Internet of Things
IN 5 DAYS

Antonio Liñán Colina • Alvaro Vives • Marco Zennaro • Antoine Bagula • Ermanno Pietrosemoli

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1. About the Book

The "IoT in five days" book is in active development by a joint effort from both academia and industrial collaborators, as they acknowledge the Internet of Things of the future to be built on top of scalable and mature protocols, such as IPv6, 6LoWPAN and IEEE 802.15.4. Open Source Operative Systems as Contiki, with more than 10 years of history and actively supported by Universities and Research centers, has been paving the Internet of Things road since the early beginnings of Wireless Sensor Networks and M2M communication, to this new IoT paradigm.

The content of the book are Open Source as well, feedback and contribution is more than welcome! The sources are maintained in the following repositories:

- https://github.com/marcozennaro/IPv6-WSN-book
- https://github.com/alignan/IPv6-WSN-book

The book has been developed in asciidoc, it can be further compiled from its sources to HTML, PDF, eBook and others.

1.2. About the Authors

Antonio Liñán defines himself as *"an engineer at day, maker at night"* (he would do both for free). He has more than 8 years and more than 20 projects of experience in Wireless Sensor Networks (WSN), Internet of Things (IoT) applications and embedded firmware development; working in Zolertia as both senior R+D engineer and CTO, buf if you ask him he just *"make things blink and chat"*. In his free time he's normally engaged in Coursera, collecting hardware platforms, dwelling in hackathons or preaching about GIT. He has a Master at the University of Los Andes (Colombia), it has worked in European Projects related to Smart Cities, Internet of Things and Security, and currently is a prominent contributor in severals Open Source communities, such as Contiki.

Alvaro Vives loves technology, problem solving, learning and teaching. Doing these things he has become a consultant, a network and systems engineer, and trainer. As a consultant, he has worked on projects in several countries, at ISPs, content providers, public organizations and enterprises. As a trainer since 2006 has given more than 46 workshops in 18 countries for ISPs, content providers, public organizations, enterprises and in events like LACNIC/

LACNOG, SANOG, WALC, and ESNOG. As network and systems administrator he has been in charge of production networks and services in several companies using different technologies and vendors. At present, he is working with WSN and IoT as a consequence of the convergence of IPv6 and IoT.

Antoine Bagula obtained his doctoral degree in 2006 from the KTH-Royal Institute of Technology in Sweden. Currently holds a lecturing positions at the University of Stellenbosch and the University of Cape Town before joining the Computer Science department at the University of the Western Cape in January 2014. Since 2006, He is a frequent consultant of the UNESCO through its International Centre for Theoretical Physics in Trieste-Italy, the World Bank and other international organizations on different telecommunication projects. He is senior associate of the UNESCO International Centre for Theoretical Physics of Trieste/Italy and have been actively involved in the centre training programme on wireless networking. His research interest is Computer Networking with a specific focus on the Internet-of-Things, Cloud Computing, Network security and Network protocols for wireless and wired mesh networks and hybrid networks

Marco Zennaro received his M.Sc. Degree in Electronic Engineering from University of Trieste in Italy. He defended his PhD thesis on “Wireless Sensor Networks for Development: Potentials and Open Issues” at KTH-Royal Institute of Technology, Stockholm, Sweden. His research interest is in ICT4D, the use of ICT for Development. In particular, he is interested in Wireless Networks and in Wireless Sensor Networks in developing countries. He has been giving lectures on Wireless technologies in more than 20 countries.

When not traveling, he is the editor of the wsnblog.com[1]. He is coauthor of the book “Wireless Networking for the Developing World”.

Ermanno Pietrosemoli is currently a researcher at the Telecommunications/ICT for Development Lab of the International Centre for Theoretical Physics in Trieste, Italy, and President of Fundación Escuela Latinoamericana de Redes “EsLaRed”, a non-profit organization that promotes ICT in Latin America through training and development projects. EsLaRed was awarded the 2008 Jonathan B.Postel Service Award by the Internet Society.

Ermanno has been deploying wireless data communication networks focusing on low cost technology, and has participated in the planning and building of wireless data networks in Argentina, Colombia, Ecuador, Italy, Lesotho, Malawi, Mexico, Morocco, Nicaragua, Peru, Trinidad, U.S.A. and Venezuela. He has presented in many conferences and published several papers related to wireless data communication and is coauthor and technical reviewer

[1] http://wsnblog.com/

of the book “Wireless Networking for the Developing World” freely available from http://wndw.net[2]. Ermanno holds a Master’s Degree from Stanford University and was Professor of Telecommunications at Universidad de los Andes in Venezuela from 1970 to 2000.

1.3. License

[2] http://wndw.net/

Chapter 2. The Internet of Things (IoT)

Building upon a complex network connecting billions of devices and humans into a multi-technology, multi-protocol and multi-platform infrastructure, the Internet-of-Things (IoT) main vision is to create an intelligent world where the real, the digital and the virtual are converging to create smart environments that provide more intelligence to the energy, health, transport, cities, industry, buildings and many other areas of our daily life.

The expectation is that of interconnecting millions of islands of smart networks enabling access to the information not only “anytime” and “anywhere” but also using “anything” and “anyone” ideally through any “path”, “network” and “any service”. This will be achieved by having the objects that we manipulate daily to be outfitted with sensing, identification and positioning devices and endowed with an IP address to become smart objects, capable of communicating with not only other smart objects but also with humans with the expectation of reaching areas that we could never reach without the advances made in the sensing, identification and positioning technologies.

While being globally discoverable and queried, these smart objects can similarly discover and interact with external entities by querying humans, computers and other smart objects. The smart objects can also obtain intelligence by making or enabling context related decisions by taking advantage of the available communication channels to provide information about themselves while also accessing information that has been aggregated by other smart objects.

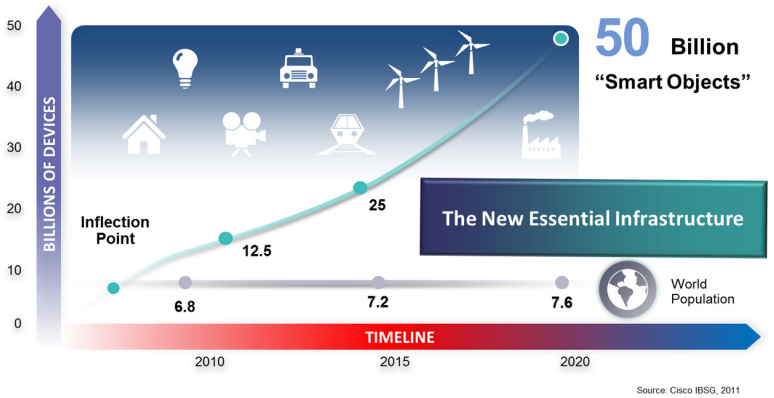

Figure 2.1. Internet-connected devices and the future evolution (Source: Cisco, 2011)

As revealed by Figure 1, the IoT is the new essential infrastructure which is predicted to connect 50 billion of smart objects in 2020 when the world population will reach 7.6 billion. As suggested by the ITU, such essential infrastructure will be built around a multi-layered architecture where the smart objects will be used to deliver different services through the four main layers depicted by Figure 2: a device layer, a network layer, a support layer and the application layer.

In the device layer lie devices (sensors, actuators, RFID devices) and gateways used to collect the sensor readings for further processing while the network layer provides the necessary transport and networking capabilities for routing the IoT data to processing places. The support layer is a middleware layer that serves to hide the complexity of the lower layers to the application layer and provide specific and generic services such as storage in different forms (database management systems and/or cloud computing systems) and many other services such as translation.

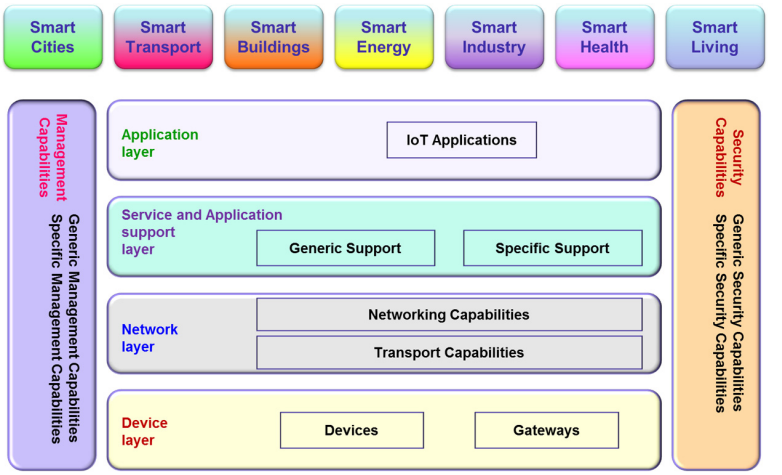

Figure 2.2. IoT Layered Architecture (Source: ITU-T)

As depicted by Figure 3, the IoT can be perceived as an infrastructure driving a number of applications services which are enabled by a number of technologies. Its application services expand across many domains such as smart cities, smart transport, smart buildings, smart energy, smart industry and smart health while it is enabled by different technologies such as sensing, nanoeletronics, wireless sensor network (wsn), radio frequency identification (RFID), localization, storage and cloud. The IoT systems and applications are designed to provide security, privacy, safety, integrity, trust, dependability, transparency, anonymity and are bound by ethics constraints.

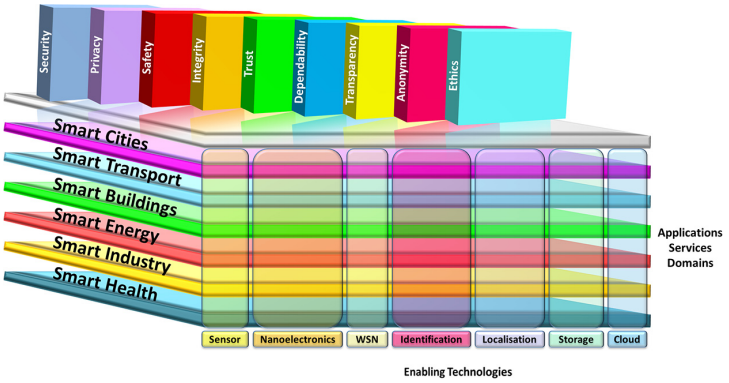

Figure 2.3. IoT 3_Dimensional View (Source: [IoT])

Experts say we are heading towards what can be called a "ubiquitous network society", one in which networks and networked devices are omnipresent. RFID and wireless sensors promise a world of networked and interconnected devices that provide relevant content and information whatever the location of the user. Everything from tires to toothbrushes will be in communications range, heralding the dawn of a new era, one in which today's Internet (of data and people) gives way to tomorrow's Internet of Things.

At the dawn of the Internet revolution, users were amazed at the possibility of contacting people and information across the world and across time zones. The next step in this technological revolution (connecting people any-time, anywhere) is to connect inanimate objects to a communication network. This vision underlying the Internet of things will allow the information to be accessed not only "anytime" and "anywhere" but also using "anything". This will be facilitated by using WSNs and RFID tags to extend the communication and monitoring potential of the network of networks, as well as the introduction of computing power in everyday items such as razors, shoes and packaging.

WSNs are an early form of ubiquitous information and communication networks. They are one of building blocks of the Internet of things.

2.1. Wireless Sensor Networks

A Wireless Sensor Network (WSN) is a self-configuring network of small sensor nodes (so-called motes) communicating among them using radio signals, and deployed in quantity to sense the physical world. Sensor nodes are essentially small computers with extremely basic functionality. They consist of a processing unit with limited computational power and a limited memory, a radio communication device, a power source and one or more sensors.

Motes come in different sizes and shapes, depending on their foreseen use. They can be very small, if they are to be deployed in big numbers and need to have little visual impact. They can have a rechargeable battery power source if they are to be used in a lab. The integration of these tiny, ubiquitous electronic devices in the most diverse scenarios ensures a wide range of applications. Some of the application areas are environmental monitoring, agriculture, health and security.

In a typical application, a WSN is scattered in a region where it is meant to collect data through its sensor nodes. These networks provide a bridge between the physical world and the virtual world. They promise unprecedented abilities to observe and understand large scale, real-world phenomena at a fine spatio-temporal resolution. This is so because one deploys sensor nodes in large numbers directly in the field, where the experiments take place. All motes are composed of five main elements as shown below:

1. Processor: the task of this unit is to process locally sensed information and information sensed by other devices. At present the processors are limited in terms of computational power, but given Moore's law, future devices will come in smaller sizes, will be more powerful and consume less energy. The processor can run in different modes: sleep is used most of the time to save power, idle is used when data can arrive from other motes, and active is used when data is sensed or sent to / received from other motes.
2. Power source: motes are meant to be deployed in various environments, including remote and hostile regions so they must use little power. Sensor nodes typically have little energy storage, so networking protocols must emphasize power conservation. They also must have built-in mechanisms that allow the end user the option of prolonging network lifetime at the cost of lower throughput. Sensor nodes may be equipped with effective power scavenging methods, such as solar cells, so they may be left unattended for months, or years. Common sources of power are rechargeable batteries, solar panels and capacitors.
3. Memory: it is used to store both programs (instructions executed by the processor) and data (raw and processed sensor measurements).
4. Radio: WSN devices include a low-rate, short-range wireless radio. Typical rates are 10-100 kbps, and range is less than 100 meters. Radio communication is often the most power-intensive task, so it is a must to incorporate energy-efficient techniques such as wake-up modes. Sophisticated algorithms and protocols are employed to address the issues of lifetime maximization, robustness and fault tolerance.
5. Sensors: sensor networks may consist of many different types of sensors capable of monitoring a wide variety of ambient conditions. Table 1 classifies the three main categories of sensors based on field-readiness and scalability. While scalability reveals if the sensors are small and inexpensive enough to scale up to many distributed systems,

the field-readiness describes the sensor's engineering efficiency with relation to field deployment. In terms of the engineering efficiency, Table1 reveals high field-readiness for most physical sensors and for a few numbers of chemical sensors while most chemical sensors lie in the medium and low levels, while biological sensors have low field-readiness.

Sensor Category	Parameter	Field-Readiness	Scalability
Physical	Temperature	High	High
	Moisture Content	High	High
	Flow rate, Flow velocity	High	Med-High
	Pressure	High	High
	Light Transmission (Turb)	High	High
Chemical	Dissolved Oxygen	High	High
	Electrical Conductivity	High	High
	pH	High	High
	Oxydation Reduction Potential	Medium	High
	Major Ionic Species (Cl-, Na+)	Low-Medium	High
	Nutrientsa (Nitrate, Ammonium)	Low-Medium	Low-High
	Heavy metals	Low	Low
	Small Organic Compounds	Low	Low
	Large Organic Compounds	Low	Low
Biological	Microorganisms	Low	Low
	Biologically active contaminants	Low	Low

Common applications include the sensing of temperature, humidity, light, pressure, noise levels, acceleration, soil moisture, etc. Due to bandwidth and power constraints, devices

primarily support low-data-units with limited computational power and limited rate of sensing. Some applications require multi-mode sensing, so each device may have several sensors on board.

Following is a short description of the technical characteristics of WSNs that make this technology attractive.

1. **Wireless Networking**: motes communicate with each other via radio in order to exchange and process data collected by their sensing unit. In some cases, they can use other nodes as relays, in which case the network is said to be multi-hop. If nodes communicate only directly with each other or with the gateway, the network is said to be single-hop. Wireless connectivity allows to retrieve data in real-time from locations that are difficult to access. It also makes the monitoring system less intrusive in places where wires would disturb the normal operation of the environment to monitor. It reduces the costs of installation: it has been estimated that wireless technology could eliminate up to 80 % of this cost.
2. **Self-organization**: motes organize themselves into an ad-hoc network, which means they do not need any pre-existing infrastructure. In WSNs, each mote is programmed to run a discovery of its neighborhood, to recognize which are the nodes that it can hear and talk to over its radio. The capacity of organizing spontaneously in a network makes them easy to deploy, expand and maintain, as well as resilient to the failure of individual points.
3. **Low-power**: WSNs can be installed in remote locations where power sources are not available. They must therefore rely on power given by batteries or obtained by energy harvesting techniques such as solar panels. In order to run for several months of years, motes must use low-power radios and processors and implement power efficient schemes. The processor must go to sleep mode as long as possible, and the Medium-Access layer must be designed accordingly. Thanks to these techniques, WSNs allow for long-lasting deployments in remote locations.

