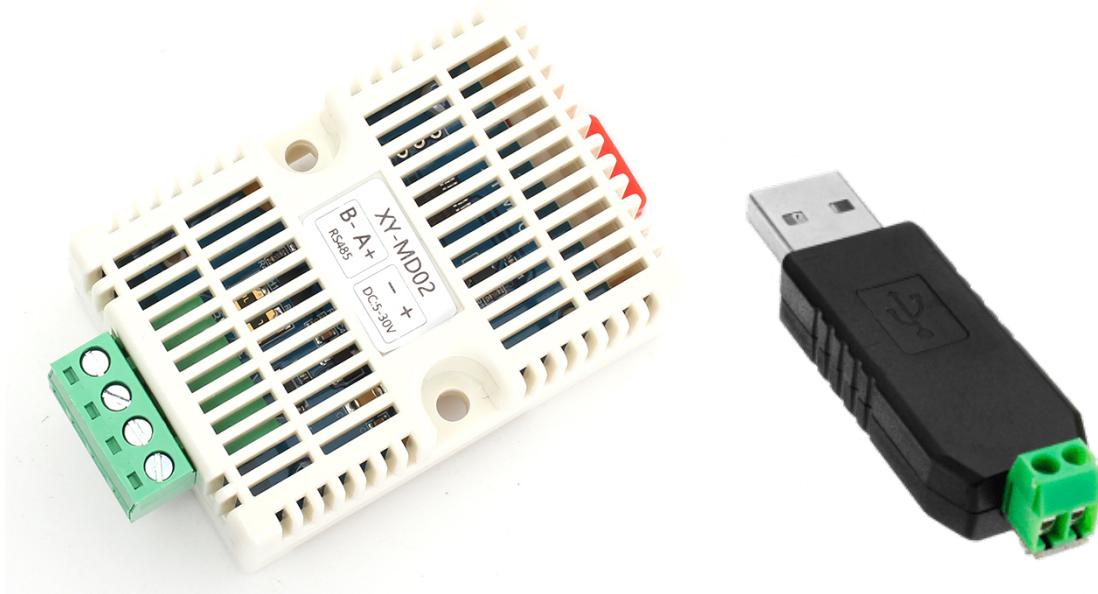


FIGURE 4.3 – XY-MD02 and USB/RS-485 Adapter

An adapter **USB/RS-485** (see figure reffig-XYMD02) is connected to a computer. It contains the two connectors A+ and B- of the RS-485 bus. The computer plays the role of primary which will interrogate the temperature sensor.

The program **QModMaster**¹ (see figure 4.4 on the facing page) allows to query or write the registers of the secondaries. In the left window you can access the registers of a secondary. The right window shows the traffic on the RS-485 bus.

In order for the primary to be able to connect to the secondary, in addition to the name of the serial port (here COM3), it needs several pieces of information that can be found in its documentation :

- the transmission speed on the bus (here 9 600 bit/s) and the coding of the transmitted characters (here 8 bits without parity bit and one stop bit)²
- the address of the secondary on the bus.

The documentation also gives the nature of the registers and their coding. The table 4.1 on the next page takes up the definition of *Input Registers*. These are registers that can only be read. The specification indicates that the temperature is stored in register 1 over a length of 2 bytes, i.e. the entire length of the register.

The documentation indicates that the secondary has the address 0x01 on the RS-485 bus. It only remains to send a Modbus request to read this register. The trace window on the right on the figure 4.4 on the facing page gives the exchanges. We will analyze the last two lines. The first one

Youtube



1. <https://sourceforge.net/projects/qmodmaster/>

2. <https://en.wikipedia.org/wiki/8-N-1>

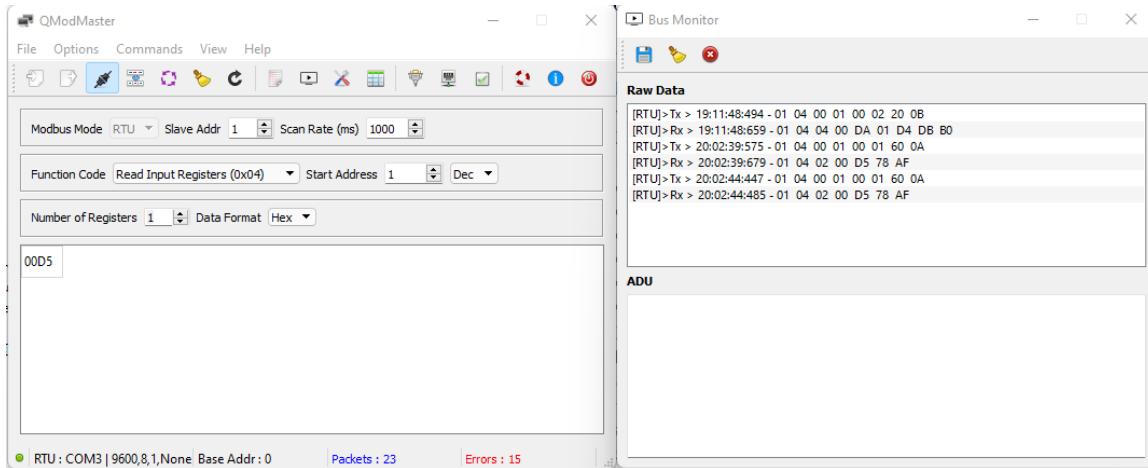


FIGURE 4.4 – QModMaster with message dump

Register Type	Register Address	Register Contents	Bytes
Input register	0x0001	Temperature	2
	0x0002	Humidity	2

TABLE 4.1 – Input Register of an XY-MD02

indicates the content of the request and the last one the answer of the secondary :

01 04 00 01 00 01 60 0A
01 04 02 00 D5 78 AF

The request starts with the address of the secondary (01), then the action (04) to read an *Input Register*, followed by the address of the register (00 01) and the number of registers to read (00 01). The request ends with the CRC validating the integrity of the frame (60 0A).

The response also contains the address of the secondary (01) and the action, followed by the size of the response in bytes (02) and the requested result (00 D5).

Question 4.1.1: Humidity

Looking at the exchanges in figure 4.4 what is the measured value for humidity ?

It remains to be able to interpret this value. The documentation indicates that the value is in tenths of degrees and that the unit is Celsius. By converting 00 D5 into decimal, we obtain 213, that is 21.3°C.

Question 4.1.2: Evolution of temperatures

Looking at the exchanges in figure 4.4 what is the evolution of the temperature over time ?

In summary, we can see that it would be very difficult to operate the equipment without documentation to know : the speed of the RS-485 bus, the address of the secondary, the addresses of the

Register Type	Register Address	Register Contents	Bytes
Holding register	0x0101	Device Address	2
	0x1202	Bit rate : • 0 : 9600 • 1 : 14400 • 2 : 19200	2
	0x0103	Temperature correction -10°C - 10°C	2
	0x0104	Humidity correction -10%RH - 10%RH	2

TABLE 4.2 – Holding Register of an XY-MD02

registers used and their contents and the coding of the information in the registers and the units used. Therefore, there is a low degree of interoperability, the two entities must agree on a large number of parameters.

Question 4.1.3:

What is the moisture content at the time of measurement ? The documentation states that it is a percentage with an accuracy of one tenth of a percent.

We could communicate with the object using the default parameters. But to insert it in a bus, it is necessary to be able to modify certain parameters. The baud rate and the byte coding must be the same, the secondaries cannot have the same address on the bus.

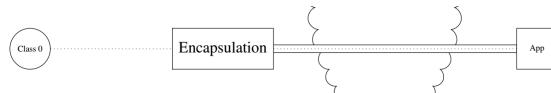
The XY-MD02 has also *holding register* allowing to parameterize it as shown in the table 4.2

4.1.4 IP Gateway

It is possible to extend the scope of a Modbus network by adding an IP gateway. This corresponds to the third method of interconnection of the figure 1.3 on page 22.

The gateway, connected on the bus where the

secondaries remain connected, has an IP address. The primary opens a TCP connection with the gateway and sends its requests. The gateway copies the data on the bus. Conversely, the responses of the objects are returned to the gateway which sends them to the primary using the TCP connection. The figure 4.5 on the next page illustrates the exchanges. We note the opening of TCP connection which is done at the starting of the primary which remains active for all the exchanges. We can also notice that the TCP messages are acknowledged. The figure 4.6 on the facing page shows the fields common to the format on the RS-485 bus and in IP packets.



As in the previous example of the temperature sensor, the specifications of the **electric counter** are necessary to understand the meaning of the usable registers. The counter encodes its values on 32 bits floating numbers whose encoding is specified by the **IEEE 754** standard. These values on 32 bits must be coded on two consecutive registers, hence the incrementation of 2 in 2 that we find on

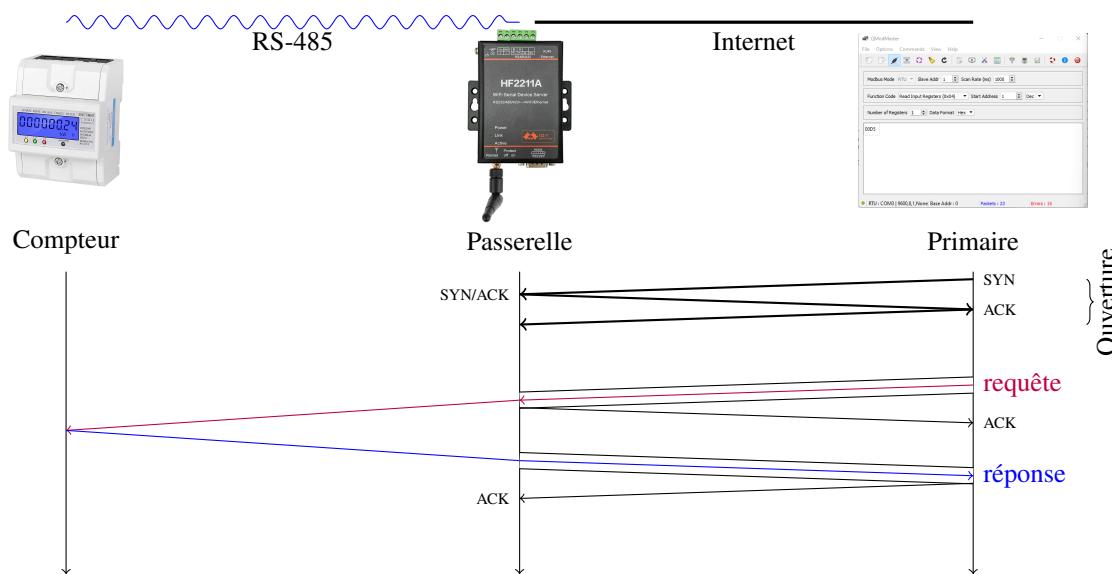


FIGURE 4.5 – Gateway between the Internet and Modbus.

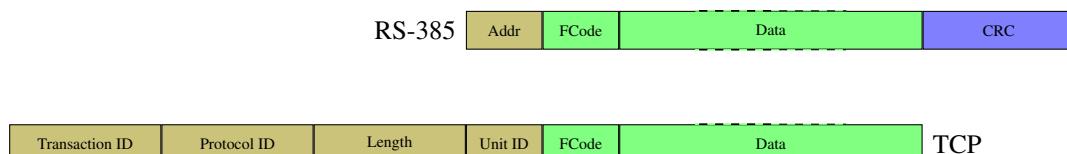


FIGURE 4.6 – Modbus messages on RS-485 bus and on IP/TCP

Register Type	Register Address	Register Contents	Unité	Format
Input register	0x0000	Voltage phase A	V	IEEE 754
	0x0002	Voltage phase B	V	IEEE 754
	0x0004	Voltage phase C	V	IEEE 754
	0x0008	Intensité phase A	A	IEEE 754
	0x000A	Intensité phase B	A	IEEE 754
	0x000C	Intensité phase C	A	IEEE 754
	0x0010	Puissance Totale	KWh	IEEE 754
	0x0012	Puissance phase A	KWh	IEEE 754
	0x0014	Puissance phase B	KWh	IEEE 754
	0x0016	Puissance phase C	KWh	IEEE 754
	0x0036	Fréquence	Hz	IEEE 754

TABLE 4.3 – some *Input Register* of the electric meter

the table 4.3, the counter being able to measure three electrical phases.

Wireshark can capture a request having circulated on the Ethernet network, primary side (cf. figure 4.7 on the facing page).

0000	98 d8 63 62 29 49 10 65 30 b0 54 bf 08 00 45 00	..cb)I.e0.T...E.
0010	00 34 db ef 40 00 80 06 00 00 c0 a8 01 fc c0 a8	.4..@.....
0020	01 57 e2 c9 01 f6 16 90 37 98 5d 57 a0 fa 50 18	.W.....7.]W..P.
0030	02 01 84 ca 00 00 00 0a 00 00 00 06 1c 04 00 00
0040	00 02 ..	

We find the encapsulations of the Ethernet, IP and TCP protocols, followed by the TCP data. They consist of three fields which do not exist in the request circulating on the RS-485 bus :

- the transaction number on two bytes incremented at each request,
- the protocol version on two bytes,
- the length in bytes of the transaction.

The following fields are identical to those of the frame on the RS-485 bus :

- the address of the secondary on one byte, here 0x1c or 28,
- the nature of the request : 0x04 to read an *input register*,
- the register to be read on two bytes, here 0x0000 corresponding to the voltage on phase A,
- the number of registers to read, here 2 to obtain the 32 bits of the value.

The primary receives the following response :

0000	10 65 30 b0 54 bf 98 d8 63 62 29 49 08 00 45 00	.e0.T...cb)I..E.
0010	00 35 c9 75 00 00 40 06 2c aa c0 a8 01 57 c0 a8	.5.u..@.,....W..
0020	01 fc 01 f6 e2 c9 5d 57 a0 fa 16 90 37 a4 50 18]W....7.P.
0030	44 70 e1 6e 00 00 00 0a 00 00 00 07 1c 04 04 43	Dp.n.....C
0040	69 9e 4a	i.J

After the protocol encapsulations of Ethernet, IP and TCP, we find the following data :

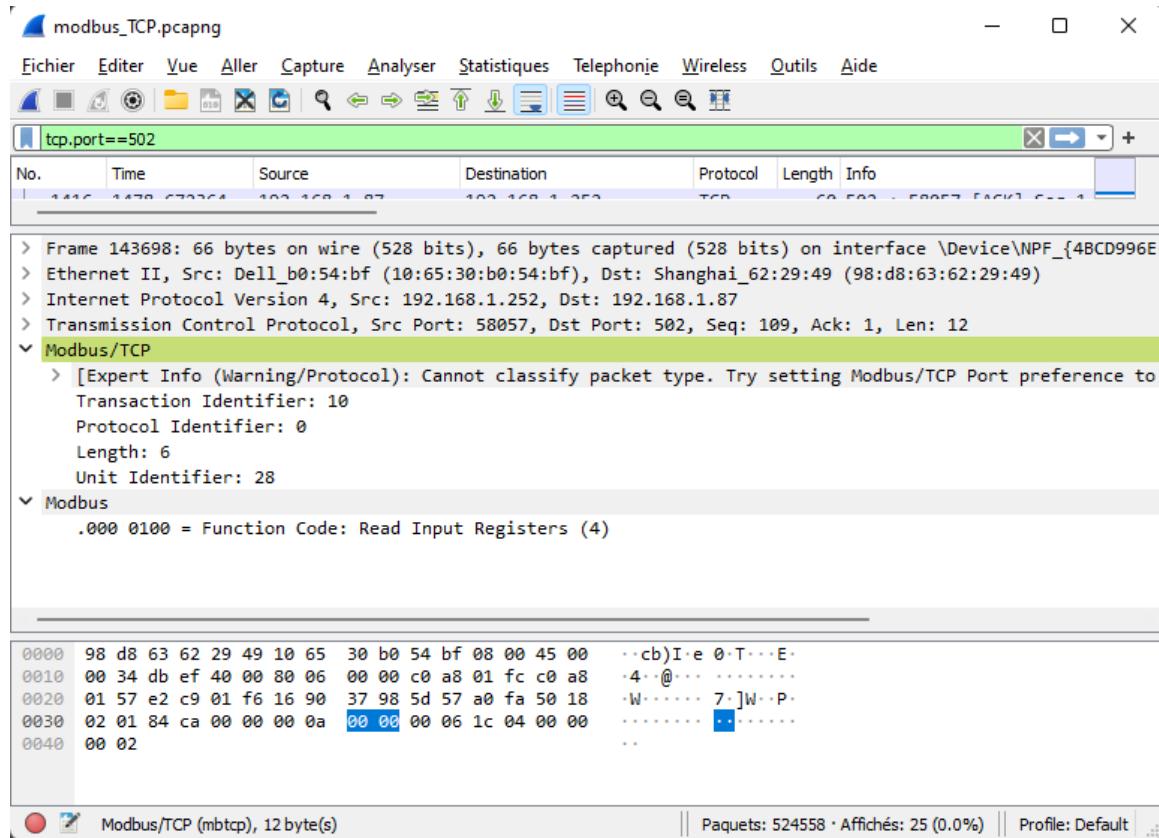


FIGURE 4.7 – Capture with **Wireshark** of a frame containing a Modbus message.

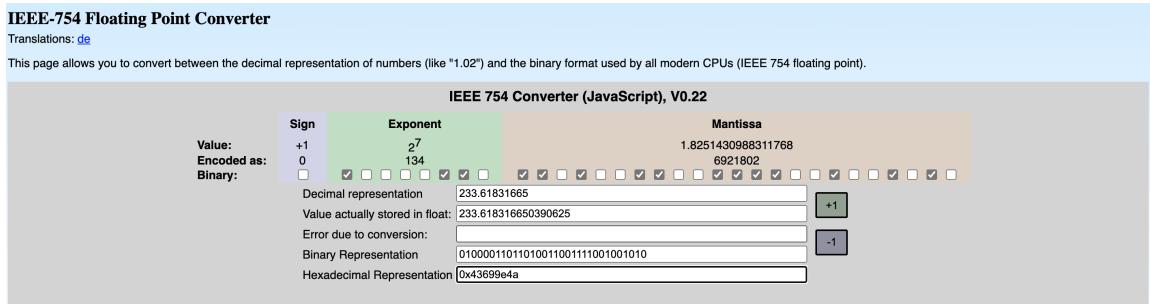


FIGURE 4.8 – Conversion of a floating number

- the transaction number that corresponds to the one used in the previous request. This makes it possible to make the link between the two messages that could have been lost in case of packet loss on the Internet network,
- the protocol version,
- the length of the response, here 7 bytes,
- the address of the secondary school that responded, here 28,
- the nature of the request,
- the number of bytes returned, here 4,
- and the value of the two registers 0x43699e4a which corresponds to a floating-point number as represented by the IEEE 754 standard. There are many sites on the Internet that allow the conversion of footnoteurl <https://www.h-schmidt.net/FloatConverter/IEEE754.html>. As shown in figure 4.8, we obtain the value 233.61831665 which corresponds well to a voltage offered by an electrical network.

Question 4.1.4: Modbus TCP request

Let the given exchange figure 4.9 on the facing page correspond to a Modbus request and a response. What is the port number used by Modbus TCP.

Question 4.1.5: Modbus TCP request

Continuing the traffic analysis, which register value is requested.

Question 4.1.6: Modbus TCP response

By analyzing the following packet, how can we verify that the response can match the previous request.

Question 4.1.7: Modbus TCP response

What value is returned. Is this consistent ?

0000	98 d8 63 62 29 49 10 65 30 b0 54 bf 08 00 45 00	..cb)I.e0.T...E.
0010	00 34 db f3 40 00 80 06 00 00 c0 a8 01 fc c0 a8	.4..@.....
0020	01 57 e2 c9 01 f6 16 90 37 b0 5d 57 a1 14 50 18	.W.....7.]W..P.
0030	02 01 84 ca 00 00 00 0c 00 00 00 06 1c 04 00 366
0040	00 02	..
0000	10 65 30 b0 54 bf 98 d8 63 62 29 49 08 00 45 00	.e0.T...cb)I..E.
0010	00 35 8b 15 00 00 40 06 6b 0a c0 a8 01 57 c0 a8	.5....@.k....W..
0020	01 fc 01 f6 e2 c9 5d 57 a1 14 16 90 37 bc 50 18]W....7.P.
0030	44 70 f1 f0 00 00 00 0c 00 00 00 07 1c 04 04 42	Dp.....B
0040	47 e9 5b	G. [

FIGURE 4.9 – Capture to be studied.

5. ARCHITECTURE FOR IOT

5.1 Introduction

The objects are characterized by a limited processing capacity and by a reduced energy consumption to preserve the autonomy imposed by a battery power supply. However, the most consuming activities for an equipment are the transmission and reception of data. To maximize the autonomy of equipment, it is necessary to review all the protocols, but by modeling them on existing architectures to ensure compatibility.

The figure 5.1 on the next page shows a number of protocol adaptations, at different layers of the model, capable of adapting to the characteristics of the constrained objects. In the following chapters, we will come back to these technologies, starting from the data representation and going to the lower layers.

Youtube



5.2 Topologies

Networks for the Internet of Things can be divided into two categories : **mesh** and **star** topologies.

Mesh Networks

Mesh networks, such as the IEEE 802.15.4 family, are an adaptation of a Wi-Fi access protocol to preserve energy. The transmission range is limited to 50 meters to limit energy consumption, and therefore messages must be relayed by other nodes to reach their destination.

The data rate is a few hundred kilobits/s and the frame size is a few hundred bytes.

These networks are good at carrying IoT data, but the routing protocol, as well as frame relaying, consumes the objects' energy.

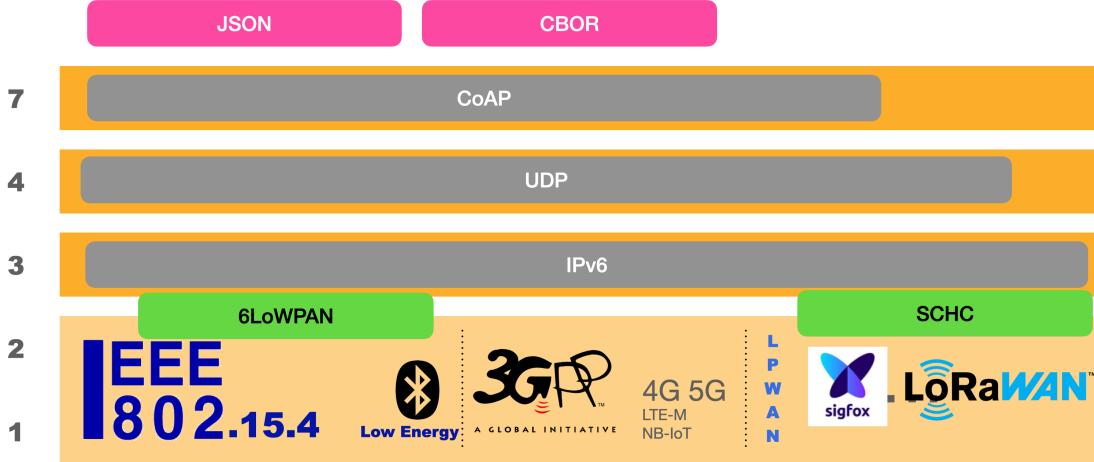


FIGURE 5.1 – IoT protocol stack

Star Network

The topologies in **star** do not require such routing mechanisms. All communications are with a central point that relays information to the destination.

The progress made in signal processing allows to extend the transmission range at low power. This family of networks is called low power wide area networks (LPWAN) like **Sigfox**, **LoRaWAN**, or even on the cellular side with evolutions of the **4G** standard and a more complete integration in **5G**. The [RFC 8376](#) gives, in English, an overview of these techniques.

With a transmission power of 25 mW, it is possible to communicate over a distance of 3 km in an urban environment and 20 km in a clear environment. The LPWANs are compatible with class 0 devices because they do not require the installation of an IP stack. The figure below describes a typical architecture for LPWANs.

The device sends raw data over the radio network. The radio signal is picked up by one or more radio gateways, and the frame is sent to a network gateway (LoRaWAN Network Server (LNS) for networks **LoRaWAN**, and Service Capability Exposure Function (SCEF) for networks 3GPP).

The owner of the device has associated the device with a connector in the LPWAN Network GateWay (NGW) which can be a URL, a Message Queuing Telemetry Transport (MQTT) broker address or a web socket. When the device sends data, it is connected to the application through this tunnel.

Some technologies such as LoRaWAN or Sigfox use unlicensed bands, imposing a duty cycle of 0.1 to 10 percent depending on the channel to ensure fairness between nodes, thus preventing one device from monopolizing the transmission channel. Since this restriction also applies to the provider's antenna, communication between the network and the device is significantly limited.

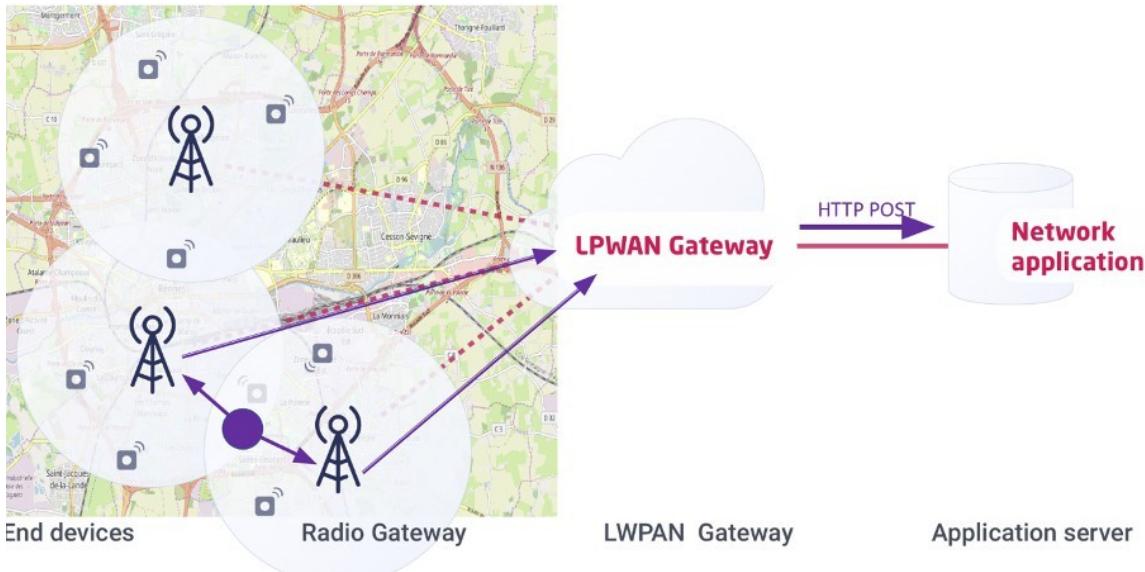


FIGURE 5.2 – Simplified LPWAN architecture

5.3 Layers 1 and 2

Concerning layer 2, the goal is to save energy during transmissions. We can already say goodbye to Ethernet because it would require the use of wired infrastructure and therefore we could not place the objects where we want, especially if they move. The communications by radio waves are privileged.

For the Internet of Things, Wi-Fi is also too power-hungry. It is therefore preferred to an evolution called **IEEE 802.15.4** which uses its operating principle but adapts it to a low speed and to small frames. In particular to save energy, the range is reduced to about ten meters and it is generally necessary to use relays to reach a destination.

Bluetooth has been adapted for objects with a low consumption Bluetooth Low Energy (BLE).

On the cellular side, protocols are evolving to take objects into account. The 4G standard has integrated lower speed communications. The 5G will include a class allowing communications with objects energy efficient and reducing latency.

5.4 IP and adaptation layers

But, as seen in Figure 5.3 on the facing page IPv6 implies larger headers, which is troublesome because Layer 2 networks carry smaller frames. An **adaptation layer** between the IP layer and Layer 2 is needed since Layer 2s designed for the Internet of Things cannot naturally carry large packets. Two actions are implemented : **compression** of header size to reduce their impact, and **fragmentation** to break the packet into smaller frames if the first measure is not sufficient.

There are two main families of adaptive layers :

- **LoWPAN RFC 4944, RFC 6282**, which will integrate a mechanism of compression of the header IPv6 and of fragmentation to send a large packet divided into small frames. Indeed, in a mesh network, it is not possible to deprive oneself of information provided by the IP layer

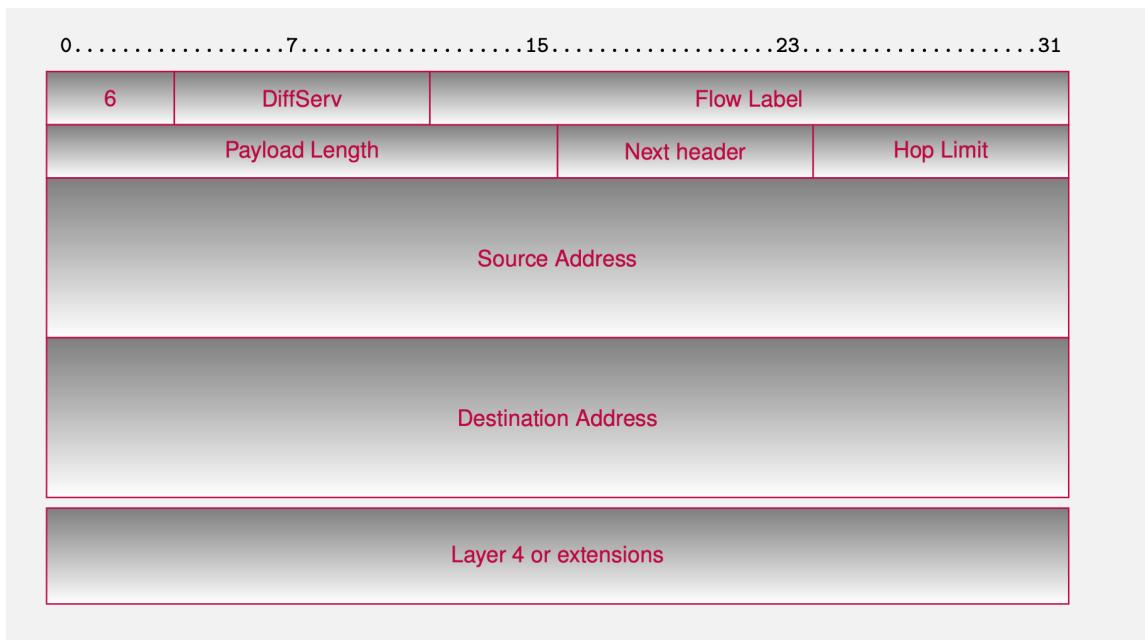


FIGURE 5.3 – IPv6 header format

because the intermediate nodes need it to route the message to the recipient. 6LoWPAN is stateless and compresses all IPv6 headers without configuration.

- Static Context Header Compression (SCHC) (pronounce chic) [RFC 8724](#) will impose rules describing the message header and will send the rule number in place of the header. Compression is much more important and can involve several protocol layers. However, to implement it, you need to have an idea of the flows that will circulate on the network. SCHC is specified for networks in **star** and more particularly for **LPWANs!** (**LPWANs!**).

5.5 Implementation of REST

Above we had seen that as HTTP was the dominant protocol, TCP was also. But for the IoT this is not optimal. Indeed TCP/HTTP are complex protocols that require a lot of memory. To reduce the impact of the protocol stack, the IETF has defined a new protocol called CoAP which requires only a few Kilobytes to work. Constrained Application Protocol (CoAP) is based on UDP which simplifies again the implementation.

To continue in the integration of the objects in the Internet, the protocol CoAP [RFC 7252](#) replaces HTTP. It takes over the naming mechanism, the use of resources, and the handling primitives between a client and a server.

The processing capacity of the sensor and its power supply are often very limited. The great strength of CoAP is to be :

- easy to implement. The implementations of CoAP require little memory ;

—

As a result, CoAP will manipulate resources, identified by URIs. It is thus possible to anchor the data provided by the objects in the current ecosystem of communications between computers,

strongly structured around the REST principles.

Security, especially data encryption, also follows the same paths as the traditional Internet. There is an encryption above UDP which, like HTTPS, encrypts the exchanges.

5.6 Data representation

For data structuring, XML is not used because it is too talkative. JSON is much more effective to transport structured information. There is a binary equivalent that we will see later : CBOR, which is much more efficient, and simple to implement, and compatible with JSON.

5.7 Alternatives to REST

It does not have to implement all the protocols defined by the IETF. It is also possible to integrate protocols specified for a business.

For example, the electric meter **Linky** that all French people know implements only a part of it. Instead of using CoAP, the electricians use their own applications following the standard DLMS/Cosem. This one is based on UDP then IPv6 and **6LoWPAN** and finally on a variant of **IEEE 802.15.4** adapted to transport the information on the electric cables.



6. THE REPRESENTATION OF DATA

6.1 Introduction

Sending data over a network is not as simple as it seems.

There is a difference between the format used to store data in the computer's memory and the one used to send it to another machine. Indeed, each machine has its own representation often linked to the capacities of their processor. This is especially true for numbers. They can be stored on a more or less important number of bits or can be represented in memory in an optimized way to accelerate their treatment.

On the other hand, the representation of (unaccented) character strings is relatively uniform because it is based on the ASCII code which is the same for all computers. A basic text is easily understandable by all machines. A solution would therefore be to use only strings of characters.

For example, if we want to send the integer with the value 123, there are several possible representations :

- send a string "123" containing the digits of the number ;
- send the binary value 1111011.

We see that just to transmit a simple value stored in the memory of a computer, there are several options and obviously for this value to be interpreted in the right way, it is necessary that both ends have agreed on a representation.

When you want to transmit several values, i.e. when you have structured data, other problems arise.

For example : what is the size of the blocks we are going to transmit ? How to indicate the end of the transmission ? For a string, how to indicate that it ends ? Another example : if we want to transmit "12" and then "3", how do we make sure that the other end does not include "123" ?

Youtube



In order for the transmission to take place correctly, the sender and receiver must adopt the same conventions. When it is a question of a set of data, it is necessary to be able to separate them. With spreadsheets, a first method is possible with the notation Comma Separated Values (CSV). As its name indicates, the values are separated by commas. The values are represented by strings. The texts are differentiated from the numerical values by the use of quotation marks. Thus, 123 will be interpreted as a number and "123" as a text.

If this representation is adapted to spreadsheets, it is relatively poor because it only allows to represent values on rows and columns. For Web uses, it was necessary to find a more flexible format allowing to represent complex data structures. Obviously, as nothing is simple, there are several of them and applications exchanging data will have to use the same one.

We can see that sending the string is not enough, it must be formatted so that the receiver can find the type of the transmitted data, so that a number is not interpreted as a string, so that a string remains a string even if it contains only numbers.

6.2 The serialization

Under this barbaric name hides the method used to transmit data from one computer to another. A data can be simple (a number, a text) or more complex (an array, a structure...). It is stored in the computer's memory according to a representation that is specific to it. For example, the size of integers can vary from one processor technology to another, the order of bytes in a number can also be different (little and big endian). For complex structures such as arrays, the elements can be stored in different locations in the memory.

Serialization consists of transforming a data structure into a sequence that can be transmitted over the network, stored in a file or a database. The opposite operation, consisting in locally reconstructing a data structure, is called deserialization.

There are several formats for serializing data. They can be binary but those generally used are based on character strings. Indeed, the representation ASCII defining the basic characters and coded on 7 bits is common to all the computers. The other advantage of the ASCII code is that it is easily readable and simplifies the development of programs.

Wikipedia gives this table (cf. figure reffig-ASCII) of the codes dating from 1972 (an eternity in computing) and recolored by us.

The characters in orange are not printable. They are used to control data communication or to manage the display by returning to the line. They can be recognized because the binary sequence starts with 00X XXXX. One recalls that the ASCII code is on 7 bits ; the additional bit (parity bit) leading to 1 byte was used to detect transmission errors. The values from 0x30 to 0x39 code the digits from 0 to 9.

Hexlify

In Python, there is the module **binascii** very practical which makes it possible to convert a binary sequence into a character string or conversely :

- `hexlify` takes an array of bytes and converts it to a more specialist-readable hexadecimal string. This allows you to view any sequence of data.
- `unhexify` does the opposite. It takes a string and converts it to an array of bytes. This can make programming easier for you because in your code it is easier to manipulate strings.

USASCII code chart												
b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁						
Row		Column		0	1	2	3	4	5	6	7	
0	0	0	0	0	NUL	DLE	SP	0	@	P	'	p
0	0	0	0	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	8	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(8	H	X	h	x
1	0	0	1	9	HT	EM)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[k	{
1	1	0	0	12	FF	FS	.	<	L	\	l	/
1	1	0	1	13	CR	GS	-	=	M]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	S1	US	/	?	O	—	o	DEL

FIGURE 6.1 – ASCII character encoding

In the following, we will use these functions to manipulate identifiers. For example, this piece of code illustrates the use of these functions :

```
mac = lora.mac()
print ('devEUI: ', binascii.hexlify(mac))

# create an OTAA authentication parameters
app_eui = binascii.unhexlify('70 B3 D5 7E D0 03 3A E3'.replace(' ',''))
```

As we will see later, the function `lora.mac()` returns an array of bytes. The function `hexlify` in the following line converts it to a string for a cleaner display.

Conversely, we must assign a binary sequence to the variable `app_eui`. We put this hexadecimal sequence into a string. Spaces offer more readability. They are removed by the `replace` method and the result is converted into binary thanks to `unhexify`

6.3 Base64

The passage from a binary sequence to an ASCII character string representing the values leads to a doubling of the size. Each block of 4 bits will lead to produce a byte corresponding to the character of a digit or a letter from A to F. The rest of the codes are not used.

The encoding **base64** offers a better performance by using 64 bits to encode the values. A dictionary maps 64 values to an ASCII character. However, if we want to encode 4 bytes, or 32 bits, we will need 5 blocks of 6 bits, and there will be 2 bits left. The symbol = indicates that 2 bits are added at the end of the coding. So, in our case, it will be necessary to add two symbols = as shown in the figure below :

Note that for small sequences, this coding is not better than the transformation of the hexadecimal sequence into a string. Here, 8 characters are needed to encode 4 bytes.

There are many online tools to make conversions between these different representations, such as the site www.asciitohex.com.

Python module : base64

In Python3, the `base64` module allows to do these conversions. This module is a bit touchy about the data types to use.

```
1 import base64
2
3 val = b"\x01\x0234"
4 ser = base64.b64encode(val)
5 print (ser)
6 print (ser.decode())
7 ori = base64.b64decode(ser)
8 print (ori)
```

which gives as a result of the execution :

```
b'AQIzNA=='
AQIzNA==
b'\x01\x0234'
```

Note that the use of `ser.decode()`, line 6, to transform a string of bytes into a string of characters, i.e. remove the `b` at the beginning, can be used in certain cases.

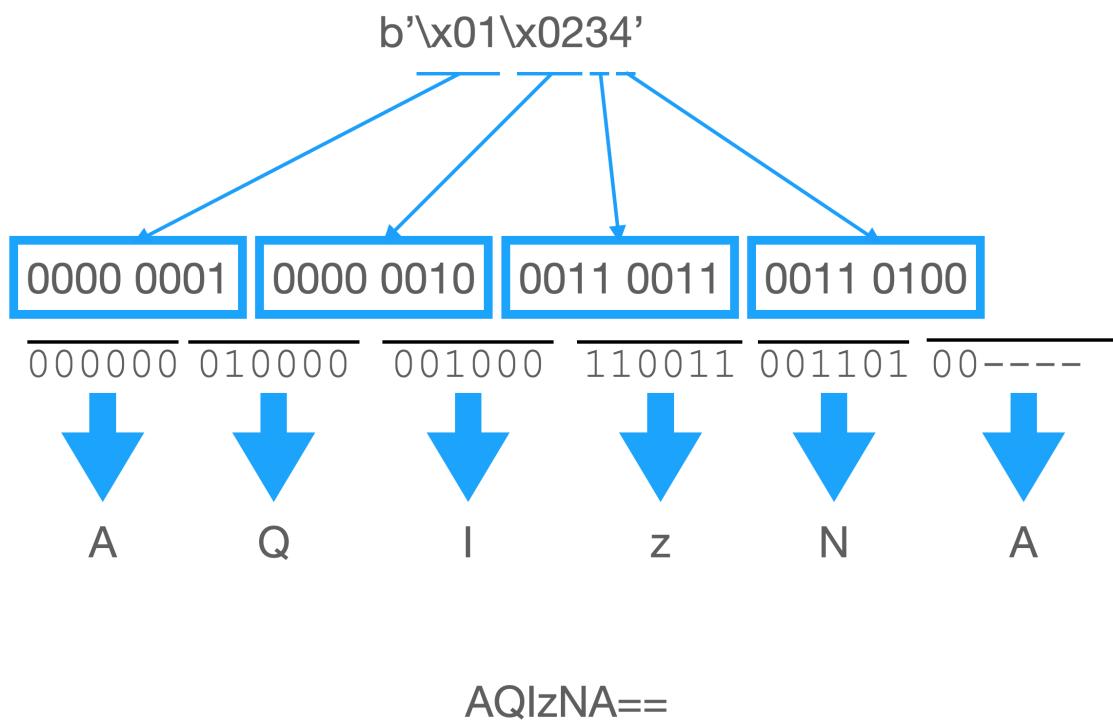


FIGURE 6.2 – Base64 coding of binary data

```
<p>Les s&eacute;rialisations en cha&icirc;nes de caract&egrave;res (par exemple en Python via la commande <span style="font-family: 'courier new', courier;">hexlify</span></span> ou en base64 concernent surtout des donn&eacute;es binaires, mais la donn&eacute;e peut &ecirc;tre aussi structur&eacute;e, par exemple la page d'un tableau. Il faut donc formater le document pour &eacute;viter&nbsp;fusion entre les diff&eacute;rents champs. Le format CSV (<em>Comma-Separated Values</em>) comme son nom l'indique s&eacute;pare les donn&eacute;es par des virgules (<em>comma</em> en anglais). Mais si ce format s'applique bien aux donn&eacute;es d'un tableau, c'est &agrave; dire un tableau de lignes et de colonnes, il est tr&egrave;s limit&eacute; pour repr&eacute;enter une information telle que la mise en forme d'une page web.</p>
```

FIGURE 6.3 – HTML coding of a Web page

```
<img src ="/asset-v1:MinesTelecom+04038+session01+type@asset+block/Mooc-internet_objets.jpg" width="300" type="saveimage" target="[object Object]" />
<br>
<p><strong>Enseignants </strong><br>
<img src ="/asset-v1:MinesTelecom+04038+session01+type@asset+block/Laurent-Toutain.jpg" width="100" type="saveimage" target="[object Object]" />
<br>Laurent Toutain<br>
</p>
<img src ="/asset-v1:MinesTelecom+04038+session01+type@asset+block/Photo_Marc_Girod_Genet.jpg" width="100" type="saveimage" target="[object Object]" />
<br>Marc Girod-Genet<br>
</p>
<img src ="/asset-v1:MinesTelecom+04038+session01+type@asset+block/Kamal_Singh.png" width="100" type="saveimage" target="[object Object]" />
<br>Kamal Singh<br>
</p>
<img src ="/asset-v1:MinesTelecom+04038+session01+type@asset+block/BattonHubert.jpg" width="100" type="saveimage" target="[object Object]" />
<br>Mireille Batton Hubert<br>
<br>
<img src ="/asset-v1:MinesTelecom+04038+session01+type@asset+block/PatrickMaille.jpg" width="100" type="saveimage" target="[object Object]" />
<br>Patrick Maille<br>
<br>
<p><strong>Support p&eacute;dagogique</strong>
</p>
<img src ="/asset-v1:MinesTelecom+04038+session01+type@asset+block/DenisMedalSmall.png" width="100" type="saveimage" target="[object Object]" />
<br>Denis Moalic<br>
```

FIGURE 6.4 – Capture a web page

6.4 HTML

Serialization in strings (for example in Python via the command `hexlify`) or in Base64 concerns mainly binary data. But the data can also be structured, for example the page of a spreadsheet. The document must therefore be formatted to avoid merging the various fields.

HTML, without going into detail, defines a format where fields are marked up with markup. A beginning tag is a keyword between `<>` and, for an ending **tag**, the keyword is preceded by the character `/`. For example, the figure 6.3 with the markup, the first paragraph is formatted this way in the MOOC :

Tags can also take arguments, like the **span** tag in the previous example. Thus, if we look at a Web page, as shown in figure 6.4, the browser is able to analyze it to find the URI it contains. With the `text` tag indicating that it is an image, the client can query the server to display it on the screen. This structured serialization format allows us to implement a feature of REST, i.e. the links between resources.

6.5 XML

If HTML is dedicated to the formatting on screen of textual data and to the navigation on the Web. XML¹ defined by the World Wide Web Consortium (W3C), is a format for exchange between

1. <https://www.w3.org/TR/xml/>

two applications. For example, to exchange student grades between the FUN platform and a course certification authority, one could use the following format :

```
<etudiant>
  <prenom>John </prenom>
  <nom>Deuf </nom>
  <note>18</note>
</etudiant>
```

It is easy by reading the example to find the student's first name, last name and grade. It can be noted that there is no difference between the grade and the student's name. They are characters.

If it is syntactically correct, there is no guarantee that the creator is providing something correct that can be interpreted by another instance. can include a grammar or schema that is used to validate that the information represented in the file is not only syntactically compliant with the language, but also complies with the schema. This schema will describe the expected fields and their type (text, number...). You can access this course if you want to know more about schemas.

From the point of view of the Internet of Things, even if XML could be a good candidate for the exchange of information, it is too heavy a format and therefore energy consuming. We can note that to send a note out of 20 which, in the absolute, would take 6 bits, we transmit `<note>18</note>`, that is to say 15 characters or 120 bits !

6.6 JSON

JSON offers a way to structure information in a more compact way than XML. JSON is emerging as the common language for exchanging information. Originally, JSON was used by **Javascript** to exchange information ; for example, to display in real time the evolution of stock exchange prices or to display dynamic graphs on the user's screen.

Youtube



JSON [RFC 8259](#) is a simple exchange format. It defines 4 data types :

- **Number** : Numbers are composed of digits and can be positive, negative, integers or floats.
- **Text** : The text is delimited by single or double quotation marks.
- **Array** : Arrays are lists of elements separated by commas and surrounded by square brackets.
- **Object** : The object is a list of pairs composed of an **key** and a value. The key is a string and the value can be of any type. The key must be unique within an object, and fully references the value that follows it. The couple key - value is separated by the colon character : . The elements of the object are separated by commas. The object is delimited by braces.

For example, some JSON structures :

- `[1, -2, 0.3, 4e1]` is an array that contains 4 numbers ;
- `[1, "2", "34"]` is an array containing a number and two strings ;
- `[1, [2, 3, "4"]]` is an array of two elements whose second element is also an array of 3 elements ;
- `{"color": [34, 16, 3]}` is an object that contains an element and the value is an array ;

— `{"name" : "bob", "age" : 30}` is an object that contains two elements referenced by the strings (or index) "name" and "age".

The order in which the elements are placed is irrelevant. For example, `{"age": 30, "name": "bob"}` is equivalent to the last example.

This imposes that the index used to access a value must be unique in the object structure : `{"name" : "bob", "name" : "alice"}` is prohibited.

The following listing gives an example of a JSON structure from the [RFC 8259](#). It contains a JSON object with a single key "Image". The value of this key is another structure that contains six elements.

```
{
  "Image": {
    "Width": 800,
    "Height": 600,
    "Title": "View from 15th Floor",
    "Thumbnail": {
      "Url": "http://www.example.com/image/481989943",
      "Height": 125,
      "Width": 100
    },
    "Animated": false,
    "IDs": [11, 943, 234, 38793]
  }
}
```

Key markup is a fundamental element in the structure of data. It is essential to be consistent and to ensure that the sender and receiver agree on the name of the key in order to be able to retrieve the desired information. In the same way, it is necessary to agree on the units of measurement : an interpretation of a measurement in centimeters when it is in pixels can be disastrous ; it is an interoperability problem.

JSON is easily exploitable in other languages. For example in Python, the JSON module can be used to convert a JSON structure that is an ASCII string into a Python internal representation. Arrays are converted into lists and objects into dictionaries.