2.2. Applications of WSN

The integration of these tiny, ubiquitous electronic devices in the most diverse scenarios ensures a wide range of applications. Some of the most common application areas are environmental monitoring, agriculture, health and security. In a typical application, a WSN include:

1. Tracking the movement of animals. A large sensor network has been deployed to study the effect of micro climate factors in habitat selection of sea birds on Great Duck Island in Maine, USA. Researchers placed their sensors in burrows and used heat to detect

the presence of nesting birds, providing invaluable data to biological researchers. The deployment was heterogeneous in that it employed burrow nodes and weather nodes.

2. Forest fire detection. Since sensor nodes can be strategically deployed in a forest, sensor nodes can relay the exact origin of the fire to the end users before the fire is spread uncontrollable. Researchers from the University of California, Berkeley, demonstrated the feasibility of sensor network technology in a fire environment with their FireBug application.
3. Flood detection. An example is the ALERT system deployed in the US. It uses sensors that detect rainfall, water level and weather conditions. These sensors supply information to a centralized database system.
4. Geophysical research. A group of researchers from Harvard deployed a sensor network on an active volcano in South America to monitor seismic activity and similar conditions related to volcanic eruptions.
5. Agricultural applications of WSN include precision agriculture and monitoring conditions that affect crops and livestock. Many of the problems in managing farms to maximize production while achieving environmental goals can only be solved with appropriate data. WSN can also be used in retail control, particularly in goods that require being maintained under controlled conditions (temperature, humidity, light intensity, etc) [SusAgri].
6. An application of WSN in security is predictive maintenance. BP's Loch Rannoch project developed a commercial system to be used in refineries. This system monitors critical rotating machinery to evaluate operation conditions and report when wear and tear is detected. Thus one can understand how a machine is wearing and perform predictive maintenance. Sensor networks can be used to detect chemical agents in the air and water. They can also help to identify the type, concentration and location of pollutants.
7. An example of the use of WSN in health applications is the Bi-Fi, embedded system architecture for patient monitoring in hospitals and out-patient care. It has been conceived at UCLA and is based on the SunSPOT architecture by Sun. The motes measure high-rate biological data such as neural signals, pulse oximetry and electrocardiographs. The data is then interpreted, filtered, and transmitted by the motes to enable early warnings.

2.3. Roles in a WSN

Nodes in a WSN can play different roles.

1. Sensor nodes are used to sense their surroundings and transmit the sensor readings to a sink node, also called "base station". They are typically equipped with different kinds of sensors. A mote is endowed with on-board processing, communication capabilities and sensing capabilities.

2. Sink nodes or "base stations" are tasked to collect the sensor readings of the other nodes and pass these readings to a gateway to which they are directly connected for further processing/analysis. A sink node is endowed with minimal on-board processing and communication capabilities but does not have sensing capabilities.
3. Actuators are devices which are used to control the environment, based on triggers revealed by the sensor readings or by other inputs. An actuator may have the same configuration as a mote but it is also endowed with controlling capabilities, for example to switch a light on under low luminosity.

Gateways often connected to sink nodes, are usually fed by a stable power supply since they consume considerable energy. These devices are normal computing devices such as laptops, notebooks, desktops, mobile phones or other emerging devices which are able to store, process and route the sensor readings to the processing place. However, they may not be endowed with sensing capabilities. Being range-limited, sensor motes require multi-hop communication capabilities to allow: 1) spanning distances much larger than the transmission range of a single node through localized communication between neighbor nodes 2) adaptation to network changes, for example, by routing around a failed node using a different path in order to improve performance and 3) using less transmitter power as a result of the shorter distance to be spanned by each node. They are deployed in three forms : (1) Sensor node used to sense the environment (2) Relay node used as relay for the sensor readings received from other nodes and (3) Sink node also often called base station which is connected to a gateway (laptop, tablet, iPod, Smart phone, desktop) with higher energy budget capable of either processing the sensor readings locally or to transmit these readings to remote processing places.

Chapter 3. Introduction to IPv6

IPv6 stands for Internet Protocol version 6, so the importance of IPv6 is implicit in its name, it's as important as Internet! The Internet Protocol (IP from now on) was intended as a solution to the need to interconnect different data networks, and has become the "de facto" standard for all kinds of digital communications. Nowadays IP is present in all devices that are able to send and receive digital information, not only the Internet.

IP is standardized by the IETF (Internet Engineering Task Force), the organization in charge of all the Internet standards, guaranteeing the interoperability among different vendor's software. The fact that IP is a standard is of vital importance, because today everything is getting connected to the Internet where IP is used. All available Operating Systems and networking libraries have IP available to send and receive data. Included in this "everything-connected-to-Internet" is the IoT, so now you know why you are reading this chapter about IPv6, the last version of the Internet Protocol. In other words, today, the easiest way to send and receive data is using the standards used in the Internet, including the IP .

The objectives of this chapter are:

- Briefly describe the history of the Internet Protocol.
- Find out what IPv6 is used for.
- Get the IPv6 related concepts needed to understand the rest of the book.
- Provide a practical overview of IPv6, including addresses and a glimpse of how an IPv6 network looks like.

3.1. A little bit of History

ARPAnet was the first attempt of the US Department of Defense (DoD) to devise a decentralized network that was more resilient to an attack, while able to interconnect completely different systems. ARPAnet was created in the seventies, but it was in 1983 when a new brand protocol stack was introduced, the TCP/IP. The first widely used network protocol for this purpose was called IPv4 (Internet Protocol version 4) which gave rise to the civilian Internet. Initially only research centers and Universities were connected, supported by the NSF (National Science Foundation), and commercial applications where not allowed, but when the network started growing exponentially the NSF decided to transfer its operation and funding to private operators, and restrictions to commercial traffic where lifted. While the main

applications where email and file transfers, it was with the development of the World Wide Web HTML and specifically with the MOSAIC graphic interface browser and its succesors that the traffic really exploded and the Internet began to be used by the masses. As a consequence there was a rapid depletion in the number of IP addresses available under IPv4, which was never designed to scale to these levels.

In order to have more addresses, you need more bits, which means a longer IP address, which means a new architecture, which means changes to all of the routing and network software. After examining a number of proposals, the IETF settled on IPv6, recommended in January 1995 in RFC 1752, sometimes also referred to as the Next Generation Internet Protocol, or IPng. The IETF updated the IPv6 standard in 1998 with the current definition included in RFC 2460. By 2004, IPv6 was widely available from industry and supported by most new network equipment. Today IPv6 coexist with IPv4 in the Internet and the amount of IPv6 traffic is quickly growing as more and more ISPs and content providers have started to make IPv6 available.

As you can see, the history of IP and Internet are almost the same, and because of this the growth of Internet is been hampered by the limitations of IPv4, and has led to the development of a new version of IP, IPv6, as the protocol to be used to interconnect all sorts of devices to send and/or receive information. There are even some technologies that are being developed only with IPv6 in mind, a good example in the context of the IoT is 6LowPAN.

From now on we will only center on IPv6. If you know something about IPv4, then you have half the way done, if not, don't worry we will cover the main concepts briefly and gently.

3.2. IPv6 Concepts

We will cover the basics of IPv6, the minimum you need to know about the last version of the Internet Protocol to understand why it's so useful for the IoT and how it's related with other protocols like 6LowPAN covered later in this book. It would be useful to have the basic networking concepts, and be familiar with bits, bytes, networking stack, network layer, packets, IP header, etc. In any case, things will be explained from scratch to allow everybody to understand IPv6 and its features. You should understand that IPv6 is a different protocol, non-compatible with IPv4.

In the following figure we represent the layered model used in the Internet.

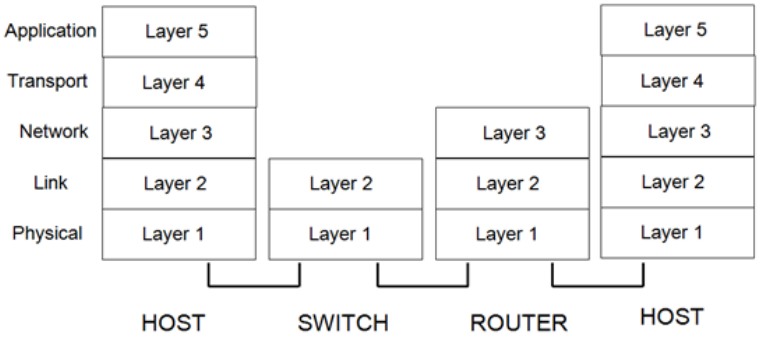

Figure 3.1. Internet Protocol stack

IPv6 operates in layer 3, also called network layer. The pieces of data handled by layer 3 are called packets. Devices connected to the Internet can be hosts or routers. A host can be a PC, a laptop or a sensor board, sending and/or receiving data packets. Hosts will be the source or destination of the packets. Routers instead are in charge of packet forwarding, and are responsible of choosing the next router that will forward them towards the final destination. Internet is composed of a lot of interconnected routers, which receive data packets in one interface and send then as quick as possible using another interface towards another forwarding router.

3.2.1. IPv6 packet

The first thing you have to know is what an IPv6 packet looks like. In the layered model we saw before, each layer introduces its own information in the packet, and this information is intended for, and can only be processed by the same layer in other IP device. This "conversation" between layers at the same level on different devices is known as a protocol.

The layer defined are:

- **Application**: Here resides the software developed by programmers, that will use network services offered by the network stack. An example is the web browser that opens a network connection towards a web server. Another example is the web server software that runs in a server somewhere in the Internet waiting to answer request from client's browsers. Examples of application protocols are HTTP or DNS.
- **Transport**: Is a layer above the network layer that offers services beyond the basic ones offered by the network layer, for example, retransmit lost packets or guarantee the ordered delivery of the packets, in the same order they have been sent. This layer will be the one that shows a "network service" to the application layer, a service they can use to send or receive data. Two are the most used transport protocols used in Internet, TCP and UDP.

- **Network**: This is the layer in charge of the correct delivery of the data received from the transport layer to its destination, as well as the reception of the received data from the link layer in the data destination. In Internet there is only one network protocol, the IP. Source and destination are identified by means of the IP addresses.
- **Link**: Link layer is in charge of sending and receiving frames, a collection of bytes sent/received from the network layer, in the scope of a local area network or LAN. This layer has its own addresses that will change depending on the technology used.
- **Physical**: This layer is in charge of all the electromagnetic signals, codifications, etc. needed for the digital information to go from one node to the next one. All physical media are included, both wired and wireless.

The following figure illustrates the idea that each of the above layers receive some bytes and add some specific information for that layer to be processed in other host. In the figure data is being send, starting in the application layer and being sent to another node in the physical layer.

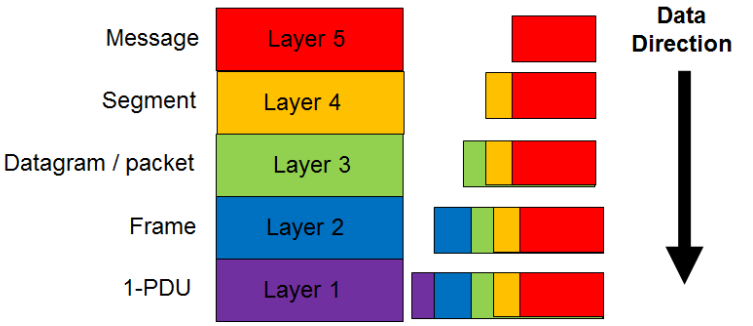

Figure 3.2. Data flow in the protocol stack

If we focus on the network layer, specifically in IPv6, the bytes sent and received in the IP packet have an standardized format. The following figure shows the basic IPv6 header:

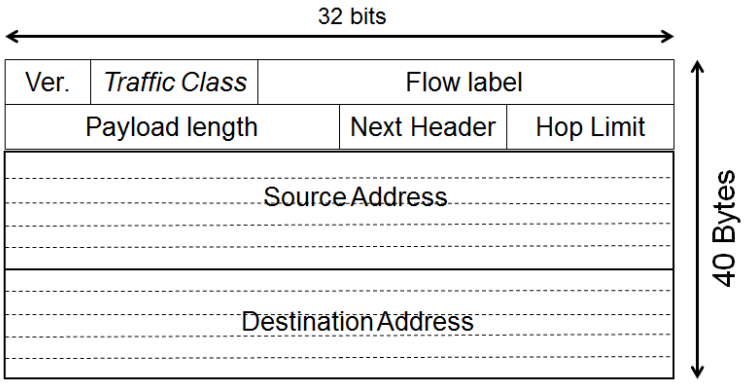

Figure 3.3. IPv6 Header

First you have the **basic IPv6 header** with a fixed size of 40 bytes, followed by upper layer data and optionally by some extension headers, that will be covered later. As you can see there are several fields in the packet header, with some improvements as compared with IPv4 header:

- The number of fields have been reduced from 12 to 8.
- The basic IPv6 header has a fixed size of 40 bytes and is aligned with 64 bits, allowing a faster hardware-based packet forwarding on routers.
- The size of addresses increased from 32 to 128 bits.

The most important fields are the source and destination addresses. As you already know, every IP device has a unique IP address that identifies it in the Internet. This IP address is used by routers to take their forwarding decisions.

IPv6 header has 128 bits for each IPv6 address, this allows for 2^{128} addresses (approximately 3.4×10^{38},i.e., 3.4 followed by 38 zeroes), compared with IPv4 that have 32 bits to encode the IPv4 address allowing for 2^{32} addresses (4,294,967,296).

We have seen the basic IPv6 header, and mentioned the **extension headers**. To keep the basic header simple and of a fixed size, additional features are added to IPv6 by means of extension headers.

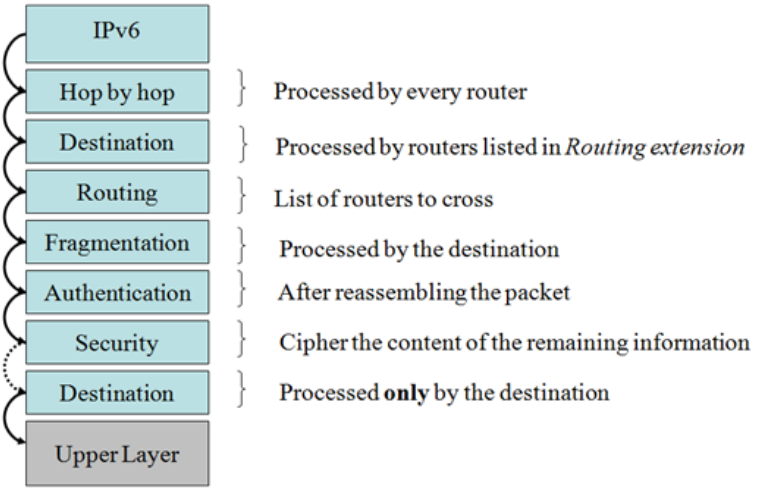

Figure 3.4. IPv6 Extension headers

Several extension headers have been defined, as you can see in the previous figure, and they have to follow the order shown. Extensions headers:

- Provide flexibility, for example, to enable security by ciphering the data in the packet.
- Optimize the processing of the packet, because with the exception of the hop by hop header, extensions are processed only by end nodes, (source and final destination of the packet), not by every router in the path.
- They are located as a "chain of headers" starting always in the basic IPv6 header, that use the field next header to point to the following extension header.

3.2.2. IPv6 addressing

As can be seen in the basic IPv6 header figure, both the source address and destination address fields have 128 bits to codify an IPv6 address. The use of 128 bits for addresses brings some benefits:

- Provide much more addresses, to satisfy current and future needs, with ample space for innovation.
- Easy address auto-configuration mechanisms.
- Easier address management/delegation.
- Room for more levels of hierarchy and for route aggregation.
- Ability to do end-to-end IPsec.

IPv6 addresses are classified into the following categories (these categories also exist for IPv4):

- **Unicast** (one-to-one): used to send a packet from one source to one destination. Are the commonest ones and we will talk more about them and the sub-classes that exist.
- **Multicast** (one-to-many): used to send a packet from one source to several destinations. This is possible by means of multicast routing that enable packets to replicate in some places.
- **Anycast** (one-to-nearest): used to send a packet from one source the nearest destination from a set of them.
- **Reserved**: Addresses or groups of them that have special uses defined, for example addresses to be used on documentation and examples.

Before entering into more detail about IPv6 addresses and the types of unicast addresses, let's see how do they look like and what are the notation rules. You need to have them clear because probably the first problem you will find in practice when using IPv6 is how to write an address.

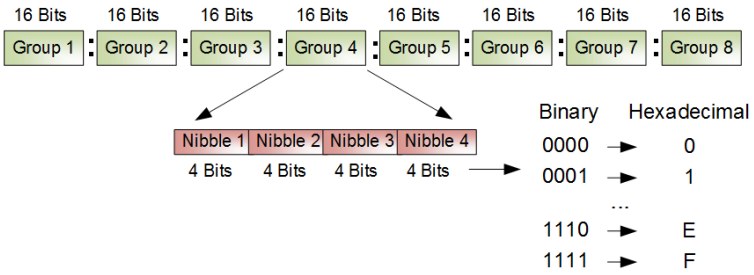

Figure 3.5. IPv6 address

IPv6 addresses notation rules are:

- 8 Groups of 16 bits separated by ":".
- Hexadecimal notation of each nibble (4 bits).
- Non case sensitive.
- Network Prefixes (group of addresses) are written Prefix / Prefix Length, i.e., prefix length indicate the number of bits of the address that are fixed.
- Leftmost zeroes within each group can be eliminated.
- One or more all-zero-groups can be substituted by "::". This can be done only once.

The first three rules tell you the basis of IPv6 address notation. They use hexadecimal notation, i.e., numbers are represented by sixteen symbols between 0 and F. You will have eight groups of four hexadecimal symbols, each group separated by a colon ":". The last two rules are for address notation compression, we will see how this works with some examples.

Let's see some examples:

1) If we represent all the address bits we have the preferred form, for example: `2001:0db8:4004:0010:0000:0000:6543:0ffd`

2) If we use squared brackets around the address we have the literal form of the address: `[2001:0db8:4004:0010:0000:0000:6543:0ffd]`

3) If we apply the fourth rule, allowing compression within each group by eliminating leftmost zeroes, we have: `2001:db8:4004:10:0:0:6543:ffd`

4) If we apply the fifth rule, allowing compression of one or more consecutive groups of zeroes using "::", we have: `2001:db8:4004:10::6543:ffd`

Care should be taken when compressing and decompressing IPv6 addresses. The process should be reversible. It's very common to have some mistakes. For example, the following address `2001:db8:A:0:0:12:0:80` could be compressed even more using "::". we have two options:

a) `2001:db8:A::12:0:80` b) `2001:db8:A:0:0:12::80`

Both are correct IPv6 addresses. The following address `2001:db8:A::12::80` is wrong, as it does not follow the last compression rule we saw above. The problem with this badly compressed address is that we can't be sure how to expand it, its ambiguous. We can't know if it expands to `2001:db8:A:0:12:0:0:80` or to `2001:db8:A:0:0:12:0:80`.

3.2.3. IPv6 network prefix

Last but not least you have to understand the concept of a **network prefix**, that indicates some fixed bits and some non-defined bits that could be used to create new sub-prefixes or to define complete IPv6 addresses assigned to hosts.

Let's see some examples:

1) The network prefix `2001:db8:1::/48` (the compressed form of `2001:0db8:0001:0000:0000:0000:0000:0000`) indicates that the first 48 bits will always be the same (`2001:0db8:0001`) but that we can play with the other

80 bits, for example, to obtain two smaller prefixes: `2001:db8:1:a::/64` and `2001:db8:1:b::/64`.

2) If we take one of the smaller prefixes defined above, `2001:db8:1:b::/64`, where the first 64 bits are fixed we have the rightmost 64 bits to assign, for example, to an IPv6 interface in a host: `2001:db8:1:b:1:2:3:4`. This last example allow us to introduce a basic concept in IPv6: **In a LAN (Local Area Network) a /64 prefix is always used**. **The rightmost 64 bits, are called the interface identifier (IID) because they uniquely identify a host's interface in the local network defined by the /64 prefix**. The following figure illustrates this statement:

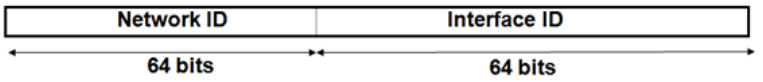

Figure 3.6. Network and Interface ID

Now that you have seen your first IPv6 addresses we can enter into more detail about two types of addresses you will find when you start working with IPv6: reserved and unicast.

The following are some reserved or special purpose addresses:

- The **unspecified address**, used as a placeholder when no address is available: `0:0:0:0:0:0:0:0` (::/128)
- The **loopback address**, for an interface sending packets to itself: `0:0:0:0:0:0:0:1` (::1/128)
- **Documentation Prefix**: `2001:db8::/32`. This prefix is reserved to be used in examples and documentation, you have already seen it in this chapter.

As specified in [RFC6890] there is a registry maintained by IANA about special purpose IPv6 addresses [IANA-IPV6-SPEC].

The following are some types of unicast addresses [RFC4291]:

- **Link-local**: Link-local addresses are always configured in any IPv6 interface that is connected to a network. They all start with the prefix `FE80::/10` and can be used to communicate with other hosts on the same local network, i.e., all hosts connected to the same switch. They cannot be used to communicate with other networks, i.e., to send or receive packets through a router.
- **ULA** (Unique Local Address) [RFC4193]: All ULA addresses start with the prefix FC00::/7, what means in practice that you could see `FC00::/8` or `FD00::/8`. Intended for local communications, usually inside a single site, they are not expected to be routable on

the Global Internet but routable inside a more limited network managed by the same organization.

- **Global Unicast**: Equivalent to the IPv4 public addresses,they are unique in the whole Internet and could be used to send a packet from anywhere in the Internet to any other destination in Internet.

3.3. What is IPv6 used for?

As we have seen IPv6 has some features that facilitates things like global addressing and hosts address autoconfiguration. Because IPv6 provides as much addresses as we may need for some hundred of years, we can put a global unicast IPv6 address on almost anything we may think of. This brings back the initial Internet paradigm that every IP device could communicate with every IP device. This end-to-end communication allow for bidirectional communication all over the Internet and between any IP device, which could result in collaborative applications and new ways of storing, sending and accessing the information.

In the context of this book we can, for example, think on IPv6 sensors all around the world collecting, sending and being accessed from different places to create a world-wide mesh of physical values measured, stored and processed.

The availability of a huge amount of addresses has allowed a new mechanism called **stateless address autoconfiguration** (SLAAC) that didn't exist with IPv4. Following is a brief summary of the ways you can configure an address on an IPv6 interface:

- **Statically**: You can decide which address you will give to your IP device and then manually configure it into the device using any kind of interface: web, command line, etc. Commonly you also have to configure other network parameters like the gateway to use to send packets out of your network.
- **DHCPv6** (Dynamic Host Configuration Protocol for IPv6) [RFC3315]: A similar mechanism already existed for IPv4 and the idea is the same. You need to configure a dedicated server that after a brief negotiation with the IP device assigns an IP address to it. DHCPv6 allows IP devices to be configured automatically, this is why it is named stateful address autoconfiguration, because the DHCPv6 server maintains a state of assigned addresses.
- **SLAAC**: Stateless address autoconfiguration [RFC4862] is a new mechanism introduced with IPv6 that allows to configure automatically all network parameters on an IP device using the router that gives connectivity to a network.

The advantage of SLAAC is that it simplifies the configuration of "dumb" devices, like sensors, cameras or any other device with low processing power. You don't need to use any interface

in the IP device to configure anything, just "plug and net". It also simplifies the network infrastructure needed to build a basic IPv6 network, because you don't need additional device/server, you use the same router you need to send packets outside your network to configure the IP devices. We are not going to enter into details, but you just need to know that in a local network, usually called a LAN (Local Area Network), that is connected to a router, this router is in charge of sending all the information needed to its hosts using an RA (Router Advertisement) message. The router will send RAs periodically, but in order to expedite the process, a host can send an RS (Router Solicitation) message when its interface gets connected to the network. The router will send an RA immediately in response to the RS.

The following figure show the packet exchange between a host that has just connected to a local network and some IPv6 destination in the Internet:

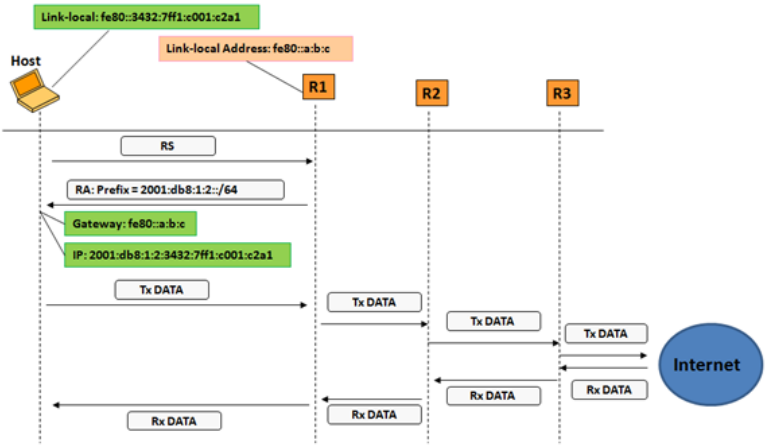

Figure 3.7. Packet exchange in IPv6

1) R1 is the router that gives connectivity to the host in the LAN and is periodically sending RAs .

2) Both R1 and Host have a link-local address in their interfaces connected to the host's LAN, this address is configured automatically when the interface is ready. Our host creates it's link-local address by combining the 64 leftmost bits of the link-local's prefix (`fe80::/64`) and the 64 rightmost bits of a locally generated IID (`:3432:7ff1:c001:c2a1`). These link-local addresses can be used in the LAN to exchange packets, but not to send packets outside the LAN.

3) The hosts needs two basic things to be able to send packets to other networks: a global IPv6 address and the address of a gateway, i.e., a router to which send the packets it wants to get routed outside its network.

4) Although R1 is sending RAs periodically (usually every several seconds) when the host get connected and has configured its link-local address, it sends an RS to which R1 responds immediately with an RA containing two things

1. A **global prefix of length 64 bits** that is intended for SLAAC. The host takes the received prefix and add to it a locally generated IID, usually the same as the one used for link-local address. This way a global IPv6 address is configured in the host and now can communicate with the IPv6 Internet
2. Implicitly included is the **link-local address of R1**, because it is the source address of the RA. Our host can use this address to configure the **default gateway**, the place to which send the packets by default, to reach an IPv6 host somewhere in Internet.

5) Once both the gateway and global IPv6 address are configured, the host can receive or send information. In the figure it has something to send (Tx Data) to a host in Internet, so it creates an IPv6 packet with the destination address of the recipient host and as source address the just autoconfigured global address, which is sent to its gateway, R1's link-local address. The destination host can answer with some data (Rx Data).

3.4. Network Example

Following we show how a simple IPv6 network looks like, displaying IPv6 addresses for all the networking devices.

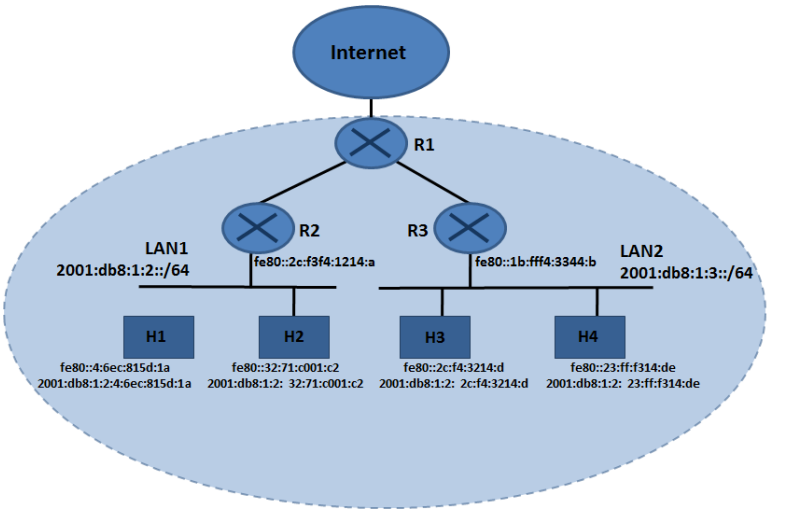

Figure 3.8. Simple IPv6 network

We have four hosts, (sensors, or other devices), and we want to put two of them in two different places, for example two floors in a building. We are dealing with four IP devices but you can have up to 2^{64} (18,446,744,073,709,551,616) devices connected on the same LAN.

We create two LANs with a router on each one, both routers connected to a central router (R1) that provides connectivity to Internet. LAN1 is served by R2 (with link-local address `fe80::2c:f3f4:1214:a` on that LAN) and uses the prefix `2001:db8:1:2::/64` announced by SLAAC. LAN2 is served by R3 (with link-local address `fe80::1b:fff4:3344:b` on that LAN) and uses the prefix `2001:db8:1:3::/64` announced by SLAAC.

All hosts have both a link-local IPv6 address and a global IPv6 address autoconfigured using the announced prefix by the corresponding router by means of RAs. In addition, remember that each host also configure the gateway using the link-local address used by the router for the RA. Link-local address can be used for communication among hosts inside a LAN, but for communicating with hosts in other LANs or any other network outside its own LAN a global IPv6 address is needed.

3.5. Short intro to Wireshark

What is Wireshark?

Figure 3.9. Wireshark logo

Wireshark is a free and open-source packet analyzer, allows packet traces to be sniffed, captured, and analysed.

A packet trace is a record of traffic at some location on the network, as if a snapshot was taken of all the bits that passed across a particular wire. The packet trace records a timestamp for each packet, along with the bits that make up the packet, from the low-layer headers to the higher-layer contents.

Wireshark runs on most operating systems, including Windows, MAC and Linux. It provides a graphical user interface that shows the sequence of packets and the meaning of the bits when interpreted as protocol headers and data. The packets are color-coded to convey their meaning, and Wireshark includes various ways to filter and analyze them to let you investigate different aspects of behavior. It is widely used to troubleshoot networks.

A common usage scenario is when a person wants to troubleshoot network problems or look at the internal workings of a network protocol. A user could, for example, see exactly what happens when he or she opens up a website or set up a wireless sensor network. It is also possible to filter and search on given packet attributes, which facilitates the debugging process.

More information and installation instructions are available at:

https://www.wireshark.org/

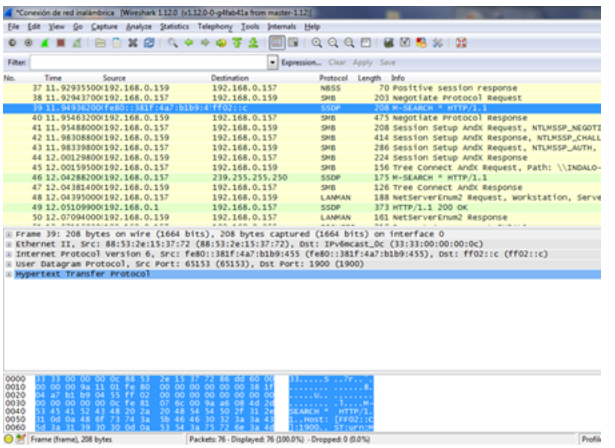

Figure 3.10. Wireshark Screenshot

When you open Wireshark, there are four main areas, from top to bottom: menus and filters, list of captured packets, detailed information about the selected packet, full content of the selected packet in hexadecimal and ASCII. Online directly links you to Wiresharks site, a handy user guide, and information on the security of Wireshark. Under Files, you'll find Open, which lets you open previously saved captures, and Sample Captures. You can download any of the sample captures through this website, and study the data. This will help you understand what kind of packets Wireshark can capture.

The Capture section let you choose your Interface. You can see each of the interfaces that are available. It'll also show you which ones are active. Clicking details will show you some pretty generic information about that interface.

Under Start, you can choose one or more interfaces to check out. Capture Options allows you to customize what information you see during a capture. Take a look at your Capture Options – here you can choose a filter, a capture file, and more. Under Capture Help, you can read up on how to capture, and you can check info on Network Media about which interfaces work on which platforms.

Let's select an interface and click Start. To stop a capture, press the red square in the top toolbar. If you want to start a new capture, hit the green triangle which looks like a shark fin next to it. Now that you have got a finished capture, you can click File, and save, open, or merge the capture. You can print it, you can quit the program, and you can export your packet capture in a variety of ways.

Under edit you can find a certain packet, with the search options you can copy packets, you can mark (highlight) any specific packet or all the packets. Another interesting thing you can do under Edit, is resetting the time value. You'll notice that the time is in seconds incrementing. You can reset it from the packet you've clicked on. You can add a comment to a packet, configure profiles and preferences.