Listing 6.1 – example_json.py

```

1 import json
2 import pprint
3
4 struct_python = {
5   "Image": {
6     "Width": 800,
7     "Height": 600,
8     "Title": "View\u00a9from\u00a915th\u00a9Floor",
9     "Thumbnail": {
10       "Url": "http://www.example.com/image/481989943",
11       "Height": 125,
12       "Width": 100
13     }
14   }
15 }
```

```

14     },
15     "Animated" : False,
16     "Copyright" : None,
17     "IDs": [0x11, 0x943, 234, 38793],
18     "Title": "Empty picture"
19   }
20
21 print (struct_python)
22 pprint.pprint(struct_python)

24 struct_json = json.dumps(struct_python)

26 print(struct_json)

28 struct_python2 = json.loads(struct_json)

30 pprint.pprint (struct_python2)

```

The program `example_json.py` uses the previous structure. The variable `struct_python` is a Python structure. We can see that the values for "Animated" and "Copyright" are the Python keywords `False` (with a capital F) and `None`. The program displays this value twice with the standard command `print` then with the module `pprint` to have a more readable display. You can notice that the order of display of the keys is different. As "Title" was defined twice, only the last one is kept in the Python structure.

```
{"Image": {"IDs": [17, 2371, 234, 38793], "Height": 600, "Animated": False, "Title": "Empty picture", "Thumbnail": {"Url": "http://www.example.com/image/481989943", "Width": 100, "Height": 125}, "Width": 800, "Copyright": None}}
{"Image": {"Animated": False,
'Copyright': None,
'Height': 600,
'IDs': [17, 2371, 234, 38793],
'Thumbnail': {'Height': 125,
'Url': 'http://www.example.com/image/481989943',
'Width': 100},
>Title': 'Empty picture',
'Width': 800}}
```

Thanks to the function `dumps` of the module `json`, the variable `struct_python` is transformed into JSON. The keywords `False` and `None` are replaced by `false` and `null`. The program displays a string.

```
{"Image": {"IDs": [17, 2371, 234, 38793], "Height": 600, "Animated": false, "Title": "Empty picture", "Thumbnail": {"Url": "http://www.example.com/image/481989943", "Width": 100, "Height": 125}, "Width": 800, "Copyright": null}}
```

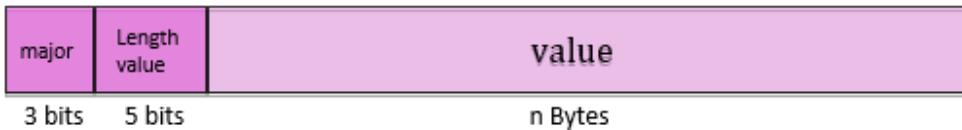
To transform it back, from JSON to Python variable, we use the inverse function `loads` which translates a string into a Python variable.

```
{"Image": {"Animated": False,
'Copyright': None,
'Height': 600,
'IDs': [17, 2371, 234, 38793],
'Thumbnail': {'Height': 125,
'Url': 'http://www.example.com/image/481989943',
'Width': 100},
>Title': 'Empty picture',
'Width': 800}}
```

Other programming languages also have their own libraries for translation.

Compared to XML, JSON is much more permissive and lacks a formalism to describe the structure. The JavaScript Object Notation for Linked Data (JSON-LD) defined by the W3C reinforces the interoperability of JSON by introducing specific keys describing the data structure, a reference to units, etc. We will see these concepts in the rest of the course.

1 Byte



- 0 : Positive Integer**
- 1 : Negative Integer**
- 2 : Byte string**
- 3 : Text string**
- 4 : Array**
- 5 : Object list**
- 6 : Optional semantic tagging**
- 7 : Simple value and float**

FIGURE 6.5 – Definition of major in CBOR

6.7 CBOR

JSON and Concise Binaire Object Representation (CBOR) are both data encoding modes.

JSON introduces a very flexible notation allowing to represent all data structures. The choice of the ASCII makes this format universal and any computer will be able to understand it. But the use of the ASCII does not make it possible to transmit information in an optimal way on a network. When the networks have a reasonable flow, it does not pose a problem. When it comes to the Internet of Things, we must take into account the limited processing capacity of the equipment and the small size of the messages exchanged.

Thus, in ASCII, the value 123 is coded on 3 bytes (one byte by character) while in binary it would occupy only one byte : 0111 1011.

CBOR, defined in the [RFC 8949](#), allows to represent the structures of JSON but according to a binary representation. As we will see later, if CBOR is completely compatible with JSON, it is possible to represent other types of information very useful in the Internet of Things.

The size of the information is reduced and the processing simplified. You have to know how to juggle with the binary representation but it remains basic.

CBOR defines 8 major types which are represented by the first 3 bits of a CBOR structure (cf. figure ?? on page ??). These major types thus have values ranging from 0 to 7 (000 to 111 in binary).

The next five bits contain either a value or a length indicating how many bytes are needed to



encode the value. , this type offers optimizations that allow to reduce the total length of the data structure as we will see later when studying the different major types.

6.7.1 CBOR in Python

The following examples can be tested on your computer with Python3. If an error occurs when defining the module `Indexcbor2`, you must install it on your computer by typing the command :

```
# pip3 install cbor2
```

6.7.2 Type Positive Integer

JSON does not make a difference between numbers, integers, decimals, positive or negative. CBOR reintroduces a distinction to optimize the representation.

The first major type corresponds to the positive integers. It is coded by 3 bits at 0 ; the 5 following bits end the byte and, according to their value, will have a different meaning :

- from 0 to 23, it is the value of the integer to be coded ;
- 24 indicates that the integer is coded on 1 byte which will be coded in the following byte ;
- 25 indicates that the integer is coded on 2 bytes which will be coded in the two following bytes ;
- 26 indicates that the integer is coded on 4 bytes which will be coded in the four following bytes ;
- 27 indicates that the integer is coded on 8 bytes which will be coded in the eight following bytes.

One can note that there is no extra cost to code an integer from 0 to 23. Thus, the value 15 will be coded 0x0F (000-0 1111) while, for all the other higher values, the overcost will be only of one byte. The value 100 will be coded 000-1 1000² followed by the coding on 1 byte of the value 100 (0110 0100).

Listing 6.2 – cbor-integer-ex1.py

```
1 import cbor2 as cbor
2
3 v = 1
4
5 for i in range (0, 19):
6     c = cbor.dumps(v)
7     print ("{0:3}{1:30}{2}.".format(i, v, c.hex()))
8
9     v *= 10
```

The program `cbor-integer-ex1.py` displays the powers of 10 between 10^0 and 10^{18} :

- Line 1, the program imports the module `Indexcbor2` and renames it for simplicity `cbor`.
- Line 5, the loop allows to have the multiples of 10 (variable `v`).
- Line 6, the module `cbor` uses as for JSON the method `dumps` to serialize an internal Python structure into the requested representation. Conversely, the `loads` method will be used to import a CBOR structure into an internal representation.

2. 11000 corresponds to 24

- On line 7, the print is used to align the data so that the display is clearer; between the braces, the first number indicates the position in the format arguments ; the second, after the :, the number of characters. For example, { :1:30} indicates the format argument v displayed in 30 characters.

The program gives the following result :

```
# python3 cbor-integer-ex1.py
0           1 01
1           10 0a
2           100 1864
3           1000 1903e8
4           10000 192710
5           100000 1a000186a0
6           1000000 1a000f4240
7           10000000 1a00989680
8           100000000 1a05f5e100
9           1000000000 1a3b9aca00
10          10000000000 1b0000002540be400
11          100000000000 1b000000174876e800
12          1000000000000 1b000000e8d4a51000
13          10000000000000 1b000009184e72a000
14          100000000000000 1b00005af3107a4000
15          1000000000000000 1b00038d7ea4c68000
16          10000000000000000 1b002386f26fc10000
17          100000000000000000 1b016345785d8a0000
18          1000000000000000000 1b0de0b6b3a7640000
```

It is easy to see that the values 1 and 10 are coded on 1 byte ; that 100 is coded on 2 bytes while the values 1 000 and 10 000 are coded on 3 bytes. The values between 100 000 and 1 000 000 000 require 5 bytes and the following values, 9 bytes.

The size of the representation adapts to the value. Thus, it is not necessary to define a fixed size to encode a data.

We can also note that as the major type is on 3 bits, this type can be recognized in a hexadecimal reading of the result because the sequence always starts with the symbol 0 or 1.

6.7.3 Type Negative Integer

The major type negative integer is roughly similar to the positive integer. The major type is 001 and the encoding of the value is done on the absolute value of the number to which we subtract 1. This avoids two different codes for the values 0 and -0.

Thus, to code -15, we will code the value 14, which gives in binary 001-1 1110. Thus, -24 can also be coded on 1 byte while +24 will be coded on 2 bytes.

Listing 6.3 – cbor-integer-ex2.py

```
1 import cbor2 as cbor
2
3 v = -1
4
```

```

6   for i in range (0, 19):
7       c = cbor.dumps(v)
8       print ("{:0:3}{:1:30}{:2} ".format(i, v, c.hex()))
9
10      v *= 10

```

The program `cbor-integer-ex2.py` uses the same code as the previous program, but the variable `v` is initialized with the value -1. This program will process negative powers of 10.

```

# python3.5 cbor-integer-ex2.py
0          -1 20
1          -10 29
2          -100 3863
3          -1000 3903e7
4          -10000 39270f
5          -100000 3a0001869f
6          -1000000 3a000f423f
7          -10000000 3a0098967f
8          -100000000 3a05f5e0ff
9          -1000000000 3a3b9ac9ff
10         -10000000000 3b0000002540be3ff
11         -100000000000 3b000000174876e7ff
12         -1000000000000 3b000000e8d4a50fff
13         -10000000000000 3b000009184e729fff
14         -100000000000000 3b00005af3107a3fff
15         -1000000000000000 3b00038d7ea4c67fff
16         -10000000000000000 3b002386f26fc0ffff
17         -100000000000000000 3b016345785d89ffff
18         -1000000000000000000 3b0de0b6b3a763ffff

```

6.7.4 Type Binary sequence or Character string

Binary sequences and strings have the same behavior. The major type is respectively 010 and 011. It is followed by the length of the sequence or the string. The same type of encoding as for integers is used :

- if the length is lower than 23, it is coded in the continuation of the first byte. One finds then the number of bytes or characters corresponding to this length ;
- if the length can be coded in 1 byte (thus lower than 255), the continuation of the first byte contains 24 then the following byte contains the length followed by the number of bytes or characters corresponding.
- if the length can be coded in 2 bytes (thus lower than 65535), the continuation of the first byte contains 25 then the following byte contains the length followed by the number of bytes or characters corresponding.
- if the length can be coded in 4 bytes, the continuation of the first byte contains 26 then the following byte contains the length followed by the number of bytes or characters corresponding.
- if the length can be encoded in 8 bytes, the continuation of the first byte contains 27 then the following byte contains the length followed by the number of bytes or characters corresponding.

This coding is also quite optimal. It is rare to send more than 23 characters.

Listing 6.4 – cbor-string.py

```
import cbor2 as cbor

for i in range (1, 10):
    c = cbor.dumps("LoRaWAN"+str(i))

    print ("{0:3} {1}".format(i, c.hex()))

bs = cbor.dumps(b"\x01\x02\x03")
print (bs.hex())
```

The program `cbor-string.py` shows the representation of strings of increasing length and a binary sequence :

- line 3, the variable `i` takes values from 1 to 9.
 - line 6, The multiplication of a string by an integer (line 4) indicates the number of repetitions of it.
 - lines 8 and 9 show the encoding of a byte string. The variable `bs` contains the CBOR representation of a Python byte string (represented by the character `b` before the quotation marks, the values which do not correspond to ASCII characters are preceded by the symbols `\x`). The hexadecimal representation of the CBOR object is then displayed.

The result is as follows :

Up to 3 repetitions of the string "LoRaWAN", the length coding is optimal (coded on 2 bytes).

6.7.5 Type Array

The array type will group a set of elements. Each of these elements being a CBOR structure, the only information needed to know the beginning and the end of an array is its number of elements. The major type is 100. There are two methods to encode the length of an array :

- if this one is known at the time of coding, it is enough to indicate it with a coding identical to the one used to indicate the length of a character string;
 - if this is not known at the time of coding, there is a special code to indicate the end of the table. We will talk about this later.

Listing 6.5 – cbor-array.py

```
1 import cbor2 as cbor
2
3 c1 = cbor.dumps([1,2,3,4])
4 print(c1.hex())
5 print()
6
7 c2 = cbor.dumps([1,[2, 3], 4])
8 print(c2.hex())
9 print()
```

```
11 c3 = cbor.dumps([1000, +20, -10, +100, -30, -50, 12])
print (c3.hex())
```

The program `cbor-array.py` gives some examples of array coding :

- [1,2,3,4] defined on line 3, becomes 8401020304. We can guess the structure of the CBOR message : 0x84 indicates an array of 4 elements (be careful, decoding is not always so simple). The 4 elements are integers lower than 23 ;
- [1,[2, 3], 4] defined on line 7 becomes 830182020304. This is an array of 3 elements, the second of which is an array of two elements ;
- [1000, +20, -10, +100, -30, -50, 12] defined on line 11, becomes 871903e814291864 381d38310c. Note that the coding of the elements is of variable length, but as each element codes its length, it is only necessary to know the number of elements.

The type List of pairs or Map is indicated by the value 101. It works in the same way as arrays by counting the number of elements. But this time, the value represents a pair, i.e. two consecutive CBOR objects.

Listing 6.6 – `cbor-mapped.py`

```
1 import cbor2 as cbor
2
3 c1 = {"type" : "hamster",
4       "taille" : 300,
5       2 : "test",
6       0xF: 0b01110001,
7       2 : "program"};
8
9 print (cbor.dumps(c1).hex())
```

The program `cbor-mapped.py` gives an example of encoding. Note that the structure to be encoded is not directly compatible with JSONfootnote`json.dumps` would have converted the numeric keys into strings '`{"type": "hamster", "2": "program", "taille": 300, "15": 113}`', some keys are not strings.

The result is `a464747970656768616d73746572667461696c6c6519012c026770726f6772 616d0f1871` which is not very easy to read.

cbor.me

The web site <https://cbor.me> allows to do automatically the encoding in a direction or in the other. The left column represents the data in JSON and the right one in CBOR (called "canonical representation" which facilitates the reading). Having entered the above hexadecimal sequence, the site presents it as shown in figure 6.6 on the following page. The CBOR part is indexed and commented to make the CBOR object more readable. It can also be translated into a JSON equivalent, although some keys remain numeric.

On these examples, we can see that CBOR is much more permissive and complete than JSON, the first field of the CBOR map can be numeric and does not have to be unique in the whole structure. Nevertheless CBOR defines a strict mode in which these keys must be ASCII encoded and unique to be compatible with JSON. If a key is repeated several times in a CBOR structure, it is necessary to process the information

Youtube



The screenshot shows a web-based CBOR playground at <https://cbor.me>. The interface includes a toolbar with various icons, a status bar indicating the URL, and a message about the CBOR specification and RFC 8949. The main area features a large title "CBOR" in purple. Below it, there's a "Diagnostic" button with a green background and white text. To the right of the button, there are several checkboxes: "36 Bytes" (checked), "as text" (unchecked), "utf8" (unchecked), "emb cbor" (unchecked), and "cborseq" (unchecked). A text input field below these checkboxes contains the JSON string: {"type": "hamster", "taille": 300, 2: "program", 15: 113}. To the right of the input field is a hex dump of the CBOR bytes. The hex dump is as follows:

A4	# map(4)
64	# text(4)
74797065	# "type"
67	# text(7)
68616D73746572	# "hamster"
66	# text(6)
7461696C6C65	# "taille"
19 012C	# unsigned(300)
02	# unsigned(2)
67	# text(7)
70726F6772616D	# "program"
0F	# unsigned(15)
18 71	# unsigned(113)

FIGURE 6.6 – Definition of major in CBOR

directly in the CBOR structure and not to try to convert or deserialize it because there is a risk of losing information.

6.7.6 Type Tag

CBOR enriches the typing of data; this makes it easier to manipulate data. For example, a character string can represent a date, a URI, or even a URI encoded in base 64. The type 110 can be followed by a value or **tag** of which an exhaustive list is maintained by the IANAfootnoteurlhttps://www.iana.org/assignments/cbor-tags/cbor-tags.xhtml.

Listing 6.7 – cbor-tag.py

```

1 import cbor2 as cbor
2 from datetime import date, timezone
3
4 print(date.today())
5 c1 = cbor.dumps(date.today(), timezone=timezone.utc, date_as_datetime=True)
6
7 print(c1.hex())
8
9 print(cbor.loads(c1))
10 print(type(cbor.loads(c1)))

```

For example, the program `cbor-tag.py` returns the following results :

```

# python3.5 cbor-tag.py
2018-05-22
c074323031382d30352d32325430303a30303a30305a
2018-05-22 00:00:00+00:00
43010203
<class 'datetime.datetime'>

```

The canonical representation shows more easily the tag in the binary sequence :

C0	# tag(0)
74	# text(20)
	323031382D30352D32325430303A30303A30305A # "2018-05-22T00:00:00Z"

The tag 0 implies a normalized format for the date; hence the addition of hours, minutes and seconds, even though they were not initially specified. We can also notice that `loads` returns a type `date` and not a character string.

6.7.7 The floating type and particular values

The last major type (111) allows to encode floating numbers by using the representation defined by the **IEEE 754**. According to the size of the representation, the continuation of the byte contains the values 25 (half precision on 16 bits), 26 (simple precision on 32 bits) or 27 (double precision on 64 bits).

This type also allows to encode the values defined by JSON : True (value 20), False (value 21) or None (value 22).

Finally, this type can indicate the end of an array or a list of pairs when the size is not known at the beginning of the coding.

6.8 Questions about CBOR

Question 6.8.1: Benefits of CBOR

What are the advantages of CBOR over JSON (2 answers) ?

- It is more compact in its data representation.
- It allows to represent floating numbers.
- It compresses strings.
- It is easier to implement.

Question 6.8.2: Floating

Is a float always more compact in CBOR than in JSON ? - You can help yourself with <https://cbor.me>

- Yes, that is the goal of CBOR.
- Yes, for floats that have the decimal part at 0.
- Yes, for small precision floats (up to a hundredth).
- Yes, for high precision floats (6 digits after the decimal point).

Question 6.8.3: IgfChaîne de caractères

Consider a string in CBOR.

- It is compressed with an entropy algorithm (e.g. Huffman coding).
- It can contain accented characters.
- Each character is coded on 6 bits.

Question 6.8.4: Variable length

In CBOR, the size of an integer varies according to its value !

- True
- False
- It depends on how this integer was declared.

Question 6.8.5: Array

In CBOR, an array can contain objects of different types.

- True
- False
- It depends on how this table was declared.

Question 6.8.6: Fractional

We want to define an array of two elements as a fraction. What tag should precede the structure ? (you can use the [RFC 8949](#)).

6.9 SenML

is a specification that leverages JSON or CBOR. It lists a set of names/units/measures and standardizes them into a unique key name. By using this standardization, interoperability is facilitated. The keys and values are therefore regulated and typed to avoid any interoperability conflicts. The format is defined in the RFC8428 and is based on a table structure grouping objects as shown in the following figure taken from the RFC.

```
[  
  {"bn" : "urn:dev:ow:10e2073a01080063", "bt":1.320067464e+09,  
   "bu" : "%RH", "v":21.2},  
  {"t":10, "v":21.3},  
  {"t":20, "v":21.4},  
  {"t":30, "v":21.4},  
  ...
```

SenML defines the keys used in the object. To have a compact notation, they are limited to 1 or 2 characters. Among them, "bn" indicates a base name and "n" the name of a device. If several devices send the common part of the device identifier, we can put the "bn" to avoid repeating it each time.

The base time (or "bt") is also a way to compact the time notation. The time ("t") gives the offset and leads to a smaller value as seen in the example.

The base unit ("bu") indicates the default unit if the other objects do not have a keyword indicating the unit ("u").

The [RFC 8428](#) defines a list of units such as kilogram ("kg"), volt ("V"), etc. In the example, "%RH" refers to a percentage of relative humidity. A numeric value uses the letter "v", a string uses the "vs" key.

CBOR uses the same structure but small positive and negative integers are substituted in the keys of CBOR objects : "bn", "bt", "bu" will be represented by -1, -2 and -3 respectively and "n", "t", "u" by +0, +2 and +6.

7. Implementing a Virtual Sensor

The programs related to this section are located in the directory `plido-tp3`. It is recommended to use the virtual machine with one terminal window for the object and another for the server.

7.1 JSON

We've been talking about the Internet of Things for a long time, it's time to put our knowledge into practice. We will start with a simple implementation in Python on your machine. The goal in this part is to test everything we have seen without any other hardware than a computer. We will open two terminal windows. In one we will run a program that will emulate the object with three sensors. This object will communicate with a server that will collect the information. The communication will be done internally with a socket using the interface **loopback** (cf. figure 7.1 on the next page).

Youtube



7.1.1 Minimal Server

Listing 7.1 – `minimal_server.py`

```
1 import socket
2 import binascii
3
4 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
5 s.bind(('0.0.0.0', 33033))
6
7 while True:
8     data, addr = s.recvfrom(1500)
9     print (data, "=>", binascii.hexlify(data))
```

Let's start by building a summary server (`minimal_server.py`) which will display everything it receives.

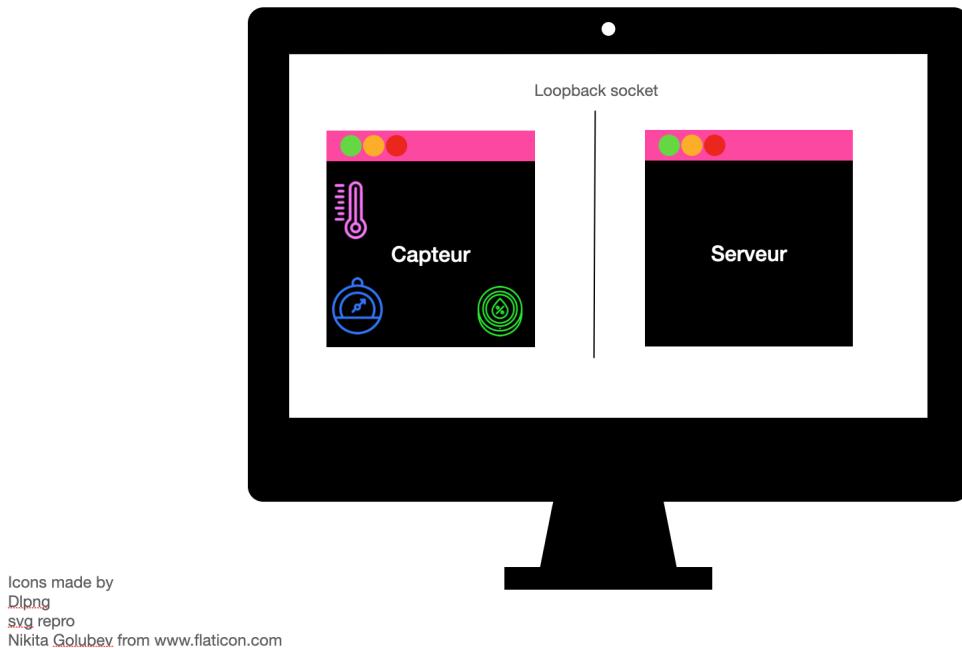


FIGURE 7.1 – Client Server architecture

- lines 1 and 2 the modules `socket` for the communication and `binascii` to transform the binary into string are imported.
- line 4, the `socket` function creates the socket `s` with parameters for IP (`AF_INET`) and UDP (`SOCK_DGRAM`) communications.
- line 5, the socket is associated with the port number 33033 via the function `bind`. The address `0.0.0.0` indicates that the data can come from any interface (Ethernet, Wi-Fi, loopback,...).
- line 8, in the endless loop, the function `recvfrom` will get stuck waiting for data. It returns the data and the address of the sender.
- line 9, the data are displayed in byte string and in hexadecimal.

We launch the server program. As nobody talks to it, it does not display anything.

7.1.2 Virtual sensor

Listing 7.2 – virtual_sensor.py

```

1 import random
2
3 class virtual_sensor:
4
5     def __init__(self, start=None, variation=None, min=None, max=None):
6         if start:
7             self.value = start
8         else:
9             self.value = 0

```

```

11         self.variation = variation
12         self.min = min
13         self.max = max
14
15     def read_value(self):
16         self.value += random.uniform(self.variation*-1, self.variation )
17
18         if self.min and self.value < self.min: self.value = self.min
19         if self.max and self.value > self.max: self.value = self.max
20
21     return self.value
22
23 if __name__ == "__main__":
24     import time
25
26     temperature = virtual_sensor(start=20, variation = 0.1)
27     pressure = virtual_sensor(start=1000, variation = 1)
28     humidity = virtual_sensor(start=30, variation = 3, min=20, max=80)
29
30     while True:
31         t = temperature.read_value()
32         p = pressure.read_value()
33         h = humidity.read_value()
34
35         print ("{:7.3f} {:10.3f} {:7.3f}".format(t, p, h))
36
37         time.sleep(1)

```

The module **virtual_sensor** with the class of the same name, reflects in a more or less realistic way the behavior of a sensor. We can see in the main program (line 23 to 37) that we have created three virtual sensors : one for temperature (line 31), another for pressure (line 32), and the third for humidity (line 34). The argument `start` specifies the starting value, `variation` the range of variation between two measurements and `min` and `max` the values not to exceed. The endless loop that displays the different values every second.

```

> python3 virtual_sensor.py
54.956    999.850   30.609
54.963    1000.473   32.505
55.062    1000.845   31.870
55.017    1001.619   32.257
55.083    1000.767   31.757
55.027    1001.442   31.742

```

Direct sending

Listing 7.3 – minimal_client1.py

```

1 from virtual_sensor import virtual_sensor
2 import time
3 import socket
4
5 temperature = virtual_sensor(start=20, variation = 0.1)
6 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
7
8 while True:

```

```

9     t = temperature.read_value()
10    s.sendto (t, ("127.0.0.1", 33033))
11    time.sleep(10)

```

Scripts using this module can be written, as for example the program `minimal_client1.py`.

The variable `t` contains the temperature which is transmitted on port 33033 at the *loopback* address. We thus obtain a communication between two programs in your computer whose local IP address is 127.0.0.1. But when you launch the program `minimal_client1.py`, you get the following error :

```

>python3.5 minimal_client1.py
Traceback (most recent call last):
  File "minimal_client1.py", line 11, in <module>
    s.sendto (t, ("127.0.0.1", 33033))
TypeError: a bytes-like object is required, not 'float'

```

Question 7.1.1: Bug?

Why doesn't the client program work ?

- The IP address is not correct.
- The data serialization process is missing.
- The variable `t` is not defined.
- The variable `t` cannot be read.

Sending a byte string

Listing 7.4 – `minimal_client2.py`

```

10   t = temperature.read_value()
11   s.sendto (str(t).encode(), ("127.0.0.1", 33033))

```

In the program `minimal_client2.py`, the floating number contained in the variable `t` is transformed into character string with the function `Indexstr`, then into bytes string with the method `encode` to be compatible with the argument expected by the method `sendto`. On the server side the function `Indexstr` converts the received byte string into a float.

Sending several values

Listing 7.5 – `minimal_client3.py`

```

12  while True:
13      t = temperature.read_value()
14      p = pressure.read_value()
15      h = humidity.read_value()
16
17      msg = "{} , {} , {}".format(t, p, h)
18      s.sendto (msg.encode(), ("127.0.0.1", 33033))

```

To send simultaneously the values of the three sensors, the representation via a string is a little more complicated to implement. If the program `minimal_client3.py` uses `format` to send data

separated by vigulas. On the server side, you have to decode this string to find the integers. And this is much more complex to do !

JSON

Listing 7.6 – minimal_client4.py

```

12 while True:
13     t = temperature.read_value()
14     p = pressure.read_value()
15     h = humidity.read_value()
16
17     j = [t, p, h]
18     s.sendto (json.dumps(j).encode(), ("127.0.0.1", 33033))
19     time.sleep(10)

```

The simplest solution is to put these three values in a python array and transform it into a JSON representation using the `dumps` function of the `json` module.

This JSON string is in turn transformed into a byte string with encoding and sent to the server. In our case, the server only displays the string but you can use the `loads` method of the `json` module to deserialize it and turn it into a Python structure in the server, on which it is now easy to perform operations such as, for example, an average calculation.

```
% python3 minimal_server.py
b'[19.93044784157464, 999.1552628155773, 35.723583473834566]' => b'5b31392e39333034343738343135373436342c203939392e313535c...
b'[19.940155545405723, 998.7581534530281, 35.820037116376184]' => b'5b31392e3934303135353534353430353732332c203939382e3735...
b'[20.003803212269627, 999.3517302791449, 34.33544522779677]' => b'5b32302e3030333830333231323236393632372c203939392e33353...
```

7.2 CBOR

The passage from JSON to CBOR is very simple. It is enough to change a module `cbor` instead of the module of `json`. The program `minimal_client5.py` :

- line 1 calls the `cbor2` module and renames it `cbor`
- The rest of the program is identical to the previous one, it is only in the serialization, line 18, that the function `json.dumps` is replaced by `cbor.dumps`. The format returned being a string of bytes, it is no longer necessary to call the `encode` method.

Youtube



The program `minimal_server.py` is not modified since it only displays what it receives.

```
> python3 minimal_server.py
b'\x83...' => b'83fb40341086f3e8b66fb408f3b7791c8d61ffb403fac15ba06088e',
b'\x83...' => b'83fb40341d4d28495268fb408f33d502185c3dfb403d95a2c4981444',
```

Listing 7.7 – minimal_client5.py

```

1 from virtual_sensor import virtual_sensor
2 import time
3 import socket
4 import cbor2 as cbor
5

```

```

temperature = virtual_sensor(start=20, variation = 0.1)
7 pressure     = virtual_sensor(start=1000, variation = 1)
humidity    = virtual_sensor(start=30, variation = 3, min=20, max=80)
9
10 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
11
12 while True:
13     t = temperature.read_value()
14     p = pressure.read_value()
15     h = humidity.read_value()
16
17     j = [t, p, h]
18     s.sendto(cbor.dumps(j), ("127.0.0.1", 33033))
19     time.sleep(10)

```

We get the following result. The CBOR sequence is 28 bytes long, the JSON equivalent would have been 60 bytes. Even if this divides by two the size of the data to be transmitted, the result is not compact. This is due to the representation of floats in CBOR, because here floats are coded on 8 bytes.

Question 7.2.1: Decoding

Let us analyze the received sequence : 83fb40341086f3e8b66bfb408f3b7791c8d61ffb403fac
15ba06088e

What does the byte 0x83 that starts the received CBOR structure correspond to ?

- to the coding of the positive integer 131.
- to the coding of the negative integer 132.
- to the definition of an array of 3 elements.
- the definition of a CBOR map of 3 elements.
- to the definition of an array of undefined size.

Question 7.2.2: Floating

In this structure, what is the CBOR marker (in hexadecimal) that indicates that we have a floating number ?

Question 7.2.3: Floating size

What is the size of this floating number in bytes ?

Use of integers

To reduce the size of the transmitted data, we will use integers. We will need a precision to the hundredth (two digits after the decimal point). To do this, on the client side, we just take the integer part of the number multiplied by 100 and on the server side, we divide the received value by 100. The modification of the code is minor.

Listing 7.8 – minimal_client6.py

```

j = [int(t*100), int(p*100), int(h*100)]
18 s.sendto(cbor.dumps(j), ("127.0.0.1", 33033))

```

```
> python3 minimal_server.py
b'\x83\x19\x07\xd7\x1a\x00\x01\x860\x19\x0c\xa7' => b'831907d71a0001864f190ca7',
b'\x83\x19\x07\xd4\x1a\x00\x01\x86f\x19\x0cJ' => b'831907d41a00018666190c4a',
b'\x83\x19\x07\xd4\x1a\x00\x01\x86\x92\x19\rP' => b'831907d41a00018692190d50',
```

The change is minor and fits in 12 bytes, but there is a decrease in interoperability, as both entities need to know the transformation of the value related to the multiplication by 100.

Question 7.2.4:

What is the minimum and maximum size of the CBOR structure sent, taking into account the possible values.

8. Time series

The virtual sensors that we have programmed so far emit data each time they have a measurement. This allows the server to follow the behavior of the studied system in real time. But in some cases, real time is not necessary and it is preferable to limit the number of emissions, for example to save the energy of the sensor.

To do this, we can use an array that will accumulate the values and send it when the array reaches a certain size.

Youtube



8.1 Sending an array

The program `minimal_humidity1.py` accumulates the data in an array `humidity` when this one reaches 30 elements (line 17), the data are sent to the server.

Listing 8.1 – `minimal_humidity1.py`

```
from virtual_sensor import virtual_sensor
import time
import socket
import cbor2 as cbor

humidity      = virtual_sensor(start=30, variation = 3, min=20, max=80)

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
NB_ELEMENT = 30
h_history = []

while True:
    h = int(humidity.read_value()*100)

    # No more room to store value, send it.
    if len(h_history) == NB_ELEMENT:
```

```

18     if len(h_history) >= NB_ELEMENT:
19         s.sendto (cbor.dumps(h_history), ("127.0.0.1", 33033))
20         h_history = []
21
22     h_history.append(h)
23     print (len(h_history), len(cbor.dumps(h_history)), h_history)
24
25     time.sleep(10)

```

```

1 4 [3241]
2 7 [3241, 2945]
3 10 [3241, 2945, 2762]
4 13 [3241, 2945, 2762, 2625]
5 16 [3241, 2945, 2762, 2625, 2480]
6 19 [3241, 2945, 2762, 2625, 2480, 2769]

```

The first digit of the line indicates the number of elements and the second the size in the CBOR coding. We notice that adding an element increases the size of the array by 3 bytes. The values corresponding to a moisture measurement do not vary greatly. Thus an array of 30 measurements has a size of 92 bytes.

8.2 Differential coding

The amount of data transferred can be optimized by using delta coding (i.e. the variation in humidity). The first value in the table is the measured value while the following values represent the difference between the measured value and the previous one.

Listing 8.2 – minimal_humidity2.py

```

18     if len(h_history) == 0:
19         h_history = [h]
20     elif len(h_history) >= NB_ELEMENT:
21         print ("send")
22         s.sendto (cbor.dumps(h_history), ("127.0.0.1", 33033))
23         h_history = [h]
24     else:
25         h_history.append(h-prev)
26
27     prev = h

```

The program `minimal_humidity2.py` handles the filling of the array differently :

- line 14 and 15, if the table is empty, the table is created with the measured value,
- line 16 to 22, otherwise if the table is full, it is serialized in CBOR and sent to the server, then reset with the measured value,
- line 23 and 24, otherwise the difference between the previous value and the measured value is stored in the table.

The following listing gives an example of execution.

```

1 4 [2521]
2 6 [2521, 79]
3 8 [2521, 79, 224]
4 10 [2521, 79, 224, -40]

```

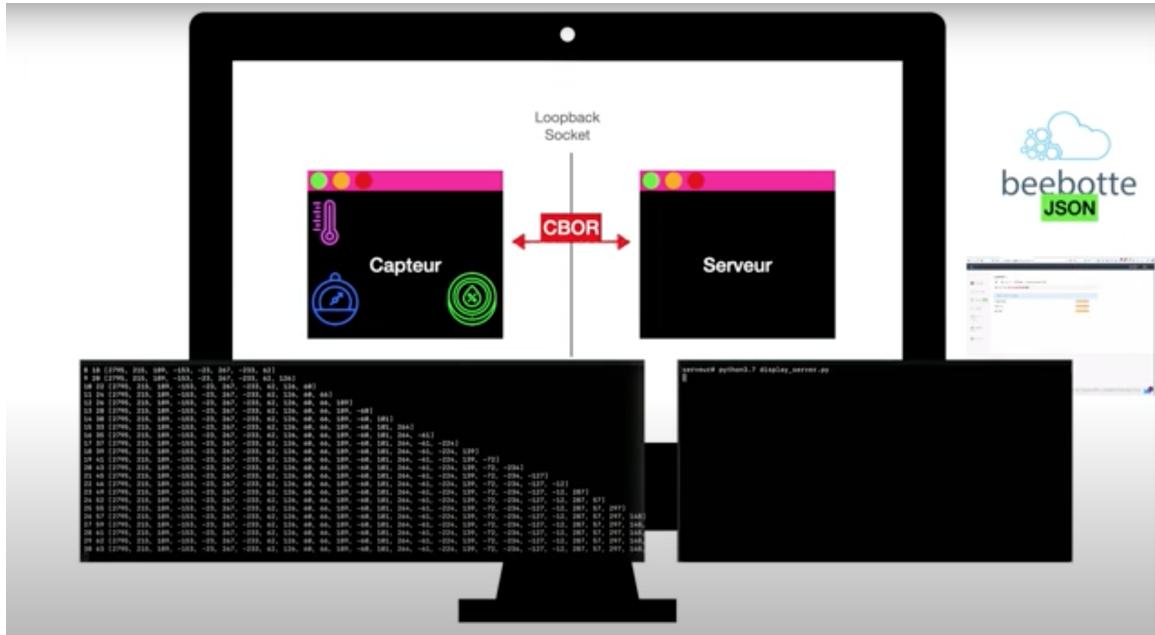


FIGURE 8.1 – Architecture Client/Serveur

```

5 12 [2521, 79, 224, -40, -112]
6 13 [2521, 79, 224, -40, -112, 1]
7 15 [2521, 79, 224, -40, -112, 1, 130]
8 18 [2521, 79, 224, -40, -112, 1, 130, -288]
9 21 [2521, 79, 224, -40, -112, 1, 130, -288, 299]
    
```

Ceci met en valeur deux souplesses de CBOR :

- la taille du tableau est dynamique. Si l'on change le nombre de valeurs à transmettre, le tableau l'indique et l'on n'a pas besoin de modifier le code du récepteur ;
- la taille des données dépend de leur valeur. Pour les variations entre -24 et +23, un seul octet sera nécessaire. On le voit sur l'exemple précédent : l'ajout de la valeur '1' dans le tableau fait passer la taille de la représentation CBOR de 12 à 135 octets. Les valeurs entre 256 et +255 sont transmises sur 2 octets ; il est donc possible de cette manière d'optimiser la transmission sans ajouter de contrainte. S'il y avait une brusque variation de l'humidité, la représentation CBOR s'adapterait pour la transmettre.

La taille est réduite d'un tiers (environ 66 octets) pour transmettre la même information.

8.3 Architecture

La figure 8.1 représente l'architecture générale du système. Le programme `minimal_humidity2.py` fournit les séries temporelles. Il reste à définir le programme serveur qui va les traiter et faire appel à un autre service pour les afficher sous forme de graphe.

Si l'on suit le flux d'information, le capteur va produire des données au format CBOR pour être compact et le programme serveur va transformer cette information en une structure JSON respectant les spécifications du service d'affichage.

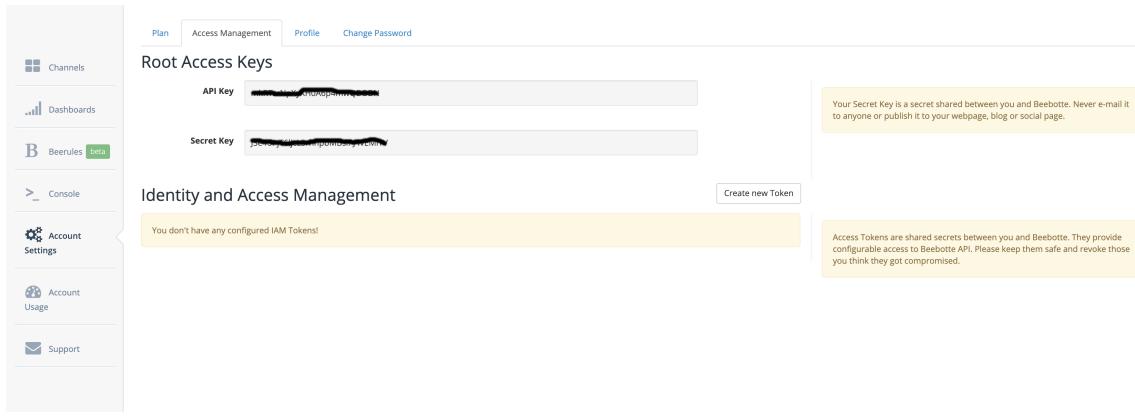


FIGURE 8.2 – Clé et secret pour l’authentification

8.4 Beebotte

Il existe plusieurs sites qui permettent de le faire. Nous allons utiliser <https://beebotte.com>, mais ce que nous allons présenter peut très bien s’appliquer à d’autres sites.

8.4.1 Configuration

La première étape consiste à créer un compte en cliquant sur *Sign Up* sur la page de garde et en remplissant un formulaire classique avec votre login, adresse de courrier électronique et mot de passe. Une fois le compte validé, le service est accessible.

Le compte nous permet de nous authentifier pour gérer les données sur le site, mais il faut également disposer d’autorisation pour pouvoir y déposer des données via l'**API REST**. Pour cela, il faut se rendre sur la page *Account Setting* puis l’onglet *Access Management*. Cette page (cf. figure 8.2) donne une clé et un secret pour gérer l’ensemble des données sur le site.

Notez ces valeurs et stockez les dans un fichier `config_bbt.py` qui a cet aspect (vos valeurs sont forcément différentes) :

Listing 8.3 – config_bbt.py

```
1 API_KEY      = "GAJ3SFmUZSXmB2zqdcmcuXc"
2 SECRET_KEY   = "4NCsrM1cfmFdMZF4E47aTfmCaU3UfyQo"
```



Nous allons maintenant créer un canal (/texttchannel) dans lequel nous allons définir les objets correspondant aux capteurs. En Cliquant sur *Channels* puis *Create New*, la page représentée figure ?? on page ?? apparaît.

Il faut donner un nom au channel (*capteurs* dans l'exemple), cocher la case *public* et créer trois ressources pour les trois valeurs qui nous intéressent (*temperature*, *humidity*, *presure*) et faire correspondre les unités.

8.4.2 Enregistrement des ressources

Le programme `display_server.py` permet de correspondre avec Beebotte via son API REST. Il commence par l’importation des modules nécessaires :

Listing 8.4 – display_server.py

```

1 import socket
2 import binascii
3 import cbor2 as cbor
4 import beebotte
5 import config_bbt #secret keys
6 import datetime
7 import time
8 import pprint

```

- ligne 4, le module Python beebotte est disponible pour simplifier la manipulation des données¹.
- ligne 5, le module contient la clé et le secret nécessaire à la connexion obtenu précédemment.

```

10 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
11 s.bind(('0.0.0.0', 33033))

```

- ligne 10 et 11 permettent d'ouvrir la socket pour communiquer avec les capteurs.

```

12 bbt = beebotte.BBT(config_bbt.API_KEY, config_bbt.SECRET_KEY)

```

- ligne 13 une instance permettant la connexion avec les serveurs de Beebotte est définie grâce à la fonction BBT. Les paramètres de connexion provenant du module config_bbt sont pris en compte.

```

13 while True:
14     data, addr = s.recvfrom(1500)
15
16     j = cbor.loads(data)
17     to_bbt("capteurs", "temperature", j, factor=0.01)

```

Dans le programme principal, un boucle sans fin attend la série temporelle codée en CBOR venant du capteur (ligne 42), les transforme tableau Python (ligne 44) et appelle la fonction to_btt en précisant

- le canal et la ressource qui ont été définie précédemment sur Beebotte ;
- la série temporelle
- la précision pour transformer ces entiers en flottant.

```

18 def to_bbt(channel, res_name, cbor_msg, factor=1, period=10, epoch=None):
19     global bbt
20
21     prev_value = 0
22     data_list = []
23     if epoch:
24         back_time = epoch
25     else:
26         back_time = time.mktime(datetime.datetime.now().timetuple())
27
28     back_time -= len(cbor_msg)*period

```

1. S'il n'était pas présent sur votre ordinateur, vous devriez l'installer avec la commande pip3 install beebotte.

```

28     for e in cbor_msg:
29         prev_value += e
30
31         back_time += period
32
33         data_list.append({"resource": res_name,
34                           "data" : prev_value*factor,
35                           "ts" : back_time*1000} )
36
37         pprint.pprint (data_list)
38
39         bbt.writeBulk(channel, data_list)

```

La fonction `to_bbt` fait l'essentiel du travail de transformation. Elle prend en argument :

- le nom du canal créé sur Beebotte. Dans notre cas, ce sera `capteurs`;
- le nom de l'objet dans ce canal que nous avons également créé sur le site web. Dans notre cas, ce sera `humidity`;
- le tableau python des mesures codées en delta;
- le facteur multiplicatif, c'est-à-dire la précision. Ici, il faudra diviser par 100;
- la période entre deux mesures; cela nous permettra de calculer l'instant de la mesure. Par défaut, la période est de 10 secondes;
- le temps de réception du message pour dater les échantillons. S'il n'est pas spécifié, le temps courant est pris.

Cette fonction transforme le tableau Python suivant :

```
[3311, 124, -144, -188, -94, 289, -1, -72, 1 ...]
```

en un tableau de dictionnaire :

```
[{'data': 33.11, 'resource': 'humidity', 'ts': 1596730115000.0},
 {'data': 34.35, 'resource': 'humidity', 'ts': 1596730125000.0},
 {'data': 32.91, 'resource': 'humidity', 'ts': 1596730135000.0},
 {'data': 31.03, 'resource': 'humidity', 'ts': 1596730145000.0},
 ...]
```

Chaque dictionnaire contient trois éléments imposés par Beebotte :

- le nom de la ressource (`resource`) telle qu'elle a été définie sur l'interface pour le canal;
- la valeur associée pour cette ressource (`data`);
- l'instant à laquelle cette mesure a été faite (`ts`). Le temps est représenté suivant le format **Epoch** qui compte le nombre de secondes depuis le premier Janvier 1970².

Le calcul du *timestamp* (`ts`) est l'opération la plus complexe de cette fonction mais les module `time` et `datetime` facilitent le calcul. Si l'argument `epoch` a été fourni lors de l'appel, la fonction prend cette valeur, sinon le calcule ligne 23. La fonction `now` retourne la date et l'heure courante, qui est transformé en un tuple grâce à la fonction `timetuple`. A partir de ce dernier, la fonction `maketime` le converti en epoch.

Ligne 25, l'epoch à laquelle la première mesure du tableau a été faite est calculé en prenant le temps actuel (cela suppose que l'on néglige le temps de traitement et de transmission) auquel on

2. voir <https://www.epochconverter.com/> pour les conversions.

Configured resources		
temperature	No Persisted Data	
pressure	No Persisted Data	
humidity	26.51 %	2 minutes ago

FIGURE 8.3 – État des ressources

FIGURE 8.4 – Crédit d'un widget

retranche la durée de la capture, c'est-à-dire comme le nombre d'éléments du tableau multiplié par l'intervalle entre chaque mesure (*period*).

Ligne 32 à 34 la structure attendue par Beebotte est construite. le résultat est envoyé, ligne 38, grâce à la fonction `writeBulk` qui permet d'envoyer un ensemble de valeurs dans un tableau.

On peut vérifier que Beebotte a reçu des données en visualisant le canal capteurs sur l'interface Web. On peut voir sur la figure 8.3 que seule la ressource `humidity` a reçu des données. L'interface affiche la dernière valeur reçue et la date de réception.

8.4.3 Visualisation des ressources

Maintenant que les ressources sont stockées dans les serveurs de Beebotte, il est possible de les visualiser graphiquement, en allant dans *Dashboard* puis *create Dashboard* et *Add Widget* pour sélectionnez un widget comme *Multi-line chart*.

Puis, configurez le widget en définissant le canal et la ressource de ce canal comme le montre la figure 8.4.

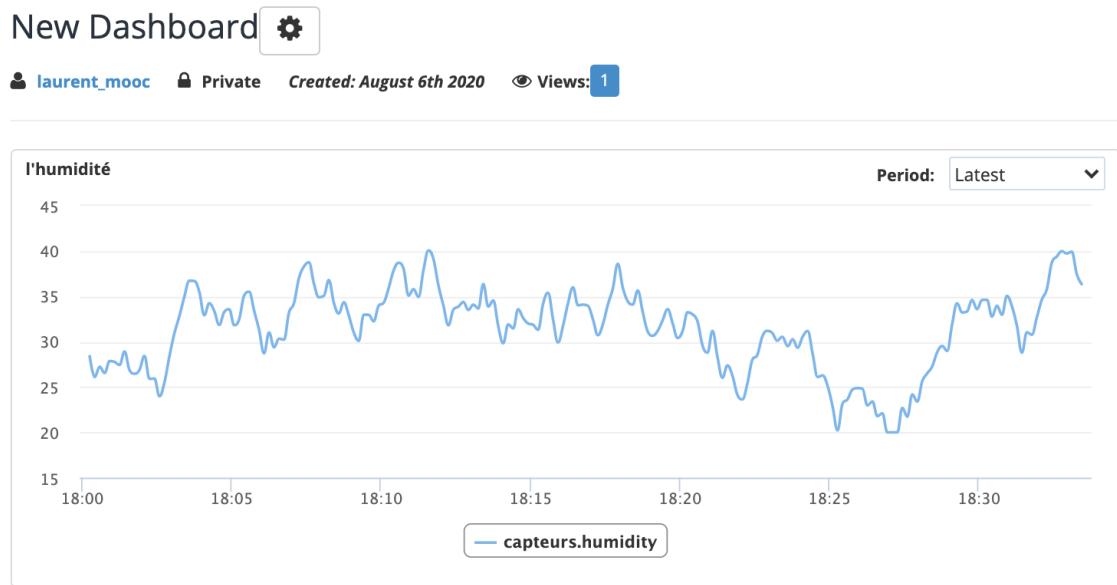


FIGURE 8.5 – Suivi de l’humidité

En retournant sur le dashboard, on peut voir l’évolution de l’humidité au cours du temps (cf. figure 8.5).