When we select a packet from the list of captured ones, Wireshark shows detailed information of the different protocols used by that packet, for example Ethernet:

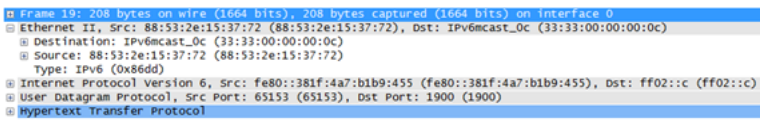

Figure 3.11. Ethernet packet

Or IPv6, where we can see the fields we saw before in the theory: Version, Traffic class, flowlabel, payload length, next header, etc.:

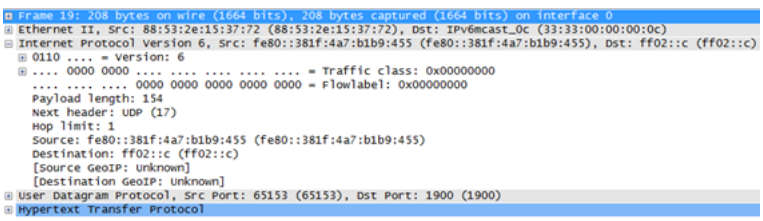

Figure 3.12. IPv6 packet

There are two methods to apply filters to the list of captured packets:

- Write a filter expression on the specific box and then apply it. Protocols could be specified (ip, ipv6, icmp, icmpv6), fields of a protocol (ipv6.dst, ipv6.src) and even complex expressions could be created using operators like AND (&&), OR (||) or the negation (!).

Figure 3.13. Wireshark Filter

- Another option to create filters it to right click in one filed of a captured packet, in the list of captured packets. There will appear a menu option "Apply as filter", with several options on how to use that field.

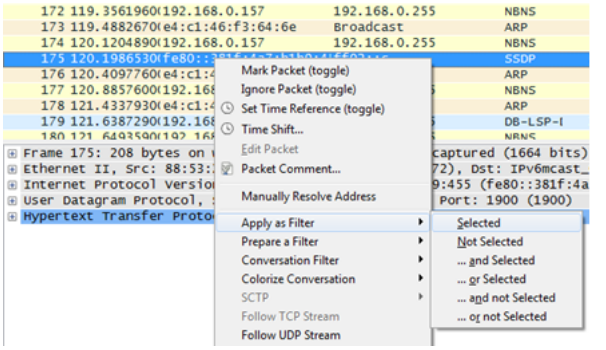

Figure 3.14. Wireshark Captured packets

Another useful and interesting option of Wireshark is the possibility to see statistics about the captured traffic. If we have applied filters, the statistics will be about the filtered traffic. Just go to Statistics menu and select, for example, Protocol Hierarchy:

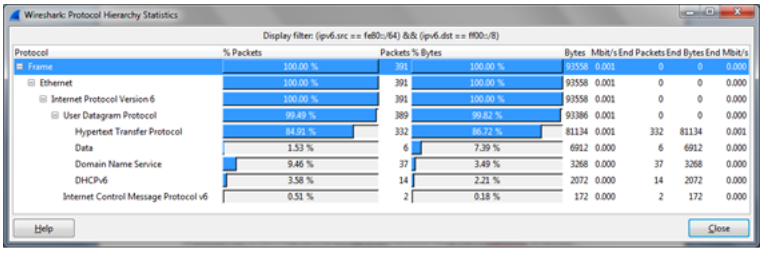

Figure 3.15. Wireshark statistics

Other interesting options are:

- Conversation List → IPv6

- Statistics → Endpoint List → IPv6
- Statistics → IO Graph

This last option allow to create graphs with different lines for different types of traffic and save the image:

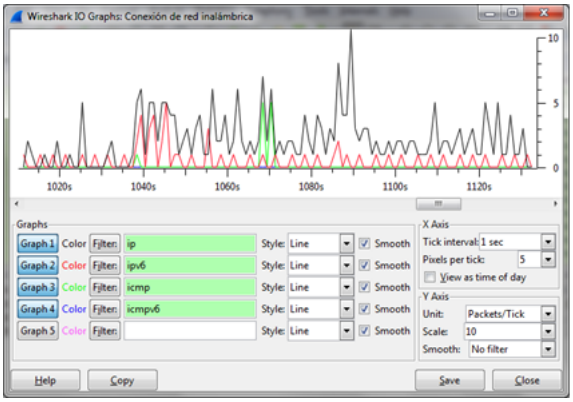

Figure 3.16. Wireshark charts

3.6. IPv6 Exercises

Let's test your IPv6 knowledge with the following exercises:

1) What size are IPv4 and IPv6 addresses, respectively?

a. 32-bits, 128-bits

b. 32-bits, 64-bits

c. 32-bits, 112-bits

d. 32-bits, 96-bits

e. none of these

2) Which of the following is a valid IPv6 address notation rule?

a. Zeroes on the right inside a group of 16 bits can be eliminated

b. The address is divided in 5 groups of 16 bits separated by ":"

c. The address is divided in 8 groups of 16 bits separated by "."

d. One or more groups of all zeroes could be substituted by "::"

e. Decimal notation is used grouping bits in 4 (nibbles)

3) Interface Identifiers (IID) or the rightmost bits of an IPv6 address used on a LAN will be 64 bits long.

a. True

b. False

4) Which of the following is a correct IPv6 address?

a. `2001:db8:A:B:C:D::1`

b. `2001:db8:000A:B00::1:3:2:F`

c. `2001:db8:G1A:A:FF3E::D`

d. `2001:0db8::F:A::B`

5) Which ones of the following sub-prefixes belong to the prefix `2001:db8:0A00::/48`? (Choose all that apply)

a. `2001:db9:0A00:0200::/56`

b. `2001:db8:0A00:A10::/64`

c. `2001:db8:0A:F:E::/64`

d. `2001:db8:0A00::/64`

6) IPv6 has a basic header with more fields than IPv4 header?

a. True

b. False

7) Extension headers could be added in any order

a. True

b. False

8) Autoconfiguration of IP devices is the same in IPv4 and IPv6

a. True

b. False

9) Which one is not an option for configuring an IPv6 address in an interface?

a. DHCPv6

b. Fixed address configured by vendor

c. Manually

d. SLAAC (Stateless Address Autoconfiguration)

10) Which packets are used by SLAAC to autoconfigure an IPv6 host?

a. NS/NA (Neighbor Solicitation / Neighbor Advertisement)

b. RS/RA (Router Solicitation / Router Advertisement)

c. Redirect messages

d. NS / RA (Neighbor Solicitation / Router Advertisement)

3.7. Addressing Exercises

A) Use the two compression rules to compress up to the maximum the following addresses:

1. `2001:0db8:00A0:7200:0fe0:000B:0000:0005`
2. `2001:0db8::DEFE:0000:C000`
3. `2001:db8:DAC0:0FED:0000:0000:0B00:12`

B) Decompress up to the maximum (representing all the 32 nibbles in hexadecimal) the following addresses:

1. `2001:db8:0:50::A:123`
2. `2001:db8:5::1`
3. `2001:db8:C00::222:0CC0`

C) You receive the following IPv6 prefix for your network: `2001:db8:A:0100::/56`

You have the following network:

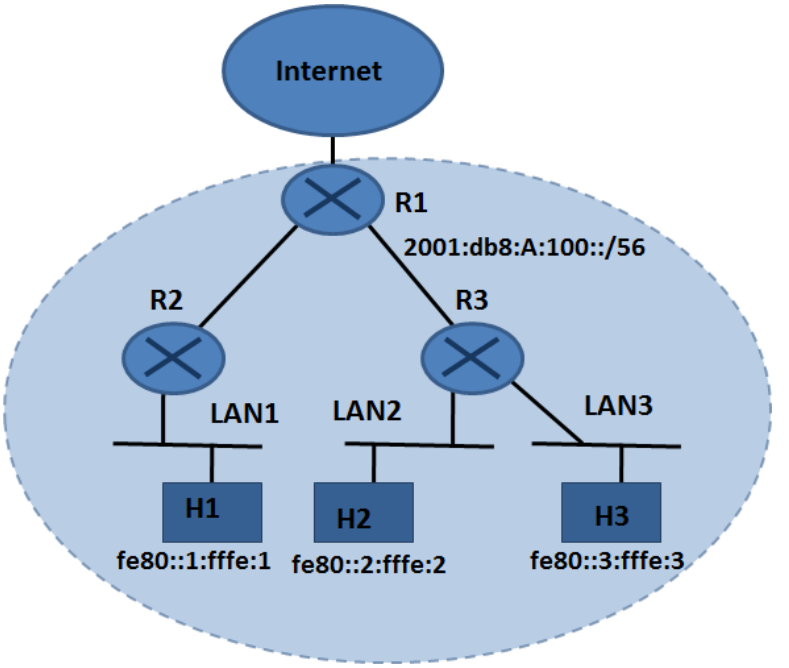

Figure 3.17. LAN Example

You have to define the following:

a. IPv6 prefix for LAN1, a /64 prefix taken from the /56 you have.

b. IPv6 prefix for LAN2, a /64 prefix taken from the /56 you have.

c. IPv6 prefix for LAN3, a /64 prefix taken from the /56 you have.

d. A global IPv6 address using the LAN1 prefix for H1 host (added to the link-local address already used).

e. A global IPv6 address using the LAN2 prefix for H2 host (added to the link-local address already used).

f. A global IPv6 address using the LAN3 prefix for H3 host (added to the link-local address already used).

Hint: To divide the /56 prefix into /64 prefixes, you have to change the value of the bits 57 to 64, i.e., the XY values in `2001:db8:A:01XY::/64`.

Chapter 4. Connecting our IPv6 Network to the Internet

As have been said in the introduction of this book, network communications is one of the four basic elements of an IoT system. We already have seen that IPv6 brings the possibility of giving an IP address to almost anything we can think of, and can do this making it easy to autoconfoigure network parameters on our devices.

Once we have all our "things" connected using IPv6, they can use it to communicate between them locally or with any other "thing" on the IPv6 Internet. In this chapter we will **focus on the Internet side of the communication of the "things"** composing the Internet of Things.

As we will see in this book, having the possibility to connect our devices to the Internet brings new possibilities and services. For example, if we could connect our wireless sensors networks to a centralized repository, where all the sensed information could be processed together and stores for historical records, this will allow to know better what has happened and maybe foresee future events. This basic idea is what nowadays is called "The Big Data" and has a whole set of it own concepts and techniques.

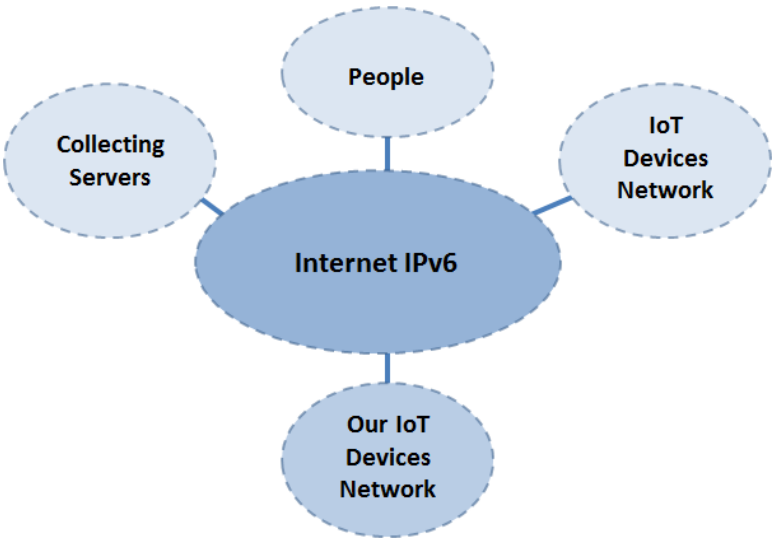

Figure 4.1. IPv6 Connectivity

Getting back to our network connectivity domain, our objective is to connect our IoT devices to the Internet using IPv6, allowing for communication with other IoT devices, collecting servers or even with people.

Related with the IPv6 connectivity to Internet is an important idea: **communication between IoT devices and the IPv6 Internet could be bidirectional**. This is important to remark because with IPv4, connectivity is usually designed as a one direction channel between a client and a server. This changes with IPv6.

Having a bidirectional communication with the IoT devices, will bring useful possibilities, because its not just that the device can send information to somewhere in the Internet, but that anybody in the Internet would be able to send information, requests or orders to the IoT device. This can be used in different scenarios:

- **Management**: To manage the IoT device performing some status tests, updating some parameters/configuration/firmware remotely allowing for a better and efficient use of the hardware platform and improving the infrastructure security.
- **Control**: Send commands or to control actuators that make the IoT device to perform an action.
- **Communication**: Send information to the IoT device, that can use it to be shown using some kind of interface.

IPv6 is still being deployed all over the different networks that compose the Internet, what means that different scenarios can be found when creating our IoT devices network and trying to connect it to the IPv6 Internet. Following are the three most possible scenarios, ordered by preference. This means that the objective, from the IPv6 connectivity point of view, is the first one: Native IPv6 connectivity.

- **Native IPv6 Connectivity**: This scenario is based on the support of IPv6 of both the ISP providing connectivity to the Internet and the router(s) and networks devices used in our network. Native IPv6 means that the IPv6 packets will flow without being changed or tunnelled all over its path from origin to destination. It's common to find what is called dual-stack networks, where both native IPv6 and native IPv4 is being used at the same time in the same interfaces and devices. This native IPv6 scenario covers both cases: IPv6-only or dual-stack.

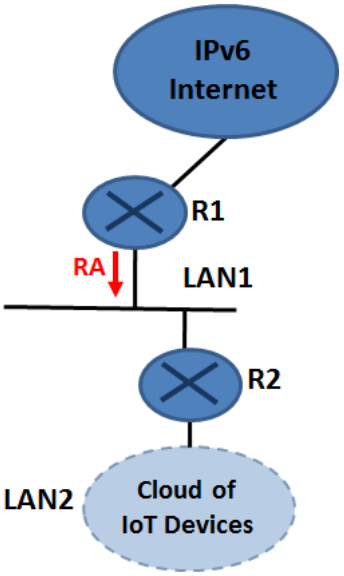

Figure 4.2. Native IPv6

As could be seen in the figure, our IoT devices cloud is connected to a router (R2) that provides them a prefix creating a LAN (LAN2). The router that provides connectivity to the IPv6 Internet (R1) will be in charge of autoconfiguring IPv6 devices in LAN1, for example R2, sending RAs (Router Advertisements) as seen when SLAAC was explained.

- **No IPv6 connectivity**: In this scenario we face a common problem nowadays, the lack of IPv6 connectivity from an ISP. Although we have IPv6 support on the router that connects our network to Internet the ISP does not provide IPv6 connectivity, only IPv4. The solution is to use one of the so called IPv6 Transition Mechanism. The most simple and useful in this case would be the 6in4 tunnel, based on creating a point-to-point static tunnel that encapsulates IPv6 packets into IPv4.

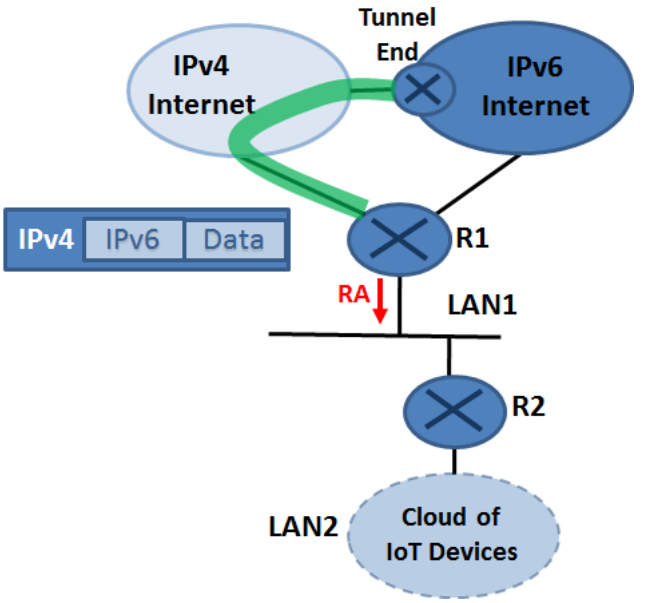

Figure 4.3. IPv4 tunneled IPv6

The figure shows the solution using a 6in4 tunnel, created from R1 to a tunnel end point, that will be a router having connectivity to both the IPv4 and IPv6 Internet. The native IPv6 traffic from our networks (LAN1 and LAN2) will reach R1, that will take the whole IPv6 packet with it's data, and put it inside a new IPv4 packet with destination IPv4 address the one of the Tunnel End router. The Tunnel End router will take off the IPv6 packet and put it as native IPv6 traffic into the IPv6 Internet. The same encapsulation occurs when IPv6 traffic is sent from the IPv6 Internet to our networks.

- **No IPv6 connectivity and no IPv6 capable router**: This scenario covers the case where there is no IPv6 connectivity from the ISP, nor IPv6 support on the router connecting our network to the Internet. As seen before, to solve the lack of IPv6 connectivity from the ISP we can use a 6in4 tunnel, but in this scenario we also have to solve the lack of IPv6 support on the router that won't be able to create the tunnel. The solution is to add a new router that supports IPv6, and IPv4, and create a 6in4 tunnel from this router to a tunnel end router somewhere on the IPv4 Internet.

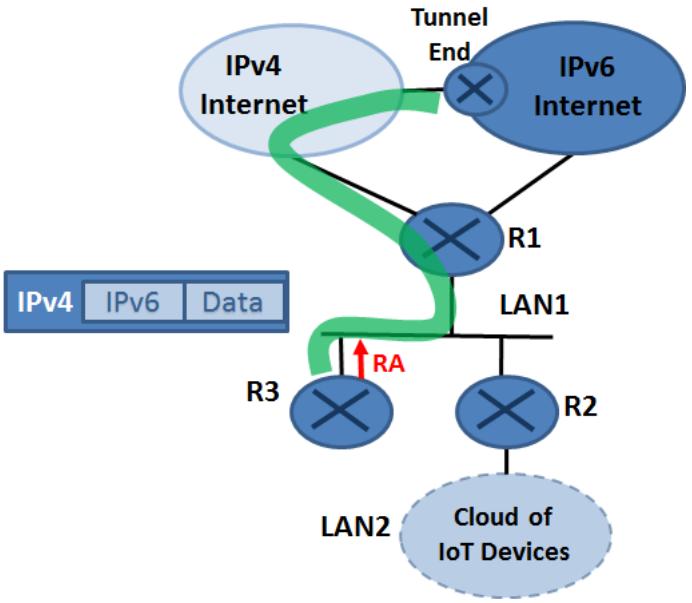

Figure 4.4. Local router does not support IPv6

In this scenario a new router R3 is added to both create a 6in4 tunnel towards the Tunnel End router and serve as an IPv6 gateway to our networks, sending RAs to autoconfigure IPv6 devices in LAN1. The encapsulation/decapsulation process will work exactly the same as in the previous scenario. The main difference here is that 6in4 tunnels need a public IPv4 address, so R3 will need to have a public IPv4 address to be able to create the 6in4 tunnel. This is easy to get in routers connected to ISPs, but not so common inside our network where we can have private addresses and NAT.

The scenarios showed above are based on a good infrastructure, where we have at least two routers and a couple of LANs. All three scenarios could be simplified into a just one router scenario shown in the following figure:

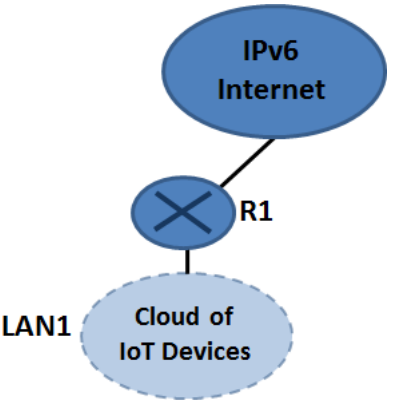

Figure 4.5. Simplified Scenario

Considerations about lack of IPv6 connectivity from the ISP and IPv6 support on the router are the same, although for the latter the solution is to change the only router by one that supports IPv6.

This is common because IoT or WSN could be deployed anywhere, including far away networks connected using any wireless technology that can reach remote places. In this scenarios there are severe restrictions on the number of devices, power consumption, etc. For example, a cloud of sensors could be deployed in the country to sense temperature and moisture, and all of them get connectivity through just one router connected using IPv6 mobile phone network (GPRS, 3G or LTE).

Chapter 5. Introduction to 6LoWPAN

One of the drivers of the IoT, where anything can be connected, is the use of wireless technologies to get a communication channel to send and receive information. This wide adoption of wireless technologies allows increasing the number of connected devices but results in limitations in terms of cost, battery life, power consumption, or communication distance for the devices. New technologies and protocols should tackle a new environment, usually called Low power and Lossy networks (LLNs), with the following characteristics:

1. Significantly more devices than those on current local area networks.
2. Severely limited code and ram space in devices.
3. Networks with limited communications distance (range), power and processing resources.
4. All elements should work together to optimize energy consumption and bandwidth usage.

Another feature that is being widely adopted within IoT is the use of IP as the network protocol. The use of IP provides several advantages, because it is an open standard that is widely available, allowing for easy and cheap adoption, good interoperability and easy application layer development. The use of a common standard like an end-to-end IP-based solution avoids the problem of non-interoperable networks interconnected by protocol

For wireless communication technology, the IEEE 802.15.4 standard [IEEE802.15.4] is very promising for the lower (link and physical) layers, although others are also being considered as good options like Low Power WiFi, Bluetooth ® Low Energy, DECT Ultra Low Energy, ITU-T G.9959 networks, or NFC (Near Field Communication).

One component of the IoT that has received significant support from vendors and standardization organizations is that of WSN (Wireless Sensor Networks).

The IETF has different working groups (WGs) developing standards to be used by WSN:

1. **6lowpan**: IPv6 over Low-power Wireless Personal Area Networks [sixlowpan], defined the standards needed to have IPv6 communication over the IEEE 802.15.4 wireless communication technology. 6lowpan act as an adaptation layer between the standard IPv6 world and the low power and lossy communications wireless media offered by IEEE 802.15.4. Note that this standard is only defined with IPv6 in mind, no IPv4 support available.
2. **roll**: Routing Over Low power and Lossy networks [roll]. LLNs have specific routing requirements that could not be satisfied with existing routing protocols. This WG focuses on routing solutions for a subset of all possible application areas of LLNs (industrial, connected

home, building and urban sensor networks), and protocols are designed to satisfy their application-specific routing requirements. Here again the WG focuses only on IPv6 routing architectural framework.

3. **6lo**: IPv6 over Networks of Resource-constrained Nodes [sixlo]. This WG deals with IPv6 connectivity over constrained node networks. It extends the work of the 6lowpan WG, defining IPv6-over-foo adaptation layer specifications using 6LoWPAN for link layer in constrained node networks.

As seen, 6LoWPAN is the basis of the work carried out in standardization at IETF to communicate constrained resources nodes in LLNs using IPv6. The work on 6LoWPAN is completed and is being further complemented by the roll WG to satisfy routing needs and the 6lo WG to extend the 6lowpan standards to any other link layer technology. Following are more details about 6LoWPAN, as the first step into the IPv6 based WSN/IoT. 6LoWPAN and related standards are concerned about providing IP connectivity to devices, irrelevantly of the upper layers, except for the UDP transport layer protocol that is specifically considered.

5.1. Overview of LoWPANs

Low-power and lossy networks (LLNs) is the term commonly used to refer to networks made of highly constrained nodes (limited CPU, memory, power) interconnected by a variety of "lossy" links (low-power radio links). They are characterized by low speed, low performance, low cost, and unstable connectivity.

A LoWPAN is a particular instance of an LLN, formed by devices complying with the IEEE 802.15.4 standard.

The typical characteristics of devices in a LoWPAN are:

1. **Limited Processing Capability**: Different types and clock speeds processors, starting at 8-bits.
2. **Small Memory Capacity**: From few kilobytes of RAM with a few dozen kilobytes of ROM/flash memory, it’s expected to grow in the future, but always trying to keep at the minimum necessary.
3. **Low Power**: In the order of tens of milliamperes.
4. **Short Range**: The Personal Operating Space (POS) defined by IEEE 802.15.4 implies a range of 10 meters. For real implementations it can reach over 100 meters in line-of-sight situations.
5. **Low Cost**: This drives some of the other characteristics such as low processing, low memory, etc.

All this constraints on the nodes are expected to change as technology evolves, but compared to other fields it's expected that the LoWPANs will always try to use very restricted devices to allow for low prices and long life which implies hard restrictions in all other features.

A LoWPAN typically includes devices that work together to connect the physical environment to real-world applications, e.g., wireless sensors, although a LoWPAN is not necessarily comprised of sensor nodes only, since it may also contain actuators.

It's also important to identify the characteristics of LoWPANs, because they will be the constraints guiding all the technical work:

1. Small packet size: Given that the maximum physical layer frame is 127 bytes, the resulting maximum frame size at the media access control layer is 102 octets. Link-layer security imposes further overhead, which leaves a maximum of 81 octets for data packets.
2. IEEE 802.15.4 defines several addressing modes: It allows the use of either IEEE 64-bit extended addresses or (after an association event) 16-bit addresses unique within the PAN (Personal Area Network).
3. Low bandwidth: Data rates of 250 kbps, 40 kbps, and 20 kbps for each of the currently defined physical layers (2.4GHz, 915MHz, and 868MHz, respectively).
4. Topologies include star and mesh.
5. Large number of devices expected to be deployed during the lifetime of the technology. Location of the devices is typically not predefined, as they tend to be deployed in an ad-hoc fashion. Sometimes the location of these devices may not be easily accessible or they may move to new locations.
6. Devices within LoWPANs tend to be unreliable due to variety of reasons: uncertain radio connectivity, battery drain, device lockups, physical tampering, etc.
7. Sleeping mode: Devices may sleep for long periods of time in order to conserve energy, and are unable to communicate during these sleep periods.

5.2. About the use of IP on LoWPANs

As said before, it seems that the use of IP, and specifically IPv6, is being widely adopted because it offers several advantages. 6LoWPANs are IPv6-based LoWPAN networks.

In this section we will see these advantages and some problems raised by the use of IP over LoWPANs.

The application of IP technology and, in particular, IPv6 networking is assumed to provide the following benefits to LoWPANs:

a. The pervasive nature of IP networks allows leveraging existing infrastructure.
b. IP-based technologies already exist, are well-known, proven to be working and widely available. This allows for an easier and cheaper adoption, good interoperability and easier application layer development.
c. IP networking technology is specified in open and freely available specifications, which is able to be better understood by a wider audience than proprietary solutions.
d. Tools for IP networks already exist.
e. IP-based devices can be connected readily to other IP-based networks, without the need for intermediate entities like protocol translation gateways or proxies.
f. The use of IPv6, specifically, allows for a huge amount of addresses and provides for easy network parameters autoconfiguration (SLAAC). This is paramount for 6LoWPANs where large number of devices should be supported.

On the counter side using IP communication in LoWPANs raise some issues that should be taken into account:

a. IP Connectivity: One of the characteristics of 6LoWPANs is the limited packet size, which implies that headers for IPv6 and layers above must be compressed whenever possible.
b. Topologies: LoWPANs must support various topologies including mesh and star: Mesh topologies imply multi-hop routing to a desired destination. In this case, intermediate devices act as packet forwarders at the link layer. Star topologies include provisioning a subset of devices with packet forwarding functionality. If, in addition to IEEE 802.15.4, these devices use other kinds of network interfaces such as Ethernet or IEEE 802.11, the goal is to seamlessly integrate the networks built over those different technologies. This, of course, is a primary motivation to use IP to begin with.
c. Limited Packet Size: Applications within LoWPANs are expected to originate small packets. Adding all layers for IP connectivity should still allow transmission in one frame, without incurring excessive fragmentation and reassembly. Furthermore, protocols must be designed or chosen so that the individual "control/protocol packets" fit within a single 802.15.4 frame.
d. Limited Configuration and Management: Devices within LoWPANs are expected to be deployed in exceedingly large numbers. Additionally, they are expected to have limited display and input capabilities. Furthermore, the location of some of these devices may be hard to reach. Accordingly, protocols used in LoWPANs should have minimal configuration, preferably work "out of the box", be easy to bootstrap, and enable the network to self heal given the inherent unreliable characteristic of these devices.

e. Service Discovery: LoWPANs require simple service discovery network protocols to discover, control and maintain services provided by devices.

f. Security: IEEE 802.15.4 mandates link-layer security based on AES, but it omits any details about topics like bootstrapping, key management, and security at higher layers. Of course, a complete security solution for LoWPAN devices must consider application needs very carefully.

5.3. 6LoWPAN

We have seen that there is a lower layer (physical and link layers on TCP/IP stack model) that provide connectivity to devices in what is called a LoWPAN. Also that using IPv6 over this layer would bring several benefits. The main reason for developing the IETF standards mentioned in the introduction is that between the IP (network layer) and the lower layer there are some important issues that need to be solved by means of an adaptation layer, the 6lowpan.

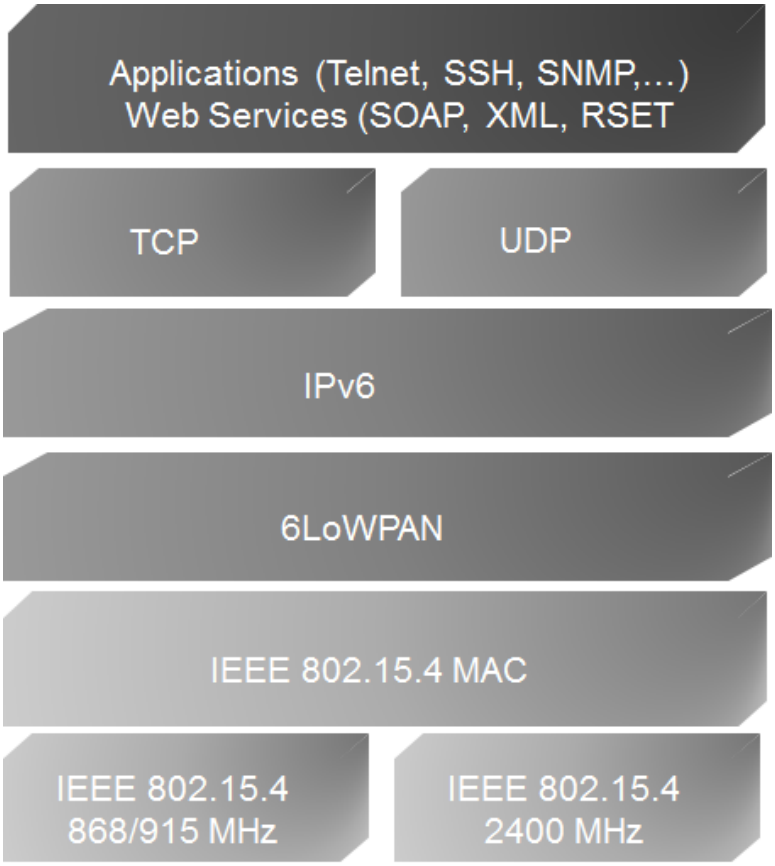

Figure 5.1. 6LoWPAN in the protocol stack

The main goals of 6lowpan are:

1. Fragmentation and Reassembly layer: IPv6 specification [RFC2460] establishes that the minimum MTU that a link layer should offer to the IPv6 layer is 1280 bytes. The protocol data units may be as small as 81 bytes in IEEE 802.15.4. To solve this difference a fragmentation and reassembly adaptation layer must be provided at the layer below IP.

2. Header Compression: Given that in the worst case the maximum size available for transmitting IP packets over an IEEE 802.15.4 frame is 81 octets, and that the IPv6 header is 40 octets long, (without optional extension headers), this leaves only 41 octets for upper-layer protocols, like UDP and TCP. UDP uses 8 octets in the header and TCP uses 20 octets. This leaves 33 octets for data over UDP and 21 octets for data over TCP. Additionally, as pointed above, there is also a need for a fragmentation and reassembly layer, which will use even more octets leaving very few octets for data. Thus, if one were to use the protocols as is, it would lead to excessive fragmentation and reassembly, even when data packets are just 10s of octets long. This points to the need for header compression.

3. Address Autoconfiguration: specifies methods for creating IPv6 stateless address auto configuration (in contrast to stateful) that is attractive for 6LoWPANs, because it reduces the configuration overhead on the hosts. There is a need for a method to generate the IPv6 IID (Interface Identifier) from the EUI-64 assigned to the IEEE 802.15.4 device.

4. Mesh Routing Protocol: A routing protocol to support a multi-hop mesh network is necessary. Care should be taken when using existing routing protocols (or designing new ones) so that the routing packets fit within a single IEEE 802.15.4 frame. The mechanisms defined by 6lowpan IETF WG are based on some requirements for the IEEE 802.15.4 layer:

5. IEEE 802.15.4 defines four types of frames: beacon frames, MAC command frames, acknowledgement frames and data frames. IPv6 packets must be carried on data frames.

6. Data frames may optionally request that they be acknowledged. It is recommended that IPv6 packets be carried in frames for which acknowledgements are requested so as to aid link-layer recovery.

7. The specification allows for frames in which either the source or destination addresses (or both) are elided. Both source and destination addresses are required to be included in the IEEE 802.15.4 frame header.

8. The source or destination PAN ID fields may also be included. 6LoWPAN standard assumes that a PAN maps to a specific IPv6 link.