8.5 Interopérabilité

La chaîne de collecte de l’information que nous venons de construire allant du capteur à l’affichage, n’est pas complètement interopérable. Certes le capteur envoie des données au format CBOR qui peuvent être interprétés par l’autre extrémité, mais le récepteur ne sait pas :

- qu’il s’agit d’une série temporelle codée avec des deltas ;
- que les données ont été multipliées par 100 pour pouvoir envoyer des nombres entiers, plus compacts sans perdre trop de précision ;
- que le pas de mesure est de 10 secondes
- que les données concernent le taux d’humidité.

Ces informations ont été précisées dans le programme `display_server.py`, de même la transformation de la structure de tableau de la série temporelle en un dictionnaire avec des mots clés spécifique à Beebotte ont été gravé dans le programme.

Nous verrons par la suite comment améliorer cette interopérabilité.

8.6 et SenML ?

Dans la communication avec Beebotte, le site structure l’envoi des mesures en définissant un dictionnaire JSON avec des mots clés particuliers. Pour utiliser un autre site, le format des échanges doit être modifié même si les informations restent identiques.

De plus, lors de la configuration des ressources sur le site de Beebotte, la nature de la mesure a

du être précisée ; par exemple, s'il s'agit d'une température, d'un taux d'humidité... Il faut également parfois indiquer le type de la mesure (texte, entier, flottant...) voire les unités.

Sensor Measuring List (SenML) défini dans le [RFC 8428](#) propose une structuration des données fournie par le capteur. Pour réduire l'impact de la transmission, les noms des champs ont été choisis pour être le plus compact possible. Par exemple, la lettre v va indiquer une valeur (à comparer avec la clé data utilisée lors de la communication avec Beebotte). Pour être encore plus compact, la représentation en CBOR utilisera des entiers courts au lieu de caractères.

Il est également possible de transporter l'unité de la mesure avec le mot clé u .

SenML ne définit pas que des unités du système international, mais également des unités secondaires pour limiter la taille de la représentation. Il sera plus compact de transmettre :

```
{"u": "MHz", "v": 868}
```

que

```
{"u": "Hz", "v": 868000000}.
```

Le standard définit aussi des temps de base et des valeurs de base auxquelles les temps et les valeurs vont se référer; ce qui permet également de réduire la taille des valeurs. Finalement, le ou les objets peuvent s'identifier dans les données transmises en définissant un nom de base (*bn : base name*), le nom du capteur (*n : name*) vient compléter le nom de base.

Émission

Listing 8.5 – minimal_senml_client.py

```

1 from virtual_sensor import virtual_sensor
2 import time
3 import socket
4 import json
5 import kpn_senml as senml
6 import pprint
7 import binascii
8 import datetime
9 import time
10 import pprint
11
12 NB_ELEMENT = 5
13
14
15 temperature = virtual_sensor(start=20, variation = 0.1)
16 pressure = virtual_sensor(start=1000, variation = 1)
17 humidity = virtual_sensor(start=30, variation = 3, min=20, max=80)
18
19 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
20
21 while True:
22     pack = senml.SenmlPack("device1")
23     pack.base_time = time.mktime( datetime.datetime.now().timetuple() )
24
25     for k in range(NB_ELEMENT):
26         t = round(temperature.read_value(), 2)
27
28         if k > 0:
29             pack.add("temperature", t)
30         else:
31             pack.add("temperature", t, base_time=True)
```

```

29     h = round(humidity.read_value(), 2)
30     p = int(pressure.read_value() *100) # unit is Pa not hPa
31
32     rec = senml.SenmlRecord("temperature",
33         unit=senml.SenmlUnits.SENML_UNIT_DEGREES_CELSIUS,
34         value=t)
35     rec.time = time.mktime(datetime.datetime.now().timetuple())
36     pack.add(rec)
37
38     rec = senml.SenmlRecord("humidity",
39         unit=senml.SenmlUnits.SENML_UNIT_RELATIVE_HUMIDITY,
40         value=h)
41     pack.add(rec)
42
43     rec = senml.SenmlRecord("pressure",
44         unit=senml.SenmlUnits.SENML_UNIT_PASCAL,
45         value=p)
46     pack.add(rec)
47
48     time.sleep(10)
49
50     pprint.pprint(json.loads(pack.to_json()))
51     print ("JSON length:", len(pack.to_json()), "bytes")
52     print ("CBOR length:", len(pack.to_cbor()), "bytes")
53
54     s.sendto(pack.to_cbor(), ("127.0.0.1", 33033))

```

Le programme `minimal_senml_client.py` illustre le fonctionnement de SenML. Il repose sur deux objets :

- l'objet `SenmlPack` inclus les informations commune à l'objet, comme le nom de base (ici `device1` ligne 23) ou la base de temps, ligne 24.
- l'objet `SenmlRecord` contient une mesure où l'on peut préciser son nom, son unité et sa valeur (lignes 32, 38 et 43). Le temps est également précisé ligne 35. Ces enregistrements sont ajoutés à l'objet `pack`.

Le programme récupère les trois valeurs de température, humidité et pression (lignes 27 à 29) en les arrondissant à 2 chiffres après la virgule pour la température et l'humidité et converti la pression, d'hecto Pascal en Pascal puisque c'est l'unité définie par SenML.

Les mesures se font toutes les 10 seconde (délais ligne 48) et quand le nombre de mesures défini ligne 13 est atteint, le codage SenML en CBOR est envoyé au serveur.

```

[{'bn': 'device1', 'bt': 1640110457.0,
 'n': 'temperature', 't': 0.0, 'u': 'Cel', 'v': 19.98},
 {'n': 'humidity', 'u': '%RH', 'v': 28.46},
 {'n': 'pressure', 'u': 'Pa', 'v': 100093}]
JSON length: 177 bytes
CBOR length: 104 bytes

```

Ce premier listing montre le premier enregistrement pour les trois grandeurs mesurées. Il s'agit d'un tableau de 3 éléments. Le premier contient les valeurs de bases (ici le nom et l'heure de référence) suivi de la grandeur à mesurer, de son unité et sa valeur. Le deuxième et le troisième éléments, mettent à jours le nom de la grandeur, son unité et sa valeur, les autres informations

précédemment définies restent valables.

```
[{'bn': 'device1', 'bt': 1640110457.0,
 'n': 'temperature', 't': 0.0, 'u': 'Cel', 'v': 19.98},
 {'n': 'humidity', 'u': '%RH', 'v': 28.46},
 {'n': 'pressure', 'u': 'Pa', 'v': 100093},
 {'n': 'temperature', 't': 10.0, 'u': 'Cel', 'v': 20.03},
 {'n': 'humidity', 'u': '%RH', 'v': 26.86},
 {'n': 'pressure', 'u': 'Pa', 'v': 100065}]
JSON length: 318 bytes
CBOR length: 188 bytes
```

Quand on ajoute 10 secondes plus tard de nouvelles mesures, un temps relatif de 10 secondes est indiqué pour l'enregistrement des températures et il reste valable pour les enregistrements suivants.

Question 8.6.1: codage

A quoi correspond la clé 'u' : 'Cel' que l'on retrouve dans la structure précédente ?

Question 8.6.2: Accroissement

Dans les deux représentations JSON et CBOR, de combien la taille est-elle accrue par l'ajout des mesures effectuées ? d'où viennent ces différences ?

Question 8.6.3: Une seule grandeur

Si on ne s'intéressait qu'à une seule grandeur, par exemple l'humidité. A quoi ressemblerait la structure SenML en JSON ?

Réception

Le traitement par le module SenML tel qu'il est mis en œuvre n'est pas complet, il ne gère pas correctement les timestamps. Mais, il n'est pas vraiment nécessaire pour traiter ces messages. En effet, comme on l'a vu précédemment, les objets SenML sont cumulatifs, une clé reste présente dans les enregistrements suivants sauf si elle est redéfinie.

Listing 8.6 – minimal_senml_server.py

```
1 import socket
2 import pprint
3 import binascii
4 import pprint
5 import cbor2 as cbor
6
7 import beebotte
8 import config_bbt #secret keys
9
10 naming_map = {'bn': -2, 'bt': -3, 'bu': -4, 'bv': -5, 'bs': -16,
11                 'n': 0, 'u': 1, 'v': 2, 'vs': 3, 'vb': 4,
12                 'vd': 8, 's': 5, 't': 6, 'ut': 7}
13
14 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
15 s.bind((0.0.0.0, 33033))
```

```

17 bbt = beebotte.BBT(config_bbt.API_KEY, config_bbt.SECRET_KEY)

19
20 while True:
21     data, addr = s.recvfrom(1500)

23     sml_data = cbor.loads(data)

25     sml_record = {}
26     bbt_record = []

27     for e in sml_data:
28         sml_record = {**sml_record, **e} # merge dict
29         print(sml_record)

31     ts = sml_record[naming_map["t"]]
32     if naming_map["bt"] in sml_record:
33         ts += sml_record[naming_map["bt"]]

35     res = sml_record[naming_map["n"]]

37     data = sml_record[naming_map["v"]]
38     if naming_map["bv"] in sml_record:
39         data += sml_record[naming_map["bv"]]

41

43     bbt_record.append({"resource": res, "data": data, "ts": ts*1000})

45     pprint.pprint(bbt_record)
46     channel = sml_record[naming_map["bn"]]
47     bbt.writeBulk(channel, bbt_record)

```

Le programme `minimal_senml_server.py` va convertir le format SenML codé en CBOR dans le format attendu par Beebotte. La version CBOR utilise des nombres plutôt que des tags. Le dictionnaire `naming_map` défini lignes 10 à 12 permet la correspondance utilisée par la suite pour rendre le code plus lisible.

Les lignes 14 à 17 initialisent les communications venant du capteur et celles allant à Beebotte.

Les données reçues ligne 21 sont transformées en structure Python ligne 23. Cette correspondance est possible car Python autorise des clés numériques et celles-ci ne sont pas répétées plusieurs fois dans une map CBOR.

La boucle commençant ligne 28 permet d'explorer tous les éléments du tableau SenML, les nouvelles entrées sont fusionnées avec les anciennes (ligne 29)³.

Les informations concernant le temps sont ensuite recherchées. D'abord le temps (ligne 32) et si un temps de base existe (ligne 33) il est ajouté. On procède de même pour la valeur (lignes 38 à 40). Pour le nom, il n'y a pas de concaténation car le nom de base sera utilisé comme canal Beebotte, il est récupéré à la fin ligne 46.

A partir de ces informations, la structure attendue par Beebotte est construite ligne 43 en ajoutant le dictionnaire dans le tableau `bbt_record`.

Ligne 47, l'information est envoyée à Beebotte. Si les clés d'authentification, le nom du canal et des ressources sont corrects, les informations s'affichent sur le site, comme précédemment.

3. Dans les versions plus récentes de Python, il est possible d'utiliser l'opérateur `|`.

```
{0: 'temperature', 1: 'Cel', 2: 19.44, 6: 0.0, -3: 1640168049.0, -2: 'device1'}
{0: 'humidity', 1: '%RH', 2: 24.13, 6: 0.0, -3: 1640168049.0, -2: 'device1'}
{0: 'pressure', 1: 'Pa', 2: 101080, 6: 0.0, -3: 1640168049.0, -2: 'device1'}
{0: 'temperature', 1: 'Cel', 2: 19.48, 6: 10.0, -3: 1640168049.0, -2: 'device1'}
{0: 'humidity', 1: '%RH', 2: 21.85, 6: 10.0, -3: 1640168049.0, -2: 'device1'}
{0: 'pressure', 1: 'Pa', 2: 101090, 6: 10.0, -3: 1640168049.0, -2: 'device1'}
{0: 'temperature', 1: 'Cel', 2: 19.57, 6: 20.0, -3: 1640168049.0, -2: 'device1'}
{0: 'humidity', 1: '%RH', 2: 21.09, 6: 20.0, -3: 1640168049.0, -2: 'device1'}
{0: 'pressure', 1: 'Pa', 2: 101058, 6: 20.0, -3: 1640168049.0, -2: 'device1'}
{0: 'temperature', 1: 'Cel', 2: 19.52, 6: 30.0, -3: 1640168049.0, -2: 'device1'}
{0: 'humidity', 1: '%RH', 2: 20.39, 6: 30.0, -3: 1640168049.0, -2: 'device1'}
{0: 'pressure', 1: 'Pa', 2: 101120, 6: 30.0, -3: 1640168049.0, -2: 'device1'}
{0: 'temperature', 1: 'Cel', 2: 19.44, 6: 40.0, -3: 1640168049.0, -2: 'device1'}
{0: 'humidity', 1: '%RH', 2: 21.94, 6: 40.0, -3: 1640168049.0, -2: 'device1'}
{0: 'pressure', 1: 'Pa', 2: 101128, 6: 40.0, -3: 1640168049.0, -2: 'device1'}
[{'data': 19.44, 'resource': 'temperature', 'ts': 1640168049000.0},
 {'data': 24.13, 'resource': 'humidity', 'ts': 1640168049000.0},
 {'data': 101080, 'resource': 'pressure', 'ts': 1640168049000.0},
 {'data': 19.48, 'resource': 'temperature', 'ts': 1640168059000.0},
 {'data': 21.85, 'resource': 'humidity', 'ts': 1640168059000.0},
 {'data': 101090, 'resource': 'pressure', 'ts': 1640168059000.0},
 {'data': 19.57, 'resource': 'temperature', 'ts': 1640168069000.0},
 {'data': 21.09, 'resource': 'humidity', 'ts': 1640168069000.0},
 {'data': 101058, 'resource': 'pressure', 'ts': 1640168069000.0},
 {'data': 19.52, 'resource': 'temperature', 'ts': 1640168079000.0},
 {'data': 20.39, 'resource': 'humidity', 'ts': 1640168079000.0},
 {'data': 101120, 'resource': 'pressure', 'ts': 1640168079000.0},
 {'data': 19.44, 'resource': 'temperature', 'ts': 1640168089000.0},
 {'data': 21.94, 'resource': 'humidity', 'ts': 1640168089000.0},
 {'data': 101128, 'resource': 'pressure', 'ts': 1640168089000.0}]
```

Le listing précédent montre cette transformation. Les premières lignes correspondent aux enregistrements fusionnées et le tableau final, ce qui a été envoyé à Beebotte.

Question 8.6.4: base value

Pourrait-on utiliser le champ SenML *base value* pour diminuer la taille des données de pression atmosphérique ?

9. Découvrons le LoPy

Les programmes relatifs à cette section se trouvent dans le répertoire `plido-tp3` pour le serveur et `pycom` pour le LoPy.

9.1 Introduction

Grâce aux émulateurs de capteurs décrits au chapitre précédent, vous avez pu appliquer les concepts essentiels de l’IoT sur votre ordinateur.

Cependant, si vous le pouvez, nous vous invitons à le faire sur de vrais objets connectés en utilisant des **LoPy4** (plateforme de prototypage IoT) de la société **Pycom** et des capteurs de température, humidité et pression **BME280** (cf. figure 9.1).

Un LoPY4 se programme en Python (ou plutôt **micro-python** qui est la version du langage pour systèmes embarqués) pour traiter les données. Dans un premier temps, nous allons utiliser le Wi-Fi pour communiquer avec votre ordinateur mais, par la suite, nous mettrons en place une

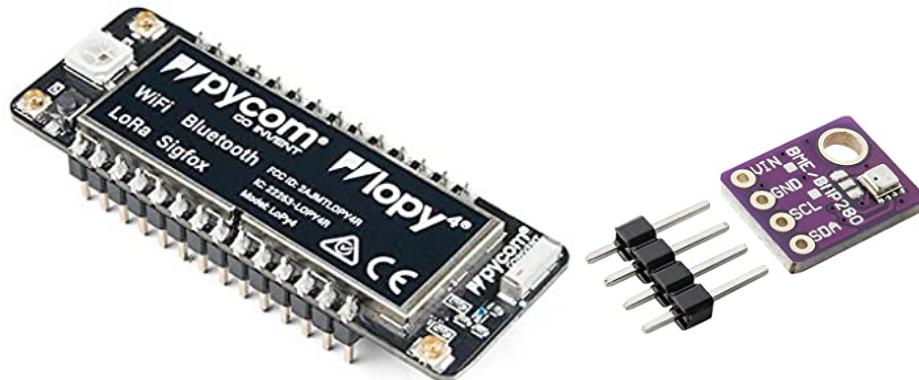


FIGURE 9.1 – LoPY4 et capteur BME280

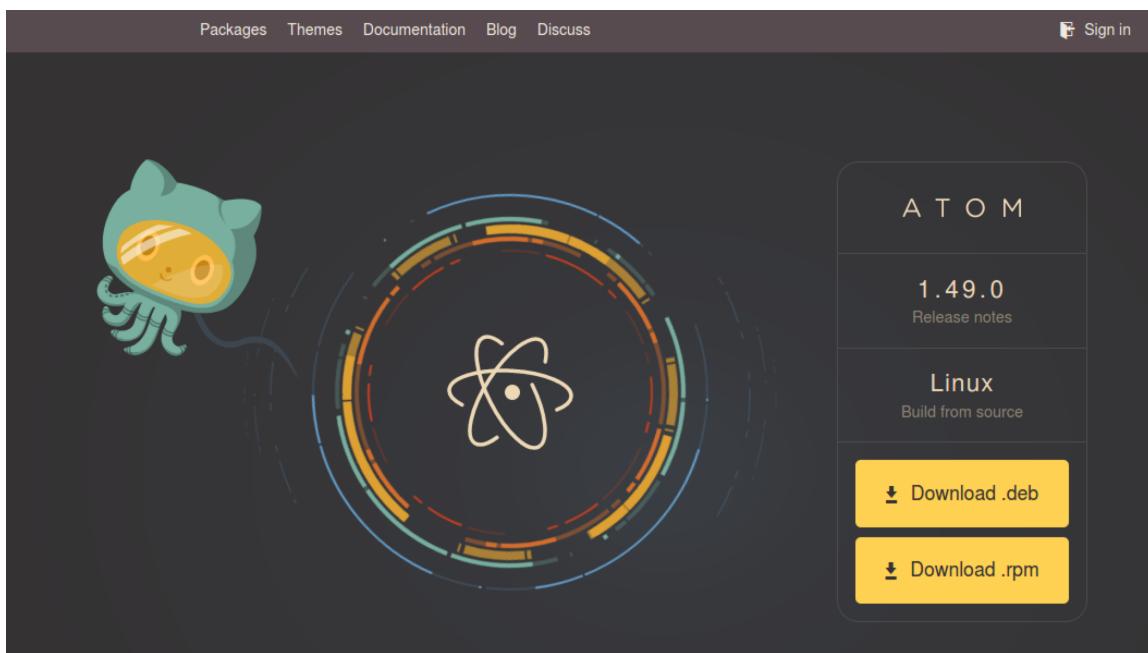


FIGURE 9.2 – Page d'accueil d'Atom

communication via **LoRaWAN** ou **Sigfox** qui peut vous demander plus de configuration mais vous permettra de mieux comprendre ces protocoles.

Même si vous n'avez pas de LoPy4, vous pouvez parcourir cette section pour voir les contraintes supplémentaires liées aux objets connectés.

9.2 Installation d'Atom

Atom est un éditeur de texte performant, spécifiquement conçu pour le codage en différents langages. Atom va nous aider à programmer en Python et va également gérer la communication avec notre LoPy via le port **USB** (grâce au plugin **pymakr**). Atom fonctionne à peu près de la même manière sur **Mac OS**, **Windows** et **Linux**, mais en s'adaptant aux particularités du système d'exploitation (place dans les menus, nom des liens séries...). Donc, il se peut que vous ayez quelques différences entre ce que vous avez dans cet ouvrage et l'écran de votre ordinateur. Les menus et sites Web indiqués peuvent également changer au cours du temps même si nous nous efforçons de faire des mises à jour régulières du cours.

Pour commencer à programmer avec votre LoPy, vous devez installer sur votre ordinateur le logiciel Atom (voir sur <http://atom.io>). Vous pouvez télécharger le package correspondant à votre système d'exploitation (cf. figure suivante).

— Pour Mac et Windows, cliquez sur l'icône "télécharger" pour l'installer¹.

1. il y a des risques d'incompatibilité des dernières versions d'Atom avec le package de gestion du LoPy (pymakr). Nous vous recommandons d'utiliser une version plus ancienne, comme la 1.43 disponible dans les archives d'Atom Release <https://github.com/atom/atom/releases/tag/v1.43.0>(AtomSetup-x64.exe pour Windows ou atom-mac.zip pour Mac OS).

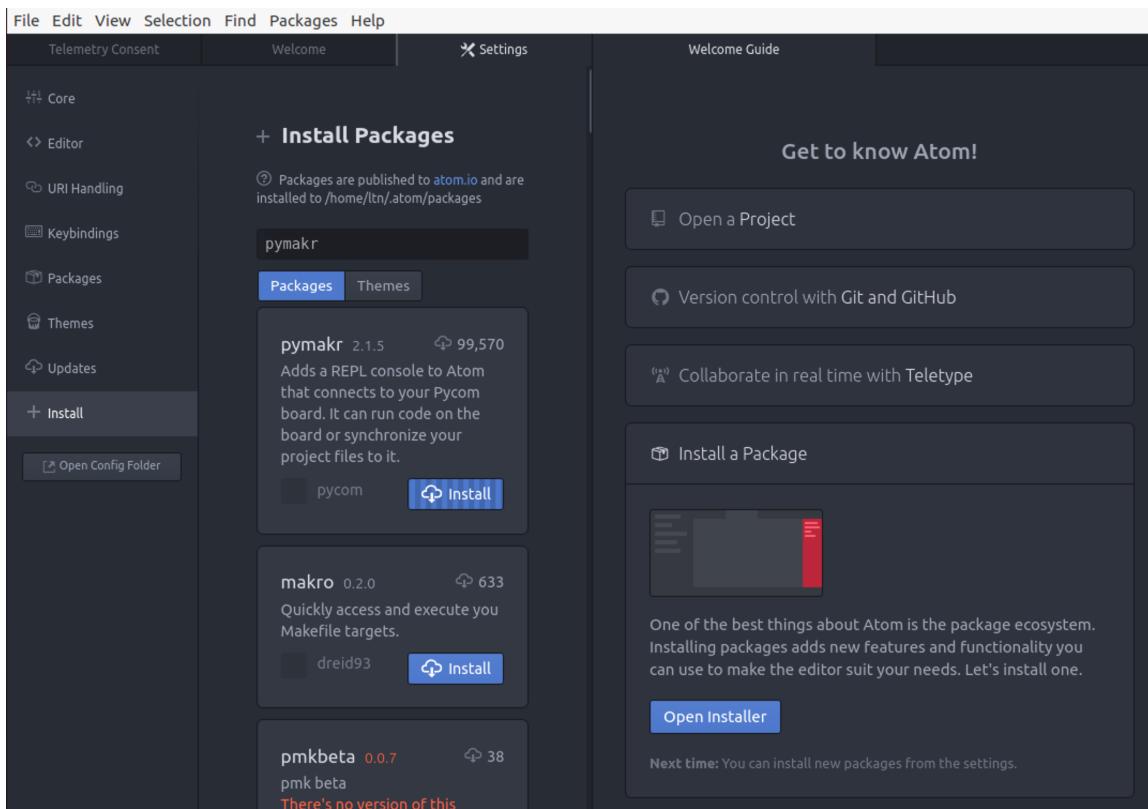


FIGURE 9.3 – Installation de Paquetages

— Pour Linux, téléchargez le .deb et tapez `sudo dpkg -i atom-amd64.deb`². Lancez Atom en cliquant sur l’icône ou, sous Linux, en tapant `atom` dans un terminal.

Lancez Atom en cliquant sur l’icône ou, sous Linux, en tapant `atom` dans un terminal. L’écran d’accueil apparaît.

9.2.1 Communiquez avec votre Pycom

Pour communiquer avec le Pycom à travers Atom, vous devez installer le package `pymakr`.

Cliquez sur *Install a Package* puis *Open Installer*. Une autre fenêtre s’ouvre (cf. figure 9.3). Tapez `pymakr` dans le menu. Un package apparaît portant ce nom. Cliquez sur *Install*. L’installation peut prendre plusieurs minutes. Vous avez le temps de prendre un café.

Une fois le café bu et l’installation terminée, une nouvelle fenêtre (terminal) s’ouvre en bas d’Atom.

Ce terminal (cf. figure 9.4 on the facing page) vous permettra de dialoguer avec le LoPy. Branchez le LoPy à votre ordinateur. Vous devriez voir l’invite `>>>` caractéristique d’un interpréteur Python³.

Toutes les commandes que vous allez taper dans cette fenêtre vont s’exécuter sur votre LoPy. Par

2. Il est possible qu’un message vous dise que git n’est pas installé. Dans ce cas, tapez `sudo apt-get install`

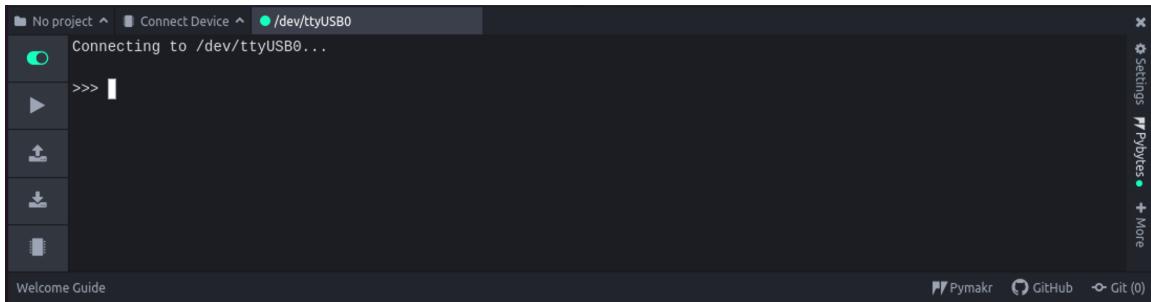


FIGURE 9.4 – Fenêtre Pymakr

exemple, si vous tapez⁴ :

```
Connecting to /dev/ttyUSB0...
>>> 1+1
2
>>>
```

L'addition se fait sur le LoPy.

Sur le coté gauche de la fenêtre pymakr, plusieurs icônes sont présentes :

- l'interrupteur permet d'activer ou de désactiver la connexion avec le LoPy ;
- le triangle permet d'exécuter le programme affiché dans la fenêtre d'Atom sur le LoPy ;
- la flèche vers le haut, permet de recopier le répertoire actif dans la mémoire du LoPy. Cela sera utile pour installer de nouveaux modules sur le LoPy ;
- inversement la flèche vers le bas, permet de recopier la mémoire du LoPy sur l'ordinateur ;
- le processeur permet d'avoir des informations sur le LoPy.

Sur la partie droite, l'onglet vertical *Setting* permet de modifier les paramètres de connexion avec le LoPy.

9.2.2 Installez votre environnement de travail

Pour programmer le LoPy, il faut récupérer les modules micropython. Le dépôt peut être téléchargé dans le répertoire de votre choix :

```
> git clone https://github.com/ltn22/PLIDObis.git
```

Dans le menu *Files>Open Folder* d'Atom, sélectionnez le répertoire pycom du dépôt téléchargé, et validez. Sur la partie gauche de l'écran, l'ensemble des fichiers composant ce répertoire apparaissent. Il y en a beaucoup, car ils vont nous servir par la suite.

En cliquant dans la fenêtre **pymakr** sur le bouton *Upload project to device*, les fichiers de ce répertoire vont être copiés dans la mémoire du LoPy. Par la suite, si un module est modifié, il devra être resynchronisé dans la mémoire du LoPy.

git et suivez les instructions).

3. Sous Linux, vous devez être membre du groupe dialout pour pouvoir gérer la communication sur le port USB. Si vous ne voyez pas l'invite, tapez `sudo adduser login dialout` en remplaçant login par le nom de votre compte Linux. Reconnectez-vous sous votre compte.

4. Les fenêtres sur fond gris montrent le code micropython et leur résultat.

9.3 Connexion au réseau Wi-Fi

Pour rattacher de LoPY à un réseau **Wi-Fi**, il doit dans un premier temps être configuré via la liaison USB de l'ordinateur pour lui donner les paramètres nécessaires à la connexion.

Le fichier boot.py a été copié lors du téléchargement des fichiers dans la mémoire du LoPy. Au démarrage du LoPy, ce programme va chercher à se connecter à un réseau Wi-Fi. Comme le nom du réseau et la clé secrète n'ont pas été fournie, il n'y arrive pas. Le LoPy se transforme en point d'accès. Au démarrage, le LoPy a dû afficher le message suivant, indiquant que le LoPy devient point d'accès Wi-Fi et va déployer son propre réseau sur lequel votre ordinateur peut se connecter. Le nom de ce réseau est de la forme PLIDO_XXXX où XXXX est une séquence hexadécimal propre à l'équipement. La clé est www.pycom.io. Le LoPy a l'adresse 192.168.4.1 sur ce réseau.

```
Failed to connect to any known network, going into AP mode
To connect look for 'PLIDO_5bac' access point, key = 'www.pycom.io'
```

Mais ce n'est pas très intéressant car votre ordinateur va perdre sa connexion à l'internet. Pas très pratique pour suivre le MOOC. Avant d'afficher ce message, le LoPy a montré la liste des réseaux Wi-Fi qu'il a détecté. Vous pouvez à l'inverse le connecter à un de ces réseaux en renseignant le fichier wifi_conf.py qui se trouve dans le répertoire pycom.

Listing 9.1 – wifi_conf.py

```
known_nets = {
    'MON_SSID': { 'pwd': 'MON_MOT_DE_PASSE' }
}
```

MON_SSID doit être remplacé par le nom du réseau Wi-Fi ou Service Set IDentifier (SSID) et MON_MOT_DE_PASSE par la clé qui y est associée. Notez que plusieurs réseaux Wi-Fi peuvent être ajoutés, puisque MON_SSID est vu comme une clé de l'objet JSON. L'édition de fichier s'est faite sur l'ordinateur, il doit être recopié dans la mémoire du LoPy en cliquant sur la flèche vers le haut.

Le Pycom redémarre et doit maintenant afficher un message du genre :

```
net to use ['MONWIFI']
Connected to MONWIFI with IP address: 192.168.1.76
Pycom MicroPython 1.20.2.r1 [v1.11-a5aa0b8] on 2020-09-09; LoPy4 with ESP32
Type "help()" for more information.
>>>
```

Il est possible de pinguer ou de se connecter avec **FTP** ou **telnet** en utilisant cette adresse IP.

```
# telnet 192.168.1.76
Trying 192.168.1.86...
Connected to 192.168.1.86.
Escape character is '^].
MicroPython v1.8.6-760-g90b72952 on 2017-09-01; LoPy with ESP32
Login as: micro
Password: python
Login succeeded!
Type "help()" for more information.
>>>
```

Youtube



Ça peut être utile pour suivre le comportement de votre objet sans lancer atom si l'objet n'est plus connecté via la liaison USB à l'ordinateur.

Atom peut également être configuré pour utiliser cette adresse IP. La configuration se fait dans le menu *setting*, et en entrant l'adresse ip du LoPy et en désactivant *auto connect*. Au prochain lancement d'Atom, il sera possible joindre le LoPy en Wi-Fi.

9.4 Mise en place d'un client

Un programme relativement simple permet de vérifier la communication entre le LoPy et le serveur. La commande **ifconfig**⁵ donne l'adresse IP du serveur. L'adresse doit être différente de celle que l'on avait obtenu sur le LoPy.

Si le serveur tourne dans un environnement local, l'adresse devrait commencer par 192.168 ou 10. Si le LoPy et l'ordinateur sont connectés au même réseau Wi-Fi, les premiers chiffres doivent être identiques.

Si le serveur tourne sur un serveur à l'extérieur (i.e. le *cloud*), l'adresse est quelconque.

La commande suivante est tapée depuis le terminal de l'ordinateur, on remarque les deux interfaces disponibles l'une pour le réseau Ethernet ou Wi-Fi et l'une pour le *loopback*, les noms des interfaces peuvent changer d'une configuration à une autre :

```
> ifconfig
eth1      Link encap:Ethernet HWaddr 10:65:30:b0:54:bf
          inet addr:192.168.1.237 Bcast:192.168.1.255 Mask:255.255.255.0
          inet6 addr: fe80::d8ac:86e7:8bdb:e333/64 Scope:Unknown
            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0
            RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Unknown
            UP LOOPBACK RUNNING MTU:1500 Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0
            RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

Le programme `minimal_server.py` présenté au chapitre 7.1.1 on page 82 n'a pas été modifié et attend des données sur le port 33033.

Listing 9.2 – sending_client.py

```
1 import socket
2
3 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

5. Sous Linux, il faut ajouter le paquetage **net-tools** `sudo apt install net-tools`.



```
4 s.sendto("message", ("192.168.1.237", 33033))
```

Le programme `sending_client.py` doit être modifié (ligne 4) pour prendre en compte l'adresse IP du serveur obtenue avec la commande `ifconfig`. Si à chaque exécution sur le LoPy de ce programme, le serveur reçoit la valeur, la communication est établie entre les deux équipements.

```
> python3 minimal_server.py
b'message' => b'6d657373616765'
b'message' => b'6d657373616765'
```

9.5 BME 280

Au lieu de générer de fausses données, nous allons dans cette section utiliser un vrai capteur de température humidité pression : le BME 280 de chez Bosch.

9.5.1 Le bus I2C

Le bus I2C est normalisé par le fabricant de composants électroniques **NXP**⁶ ce qui permet une meilleure interopérabilité entre les composants.

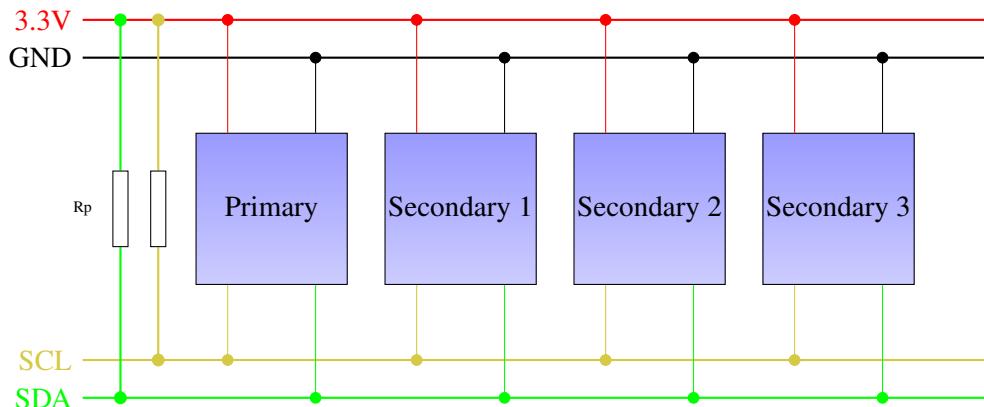


FIGURE 9.5 – bus I2C

Sur un des fils le signal d'horloge va être émis par le primaire. Sur l'autre fil, les données seront codées soit dans le sens primaire/secondaire, soit dans l'autre. Comme avec **Modbus**, les communications entre un secondaire et le primaire seront gérées par le primaire. Chaque secondaire est configuré avec une adresse unique sur le bus. Soit le maître envoie des données vers cette adresse⁷, soit le primaire interroge l'esclave pour obtenir ses données. Le fil sera donc exploité dans les deux directions.

La lecture de l'information binaire se fait lorsque le signal d'horloge est à l'état haut (cf. figure 9.6 on the facing page). Quand le signal SDA est à l'état haut, un bit à 1 est transmis et dans l'état bas

6. <https://www.nxp.com/docs/en/user-guide/UM10204.pdf>

7. Il s'agit d'un abus de langage, en effet les données émises par le primaire sont reçues par l'ensemble des secondaires, mais seul celui qui est destinataire (i.e. qui reconnaît son adresse) va les traiter, les autres équipements ignoreront l'information.

un bit à 0 est transmis. Les changements d'état du signal SDA se font donc quand le signal d'horloge est à l'état bas.

Il existe malgré tout deux exceptions : si le signal SDA passe de l'état haut à l'état bas tandis que le signal d'horloge est à l'état haut cela indique un début de transmission de données. Si le signal SDA passe de l'état bas à l'état haut dans les mêmes conditions, cela indique la fin de transmission de données. Entre les deux les données binaires forment une trame (ou un PDU dans le vocabulaire ISO) qui est structuré comme indiqué figure 9.6.

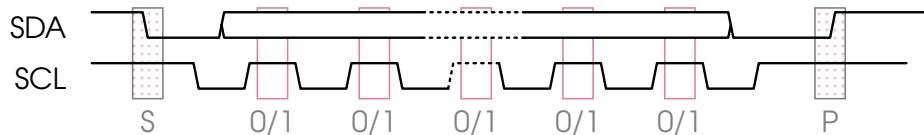


FIGURE 9.6 – Exemple de communication avec le bus I2C

La figure 9.7 donne les formats les plus utilisés.

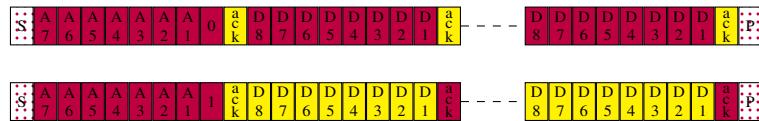


FIGURE 9.7 – Exemple de communication avec le bus I2C

Le premier train binaire illustre la transmission de données du primaire vers un secondaire. Le primaire commence par émettre un signal non binaire (S) indiquant un début de transmission. Les 7 bits suivants donnent l'adresse du secondaire et le bit suivant indique si le primaire veut envoyer des données (valeur à 0) ou recevoir des informations du secondaire (valeur à 1).

Si un secondaire reconnaît son adresse sur le bus, alors il écrit le bit suivant dans le train binaire. Le primaire est donc informé en lisant cette valeur que le secondaire est bien présent sur le bus et qu'il peut recevoir des données. Le primaire va donc les envoyer octet par octet. Chaque octet étant acquitté de la même manière par le secondaire. Le train binaire se termine par le signal non binaire (P).

Dans le cas où le primaire souhaite recevoir, une fois le bit de début et les bits de l'adresse émis, le huitième bit est positionné à 1. Le secondaire acquitte puis transmet ses octets que le primaire acquitte.

Question 9.5.1: scan

Le module I2C du LoPy dispose d'une fonction `scan` qui affiche les adresses des secondaires connectés. Comment cette détection est possible ?

Question 9.5.2: Diffusion

Est-ce que la norme une adresse qui permet de parler à tous les secondaires en même temps ?

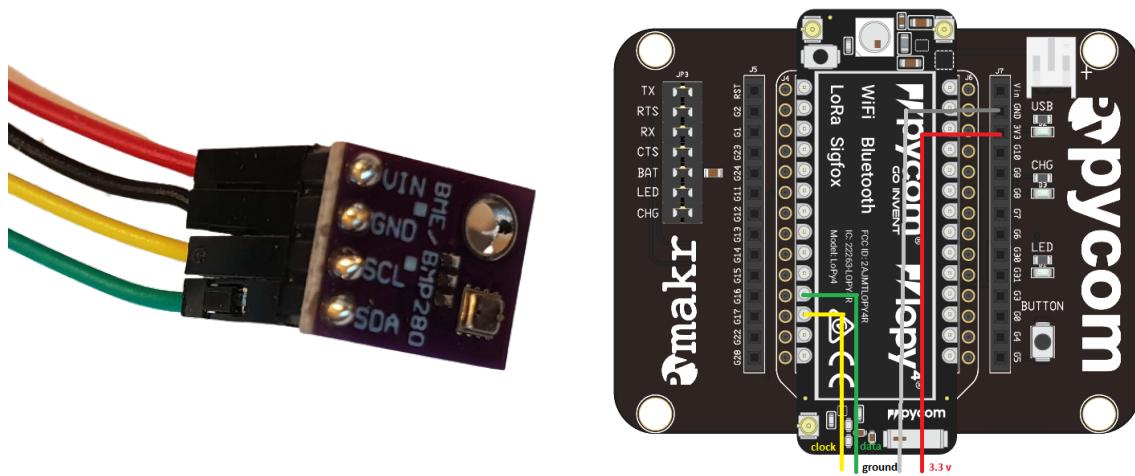


FIGURE 9.8 – capteur BME280 et connecteurs

9.5.2 Mesure de la température

La communication entre le LoPy et le composant se fait via le bus **I2C**. Il nous faut donc quatre fils pour le relier (cf. figure 9.8) :

- la masse (GND),
- une alimentation électrique de 3.3v (VIN),
- un fils pour l'horloge (SCL) et
- un autre finalement pour les données (SDA).

Pour ce faire, il faut connecter :

- la masse GND du LoPy sur la broche GND du composant avec un fil noir,
- l'alimentation en 3.3V du LoPy (3V3) sur VIN du composant avec un fil rouge (ce port s'appelle également **DCC** sur certaines cartes),
- Le signal d'horloge du port G18/P10 du LoPy (ou G17 sur les versions LoPy 1) sur le port SCL du composant avec un fil jaune (ce port s'appelle également **CLC** sur certaines cartes),
- Le fil de données du port G16/P9 du Pycom sur le port SDA du composant avec un fil vert.

Si le BME280 est connecté correctement sur les connecteurs du LoPy, vous devez obtenir le résultat suivant :

```
>>> Running BME280.py
>>>
>>>
[118]
temp 550576 27.98 - hum 26626 48.784 % - pres 389338 pres 1004.494 hPa [delta -389338 ]
temp 551952 28.42 - hum 26587 48.557 % - pres 389712 pres 1004.527 hPa [delta -374 ]
temp 551792 28.37 - hum 26577 48.491 % - pres 389664 pres 1004.621 hPa [delta 48 ]
temp 551712 28.34 - hum 26594 48.596 % - pres 389648 pres 1004.654 hPa [delta 16 ]
temp 551632 28.32 - hum 26587 48.546 % - pres 389616 pres 1004.713 hPa [delta 32 ]
```

Vous pouvez soit toucher le capteur, soit souffler dessus pour faire augmenter la température ou la pression.

Listing 9.3 – BME280.py

```
282 if __name__ == "__main__":
```

```

284     from machine import I2C
285     import time
286
287     i2c = I2C(0, I2C.MASTER, baudrate=400000)
288     print(i2c.scan())
289
290     bme = BME280(i2c=i2c)
291
292     ob = 0
293
294     while True:
295         ar = bme.read_raw_temp()
296         a = bme.read_temperature()
297
298         br = bme.read_raw_pressure()
299         b = bme.read_pressure()
300
301         cr = bme.read_raw_humidity()
302         c = bme.read_humidity()
303
304         print("temp", ar, a,
305               "hum", cr, c,
306               "%pres", br,
307               "pres", b,
308               "hPa[delta]", ob - br, "]")
309
310         time.sleep(5)

```

Le programme BME280.py peut fonctionner comme un module mais la dernière partie donne un exemple d'exploitation des résultats :

- Le programme principal commence par importer (ligne 283) le module gérant le bus I2C. Il est invoqué à la ligne 286. Le LoPy va gérer la communication avec le BME280, donc l'adresse est 0 et son statut estMASTER. La vitesse de communication est ensuite spécifiée.
- La ligne suivante scanne le bus pour trouver des composants. Si tout va bien, il devrait en trouver un à l'adresse 118 qui correspond à l'adresse par défaut du BME280⁸. Sinon, revoyez votre câblage.
- Le module BME280 est initialisé en lui passant en paramètre la référence du bus I2C précédemment défini.
- Le programme va ensuite afficher les valeurs captées par le composant. Il existe deux types de valeurs : brutes (*raw*) et calibrées. Les premières réagissent plus rapidement aux changements cependant sont beaucoup plus bruitées que les seconde qui subissent un traitement mathématique.
Ainsi, les première et deuxième colonnes donnent les températures brute et calibrée. On y accède par les méthodes `read_raw_temp` et `read_temperature`. Il en va de même pour les deux autres grandeurs, humidité et pression.
- Le programme affiche également l'écart de pression brute entre deux mesures. Cela permet de mettre plus facilement en évidence le fait que l'on souffle sur le capteur.

⁸. Si une autre valeur est indiquée, vous pouvez à l'instantiation du module BME280, ligne 289, ajouter le paramètre `addr=valeur`.

9.6 Thermomètre Wi-Fi

Vous avez maintenant tous les outils pour récupérer la température de votre logement, la transmettre à votre ordinateur via le Wi-Fi, et la transmettre à Beebotte pour l'afficher. Il y a très peu de changement par rapport à la version complètement sur ordinateur. Le programme de transformation de la structure CBOR de représentation des séries temporelles en JSON compréhensible par Beebotte reste le même. Il faudra juste modifier le nom du capteur d'humidité à température.

Le programme que vous avez construit lors du TP précédent utilisait le module `virtual_sensor` pour produire des séries temporelles aléatoires symbolisant le comportement d'un capteur. Maintenant que nous avons un **BME280**, nous allons pouvoir traiter de vraies valeurs.

Youtube



Le programme `wifi_temperature.py` montre cette adaptation.

Listing 9.4 – wifi_temperature.py

```

1 import BME280
2 import time
3 import socket
4 import kpn_senml.cbor_encoder as cbor
5 from machine import I2C
6
7
8 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
9 NB_ELEMENT = 30
10 t_history = []
11
12 i2c = I2C(0, I2C.MASTER, baudrate=400000)
13 print(i2c.scan())
14
15 bme = BME280.BME280(i2c=i2c)
16
17 while True:
18
19     t = int(bme.read_temperature()*100)
20
21     # No more room to store value, send it.
22     if len(t_history) == 0:
23         t_history = [t]
24     elif len(t_history) >= NB_ELEMENT:
25         print("send")
26         s.sendto(cbor.dumps(t_history), ("192.168.1.47", 33033))
27         t_history = [t]
28     else:
29         t_history.append(t)
30
31     prev = t
32
33     print(len(t_history), len(cbor.dumps(t_history)), t_history)
34
35     time.sleep(10)

```

Au niveau des importations de modules, BME280 et remplacent `virtual_sensor`, et le module CBOR est celui de `kpn_senml`. Mais son comportement reste le même, en particulier la fonction `dumps` qui convertit une structure Python en CBOR. Il vous reste à adapter la ligne 26 pour mettre l'adresse IP de votre ordinateur.

Côté ordinateur, vous devez relancer le programme `display_server.py`, mais en modifiant le nom du capteur de "humidity" à "temperature".

Sur votre compte Beebotte, au bout de 300 secondes, vous devez voir des données associées au capteur "temperature".

Question 9.6.1: changement de pas

Que se passe-t-il si dans le programme `wifi_temperature.py` vous modifiez le pas de mesure ligne 36, pour le mettre par exemple à 60 secondes.

10. Sigfox

Sigfox est l'un des tous premiers réseaux entièrement dédié à l'Internet des Objets. Comme nous l'avons vu auparavant, il est de la famille des **LPWAN** qui privilégie la portée et la consommation d'énergie au débit. Les communications sont également fortement asymétrique, ce qui fait que les échanges ne sont pas comme sur un réseau Wi-Fi.

Les LoPy peuvent utiliser le réseau et bénéficie d'un an de connectivité gratuite sur le réseau Sigfox. Après les coûts d'abonnement sont relativement limités.

Youtube



10.1 Récupération des identifiants

Dans un premier temps, vous devez enregistrer votre capteur sur le site de Sigfox. Il vous faut deux éléments : son identifiant et son mot de passe appelé PAC (Porting Authorization Code). Ce dernier doit rester secret car il permet à toute personne qui le possède d'enregistrer un objet ou d'en changer le propriétaire.

Le programme `sigfox_id.py` permet d'afficher ces deux valeurs et d'envoyer un message sur le réseau Sigfox. Avant de l'exécuter, vérifiez que vous avez branché une antenne sur le connecteur de droite, celui opposé au bouton Reset sur le Pycom, cote LED.

Listing 10.1 – `sigfox_id.py`

```
1 from network import Sigfox
2 import binascii
3 import socket
4
5 # initialise Sigfox for RCZ1 (You may need a different RCZ Region)
6 # RCZ1: Europe, Oman, South Africa
7 # RCZ2: USA, Mexico, Brazil
8 # RCZ3: Japan
9 # RCZ4: Australia, New Zealand, Singapore, Taiwan, Hong Kong, Columbia, Argentina
```

```

11  sigfox = Sigfox(mode=Sigfox.SIGFOX, rcz=Sigfox.RCZ1)
12
13 # print Sigfox Device ID
14 print("Sigfox_ID:", binascii.hexlify(sigfox.id()))
15
16 # print Sigfox PAC number
17 print("PAC_Number:", binascii.hexlify(sigfox.pac()))
18
19 s = socket.socket(socket.AF_SIGFOX, socket.SOCK_RAW)
20 s.send("Hi!_Sigfox")

```

Ce programme importe l'objet Sigfox du module `network` (ligne 1) et crée une instance `sigfox` (ligne 10). Il est important de bien spécifier la bonne région d'utilisation car les bandes de fréquences peuvent différer d'un continent à un autre. En plus d'émettre dans l'illégalité, le réseau Sigfox ne recevra pas les messages.

Les lignes 13 et 16 affichent les identifiants Sigfox de votre LoPy. Notez-les, ils nous serviront pour enregistrer l'objet sur le réseau de Sigfox.

La ligne 18 crée une `socket`, à l'instar de ce qui avait été fait avec UDP au chapitre précédent. On peut donc utiliser les mêmes primitives en Sigfox qu'en UDP. La ligne 19 permet d'envoyer un message que vous pouvez personnaliser dans la limite de 12 caractères ; taille maximale des trames Sigfox.