9. Both 64-bit extended addresses and 16-bit short addresses are supported, although additional constraints are imposed on the format of the 16-bit short addresses.
10. Multicast is not supported natively in IEEE 802.15.4. Hence, IPv6 level multicast packets must be carried as link-layer broadcast frames in IEEE 802.15.4 networks. This must be done such that the broadcast frames are only heeded by devices within the specific PAN of the link in question.

The 6LoWPAN adaptation format was specified to carry IPv6 datagrams over constrained links, taking into account limited bandwidth, memory, or energy resources that are expected in applications such as wireless sensor networks. For each of these goals and requirements there is a solution provided by the 6lowpan specification:

1. A Mesh Addressing header to support sub-IP forwarding.
2. A Fragmentation header to support the IPv6 minimum MTU requirement.
3. A Broadcast Header to be used when IPv6 multicast packets must be sent over the IEEE 802.15.4 network.
4. Stateless header compression for IPv6 datagrams to reduce the relatively large IPv6 and UDP headers down to (in the best case) several bytes. These header are used as the LoWPAN encapsulation, and could be used at the same time forming what is called the header stack. Each header in the header stack contains a header type followed by zero or more header fields. When more than one LoWPAN header is used in the same packet, they must appear in the following order: Mesh Addressing Header, Broadcast Header, and Fragmentation Header.

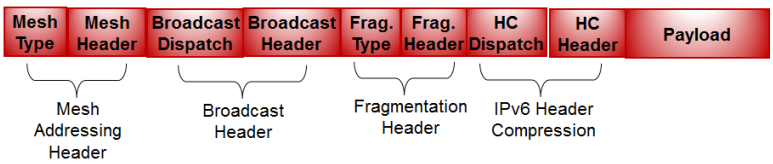

Figure 5.2. 6LoWPAN headers

5.4. IPv6 Interface Identifier (IID)

As already said an IEEE 802.15.4 device could have two types of addresses. For each one there is a different way of generating the IPv6 IID.

1. IEEE EUI-64 address: All devices have this one. In this case, the Interface Identifier is formed from the EUI-64, complementing the "Universal/Local" (U/L) bit, which is the next-

to-lowest order bit of the first octet of the EUI-64. Complementing this bit will generally change a 0 value to a 1.

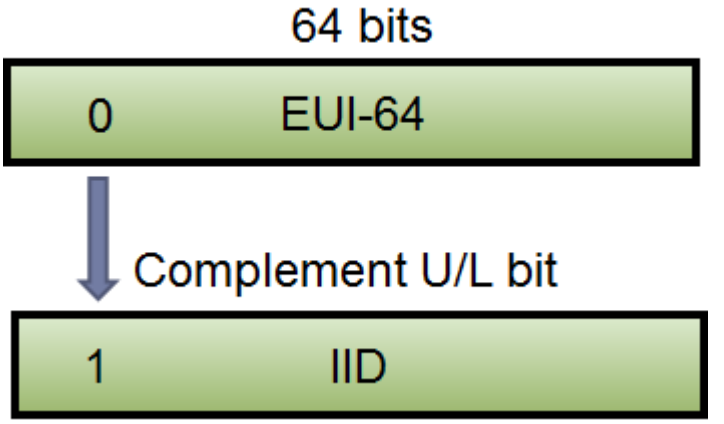

Figure 5.3. EUI-64 derived IID

1. 16-bit short addresses: Possible but not always used. The IPv6 IID is formed using the PAN (or zeroes in case of not knowing the PAN) and the 16 bit short address as in the figure below.

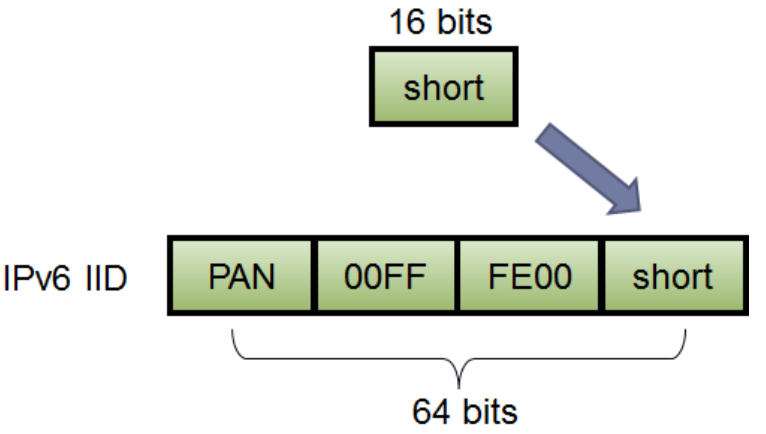

Figure 5.4. IPv6IID

5.5. Header Compression

Two encoding formats are defined for compression of IPv6 packets: LOWPAN_IPHC and LOWPAN_NHC, an encoding format for arbitrary next headers.

To enable effective compression, LOWPAN_IPHC relies on information pertaining to the entire 6LoWPAN. LOWPAN_IPHC assumes the following will be the common case for 6LoWPAN communication:

1. Version is 6.
2. Traffic Class and Flow Label are both zero.
3. Payload Length can be inferred from lower layers from either the 6LoWPAN Fragmentation header or the IEEE 802.15.4 header.
4. Hop Limit will be set to a well-known value by the source.
5. Addresses assigned to 6LoWPAN interfaces will be formed using the link-local prefix or a small set of routable prefixes assigned to the entire 6LoWPAN.
6. Addresses assigned to 6LoWPAN interfaces are formed with an IID derived directly from either the 64-bit extended or the 16-bit short IEEE 802.15.4 addresses. Depending on how closely the packet matches this common case, different fields may not be compressible thus needing to be carried "in-line" as well. The base format used in LOWPAN_IPHC encoding is shown in the figure below.

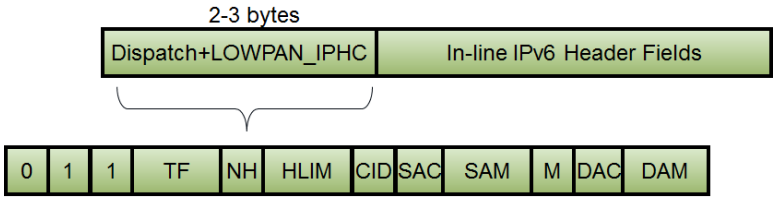

Figure 5.5. Header compression

Where:

- TF: Traffic Class, Flow Label.
- NH: Next Header.
- HLIM: Hop Limit.
- CID: Context Identifier Extension.
- SAC: Source Address Compression.
- SAM: Source Address Mode.
- M: Multicast Compression.

- DAC: Destination Address Compression.
- DAM: Destination Address Mode.

Not going into details, it's important to understand how 6LoWPAN compression works. To this end, let's see two examples:

1. HLIM (Hop Limit): Is a two bits field that can have four values, three of them make the hop limit field to be compressed from 8 to 2 bits:
   a. 00: Hop Limit field carried in-line. There is no compression and the whole field is carried in-line after the LOWPAN_IPHC.
   b. 01: Hop Limit field compressed and the hop limit is 1.
   c. 10: Hop Limit field compressed and the hop limit is 64.
   d. 11: Hop Limit field compressed and the hop limit is 255.
2. SAC/DAC used for the source IPv6 address compression. SAC indicates which address compression is used, stateless (SAC=0) or stateful context-based (SAC=1). Depending on SAC, DAC is used in the following way:
   a. If SAC=0, then SAM:
      - 00: 128 bits. Full address is carried in-line. No compression.
      - 01: 64 bits. First 64-bits of the address are elided, the link-local prefix. The remaining 64 bits are carried in-line.
      - 10: 16 bits. First 112 bits of the address are elided. First 64 bits is the link-local prefix. The following 64 bits are 0000:00ff:fe00:XXXX, where XXXX are the 16 bits carried in-line.
      - 11: 0 bits. Address is fully elided. First 64 bits of the address are the link-local prefix. The remaining 64 bits are computed from the encapsulating header (e.g., 802.15.4 or IPv6 source address).
   b. If SAC=1, then SAM:
      - 00: 0 bits. The unspecified address (::).
      - 01: 64 bits. The address is derived using context information and the 64 bits carried in-line. Bits covered by context information are always used. Any IID bits not covered by context information are taken directly from the corresponding bits carried in-line.
      - 10: 16 bits. The address is derived using context information and the 16 bits carried in-line. Bits covered by context information are always used. Any IID bits not covered by context information are taken directly from their corresponding bits in the 16-bit

to IID mapping given by 0000:00ff:fe00:XXXX, where XXXX are the 16 bits carried in-line.

- 11: 0 bits. The address is fully elided and it is derived using context information and the encapsulating header (e.g., 802.15.4 or IPv6 source address). Bits covered by context information are always used. Any IID bits not covered by context information are computed from the encapsulating header.

The base format is two bytes (16 bits) long. If the CID (Context Identifier Extension) field has a value of 1, it means that an additional 8-bit Context Identifier Extension field immediately follows the Destination Address Mode (DAM) field. This would make the length be 24 bits or three bytes.

This additional octet identifies the pair of contexts to be used when the IPv6 source and/or destination address is compressed. The context identifier is 4 bits for each address, supporting up to 16 contexts. Context 0 is the default context. The two fields on the Context Identifier Extension are:

- SCI: Source Context Identifier. Identifies the prefix that is used when the IPv6 source address is statefully compressed.
- DCI: Destination Context Identifier. Identifies the prefix that is used when the IPv6 destination address is statefully compressed.

The Next Header field in the IPv6 header is treated in two different ways, depending on the values indicated in the NH (Next Header) field of the LOWPAN_IPHC enconding shown above.

If NH = 0, then this field is not compressed and all the 8 bits are carried in-line after the LOWPAN_IPHC.

If NH = 1, then the Next Header field is compressed and the next header is encoded using LOWPAN_NHC encoding. This results in the structure shown in the figure below.

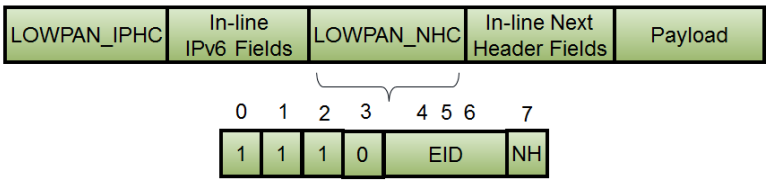

Figure 5.6. LoWPAN header

For IPv6 Extension headers the LOWPAN_NHC has the format shown in the figure, where:

- EID: IPv6 Extension Header ID:

- 0: IPv6 Hop-by-Hop Options Header.
- 1: IPv6 Routing Header.
- 2: IPv6 Fragment Header.
- 3: IPv6 Destination Options Header.
- 4: IPv6 Mobility Header.
- 5: Reserved.
- 6: Reserved.
- 7: IPv6 Header.

- NH: Next Header
  - 0: Full 8 bits for Next Header are carried in-line.
  - 1: Next Header field is elided and is encoded using LOWPAN_NHC. For the most part, the IPv6 Extension Header is carried unmodified in the bytes immediately following the LOWPAN_NHC octet.

5.6. NDP optimization

IEEE 802.15.4 and other similar link technologies have limited or no usage of multicast signalling due to energy conservation. In addition, the wireless network may not strictly follow the traditional concept of IP subnets and IP links. IPv6 Neighbor Discovery was not designed for non-transitive wireless links, as its reliance on the traditional IPv6 link concept and its heavy use of multicast make it inefficient and sometimes impractical in a low-power and lossy network.

For this reasons, some simple optimizations have been defined for IPv6 Neighbor Discovery, its addressing mechanisms and duplicate address detection for LoWPANs [RFC6775]:

1. Host-initiated interactions to allow for sleeping hosts.
2. Elimination of multicast-based address resolution for hosts.
3. A host address registration feature using a new option in unicast Neighbor Solicitation (NS) and Neighbor Advertisement (NA) messages.
4. A new Neighbor Discovery option to distribute 6LoWPAN header compression context to hosts.
5. Multihop distribution of prefix and 6LoWPAN header compression context.

6. Multihop Duplicate Address Detection (DAD), which uses two new ICMPv6 message types.

The two multihop items can be substituted by a routing protocol mechanism if that is desired.

Three new ICMPv6 message options are defined:

1. The Address Registration Option (ARO).
2. The Authoritative Border Router Option (ABRO).
3. The 6LoWPAN Context Option (6CO)

Also two new ICMPv6 message types are defined:

1. The Duplicate Address Request (DAR).
2. The Duplicate Address Confirmation (DAC)

Chapter 6. Introduction to Contiki

Contiki is an open source operating system for the Internet of Things, it connects tiny low-cost, low-power microcontrollers to the Internet.

Contiki provides powerful low-consumption Internet communication, it supports fully standard IPv6 and IPv4, along with the recent low-power wireless standards: 6lowpan, RPL, CoAP. With Contiki's ContikiMAC and sleepy routers, even wireless routers can be battery-operated.

With Contiki, development is easy and fast: Contiki applications are written in standard C, with the Cooja simulator Contiki networks can be emulated before burned into hardware, and Instant Contiki provides an entire development environment in a single download.

More information available at:

http://contiki-os.org/

The remainder of the chapters aim to be a thoughtful introduction to Contiki 3.0, its core components and features. The following references are the must-go places to search for detailed information and answers are:

- The Contiki wiki page: https://github.com/contiki-os/contiki/wiki
- The Contiki mailing list: http://sourceforge.net/p/contiki/mailman/contiki-developers
- Zolertia Wiki page: https://github.com/Zolertia/Resources/wiki

6.1. Install Contiki

There are several ways to install Contiki, from scratch by installing from sources or using virtual environments, depending on the flavour and time willing to spend.

To work with Contiki three steps are required:

- The Contiki source code.
- A target platform (virtual platform or a real hardware one).
- A toolchain to compile the source code for such target platform.

The remainder of the book assumes Contiki will run in an Unix environment, as the virtualized environments run on Ubuntu.

6.1.1. Install from sources

Contiki source code is actively supported by contributors from Universities, Research centers and developers from all over the world.

The source code is hosted in GitHub:

https://github.com/contiki-os/contiki

To grab the source code open a terminal and execute the following:

```
sudo apt-get -y install git
git clone --recursive https://github.com/contiki-os/contiki.git
```

What is git?

Git is a free and open source distributed version control system, designed for speed and efficiency.

The main difference with other change control tools is the possibility to work locally as your local copy is a repository, and you can commit to it and get all benefits of source control.

There are some great tutorials online to learn more about git:

http://try.github.io

http://rogerdudler.github.io/git-guide/

GitHub is a GIT repository web-based hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features. GitHub provides a web-based graphical interface and desktop as well as mobile integration. It also provides access control and several collaboration features such as wikis, task management, bug tracking and feature requests for every project.

The advantage of using GIT and hosting the code at github is that of allowing people to fork the code, develop on its own, and then contribute back and share your progress.

To install the toolchain and required dependencies, run in a terminal the following:

```
sudo echo "deb http://ppa.launchpad.net/terry.guo/gcc-arm-embedded/ubuntu trusty
 main" > /etc/apt/sources.list.d/gcc-arm-embedded.list
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-key
 FE324A81C208C89497EFC6246D1D8367A3421AFB
sudo apt-get update
sudo apt-get -y install build-essential automake gettext
sudo apt-get -y install gcc-arm-none-eabi curl graphviz unzip wget
sudo apt-get -y install gcc gcc-msp430
sudo apt-get -y install openjdk-7-jdk openjdk-7-jre ant
```

This will install support for the ARM Cortex-M3 and MSP430 platforms, as well as support for Cooja, the Contiki's emulator to be discussed in the next sections.

6.1.2. Instant Contiki Virtual Machine

Instant Contiki is an entire Contiki development environment in a single download. It is an Ubuntu Linux virtual machine and has Contiki OS and all the development tools, compilers, and simulators required already pre-installed.

Grab Instant Contiki from Contiki website:

http://www.contiki-os.org/start.html

The latest Instant Contiki release is the 3.0, following the Contiki 3.0 source code release.

This book is heavily based on this release, using earlier versions is not recommended.

The release tag is available at:

https://github.com/contiki-os/contiki/tree/release-3-0

On Windows and Linux host you can use VMWare player to run the Virtual Machine:

https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/6_0

On OSX you can download VMWare Fusion:

http://www.vmware.com/products/fusion

Using VMWare just open the `Instant_Contiki_Ubuntu_12.04_32-bit.vmx` file, if prompted about the VM source just choose `I copied it` then wait for the virtual Ubuntu Linux boot up.

Log into Instant Contiki. The password and user name is `user`. Don't upgrade right now.

Remember to update the Contiki repository and get the latest upgrades:

```
cd /home/user/contiki
git fetch origin
git pull origin master
```

6.2. Test Contiki installation

Let us first check the toolchain installation. The MSP430 toolchain can tested with:

```
msp430-gcc --version
msp430-gcc (GCC) 4.7.0 20120322 (mspgcc dev 20120716)
Copyright (C) 2012 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

And the ARM Cortex-M3 toolchain:

```
arm-gcc-none-eabi-gcc --version
arm-none-eabi-gcc (GNU Tools for ARM Embedded Processors) 4.9.3 20150529 (release)
 [ARM/embedded-4_9-branch revision 224288]
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

6.3. Contiki structure

Contiki has the following file structure:

Folder	**Description**	**Zolertia files**
examples	Ready to build examples	examples/zolertia
app	Contiki applications	-
cpu	Specific MCU files	msp430, cc2538
dev	External chip and devices	cc2420, cc1200
platform	Specific files and platform drivers	z1, zoul
regression-tests	**nightly regression tests**	**-**

Folder	Description	Zolertia files
core	Contiki core files and libraries	-
tools	Tools for flashing, debuging, simulating, etc.	zolertia, sky
doc	Self-generated doxygen documentation	-
regression-tests	**nightly regression tests**	**-**

In the following sections we will cover the examples and specific platform files for the Zolertia platforms.

6.4. Run Contiki on real hardware

For the remainder of the book we will use Zolertia Z1 and RE-Mote hardware development platforms. Other platforms can be used as well, but are out of the scope of this book.

Contiki drivers and libraries are (normally) platform independent, most examples can be run in both Z1 and RE-Mote platforms by taking into consideration specific platform settings.

There are platform-specific examples for the Z1 platform at `examples/zolertia/z1`, and at `examples/zolertia/zoul` for the RE-Mote.

More information about both platforms and updates guides can be found at:

- Zolertia website: http://www.zolertia
- Zolertia RE-Mote wiki page: https://github.com/Zolertia/Resources/wiki/RE-Mote
- Zolertia Z1 wiki page: https://github.com/Zolertia/Resources/wiki#the-z1-mote

6.4.1. Zolertia Zoul module and the RE-Mote development platform

The **Zoul** is a module based in the CC2538 ARM Cortex-M3 system on chip (SoC), with an on-board 2.4 GHz IEEE 802.15.4 RF interface, running at up to 32 MHz with 512 KB of programmable flash and 32 KB of RAM, bundled with a CC1200 868/915 MHz RF transceiver to allow dual band operation.

The Zoul allows a fast reusability on designs and to quickly scale from prototyping to production.

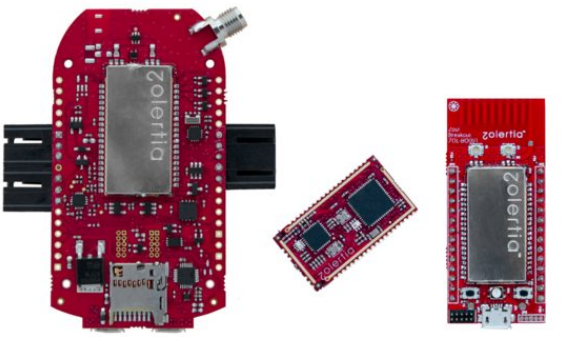

Figure 6.1. Zolertia Zoul module and the RE-Mote platform

The **RE-Mote** has a Zoul on board and also packs:

- ISM 2.4-GHz IEEE 802.15.4 & Zigbee compliant radio.
- ISM 863-950-MHz ISM/SRD band radio.
- AES-128/256, SHA2 Hardware Encryption Engine.
- ECC-128/256, RSA Hardware Acceleration Engine for Secure Key Exchange.
- Consumption down to 150 nA using the shutdown mode.
- Programming over BSL without requiring to press any button to enter bootloader mode.
- Built-in battery charger (500 mA), facilitating Energy Harvesting and direct connection to Solar Panels and to standards LiPo batteries.
- Wide range DC Power input: 2-16 V.
- Small form-factor.
- MicroSD over SPI.
- On board RTC (programmable real time clock) and external watchdog timer (WDT).
- Programmable RF switch to connect an external antenna either to the 2.4 GHz or to the Sub 1 GHz RF interface through the RP-SMA connector.

The **RE-Mote** has been developed jointly with universities and industrial partners in the frame of the European research project **RERUM**(**RERUM: REliable, Resilient and secUre IoT for sMart city applications**).

6.4.2. Zolertia Z1 mote

The **Z1 mote** features a second generation MSP430F2617 low power 16-bit RISC CPU @16 MHz MCU, 8 kB RAM and a 92 kB Flash memory. It includes the well known CC2420

transceiver, IEEE 802.15.4 compliant, which operates at 2.4 GHz with an effective data rate of 250 kbps.

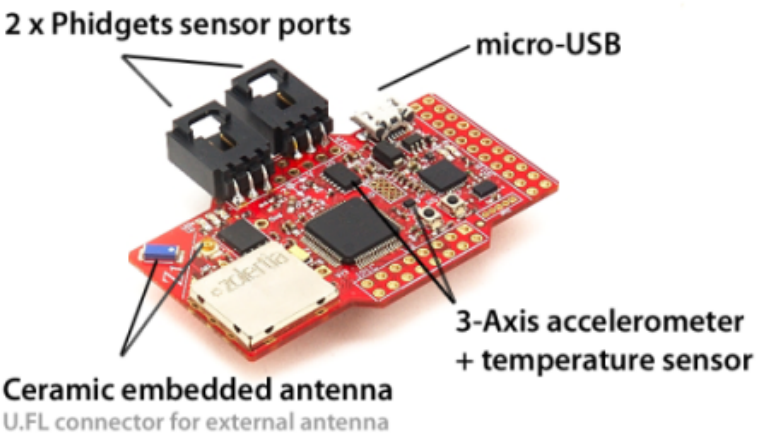

Figure 6.2. Zolertia Z1 mote

The Zolertia Z1 mote can run TinyOS, Contiki OS, OpenWSN and RIOT, and has been used actively for over 5 years in universities, research and development centers and commercial products in more than 43 countries, being featured in more than 50 scientific publications.

The Z1 mote is fully emulated in both MSPSIM and Cooja.

6.4.3. What are the differences between the RE-Mote and the Z1 platforms?

In a nutshell: power and coolness.

The RE-Mote has 4 times more RAM than the Z1 mote, 5 times more flash memory, twice the operation frequency and 120 times less power consumption in its lowest power mode (shutdown mode).

Another major difference is (at the present time of this book) the Z1 mote being supported by Cooja, the Contiki emulator, and the RE-Mote not. However efforts are currently being done to provide emulation framework support in the EMUL8[1] project.

To check in depth the differences between the RE-Mote and the Z1 mote, and also guidelines to port applications developed for the Z1 to the RE-Mote, visit the following link:

[1] http://emul8.org/

https://github.com/Zolertia/Resources/wiki/Migrate-From-Z1-to-RE-Mote

6.5. Start with Contiki!

Let's compile our first Contiki example! Open a terminal and write:

```
cd examples/hello-world
make TARGET=zoul savetarget
```

This will tell Contiki from now on to compile the hello world example for the RE-Mote platform. Alternatively to use the Z1 mote instead just run:

```
make TARGET=z1 savetarget
```

You need to do this only once per application. Not let's compile the application:

```
make hello-world
```

To start compiling the code (ignore the warnings), if everything works OK you should see something like:

```
CC        symbols.c
AR        contiki-z1.a
CC        hello-world.c
CC        ../../platform/z1/./contiki-z1-main.c
LD        hello-world.z1
rm obj_z1/contiki-z1-main.o hello-world.co
```

The `hello-world.z1` file should have been created and we are ready to flash the application to the device.

At any given point you can override the saved target and redefine at compilation time by running instead:

```
make TARGET=zoul hello-world
  CC        hello-world.c
  LD        hello-world.elf
arm-none-eabi-objcopy -O binary --gap-fill 0xff hello-world.elf hello-world.bin
```

This will overlook the saved `Makefile.target` file and use the target `zoul` instead.

In the following sections and chapters the compiled examples can be compiled for both Z1 and RE-Mote platform, unless specified otherwise.

> The RE-Mote takes two arguments: `TARGET` and `BOARD`, while the Z1 mote only uses the first one with the `z1` string as noted before. For the RE-Mote the `TARGET` is always `zoul` and `BOARD` is `remote`, but when we are compiling if the `BOARD` flag is not explicitly defined, it will default to the `remote`. This approach is done as there are other platforms based on the Zoul sharing the same code base and modules, and in the case of the RE-Mote, it has also its own specific platform files and definitions.

6.5.1. Hello world explained

Let's see the main components of the Hello World example. Browse the code with:

```
gedit hello-world.c
```

Or your preferred text editor.

When starting Contiki, you declare processes with a name. In each code you can have several processes. You declare the process like this:

```
PROCESS(hello_world_process, "Hello world process"); ❶
AUTOSTART_PROCESSES(&hello_world_process); ❷
```

❶ `hello_world_process` is the name of the process and `"Hello world process"` is the readable name of the process when you print it to the terminal.

❷ The `AUTOSTART_PROCESSES(&hello_world_process)` tells Contiki to start that process when it finishes booting.

```
/*---------------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data) ❶
{
  PROCESS_BEGIN(); ❷
```

```
  printf("Hello, world\n"); ❸
  PROCESS_END(); ❹
}
```

❶ You declare the content of the process in the process thread. You have the name of the process and callback functions (event handler and data handler).
❷ Inside the thread you begin the process,
❸ do what you want and
❹ finally end the process.

6.5.2. Makefile explained

Applications require a Makefile to compile, let us take a look at the `hello-world` Makefile:

```
CONTIKI_PROJECT = hello-world ❶
all: $(CONTIKI_PROJECT) ❷
CONTIKI = ../.. ❸
include $(CONTIKI)/Makefile.include ❹
```

❶ Tells the build system which application to compile
❷ If using `make all` it will compile the defined applications
❸ Specify our indentation level respect to Contiki root folder
❹ The system-wide Contiki Makefile, also points out to the platform's Makefile

We can define specific compilation flags in the Makefile, although the recommended way would be to add a `project-conf.h` header, and define there any compilation flag or value. This is done by adding this to the Makefile:

```
DEFINES+=PROJECT_CONF_H=\"project-conf.h\"
```

And then creating a `project-conf.h` header file in the example location.

Before you embark in Contiki, it is useful to remember that normally the drivers have a switch like:

```
#define DEBUG 0
#if DEBUG
#define PRINTF(...) printf(__VA_ARGS__)
#else
```

```
#define PRINTF(...)
#endif
```

Enable the `DEBUG` by changing to `1` or `DEBUG_PRINT`, this will print debug information in console. Normally you should do it in every driver file you wish to debug.

6.5.3. Adding an LED to the example

The next step is adding an LED (light emitting diode) to interact with our application.

You have to add the `dev/leds.h` which is the library to manage the LEDs. To check the available functions go to `core/dev/leds.h`.

Available LEDs commands:

```
unsigned char leds_get(void);
void leds_set(unsigned char leds);
void leds_on(unsigned char leds);
void leds_off(unsigned char leds);
void leds_toggle(unsigned char leds);
```

Normally all platforms comply to the following available LEDs:

```
LEDS_GREEN
LEDS_RED
LEDS_BLUE
LEDS_ALL
```

In the Z1 mote these LEDs are defined in `platform/z1/platform-conf.h`, as well as other hardware definitions.

The RE-Mote uses an RGB LED, basically 3-channel LEDs in a single device, allowing to show any color by the proper combination of Blue, Red and Green. In `platforms/zoul/remote/board.h` header the following are defined:

```
LEDS_LIGHT_BLUE
LEDS_YELLOW
LEDS_PURPLE
LEDS_WHITE
```

Now try to turn on only the red LED and see what happens, create a new example named `test-leds.c`:

```
#include "contiki.h"
#include "dev/leds.h"
#include <stdio.h>
/*---------------------------------------------------*/
PROCESS(test_leds_process, "Test LEDs");
AUTOSTART_PROCESSES(&test_leds_process);
/*---------------------------------------------------*/
PROCESS_THREAD(test_leds_process, ev, data)
{
  PROCESS_BEGIN();
  leds_on(LEDS_RED);
  PROCESS_END();
}
```

Now let's compile and upload the new project with:

```
make clean && make test-leds.upload
```

The `make clean` command is used to erase previously compiled objects.

Now the red LED should be on!

If you make changes to the source code, you must rebuild the files, otherwise your change might not be pulled in. It is always recommended to do a `make clean` command before compiling.

Exercise: try to switch on the other LEDs.

6.5.4. Printing messages to the console

You can use printf to visualize on the console what is happening in your application. It is really useful to debug your code, as you can print values of variables, when certain block of code is being executed, etc.

Let's try to print the status of the LED, using the `unsigned char leds_get(void);` function that is available in the documented functions (see above).

Get the LED status and print its status on the screen, modify the now existing `test-leds.c` file as follows:

```
#include "contiki.h"
#include "dev/leds.h"
#include <stdio.h>
char hello[] = "hello from the mote!";
/*---------------------------------------------------*/
PROCESS(test_leds_process, "Test LEDs");
AUTOSTART_PROCESSES(&test_leds_process);
/*---------------------------------------------------*/
PROCESS_THREAD(test_leds_process, ev, data)
{
  PROCESS_BEGIN();
  leds_on(LEDS_RED);
  printf("%s\n", hello);
  printf("The LED %u is %u\n", LEDS_RED, leds_get());
  PROCESS_END();
}
```

If one LED is on, you will get the LED number, which is defined by each platform in its `platform-conf.h` or `board.h` header.

Exercise: what happens when you turn on more than one LED? What number do you get? are these the same numbers for the Z1 and the RE-Mote?

6.5.5. Adding button events

We now want to detect if the **user button** has been pressed.

The button in Contiki is considered as a sensor. We are going to use the `core/dev/button-sensor.h` library.

The RE-Mote platform has extra button functionalities, such as detect long-press sequences, enabling to further expand the events that can be triggered using the button. Check `platform/zoul/dev/button-sensor.c` for more details, and `examples/zolertia/zoul/zoul-demo.c` for an example.

Create a new example called `test-button.c` and add the `dev/button-sensor.h` header and button event as follows:

```
#include "contiki.h"
#include "dev/leds.h"
#include "dev/button-sensor.h"
#include <stdio.h>
/*---------------------------------------------------*/
PROCESS(test_button_process, "Test button");
AUTOSTART_PROCESSES(&test_button_process);
/*---------------------------------------------------*/
PROCESS_THREAD(test_button_process, ev, data)
{
  PROCESS_BEGIN();
  SENSORS_ACTIVATE(button_sensor);
  while(1) {
    PROCESS_WAIT_EVENT_UNTIL((ev==sensors_event) &&
                             (data == &button_sensor));
    printf("I pushed the button!\n");
  }
  PROCESS_END();
}
```

This process has an infinite loop, given by the `wait()`, to wait for the button the be pressed. The two conditions have to be met (event from a sensor and that event is the button being pressed), as soon as you press the button, you get the string printed.

Exercise: switch on the LED when the button is pressed. Switch off the LED when the button is pressed again.

6.5.6. Timers

Using timers will allow us to trigger events at a given time, speeding up the transition from one state to another and making a given process or task automated, for example blinking an LED every 5 seconds, without the user having to press the button each time.

Contiki OS provides 4 kind of timers:

- Simple timer: A simple ticker, the application should check *manually* if the timer has expired. More information at `core/sys/timer.h`.
- Callback timer: When a timer expires it can callback a given function. More information at `core/sys/ctimer.h`.
- Event timer: Same as above, but instead of calling a function, when the timer expires it posts an event signalling its expiration. More information at `core/sys/etimer.h`.

- Real time timer: The real-time module handles the scheduling and execution of real-time tasks, there's only 1 timer available at the moment. More information at `core/sys/rtimer.h`

For our implementation we are going to choose the event timer, because we want to change the application behavior when the timer expires every given period.

We create a timer structure and set the timer to expire after a given number of seconds. When the timer expires we execute the code and restart the timer.

Create an example called `test-timer.c` as follows:

```
#include "contiki.h"
#include "dev/leds.h"
#include <stdio.h>
/*---------------------------------------------------*/
#define  SECONDS 2
/*---------------------------------------------------*/
PROCESS(test_timer_process, "Test timer");
AUTOSTART_PROCESSES(&test_timer_process);
/*---------------------------------------------------*/
PROCESS_THREAD(test_timer_process, ev, data)
{
  PROCESS_BEGIN();
  static struct etimer et;
  while(1) {
    etimer_set(&et, CLOCK_SECOND*SECONDS); ❶
    PROCESS_WAIT_EVENT(); ❷
    if(etimer_expired(&et)) {
      printf("Hello world!\n");
      etimer_reset(&et);
    }
  }
  PROCESS_END();
}
```

❶ `CLOCK_SECOND` is a value related to the number of the microcontroller's ticks per second. As Contiki runs on different platforms with different hardware, the value of `CLOCK_SECOND` also differs.