10.2 Enregistrement de l'objet

Maintenant que vous avez les précieux identifiants Sigfox, connectez-vous avec un navigateur sur le site <https://backend.sigfox.com/activate>. Le processus est très simple. Il suffit de remplir les champs des différents formulaires :

- indiquez votre pays ;
 - remplissez le formulaire avec l'identifiant de l'objet et le **PAC** que vous avez obtenus avec le programme `sigfox_id.py` ;
 - indiquez à des fins de statistique ce qui vous amène ici ;
 - créez votre compte Sigfox ;
- ce qui va conduire à enregistrer l'objet et à vous créer un compte sur le backend de Sigfox.

10.3 Visualisation des données émises par le Pycom

Rendez-vous sur le **backend** de Sigfox (<https://backend.sigfox.com/>) et identifiez-vous avec le compte que vous venez de créer.

Les onglets en haut de la page (cf. figure 10.1 on the following page) vont vous permettre de naviguer dans différents types d'information. Dans cette partie, nous n'utiliserons que l'onglet **DEVICE**. Il donne accès aux objets enregistrés vous appartenant (cf. figure 10.2 on the next page). On y retrouve :

- un nom créé par Sigfox en fonction du type d'objet ;
- le propriétaire ;
- l'ID que vous avez donné lors de l'enregistrement ;
- la date du dernier message reçu par Sigfox pour cet objet.

Il faut cliquer :

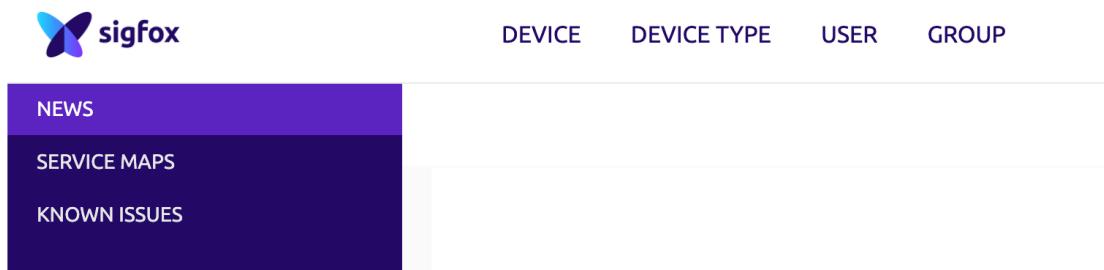


FIGURE 10.1 – Page d'accueil

Communication status	Device type	Group	Id	Last seen	Name	Token state
○	PYCOM_DevKit_1	IMT Atlantique	4D55AC	2020-08-12 16:22:07	PYCOM_DevKit_1-device	<input checked="" type="checkbox"/>

FIGURE 10.2 – Objets enregistrés

- sur le nom de l'objet pour le configurer ;
- sur le nom du groupe pour accéder à des paramètres d'administration des objets ;
- sur l'ID de l'objet pour obtenir des informations concernant les messages reçus puis MESSAGES dans le menu de gauche, pour avoir la liste des messages reçus par Sigfox, comme montré à la figure 10.3 on the facing page¹.

10.4 Que s'est-il passé côté radio

La figure 10.4 on the next page montre sur un **analyseur de spectre**, l'émission de 4 trames de données, dont une en cours, par un objet. Les petits traits verticaux correspondent à une émission. Ces traits sont très fin ; la bande passante utilisée est très petite, d'où le terme anglais de (Ultra Narrow-Band (UNB). Cela limite le risque de **collision** qui rendrait la donnée incompréhensible lié à l'émission simultanée d'un autre équipement sur la même fréquence.

En fait, le même message est émis 3 fois sur des fréquences différentes et aléatoire, augmentant ses chances d'être reçu.

10.5 Récupération des données

L'idéal serait d'avoir directement accès à ces données pour pouvoir les manipuler dans un programme. Pour ce faire, nous pouvons utiliser l'API REST développée par Sigfox. Dans l'onglet **DEVICE** (figure 10.2), il faut cliquer cette fois ci sur :

- le nom de votre groupe ;
- dans le menu de gauche, sur API ACCESS ;
- en haut à droite, sur le tout petit bouton *New*.



1. La séquence 48 69 21 20 53 69 67 66 6f 78 correspondant à la chaîne de caractères Hi! Sigfox.

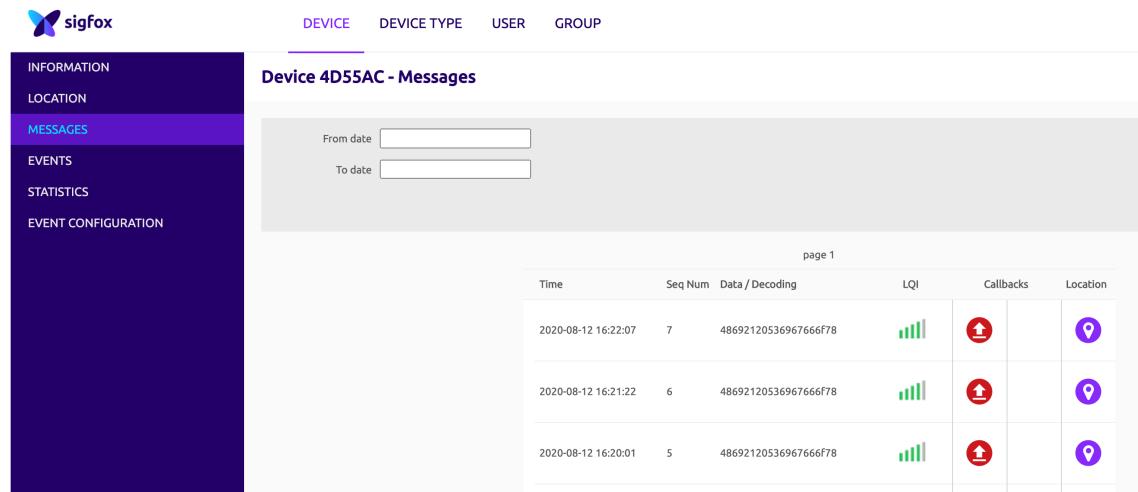


FIGURE 10.3 – Liste des messages reçus

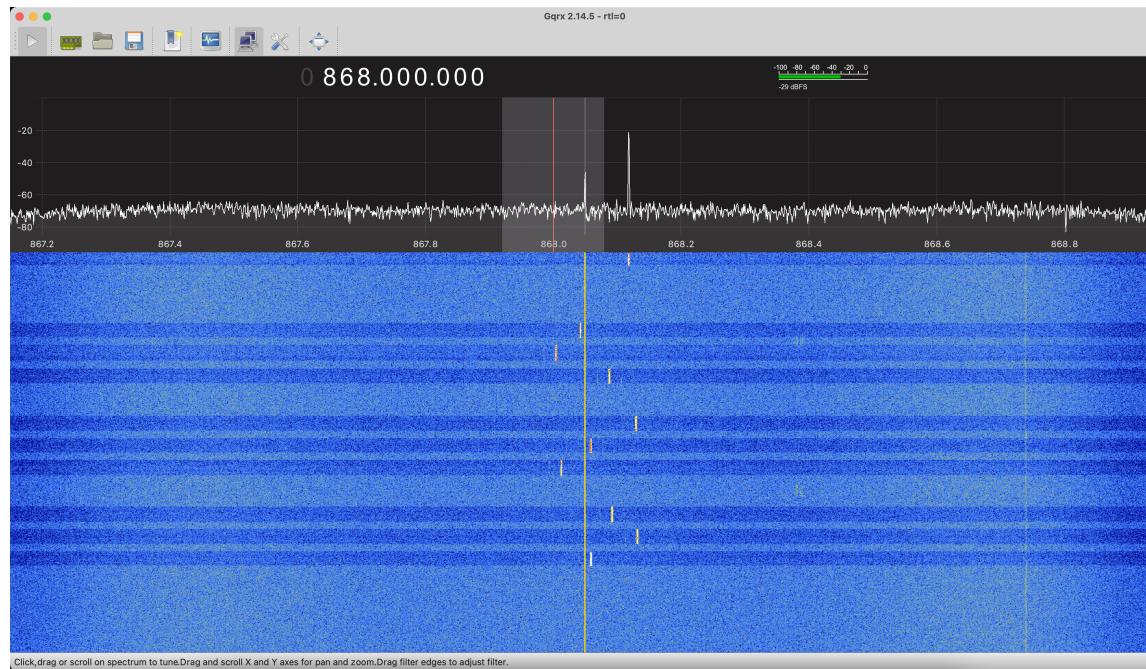


FIGURE 10.4 – Emissions radio liées à l'envoi de trame Sigfox.

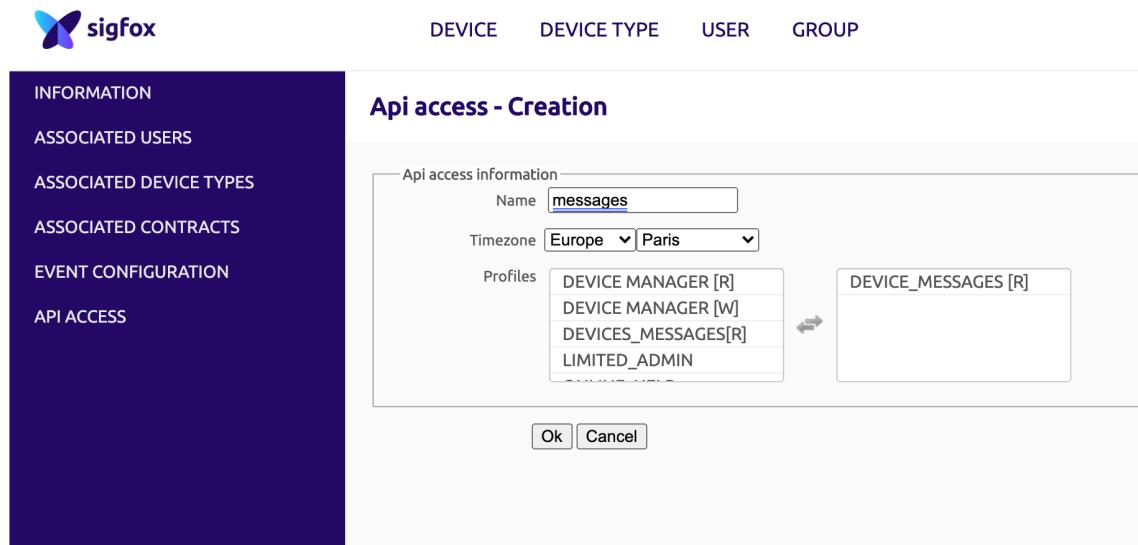


FIGURE 10.5 – Configuration de l’API REST.

Une page similaire à la figure 10.5 s’affiche. Donnez un nom à cet accès et choisissez, dans le menu *Profiles* le choix *DEVICE MESSAGE [R]* pour avoir le droit de lire les messages. Puis évidemment, sur Ok.

Vous voyez apparaître une nouvelle page avec deux champs en hexadécimal : *login* et *password*, que vous devez, comme pour l’API de Beebotte, noter quelque part ou apprendre par cœur pour la suite.

Le mieux étant de remplir un fichier de configuration avec ces valeurs comme le montre le programme config_sigfox.py.

Listing 10.2 – config_sigfox.py

```

1 API_USER = "603f575525643207b6322e9b"
2 API_PASSWORD = "93fa105063c2d0aee2bfdbadccfd2460"
3 DEVICEID = "1B28CF4"
```

Tous les éléments sont maintenant réunis pour émettre un relevé de température en utilisant le réseau Sigfox.

10.5.1 Sur le serveur

Nous avons les clés d'accès à l'API, il suffit maintenant d'écrire un petit script Python.

Pour des raisons de sécurité, nous vous invitons à prendre l'habitude de mettre les informations sensibles dans un fichier séparé.

Le programme device_messages.py permet de lister les messages reçus par Sigfox pour un objet particulier sur votre ordinateur.

Listing 10.3 – device_messages.py

```

1 import requests
2 import os
```

```

3 from requests.auth import HTTPBasicAuth
4 import pprint
5 import json
6 from config_sigfox import API_USER, API_PASSWORD, DEVICEID
7 import binascii

9 url = 'https://backend.sigfox.com/api/v2/devices/' + DEVICEID + '/messages'

11 print(url)

13 r = requests.get(url, auth=HTTPBasicAuth(API_USER, API_PASSWORD))
14 print(r.status_code)

15 if r.status_code != 200:
16     exit

19 resp = json.loads(r.text)
20 pprint.pprint(resp)
21 for v in resp["data"]:
22     print ("{:10d} {:2d} {:25} {:20} received {}".format(
23         v["time"], v["seqNumber"],
24         v["data"], "[" + str(binascii.unhexlify(v["data"])) + "]",
25         len(v["rinfos"])))
26

```

Le programme :

- Ligne 1, importe le module `requests` pour pouvoir envoyer des requêtes http à un serveur.
- Ligne 3, le module `HTTPBasicAuth` est utilisé pour s'identifier de façon simple en utilisant un login et un mot de passe.
- Ligne 6 ce login et ce mot de passe sont extrait du fichier rempli au chapitre précédent lors de la création de l'API REST.
- Ligne 9, l'URL comportant l'ID de l'objet est construite et
- ligne 13 la requête HTTP est envoyée avec la méthode d'authentification basée sur le mot de passe. la variable `r` est une structure contenant plusieurs informations.
- lignes 14 à 17, Si le code retourne est 200, tout s'est bien passé et `r.text` contient la réponse.
- ligne 19, cette réponse est une chaîne de caractères qui est déserialisée de la représentation JSON pour en faire une structure Python grâce à la fonction `loads`.
- ligne 20, la réponse est affichée et ensuite certains éléments sont donnés. On y retrouve tous les messages qui ont été émis par le LoPy.

```

>python3 device_messages.py
https://backend.sigfox.com/api/v2/devices/1B28CF4/messages
200
{'data': [{}{'country': 'FRA',
             'data': '48692120536967666f78',
             'device': {'id': '1B28CF4'},
             'lqi': 3,
             'nbFrames': 3,
             'operator': 'SIGFOX_France',
             'rinfos': [],
             'rolloverCounter': 0,
             'satInfos': [],
             'seqNumber': 13,
             'time': 1525811515,
             'txPower': -100}]}

```

```

        'time': 1640279367000},
        {'country': 'FRA',
         'data': '48692120536967666f78',
         ...
         'rolloverCounter': 0,
         'satInfos': [],
         'seqNumber': 11,
         'time': 1640279155000}],
        'paging': {}}
1640279367000: 13 48692120536967666f78 [b'Hi! Sigfox'] received 0
1640279338000: 12 48692120536967666f78 [b'Hi! Sigfox'] received 0
1640279155000: 11 48692120536967666f78 [b'Hi! Sigfox'] received 0

```

La fin de la trace affiche un résumé plus lisible de l'information reçue :

- **time** donne l'heure de réception codée suivant le format **Epoch**, évoqué lors de la communication avec Beebotte au chapitre précédent;
- **seqNumber** contient le numéro de la trame et est remis à 0 quand l'objet est reflashe. Il permet de détecter des pertes de données si les numéros ne sont pas contigus;
- "data" contient les données codées dans une chaîne hexadécimale. Le programme utilise la fonction `unhexify` pour la reconvertisir en séquence d'octets qui peuvent être affichés s'il s'agit de caractères ASCII;
- **rinfos** donne les informations sur les différentes passerelles radio de l'opérateur qui ont reçu le message.

10.5.2 Sur le LoPy

Le programme `sigfox_temperature.py` est une adaptation de `wifi_temperature.py`, listing 9.6 on page 112 dédié au Wi-Fi pour des transmission sur le réseau Sigfox.

Listing 10.4 – `sigfox_temperature.py`

```

1 import BME280
2 import time
3 import socket
4 import kpn_senml.cbor_encoder as cbor
5 from machine import I2C
6 from network import Sigfox
7 import binascii
8 import socket
9
10 # initialise Sigfox for RCZ1 (You may need a different RCZ Region)
11 sigfox = Sigfox(mode=Sigfox.SIGFOX, rcz=Sigfox.RCZ1)
12 s = socket.socket(socket.AF_SIGFOX, socket.SOCK_RAW)
13
14 FRAME_MAX = 12
15 t_history = []
16
17 i2c = I2C(0, I2C.MASTER, baudrate=400000)
18 print(i2c.scan())
19 bme = BME280.BME280(i2c=i2c)
20
21 while True:
22
23     t = int(bme.read_temperature()*100)

```

```

25     # No more room to store value, send it.
26     if len(t_history) == 0:
27         t_history = [t]
28     else:
29         t_history.append(t_prev)
30
31     print (t_history, len(cbor.dumps(t_history)))
32
33     if len(cbor.dumps(t_history)) > FRAME_MAX:
34         # oops too big for Sigfox
35         t_history = t_history[:-1] # remove last item
36         s.send (cbor.dumps(t_history))
37         t_history = [t]
38
39     prev = t
40
41     time.sleep(10)

```

- ligne 5, la classe Sigfox du module network est installée ;
- ligne 11, un objet texttt est instancié avec, ici, les paramètres pour l’Europe ;
- ligne 12, au lieu de faire appel à AF_INET pour utiliser la pile protocolaire TCP/IP, la valeur AF_SIGFOX est utilisée.
- ligne 14, la taille de la trame est fixée à 12 octets pour être compatible avec le réseau.

Le programme s’exécute sur le LoPy.

```

>>> Running sigfox_temperature.py
>>>
>>>
[118]
[2192] 4
[2192, -89] 6
[2192, -89, -16] 7
[2192, -89, -16, -12] 8
[2192, -89, -16, -12, -15] 9
[2192, -89, -16, -12, -15, -23] 10
[2192, -89, -16, -12, -15, -23, -11] 11
[2192, -89, -16, -12, -15, -23, -11, -14] 12
[2192, -89, -16, -12, -15, -23, -11, -14, -13] 13
[1999, -12] 5
[1999, -12, -5] 6
[1999, -12, -5, -8] 7
[1999, -12, -5, -8, -9] 8
[1999, -12, -5, -8, -9, -6] 9
[1999, -12, -5, -8, -9, -6, 2] 10
[1999, -12, -5, -8, -9, -6, 2, 0] 11
[1999, -12, -5, -8, -9, -6, 2, 0, -5] 12
[1999, -12, -5, -8, -9, -6, 2, 0, -5, -2] 13
[1954, -4] 5
[1954, -4, -1] 6

```

Le programme device_messages.py récupère également ces valeurs depuis l’ordinateur.

1640281923000: 15 891907cf2b24272825020024	[b"\x89\x19\x07\xcf+\$'(\%\x02\x00\$"] received 0
1640281823000: 14 8819089038582f2b2e362a2d	[b'\x88\x19\x08\x908\x/+.*-'] received 0
1640279367000: 13 48692120536967666f78	[b'Hi! Sigfox'] received 0
1640279338000: 12 48692120536967666f78	[b'Hi! Sigfox'] received 0
1640279155000: 11 48692120536967666f78	[b'Hi! Sigfox'] received 0

Attention suivant les variations de la température, le message CBOR grandit plus au moins vite. Dans le cas précédent, il y avait une émission toutes les 90 secondes, or l’abonnement au réseau Sigfox limite le nombre d’émission à 140 messages par jours. Au bout de 3 heures le quota de message sera épousé. Cette petite période d’émission permet de tester plus rapidement les solutions, mais il convient d’augmenter la période pour une utilisation régulière.

10.5.3 requête GET depuis le serveur

Côté ordinateur, la stratégie la plus simple à mettre en œuvre consiste à étendre le programme `device_message.py` vu précédemment et d'interroger périodiquement le *backend* de Sigfox pour voir si de nouvelles données sont arrivées. Cela donne le programme `display_sigfox.py` suivant :

Listing 10.5 – `display_sigfox.py`

```

1 import requests
2 import os
3 from requests.auth import HTTPBasicAuth
4 import pprint
5 import json
6 from config_sigfox import API_USER, API_PASSWORD, DEVICEID
7 import binascii
8 import time
9 import datetime
10 import cbor2 as cbor
11 import beebotte
12 import config_bbt
13
14 bbt = beebotte.BBT(config_bbt.API_KEY, config_bbt.SECRET_KEY)

```

Les importations incluent les modules pour envoyer les données à Beebotte et pour interroger Sigfox. Ligne 14, la communication avec les serveurs de Beebotte est établie en utilisant les secrets du module `config_btt`.

```

url = 'https://backend.sigfox.com/api/v2/devices/' + DEVICEID + '/messages'
last_epoch = 0

# get the last message to find the starting epoch
parameters = {'limit': 1}
r = requests.get(url, auth=HTTPBasicAuth(API_USER, API_PASSWORD),
                  params=parameters)

print(r.status_code)
if r.status_code != 200:
    exit

j = json.loads(r.text)
last_epoch = j["data"][0]["time"]

```

la fonction `to_bbt` n'a pas été modifiée, on passe donc à la récupération des données sur les serveurs de Sigfox. Le programme va interroger régulièrement le serveur de Sigfox pour récupérer les nouveaux messages. Comme Sigfox stocke l'ensemble des messages reçus, cela peut conduire à un trafic conséquent. Pour limiter le trafic, le programme n'affichera que les nouvelles données qui arrivent pendant son exécution.

Le programme :

- ligne 41, construit l'URL pour la requête en incluant l'identificateur du LoPy ;
- ligne 42, initialise la variable `last_epoch`. Elle contient l'epoch du dernier message reçu par Sigfox pour cet objet ;
- ligne 45, une structure JSON est créée, elle contient la clé `limit` et la valeur 1, pour indiquer à Sigfox que l'on souhaite recevoir en réponse que le dernier message reçu ;
- lignes 46 à 51, la requête est envoyée avec le paramètre précédemment défini. Si le status n'est pas 200, le programme s'arrête.

— lignes 53 et 54, la variable `last_epoch` reçoit l'instant d'arrivée du dernière message.

```
56 # delay next request to avoid 429 status error  
time.sleep(10)
```

Un délai de 10 secondes est introduit entre deux requêtes, car Sigfox limite le nombres de requête pour éviter la saturation de ses serveurs.

```
# look periodically if new data arrived.
60 while True:
61     if last_epoch > 0:
62         parameters = {"since": last_epoch+1}
63     else:
64         parameters = None
65
66     print (parameters)
67
68     r = requests.get(url, auth=HTTPBasicAuth(API_USER, API_PASSWORD),
69                      params=parameters)
70
71     print (r.status_code)
72     if r.status_code != 200:
73         break
```

Le programme entre dans une boucle sans fin où il va interroger régulièrement le serveur Sigfox. Les interrogations sont identiques à la précédente, mais le paramètre contient un objet JSON avec la clé `since` et l'`epoch`. Si la réponse à la requête n'est pas 200: OK, le programme s'arrête.

```
76     resp = json.loads(r.text)
77
78     for v in resp["data"]:
79         if last_epoch < v["time"]:
80             last_epoch = v["time"]
81
82         print ("{:10d}: {:2d} {:25} {:20} received {}".format(
83             v["time"], v["seqNumber"],
84             v["data"], "["+str(binascii.unhexlify(v["data"]))+"]",
85             len(v["rinfos"]))
86
87
88         measure = cbor.loads(binascii.unhexlify(v["data"]))
89         sending_time = v["time"]
90         print (sending_time, measure)
91         to_bbt("capteurs", "temperature", measure, factor=0.01,
92                period=10, epoch=sending_time)
```

La structure JSON reçue est déserialisée ligne 75 pour en faire un tableau de messages enrichis de paramètres ajoutés par Sigfox. Pour chacun de ces éléments qui correspondent aux nouveaux messages reçus, Le programme va faire avancer la variable last_epoch sur la plus grande valeur (lignes 78 et 79), puis ligne 87 va prendre les données indiquées par la clé data dans le dictionnaire. Ces données, correspondent au tableau CBOR envoyé par l'objet, codés en une chaîne de caractère hexadécimaux. La fonction unhexlify la convertie en séquence binaire dans une chaîne d'octet puis loads transforme le tableau CBOR en un tableau Python.

Ligne 91 et 92, la fonction `to_btt` (ligne 16 à 39 non représenté dans ce listing) est appelée. Elle est identique à celle qui avait été présentée au chapitre précédent. Mais comme la procédure est

asynchrone, le programme traite les données quand il en fait la demande, pas quand le capteur émet des données, le timestamp doit être celui indiqué par Sigfox, il est passé dans le paramètre epoch.

```
time.sleep(60)
```

Le programme attend 60 seconde avant de faire une nouvelle requête.

10.5.4 requête POST vers le serveur

Nous avons dû modifier la logique du programme `display_sigfox.py`. Au lieu d'attendre des données comme le faisait `display_server.py` en Wi-Fi, il va chercher activement les données sur le *backend*. Cela est imposé par les limitations de l'adressage IPv4 privé que vous avez certainement sur votre réseau local. En effet, il n'est pas possible d'être joint par l'extérieur, les connexions ne se font qu'à l'initiative de l'équipement qui a une adresse privée. Pour revenir au cas précédent où le *backend* pourrait pousser des données, on peut soit faire tourner le programme sur une machine virtuelle dans le Cloud (ex : Virtual Private Server (VPS) d'OVH) ou configurer le Network Address Translation (NAT) de votre boîtier Internet pour autoriser des connexions entrantes sur un port particulier. C'est cette dernière option que nous allons détailler. Bien entendu, les caractéristiques des NAT changent d'un opérateur à un autre ; nous ne pourrons pas détailler sa configuration.



Configuration du *callback*

Dans un premier temps, rendez-vous sur <https://backend.sigfox.com/device/list>, cliquez sur le nom de l'objet puis, dans le menu de gauche, sur *Callback*, puis *New*. Une page apparaît avec la possibilité de connecter l'objet à différentes plateformes. Nous choisissons *Custom callback* car nous voulons garder notre indépendance. La figure 10.6 on the next page montre le formulaire.

Ensuite, il faut déterminer l'adresse IP publique derrière votre NAT². L'URI de votre serveur sera quelque chose comme indiquée ci-dessous pour l'*Url Pattern*, où *aaa.bbb.ccc.ddd* représente l'adresse IP publique³.

```
http://aaa.bbb.ccc.ddd:9999/sigfox
```

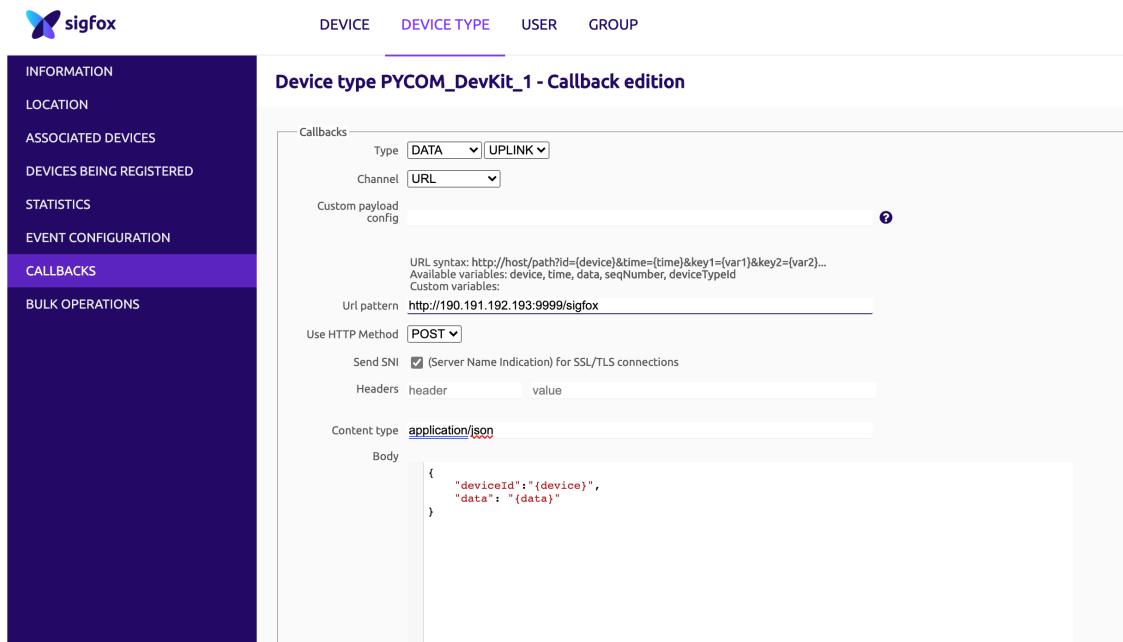
Changez la méthode de **GET** par un **POST**, c'est quand même plus propre. Le GET étant un moyen de récupérer de l'information pas d'en transmettre si on veut rester compatible avec REST.

Ce n'est pas fini. Il faut maintenant formater le contenu. Dans Content type, remplacez la valeur par *application/json* car c'est ce que l'on sait faire de mieux. Dans la fenêtre en dessous, nous allons définir notre format JSON avec deux clés : *deviceId* et *data*, suivies de deux variables entre accolades que Sigfox remplacera par les vraies valeurs.

Finalement, cliquez sur Ok. Maintenant, quand Sigfox recevra des données de l'objet, il enverra une requête POST à l'URI indiquée.

2. Un site comme <https://wtfismyip.com/> peut aider.

3. Attention, certains opérateurs changent régulièrement l'adresse IP allouée au NAT. Il est donc préférable d'utiliser un DNS dynamique si vous voulez l'utiliser à long terme.

FIGURE 10.6 – Configuration d'un *callback*.

Configuration du NAT

Il vous reste à configurer le NAT de votre routeur d'accès pour que les paquets à destination du port 9999 (valeur choisie dans l'URI) TCP soient envoyés à l'adresse privée de votre ordinateur. La démarche est généralement la même : configurer DHCP pour que votre ordinateur ait toujours la même adresse dans la maison puis configurer le NAT pour que les paquets adressés à un port soit envoyé à cet ordinateur.⁴

Wireshark peut vérifier que les requêtes traversent bien le NAT et arrive à votre ordinateur en regardant le trafic TCP sur le port 9999.

Traitement des requêtes POST

Les données contenues dans la requête POST venant du *backend* de Sigfox seront transférées sur le lien *loopback* de l'ordinateur vers le port UDP 33033. Le programme `display_server.py` attendant les données sur ce port pourra être utilisé pour transformer la série temporelle codée en CBOR en structure JSON attendue par Beebotte. De cette manière, nous banalisons le programme qui pourra traiter ces trois sources d'information :

- objet émulé par un programme Python,
- LoPy avec Wi-Fi,
- LoPy avec Sigfox,
- et bientôt LoPy avec LoRaWAN.

Le programme `generic_relay.py` le fait pour Sigfox et différents serveurs LoRaWAN. Nous ne dévoilerons ici que les lignes liées au traitement de Sigfox.

4. Si vous ne pouvez pas le faire, vous pouvez casser votre tirelire pour vous acheter un VPS avec une adresse IP publique pour quelques mois. Dans ce cas, il faudra installer Python3 et les modules nécessaires (Beebotte, cbor2...)

Listing 10.6 – generic_relay.py

```

54 def get_from_sigfox():
55
56     fromGW = request.get_json(force=True)
57     print ("SIGFOX_POST_RECEIVED")
58     pprint.pprint(fromGW)
59
60     downlink = None
61     if "data" in fromGW:
62         payload = binascii.unhexlify(fromGW["data"])
63         downlink = forward_data(payload)
64
65     resp = Response(status=200)
66     print (resp)
67     return resp

```

On se rappelle du serveur **Flask** que l'on avait vu au chapitre 3.4 on page 46, dans ce programme on retrouve les mêmes principes :

- ligne 54, le decorateur permet de lier la fonction `get_from_sigfox` au chemin d'URI `/sigfox` pour la methode POST.
- ligne 57 déserialise le contenu du POST qui est contenu dans la variable `request`. Le paramètre `force` est là au cas où vous auriez oublié de mettre sur le *backend* le format du contenu à `application/json`. Tout ce qu'il reçoit est considéré comme du JSON.
- lignes 62 à 64, si le POST contient des données (clé `data` dans l'objet JSON, alors elles sont envoyées sur le lien *loopback* via la fonction `forward_data`
- lignes 66 à 68, le POST est acquitté avec le status 200 pour indiquer que le traitement de la requête s'est bien déroulé.

A noter que la fonction `forward_data` peut retourner une réponse qui sera envoyée au client HTTP. Pour Sigfox, cette possibilité n'est pas prise en compte car Sigfox n'autorise que 4 messages descendant par jour. Nous détaillerons son fonctionnement lorsque nous étudierons les réseaux LoRaWAN⁵.

```

192 parser = argparse.ArgumentParser()
193 parser.add_argument("-v", "--verbose",
194                     action="store_true",
195                     help="show uplink and downlink messages")
196 parser.add_argument('--http_port', default=9999,
197                     help="set http port for POST requests")
198 parser.add_argument('--forward_port', default=33033,
199                     help="port to forward packets")
200 parser.add_argument('--forward_address', default='127.0.0.1',
201                     help="IP address to forward packets")
202
203 args = parser.parse_args()
204 verbose = args.verbose
205 defPort = args.http_port
206 forward_port = args.forward_port
207 forward_address = args.forward_address
208
209 app.run(host="0.0.0.0", port=defPort)

```

5. Pour plus de détails sur l'envoi de messages descendant voir page 141.

La partie principale du programme analyse les arguments utilisés lors de son appel. Si l'option `-v` est utilisé, le programme affichera les messages relayés. D'autres options sont définies pour modifier les numéro de port. Ces dernières sont utiles si ces programmes sont lancés sur un même machine dans le cloud.

Exemple

L'exemple suivant trace le parcours d'une série temporelle depuis un LoPy jusqu'à son envoi au serveur Beebotte pour visualisation.

```
>>> Running sigfox_temperature.py

>>>
>>>
[118]
[1895] 4
[1895, -4] 5
[1895, -4, -2] 6
[1895, -4, -2, 1] 7
[1895, -4, -2, 1, -3] 8
[1895, -4, -2, 1, -3, -1] 9
[1895, -4, -2, 1, -3, -1, 0] 10
[1895, -4, -2, 1, -3, -1, 0, 0] 11
[1895, -4, -2, 1, -3, -1, 0, 0, 6] 12
[1895, -4, -2, 1, -3, -1, 0, 0, 6, 2] 13
```

Le LoPy collecte les mesures et construit sa série temporelle qui est envoyée quand la capacité de la trame est atteinte. Dans le cas de Sigfox, il s'agit de 12 octets, la dernière ligne montre que la capacité est dépassée, donc le dernier élément est retiré.

```
>python3.5 generic_relay.py -v
SIGFOX POST RECEIVED
{'data': '891907672321012220000006', 'deviceId': '4D3D0E'}
--UP-> b'891907672321012220000006',
no DW
<Response 0 bytes [200 OK]>
185.110.98.2 - - [24/Dec/2021 10:14:26] "POST /sigfox HTTP/1.1" 200 -
```

Le programme `generic_relay.py` reçoit la requête POST du réseau Sigfox sur le chemin d'URI `/Sigfox`. Elle contient l'élément CBOR envoyé par le LoPy. Ces données sont envoyées sur le port 33033. Aucune réponse n'est reçue, la requête POST est simplement acquittée.

```
>python3 display_server.py
[{'data': 18.95, 'resource': 'temperature', 'ts': 1640337186000.0},
 {'data': 18.91, 'resource': 'temperature', 'ts': 1640337196000.0},
 {'data': 18.89, 'resource': 'temperature', 'ts': 1640337206000.0},
 {'data': 18.90, 'resource': 'temperature', 'ts': 1640337216000.0},
 {'data': 18.87, 'resource': 'temperature', 'ts': 1640337226000.0},
 {'data': 18.86, 'resource': 'temperature', 'ts': 1640337236000.0},
 {'data': 18.86, 'resource': 'temperature', 'ts': 1640337246000.0},
 {'data': 18.86, 'resource': 'temperature', 'ts': 1640337256000.0},
 {'data': 18.92, 'resource': 'temperature', 'ts': 1640337266000.0}]
```

Le programme `display_server.py` traite la donnée CBOR reçue sur le port 33033 et la convertie dans le format JSON attendu par Beebotte.

10.6 Conclusion

On a construit un prototype de capteur qui fonctionne sur Sigfox. À vous de l'améliorer. En particulier, il faut augmenter l'intervalle entre deux mesures qui a été fixe à 10 secondes pour ne pas avoir à faire des tests trop longs. Comme la taille d'un message est de 12 octets, CBOR va prendre 1 octet pour coder le tableau et la valeur de référence est codée sur 3 octets. Si les deltas sont petits, ils tiendront sur un octet. Il reste de la place pour 8 deltas. Donc, la période d'émission est de 90 secondes si les intervalles entre deux mesures restent à 10 secondes.

Comme Sigfox n'autorise que 140 messages par jours, la durée de mesure ne sera que de 210 minutes par jours, soit 3 heures et demie. Il est donc préférable de prendre des intervalles plus grands. Vous pouvez calibrer votre LoPy et votre programme de réception pour pouvoir suivre la température sur une journée. N'oubliez pas de changer également cette période dans le programme `display_server.py`.

Le programme `display_server.py` permet de récolter des informations de plusieurs sources :

- un programme en local qui émule les mesures,
- du réseau Wi-Fi,
- par l'intermédiaire de `generic_relay.py` provenant d'un réseau LPWAN.

Nous avons notre convention de représentation de l'information basée sur CBOR pour définir une série temporelle. En revanche, nous avions été obligés de modifier le programme `display_server.py` quand nous étions passé d'une série temporelle représentant l'évolution de l'humidité à une autre traitant de la température et si la période change.

Nous avons vu la différence entre les requêtes GET et POST qui induisent des comportements différents. Si GET est plus universel et peut fonctionner derrière un NAT, il demande des interrogations régulières pour savoir si la ressource a changé ou non. POST impose que le serveur soit accessible et ainsi l'expose, mais les résultats sont reçus instantanément.

La notion de client et de serveur est floue, il ne peut pas être attribuée à un programme ou à un équipement, par exemple, le programme `display_server.py` est serveur pour Sigfox et client pour Beebotte. Le site de Sigfox est serveur, si l'on utilise une méthode GET pour obtenir les données et client s'il fait un POST.

Pour revenir au paradigme de REST, nous avons, avec CBOR, défini le contenu de la ressource mais nous ne l'avons pas nommée. Le récepteur doit connaître le format (du CBOR contenant une série temporelle codée en différentiel) et ce qu'elle désigne. C'est ce que nous allons voir dans la prochaine session avec le protocole **CoAP** où nous pourrons définir, comme en HTTP, le nom de la ressource et son contenu.

11. LoRaWAN

Avec **Sigfox**, ce n'était que du bonheur ! Nous avions un environnement unique développé par un seul acteur. L'attachement d'un objet au réseau, la récupération des données s'en trouve grandement simplifiée.

Avec **LoRaWAN**, l'écosystème est plus complexe. Pour rappel, **LoRa** est une modulation longue portée et LoRaWAN un protocole de niveau 2 qui a pour but de fédérer les acteurs (fabricants d'objets, utilisateurs d'objets, opérateur de réseaux) autour de la communication radio entre l'objet et le cœur de réseau appelé ici LNS . Mais l'enregistrement des objets, le lien entre les applications et le LNS diffèrent d'un réseau à un autre.



Même si vous n'avez pas l'intention de mettre en œuvre le réseau Sigfox, nous vous recommandons de lire le chapitre précédent (ou à la rigueur de le survoler) car nous y ferons référence dans la suite du texte.

LoRaWAN, comme Sigfox, opère dans la même bande de fréquences non licenciées. Il existe plusieurs opérateurs de réseaux LoRaWAN :

- The Things Network (TTN)¹ propose une approche communautaire. Chaque personne peut mettre à disposition des passerelles radio pour la communauté et inscrire des objets. TTN gère le cœur de réseau. Si cette approche est sympathique, la couverture va être très parcellaire et va dépendre de la densité de geeks dans une région. De plus, la couverture du réseau va dépendre de la position des antennes qui doivent être placées sur un point haut. Avoir une antenne sur son balcon n'implique pas une grande couverture de la zone. Si vous avez de la chance, vous pourrez peut-être bénéficier d'un accès via TTN. Nous verrons comment s'y connecter.
- des opérateurs nationaux comme, en France, **Orange** ou **Bouygues Télécom**. Mais il faut généralement bourse délier pour connecter les objets et pour limiter l'usage du downlink ; l'envoi des messages vers les capteurs est facturé en supplément. En revanche, la couverture

1. <https://www.thethingsnetwork.org/>

- est supérieure.
- les réseaux privés. Il est possible de faire tourner son propre LNS. Cela implique d'acheter des composants pour faire une passerelle radio. Mais il existe des mises en oeuvre ouverte du LNS comme **chirpstack**².

11.1 Information sur le LoPy

Chaque objet va avoir un identifiant unique codé sur 64 bits appelé ***devEUI***. Cette information est nécessaire pour faire entrer un équipement sur le réseau. Le programme `lorawan_devEUI.py` donne accès à cette valeur qui est unique pour chaque LoPy.

Listing 11.1 – `lorawan_devEUI.py`

```
from network import LoRa
2import pycom
import binascii
#
lora = LoRa(mode=LoRa.LORAWAN)
mac = lora.mac()
#
print ('devEUI:', binascii.hexlify(mac))
#
```

L'objet Python LoRa est importé du module `network` et une instance est créée ligne 5. La variable `mac` va stocker l'adresse MAC (autre nom du *devEUI*) retournée par l'objet `lora` et l'afficher en hexadécimal, comme le montre l'exemple suivant :

```
>>> Running lorawan_devEUI.py
>>>
>>>
devEUI: b'70b3d54994c61237'
```

On a le *devEUI*. Il manque encore 3 informations :

- ***appEUI*** : c'est un identifiant sur 64 bits comme le *devEUI* qui va servir à identifier l'application.
On peut le mettre à 0 ;
- ***AppKey*** : c'est une valeur sur 128 bits qui n'est connue que de l'objet et du LNS. Elle permet de dériver les clés de chiffrement utilisées par la suite ;

Et sur le LNS, il faudra aussi configurer le connecteur : il s'agit d'indiquer au LNS comment envoyer les données vers une application. Il peut s'agir de POST HTTP comme nous avons vu avec la configuration du ***backend*** avec Sigfox. Bien entendu, une fois ces grands principes établis, ça serait trop simple si tous les réseaux fonctionnaient de la même façon. Nous allons donc voir comment faire avec le réseau TTN.

11.2 The Things Network

2. <https://www.chirpstack.io/>

The Things Network, ou TTN pour les intimes, est un réseau communautaire qui permet à tout un chacun de connecter soit une passerelle radio pour le bénéfice de la communauté, soit un objet. TTN se charge de faire tourner un LNS et de renvoyer les informations collectées à son propriétaire. On espère qu'un bon samaritain a déployé une antenne près de chez vous pour capter votre trafic. On n'a aucun moyen de savoir, si on n'essaye pas.



La première étape consiste à se créer un compte, gratuit comme il se doit, sur le site de TTN : <https://www.thethingsnetwork.org/>.

Une fois le formulaire de création de compte rempli et validé, vous disposez d'un accès au réseau TTN. Votre identifiant apparaît en haut à droite de la page Web.

- Cliquez sur votre nom,
- sélectionnez Console,
- puis votre zone géographique (cela n'a rien à voir avec le plan de fréquence. Il est préférable de choisir le serveur le plus proche pour optimiser les communications).

On peut soit définir une application et y associer des objets, soit connecter une passerelle radio (*Go To Gateways*). Par la suite, nous explorerons cette option si vous voulez installer votre propre passerelle radio. Nous allons nous intéresser à la création d'application en choisissant *Go To Applications*. TTN vous invite à créer notre première application ; cliquez sur le lien *+Add Application*.

Définir une application

Une application pour TTN va intégrer plusieurs objets LoRaWAN qui enverront leur information à une application tournant sur un Application Server (AS). Il faut remplir le formulaire (cf. figure 11.1 on the next page) en indiquant :

- l'identifiant de l'application. Il s'agit d'un nom uniquement composé de lettres et de chiffres ainsi que du tiret. Il doit être unique pour TTN. Donc, faites preuve d'imagination car il doit être unique pour TTN. Ce nom se retrouvera dans les URI permettant de piloter l'application ;
- le nom de l'application qui apparaîtra dans les menus. Il n'y a pas de contraintes ; il vaut mieux choisir quelque chose d'explicite ;
- une description de l'application si nécessaire.

Cliquez sur *Create Application* pour enregistrer cette application. Vous arrivez sur une page de contrôle de votre application.

Ajout d'un objet

Nous devons enregistrer un ou plusieurs objets dans notre application :

- Cliquez sur *end devices* dans le menu de gauche ;
- puis *+ Add end device* ;
- et *try manual registration* en dessous du menu déroulant. TTN offre des aides à la configuration pour certains objets, mais comme nous sommes des experts, nous n'en n'avons pas besoin.

Le menu, décrit figure 11.2 on page 133 apparaît :

- Entrez le plan fréquence compatible avec votre région. En Europe, vous pouvez choisir d'avoir

General settings

Application ID *

Name

Description

Attributes

+ Add attributes

Attributes can be used to set arbitrary information about the entity, to be used by scripts, or simply for your own organization

Skip payload encryption and decryption

Enabled

Skip decryption of uplink payloads and encryption of downlink payloads

FIGURE 11.1 – Configuration d'une application.

la voie descendante en SF9³ ou en SF12. Le premier est un bon compromis entre la portée et la durée d'émission ; le second est à choisir si votre objet est difficilement accessible (profondément enfoui ou loin d'une passerelle radio) ;

- la version du protocole LoRaWAN compatible avec les LoPy :MAC V1.0.2 ;
- les paramètres de la couche physique du LoPy : PHY V1.0.2 REV A ;
- Vous devez récupérer le *devEUI* de votre LoPY en lançant sur Atom le programme *lorawan_devEUI.py* (cf. Listing 11.1 on page 130) ;
- mettez l'*AppEUI* à 0,
- et générez une clé de chiffrement *AppKey* et n'oubliez surtout pas de copier sa valeur quelque part, on en aura besoin pour configurer notre objet.

L'interface de TTN vous propose un identifiant pour l'objet basé sur son *devEUI*. Vous pouvez le garder, à moins de vouloir quelque chose de plus explicite.

Finalement, cliquez sur *register end device* pour sauvegarder votre oeuvre.

L'interface affiche une page récapitulative. Si vous avez oublié de le faire à l'étape précédente, copiez l'*AppKey* pour la mettre dans le programme *lorawan_send_and_receive.py*.

Connexion de l'objet

Listing 11.2 – *lorawan_send_and_receive.py*

```
from network import LoRa
import socket
import time
import pycom
import binascii
```

3. Le Spreading Factor définit la vitesse de transmission de l'information, elle diminue de moitié à chaque incrémentation donc le SF12 est 16 fois plus lent que le SF9, mais beaucoup plus robuste.

Register end device

FIGURE 11.2 – Ajout d'un objet.

```

6 lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)
8 #
9 mac = lora.mac()
10 print('devEUI:', binascii.hexlify(mac))

```

Le programme `lorawan_send_and_receive.py` commence, comme `lorawan_devEUI.py`, par créer un objet `lora` et afficher le *devEUI* de l'objet, c'est toujours pratique pour le débogage.

```

12 # create an OTAA authentication parameters
13 app_eui = binascii.unhexlify(
14     '00\u00\u00\u00\u00\u00\u00\u00'.replace('u', ''))
15
16 app_key = binascii.unhexlify(
17     '4EAE56D0689F6F8B02C2AFA7E08DADBA'.replace('u', ''))

```

On met dans deux variables, les identifiants présents sur le LNS de TTN, à savoir le *app_eui* que l'on avait mis à zéro et de *app_key* que l'on généré aléatoirement sur le site de TTN et que l'on vous avait demandé de copier quelque part.

Pour éviter de manipuler des séquences binaires, ces valeurs sont vues comme des chaînes de caractères (des espaces peuvent être insérés pour plus de lisibilité, la fonction `replace` permet de les éliminer). La séquence binaire est obtenue grâce à la fonction `unhexlify`.

```

20 pycom.heartbeat(False)
21 pycom.rgbled(0x101010) # white

```

On arrête le clignotement périodique bleu de la LED du LoPy en appelant la fonction `pycom.heartbeat` avec l'argument `False`. Puis on allume la LED en blanc avec la fonction `rgbled`. L'argument donne l'intensité des trois composantes Rouge, Vert et Bleu.

La LED va nous permettre de suivre pas à pas l'état du LoPy.

```

22 # join a network using OTAA (Over the Air Activation)
23 lora.join(activation=LoRa.OTAA, auth=(app_eui, app_key), timeout=0)
24
25 # wait until the module has joined the network
26 while not lora.has_joined():
27     time.sleep(2.5)
28     print('Not yet joined... ')
29
30 pycom.rgbled(0x000000) # black

```

Le LoPy effectue la connexion au réseau, cette procédure est appelée ***Join***. Elle correspond à l'émission d'un message vers le LNS. Si celui-ci reconnaît et accepte l'objet, il renverra quelques secondes plus tard un message *Accept*. Cette phase permet à l'Objet d'obtenir les clés de chiffrement des messages et une adresse plus courte appelé *devAddr*.

Pour se faire, le programme appelle la fonction `join` avec les paramètres suivants :

- le type de connexion, très souvent Over The Air Authentication (OTAA) pour récupérer dynamiquement les paramètres⁴
- les paramètres nécessaires, à savoir l'*AppEUI* et l'*AppKey* que nous avons précédemment configurés.

4. Il existe aussi une méthode appelé Authentication By Personalisation (ABP) qui consiste à configurer l'objet avec tous ses paramètres (comme pour Sigfox), mais elle est moins souple et moins adaptée à l'environnement LoRaWAN où plusieurs opérateurs coexistent sur les mêmes fréquences.

- un temporisateur (*timeout*). La valeur à 0 indique que l’objet va essayer de joindre le réseau jusqu’à ce que celui-ci l’accepte.

La fonction *join* n’étant pas bloquante, la procédure de *join* va se dérouler en tâche de fond. La fonction *has_joined* permet de suivre l’état du LoPy. D'où la boucle (lignes 26 à 28) dont on ne sortira que lorsque le LoPy sera accepter sur le réseau.

Toutes de 2.5 seconde un message sera affiché pour dire qu'il est toujours en attente d'une réponse. Ces messages ne sont absolument pas synchronisés avec l'envoi des trames Join qui ont lieu toutes les 15 secondes.

L’extinction de la LED (ligne 30) indique que le LoPy a joint le réseau LoRaWAN.

```
32 s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
33 s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)
34 s.setsockopt(socket.SOL_LORA, socket.SO_CONFIRMED, False)
```

Le programme ouvre la socket, le premier paramètre est AF_LORA pour indiquer que l'on utilise le protocole LoRaWAN. Les deux lignes suivantes, avec les fonctions *setsockopt*, permettent de configurer des paramètres LoRaWAN, comme :

- le Data Rate (DR). Sous ce terme, plusieurs paramètres des la couche physique sont réunis. Plus le DR est élevé, plus la transmission est rapide. En contre partie, elle est moins robuste et peut porter moins loin. Le DR varie entre 0 et 6.
- la capacité de LoRaWAN a acquitter les trames. Il n'est pas recommandé de l'activer car cela induit des émissions qui sont prises en compte dans le calcul du **Duty Cycle** (voir chapitre 5.2 on page 59).

```
36 while True:
37     pycom.rgbled(0x100000) # red
38     s.setblocking(True)
39     s.settimeout(10)
40
41     try:
42         s.send('Hello\\LoRa')
43     except:
44         print ('timeout\\in\\sending')
45
46     pycom.rgbled(0x000010) # blue
```

On entre dans une boucle sans fin pour envoyer et recevoir périodiquement des messages :

- ligne 37, la LED éclaire en rouge pour indiquer une transmission.
- ligne 38, les appels aux sockets sont rendu bloquant, c'est-à-dire qu'il ne finiront que quand l'action sera effectuée par la couche inférieure.
- ligne 39, par contre au bout de 10 secondes d'attente une erreur sera déclenchée.
- ligne 41, on récupère cette erreur grâce aux instructions *try* et *except* de Python. Si l'appel à *send* reste bloqué plus des 10 secondes, le traitement de l'exception affiche *timeout in sending*. Cet événement est rare et est causé par une défaillance locale à la couche 2⁵.
- ligne 46 dans tous les cas, la LED passe en bleu.

```
50     data = s.recv(64)
```

5. Il faut rester vigilant sur cette erreur, par exemple si l'on essaie d'envoyer un entier sans sérialisation, cela conduira à une erreur qui sera prise en compte de cette manière.

```

52     print(data)
53     pycom.rgbled(0x001000) # green
54 except:
55     print('timeout in receive')
56     pycom.rgbled(0x000000) # black
57
58 s.setblocking(False)
59 time.sleep(29)

```

Finalement, on attend des données. Dans le mode le plus fréquent de LoRaWAN, un équipement ne peut recevoir des données que dans une courte fenêtre après une émission. Si des données arrivent en dehors de cette fenêtre, le LNS mémorise l'information et la transmettra quand il recevra une trame de l'objet. Cela permet d'économiser beaucoup d'énergie car le capteur peut dormir de ses deux oreilles, il sait qu'il ne manquera pas une information.

La voie descendante est une ressource rare, et il ne faut généralement pas trop l'utiliser⁶, donc il ne devrait pas y avoir souvent de messages en retour. Sauf ici, pour tester toute la chaîne de transmission :

- ligne 49, l'appel à `recv` est aussi bloquant tant que des données ne sont pas arrivées.
- lignes 51 et 52, si des données arrivent, elles sont affichées et la LED passe au vert.
- lignes 53 à 55, si aucune données arrivent, le temporisateur se déclenche au bout de 10 secondes générant l'exception. Un message averti l'utilisateur de l'absence de données et la LED est éteinte.

Si tout va bien, qu'une passerelle radio est à portée de votre objet, le programme `lorawan_send_and_receive.py` va se connecter au réseau après avoir affiché 3 fois `Not yet joined...` :

```

>>> Running lorawan_send_and_receive.py
>>>
>>>
devEUI: b'70b3d54994c61237'
Not yet joined...
Not yet joined...
Not yet joined...
timeout in receive
timeout in receive

```

correspondant au 5 secondes prévues par le standard avant d'envoyer le message d'acceptation de l'objet. La LED passe du blanc au rouge indiquant qu'un message a été émis puis doit s'éteindre et le message `timeout in receive` est affiché indiquant que l'Objet n'a pas reçu de message en réponse à son émission.

Analyse des traces

Côté LNS, il est possible de voir le trafic sur la page résumant les caractéristiques de l'objet dans la fenêtre *live data* (cf. figure 11.3 on the next page).

Les événements sur la figure se lisent de bas en haut comme le confirme l'horodatage, en cliquant sur la ligne correspondante, TTN affiche les messages JSON utilisés par le LNS pour traiter l'information. On y retrouve les données échangées :

6. TTN ne facture pas les données descendantes, mais d'autres opérateurs le font, faites attention si vous adaptez ces exemples à d'autres environnements, car dans les premiers tests, nous utiliseront la voie de retour.

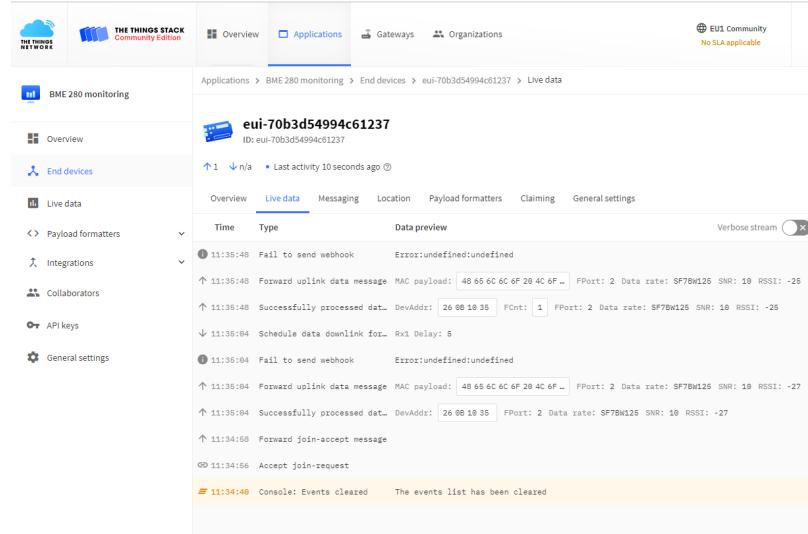


FIGURE 11.3 – Trace d'un objet.

- L'événement **Accept Join-request** indique que le LNS a reçu le message Join de l'objet, indiquant qu'il voulait rejoindre le réseau. Le LNS l'a traité, a vérifié qu'il y avait un objet déclaré avec le bon **devEUI** et que l'**AppKey** utilisée par l'objet pour signer sa demande était identique à la valeur configurée. Dans la trace suivante, on peut retrouver le **devEUI** de l'objet et la **devAddr**, qui lui a été attribuée temporairement. Cette dernière sera utilisée par la suite.

```
"device_ids": {
  "device_id": "eui-70b3d54994c61237",
  "application_ids": {
    "application_id": "plido-appl"
  },
  "dev_eui": "70B3D54994C61237",
  "join_eui": "0000000000000000",
  "dev_addr": "260B1035"
}
```

- L'événement **Forward Join-accept message** indique que le LNS envoie un message d'acceptation à l'Objet. Ce message est différé de 5 secondes pendant lesquelles l'Objet dort.
- L'événement **Successfully processed data message** montre que le LNS a reçu correctement le message de données émis par l'Objet. La trace donne des indications sur certains paramètres physiques, tels que le Spreading Factor (ici SF7 soit la modulation la plus rapide et la moins robuste), le SNR qui indique le ratio entre la puissance du signal reçu par rapport au bruit et le Received Signal Strength Indicator (RSSI) donnant la force du signal.
- L'événement **Forward uplink data message** donne plus d'information sur le message et son contenu. Le champs **frm_payload** donne le contenu déchiffré par le LNS mais codé en **base64**. On y retrouve le message d'origine Hello LoRa.

```
"uplink_message": {
  "session_key_id": "AX4VN0uMbClcPKF63sKy6A==",
  "f_port": 2,
  "frm_payload": "SGVsbgG8gTG9SYQ=="}
```

```

    "rx_metadata": [
        {
            "gateway_ids": {
                "gateway_id": "gateway-lo",
                "eui": "B827EBFFFE08F8A8"
            },
            "time": "2022-01-01T10:35:04.247807Z",
            "timestamp": 2661650827,
            "rssi": -27,
            "channel_rssi": -27,
            "snr": 10,
            "uplink_token": "ChgKFgoKZ2FOZX....",
            "channel_index": 1
        }
    ]
}

```

Question 11.2.1: Mauvais appKey

Quel message de trace obtenez vous du LNS, si l'Objet n'est pas configuré avec le même **AppKey** que le LNS ?

Question 11.2.2: Autonomie

Quel impact sur les batteries ?

Le **fPort** que l'on retrouve dans les trames de données LoRaWAN permet d'indiquer le programme qui va traiter l'information. La valeur 0 est spéciale et indique une trame de configuration de la couche LoRaWAN, les autres valeurs jusqu'à 223 désignent des applications spécifiques. Bien qu'il soit peu utilisé, il est parfois utile de la changer.

Question 11.2.3: fPort

Dans les traces précédentes, quelle est la valeur du fPort par défaut utilisé par le LoPy ?

Question 11.2.4: Changement de fPort

L'instruction bind permet de modifier le **fPort** pour une transmission. Modifiez le programme pour utiliser le fPort 10 et vérifiez l'effet dans les traces de TTN.

Configuration du connecteur

Le LNS n'a pas été configuré pour envoyer le message à un Serveur d'Application (Application Server). Comme pour **Sigfox**, il faut définir une URI pour indiquer au LNS où poster les données.

Pour configurer un connecteur, il faut aller sur le menu de gauche *Integrations*. TTN propose des intégrations directes avec des services cloud existants, mais il est également possible de configurer son propre connecteur :

- via **MQTT**, TTN joue le rôle de *broker* et fourni le nom du **topic** et le secret.

- en stockant localement des données par le choix ***Storage integration*** et en y accédant par une requête GET, comme Sigfox le propose (cf. chapitre 10.5.3 on page 122). Cela permet de recevoir les données si l'on est situé derrière un NAT, mais rend la réception asynchrone puisque l'application doit interroger régulièrement le serveur de TTN.
- en configuration une URI sur le serveur pour que celui-ci produit une requête POST quand une trame de donnée est reçue (choix ***Webhooks***).

Nous allons également privilégier pour TTN ce dernier mode de fonctionnement, car si il nécessite une configuration du NAT, il permet de recevoir les données de manière synchrone lorsqu'elles sont traitées par le LNS.

Il faut cliquer sur ***Webhooks***. Un certain nombre de services sont proposés, mais nous allons définir le nôtre à l'instar de ce que nous avons fait avec Sigfox (cf. chapitre 10.5.4 on page 124). Pour préparer la connectivité, il est nécessaire :

- Si vous êtes dans le cloud, l'adresse s'obtient en tapant la commande `ifconfig`.
- Si vous êtes derrière une box ou le routeur de votre fournisseur d'accès, vous avez une adresse privée. Un site web comme `myip.wtf` vous retournera l'adresse IPv4 publique. Vous devez ensuite configurer votre router d'accès pour qu'il envoie les paquets TCP ayant le numéro de port 9999 à votre ordinateur.

Cela va permettre de former l'URI suivante :

```
http://aaa.bbb.ccc.ddd:9999/ttn
```

où `aaa.bbb.ccc.ddd` représente l'adresse IP publique de votre équipement. Comme cette adresse peut changer au cours du temps, il est préférable d'utiliser un service de DNS dynamique pour obtenir un nom de domaine plus stable sur la durée.

La page listant les ***Webhooks*** associés à l'application est vide. Il faut cliquer sur `pour en ajouter un nouveau`, il convient de choisir *Custom webhook*, les autres définissent les connecteurs vers des services de gestion des données (cf. figure 11.4 on the following page).

Dans ce menu, il faut indiquer :

- l'identifiant du webhook :
- le format JSON :
- l'URI que nous venons de construire :
- il n'est pas nécessaire pour l'instant de remplir la *Download API Key* qui servira pour envoyer des données à l'Objet. Nous y reviendrons par la suite.
- il faut choisir les événements qui déclencheront une requête POST. Dans notre cas, on ne choisit que la réception d'un message montant (*Uplink message*). On peut compléter le chemin de l'URI précédemment définie avec des éléments qui permettront au serveur REST de mieux identifier le type de requête. Dans notre cas, cela n'est pas nécessaire. `/ttn` identifie par conséquent les messages reçus par le LNS des Objets.

Une fois le *webhook* validé, le programme `generic_relay.py` doit être lancé. Il va créer un serveur Web qui va attendre les requêtes POST du LNS de TTN.

```
94 @app.route('/ttn', methods=['POST']) # API V3 current
95 def get_from_ttn():
96     fromGW = request.get_json(force=True)
```

Applications > BME 280 monitoring > Webhooks > Add > Custom webhook

Add webhook

General settings

Webhook ID *
plido-webhook

Webhook format *
JSON

Endpoint settings

Base URL
http://[REDACTED]:9999/ttn

Downlink API key
[REDACTED]

The API key will be provided to the endpoint using the "X-Downlink-Apikey" header

Additional headers
+ Add header entry

Enabled messages

For each enabled message type, an optional path can be defined which will be appended to the base URL

Uplink message
 Enabled /path/to/webhook

Join accept
 Enabled

FIGURE 11.4 – Configuration d'un webhook.

```

98     downlink = None
99     if "uplink_message" in fromGW:
100
101         payload = base64.b64decode(fromGW["uplink_message"]["frm_payload"])
102         downlink = forward_data(payload)
103
104     if downlink != None:
105         from ttn_config import TTN_Downlink_Key
106
107         downlink_msg = {
108             "downlinks": [
109                 {
110                     "f_port": fromGW["uplink_message"]["f_port"],
111                     "frm_payload": base64.b64encode(downlink).decode()
112                 }
113             ]
114         }
115         downlink_url = \
116             "https://eu1.cloud.thethings.network/api/v3/as/applications/" + \
117             fromGW["end_device_ids"]["application_ids"]["application_id"] + \
118                 "/devices/" + \
119                 fromGW["end_device_ids"]["device_id"] + \
120                     "/down/push"
121
122         headers = {
123             'Content-Type': 'application/json',
124             'Authorization' : 'Bearer' + TTN_Downlink_Key
125         }
126
127         x = requests.post(downlink_url,
128                             data = json.dumps(downlink_msg),
129                             headers=headers)
130
131     resp = Response(status=200)
132     return resp

```

Ce programme utilise le module **Flask** pour créer un serveur Web. Il associe le chemin /ttn à la fonction get_from_ttn pour les requêtes de type POST (lignes 93 et 94). Quand une requête de ce type arrive, la variable fromGW contient l'objet JSON⁷ (ligne 95). Si elle contient la clé uplink_message, les données sont extraites (ligne 100) pour être envoyé au travers de l'interface *loopback* grâce à la fonction forward_data à un programme qui les traite. Si cette fonction retourne la valeur None, le programme ne souhaite pas répondre à l'Objet, le serveur Web acquitte positivement (code 200) la requête venant du LNS (ligne 130 et 131).

```

def forward_data(payload):
28     global verbose, forward_port, forward_address
29
30     inputs = [sock]
31     outputs = []
32
33     if verbose:
34         print ("--UP->", binascii.hexlify(payload))
35         sock.sendto(payload, (forward_address, forward_port))
36
37     readable, writable, exceptional = select.select(inputs, outputs, inputs, 0.1)
38
39     if readable == []:

```

7. convertie implicitement en dictionnaire Python.

```

40     if verbose:
41         print ("no DW")
42     return None
43
44     for s in readable:
45         replyStr = s.recv(1000)
46         if verbose:
47             print ("<-DW--", binascii.hexlify(replyStr))
48         return replyStr
49
50     return None

```

Si le mode verbose est activé, le message à envoyer au programme de traitement des données est affiché en hexadécimal grâce à la fonction `hexlify` (lignes 33 et 34). Puis le message est effectivement émis (ligne 35)⁸

Pour attendre une réponse qui n'est pas systématique, on ne peut pas utiliser l'appel `recvfrom` car celui-ci est bloquant. A la place, on utilise l'appel `select` du module du même nom. Il permet d'attendre sur plusieurs événements à la fois provenant de différentes sockets (ligne 37) où :

- `inputs` est un tableau des sockets pouvant avoir reçu des données. Dans notre cas (ligne 30) il est initialisé avec la socket dialoguant avec le programme traitant les données;
- `outputs` est un tableau qui contient les sockets dans lesquelles il serait possible d'écrire. Dans notre cas, ce tableau est vide (ligne 31).
- le troisième concerne les exceptions. Dans notre cas, on répète le tableau des sockets `inputs`
- finalement le quatrième paramètre indique la durée d'attente dans la fonction `select`. Dans notre cas, il est fixé à 0.1 seconde. ce temps est à la fois compatible avec une réponse du programme de traitement des données, et la réponse que l'on doit envoyer au LNS pour que la réponse soit associée au message montant.

`select` retourne dans un tuple de trois éléments, la ou les sockets qui ont déclenché son retour. Dans notre cas, se sont :

- soit l'expiration du temporisateur, dans ce cas, la variable `readable` qui contient la liste des sockets ayant reçues des données est vide (ligne 39) et la valeur `None` est renournée.
- soit des données qui sont arrivées dans une ou plusieurs socket. la variable `readable` contient le tableau de ces sockets. La variable `s` contient la première socket du tableau, les données sont lues et envoyées en retour.

Le programme `generic_relay.py`, lancé avec l'option `-v` montre de manière synthétique le trafic.

```

>python3 generic_relay.py -v
--UP-> b'48656c6c6f204c6f5261',
no DW
63.34.43.96 - - [01/Jan/2022 18:27:45] "POST /ttn HTTP/1.1" 200 -
--UP-> b'48656c6c6f204c6f5261',
no DW
34.255.49.188 - - [01/Jan/2022 18:28:31] "POST /ttn HTTP/1.1" 200 -

```

Le contenu du message montant est affiché en hexadécimal et correspond au fameux contenu Hello LoRa. Comme il n'y a pas de réponse, no DW est ensuite affiché.

8. l'adresse du destinataire et le port sont configurés lors de l'analyse des paramètres.

Traitement des données

La fonction `forward_data` du programme `generic_relay.py` envoie les données à l'adresse de **loopback** 127.0.0.1 et sur le port 33033⁹.

Le programme `display_receive_and_send.py` traite les données de l'Objet.

Listing 11.3 – `display_receive_and_send.py`

```

1 import socket
2 import binascii
3
4 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
5 s.bind(('0.0.0.0', 33033))
6
7 while True:
8     data, addr = s.recvfrom(1500)
9     print (data)
10    s.sendto("Pleased to meet you!".encode(), addr)
```

Il attend les données sur le port 33033. La fonction `recvfrom` retourne les données stockées dans la variable `data` et l'adresse de celui qui a envoyé les données stockée dans la variable `addr`. Les données sont affichée ligne 9 et le message `Pleased to meet you` est retourné à l'émetteur, en ligne 10 grâce à la fonction `sendto`.

En lançant le programme dans une nouvelle fenêtre, on reçoit correctement le message du LoPy.

```
>python3 display_receive_and_send.py
b'Hello LoRa'
```

En revanche, le programme `generic_relay.py` produit une erreur.

```
--UP-- b'48656c6c6f204c6f5261',
<-DW-- b'506c656173656420746f206d65657420796f7521',
63.34.43.96 - - [01/Jan/2022 19:06:51] "POST /ttn HTTP/1.1" 500 -
Traceback (most recent call last):
... 
```

Le message en retour a été reçu par la fonction `forward_data`, mais la voie descendante n'a pas encore été configurée. L'erreur est liée à l'absence du fichier `ttn_config.py`, ligne 104, ce qui force Flask à retourner le status 500 et afficher les fonctions qui ont conduit à l'erreur.

Voie retour

Il n'y a aucun problème de sécurité quand le LNS envoie au serveur d'application des données reçues de capteurs, car l'endroit où il les transmet a été configuré par leur propriétaire. Par contre, dans l'autre sens, le LNS doit s'assurer que les données à transmettre à l'Objet, proviennent bien d'un serveur d'application autorisé.

Comme pour Beebotte, une clé secrète va être utilisée. A cette clé, le LNS peut associer des droits pour plus ou moins limiter les accès. Dans le menu de gauche, cliquez sur *API key* puis sur *+ Add API Key*. Dans l'écran suivant, donnez un nom à votre clé et choisissez *Grant Individual Right* pour limiter l'usage de la clé. Sélectionnez *Write downlink application traffic* et validez.

⁹. Ces deux valeurs peuvent être respectivement changées en utilisant les arguments `--forward_address` et `--forward_port`.

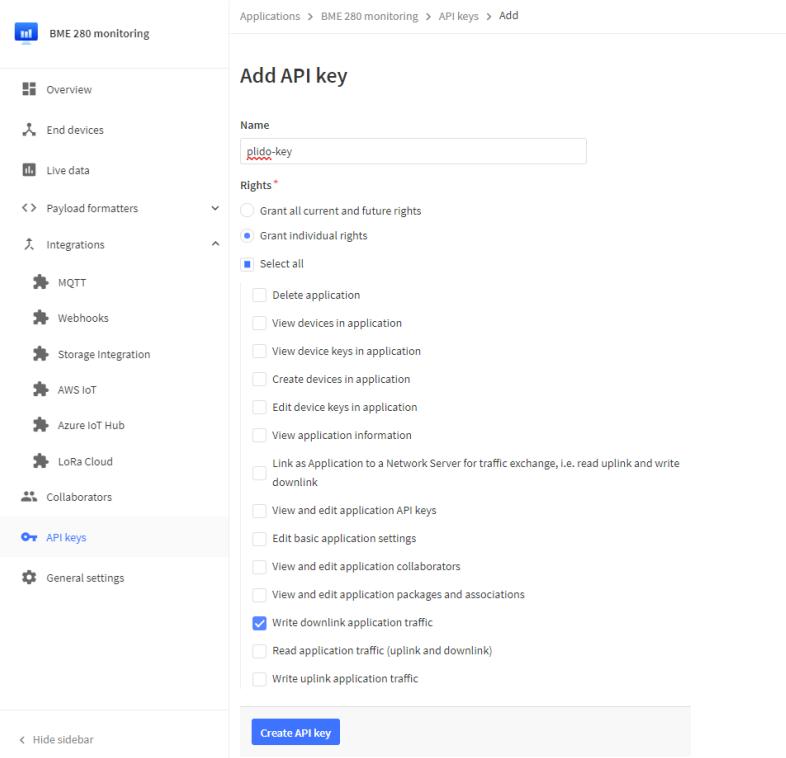


FIGURE 11.5 – Création d'une clé

Recopiez la clé et confirmer que vous l'avez bien fait en validant *I have copied the key*. Editez le fichier `ttn_config.py` en y insérant la clé obtenue.

Listing 11.4 – `ttn_config.py`

```
1 TTN_Downlink_Key = "ENTER YOUR KEY"
```

Relancez le programme `generic_relay.py`, le LoPy affiche la réponse.

Le listing suivant reprend la construction de la requête POST vers le LNS pour transporter le message descendant :

```
104     if downlink != None:
105         from ttn_config import TTN_Downlink_Key
106
107         downlink_msg = {
108             "downlinks": [
109                 "f_port": fromGW["uplink_message"]["f_port"],
110                 "frm_payload": base64.b64encode(downlink).decode()
111             ]
112         }
113         downlink_url = \
114             "https://eu1.cloud.thethings.network/api/v3/as/applications/" + \
115             fromGW["end_device_ids"]["application_ids"]["application_id"] + \
116                 "/devices/" + \
117                 fromGW["end_device_ids"]["device_id"] + \
118                 "/down/push"
```

```

118     headers = {
119         'Content-Type': 'application/json',
120         'Authorization' : 'Bearer' + TTN_Downlink_Key
121     }
122
123     x = requests.post(downlink_url,
124                         data = json.dumps(downlink_msg),
125                         headers=headers)

```

- ligne 104, la clé autorisant l'envoi de message descendant est copié dans la variable TTN_Downlink_Key.
- lignes 106 à 110, le format JSON attendu par le LNS pour envoyer un message descendant est construit, plusieurs messages peuvent être envoyés à la fois, d'où l'utilisation d'un tableau. Chacun des éléments contient un fPort qui est recopié du message montant et des données qui proviennent de la socket (variable downlink codé en base64).

```
f'downlinks': [f'frm_payload': 'UGx1YXN1ZCB0byBtZWV0IH1vdSE=', 'f_port': 10}]}]
```

- lignes 111 à 116, l'URI pour accéder au service est construite. Elle contient deux éléments variables qui identifient l'application et l'Objet. Ceux-ci sont également extraits du message montant.

```
https://eui.cloud.thethings.network/api/v3/as/applications/plido-appl/devices/eui-70b3d54994c61237/down/push
```

- lignes 118 à 121, les options qui seront ajoutées à l'en-tête HTTP sont indiquées. Elles contiennent la clé d'authentification.
- finalement, ligne 123 à 125, ces différents éléments sont passés à la fonction post du module requests.

11.3 Ajout d'une passerelle radio

Vous voulez installer une passerelle radio LoRaWAN chez vous pour bénéficier d'un réseau longue portée. Nous avons choisi d'utiliser un **Pygate** qui nous permettra de rester dans l'univers Pycom que nous connaissons bien maintenant et surtout qui est une des solutions les moins chères.

11.3.1 Installation du Pygate

Pour cela vous devez posséder une carte spécifique qui agit comme une carte d'extension. On y connecte un processeur. Ce dernier n'est pas forcément un LoPy car nous n'utiliserons pas sa partie radio LoRa. En effet, le Pygate dispose d'un autre composant radio Semtech qui permet l"écoute simultanée de huit canaux LoRa. Un LoPy lui ne peut écouter que sur un seul canal à la fois. En effet lors de l'émission d'une donnée, l'objet choisi aléatoirement un canal radio pour que le message soit reçu. Il faut que la passerelle radio écoute sur tous les canaux possibles.

On insère le Pycom sur la Pygate et on connecte l'antenne sur le connecteur du Pygate. Dans un second temps on doit adapter le Pycom à ses nouvelles fonctions, en téléchargeant le bon microprogramme. On le fait en lançant le programme Pycom Firmware Update et en choisissant Pygate.

Youtube



Dans un troisième temps, on ouvre le répertoire Pygate avec Atom. On y trouve le fichier `wifi_conf.py` qui va permettre à la Gateway d'avoir accès à votre réseau wifi, il peut contenir les mêmes identifiants que lorsque vous aviez connecter votre LoPY à votre réseau wifi. Le programme `main.py` est préconfigurée pour utiliser le LNS européen de The Things Network.

Le fichier `config.json` contient la description des fréquences utilisables dans cette zone. En y jetant un coup d'œil, on peut voir huit fréquences qui sont écoutées par la passerelle radio. Vous trouverez sur le site de The Things Network les autres configurations possibles.

On téléverse le programme dans la mémoire du Pycom de la passerelle radio et comme il s'appelle `main.py` il va se lancer tout seul au redémarrage. Il se connecte au wifi, puis va afficher l'identifiant la passerelle. Recopiez le car nous allons avoir besoin plus tard. Après on a plein de texte indiquant que la passerelle est en train de configurer le réseau LoRaWAN. Quand la led passe au vert, la passerelle est active.

11.3.2 Configuration de The Things Network

Maintenant nous devons informer The Things Networks de l'existence de notre passerelle radio. Si ce n'est pas le cas vous devrez créer un compte sur The Things Network. Aller sur console choisissez *Europe 1* si vous êtes en Europe ou la zone géographique correspondante. Ajoutez une passerelle en donnant :

- un identifiant textuel qui est un non lisible sans espaces ;
- l'identifiant de la passerelle qui correspond à la valeur hexadécimale affiché au démarrage ;
- un texte pour mieux décrire la passerelle ;
- finalement le plan fréquences¹⁰.

On valide la passerelle et elle est ajoutée au LNS. Vous allez pouvoir visualiser les trames qu'elle reçoit.

11.4 Vue générale des échanges

La figure 11.6 on the next page en illustre le fonctionnement global. La première émission du LoPy est captée par un ou plusieurs passerelles radio, qui relaient le message via un format JSON spécifique appelé ***Packet Forwarder***, cela peut être de directement en UDP ou en utilisant MQTT. Le LNS après identification de l'Objet, envoie une requête POST contenant les données émise vers une destination configurée par une URI. Cette requête arrive au routeur du domicile, qui l'envoie à un équipement dans le domicile en fonction de la configuration du NAT. Le programme `generic_relay.py` traite la requête, en extrait le contenu et l'envoie via une socket au programme. Si ce dernier ne répond pas, le temporisateur se déclenche et la requête est acquittée.

Sinon, si le programme `generic_relay.py` reçoit des données en réponse avant le déclenchement du temporisateur, il initie une nouvelle requête POST vers le LNS. Dans notre cote, quand le LNS l'acquitte, la requête initiale est à son tour acquittée.

10. si les objets sont facilement accessibles on peut choisir le Spreading Factor 9 pour la voie descendante sinon si vous n'arrivez pas à joindre vos objets, vous pouvez vous replier sur le Spreading Factor 12

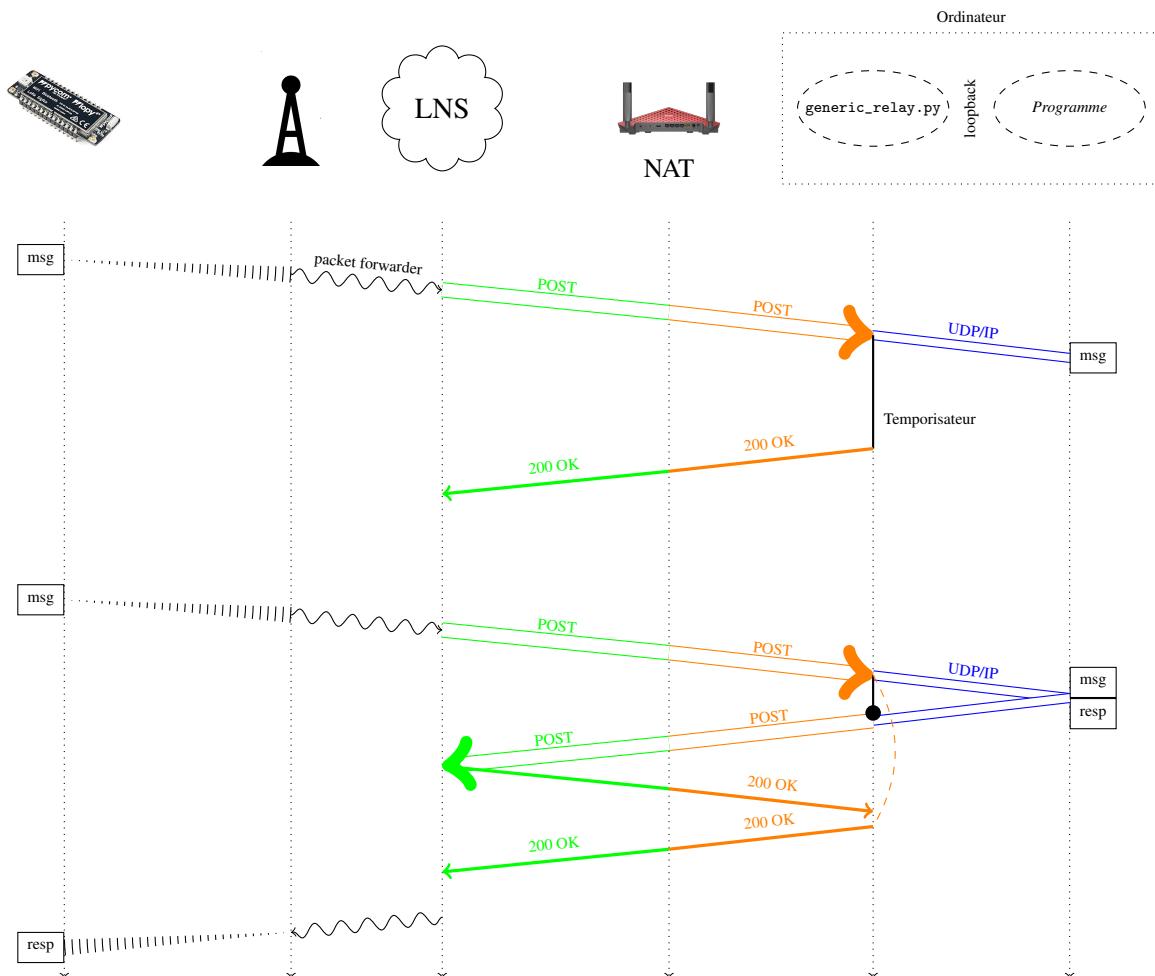


FIGURE 11.6 – Diagramme temporel des échanges.

Le message descendant est gardé en mémoire dans le LNS jusqu'à ce que la fenêtre de réception du Lopy s'ouvre.

11.5 Thermomètre LoRaWAN

Le programme `lorawan_temperature.py` combine la manière de rejoindre le réseau LoRaWAN que l'on a vu précédemment et la partie envoie des séries temporelles différentielles des mesures de température. La seule difficulté réside dans la taille de cette série temporelle. En effet, suivant le **Data Rate** le volume de données transportées diffère. Le tableau 12.2 sur la page 154 donne la correspondance entre le *Data Rate* et les autres paramètres physiques. Il indique également la tailles maximales des données transportées.

Ce changement de taille peut poser des problèmes de compatibilités, si l'on choisit une taille trop grande pour être transmise. En théorie, l'on connaît le *Data Rate* choisi et on peut adapter la taille en conséquence, mais l'opérateur peut aussi modifier le *Data Rate* de l'Objet en fonction des conditions de transmission, et il n'existe aucune instruction en micro-python pour récupérer le *Data*

Data Rate	Spreading Factor	Largeur de bande	Taille Maximum (octets)
DR0	SF12	125 KHz	51
DR1	SF11	125 KHz	51
DR2	SF10	125 KHz	51
DR3	SF9	125 KHz	115
DR4	SF8	125 KHz	242
DR5	SF7	125 KHz	242
DR6	SF7	250 KHz	242

TABLE 11.1 – Data Rate en Europe

Rate réellement utilisé. Pour simplifier la mise en œuvre, la taille maximale de 50 octets a été choisie.

Il suffit de remplacer `display_receive_and_send.py` par `display_server.py` pour que la série temporelle soit traitée et le résultat envoyé à Beebotte.

Question 11.5.1: US902

En vous aidant du lien suivant <https://www.thethingsnetwork.org/docs/lorawanRegional-parameters/> indiquant les paramètres régionaux. Est ce que le programme `lorawan_temperature.py` fonctionnerait sur un réseau LoRaWAN situé en Amérique du Nord ?



12. CoAP

Les principes REST avec leur représentation de l'information par des ressources pointées par des identifiants globalement unique est une des clés du succès de l'Internet et de la composition de services distribués. Même si ce n'est pas l'unique solution, il est indispensable que les informations provenant d'objets contraints puissent s'intégrer dans cette toile d'araignée mondiale. CoAP, pour Constrained Application Protocol, permet cette intégration à un meilleur coût en termes d'empreinte mémoire ou protocolaire que ne le permettrait le protocole HTTP.

12.1 Introduction

Dans les chapitres précédents, nous avons vu que CBOR permettait d'envoyer des données structurées de manière efficace, que le récepteur pouvait faire la différence entre un entier ou une chaîne de caractères et également savoir combien d'éléments comptaient un dictionnaire ou un tableau. En plus d'un gain de place par rapport à JSON, la complexité pour sérialiser ou déserialiser était limitée, conduisant à des implémentations peu gourmandes en mémoire.

Mais CBOR n'est pas suffisant pour une bonne interopérabilité. Quand un récepteur reçoit les données, il faut qu'il sache qu'il s'agit d'un codage CBOR et pas d'une autre structure. De plus, il faut que le récepteur sache quoi faire de ces données. Dans les exercices, nous avons transmis des séries temporelles correspondant à des relevés de température. Mais si nous voulions également transmettre l'humidité et la pression, comment le récepteur ferait la différence ?

Nous avons une solution pour répondre à ces questions : l'utilisation de ressources.

Nous avons vu qu'avec le paradigme REST, les ressources étaient nommées. Donc, pour distinguer les différentes séries temporelles, il suffit d'utiliser un nom (ou un URI) différent. Les ressources contiennent également des méta-informations et il est donc possible de transporter le format de codage pour indiquer qu'il s'agit de CBOR, de JSON, de CSV...

Malgré son universalité, ce modèle pose un problème pour les objets contraints :

Ver	T	TKL	code	Message ID
Token (if any)				
Options (if any)				
11111111		Payload (if any)		

FIGURE 12.1 – Format d'un message CoAP

- HTTP utilise TCP pour fiabiliser les communications entre le client et le serveur. Or, TCP est gourmand en ressources. Il faut de la mémoire pour stocker les paquets non acquittés ou hors séquence, un grand nombre d'heuristiques doivent être mises en œuvre pour améliorer ses performances ; la souplesse pour créer des en-têtes peut s'avérer être un désavantage pour un objet contraint. Ainsi, la requête HTTP vers le serveur `www.arduino.cc` peut avoir cette forme :

```
GET / HTTP/1.1\r\n
Host: www.arduino.cc\r\n
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:25.0) Gecko/20100101 Firefox/25.0\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
Accept-Language: en-US,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Connection: keep-alive\r\n
\r\n
```

En dessous de la première ligne qui demande la ressource à la racine (généralement `index.html`), on trouve un certain nombre de lignes sous le format :

```
Nom du champ : valeur du champ\r\n
```

qui vont indiquer au serveur le nom du serveur que l'on veut atteindre, la description du navigateur et les formats que celui-ci peut accepter.

Le but de CoAP est de définir un protocole beaucoup plus strict qui sera donc plus facile à mettre en œuvre mais qui pourra interopérer avec HTTP afin de préserver les principes définis par l'architecture REST et profiter du nommage des ressources pour que les ressources contraintes participent à la grande toile d'araignée mondiale.

12.2 Format d'une en-tête CoAP

L'en-tête des messages CoAP est de taille variable mais très structurée, comme le montre la figure 12.1.

Le premier mot de 32 bits est présent dans tous les messages CoAP :

- le champ Ver, sur 2 bits, contient le numéro de version du protocole, qui vaut 01 dans la version actuelle ;
- le champ T pour **Type**, également sur 2 bits, indique la nature du message (00 : **C**ONfirmable, 01 : **N**ON confirmable, 10 : Acquittement, 11 : Reset) ;
- le champ TKL, sur 4 bits, donne la longueur en octets du champ token démarrant au deuxième mot de 32 bits. Si la valeur est 0, ce champ est absent. Les valeurs de 1 à 8 indiquent la longueur. Les valeurs de 9 à 15 ne sont pas autorisées ;



- le champ **Code**, sur 1 octet, permet un codage assez subtil de la nature de la requête ou de la réponse (cf. chapitre suivant) ;
- le champ **Message ID**, sur 2 octets, identifie les requêtes.

12.2.1 Codage de code

Dans beaucoup de protocoles applicatifs comme FTP ou HTTP, le serveur renvoie un code sur 3 caractères indiquant si la requête s'est exécutée correctement ou non. Les codes commençant par le chiffre :

- 1 informent que la requête est en train d'être traitée normalement. Ce type de notification n'est pas pris en compte avec CoAP ;
- 2 indiquent que la requête a été acceptée et traitée correctement ;
- 3 permettent d'indiquer une indirection ;
- 4 font référence à une erreur du côté client, due à une mauvaise syntaxe ou une requête qui ne peut être traitée. Ainsi la célèbre erreur 404 indique que le client a demandé une page qui n'existe pas sur le serveur ;
- 5 désignent une erreur du côté du serveur.

Le site web de l'Internet Assigned Numbers Authority (IANA)¹ donne les erreurs que l'on retrouve dans le protocole HTTP. Comme indiqué précédemment, le chiffre de gauche varie entre 1 et 5 tandis que les deux chiffres de droite, précisant la raison de la notification, varient généralement entre 0 et 31.

Pour permettre une représentation plus compacte, CoAP va coder cette chaîne de caractères dans un octet. Les trois bits de gauche désignent la nature du code et les 5 à droite donneront la raison.

Ainsi le code d'erreur HTTP 415 (*Unsupported Media Type*) se note en CoAP 4.15, s'écrit en binaire 100.01111 et en décimal 143. Cette notation concerne les réponses aux requêtes mais elle laisse de la place pour coder également les requêtes. En effet, le code avec les trois premiers bits à 0 n'est pas utilisé pour coder les notifications.

Plusieurs requêtes compatibles avec l'architecture REST peuvent être codées :

- **GET**, codé 0x01, retrouve le contenu d'une ressource présente sur le serveur et désignée par un URI ;
- **POST**, codé 0x02, stocke une valeur sur une ressource existante présente sur le serveur ;
- **PUT**, codé 0x03, crée une ressource sur le serveur et lui affecte une valeur ;
- **DELETE**, codé 0x04, supprime une ressource sur le serveur.

Notez que la valeur 0x00 peut être utilisée dans certains cas.

12.2.2 Utilisation du champ Message ID

Le champ **Message ID** sur 2 octets sert à identifier les messages CoAP afin de détecter les duplicitas. Cette valeur est recopiée dans les acquittements pour permettre de savoir quel message est acquitté. Ils ne doivent pas être réutilisés pendant une période fixée.

1. <http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml>
[http-status-codes-1](http://www.iana.org/assignments/http-status-codes-1)



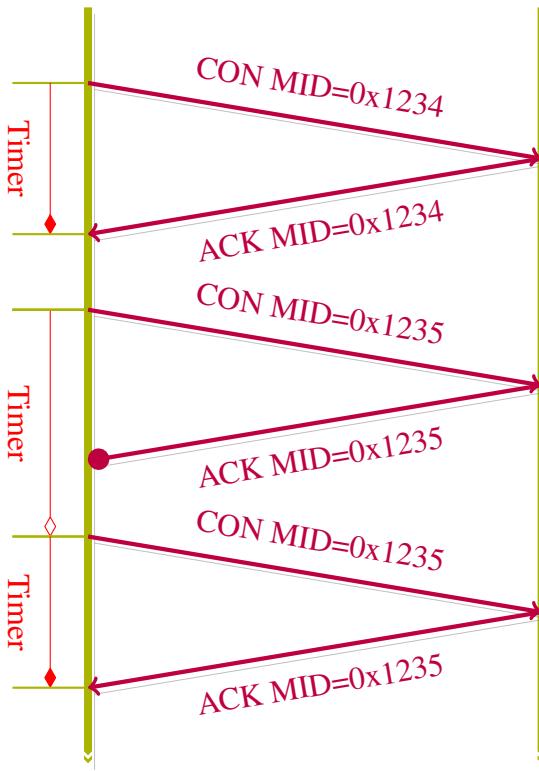


FIGURE 12.2 – Échanges fiabilisés avec CoAP

Le protocole CoAP repose sur la couche UDP pour des raisons de simplicité de mise en œuvre. Il peut être parfois nécessaire de fiabiliser le transfert des données. Pour se faire, CoAP dispose d'une sous-couche implantant un protocole très simple. Chaque message contient un champ `message ID` et trois types de trames sont disponibles :

- les messages de type **CON** pour *confirmable* indiquent qu'ils doivent être acquitté par le récepteur.
- les message de type **ACK** contiennent cet acquittement, le champ `message ID` du message à acquitter est recopié dans ce message. Ce message peut également contenir des données.
- les message de type **NON** pour *non confirmable* sont de purs datagrammes, ils ne seront pas acquittés par le récepteur, leur perte ne sera pas détectée par le protocole CoAP. Par contre, le champ `message ID` permet de détecter des messages dupliqués.

La notion d'émetteur/récepteur est dissociée des rôles de client ou de serveur définis par REST. Un client REST peut émettre des trames **CON**, **NON** ou **ACK**, de même pour un serveur.

Un message de type **RST** s'ajoute aux trois types précédents, il peut être émis par exemple quand un des nœuds a perdu son contexte suite à un redémarrage et ne sait plus traiter les réponses qu'il reçoit.

Paramètre	Valeur par défaut
ACK_TIMEOUT	2s
ACK_RANDOM_FACTOR	1.5
MAX_RETRANSMIT	4
MAX_LATENCY	100s
PROCESSING_DELAY	2s

TABLE 12.1 – Valeurs par défaut proposées par le [RFC 7252](#)

La figure 12.2 on the facing page montre des échanges fiabilisés avec le protocole CoAP. Les messages de type CON impliquent un acquittement en retour. L'émetteur arme un temporisateur et à son expiration ré-émet le message. Si il reçoit un message d'acquittement contenant la même valeur dans le champ Message ID, le message est acquitté. Ce cas est illustré avec le Message ID valant 0x1234.

Si par contre, à l'expiration du temporisateur, l'acquittement n'est pas arrivé, le message initial, gardé en mémoire est retransmis. Le [RFC 7252](#) suggère un temporisateur initial de 2 secondes dont la valeur double à chaque retransmission, et 4 transmissions d'un même message sont possibles. Ces valeurs peuvent être changées pour s'adapter au contexte. Les durées de tempéroration incluent un aléa, pour éviter une synchronisation entre plusieurs émetteurs pouvant favoriser des collisions de trame.

La valeur du champ Message ID n'a de sens que pour un échange, si par exemple un récepteur reçoit les valeurs 0x1234 et 0x1236, il ne doit pas en déduire que le message 0x1235 s'est perdu. Plusieurs transmission peuvent également se dérouler en parallèle ; un émetteur n'a pas besoin d'attendre un acquittement pour envoyer la trame suivante.

Les messages non confirmés (type NON) utilisent également un champ message ID différent à chaque nouveau message. Il permet de détecter des duplications qui pourraient survenir dans les couches protocolaire inférieures lors du transport du message.

Pour rejeter les doublons², le récepteur doit garder une copie des Message ID émis par une source. Un simple calcul permet de définir la période de rétention des valeurs.

Un peu d'algèbre élémentaire permet de calculer cette durée. La figure 12.3 on page 155 montre le calcul du pire cas. Il s'obtient quand toutes les messages CON se perdent et sont retransmis et que seul le dernier est acquitté. Comme la durée de déclenchement du temporisateur est doublée à chaque tentative, le temps passé dans cette étape est

$$(1 + 2 + 4 + \dots) \times ACK_TIMEOUT$$

ou

$$2^{MAX_RETRANSMIT} - 1 \times ACK_TIMEOUT$$

2. dus aux duplications des couches inférieures, aux pertes des messages d'acquittement forçant une réémission (cas des messages ID 0x1235 de la figure 12.2 on the preceding page) ou à des temporiseurs mal dimensionnés déclenchant une réémission avant la réception de l'acquittement.

Paramètre	Valeurs déduites
MAX_TRANSMIT_SPAN	45s
MAX_RTT	202s
EXCHANGE_LIFETIME	247s
NON_LIFETIME	145s

TABLE 12.2 – Valeurs déduites à partir des paramètres par défaut proposées par le [RFC 7252](#)

Pour éviter les synchronisations entre les noeuds, la valeur du temporisateur est multiplié par un facteur *ACK_RANDOM_FACTOR* compris entre 1 et 1.5. Comme on se place dans le cas le plus défavorable, on prend la valeur maximale.

On en déduit la valeur d'attente maximale avant une transmission correcte *MAX_TRANSMIT_SPAN* qui est de 45 secondes.

Une fois le message transmis, il doit arriver à destination. Le [RFC 7252](#) prend une valeurs très importante de 100s pour la latence entre l'objet et le serveur³ et 2s pour le temps de traitement. Le temps d'aller retour ou Round Trip Time (RTT) maximal est de 202s.

Pour les messages **CONFIRMÉS**, le temps maximal *EXCHANGE_LIFETIME* est donc de 245s. Pour les message **NON** confirmé, on peut supposer qu'un message sera transmis plusieurs fois pour s'assurer qu'il a été correctement reçu par le destinataire. On retrouve le même calcul sans la partie acquittement, d'où une durée de 145s.

Le standard prévoit que, par défaut, la durée d'activité d'un Message ID est d'environ 5 minutes (247 s) pour les messages confirmés, et 2,5 minutes (145 s) pour les messages non confirmés. Avoir cette notion en tête peut vous éviter des heures de débogage. Supposons qu'un client commence toujours par numérotier ses messages à 1. Le récepteur va donc garder les valeurs des messages ID pendant 5 minutes. Si vous redémarrez l'objet, il va émettre de nouveaux messages, mais avec les mêmes valeurs de Messages ID qui seront acquittés, mais pas traités ignorés par le récepteur.

12.2.3 Les Tokens

CoAP utilise le protocole UDP pour communiquer. Contrairement à TCP, il n'y a pas de notion d'établissement de connexion. Il est donc difficile de faire le lien entre les établissements et les réponses, surtout si elles ne sont pas immédiates. La figure suivante illustre ce phénomène. Une requête GET est envoyée par un client à un serveur.

La réponse ne peut pas être immédiate (par exemple il faut lire une valeur sur un capteur qui demande d'être active). Le message d'acquittement ne peut pas être différé sinon, le client ne voyant pas sa requête acquittée, la retransmettrait. Le serveur acquitte avec un message Ack vide (cf. figure 12.4 on the facing page). Quand le serveur peut envoyer la ressource, il le fait à son tour dans un message de type CON qui sera à son tour acquitté. Vous pouvez remarquer que



3. Le schéma figure 12.3 on the next page n'est pas à l'échelle.

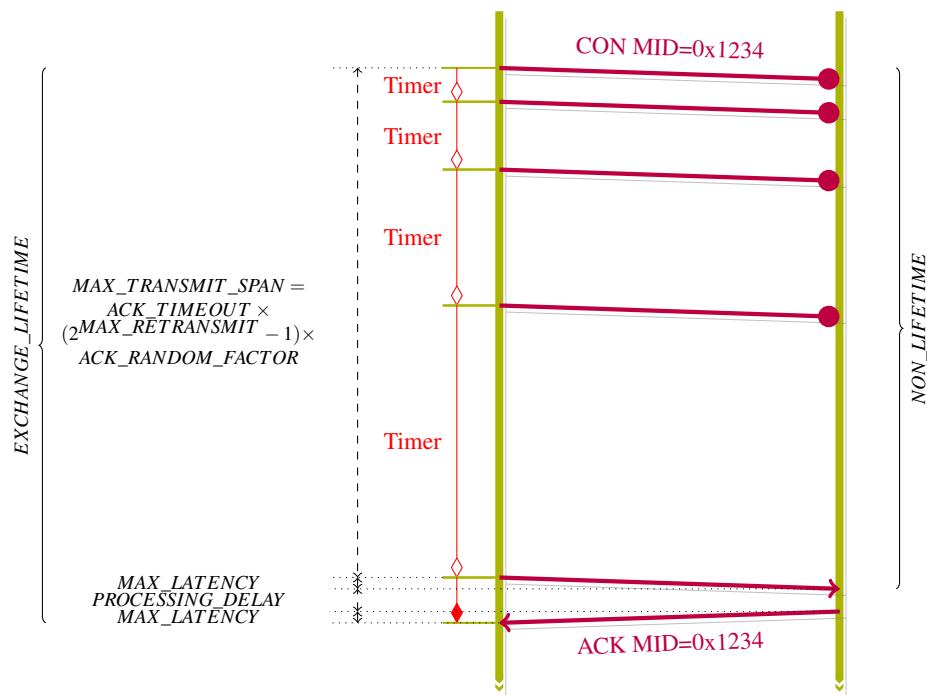


FIGURE 12.3 – Échanges fiables avec CoAP

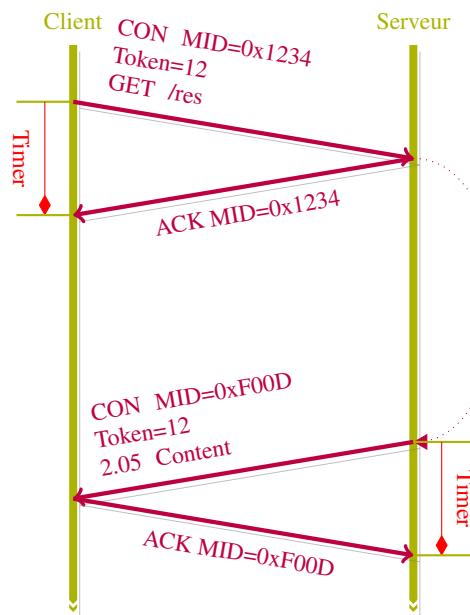


FIGURE 12.4 – Utilisation du Token

les valeurs du champ Message ID sont complètement décorrélées. Pour faire le lien entre la requête et la réponse, un token fourni par le client est recopié par le serveur. C'est pour cela que l'on peut considérer une valeur de Token comme une "connexion" entre le client et le serveur.

Le Token est une séquence binaire facultative dont la taille est comprise entre 0 (pas de token) et 8 octets. La longueur est indiquée au début de l'en-tête dans le champ Token Length (TKL) et la valeur suit immédiatement l'en-tête obligatoire avant les options.

On voit bien sur cet exemple, figure 12.4 on the previous page la décorélation entre la machine protocolaire de bas niveau basée sur les Message ID et la machine protocolaire REST. Le client envoie un message CON contenant sa requête et reçoit la réponse dans un message ACK. Dans cet exemple, il y a aussi deux niveau d'acquittement au niveau des messages et avec les notifications REST.

12.2.4 Les options CoAP

Le champ Option va contenir des **options** qui vont soit servir à améliorer le protocole de transfert des données entre le client et le serveur, soit servir à coder les en-têtes des requêtes et des réponses en garantissant une certaine compatibilité avec les en-têtes HTTP.

La structure utilisée (cf. figure 12.5 on the facing page) est dite Type Length Value (TLV) ou Type Longueur Valeur. Chaque champ contient au moins ces deux informations :

- Type indiquant la nature de l'option ;
- Longueur indiquant la taille des données en octets. Si ce dernier n'est pas nul, les données vont se trouver après.

CoAP complique un peu la chose en optant pour un codage différentiel de la valeur de l'option. Ainsi, si l'on doit envoyer une option de type 5 puis deux de type 6, le codage contiendra $\Delta T = 5$, $\Delta T = 1$, $\Delta T = 0$.

Mais comme le champ ΔT ne fait que 4 bits, on ne peut pas aller bien loin pour coder ces valeurs. Un mécanisme d'échappement est mis en place pour les différences supérieures à 13. Dans ce cas, la valeur 13 est mise dans le champ T et l'octet suivant code la différence moins 13.

Par exemple, si l'on doit coder deux options de valeurs 5 et 20, la différence est de 15. La première option est codée normalement avec le ΔT à 5. Pour la seconde option, le ΔT est mis à 13 et l'octet suivant prendra la valeur 2.

Notez que la valeur 14 mise dans le champ ΔT indique que la différence nécessite deux octets pour être codée. Les principales options utilisés dans CoAP se retrouvent listés dans le tableau 12.3 on page 159. Celles apparaissant sur fond bleu, seront traitées plus en détail dans cet ouvrage.

Pour la longueur on retrouve le même principe : les longueurs inférieures à 13 sont codées directement ; si elles sont supérieures ou égale à 13, la valeur moins 13 est codée dans un octet supplémentaire. Une valeur de 14 indique que deux octets sont utilisés pour coder la longueur moins 269.

La figure 12.5 on the facing page illustre le codage d'une option dans l'en-tête d'un message CoAP.

Il se peut qu'il y ait des données après les options. Dans ce cas, un séparateur avec la valeur 0xFF est inséré. Il ne peut pas être confondu avec le codage d'une option puisque les champs ΔT et



FIGURE 12.5 – Format des options

Longueur n'évoluent qu'entre 0 et 14.

S'il n'y a pas de données à transmettre (par exemple dans le cas d'une requête GET), le message CoAP se termine après les options.

12.2.5 Options CoAP

Le premier bit servant à coder le type décrit, quand il est positionné à 1, si ce type doit être connu du récepteur (critique). Dans ce cas, si un destinataire reçoit une option de ce type est qu'il ne la connaît pas, il doit produire un message d'erreur. Dans le cas contraire cette option est ignorée du récepteur qui poursuit le traitement des options suivantes. Ainsi les options paires sont facultatives et les impaires critiques.

12.2.6 Représentation des URI

On comprend mieux la signification de certaines options données dans le tableau précédent quand la syntaxe d'un URI est connue. (voir [RFC 2396](#)) . On a déjà vu que la syntaxe générale est :

```
<schema>:<schema-specific-part>
```

où *schema* va définir le schéma de notation. De manière générale, le schéma va indiquer comment est structuré la suite de l'URI. Quand l'URI est aussi un localisateur (donc un URL), le schéma fait référence au protocole qui pourra être utilisé pour retrouver la ressource comme http ou https voire coap. Après le schéma, on trouve deux zones, l'autorité qui va indiquer qui est responsable de nommer les ressources. Dans le cas d'un URL, il peut s'écrire de la manière suivante :

```
<schema>://userinfo@host:@port@/path?query
```

où les champs en italique *userinfo@* et *:port* sont facultatifs. Ils contiennent respectivement le nom de l'utilisateur et le numéro de port sur lequel tourne le service.

path va être composé d'une série de segments séparés par des caractères/ qui identifient la ressource sur le serveur. L'URI peut se terminer par des questions, c'est-à-dire une chaîne de caractères qui sera interprétée par le serveur précédemment désigné. Une question, c'est-à-dire des paramètres fournis pour construire la ressource, peut contenir plusieurs parties séparées par le caractère .

Ceci peut être vérifié par le programme de désassemblage. L'URI :

```
coap://192.168.1.52/capteur1/temperature?max_value&date=20131206
```

utilise le schéma de nommage de coap. Le serveur est 192.168.1.52, le chemin (*path*) est composé de deux segments, suivi par deux questions. Le serveur reçoit la requête suivante :

```
Received packet of size 50
40 01 BE BF|B8 63 61 70 74 - 65 75 72 31|0B 74 65 @....capt eur1.te
6D 70 65 72 61 74 75 72 65 -|49 6D 61 78 5F 76 61 mperature Imax_va
6C 75 65|0D 00 64 61 74 65 - 3D 32 30 31 33 31 32 lue..date =201312
30 36 06
ver:1 Type = 0 (CON) Token Length = 0 code 1 (GET) Msg id = BEBF
Option = 11 (+11) length = 8
```



Valeur	Nom	Type	Nature	répété	Commentaire
0	Reservé				
1	If-Match	opaque	critique	oui	Utilisé pour indiquer à un serveur de ne pas effectuer la requête que sous certaines conditions.
3	Uri-Host	string	critique		Contient le nom du serveur d'une URI (nom, adresse IPv4 ou IPv6). Généralement, il n'est pas nécessaire de le préciser puisque les messages CoAP sont envoyés à cette adresse.
4	ETag	opaque	facultative	oui	Utilisé pour gérer la mise en cache des ressources
5	If-None-Match	vide	critique		Utilisé pour indiquer à un serveur de ne pas effectuer la requête que sous certaines conditions.
6	Observe	entier	facultative		Permet à un serveur d'envoyer une requête aux changements d'état d'une ressource. Dans la réponse la valeur doit toujours augmenter.
7	Uri-Port	entier	critique		Contient le numéro du port UDP sur lequel CoAP est lancé. Généralement ce champ n'est pas nécessaire vu que le serveur CoAP attend déjà des messages sur ce port.
8	Location-Path	string	facultative	oui	Utilisé en réponse à une requête POST pour indiquer un segment du chemin de la ressource.
11	Uri-Path	string	critique	oui	Contient un ou plusieurs segments de l'URI
12	Content-Format	entier	facultative		Définit le format dans lequel sont codées des données
14	Max-Age	entier	facultative		Durée pendant laquelle la ressource peut être mise en cache.
15	Uri-Query	string	critique	oui	Contient les segments d'interrogation que l'on retrouve dans les URIs.
17	Accept	entier	critique		Indique les formats que le client peut accepter.
20	Location-Query	string	facultative	oui	Utilisé en réponse à une requête POST pour indiquer le chemin de la ressource.
35	Proxy-Uri	string	critique		Contient une URI qui doit être prise en compte par le proxy.
39	Proxy-Scheme	string	critique		Indique le protocole.
60	Size1	entier	facultative		Indique la taille de la ressource.
258	No-Response	entier	facultative		Limite les notifications REST

TABLE 12.3 – Types du protocole CoAP

```

Uri-Path capteur1
Option = 11 (+0) length = 11
Uri-Path temperature

```

```
Option = 15 (+4) length = 9
Uri-Query max_value
Option = 15 (+0) length = 13
Uri-Query date=20131206
```

Le listing précédent montre ce que reçoit le serveur. L'URI n'est pas complète, car la partie qui a servi à le localiser n'est pas indispensable. Seules les parties "chemin" et "question" sont indiquées. Le schéma coap : n'est pas précisé ; de même que Uri-Host et Uri-Port car le serveur connaît son adresse IP et le numéro de port sur lequel s'exécute le serveur CoAP.

Ils pourraient être utiles en cas de virtualisation du serveur, c'est-à-dire si plusieurs instances de CoAP tournaient, soit à des noms différents, soit sur des numéros de port différents. Si cette possibilité existe, pour l'instant, la faible capacité des ressources ne pousse pas vers une virtualisation.

Si on reprend la partie optionnelle, la première option qui commence par l'octet 0xB8. 0xb (=11) indique qu'il s'agit d'une option **Uri-path** (comme c'est la première option, le delta se confond avec la valeur de l'option) et de longueur 8 octets qui correspondent à la valeur capteur1. La seconde option débute par 0x0B. Le delta est nul. On reste sur une option Uri-path de longueur de 11 octets. La troisième option s'ouvre avec l'octet 0x49. L'incrément étant de 4, le numéro de l'option passe à 15 soit **Uri-query** avec une valeur sur 9 octets. L'option suivante démarre par 0x0D. On reste sur une option Uri-query mais la longueur 0xD informe que l'octet suivant contient la longueur. La valeur vaut étrangement 0x00 car elle est diminuée de 13 pour respecter le codage défini par CoAP. la longueur est donc de 13 octets.

Questions sur les URI

Soit le message CoAP suivant :

```
40020001b474656d700573656e7331436d6178ff32332e30
```

Question 12.2.1: Code ?

Que représente ce message ?

- Une requête GET
- Une requête POST
- Une requête PUT
- Une requête DELETE
- Une notification positive

Question 12.2.2: Token or not Token ?

Quelle est la valeur du champ token ?

- Vide
- 0xb4
- 0xb474
- 0xb47465
- 0xb474656d

Question 12.2.3: chemin d'URI

Quel élément de l'URI contient ce message ?

- Aucun
- /temp
- /temp/sens1 et /max
- /temp/sens1/max
- /temp/sens1?max

12.2.7 Représentation des données

Pour que le client puisse interpréter les données, il faut qu'il puisse comprendre comment elles sont représentées. Cela peut dépendre de la police de caractères. Ainsi, une lettre accentuée ne sera pas représentée de la même manière suivant le type de code. Il en va de même pour la représentation. Le plus simple consiste à envoyer en ASCII la valeur demandée, par exemple la chaîne de caractères 18 indique 18 degrés. Il faut donc indiquer le type de codage/sérialisation utilisé pour décrire le contenu de la ressource. Là où HTTP utiliserait un nom, c'est-à-dire une chaîne de caractères, CoAP va utiliser une valeur numérique.

Le tableau précédent donne un extrait des valeurs utilisées pour représenter les formats⁴.

Deux options CoAP utilisent ces codes :

- **Content-format** (12) indiquant comment la ressource est codée ;
- **Accept** (17) indiquant dans une requête le format dans lequel la réponse doit être codée.

Il existe beaucoup de valeurs pour le content-format, elles permettent de spécifier de manière très économique le type de ressource et par conséquent le traitement à effectuer.

Chaque protocole utilisant CoAP aura tendance à définir de nouveaux codes. Le tableau 12.4 illustre ce phénomène pour **SenML**. La valeur va servir à la fois pour indiquer la structure de données et le format de codage.

Valeur	Type
0	text/plain ; charset=utf-8
40	application/link-format
41	application/xml
42	application/octet-stream
47	application/exi
50	application/json
60	application/cbor
110	application/senml+json
112	application/senml+cbor
11542	application/vnd.oma.lwm2m+tlv
11543	application/vnd.oma.lwm2m+json

TABLE 12.4 – Type des données

4. La liste complète peut être trouvée sur le site de l'IANA <https://www.iana.org/assignments/core-parameters/core-parameters.xhtml#content-formats>.

Question 12.2.4: ASCII

Vous avez la ressource suivante :

```
temperature = 20C
```

Quelle valeur l'option CoAP Content-type utiliser pour une réponse en CoAP ?

- text/plain
- 0
- 50

Question 12.2.5: SenML

Vous voulez recevoir une ressource dans le format SenML; CBOR. Quelle valeur doit transporter l'option Accept dans la requête ?

Question 12.2.6: Erreur

Quel code d'erreur retourne le serveur s'il ne peut pas envoyer une réponse dans ce format ?

Aidez vous du [RFC 7252](#).

- 4.04 (Not Found)
- 4.02 (Bad Option)
- 4.06 (Not Acceptable)
- 5.01 (Not Implemented))

12.3 Observe

Avec l'architecture REST, le serveur répond toujours aux requêtes d'un client. Si l'on veut suivre l'évolution d'une ressource, le client doit demander périodiquement la valeur ; le serveur ne gardant pas d'état sur les requêtes passées. Cela n'est pas toujours compatible avec les contraintes énergétiques des capteurs. Supposons que l'on ait une alarme d'incendie qui doit informer quand le taux de fumée atteint un certain seuil. Il existe deux possibilités :

- l'alarme est un client REST et envoie un POST vers un serveur quand l'alerte est déclenchée. Pour que cela fonctionne, il faut que l'alarme ait été préalablement configurée avec l'adresse du serveur vers où envoyer ses requêtes POST ;
- l'alarme est un serveur REST qui possède une ressource donnant le taux de fumée. Elle n'a pas besoin d'être configurée. Les clients l'interrogent et elle répond à leur adresse. En revanche, si l'on veut déterminer quand un seuil est atteint, il faut continuellement interroger la ressource à une fréquence élevée si l'on ne veut pas rater une information.

L'option **Observe**, définie dans le [RFC 7641](#), permet à un serveur d'envoyer périodiquement la valeur d'une ressource vers le client qui en a fait la demande. La période d'émission (ou les règles d'émission comme envoyer quand un seuil est atteint) est définie par le comportement du serveur.

La figure 12.6 on the facing page illustre ce comportement. Le client envoie une requête GET en positionnant l'option Observe avec la valeur 0, mais dans valeur. Si le client accepte cette option, il va répondre en la positionnant dans ses réponses. Elle doit dans ce cas comporter une valeur qui ne pourra que croître de réponse en réponse. Cela est utile pour permettre au client de détecter

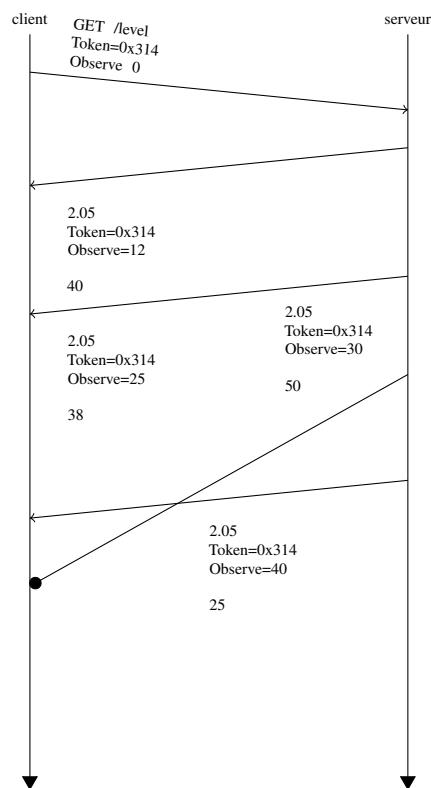


FIGURE 12.6 – Envoi périodique grâce à l'option Observe

un déséquement des réponses. Ainsi dans l'exemple, l'Observe estampillé 30 arrive après celui estampillé 40 et sera rejeté par le client.

On notera également que le champ **Token** doit être présent pour faire le lien entre la requête et les réponses.

Il faut pouvoir aussi arrêter un Observe. Il existe plusieurs cas de figure :

- le client veut stopper un Observe en cours. Il refait la même requête mais en mettant la valeur 1 dans l'option Observe.
- le client redémarre, il va perdre peut perdre son contexte concernant l'Observe, mais continuer à recevoir périodiquement des requêtes provenant du serveur. Le client ne reconnaissant pas le Token, émet un message **ReSeT**. Le serveur annule l'émission périodique vers le client.
- le client est inaccessible, il ne va pas pouvoir annuler la transmission. En règle générale, les réponses avec l'option Observe sont transportées dans des messages **NON** confirmables. Le serveur peut de temps en temps envoyer la réponse dans un message **CON**firmeable. S'il ne reçoit pas d'acquittement du client, il en déduit qu'il a disparu et arrête d'envoyer des réponses périodiques.

13. Experimentons CoAP

Les programmes se trouvent dans le répertoire pycom pour l'Objet et plido-tp4 pour le serveur.

Les programmes pour l'Objet pourront tourner indifféremment dans une fenêtre de votre ordinateur ou sur votre Pycom. Dans la suite, nous supposerons qu'il s'agit d'une deuxième fenêtre de terminal sur votre ordinateur mais libre à vous de le lancer sur votre Pycom en Wi-Fi (voire en LoRaWAN) pour plus de réalisme. L'utilisation du réseau Sigfox est plus problématique vu que les requêtes provenant de votre Pycom sont limitées à 12 caractères et que les réponses ne sont qu'au nombre de 4 par jour et limitées à 8 caractères.

Pour nous concentrer sur le fonctionnement de CoAP, nous allons tout d'abord expérimenter en local sur notre ordinateur. Nous verrons par la suite comment utiliser le programme generic_relay.py pour bénéficier des réseaux LPWAN.

Attention, le serveur ne fonctionne pas directement sous Windows. Vous devez impérativement le faire tourner dans la machine virtuelle.

13.1 Mise en œuvre du client/serveur

Nous allons mettre en œuvre le protocole CoAP entre deux processus sur votre machine puis, si vous le voulez, vous pourrez le tester sur un LoPy. Côté Objet, nous allons utiliser une mise en œuvre simple mais compacte du protocole pour comprendre son fonctionnement. À l'autre extrémité, nous utiliserons la mise en œuvre aiocoap qui est très complète mais beaucoup plus complexe et demandant plus de ressources.

Youtube



13.1.1 aiocoap

Comme son nom l'indique, aiocoap met en œuvre CoAP, avec les modules asynchronous Input/Output, permettant un fort degré de parallélisme. Le programme suivant (`coap_basic_server1.py`) donne un exemple d'un serveur simple gérant une seule ressource time :

Listing 13.1 – coap_basic_server1.py

```

1 #!/usr/bin/env python3

3 # This file is part of the Python aiocoap library project.
#
5 # Copyright (c) 2012-2014 Maciej Wasilak <http://sixpinetrees.blogspot.com/>,
#                   2013-2014 Christian Amsüss <c.amsuess@energyharvesting.at>
7 #
# aiocoap is free software, this file is published under the MIT license as
9 # described in the accompanying LICENSE file.

11 """This is a usage example of aiocoap that demonstrates how to implement a
13 simple server. See the "Usage Examples" section in the aiocoap documentation
15 for some more information."""
16

17 import datetime
18 import logging
19 import socket

20 import asyncio
21
22 import aiocoap.resource as resource
23 import aiocoap

24
25 class TimeResource(resource.Resource):
26
27     @asyncio.coroutine
28     def render_get(self, request):
29         await asyncio.sleep(5)
30
31         payload = datetime.datetime.now().strftime("%Y-%m-%d %H:%M").encode('ascii')
32
33         return aiocoap.Message(payload=payload)

34 # logging setup

35 logging.basicConfig(level=logging.INFO)
36 logging.getLogger("coap-server").setLevel(logging.DEBUG)
37
38 def main():
39     # Resource tree creation
40     with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
41         s.connect(("8.8.8.8", 80)) # connect outside to get local IP address
42         ip_addr = s.getsockname()[0]
43
44         port = 5683
45         print ("server running on", ip_addr, "at port", port)
46
47         root = resource.Site()
48
49         root.add_resource(['time'], TimeResource())
50
51         asyncio.Task(
52             aiocoap.Context.create_server_context(root,
53                                                 bind=(ip_addr, port)))
54
55         asyncio.get_event_loop().run_forever()

```

```
57 if __name__ == "__main__":
    main()
```

- Lignes 21 et 22, l'utilisation de *aicoap* se traduit par l'importation des modules qui se trouvent dans le répertoire *aoicoap*.
- Dans la fonction *main* le programme :
 - cherche son adresse IP fixe (lignes 40 à 42) ;
 - fixe à la valeur par défaut le numéro de port à la valeur affectée au serveur CoAP (ligne 44)¹ ;
 - ligne 47, La variable *root* contient l'arbre des ressources. À la ligne suivante, la ressource *time* est associée à une classe *TimeResource*.
 - La méthode :


```
asyncio.Task(aiocoap.Context.create_server_context(root, bind=(ip_addr, port)))
```

 permet de lier cet arbre de ressource à l'adresse IP et au numéro de port précédemment défini.
 - Ligne 55, le serveur est ensuite lancé dans une boucle sans fin.

Il est plus intéressant de voir le traitement effectué lorsque la ressource est appelée par le serveur. La classe *TimeResource* dérivant de la classe générique *aiocoap Resource* est utilisée (ligne 24) :

```
class TimeResource(resource.Resource):
```

Pour chaque méthode CoAP, une méthode peut être définie dans cette classe. Dans l'exemple, la méthode *render_get* permet de traiter les requêtes GET.

Pour simuler un temps de traitement, le programme commence par attendre 5 secondes (ligne 27) puis construit la chaîne de caractères contenant la date qu'il va retourner dans un objet *aiocoap Message*.

Ainsi, si tout se passe correctement, la réponse à une requête (Code = 0x01) sera 2.05 (Content). La figure 13.1 on the next page résume cet échange que nous avions vu théoriquement dans le chapitre 12.2.3 on page 154 consacré aux tokens.

13.1.2 côté Objet

Du côté du client, nous allons utiliser une mise en œuvre plus compacte qui nous permettra d'expérimenter le fonctionnement de CoAP en modifiant les valeurs des champs protocolaires.

Listing 13.2 – coap_empty_msg.py

```
1 import CoAP
2 import socket
3
4 SERVER = "192.168.1.XX" # change to your server's IP address
5 PORT   = 5683
6
7 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

1. L'utilisation d'une adresse IP et non du joker 0.0.0.0 permet de faire tourner le serveur dans un environnement Windows et MAC

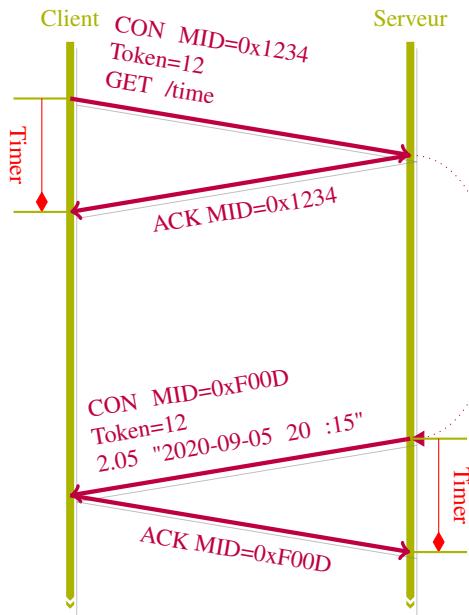


FIGURE 13.1 – Récupération de la date

```

9 coap = CoAP.Message()
10 coap.new_header()
11 coap.dump(hexa=True)

13 s.settimeout(10)
14 s.sendto(coap.to_byte(), (SERVER, 5683))

15
16 resp,addr = s.recvfrom(2000)
17 answer = CoAP.Message(resp)
18 answer.dump(hexa=True)

```

Le programme `coap_empty_msg.py` est beaucoup plus simple. Dans un premier temps, vous devez remplacer l'adresse IP par celle fournie par votre serveur CoAP. Le programme crée une socket UDP au travers de laquelle l'échange avec le serveur sera effectué. La première action consiste à créer un message CoAPCoAP (ligne 9) et à la ligne suivante créer un en-tête obligatoire avec les paramètres par défaut avec la fonction CoAP.

Le programme affiche le message avec la fonction `CoAPdump`(ligne 11) et l'envoie sur la socket. La ligne 13 permet de limiter l'attente de la réponse à 10 secondes. Cette réponse est attendue ligne 16, transformée en message CoAP ligne 17 et affichée.

Lancez une capture Wireshark pour voir le trafic passant sur le port de CoAP (`udp.port==5683` dans la fenêtre de filtrage).

Lancez maintenant le programme client.

```

b      40000001
CON 0 x0001 EMPTY
b      70000001

```

```
RST 0 x0001 EMPTY
```

Les messages suivants ont circule sur le réseau.

```
18:38:30.381449 IP 192.168.1.26.50883 > 192.168.1.26.5683: UDP, length 4
 0x0000: 4500 0020 221f 0000 4011 0000 c0a8 011a E..."@.....
 0x0010: c0a8 011a c6c3 1633 000c 83a2 4000 0001 .....3....@...
18:38:30.382107 IP 192.168.1.26.5683 > 192.168.1.26.50883: UDP, length 4
 0x0000: 4500 0020 efbb 0000 4011 0000 c0a8 011a E.....@.....
 0x0010: c0a8 011a 1633 c6c3 000c 83a2 7000 0001 .....3.....p...
```

On retrouve dans le contenu des messages UDP, le message CoAP donné par l'application. Si l'on prend le deuxième message, il commence par 0x70; ce qui correspond en binaire à 0b01_11_0000, soit version = 1, type = 3 et longueur du token = 0. L'octet suivant donne le code 0 (**Empty**) et les deux derniers octets contiennent le message ID.

Le serveur ne sachant pas quoi faire de la requête, la rejette en envoyant un message ReSeT pour essayer d'arrêter le code sur le client qui envoie ce genre de requête.

13.2 GET /time

Nous laissons tourner le serveur CoAP et nous allons construire la requête CoAP du client pour qu'il demande la ressource `/time`².

Listing 13.3 – coap_get_time.py

```
import CoAP
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

coap = CoAP.Message()
coap.new_header()
coap.new_header(code=CoAP.GET, mid=0xF001)
coap.add_option(CoAP.Uri_path, "time")
coap.dump()

s.settimeout(10)
s.sendto(coap.to_byte(), ("192.168.1.77", 5683))

resp, addr = s.recvfrom(2000)
answer = CoAP.Message(resp)
answer.dump()

s.settimeout(10)
resp, addr = s.recvfrom(2000)
answer = CoAP.Message(resp)
answer.dump()
```

la méthode `new_header` précise le code, ici GET et on ajoute l'élément d'URI time en option (ligne 10) grâce à la méthode `add_option`. Côté client, on obtient le résultat suivant :

```
> python3 coap_get_time1.py
False
b'40010001b474696d65'
CON 0x0001 GET
```

2. N'oubliez pas de mettre la bonne adresse IP du SERVER dans ce programme et dans les suivants

1208 169459.0845s 192.168.1.79	192.168.1.79	CoAP	51 CON, MID:1, GET, /time
1209 169459.1876s 192.168.1.79	192.168.1.79	CoAP	46 ACK, MID:1, Empty Message
1210 169464.0933s 192.168.1.79	192.168.1.79	CoAP	63 CON, MID:19224, 2.05 Content
1211 169464.0933s 192.168.1.79	192.168.1.79	ICMP	91 Destination unreachable (Port unreachable)

FIGURE 13.2 – Échange incomplet

```
> Uri-path : b'time'
b'60000001'
ACK 0x0001 EMPTY
```

La requête CoAP commence par le mot 40010001, indiquant un message **CON**firmeable, sans Token, un code GET et un MID de 0x0001, suivi de l'option **Uri-Path**.

La réponse est un **ACK** avec la même valeur de *Message ID* et le code est vide (0.00).

On n'obtient pas la réponse au GET, juste un acquittement. Pourtant, les log du serveur et l'analyse du réseau montrent bien que le serveur à répondu.

```
DEBUG:coap-server:Incoming message <aiocoap.Message at 0x7f3d997ace80: Type.CON GET (MID 1, empty token) remote
<UDP6EndpointAddress 192.168.1.79:52495 (locally 192.168.1.79%lo)>, 1 option(s)
DEBUG:coap-server:New unique message received
DEBUG:coap-server:Sending empty ACK: Response took too long to prepare
DEBUG:coap-server:Sending message <aiocoap.Message at 0x7f3d99742748: Type.ACK EMPTY (MID 1, empty token) remote
<UDP6EndpointAddress 192.168.1.79:52495 (locally 192.168.1.79%lo)>>
```

Les trames qui ont circulé sur le réseau sont représentées figure 13.2. Comme nous avons ajouté un délai de 5 secondes avant de répondre sur le serveur CoAP dans la méthode `render_get`, le serveur acquitte la requête et cherche 5 secondes plus tard à envoyer une nouvelle requête confirmée. Mais le client, ayant fermé sa socket, ne peut plus la recevoir et retourne une erreur ICMP.

La solution est d'attendre un second message et de le décoder comme le montre le listing suivant qui donne le code du programme `coap_get_time2.py` dans lequel vous n'aurez pas oublié de changer l'adresse IP du serveur.

Listing 13.4 – `coap_get_time2.py`

```
import CoAP
import socket

SERVER = "192.168.1.XX" # change to your server's IP address
PORT = 5683

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

coap = CoAP.Message()
coap.new_header(code=CoAP.GET)
coap.add_option(CoAP.Uri_path, "time")
coap.dump()

s.sendto(coap.to_byte(), (SERVER, PORT))

s.settimeout(10)
resp, addr = s.recvfrom(2000)
answer = CoAP.Message(resp)
answer.dump()

s.settimeout(10)
```

1299	170334.4503:	192.168.1.79	192.168.1.79	CoAP	51 CON, MID:1, GET, /time
1300	170334.5545:	192.168.1.79	192.168.1.79	CoAP	46 ACK, MID:1, Empty Message
1307	170339.4599:	192.168.1.79	192.168.1.79	CoAP	63 CON, MID:9992, 2.05 Content
1308	170342.2983:	192.168.1.79	192.168.1.79	CoAP	63 CON, MID:9992, 2.05 Content
1309	170342.2983:	192.168.1.79	192.168.1.79	ICMP	91 Destination unreachable (Port unreachable)

FIGURE 13.3 – Échange encore incomplet

```

22 resp, addr = s.recvfrom(2000)
23 answer = CoAP.Message(resp)
24 answer.dump()

```

Nous pouvons laisser éclater notre joie car nous avons la réponse :

```

> python3 coap_get_time2.py
False
CON 0x0001 GET
> Uri-path : b'time'
ACK 0x0001 EMPTY
CON 0x2708 2.05
---CONTENT
hex: b'323032312d30332d33302031343a3537',
txt: b'2021-03-30 14:57'

```

Mais notre joie est de courte durée car si on regarde plus attentivement la figure 13.3 le trafic sur le réseau, on voit que la réponse a été émise deux fois et que l'on retrouve ensuite une erreur ICMP. Cela est dû au fait que l'on n'acquitte pas le message venant du serveur. Le croyant perdu, il le retransmet et tombe sur une socket inexistante.

Pour bien faire, nous devons fournir un acquittement au serveur avec ce code de client avec le programme `coap_get_time3.py`.

Listing 13.5 – `coap_get_time3.py`

```

1 import CoAP
2 import socket
3
4 SERVER = "192.168.1.XX" # change to your server's IP address
5 PORT = 5683
6
7 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
8
9 coap = CoAP.Message()
10 coap.new_header(code=CoAP.GET)
11 coap.add_option(CoAP.Uri_path, "time")
12 coap.dump()
13
14 s.sendto(coap.to_byte(), (SERVER, PORT))
15
16 s.settimeout(10)
17 resp,addr = s.recvfrom(2000)
18 answer = CoAP.Message(resp)
19 answer.dump()
20
21 s.settimeout(10)
22 resp,addr = s.recvfrom(2000)

```

1409 171332.3768f 192.168.1.79	192.168.1.79	CoAP	51 CON, MID:1, GET, /time
1410 171332.4796f 192.168.1.79	192.168.1.79	CoAP	46 ACK, MID:1, Empty Message
1411 171337.3847f 192.168.1.79	192.168.1.79	CoAP	63 CON, MID:9993, 2.05 Content
1412 171337.3849f 192.168.1.79	192.168.1.79	CoAP	46 ACK, MID:9993, Empty Message

FIGURE 13.4 – Échange dans les règles de l’art

```

23 answer = CoAP.Message(resp)
24 answer.dump()
25
26 mid = answer.get_mid()
27 ack = CoAP.Message()
28 ack.new_header(mid=mid, type=CoAP.ACK)
29 ack.dump()
30 s.sendto(ack.to_byte(), (SERVER, PORT))

```

Ici, l’exécution est parfaite :

```

> python3 coap_get_time3.py
False
CON 0x0001 GET
> Uri-path : b'time'
ACK 0x0001 EMPTY
CON 0x2709 2.05
---CONTENT
hex: b'323032312d30332d33302031353a3133',
txt: b'2021-03-30 15:13'
ACK 0x2709 EMPTY

```

si vous n’avez pas oublié de changer l’adresse IP du serveur, et l’utilisation du réseau est optimale (cf figure 13.4).

Enfin, on peut ajouter un token pour lier les deux transactions. Ici, il n’y a pas d’ambiguïté car nous ne demandons qu’une seule ressource. Mais si nous demandions plusieurs ressources simultanément, il faudrait pouvoir associer la réponse à la requête. Le programme `coap_get_time4.py` ajoute lors de la création de l’en-tête un champ token.

Listing 13.6 – `coap_get_time4.py`

```

coap = CoAP.Message()
10 coap.new_header(code=CoAP.GET, token=0x12345)
11 coap.add_option(CoAP.Uri_path, "time")
12 coap.dump()

```

L’échange est le même mais le token est répété dans la réponse.

```

> python3 coap_get_time4.py
False
CON 0x0001 GET T=012345
> Uri-path : b'time'
ACK 0x0001 EMPTY
CON 0x270A 2.05 T=012345
---CONTENT
hex: b'323032312d30332d33302031353a3232',
txt: b'2021-03-30 15:22'
ACK 0x270A EMPTY

```

Question 13.2.1: NON

Que se passe-t-il si vous utilisez une requête non confirmable pour demander la ressource /time (mettre l'argument type=CoAP.NON dans la construction de l'en-tête obligatoire).

- Ça plante le serveur.
- Le serveur répond par une requête Confirmable.
- Le serveur retourne un RST car nous n'avons pas programmé ce cas.
- Le serveur répond par une requête Non Confirmable.
- On passe en heure d'hiver.

Question 13.2.2: Réponse immédiate et CON

Modifiez le programme du serveur pour supprimer le délai de 5 secondes avant une réponse.

Que se passe-t-il quand le client envoie une requête CONfirmable ?

- Le serveur retourne la réponse dans l'acquittement.
- Le serveur retourne une requête NON confirmable.
- Le serveur attend quand même quelques secondes pour ne pas saturer le réseau.
- Le serveur enlève le token de la réponse..

Question 13.2.3: Réponse immédiate et NON

Modifiez le programme du serveur pour supprimer le délai de 5 secondes avant une réponse.

Que se passe-t-il quand le client envoie une requête NON confirmable ?

- Le serveur retourne la réponse dans l'acquittement.
- Le serveur retourne une requête NON confirmable.
- Le serveur attend quand même quelques secondes pour ne pas saturer le réseau.
- Le serveur enlève le token de la réponse..

13.3 POST

13.3.1 Ressource codée en ASCII

Nous allons voir le traitement d'une requête POST avec *aiocoap*. Nous aurions pu ajouter ce traitement à la classe TimeResource vue précédemment, mais nous avons préféré ajouter une nouvelle ressource pour le traitement des températures.

Les lignes suivantes ont été ajoutées pour traiter le POST. Notez le nom de la méthode render_post.

Listing 13.7 – coap_basic_server2.py

```

34 class TemperatureResource(resource.Resource):
35     async def render_post(self, request):
36         print ("-"*20)
37         print ("payload:", binascii.hexlify(request.payload))
38         resp = aiocoap.Message(code=aiocoap.CHANGED)
39         return resp

```

La réponse à la requête est le code 2.04 (aiocoap.CHANGED) qui indique que la ressource a été modifiée.

On lui associe une nouvelle classe dans l'arbre des ressources :

```
root.add_resource(['temp'], TemperatureResource())
```

Le tout forme le programme `coap_basic_server2.py`.

Coté client, le principe est le même que pour le GET. Dans le programme `coap_post_temp1.py`, la méthode `add_payload` permet d'ajouter du contenu à la requête POST. Le programme émet sa requête et affiche le résultat. Nous allons également simplifier la gestion des réponses en utilisant la fonction `send_ack` du module CoAP. Cette fonction prend en argument une socket, un tuple de destination et le message CoAP à envoyer. Elle va le retransmettre au maximum 4 fois tant qu'elle n'a pas reçu l'acquittement correspondant.

Listing 13.8 – `coap_post_temp1.py`

```

1 import CoAP
2 import socket
3 import time
4
5 SERVER = "192.168.1.XX" # change to your server's IP address
6 PORT = 5683
7 destination = (SERVER, PORT)
8
9 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
10
11 coap = CoAP.Message()
12 coap.new_header(code=CoAP.POST)
13 coap.add_option(CoAP.Uri_path, "temp")
14 coap.add_payload("23.5")
15 coap.dump()
16
17 answer = CoAP.send_ack(s, destination, coap)
18 answer.dump()
```

La figure 12.3 on page 155 a montré un échange avec perte. Elles peuvent être simulées en arrêtant le programme serveur sur votre ordinateur.

Lancez le serveur `coap_basic_server2.py` et tapez ctrl-Z pour stopper son exécution, tout en laissant ouverte la socket.

```
> python3 ./coap_basic_server2.py
server running on 192.168.1.79 at port 5683 ^Z
Job 2,    python3 ./coap_basic_server2.py has stopped
>
```

Lancez le client `coap_port_temp1.py` (en n'oubliant pas de changer l'adresse IP du serveur).

```
> python3.9 coap_post_temp1.py
False
CON 0x0001 POST
> Uri-path : b'temp'
---CONTENT
hex: b'32332e35'
txt: b'23.5'
timeout 2 1
timeout 4 2
```

18	93.56226723:	192.168.1.79	192.168.1.79	CoAP	56 CON, MID:1, POST, /temp
19	105.5660928:	192.168.1.79	192.168.1.79	CoAP	56 CON, MID:1, POST, /temp
20	119.5771454:	192.168.1.79	192.168.1.79	CoAP	56 CON, MID:1, POST, /temp
27	122.0071288:	192.168.1.79	192.168.1.79	CoAP	46 ACK, MID:1, 2.04 Changed
28	122.0169795:	192.168.1.79	192.168.1.79	CoAP	46 ACK, MID:1, 2.04 Changed
29	122.0170019:	192.168.1.79	192.168.1.79	ICMP	74 Destination unreachable (Port unreachable)
30	122.0272629:	192.168.1.79	192.168.1.79	CoAP	46 ACK, MID:1, 2.04 Changed
31	122.0272841:	192.168.1.79	192.168.1.79	ICMP	74 Destination unreachable (Port unreachable)

FIGURE 13.5 – Trafic lié au POST

On voit que le client ne reçoit pas l’acquittement, déclenche un temporisateur et retransmet le message. La durée du temporisateur double à chaque tentative³.

Réactivez le serveur coap en tapant fg (foreground)

```
> fg
DEBUG:coap-server:Incoming message <aiocoap.Message at 0x7f05d621fcc0: Type.CON POST (MID 1, empty token) remote
<UDP6EndpointAddress 192.168.1.79:37286 (locally 192.168.1.79%lo)>, 1 option(s), 4 byte(s) payload>
DEBUG:coap-server>New unique message received
-----
payload: b'32332e35'
DEBUG:coap-server:Sending message <aiocoap.Message at 0x7f05d622df98: Type.ACK 2.04 Changed (MID 1, empty token)
remote <UDP6EndpointAddress 192.168.1.79:37286 (locally 192.168.1.79%lo)>>
DEBUG:coap-server:Incoming message <aiocoap.Message at 0x7f05d621fcc0: Type.CON POST (MID 1, empty token) remote
<UDP6EndpointAddress 192.168.1.79:37286 (locally 192.168.1.79%lo)>, 1 option(s), 4 byte(s) payload>
INFO:coap-server:Duplicate CON received, sending old response again
DEBUG:coap-server:Sending message <aiocoap.Message at 0x7f05d622df98: Type.ACK 2.04 Changed (MID 1, empty token)
remote <UDP6EndpointAddress 192.168.1.79:37286 (locally 192.168.1.79%lo)>>
DEBUG:coap-server:Socket error received, details: SockExtendedErr(ee_errno=111, ee_origin=2, ee_type=3, ee_code=3,
ee_pad=0, ee_info=0, ee_data=0)
DEBUG:coap-server:Incoming error 111 from <UDP6EndpointAddress 192.168.1.79:37286 (locally 192.168.1.79%lo)>
DEBUG:coap-server:Incoming message <aiocoap.Message at 0x7f05d622deb8: Type.CON POST (MID 1, empty token) remote
<UDP6EndpointAddress 192.168.1.79:37286 (locally 192.168.1.79%lo)>, 1 option(s), 4 byte(s) payload>
INFO:coap-server:Duplicate CON received, sending old response again
DEBUG:coap-server:Sending message <aiocoap.Message at 0x7f05d622df98: Type.ACK 2.04 Changed (MID 1, empty token)
remote <UDP6EndpointAddress 192.168.1.79:37286 (locally 192.168.1.79%lo)>>
DEBUG:coap-server:Socket error received, details: SockExtendedErr(ee_errno=111, ee_origin=2, ee_type=3, ee_code=3,
ee_pad=0, ee_info=0, ee_data=0)
DEBUG:coap-server:Incoming error 111 from <UDP6EndpointAddress 192.168.1.79:37286 (locally 192.168.1.79%lo)>
```

On peut observer plusieurs phénomènes :

- le serveur avait gardé en mémoire les requêtes précédentes mais ne les avait pas traitées car le programme avait été stoppé. En le redémarrant, il va pouvoir les traiter. Il répond donc au client qui peut afficher la notification 2.04 ;
- les autres requêtes sont vues comme des répétitions de la première puisque le champ Message ID est identique. Le serveur se contente de retourner une copie ;
- le client ayant terminé sa transaction a fermé la socket conduisant à l’émission d’un message ICMP qui conduit à l’affichage du message de log d’erreur (Socket error received).

Ceci peut être vérifié sur la capture Wireshark (cf. figure 13.5).

13.3.2 Ressource codée en CBOR

Le programme précédent, est correct mais nous avons utilisé le format ASCII de transfert par défaut (**Content-format = 0**). Il serait préférable de revenir au format JSON ou **CBOR** que nous avions vu précédemment. Pour ce faire, nous devons ajouter dans l’en-tête de la requête l’option Content-format. Comme son code est 12, elle doit être placée après l’option Uri-path (cf. tableau 12.3 on page 159).

Listing 13.9 – coap_post_temp2.py

```
import CoAP
```

3. Le standard recommande d’ajouter un aléa lors de la transmission pour éviter que deux équipements se synchronisent et se perturbent toujours en même temps. Il n’est pas mis en œuvre dans cet exemple.

```

2 import socket
3 import time
4 try:
5     import kpn_senml.cbor_encoder as cbor #pycom
6 except:
7     import cbor2 as cbor # terminal on computer
8
9 SERVER = "192.168.1.26" # change to your server's IP address
10 PORT = 5683
11 destination = (SERVER, PORT)
12
13 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
14
15 coap = CoAP.Message()
16 coap.new_header(code=CoAP.POST)
17 coap.add_option(CoAP.Uri_path, "temp")
18 coap.add_option(CoAP.Content_format, CoAP.Content_format_CBOR)
19 coap.add_payload(cbor.dumps(23.5))
20 coap.dump()
21
22 answer = CoAP.send_ack(s, destination, coap)
23 answer.dump()

```

Notez que ce programme, pprogcoap_post_temp2.py, peut aussi bien s'exécuter sur votre ordinateur ou sur votre LoPy. Dans le premier cas, le module `cbor2` sera utilisé. Sur le LoPy, nous ferons appel au module `kpn_senml`. Dans les deux cas, le contenu est transformé en CBOR grâce à la fonction `dumps`.

Du côté du serveur, la méthode `render_post` doit connaître le format de la ressource. Pour ce faire, elle doit accéder aux options CoAP.

Listing 13.10 – coap_basic_server3.py

```

34         return aiocoap.Message(payload=payload)
35
36 class TemperatureResource(resource.Resource):
37     async def render_post(self, request):
38         print ("-"*20)
39
40         ct = request.opt.content_format or \
41             aiocoap.numbers.media_types_rev['text/plain']
42
43         if ct == aiocoap.numbers.media_types_rev['text/plain']:
44             print ("text:", request.payload)
45         elif ct == aiocoap.numbers.media_types_rev['application/cbor']:
46             print ("cbor:", cbor.loads(request.payload))
47         else:
48             print ("Unknown format")
49             return aiocoap.Message(code=aiocoap.UNSUPPORTED_MEDIA_TYPE)
50         return aiocoap.Message(code=aiocoap.CHANGED)

```

Dans le programme `coap_basic_server3.py`, la variable `ct` contient la valeur de l'option **Content-format** si elle existe. De manière générale, `aiocoap` permet d'accéder aux valeurs de toutes les options CoAP contenues dans la requête. La valeur `None` indique que l'option n'est pas présente dans l'en-tête. Dans ce cas, la variable `ct` contiendra la valeur par défaut indiquant un texte en

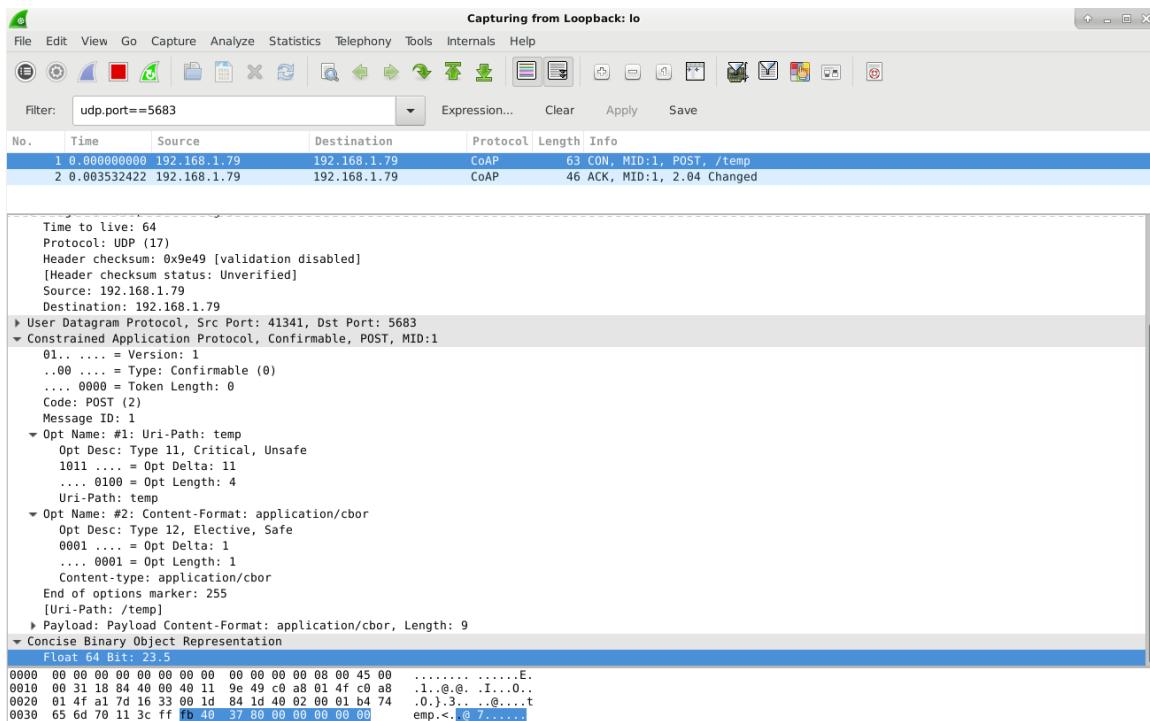


FIGURE 13.6 – Trafic complet lié au POST

ASCII.

Suivant le format de la donnée, le programme affichera le texte ou le CBOR transformé en chaîne de caractères avec la fonction `loads`. Si le format n'est pas connu du programme, un code d'erreur est retourné au client.

La figure 13.6 montre l'échange lié au POST et détaille contenu de la requête.

Question 13.3.1: Réponse

Que reçoit-on en réponse à la requête POST ?

- un message ACK
- le statut 2.00.
- le statut 2.04 CHANGED.
- rien.

Question 13.3.2: Réponse

Modifiez le programme client pour indiquer un content-format JSON. Quelle notification obtenez-vous ?

- un message RST
- le statut 4.04 NOT FOUND.
- le statut 2.15.
- le statut 5.00.

13.3.3 No Response

Que l'on utilise le type CONfirmable ou NON confirmable, le serveur va envoyer une notification :

- 2.xx quand tout se passe bien, et
- 4.xx ou 5.xx quand le client ou le serveur a commis une erreur.

Si un capteur veut utiliser CoAP sur un réseau LPWAN, à chaque POST qu'il va envoyer sur la voie montante (uplink), il va récupérer un acquittement dans la voie descendante (downlink). Or, on a vu que l'on devait ménager cette dernière qui pouvait être sujette à saturation ou à facturation.

L'option **No response** définie dans le RFC 7967 permet à un client d'informer le serveur qu'il ne souhaite pas recevoir de notifications lors d'un POST ou d'un PUT. La valeur de l'option est 258 et elle est suivie d'un octet contenant un bitmap qui va indiquer quel type de notification va être émis :

- si le deuxième bit à partir de la droite est mis à 1, le client ne veut pas recevoir les notifications de type 2.xx ;
- si le quatrième bit à partir de la droite est mis à 1, le client ne veut pas recevoir les notifications de type 4.xx ;
- si le cinquième bit à partir de la droite est mis à 1, le client ne veut pas recevoir les notifications de type 5.xx.

Par exemple, en mettant la valeur 0x02 dans cette option, le client ne reçoit pas les acquittements positifs mais uniquement les erreurs 4.xx et 5.xx. En mettant la valeur 0b00011010 (0x1a, 26), le serveur sera complètement silencieux.

Le programme `coap_post_temp4.py` ajoute cette option et le type du message CoAP a été fixé à NON confirmable. Cependant, le chemin d'URI n'est pas connu du serveur. On voit que le serveur retourne un message d'erreur. Si en revanche le POST se déroule correctement, seul le client émet une donnée et le serveur reste silencieux.

Listing 13.11 – `coap_post_temp4.py`

```

15 coap = CoAP.Message()
16 coap.new_header(type=CoAP.NON, code=CoAP.POST)
17 coap.add_option(CoAP.Uri_path, "temp")
18 coap.add_option(CoAP.Content_format, CoAP.Content_format_CBOR)
19 coap.add_option(CoAP.No_Response, 0b00000010)
20 coap.add_payload(cbor.dumps(23.5))
21 coap.dump()
```

13.4 Chaîne complète de remonté de mesures

Voilà ! nous pouvons mettre les différents concepts ensemble pour construire une chaîne complète de remontée d'informations sur la température, l'humidité et la pression.

Côté capteur, nous allons :

- si le BME280 est présent, récupérer directement les données. Sinon, nous allons utiliser les mesures virtuelles. Nous allons également définir le protocole réseau : WiFi, Sigfox, LoRaWAN ;
- si nous utilisons un réseau LoRaWAN, nous devons mettre en place un programme relais pour transmettre les données vers le serveur CoAP ;



Le serveur CoAP doit pouvoir envoyer les données au serveur Beebotte pour affichage.

Le message CoAP va être configuré de la manière suivante :

- type = NON pour éviter les acquittements des messages CoAP ;
- l’option No Response à 2 pour éviter les notifications REST positives. On garde sur la voie descendante les notifications d’erreurs pour permettre au capteur d’arrêter de transmettre ou d’augmenter sa période d’émission si le serveur n’est pas capable de traiter les données. Cela permettra de limiter l’usage du spectre et de préserver l’énergie des capteurs ;
- 3 chemins d’URI pour les 3 valeurs mesurées ;
- le codage en CBOR des séries temporelles.

Il reste un point à traiter car avec la limitation à 12 octets des trames **Sigfox**, il est impossible de transmettre des données si on utilise CoAP. En effet, l’en-tête CoAP va contenir :

- 4 octets pour l’en-tête obligatoire ;
- 0 octets de Token ;
- 2 octets au minimum pour l’option URI-path si on réduit à un caractère le chemin ; par exemple T, H, P pour représenter la température, l’humidité et la pression ;
- 2 octets pour l’option **Content-format** indispensable pour indiquer que l’on transporte du CBOR ;
- 3 octets pour l’option **No-response**, également indispensable pour indiquer que l’on ne veut surtout pas d’acquittement (Sigfox les limitant à 4 par jour).

Soit, au total 11 octets. Il n’en reste plus qu’un. Si la température dépasse les 23 °C, l’information ne sera pas transportable.

13.5 SCHC

Static Context Header Compression, défini dans le [RFC 8724](#), donne l’acronyme SCHC que l’on prononce Chic. SCHC propose un mécanisme de compression générique des en-têtes. SCHC se base sur un contexte commun à l’émetteur et au récepteur qui va permettre d’éliminer les informations connues dans le message et de ne transmettre que les données qui ne peuvent pas être prédéterminées.

Nous allons mettre en œuvre une version simplifiée. Dans le message CoAP, nous avons besoin du champ Message ID qui va servir à éliminer les doublons qui pourraient apparaître dans le réseau. Les serveurs CoAP gardent en mémoire les informations concernant les Messages ID pendant 5 minutes. Donc, la numérotation des messages doit permettre des périodes plus longues avant de réutiliser les mêmes valeurs. Nous allons également transporter 3 types de ressources : /temperature, /humidity et /pressure.

On peut donc construire manuellement un en-tête SCHC avec les champs suivants :

- 2 bits pour numérotter les règles de compression. Dans notre cas, nous n’utiliserons que la règle 00 ;
- 4 bits pour numérotter les messages ID, ce qui donne 15 valeurs possibles de 1 à 15. Au pire, il faudrait émettre plus de 3 trames par minutes pour que le serveur CoAP traite deux trames différentes comme des doublons ;
- 2 bits pour désigner les chemins d’URI ; 3 seront utilisés.

Cela fait appel à plusieurs techniques de compression définies par SCHC :

- ***not_sent*** : la valeur d'un champ n'est pas envoyée sur le réseau car elle se trouve dans les règles. Cela s'appliquera à la plupart des champs ;
- ***Least Significant Bit*** : on n'envoie que les bits de poids faible. Cela s'applique au champ Message ID duquel on ne va transmettre que les 4 bits de poids faible ;
- ***Matching_sent*** : au lieu d'envoyer la valeur, on va envoyer un index sur un tableau commun. Cela s'applique à Uri-path où l'on enverra 00 pour l'élément temperature, 01 pour l'élément pressure, et 10 pour élément humidity.

13.5.1 Emission côté client

La compression très simplifiée que nous allons effectuer se fera au moment de l'envoi des données pour Sigfox dans le programme `coap_full_sensor.py`.

Listing 13.12 – `coap_full_sensor.py`

```

174 def send_coap_message(sock, destination, uri_path, message):
175     if destination[0] == "SIGFOX": # do SCHC compression
176         global sigfox_MID
177
178         """ SCHC compression for Sigfox, use rule ID 0 stored on 2 bits,
179         followed by MID on 4 bits and 2 bits for an index on Uri-path.
180
181         the SCHC header is RRMMMMUU
182         """
183
184         uri_idx = [ 'temperature', 'pressure', 'humidity', 'memory' ].index(uri_path)
185
186         schc_residue = 0x00 # ruleID in 2 bits RR
187         schc_residue |= (sigfox_MID << 2) | uri_idx # MMMM and UU
188
189         sigfox_MID += 1
190         sigfox_MID &= 0x0F # on 4 bits
191         if sigfox_MID == 0: sigfox_MID = 1 # never use MID = 0
192
193         msg = struct.pack("!B", schc_residue) # add SCHC header to the message
194         msg += cbor.dumps(message)
195
196         print ("length", len(msg), binascii.hexlify(msg))
197
198         s.send(msg)
199         return None # don't use downlink

```

Dans le cas de Sigfox, on prend l'index en cherchant l'élément dans le tableau (ligne 183). On construit ensuite l'octet SCHC en ajoutant un numéro de règle (ligne 185) et les 4 bits de poids faible du champ Message ID qui va donc varier de 1 à 15 (ligne 186). Puis, l'on concatène les données CBOR à envoyer (ligne 193).

Pour les autres technologies de transmission, l'en-tête CoAP est construite avec les fonctions du module `CoAP.py`.

```

200     # for other technologies we send a regular CoAP message
201     coap = CoAP.Message()
202     coap.new_header(type=CoAP.NON, code=CoAP.POST)

```

```

204     coap.add_option (CoAP.Uri_path, uri_path)
205     coap.add_option (CoAP.Content_format, CoAP.Content_format_CBOR)
206     coap.add_option (CoAP.No_Response, 0b00000010) # block 2.xx notification
207     coap.add_payload(cbor.dumps(message))
208     coap.dump(hexa=True)
answer = CoAP.send_ack(s, destination, coap)

```

L'envoi de la trame se fait avec la fonction `send_ack`. Comme le message est de type NON, cette fonction n'attend pas de réponse après l'émission des données.

13.5.2 Réception côté serveur

Si les données émises par `coap_full_sensor.py` passent par un réseau LPWAN, le programme `generic_coap_relay.py` va servir d'intermédiaire pour les envoyer au serveur CoAP. Le programme est presque identique à celui que nous avions utilisé pour transmettre dans la session précédente. Les seules différences sont :

- les numéros de ports utilisés : 5683 au lieu de 33033 ;
- l'ajout de la décompression SCHC pour les données venant de Sigfox.

Listing 13.13 – `generic_coap_relay.py`

```

# Sigfox use SCHC compression, first byte is CoAP compressed header
70 SCHC_byte = payload[0]

72 ruleID = SCHC_byte >> 6
73 mID = (SCHC_byte & 0b00111100) >> 2
74 uri_idx = SCHC_byte & 0b0000_0011

76 m = aiocoap.message.Message(
77     mtype=NON,
78     code=POST,
79     mid=mID,
80     payload=payload[1:])
81
82 m.opt.uri_path = (
83     ["temperature", "pressure", "humidity", "memory"][uri_idx],
84 )
85
86 m.opt.content_format = 60
87 m.opt.no_response = 0b0000_0010
88
downlink = forward_data(m.encode())

```

Pour reconstruire l'en-tête CoAP, SCHC se base sur des règles pour rendre le traitement indépendant des champs employés par le protocole. Mais ici, pour faire plus simple, nous utilisons le module Message d'`aiocoap` pour reconstituer le message CoAP.

Dans un premier temps, on extrait du premier octet la valeur du champ message ID et l'index des URI (ligne 72 à 74) puis, à partir de ces valeurs et de celles connues à l'avance, le message CoAP est reconstitué.

Dans tous les cas, la fonction `forward_data` est utilisée. Elle retourne la réponse du serveur CoAP qui est renvoyée au réseau LPWAN suivant les principes que nous avions vus lors de la session précédente. Nous ne l'avons pas mis en œuvre pour Sigfox pour éviter une erreur vu que 4 messages par jours sont autorisés.

13.5.3 serveur CoAP

Finalement, le programme `coap_server.py` a été étendu à différents chemins d'URI pour traiter les différentes ressources que vont nous envoyer les capteurs. Et dans le traitement de la ressource, l'appel a la fonction `to_btajout`.

13.6 Pistes d'améliorations

Vous pouvez étendre cet ensemble de programmes. Voici quelques pistes d'amélioration :

- les capteurs envoient également leur mémoire disponible. Cela peut être utile pour détecter une fuite dans la mémoire ; par exemple quand une structure n'est jamais libérée. La structure CBOR est bien envoyée mais ni `coap_server.py` ni Beebotte n'ont été configurés pour afficher ces valeurs. Vous pouvez donc l'intégrer dans la chaîne de traitement ;
- le POST de la mémoire conduit à l'émission d'un message en downlink pour indiquer l'erreur 4.04. C'est une conséquence de l'utilisation de l'option CoAP **no_response** qui ne bloque que les notifications de type 2.xx. Vous pouvez modifier le module CoAP.py pour que la réponse soit traitée. Par exemple, en limitant à un envoi par jour de cette ressource ;
- en plus de la mémoire, il peut être intéressant d'envoyer le niveau de la batterie. La page [Correct formula for BATT monitoring on expansion board | Pycom user forum](#)⁴ donne des indications pour récupérer le niveau de la batterie ;
- finalement, nous n'avons pas réglé tous les problèmes d'interopérabilité. Si, par exemple, vous voulez envoyer toutes les heures un relevé de la mémoire libre et toutes les minutes la température, vous devez également modifier le programme `coap_server.py`. Si ces paramètres sont transmis de manière optimale par le serveur, le programme `coap_serveur` peut devenir complètement indépendant de la valeur mesurée.

4. <https://forum.pycom.io/topic/1690/correct-formula-for-batt-monitoring-on-expansion-board>



14. LwM2M

Il est temps de voir une vrai plateforme IoT qui met en œuvre ce que nous venons d'apprendre sur CoAP et REST. Il faut installer deux programmes **java**¹ :

```
wget https://ci.eclipse.org/leshan/job/leshan/lastSuccessfulBuild/artifact/leshan-client-demo.jar
```

et

```
wget https://ci.eclipse.org/leshan/job/leshan/lastSuccessfulBuild/artifact/leshan-server-demo.jar
```

14.1 Introduction

Lightweight Machine to Machine (LwM2M) est une architecture définie par l'Open Mobile Alliance (OMA) à l'origine pour permettre aux opérateurs de gérer certaines ressources sur les téléphones mobiles. Mais cette architecture peut être étendu à d'autres environnement.

Les spécifications sont accessibles sur le site de l'OMA². Nous allons utiliser une mise en œuvre en java appelé . Pas de panique nous n'allons pas programmer nous aurons juste besoin d'un navigateur Web et de **Wireshark**.

LwM2M comme son nom l'indique se veut léger, c'est-à-dire que les mises en œuvre ne doivent pas être trop complexes et que le trafic engendré ne doit pas non plus être trop important. LwM2M est une plate-forme et elle va donc faire plus qu'un simple trafic REST. En particulier, l'un des rôles est de permettre aux objets de s'enregistrer et de décrire leurs caractéristiques. LwM2M va également structurer fortement les ressources en imposant des règles de nomsages relativement contraintes et le format des ressources.

1. Il s'agit d'une mise en œuvre assez ancienne, de nouvelles spécifications existent et sont quelque peu différentes, mais les concepts n'ont pas changés

2. <https://omaspecworks.org/what-is-oma-specworks/iot/lightweight-m2m-lwm2m/>

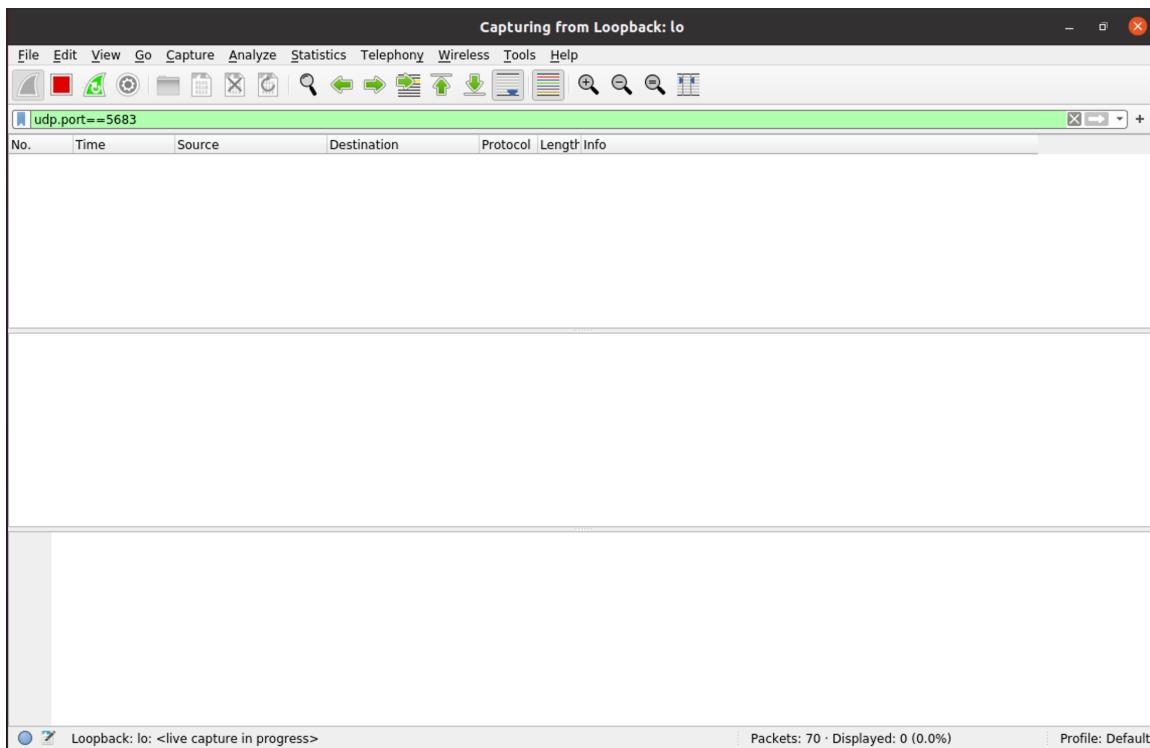


FIGURE 14.1 – Initialisation de Wireshark pour capturer le trafic CoAP

14.2 Architecture

Comme tout système qui se respecte, LwM2M fonctionne en mode client/serveur. Le serveur est la plate-forme de gestion des objets et les clients sont les objets connectés sur le réseau.

Dans un premier temps, lancez Wireshark sur l'interface **loopback** et filtrez le trafic en le limitant au port 5683 pour UDP (celui de CoAP) en tapant `udp.port==5683`. Comme le montre la figure 14.1.

Nous allons maintenant lancer le serveur LwM2M, tapez³ :

```
> java -jar ./leshan-server-demo.jar
2020-10-22 01:34:09,365 INFO LeshanServer - LWM2M server started at
coap://0.0.0.0/0.0.0.0:5683 coaps://0.0.0.0/0.0.0.0:5684
2020-10-22 01:34:09,553 INFO LeshanServerDemo - Web server started at
http://0.0.0.0:8080/.
```

Il nous indique qu'il utilise le port 5683 pour CoAP et que l'on peut superviser le serveur avec un navigateur sur le port 8080.

Lancez le navigateur sur l'URI `http://127.0.0.1:8080`, la page indiquée figure 14.2 on the next page apparaît.

Vous pouvez remarquer que l'ouverture du serveur LwM2M n'a provoqué aucun trafic CoAP sur l'analyseur réseau.

3. Sous Linux, tapez `apt install -y default-jre` pour installer Java.

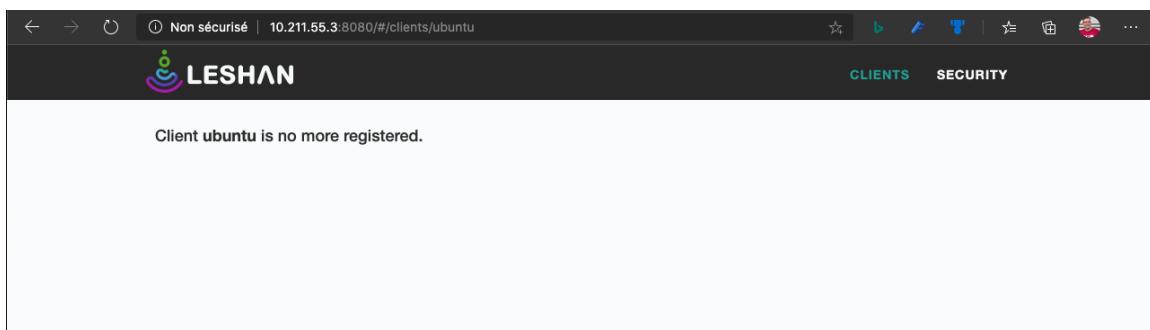


FIGURE 14.2 – Page d'accueil du serveur LwM2M

Lancez maintenant dans une autre fenêtre, le client LwM2M :

```
> java -jar ./leshan-client-demo.jar
2020-10-22 01:49:51,063 INFO LeshanClientDemo - Commands available :

- create <objectId> : to enable a new object.
- delete <objectId> : to disable a new object.
- update : to trigger a registration update.
- w : to move to North.
- a : to move to East.
- s : to move to South.
- d : to move to West.

2020-10-22 01:49:51,064 INFO LeshanClient - Starting Leshan client ...
2020-10-22 01:49:51,158 INFO CaliforniumEndpointsManager - New
endpoint created for server coap://localhost:5683 at
coap://0.0.0.0:48274
2020-10-22 01:49:51,159 INFO LeshanClient - Leshan client
[endpoint:ubuntu] started.
2020-10-22 01:49:51,160 INFO DefaultRegistrationEngine - Trying to
register to coap://localhost:5683 ...
2020-10-22 01:49:51,252 INFO DefaultRegistrationEngine - Registered
with location '/rd/BOr5Pg7yW8'.
2020-10-22 01:49:51,252 INFO DefaultRegistrationEngine - Next
registration update to coap://localhost:5683 in 53s...
```

14.3 Enregistrement d'un Objet

Comme nous utilisons une adresse *loopback*, il est un peu plus difficile de repérer dans la trafic Wireshark récupéré (cf. figure 14.3 on the following page) qui est le client et qui est le serveur. Mais comme Le serveur attend sur le port 5683, en regardant de plus près un paquet, on peut déterminer s'il a été émis par le client ou le serveur.

14.3.1 Analyse de l'en-tête CoAP

On peut voir sur la trace, que le client contacte le serveur pour lui indiquer ses propriétés.

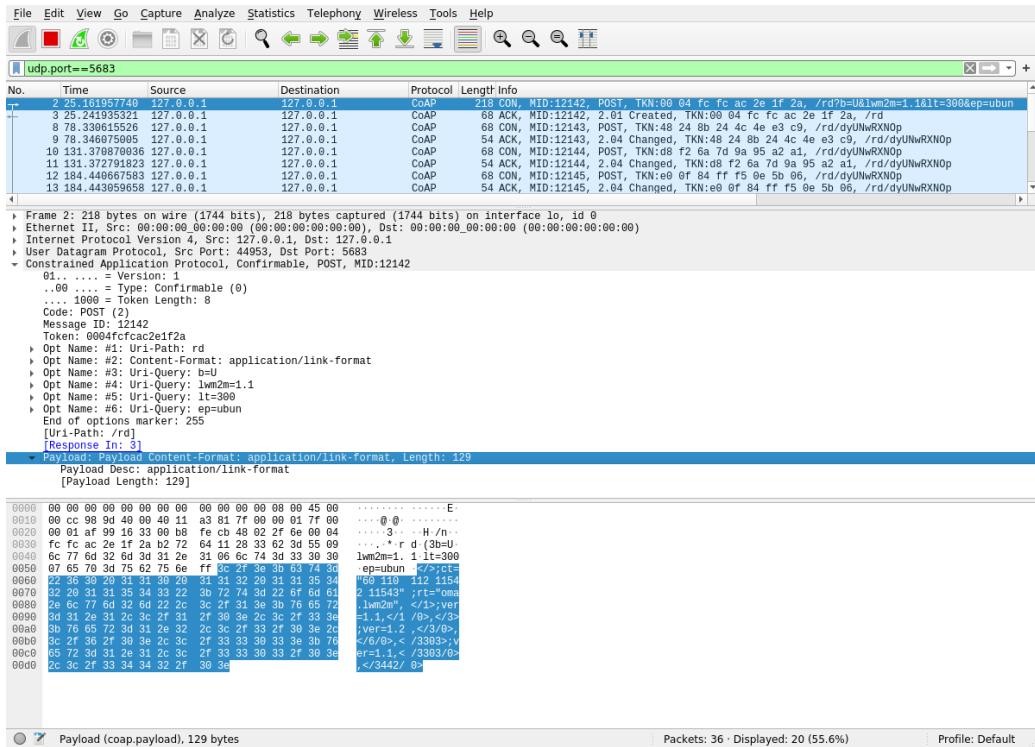


FIGURE 14.3 – Premières captures LwM2M

Question 14.3.1: Méthode

Quelle est la nature de la requête émise par le client :

- GET
- POST
- PUT
- DELETE

Question 14.3.2: URI

Quelle est le chemin de l'URI :

- vide
- /rd
- /lwm2m

Question 14.3.3: Content

Quelle est le format du contenu (content-format) :

- text
- XML
- JSON
- link-format
- CBOR

Question 14.3.4: Période

A quelle période voyez-vous d'afficher des messages CoAP ?

Pour décoder l'Uri-Query de la requête CoAP il faut d'aider du document LwM2M Core Specification⁴ et de son tableau 6.2.1 page 40.

Question 14.3.5: b=U

Que signifie b=U ?

- Les données seront émises avec les unités (Units)
- Le référentiel des unités est la représentation Universelle
- Le protocole sous-jacent est en mode datagramme (UDP)
- Le client n'est pas référencé (Unreferenced)

Question 14.3.6: lwm2m=1.1

Que signifie lwm2m=1.1 ? Il s'agit...

- de la version lwm2m du client
- de la taille mémoire de l'implémentation (1.1 ko)
- de la version lwm2m du serveur

Question 14.3.7: lt=300

Que signifie lt=300 ?

- C'est le nombre maximal d'objets (less than 300)
- C'est la taille maximale d'un échange (300 octets)
- C'est la durée de vie de l'enregistrement d'un objet (lifetime)

14.3.2 Analyse du contenu du POST

Revenons au contenu du message de la requête POST émise initialement par le client. Que veut dire **link-format**? Nous n'avons pas encore vu ce format. Heureusement, l'IANA est notre ami et en allant chercher à quoi correspond cette valeur, sur cette page⁵, on trouve que le [RFC 6690](#) définit ce contenu.

4. http://www.openmobilealliance.org/release/LightweightM2M/V1_1_1-20190617-A/OMA-TS-LightweightM2M_Core-V1_1_1-20190617-A.pdf

5. <https://www.iana.org/assignments/core-parameters/core-parameters.xhtml#content-formats>

Il utilise un format particulier qui est utilisé initialement par le Web pour définir des relations entre pages initialement définies dans le [RFC 5988](#). Il est important de remarquer, et après la lecture devient plus claire, que les URI sont notées entre <>. Ensuite, on trouve des attributs liés à cet URI séparés par des points-virgules. Les virgules séparent les définitions.

En suivant ces règles de notation, les données émises par le client peuvent être formatés de la manière suivante :

```
</>;ct="60 110 112 11542 11543";rt="oma.lwm2m",
</1>;ver=1.1,
</1/0>,
</3>;ver=1.2,
</3/0>,
</6/0>,
</3303>;ver=1.1,
</3303/0>,
</3442/0>
```

Le client utilise ce format pour décrire les 9 ressources qu'il possède et qui sont identifiées par ces 9 chemins d'URI. Le nommage des ressources peut paraître étrange ; nous verrons par la suite à quoi il correspond :

- La première ligne concerne la racine </> pour laquelle deux attributs sont associés, il seront donc applicables à l'ensemble des éléments.
 - Le premier **ct**⁶ définit les formats des représentations possibles des objets. La partie entre guillemets fait référence aux valeurs de Content-format de CoAP listé tableau 12.4 sur page 161. On y retrouve respectivement les types CBOR, SenML et pour les deux derniers un format orienté TLV propre à LwM2M.
 - Le second **rt**⁷ indique le type de ressource (*resource-type*), c'est-à-dire indique comment elles seront représentées. Ici, cela indique que les ressources suivront les spécifications LwM2M de l'OMA.
- Les lignes suivantes décrivent, toujours de manière hiérarchique, les chemins d'URI et les attributs associés. L'attribut **ver** indique la version du standard utilisée pour définir la ressource.

Question 14.3.8: rt

À quoi sert l'attribut **rt** ?

- à donner la sémantique (i.e. comment interpréter) de la ressource.
- à indiquer la taille de la ressource en faisant référence à lwm2m.
- à indiquer la version de lwm2m utilisée.
- à aider à déboguer les échanges.

14.3.3 Définition des ressources

LwM2M représente les ressources d'une manière assez originale pour être à la fois compact, universel (ce qui est quelquefois oxymorique) et flexible. Pour ce faire, tout est désigné par des chiffres. Le chapitre 7 page 63 du document principal⁸ contient le schéma figure 14.4 illustrant

6. Voir [RFC 7252](#) chapitre 7.2.

7. Voir [RFC 6690](#) chapitre 3.1.

8. http://www.openmobilealliance.org/release/LightweightM2M/V1_1_1-20190617-A/OMA-TS-LightweightM2M_Core-V1_1_1-20190617-A.pdf

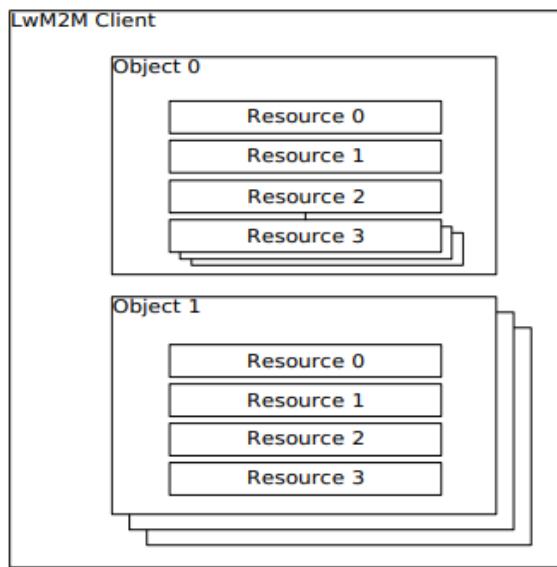


Figure: 7.1.-1 Relationship between LwM2M Client, Object, and Resources

FIGURE 14.4 – Hierarchie de nommage

cette hiérarchie :

- Un Objet physique (ie notre client) contient une liste d'objets numériques⁹ identifié par un numéro. Ce numéro est attribué par l'OMA. Par exemple, un capteur de température aura la valeur 3303. Une liste des valeurs déjà attribuées peut être trouvée à cette adresse¹⁰.
- Un client peut évidemment posséder plusieurs instance d'un objet numérique, par exemple, une station météo pourrait avoir un capteurs de température intérieur et extérieur. Le deuxième chiffre du chemin de l'URI indique cette instance. S'il n'y en a qu'un, il prend la valeur 0.
- Finalement, l'objet peut être complexe et contenir plusieurs informations appelée ressources. En cliquant sur le chiffre 3303 dans la page web précédente, on obtient la description de l'objet. On peut voir que :
 - 5700 représente la valeur mesurée par le capteur
 - 5601 la valeur minimale
 - 5602 la valeur maximale
 - 5701 indique les unités
 - 5605 permet de remettre à zéro le calcul des minima et maxima
 Ces valeurs peuvent se retrouver dans plusieurs objets numériques.
- la version 1.2 du standard introduit également la possibilité d'avoir plusieurs instances d'une ressource.

Valeurs associées aux objets numériques

Les objets numériques peuvent être définis par LwM2M, ils concernent ceux dédiés à la gestion du protocole. Par exemple, l'objet numérique :

9. LwM2M appelle cette information *object* ce qui rend en français la définition ambiguë puisque Objet est aussi la traduction de *thing*

10. <http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html#resources>

- 0 définit l'environnement de sécurité du client. Il contient l'adresse du serveur, les clés de chiffrement,...
- 1 définit les paramètres pour la communication avec le serveur, comme par exemple la durée de vie de l'information en seconde,...
- 3 décrit les caractéristiques de l'équipement comme sa version logicielle,...
- ...

Les valeurs comprises entre 2 048 et 10 240 sont définies par des organismes de standardisation partenaire de l'OMA, comme par exemple :

- l'IPSO Alliance qui va définir des objets numériques types, comme un capteur de température ayant pour référence 3303,
- la GSMA qui définit les objets pour la gestion des téléphones cellulaire,
- ...

Les valeurs supérieures sont réservées aux industriels qui peuvent enregistrer leurs propres spécifications.

Valeurs associées aux ressources

La figure 14.5 on the next page, copiée du site Web de l'OMA, donne un exemple de définition d'un objet numérique. A l'objet 3303 sont associés un certain nombre de ressources, générique que l'on peut aussi trouver dans d'autres objets numériques. Ainsi :

- 5700 fait référence à la valeur mesurée représenté comme un nombre flottant. Ce nombre ne peut que être lu (R)
- 5700 est une chaîne de caractère qui précise l'unité de mesure.
- 5601 et 5602 conservent les valeurs minimales et maximales. Elles peuvent être remise à zéro via la ressource 5605 qui conduit à une exécution d'un programme (E).
- les 5603 et 5604 valeurs correspondent à la plage d'utilisation du capteur et ne peuvent pas être modifiées.

Ainsi 3303/3/5601 représente la valeur minimale (5601) du quatrième capteur (3 car on commence à 0) de l'objet numérique température (3303).

Par rapport à **Modbus**, où l'on devait télécharger la documentation de l'Objet (cf tableau 4.3 on page 54), LwM2M offre une manière standard de décrire l'information produite ou consommée par un Objet. De plus, la définition de l'objet numérique, en plus d'être visualisable sous forme de tableau sur le site web, existe aussi en **XML** pour être traitée informatiquement et permettre l'interopérabilité.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- BSD-3 Clause License ... -->
<LWM2M xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://openmobilealliance.org/tech/profiles/LWM2M.xsd">
    <Object ObjectType="MODefinition">
        <Name>Temperature</Name>
        <Description1>
            This IPSO object should be used with a temperature sensor to report a temperature measurement. It also provides resources for minimum/maximum measured values and the minimum/maximum range that can be measured by the temperature sensor. An example measurement unit is degrees Celsius.
        </Description1>
        <ObjectID>3303</ObjectID>
        <ObjectURN>urn:oma:lwm2m:ext:3303:1.1</ObjectURN>
        <LWM2MVersion>1.0</LWM2MVersion>
        <ObjectVersion>1.1</ObjectVersion>
        <MultipleInstances>Multiple</MultipleInstances>
        <Mandatory>Optional</Mandatory>
        <Resources>
            <Item ID="5700">
```

Temperature

Description

This IPSO object should be used with a temperature sensor to report a temperature measurement. It also provides resources for minimum/maximum measured values and the minimum/maximum range that can be measured by the temperature sensor. An example measurement unit is degrees Celsius.

Object definition

Name	Object ID	Object Version	LWM2M Version
Temperature	3303	1.1	1.0
Object URN		Instances	Mandatory
urn:oma:lwm2m:ext:3303:1.1		Multiple	Optional

Resource Definitions

ID	Name	Operations	Instances	Mandatory	Type	Range or Enumeration	Units	Description
5700	Sensor Value	R	Single	Mandatory	Float			Last or Current Measured Value from the Sensor.
5601	Min Measured Value	R	Single	Optional	Float			The minimum value measured by the sensor since power ON or reset.
5602	Max Measured Value	R	Single	Optional	Float			The maximum value measured by the sensor since power ON or reset.
5603	Min Range Value	R	Single	Optional	Float			The minimum value that can be measured by the sensor.
5604	Max Range Value	R	Single	Optional	Float			The maximum value that can be measured by the sensor.
5701	Sensor Units	R	Single	Optional	String			Measurement Units Definition.
5605	Reset Min and Max Measured Values	E	Single	Optional				Reset the Min and Max Measured Values to Current Value.
5750	Application Type	RW	Single	Optional	String			The application type of the sensor or actuator as a string depending on the use case.
5518	Timestamp	R	Single	Optional	Time			The timestamp of when the measurement was performed.
6050	Fractional Timestamp	R	Single	Optional	Float	0..1	s	Fractional part of the timestamp when sub-second precision is used (e.g., 0.23 for 230 ms).
6042	Measurement Quality Indicator	R	Single	Optional	Integer	0..23		Measurement quality indicator reported by a smart sensor. 0: UNCHECKED No quality checks were done because they do not exist or can not be applied. 1: REJECTED WITH CERTAINTY The measured value is invalid. 2: REJECTED WITH PROBABILITY The measured value is likely invalid. 3: ACCEPTED BUT SUSPICIOUS The measured value is likely OK. 4: ACCEPTED The measured value is OK. 5-15: Reserved for future extensions. 16-23: Vendor specific measurement quality.
6049	Measurement Quality Level	R	Single	Optional	Integer	0..100		Measurement quality level reported by a smart sensor. Quality level 100 means that the

FIGURE 14.5 – Définition de l'objet numérique 3303 pour la température

```

<Name>Sensor Value</Name>
<Operations>R</Operations>
<MultipleInstances>Single</MultipleInstances>
<Mandatory>Mandatory</Mandatory>
<Type>Float</Type>
<RangeEnumeration></RangeEnumeration>
<Units></Units>
<Description>Last or Current Measured Value from the Sensor.</Description>
</Item>
<Item ID="5601">
    <Name>Min Measured Value</Name>
    <Operations>R</Operations>
    <MultipleInstances>Single</MultipleInstances>
    <Mandatory>Optional</Mandatory>
    <Type>Float</Type>
    <RangeEnumeration></RangeEnumeration>
    <Units></Units>
    <Description>
        The minimum value measured by the sensor since power ON or reset.
    </Description>
</Item>
...

```

Question 14.3.9: 3301/0/5602

En vous aidant de la page de description des objets numériques et des ressources^a, que représente l'URI *3301/0/5602* ?

- la température maximale du premier capteur
- l'humidité minimale du capteur 0
- la luminosité maximale du capteur 0

a. <http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html#resources>

Question 14.3.10: 10340/0/2

Que représente l'URI *10340/0/2* ?

- la température en Fahrenheit de l'objet 10340
- la longitude dans des coordonnées GPS
- le statut actif ou non d'une caméra

Question 14.3.11: Dimmer

Quelle URI permet d'accéder à l'élément contrôlant la variation lumineuse (*Dimmer*) d'un éclairage (*Light Control*) ? Quelle valeur maximale peut prendre cette ressource ?

Question 14.3.12: Hz

Quel serait le chemin d'URI pour obtenir la fréquence d'un courant électrique sur la deuxième phase ?

Question 14.3.13: Qui est qui ?

Dans le premier échange capturé :

```
</>;ct="60 110 112 11542 11543";rt="oma.lwm2m",
</1>;ver=1.1,
</1/0>,
</3>;ver=1.2,
</3/0>,
</6/0>,
</303>;ver=1.1,
</303/0>,
</3442/0>
```

1

Location

3

Device

6

LwM2M v1.1 Test Object

3303

LwM2M Server

3442

Temperature

14.4 Resource Directory

Nous allons nous focaliser sur le premier échange que nous avons commencé à analyser en regardant le message émis par le client vers le serveur LwM2M. Comme nous l'avons vu, ce premier message contient la description des ressources présentes sur le client. Les valeurs des identifiants d'objets numérique (Object ID) et de ressources (Resource ID) permettent au serveur de savoir ce que peut faire le client, vu qu'elles sont normalisées. Le client LwM2M envoie ces informations sur un chemin d'URI bien connu /rd (pour *resource description*).

Question 14.4.1: Réponse du serveur LwM2M

Que répond le serveur (cf. figure 14.6 on the next page) ?

- Il acquitte juste le message.
- Rien.
- Il renvoie une URI qui servira par la suite à identifier l'objet.
- Il retourne les données de chiffrement des messages.

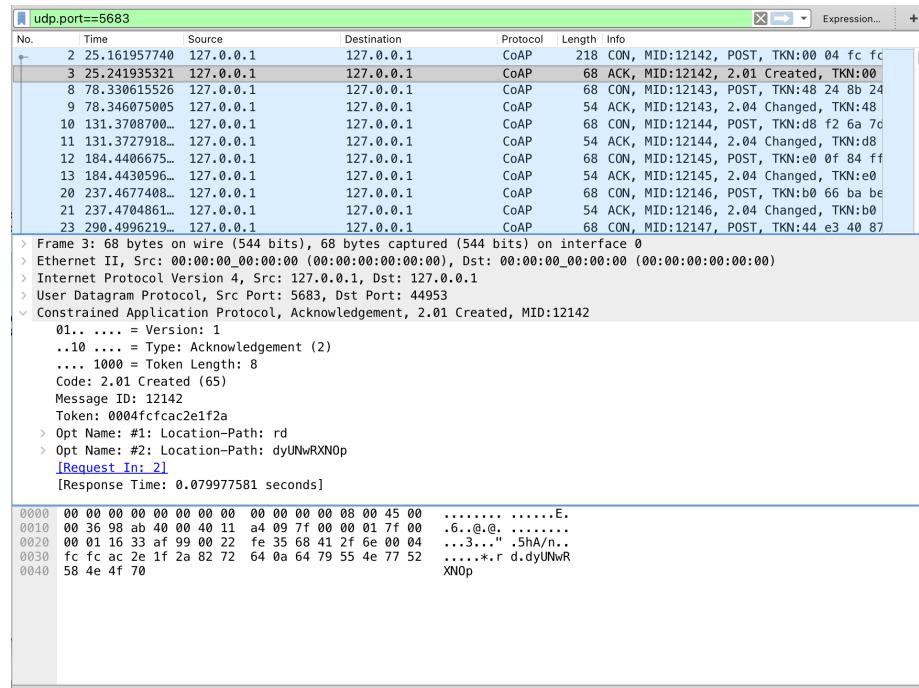


FIGURE 14.6 – Réponse du serveur LwM2M au POST du client

Question 14.4.2: Wait and See

Si on laisse la plate-forme fonctionner sans intervenir, que voit-on sur l'analyseur réseau ?

- Rien.
- Le capteur envoie des informations indiquant un changement de valeur mesurée.
- Le client envoie les valeurs mesurées même s'il elles n'ont pas changé.
- Le client envoie un message vide vers le serveur pour indiquer qu'il est toujours présent.
- Le serveur envoie un message vide vers ses clients pour savoir s'ils sont toujours présents.

14.5 Interrogation du client LwM2M

Il est temps de revenir à l'interface de la plate-forme en ouvrant la page `http://127.0.0.1:8080` depuis le navigateur. Assurez vous que Wireshark capture toujours le trafic sur l'interface *loopback*. L'objet que nous avons inscrit apparaît. Cliquez dessus. Vous devez avoir quelque chose comme ce qui est représenté figure 14.7 on the facing page.

Dans le menu de gauche, on retrouve les objets numériques LwM2M décrit lors du premier POST de l'Objet.

En cliquant sur *Temperature*, les ressources LwM2M sont affichées. Les petits logos permettent d'effectuer des actions :

- *R* pour lire une ressource, *W* pour l'écrire et *EXE* pour exécuter un code (l'engrenage permet de définir les paramètres);

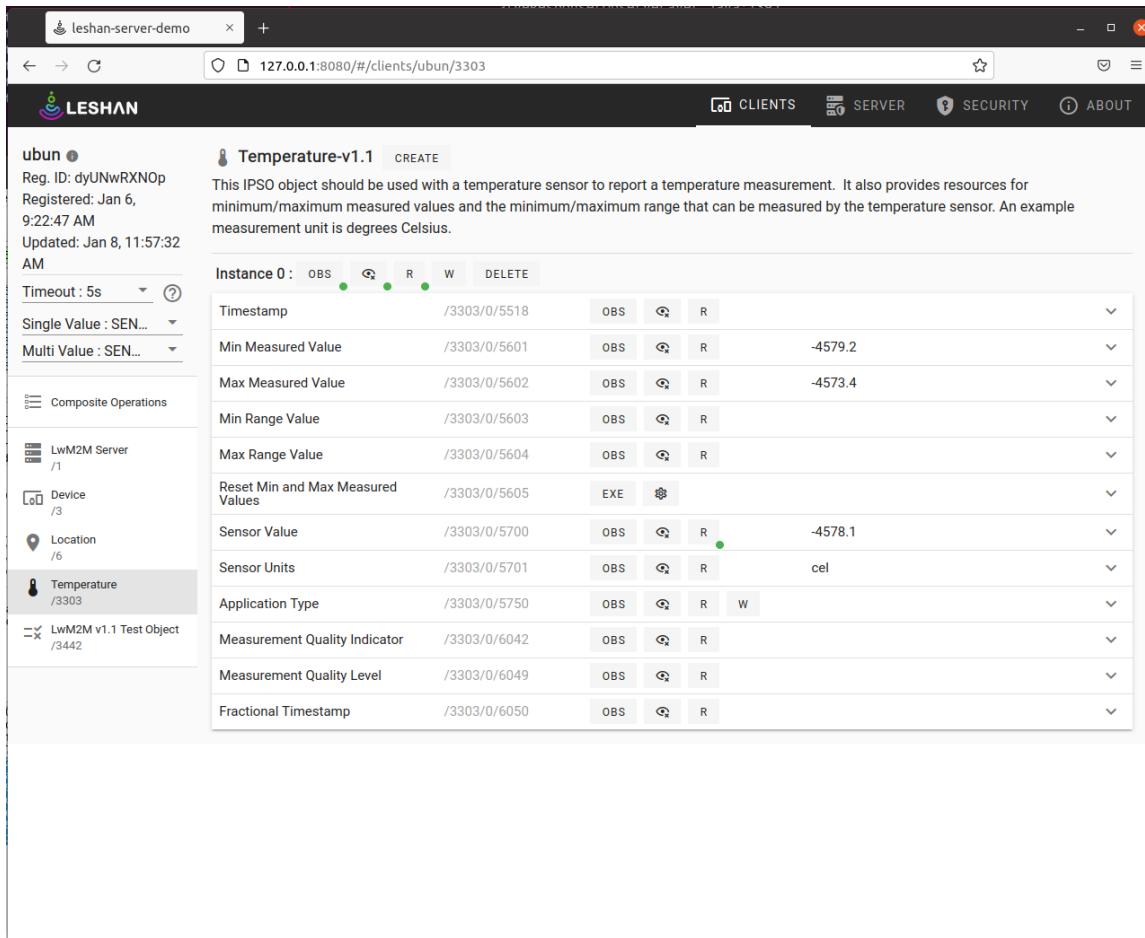


FIGURE 14.7 – Visualisation du serveur LwM2M

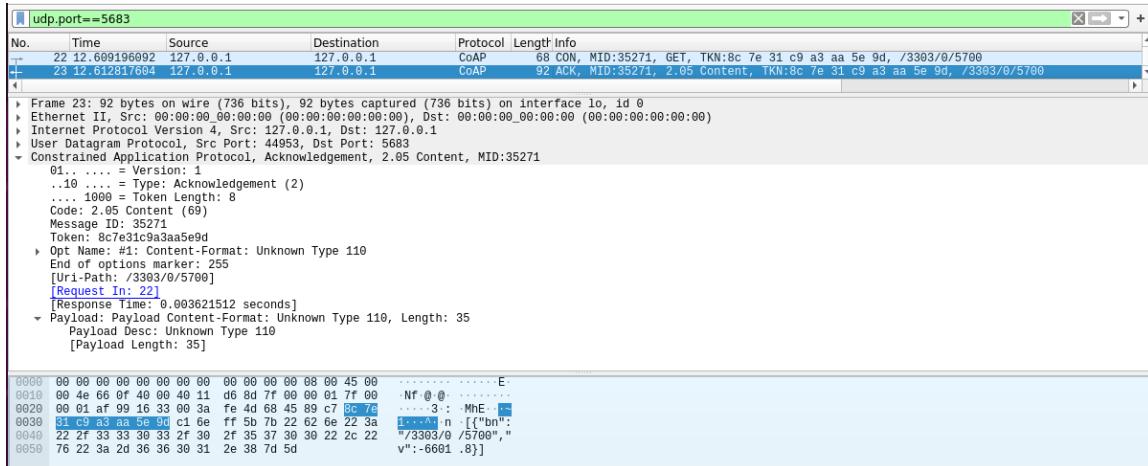


FIGURE 14.8 – Interrogation d'une ressource

- *OBS* permet de lancer un **Observe** sur une ressource, L'oeil avec la croix permet d'annuler un **Observe**.

Ces actions peuvent s'appliquer individuellement à chaque ressource, ou plus globalement à une instance d'un objet numérique.

14.5.1 Lecture simple

Dans le menu du gauche, sélectionnez SENML_JSON dans le menu déroulant *Single Value*. Cela permettra d'utiliser le format **SenML** plutôt que celui spécifié par LwM2M basé sur des TLV.

Pour l'objet numérique *Temperature* cliquez sur le bouton *R* de *Sensor Value*. La figure 14.8 monte cet échange et détaille la réponse. Le Serveur LwM2M agit comme un client REST et envoie au serveur REST du client LwM2M la requête :

GET /3303/0/5700

en indiquant le type 110 dans l'option **Accept** (cf. tableau 12.4 on page 161) qui correspond au format **SenML** codé en JSON .

La réponse doublement associée par le même **Token** et un message de type **ACK** de format SenML en JSON :

[{ "bn" : "/3303/0/5700" , "v" : -6601.8 }]

On y retrouve le nom de base (bn) correspondant au nom de la ressource et la valeur (v)¹¹.

Cette valeur s'affiche dans la page web du serveur LwM2M, le petit point vert indiquant que l'échange s'est déroulé correctement (cf. figure 14.9 on the next page).

¹¹. Il est évident que le résultat retourné n'a aucun sens physique, le client LwM2M émulant mal l'évolution d'une température sur le long terme.

Resource with max measure		/3303/0/5605	EXE	⚙️	
Values			OBS	R	-6601.8
Sensor Value	/3303/0/5700				
Sensor Units	/3303/0/5701		OBS	R	

FIGURE 14.9 – Relevé de la température

117.992509491 127.0.0.1	127.0.0.1	CoAP	96 NON, MID:17275, 2.05 Content, TKN:ff 6a 04 4d 3a 24 60 6d, /3303/0/5700
119.992474211 127.0.0.1	127.0.0.1	CoAP	96 NON, MID:17276, 2.05 Content, TKN:ff 6a 04 4d 3a 24 60 6d, /3303/0/5700
121.992141291 127.0.0.1	127.0.0.1	CoAP	96 NON, MID:17277, 2.05 Content, TKN:ff 6a 04 4d 3a 24 60 6d, /3303/0/5700
123.995776262 127.0.0.1	127.0.0.1	CoAP	96 CON, MID:17278, 2.05 Content, TKN:ff 6a 04 4d 3a 24 60 6d, /3303/0/5700
123.998316105 127.0.0.1	127.0.0.1	CoAP	46 ACK, MID:17278, Empty Message
125.992261142 127.0.0.1	127.0.0.1	CoAP	96 NON, MID:17279, 2.05 Content, TKN:ff 6a 04 4d 3a 24 60 6d, /3303/0/5700
127.991978003 127.0.0.1	127.0.0.1	CoAP	96 NON, MID:17280, 2.05 Content, TKN:ff 6a 04 4d 3a 24 60 6d, /3303/0/5700
129.991792102 127.0.0.1	127.0.0.1	CoAP	96 NON, MID:17281, 2.05 Content, TKN:ff 6a 04 4d 3a 24 60 6d, /3303/0/5700
131.992134965 127.0.0.1	127.0.0.1	CoAP	96 NON, MID:17282, 2.05 Content, TKN:ff 6a 04 4d 3a 24 60 6d, /3303/0/5700

FIGURE 14.10 – Emission d'un Observe avec un message CONFirmeable

14.5.2 Lecture d'une instance

En sélectionnant SENML_JSON dans le menu déroulant *Multi Value* et en cliquant sur le bouton *R* à droite de **Instance 0**, différents champs de l'instance se remplissent.

L'échange protocolaire est similaire à précédent. Le serveur LwM2M a émis la requête suivante, où la valeur de la ressource est omis :

```
GET /3303/0
```

et la réponse contient :

```
[{"bn":"/3303/0/","n":"5601","v":-6672.6},
 {"n":5602,"v":-4573.4},
 {"n":5700,"v":-6671.6},
 {"n":5701,"vs":"cel"}]
```

On remarque que le nom de base (bn)correspond à l'URI de l'instance et que chaque éléments contient un champs nom (n) contient l'identifiant numérique de la ressource ¹².

14.5.3 Observe

Il est aussi possible de suivre dans le temps l'état d'une ressource, nous allons le faire pour la ressource *Sensor Value* de l'objet numérique *Temperature*. Cliquez sur le bouton *OBS*.

Le serveur LwM2M émet la requête GET comme précédemment, en ayant inséré la l'option **Observe** avec la valeur 0 pour indiquer qu'il souhaite recevoir périodiquement des informations. Les réponses incluent également une option **Observe** dont la valeur est croissante.

Sur Wireshark, vous pourrez vérifier que périodiquement la réponse est envoyée en utilisant un message de type **CONFirmeable** plutôt que **NON confirmable** pour vérifier que le client REST sur le serveur LwM2M est toujours actif (cf figure 14.10)

Question 14.5.1: Fin d'observe

Un click sur l'oeil avec la croix permet au serveur d'annuler l'Observe. Quel message est émis ?

12. Le champ (vs) indique que le contenu doit être interprété comme une chaîne de caractères

Question 14.5.2: EXE

Quel message est émis, si on clique sur une ressource de type EXE, comme la remise à zéro des valeurs min et max ?



15. Answers to the questions

Question 1.8.1 page 24 Communicating objects are a brand new field, linked to the progress in miniaturization of electronic components :

- True
- False

Communicating objects have always existed, even before the deployment of Internet protocols. What is new is their integration into current Internet architectures to better process the information they produce.

Question 1.8.2 page 24 Which of these statements is true ?

- There are very few protocols to make objects communicate. As the Internet is a technology that has been very successful, its success will allow objects to communicate.
- **There are many solutions to enable objects to communicate, the Internet of Things must enable them to be federated.**

Each business has created its own protocols.

Question 1.8.3 page 24 What is the primary source of data creation in the IoT ?

- sensors
- nanocomputers (Raspberry Pi type)
- Internet
- Web servers

Measurements made by sensors are the primary source of IoT data creation.

Question 1.8.4 page 25 What are the main technological challenges for the Internet of Things (3 answers) ?

- Have a low energy consumption.

- Have a simplified protocol architecture.**
- Be able to run on open operating systems like Linux.
- Allow to secure data that may be sensitive.**
- Continuously transmit their status and measured values.

Objects are generally limited in energy and capacity. In order to limit energy consumption, it is not necessary to transmit or receive all the time, so the permanent transmission of data does not go in this direction. Similarly, an operating system such as Linux appears to be oversized.

Question 2.1.1 page 28 In the internet protocol stack, which protocols are responsible for routing packets to their destination (2 answers)

- | | | | |
|-----------------------------------|--------------------------------------|-------------------------------|-------------------------------|
| <input type="checkbox"/> Ethernet | <input type="checkbox"/> IPv4 | <input type="checkbox"/> MQTT | <input type="checkbox"/> JSON |
| <input type="checkbox"/> IEEE | <input type="checkbox"/> IPv6 | <input type="checkbox"/> HTTP | |
| <input type="checkbox"/> 802.15.5 | <input type="checkbox"/> UDP | <input type="checkbox"/> CoAP | |
| <input type="checkbox"/> Wi-Fi | <input type="checkbox"/> TCP | <input type="checkbox"/> XML | |

The protocol IP makes it possible to transport information from one end of the network to the other by using the addresses IP contained in the packets. There are two versions of this protocol : Internet Protocol version 4 (IPv4) initially deployed and IPv6 which offers much more addressing capacity.

Question 2.2.1 page 31 What is unique in the world (6 answers) ?

- A first name.
- A family name.
- a social security number used in France.**
- a passport number.**
- a cell phone number with its international prefix.**
- a full bank account number (IBAN).**
- the IP address of my machine in my private network.
- the IP address of a Coursera server (13.225.34.28).**
- the domain name plido.net.**
- the name of a city.

Neither the first name, nor the last name, nor their combination form unique sequences as John Smith would say. Passport numbers, social security numbers, telephone numbers, bank account numbers are by construction unique in their space, but there could be overlaps. This is why it is necessary to indicate the source or the authority that allocated it to guarantee uniqueness. For the passport, the authority is the country that issued it. The social security number corresponds to the registration number in the Répertoire National d'Identification des Personnes Physiques (RNIPP) (see <https://www.service-public.fr/particuliers/vosdroits/F33078> for France). The country code for the telephone number is assigned by the International Telecommunication Union (ITU) and each operator has its own numbering zone. For the bank account, it is of course the bank ; each bank having its own code contained at the beginning of the IBAN. The IP address in a private network is not unique. The [RFC 1918](#) defines address ranges that everyone can use locally. To go out on the Internet, you need a special device, called a NAT, which will convert the private address into a public address which is unique in the world. The servers must be in this public space and

therefore have a unique address in the world. Domain names are unique in the world by construction but they can be shared by several users. One can thus extend them like machine1.plido.net, machine2.plido.net... or, if it is the same machine, add a port number after to indicate the service : www.plido.net:80, www.plido.net:8080. As for the name of a city, it is not unique. It is also necessary to specify the country or even the region.

Question 2.2.2 page 33 The server keeps track of previous requests ?

- True
- False

The server responds to one request and then moves on to the next. It does not keep any state. On the other hand, a request can be used to modify the content of a resource, and the result of the modification will be kept.

Question 2.2.3 page 33 The World Wide Web is based on this principle of states for :

- work on both computers and smartphones,
- be able to serve a large number of requests,**
- encrypt communications.

See previous comment

Question 2.2.4 page 33 What formats are used to represent structured information (2 answers) :

- | | | | |
|-----------------------------------|-------------------------------|-------------------------------------|--------------------------------------|
| <input type="checkbox"/> Ethernet | <input type="checkbox"/> IPv4 | <input type="checkbox"/> MQTT | <input type="checkbox"/> JSON |
| <input type="checkbox"/> IEEE | <input type="checkbox"/> IPv6 | <input type="checkbox"/> HTTP | |
| <input type="checkbox"/> 802.15.5 | <input type="checkbox"/> UDP | <input type="checkbox"/> CoAP | |
| <input type="checkbox"/> Wi-Fi | <input type="checkbox"/> TCP | <input type="checkbox"/> XML | |

Il s'agit de XML et JSON qui permettent d'envoyer des données structurées. Les autres propositions indiquent des protocoles de transport de l'information de niveau 2, 3, 4 et 7.

Question 2.2.5 page 33 In the URI <https://plido.net/unit/definition.html>, where is the scheme ?

http : the Scheme indicates how the URI will be constructed.

Question 2.2.6 page 33 In the URI <https://plido.net:8080/unit/definition.html>, where is the authority ?

plido.net:8080 : this is the globally unique sequence.

Question 3.3.1 page 39 In the first column :

- The frame number assigned by Wireshark upon reception**
- The frame number is read directly in the Ethernet frame

These numbers are sequential, so they are assigned locally by Wireshark. Moreover there is no such field in Ethernet.

Question 3.3.2 page 39 In the second column :

- **The time of reception by Wireshark**
- The sending time of the frame

As in the previous case, this number is added by Wireshark, there is no protocol field indicating the transmission time.

Question 3.3.3 page 39 In the third and fourth columns :

- The Ethernet addresses of the hosts.
- Only the IPv4 addresses of the machines.
- **IPv4 or IPv6 addresses of machines.**

Wireshark treats IPv4 or IPv6 addresses in the same way, so they are displayed in these columns. The 48-bit Ethernet (or MAC) address is not displayed by default in this screen.

Question 3.3.4 page 39 In the fifth column :

- The application protocol (level 7).
- **The last (higher level) recognized protocol.**
- The level 4 protocol (here TCP or UDP).

Wireshark provides the higher level information. In the figure 3.2 on page 38 some frames are indicated as carrying the HTTP protocol, while others, usually acknowledgements, are indicated as TCP because they do not carry data from the higher layers.

Question 3.3.5 page 39 In the sixth column :

- The size in bits of the frame.
- **The size in bytes of the frame.**

The unit is the byte.

Question 3.3.6 page 39 In the seventh column :

- **A summary of the information carried by the higher-level protocol.**
- IPv4 options.
- The ASCII content of the highest level message.

Wireshark seeks to interpret the higher level protocol fields to provide a synthetic display of the information.

Question 3.3.7 page 41 In the following trace, we saw that the server responded to client requests with a 3-digit number. Using the [RFC 7231](#), can you assign the left digit to a category of notifications :

- | | |
|----------------------------|---|
| 1 <input type="checkbox"/> | <input type="checkbox"/> Error on the server side |
| 2 <input type="checkbox"/> | <input type="checkbox"/> Error on the client side |
| 3 <input type="checkbox"/> | <input type="checkbox"/> Unassigned |
| 4 <input type="checkbox"/> | <input type="checkbox"/> Success |
| 5 <input type="checkbox"/> | <input type="checkbox"/> Information |

- 0 : Unassigned
1 : Information
2 : Success
3 : Redirection
4 : Error on the client side
5 : Error on the server side

Question 3.3.8 page 42 Which standards organization published this document ?

- Microsoft
- ISO
- IEEE
- IETF

The prefix Request For Comments is characteristic of the IETF.

Question 3.3.9 page 43 Do HTTP headers have a fixed size (you can check the [RFC 7231](#) which gives indications on the protocol) ?

- the header is one line of 80 characters.
- a blank line separates the header from the content. The header can contain as many lines as necessary.

As we can see on the example given in the RFC for the response message, the HTTP header contains a mandatory line, followed by options. Their number is not fixed by the standard. To separate them from the data, a blank line acts as a separation.

Question 3.3.10 page 43 How are the optional lines in the header constructed ?

- keyword : values
- unformatted text
- keyword : data length : values

The header is of variable size. It has a mandatory first line giving the nature of the request or response, followed by optional information built on the "keyword : value" format. A blank line separates the header from the content.

Question 3.3.11 page 45 In the example, figure 3.4 on page 44, what is the Ethernet address of the machine sending the frame ?

10:65:30:b0:54:bf. Be careful, the frame starts with the destination address, followed by the source address.

Question 3.3.12 page 45 Does the Ethernet address 14:2e:5e:37:1e:6a found in the packet 3.4 on page 44 correspond to the Ethernet address of the recipient of the packet ? What does it correspond

to ?

No, the Ethernet address is only valid on this Ethernet network. To reach the recipient, the packet must cross several routers. As the frame was captured on the sending machine, this address is therefore that of the first router crossed.

Question 3.3.13 page 46 Using the figure 3.4 on page 44 or your Wireshark captures, what are the messages involved in closing the connection ?

The connection closing requires the sending of four messages. One of the ends, necessarily the one that opened the connection, sends a message with the END bit set. It is acknowledged by the other entity which in turn sends a message with the FIN bit set which will be acknowledged to finally close the connection.

Question 3.4.1 page 47 What URI should you enter in your browser to access this server locally.

The loopback address is 127.0.0.1 and the port is 8080, the URI path is /. The URI is therefore <http://127.0.0.1:8080/>.

Question 3.4.2 page 47 Using Wireshark, you can determine in the response the values of the HTTP options IndexContent-Type and Server.

We find the following values :

- Server: Werkzeug/2.0.2 Python/3.9.6
- Content-Type: text/html; charset=utf-8. The resource is encoded in HTML using 8-bit ASCII encoding.

Question 4.1.1 page 51 Looking at the exchanges in figure 4.4 on page 51 what is the measured value for humidity ?

Only the first exchange asks for the reading of 2 registers from the register with the address 0x0001. In the answer we obtain the value of the two consecutive registers (00 DA and 01 D4). The table 4.1 on page 51 indicates that humidity is the second register, so we have 01 D4.

Question 4.1.2 page 51 Looking at the exchanges in figure 4.4 on page 51 what is the evolution of the temperature over time ?

We find 3 values measured at 19 :11 :48, 20 :02 :39 and 20 :02 :44 : 00 DA, 00 D5 and 00 D5. That is, 21.8°C, 21.3°C and 21.3°C.

Question 4.1.3 page 52 What is the moisture content at the time of measurement ? The documentation states that it is a percentage with an accuracy of one tenth of a percent.

We had read the value 01 D4 in the register 0x02, that is 468 in decimal. Hence a humidity rate of 46.8%

Question 4.1.4 page 56 Let the given exchange figure 4.9 on page 57 correspond to a Modbus request and a response. What is the port number used by Modbus TCP.

There are two ports in the TCP header, but as it is a request, it is necessary to take the destination port :0x1f6, which is 502 in decimal

Question 4.1.5 page 56 Continuing the traffic analysis, which register value is requested.

Register 0x36 is requested, by looking in the table tab-meter-IR, it is the frequency in Hz.

Question 4.1.6 page 56 By analyzing the following packet, how can we verify that the response can match the previous request.

The sequence number 0x000c is the same in both frames.

Question 4.1.7 page 56 What value is returned. Is this consistent ?

The value of the register is 0x4247e95b corresponding to a floating number IEEE 754, converting it we obtain the value 49.9778862 Hz which is very close to the frequency of the European electrical network.

Question 6.8.1 page 79 What are the advantages of CBOR over JSON (2 answers) ?

- It is more compact in its data representation.**
- It allows to represent floating numbers.
- It compresses strings.
- It is easier to implement.**

CBOR does not compress strings. It just adds their length. Both CBOR and JSON encode floating-point numbers, so this is not an advantage of CBOR. On the other hand, using binary values instead of ASCII to represent numbers, not having brackets or braces, makes for a much more compact representation. Small numerical values are represented without any additional cost. Creating a CBOR encoder or decoder is much simpler than in JSON because the data representation is much stricter (no spaces, no line breaks, etc.). Both representations allow to use floating numbers so neither has an advantage on this point.

Question 6.8.2 page 80 Is a float always more compact in CBOR than in JSON ? - You can help yourself with <https://cbor.me>

- Yes, that is the goal of CBOR.
- Yes, for floats that have the decimal part at 0.
- Yes, for small precision floats (up to a hundredth).
- Yes, for high precision floats (6 digits after the decimal point).**

A floating number, whatever its value, is represented by 8 bytes in CBOR. In JSON, a floating number is represented by a string. So "3.0" needs 3 characters, so it is more compact than CBOR. But "3.1415926" is coded on 9 characters so less compact than CBOR.

Question 6.8.3 page 80 Consider a string in CBOR.

- It is compressed with an entropy algorithm (e.g. Huffman coding).
- It can contain accented characters.**
- Each character is coded on 6 bits.

Question 6.8.4 page 80 In CBOR, the size of an integer varies according to its value !

- True**
- False
- It depends on how this integer was declared.

Yes, an integer less than 23 will be encoded on a single byte. For larger values, the length must be added.

Question 6.8.5 page 80 In CBOR, an array can contain objects of different types.

- True**
- False
- It depends on how this table was declared.

True, we have the same flexibility as in JSON by nesting any data type in an array.

Question 6.8.6 page 80 We want to define an array of two elements as a fraction. What tag should precede the structure ? (you can use the [RFC 8949](#)).

This is tag 4, see the chapter **3.4.4. Decimal Fractions and Bigfloats** of the [RFC 8949](#).

Question 7.1.1 page 85 Why doesn't the client program work ?

- The IP address is not correct.
- The data serialization process is missing.**
- The variable t is not defined.
- The variable t cannot be read.

Question 7.2.1 page 87 Let us analyze the received sequence : 83fb40341086f3e8b66bfb408f3b7791c8d61ffb403fa15ba06088e

What does the byte 0x83 that starts the received CBOR structure correspond to ?

- to the coding of the positive integer 131.
- to the coding of the negative integer 132.
- to the definition of an array of 3 elements.**
- the definition of a CBOR map of 3 elements.
- to the definition of an array of undefined size.

0x83 is written in binary 100-0 0111. 100 is the major type for an array. The value 00111 is lower than 24. It is thus the number of elements of the array, thus an array of 3 elements.

Question 7.2.2 page 87 In this structure, what is the CBOR marker (in hexadecimal) that indicates that we have a floating number ?

0xFB, if we write it in binary we obtain 111-1 1100. The major corresponds to the category of floats and special values.

Question 7.2.3 page 87 What is the size of this floating number in bytes ?

8 bytes ; the minor part 1 1100 indicates that it is a float coded on 8 bytes

Question 7.2.4 page 88 What is the minimum and maximum size of the CBOR structure sent,

taking into account the possible values.

- The temperature can reasonably be between -30 and +50, that is to say -3000 and +5000 after the transformation into an integer. The minimum size, if we send 0, the size will be one byte. The maximum size is 3 bytes (1 for the type/length and 2 for the values)
- The pressure evolves around 1000, that is 100 000 after the transformation to integer. The size will always be 5 bytes (1 for the type/length, 4 for the values)
- The moisture content evolves between 0 and 100 , or 0 and 10 000 after the transformation in integer. The size is between 1 byte and 3 bytes.

If we add the type/length 0x83 to indicate an array of three elements, we obtain a minimum size of $1+1+5+1=8$ bytes and a maximum size of $1+3+5+3=12$ bytes.

Question 8.6.1 page 99 A quoi correspond la clé 'u' : 'Cel' que l'on retrouve dans la structure précédente ?

Unité = degrés Celcius

Question 8.6.2 page 99 Dans les deux représentations JSON et CBOR, de combien la taille est-elle accrue par l'ajout des mesures effectuées ? d'où viennent ces différences ?

Si l'on regarde le listing précédent, l'ajout des trois mesures fait augmenter la taille de 141 octets pour JSON et 84 pour CBOR. La différence vient de l'utilisation de nombre plus que de chaînes de caractères pour les clés. Ainsi 't' demande 3 caractères en JSON avec les guillemets, codé en CBOR, il faudrait 2 octets, un nombre inférieur à 23 se code sur un seul octet. Il y a également les virgules, espaces et fermeture de crochets qui ne sont pas présent en CBOR. Les nombres flottant comme 19.98 ont une représentation plus compacte en JSON (5 octets) qu'en CBOR où ils consomment 9 octets. Dans tous les cas l'accroissement est fortement dépendant du nom des éléments. Ici, il faut répéter à chaque fois temperature, humidity et pressure, soit 26 caractères.

Question 8.6.3 page 99 Si on ne s'intéressait qu'à une seule grandeur, par exemple l'humidité. A quoi ressemblerait la structure SenML en JSON ?

Chaque nouvelle entrée ajoute 35 octets à la structure : `[{'bn': 'device1', 'bt': 1640110457.0, 'n': 'humidity', 'u': '%RH', 'v': 28.46}, {'t': 10.0, 'v': 26.86}, {'t': 20.0, 'v': 26.96}, {'t': 30.0, 'v': 27.01}]`

Question 8.6.4 page 101 Pourrait-on utiliser le champ SenML *base value* pour diminuer la taille des données de pression atmosphérique ?

Cela serait possible, si cette ressource était envoyée seule.

Question 9.5.1 page 109 Le module I2C du LoPy dispose d'une fonction `scan` qui affiche les adresses des secondaires connectés. Comment cette détection est possible ?

Le primaire va tester toutes les adresses possibles et y envoyer une trame, si le bit suivant l'adresse dans la trame n'est pas positionné par le secondaire, il n'y a aucun équipement connecté à cette adresse.

Question 9.5.2 page 109 Est-ce que la norme une adresse qui permet de parler à tous les secondaires en même temps ?

La norme prévoit un *General Call* pour joindre tous les secondaires en utilisant l'adresse 0 (voir chapitre 3.1.12 du standard I2C)

Question 9.6.1 page 113 Que se passe-t-il si dans le programme `wifi_temperature.py` vous modifiez le pas de mesure ligne 36, pour le mettre par exemple à 60 secondes.

Le programme `display_server.py` doit être modifié également en ajoutant l'argument `period=60` à l'appel de `to_bbt`.

Question 11.2.1 page 138 Quel message de trace obtenez vous du LNS, si l'Objet n'est pas configuré avec le même **AppKey** que le LNS ?

```
12:52:11 Join-request to cluster-local Join Server failed MIC mismatch
12:52:01 Join-request to cluster-local Join Server failed MIC mismatch
12:51:51 Join-request to cluster-local Join Server failed MIC mismatch
```

Question 11.2.2 page 138 Quel impact sur les batteries ?

Le LoPy va émettre périodiquement tous les 10 secondes un message de type **Join-request**. Le but est pouvoir se connecter, même si la transmission n'est pas optimale. mais en cas de mauvaise configuration, cela va se traduire par une occupation plus importante du réseau, mais surtout un impact sur l'autonomie de l'Objet.

Question 11.2.3 page 138 Dans les traces précédentes, quelle est la valeur du fPort par défaut utilisé par le LoPy ?

```
"f_port": 2,
```

Question 11.2.4 page 138 L'instruction `bind` permet de modifier le **fPort** pour une transmission. Modifiez le programme pour utiliser le fPort 10 et vérifiez l'effet dans les traces de TTN.

Le programme doit être modifié par exemple après les instructions `setsockopt` en ajoutant la ligne :
`s.bind(10)`

Question 11.5.1 page 148 En vous aidant du lien suivant <https://www.thethingsnetwork.org/docs/lorawan/regional-parameters/> indiquant les paramètres régionaux. Est ce que le programme `lorawan_temperature.py` fonctionnerait sur un réseau LoRaWAN situé en Amérique du Nord ?

Non, la taille maximale des données pour DR0 est de 11 octets.

Question 12.2.1 page 160 Que représente ce message ?

- Une requête GET
- Une requête POST
- Une requête PUT
- Une requête DELETE
- Une notification positive

Le champ code (deuxième octets de l'en-tête CoAP) vaut 0x02, ou 0.02 donc il s'agit d'une requête et d'un POST.

Question 12.2.2 page 160 Quelle est la valeur du champ token ?

- Vide
- 0xb4
- 0xb474
- 0xb47465
- 0xb474656d

Le premier octet 0x40 s'écrit en binaire 0b01_00_0000, soit la version (1), le type (CON) et la taille du champ Token (0), il n'y a donc pas de Token après l'en-tête obligatoire, il y aura directement les options ou le séparateur 0xFF pour indiquer des données.

Question 12.2.3 page 160 Quel élément de l'URI contient ce message ?

- Aucun
- /temp
- /temp/sens1 et /max
- /temp/sens1/max
- /temp/sens1?max

La séquence des Tyle/Longueur est 0xb4 Uri-path avec 4 octets de données (temp), 0x05 toujours Uri-path avec 5 octets de données (sens1) et 0x43 donc un type 11+4=15 soit Uri-Query avec 3 octets de données (max). 0xff indique la fin des options

Question 12.2.4 page 161 Vous avez la ressource suivante :

temperature = 20C

Quelle valeur l'option CoAP Content-type utiliser pour une réponse en CoAP ?

- text/plain
- 0
- 50

L'entier 0 qui correspond à un codage utilisant les caractères ASCII

Question 12.2.5 page 162 Vous voulez recevoir une ressource dans le format SenML; CBOR. Quelle valeur doit transporter l'option Accept dans la requête ?

112, comme le montre le tableau 12.4 on page 161

Question 12.2.6 page 162 Quel code d'erreur retourne le serveur s'il ne peut pas envoyer une réponse dans ce format ? Aidez vous du [RFC 7252](#).

- 4.04 (Not Found)
- 4.02 (Bad Option)
- 4.06 (Not Acceptable)**
- 5.01 (Not Implemented))

Cf. Chapitre 5.10.4. du RFC. 4.06 n'est pas facile à trouver. Il faut aller sur le site de l'IANA, aller sur le RFC de CoAP qui pointe sur celui de HTTP pour la définition de ce code qui est rarement utilisé en HTTP. 4.04 n'est pas possible car la ressource existe mais pas au bon format. 4.02 n'est également pas possible, l'option Accept est critique donc doit être connue du serveur. Finalement, il s'agit d'une erreur du client et pas du serveur ; donc l'erreur 5.01 n'est pas possible non plus.

Question 13.2.1 page 172 Que se passe-t-il si vous utilisez une requête non confirmable pour demander la ressource /time (mettre l'argument type=CoAP.NON dans la construction de l'en-tête obligatoire).

- Ça plante le serveur.
- Le serveur répond par une requête Confirmable.
- Le serveur retourne un RST car nous n'avons pas programmé ce cas.
- Le serveur répond par une requête Non Confirmable.**
- On passe en heure d'hiver.

Question 13.2.2 page 173 Modifiez le programme du serveur pour supprimer le délai de 5 secondes avant une réponse.

Que se passe-t-il quand le client envoie une requête CONFirmable ?

- Le serveur retourne la réponse dans l'acquittement.**
- Le serveur retourne une requête NON confirmable.
- Le serveur attend quand même quelques secondes pour ne pas saturer le réseau.
- Le serveur enlève le token de la réponse..

Question 13.2.3 page 173 Modifiez le programme du serveur pour supprimer le délai de 5 secondes avant une réponse.

Que se passe-t-il quand le client envoie une requête NON confirmable ?

- Le serveur retourne la réponse dans l'acquittement.
- Le serveur retourne une requête NON confirmable.**
- Le serveur attend quand même quelques secondes pour ne pas saturer le réseau.
- Le serveur enlève le token de la réponse..

Question 13.3.1 page 177 Que reçoit-on en réponse à la requête POST ?

- un message ACK
- le statut 2.00.
- le statut 2.04 CHANGED.**
- rien.

A chaque requête on reçoit une notification REST indiquant que la ressource a été modifiée.

Question 13.3.2 page 177 Modifiez le programme client pour indiquer un content-format JSON. Quelle notification obtenez-vous ?

- un message RST
- le statut 4.04 NOT FOUND.
- le statut 2.15.
- le statut 5.00.

Si vous avez 5.00 c'est qu'il y a une erreur dans votre programme côté serveur. La notification pour un format inconnu est 4.15 (Unsupported Content-Format)

Question 14.3.1 page 185 Quelle est la nature de la requête émise par le client :

- GET
- POST
- PUT
- DELETE

Le client LwM2M agit comme un client REST et envoie cette requête POST :

```
Constrained Application Protocol, Confirmable, POST, MID:12142
01.. .... = Version: 1
..00 .... = Type: Confirmable (0)
.... 1000 = Token Length: 8
Code: POST (2)
Message ID: 12142
```

Question 14.3.2 page 186 Quelle est le chemin de l'URI :

- vide
- /rd
- /lwm2m

Le chemin d'URI est composé d'un seul élément :

Opt Name: 1: Uri-Path: rd

Question 14.3.3 page 186 Quelle est le format du contenu (content-format) :

- text
- XML
- JSON
- link-format
- CBOR

Après le chemin d'URI on trouve l'option Content-Format qui contient la valeur 40 soit link-format :

Opt Name: 2: Content-Format: application/link-format

Question 14.3.4 page 187 A quelle période voyez-vous d'afficher des messages CoAP ?

La figure 14.3 sur page 186 montre plusieurs échanges. Le premier à 25.16 secondes est un POST sur /rd. Le second à 78.33 secondes un POST sur /rd/dyUNwRXNOp. Le troisième à 131.37 secondes

est identique et les suivants sont identiques. La période d'envoi est de 53 secondes.

Question 14.3.5 page 187 Que signifie b=U ?

- Les données seront émises avec les unités (Units)
- Le référentiel des unités est la représentation Universelle
- **Le protocole sous-jacent est en mode datagramme (UDP)**
- Le client n'est pas référencé (Unreferenced)

Ce paramètre désigne le binding, U indique que CoAP sera transporté sur UDP. Le standard propose également d'autres modes, comme :

- T : TCP. CoAP prévoit également un mode de fonctionnement sur **TCP (RFC 8323)**. Le mode permet par exemple de traverser des réseaux qui restreignent l'usage d' **UDP** pour de soit disant raisons de sécurité ;
- S : SMS. LwM2M est défini par l'Open Mobile Alliance, il est logique d'inclure ce moyen de communication pour gérer un équipement connecté au réseau cellulaire ;
- N : **NON-IP**. Dans ce mode, les messages CoAP sont directement envoyés sur le niveau 2. Il est également connu sous le terme de Non IP Data Delivery (NIDD) dans les réseaux 4G.
- Nous nous basons sur la version 1.1 du standard, la révision 1.2 de LwM2M intègre également de LoRaWAN, MQTT (M), HTTP (H),...

Question 14.3.6 page 187 Que signifie lwm2m=1.1 ? Il s'agit...

- **de la version lwm2m du client**
- de la taille mémoire de l'implémentation (1.1 ko)
- de la version lwm2m du serveur

Il s'agit de la version du client.

Question 14.3.7 page 187 Que signifie lt=300 ?

- C'est le nombre maximal d'objets (less than 300)
- C'est la taille maximale d'un échange (300 octets)
- **C'est la durée de vie de l'enregistrement d'un objet (lifetime)**

C'est la durée de vie d'une valeur, si elle n'est pas rafraîchie avant ce délai par le client, elle disparaîtra du serveur.

Question 14.3.8 page 188 À quoi sert l'attribut rt ?

- **à donner la sémantique (i.e. comment interpréter) de la ressource.**
- à indiquer la taille de la ressource en faisant référence à lwm2m.
- à indiquer la version de lwm2m utilisée.
- à aider à déboguer les échanges.

Quand le serveur reçoit le POST du client, il est capable d'associer une représentation au chemin d'URI décrit. Le client et le serveur doivent avoir cette connaissance à priori.

Question 14.3.9 page 192 En vous aidant de la page de description des objets numériques et

des ressources¹, que représente l'URI *3301/0/5602* ?

- la température maximale du premier capteur
- l'humidité minimale du capteur 0
- la luminosité maximale du capteur 0**

En allant sur la page web, on trouve que *3301* correspond à la mesure de la luminosité (*Illuminance*). La description de cet objet numérique est identique à celle de l'objet numérique *temperature*; *3301/3/5602* indique la valeur maximale.

Question 14.3.10 page 192 Que représente l'URI *10340/0/2* ?

- la température en Fahrenheit de l'objet 10340
- la longitude dans des coordonnées GPS
- le statut actif ou non d'une caméra**

En allant sur la page web, on trouve que *10340* correspond à un objet numérique *Camera* défini par la société CloudMinds. La description de cet objet numérique montre que la ressource *2* correspond au status (*1 :Enabled, 0 :Disabled*.)

Question 14.3.11 page 192 Quelle URI permet d'accéder à l'élément contrôlant la variation lumineuse (*Dimmer*) d'un éclairage (*Light Control*) ? Quelle valeur maximale peut prendre cette ressource ?

3311/0/5851 La valeur maximale est de 100.

Question 14.3.12 page 192 Quel serait le chemin d'URI pour obtenir la fréquence d'un courant électrique sur la deuxième phase ?

3318/1/5700

Question 14.3.13 page 192 Dans le premier échange capturé :

```
</>;ct="60 110 112 11542 11543";rt="oma.lwm2m",
</1>;ver=1.1,
</1/0>,
</3>;ver=1.2,
</3/0>,
</6/0>,
</3303>;ver=1.1,
</3303/0>,
</3442/0>
```

1 : Device

3 : LwM2M Server

1. <http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html#resources>

6 : Location

3303 : Temperature

3442 : LwM2M v1.1 Test Object

Question 14.4.1 page 193 Que répond le serveur (cf. figure 14.6 on page 194) ?

- Il acquitte juste le message.
- Rien.
- Il renvoie une URI qui servira par la suite à identifier l'objet.**
- Il retourne les données de chiffrement des messages.

La réponse contient l'option **Location-Path** ce qui permet de préciser où la ressource fournie par le client a été référencée sur le serveur. Comme pour **URI-Path**, il s'agit d'une option qui peut se répéter. Dans l'exemple, figure 14.6 on page 194, l'URI fournie est donc /rd/dyUNwRXNOp.

Cette URI servira par la suite d'identifiant interne pour le client.

Question 14.4.2 page 193 Si on laisse la plate-forme fonctionner sans intervenir, que voit-on sur l'analyseur réseau ?

- Rien.
- Le capteur envoie des informations indiquant un changement de valeur mesurée.
- Le client envoie les valeurs mesurées même si elles n'ont pas changé.
- Le client envoie un message vide vers le serveur pour indiquer qu'il est toujours présent.**
- Le serveur envoie un message vide vers ses clients pour savoir s'ils sont toujours présents.

Toutes les 53 secondes, le client envoie un POST vers la ressource créée à l'enregistrement. Ce POST est vide. Il sert à indiquer au serveur que le serveur est toujours présent. Le serveur acquitte ce message indiquant au client qu'il est aussi accessible.

Question 14.5.1 page 197 Un click sur l'oeil avec la croix permet au serveur d'annuler l'Observe. Quel message est émis ?

Quand le serveur LwM2M reçoit un Observe pour une session qui a été annulée, il répond avec un message de type *ReSeT*, la valeur du champ *Message ID* permet au client LwM2M de savoir quelle session d'Observe annuler.

Question 14.5.2 page 197 Quel message est émis, si on clique sur une ressource de type EXE, comme la remise à zéro des valeurs min et max ?

Cela ne change rien au protocole, un POST sur l'URI de la ressource est émis par le serveur LwM2M.

Index

Symbols

<i>broker</i>	138
3GPP	27, 59
4G	59
5G	59
6LoWPAN	60, 62

A	
ABP	134
Accept	159, 161, 196
ACK	150, 152, 170, 196
adaptation layer	60
ADSL	16
AF_INET	121
AF_LORA	135
AF_SIGFOX	121
aiocoap	165
analyseur de spectre	116
aoicoap	181
API REST	92
AppEUI	132
appEUI	130
AppKey	130, 132, 137, 138, 208
Arduino	21

Array	69
AS	131
ASCII	39, 64
Atom	103

B

backend	115, 130
base64	66, 137
Beebotte	92
bincsii	64
BLE	60
Bluetooth	60
BME280	102, 112
bn	196, 197
boot.py	106
Bouygues Télécom	129

C

callback	124
CBOR	72, 73, 149, 161, 175
chirpstack	130
CLC	110
CoAP	61, 128
Code	151
coil	49

collision 116
 compression 60
 CON 150, 152, 154, 164, 170, 197
 Content-Format 159, 211
 Content-format 161, 175, 176, 179
 CRC 49, 51
 CSV 64, 149
 ct 188

D

Data Rate 147
 DCC 110
 DELETE 33, 151
 devAddr 134, 137
 devEUI 130, 137
 discrete input 49
 DR 135
 DTT 17
 Duty Cycle 135

E

electric counter 52
 Empty 169
 Epoch 94, 120
 ETag 159
 Ethernet 38, 45
 except 135

F

Flask 46, 126, 141
 forward_data 141, 143
 fPort 138, 145, 208
 fragmentation 60
 Fréquence 54
 FTP 106

G

GET 32, 124, 151
 GND 110

H

HEAD 32
 hexadecimal 36
 holding register 49
 horizontal 17
 hourglass 27
 HTML 24, 29, 68
 HTTP 21, 28, 29, 32, 38, 42, 61, 149, 150
 HTTPS 32, 62

I

I2C 110, 113
 IANA 151, 187
 IBAN 32, 200
 IEEE 27
 IEEE 754 52, 79
 IEEE 802.15.4 60, 62
 IEEE 802.3 45
 IETF 21, 23, 24
 If-Match 159
 If-None-Match 159
 ifconfig 107, 139
 input register 49
 Intensité 54
 IoT 18
 IP 27, 60, 200
 IPv4 200
 IPv6 28, 60, 62, 200
 IRI 30
 ISBN 31
 ISO 26
 ITU 200

J

java 183
 Javascript 69
 Join 134
 Join-accept 137
 Join-request 137, 208

JSON 28, 29, 62, 69, 71–73, 149, 161
JSON-LD 71

K

key 69
kpn_senml 113

L

layer 26
LCIM 22
Least Significant Bit 180
Leshan 183
link-format 187, 211
Linky 62
Linux 21, 103
LNS 21, 59, 129
Location-Path 159, 214
Location-Query 159
loopback 82, 107, 125, 141, 143, 184
LoPy4 102
LoRa 129
LoRaWAN 59, 103, 129
LPWAN 114
LPWANs 61
LwM2M 161, 183, 188

M

Mac OS 103
Matching_sent 180
Max-Age 159
mesh 58
Message ID 151
micro-python 102
Modbus 48, 108, 190
Module Python
 aiocoap
 Resource, 167
 TimeResource, 173
 aoicoap
 Message, 167

beebotte
 BBT, 93
 writeBulk, 95
binascii
 hexlify, 64, 66, 68, 142
 unhexify, 120
 unhexify, 64, 66
 unhexlify, 123
 unhexlify, 134

BME280
 read_raw_temp, 111
 read_temperature, 111

cbor2
 dumps, 73, 86, 176
 loads, 73, 79, 123, 177
CoAP
 add_option, 169
 add_payload, 174
 new_header, 169
 send_ack, 174

datetime
 now, 94
 timetuple, 94

Flask
 run, 47

json
 dumps, 71, 77, 86
 loads, 71, 86, 119

kpn_senml
 dumps, 113
 SenmlPack, 98
 SenmlRecord, 98

lora
 has_joined, 135
 join, 134, 135

machine
 scan, 109, 207

network
 LoRa, 130
 lora.mac(), 66
 Sigfox, 115, 121

pprint
 pprint, 71

pycom
 rgbled, 134

requests

HTTPBasicAuth	119
post	145
select	
select	142
socket	
bind	83, 138, 208
recv	136
recvfromm	143
recvfrom	83, 142
send	135
sendto	85, 143
setsockopt	135, 208
socket	115
str	
decode	66
encode	85
format	85
time	
maketime	94
MQTT	21, 59, 138

N

n	197
NAT	124, 139, 200
net-tools	107
NGW	59
NIDD	212
No response	178
No-Response	159
No-response	179
no_response	182
NON	150, 152–154, 164, 197
NON-IP	212
not_sent	180
NXP	108

O

Object	69
Observe	159, 162, 196, 197
OMA	161, 183, 188, 190
option	156
Orange	129

OTAA	134
------	-----

P

PAC	115
Packet Forwarder	146
PATCH	32
port	28
POST	32, 124, 151
Programmes micro-python	
BME280.py	111
CoAP	168
coap_empty_msg.py	168
coap_full_sensor.py	180, 181
coap_get_time.py	169
coap_get_time2.py	170, 171
coap_get_time3.py	171, 172
coap_get_time4.py	172
coap_port_temp1.py	174
coap_post_temp1.py	174
coap_post_temp2.py	176
coap_post_temp4.py	178
config.json	146
config_sigfox.py	118
lorawan_devEUI.py	130, 132, 134
lorawan_send_and_receive.py	132, 134, 136
lorawan_temperature.py	147, 148, 208
main.py	146
sending_client.py	108
sigfox_id.py	114, 115
sigfox_temperature.py	120, 121
wifi_conf.py	106, 146
wifi_temperature.py	112, 113, 120, 208
Programmes Python	
cbor-array.py	77
cbor-integer-ex1.py	73
cbor-integer-ex2.py	75
cbor-mapped.py	77
cbor-string.py	76
cbor-tag.py	79
coap_basic_server1.py	165, 167
coap_basic_server2.py	173, 174
coap_basic_server3.py	176
coap_post_temp4.py	178
coap_server.py	182

config_bbt.py	92	RFC	23
device_message.py	122	RFC 1918	200
device_messages.py	118, 119, 121	RFC 2396	158
display_receive_and_send.py	143, 148	RFC 3986.....	30
display_server.py	92, 93, 96, 113, 125, 127,	RFC 4944.....	60
	128, 148, 208	RFC 5988	188
display_sigfox.py	122, 124	RFC 6282.....	60
example_json.py.....	71	RFC 6690	187, 188
generic_coap_relay.py	181	RFC 7228	21
generic_relay.py ..	125–128, 139, 142–144,	RFC 7230.....	42
	146, 165	RFC 7231	41, 43, 202, 203
minimal_client1.py	85	RFC 7252	61, 153, 154, 162, 188, 210
minimal_client2.py	85	RFC 7641	162
minimal_client3.py	85	RFC 791	45
minimal_client4.py	86	RFC 8259	69, 70
minimal_client5.py	87	RFC 8323	212
minimal_client6.py	87	RFC 8376	59
minimal_humidity1.py.....	89, 90	RFC 8428	81, 97
minimal_humidity2.py.....	90, 91	RFC 8724	61, 179
minimal_senml_client.py.....	98	RFC 8949	72, 80, 206
minimal_senml_server.py	100	RNIPP	200
minimal_server.py	82, 86, 107	router.....	27
simple_server.py.....	47	RS-485.....	48, 50
ttn_config.py	143, 144	RSSI	137
virtual_sensor.py	84	RST	150, 169
Proxy-Scheme.....	159	rt.....	188
Proxy-Uri	159	RTT	154
publish/subscribe	33		
Puissance	54		
PUT	32, 151		
Pycom	102		
Pygate	145		
pymakr	103, 105		

Q

QModMaster	50
------------------	----

R

Raspberry Pi	21	SCEF	21, 59
ReSeT	164	SCHC	61
REST	29, 61, 68, 149	SCL	110
RESTfull	32	SDA	110
		SenML	97, 161, 196
		serialization	64
		Sigfox	59, 103, 114, 129, 138, 179
		silos	17
		Size1	159
		span	68
		spotify	31
		SSID	106
		star	58, 59, 61
		Storage integration.....	139

T

tag 68, 79
 TCP 28, 45, 61, 212
 telnet 106
 TKL 150, 156
 TLV 156, 161, 196
 to_btt 123
 Token 154, 164, 196
 topic 138
 try 135
 TTN 129
 Type 150

W

W3C 24, 68
 Webhooks 139
 Wi-Fi 45, 106
 Windows 103
 Wireshark 36, 54, 55, 125, 183
 WWW 16

X

XML 28, 68, 71, 190
 XY-MD02 49

U

UDP 28, 45, 62, 212
 UNB 116
 URI 29–32, 61, 68
 Uri-Host 159
 URI-Path 214
 Uri-Path 159, 170, 211
 Uri-path 160
 Uri-Port 159
 Uri-Query 159
 Uri-query 160
 URL 31, 46
 URN 31
 USB 50, 103

V

v 196
 ver 188
 vertical 17
 VIN 110
 virtual_sensor 84
 Voltage 54
 VPS 124, 125
 vs 197



Bibliography

- [TM03] Andreas TOLK et James A MUGUIRA. “The levels of conceptual interoperability model”. In : *Proceedings of the 2003 fall simulation interoperability workshop*. Tome 7. Citeseer. 2003, pages 1-11 (cf. page 22).