❷ `PROCESS_WAIT_EVENT()` waits for *any* event to happen.

Exercise: can you print the value of `CLOCK_SECOND` to count how many ticks you have in one second? Try to blink the LED for a certain

> number of seconds. A new application that starts only when the button is pressed and when the button is pressed again it stops.

6.5.7. Sensors

A sensor is a transducer whose purpose is to sense or detect a characteristic of its environments, providing a corresponding output, generally as an electrical or optical signal, related to the quantity of the measured variable.

Sensors in Contiki are implemented as follow:

```
SENSORS_SENSOR (sensor, SENSOR_NAME, value, configure, status);
```

This means a method to `configure` the sensor, poll the sensor `status` and request a `value` have to be implemented. The `sensor` structure contains pointer to these functions. The arguments for each function are shown below.

```
struct sensors_sensor {
   char *       type;
   int          (* value)     (int type);
   int          (* configure) (int type, int value);
   int          (* status)    (int type);
};
```

To better understand please refer to the `platform/zoul/dev/adc-sensors.c`, for an example of how analogue external sensors can be implemented (to be further discussed in the next sections).

The following functions and macros then can be used to work with the sensor:

```
SENSORS_ACTIVATE(sensor)    ❶
SENSORS_DEACTIVATE(sensor)  ❷
sensor.value(type);         ❸
```

❶ Enable the sensor, typically configures and turn the sensor on

❷ Disables the sensor, depending on the sensor it is recommended to save power

❸ Request a value to the sensor. As one sensor chip might have different types of readings (temperature, humidity, etc.), this is used to specify which measure to take.

Despise this is the default way to implement sensors in Contiki, depending on the sensor and developer, this may vary. Always check the specific sensor driver implementation for details. Normally sensors are provided in the same location as the example, in the `dev` folder, or in the `platform/zoul/dev` directory for example.

The `SENSORS` macro provide external linkage to the sensors defined for the platform and application, allowing to iterate between the available sensors. The following are the sensors defined as default for the RE-Mote platform:

```
SENSORS (&button_sensor, &vdd3_sensor, &cc2538_temp_sensor, &adc_sensors);
```

The button is considered a sensor in Contiki, as it shares the same sensor implementation

This macro extends to:

```
const struct sensors_sensor *sensors[] = {
  &button_sensor,
  &vdd3_sensor,
  &cc2538_temp_sensor,
  &adc_sensors,
  ((void *)0)
};
```

The number of sensors can be found with the `SENSORS_NUM` macro:

```
#define SENSORS_NUM (sizeof(sensors) / sizeof(struct sensors_sensor *))
```

Sensors can be of two types: analogue and digitals. In the following sections examples will be shown for both types, detailing how to connect and use both the platform's internal sensors and external ones.

Analogue Sensors

Analogue sensors typically require being connected to an ADC (analogue to digital converters) to translate the analogue (continuous) reading to an equivalent digital value in millivolts. The quality and resolution of the measure depends on both the ADC resolution (up to 12 bits in the Z1 and RE-Mote) and the sampling frequency.

As a rule of thumb, the sampling frequency must be twice that of the phenomenon you are measuring. As an example, if you want to sample human speech (8 kHz) you need to sample at twice that frequency (16 kHz minimum).

The Z1 mote has a built-in voltage sensor provided as an ADC input channel, as well as a generic implementation to read external analogue sensors.

Analogue sensors can be connected to both Z1 and the RE-Mote over the *phidget* ports, which are basically 3-pin connectors (Ground, VCC and signal) with 2.54 mm spacing. This is a legacy name from the Z1 release, as these ports were meant for the commercially available Phidget sensors. Nowadays there are also sensors from other providers, such as seeedstudio that have the same pin-out but with a different pin spacing (2 mm). This is not a problem, as there are cables to adapt the pin spacing.

Figure 6.3. Analogue sensors

The ADC results in the RE-Mote returns the reading in voltage.

For the Z1 mote the ADC output must be converted by means of the following formula: We multiply the measured value by the voltage reference and divide the product by the ADC's maximum output. As the resolution of the ADC is 12 bits (2^{12} = 4096), we get:

```
voltage, mv = (units * vref) / 4096;
```

The voltage reference limits the range of our measure, meaning if we use a `vref` of 2.5 V, the sensor will saturate when reading a higher voltage. The reference can be chosen while configuring the ADC, for both the Z1 and RE-Mote normally 3.3 V is used.

Both the Z1 and the RE-Mote allow up to 6 analogue sensors to be connected, but only two *phidget* connectors can be soldered at the same time.

For the RE-Mote it is possible to have one phidget connector for a 3.3 V analogue sensor (ADC1), and another for 5 V sensors (ADC3).

The RE-Mote is based on an ARM Cortex-M3, this allows to map any given pin to a controller (as opposite to the MSP430, for which a pin has

> predefined functions other than GPIO), but only pins in the PA port can be used as ADC.

For the Z1 mote is possible to have the following combinations (see Figure below):

- Two phidget connectors for 3.3 V sensors.
- Two phidget connectors for 5 V sensors.
- Two phidget connector, for 3.3 V and 5 V sensors.

JP1A

Features	MSP430 Port#	Pin Name	Pin#		Pin Name	MSP430 Port#	Features
Phidget powered @5V, ADC input has resistor divider to allow 5V inputs		**ADCGND**	1	2	**ADCGND**		*Phidget @3V*
		USB+5V	3	4	**Vcc+3V**		
	P6.3	ADC3*	5	6	ADC7	P6.7	
	P6.4	ADC4	7	8	ADC6/DAC0	P6.6	
	P6.2	ADC2	9	10	ADC5/DAC1	P6.5	
Phidget powered @5V, ADC input has resistor divider to allow 5V inputs		**ADCGND**	11	12	**ADCGND**		*Phidget @3V*
		USB+5V	13	14	**Vcc+3V**		
	P6.0	ADC0*	15	16	ADC1	P6.1	

Table 3 — JP1A Pinout description

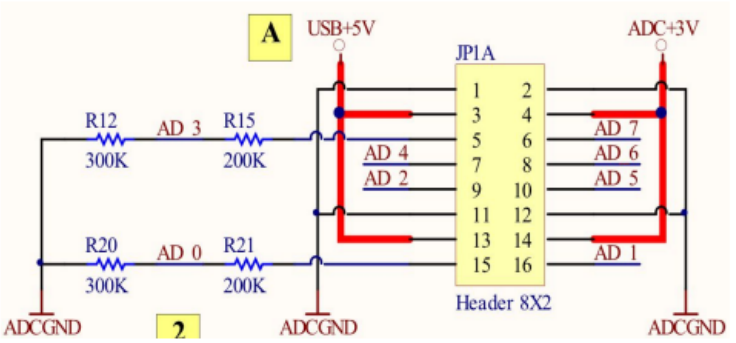

Figure 6.4. Pin assignement

As shown in the snippet below, there are mapped as default the ADC channels ADC0 (5V1), ADC3 (5V2), ADC1 (3V1) and ADC7 (3V2) to be used as input for external analogue sensors.

```
/* MemReg6 == P6.0/A0 == 5v 1 */
ADC12MCTL0 = (INCH_0 + SREF_0);
/* MemReg7 == P6.3/A3 == 5v 2 */
ADC12MCTL1 = (INCH_3 + SREF_0);
/* MemReg8 == P6.1/A1 == 3v 1 */
ADC12MCTL2 = (INCH_1 + SREF_0);
/* MemReg9 == P6.7/A7 == 3v_2 */
```

```
ADC12MCTL3 = (INCH_7 + SREF_0);
```

To take a measure from a sensor connected to the ADC7 channel of the Z1 mote, in my code I would need to make the following steps:

```
#include "dev/z1-phidgets.h"                                    ❶
SENSORS_ACTIVATE(phidgets);                                     ❷
printf("Phidget 5V 1:%d\n", phidgets.value(PHIDGET3V_2));       ❸
```

❶ Include the driver header

❷ Enable the ADC sensors, as default all 4 ADC channels are enabled

❸ Request a reading

The Z1 is powered at 3.3V, but when connected over the USB (at 5V) allows to connect 5V sensors to the phidget ports, namely 5V1 (ADC3) and 5V2 (ADC0) as there is a voltage divider in the input to adapt the reading from 5V to 3.3V.

Details about the Z1 implementation of the analogue sensors driver is available at `platform/z1/dev/z1-phidgets.c`. There is also a driver for the MSP430 internal voltage sensor at `platform/z1/dev/battery-sensor.c`, with an example at `examples/zolertia/z1/test-battery.c`.

Check the ADC implementation with an example, let's connect the Phidget 1142 precision Light Sensor.

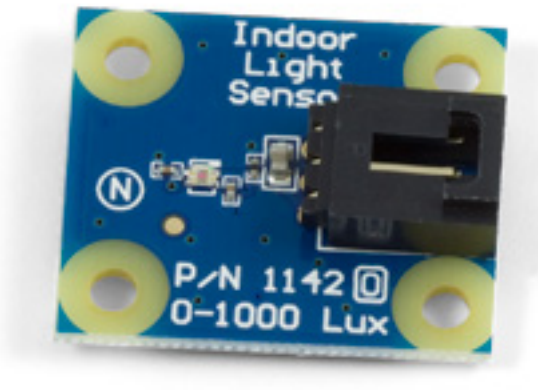

Figure 6.5. Light sensor

There is an example called `test-phidgets.c` in `examples/zolertia/z1`. This will read values from an analog sensor and print them to the terminal.

Connect the light sensor to the 3V2 Phidget connector and compile the example:

```
make clean && make test-phidgets.upload && make z1-reset && make login
```

You can chain commands to be executed one after another, in this case the `make z1-reset` restarts the mote after flashed, then `make login` prints to console the output of any `printf` command in our code. You may need to use the argument `MOTES=/dev/ttyUSB[0…9]` to specify a mote over a specific USB port.

This is the result:

```
Starting 'Test Button & Phidgets'
Please press the User Button
Phidget 5V 1:123
Phidget 5V 2:301
Phidget 3V 1:1710
Phidget 3V 2:2202
```

The light sensor is connected to the 3V2 connector, so the raw value is 2202. Try to illuminate the sensor with a flashlight (from your mobile phone, for example) and then cover it with your hand so that no light can reach it.

From the Phidget website we have the following information about the sensor:

Parameter	**Value**
Sensor type	Light
Light level min	1 lux
Supply Voltage Min	2.4 V
Supply Voltage Max	5.5 V
Max current consumption	5mA
Light level max (3.3 V)	660 lux
Light level max (5 V)	**1000 lux**

As you can see, the light sensor can be connected to either the 5 V or 3.3 V *Phidget* connector. The max measurable value changes depending where you connect it.

The formula to translate SensorValue into luminosity is: `Luminosity(lux)=SensorValue`

The **RE-Mote platform** also allows to connect 5V sensors to the ADC3 port, using the same voltage divider as the Z1 mote. As seen in the next snippet, the ADC3 port is mapped to the PA2 pin.

```
/*
 * This driver supports analogue sensors connected to ADC1, ADC2 and AND3 inputs
 * This is controlled by the type argument of the value() function. Possible
 * choices are:
 * - REMOTE_SENSORS_ADC1 (channel 5)
 * - REMOTE_SENSORS_ADC2 (channel 4)
 * - REMOTE_SENSORS_ADC3 (channel 2)
 */
```

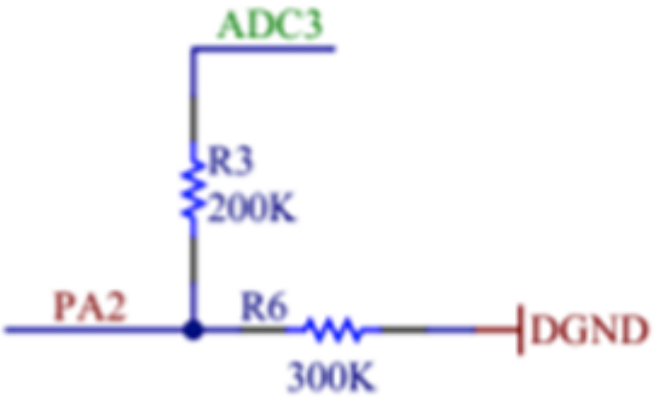

Figure 6.6. Connecting sensor

Then to read data from an attached sensor (as we did in the previous example):

```
#include "dev/zoul-sensors.h"                                          ❶
adc_sensors.configure(SENSORS_HW_INIT, REMOTE_SENSORS_ADC_ALL);        ❷
printf("Phidget ADC2 = %d raw\n", adc_sensors.value(REMOTE_SENSORS_ADC2)); ❸
printf("Phidget ADC3 = %d raw\n", adc_sensors.value(REMOTE_SENSORS_ADC3));
```

❶ Include the ADC driver header

❷ Enable and configure the ADC channels (can selectively choose which to enable)

❸ Request a reading

Check the RE-Mote example at `example/zolertia/zoul/zoul-demo.c` for more details.

Exercise: make the sensor take sensor readings as fast as possible. Print on the screen the ADC raw values and the millivolts (as this sensor is linear, the voltage corresponds to the luxes). What are the max and min values you can get? What is the average light value of the room? Create an application that turns the red LED on when it is dark. When it is light, turn the green LED on. In between, switch off all the LEDs. Add a timer and measure the light every 10 seconds.

Digital Sensors

> Digital sensors are normally interfaced over a digital communication protocol such as I2C, SPI, 1-Wire, Serial or depending on the manufacturer, a proprietary protocol normally on a ticking clock.
>
> Digital sensors allow a more extended set of commands (turn on, turn off, configure interrupts). With a digital light sensor for example, you could set a threshold value and let the sensor send an interrupt when reached, without the need for continuous polling.
>
> Remember to check the specific sensor information and data sheet for more information.

The **Z1 mote** has **two built in digital sensors**: temperature and 3-axis accelerometer. Let us start with this last one, there is an example called `test-adxl345.c` available for testing.

The ADXL345 is an I2C ultra-low power sensor able to read up to 16g, well suited for mobile device applications. It measures the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion or shock. Its high resolution (3.9mg/LSB) enables measurement of inclination changes less than 1.0°.

```
[37] DoubleTap detected! (0xE3) -- DoubleTap Tap
x: -1 y: 12 z: 223
[38] Tap detected! (0xC3) -- Tap
x: -2 y: 8 z: 220
x: 2 y: 4 z: 221
x: 3 y: 5 z: 221
x: 4 y: 5 z: 222
```

The accelerometer can give data in x, y and z axis and has three types of interrupts: a single tap, a double tap and a free-fall (pay attention not to damage the mote!).

The code has two threads, one for the interruptions and the other for the LEDs. When Contiki starts, it triggers both processes.

The `led_process` thread triggers a timer that waits before turning off the LEDs. This is mostly done to filter the rapid signal coming from the accelerometer. The other process is the acceleration. It assigns the callback for the `led_off` event. Interrupts can happen at any given time, are non periodic and totally asynchronous.

Interrupts can be triggered by external sources (sensors, GPIOs, *Watchdog Timer*, etc) and should be cleared as soon as possible. When an interrupts happens, the interrupt handler

(which is a process that checks the interrupt registers to find out which is the interrupt source) manages it and forwards it to the subscribed callback.

In this example, the accelerometer is initialized and then the interrupts are mapped to a specific callback functions. Interrupt source 1 is mapped to the *free fall* callback handler and the tap interrupts are mapped to the interrupt source 2.

```
/*
 * Start and setup the accelerometer with default
 * values, _i.e_ no interrupts enabled.
 */
accm_init();
/* Register the callback functions */
ACCM_REGISTER_INT1_CB(accm_ff_cb);
ACCM_REGISTER_INT2_CB(accm_tap_cb);
```

We then need to enable the interrupts like this:

```
accm_set_irq(ADXL345_INT_FREEFALL,
             ADXL345_INT_TAP +
             ADXL345_INT_DOUBLETAP);
```

In the *while* loop we read the values from each axis every second. If there are no interrupts, this will be the only thing shown in the terminal.

Exercise: put the mote in different positions and check the values of the accelerometer. Try to understand what is x, y and z. Measure the maximum acceleration by shaking the mote. Turn on and off the LED according to the acceleration on one axis.

For the next example we will use the **ZIG-SHT25**, an I2C digital temperature and humidity sensor based on the SHT25 sensor from Sensirion.

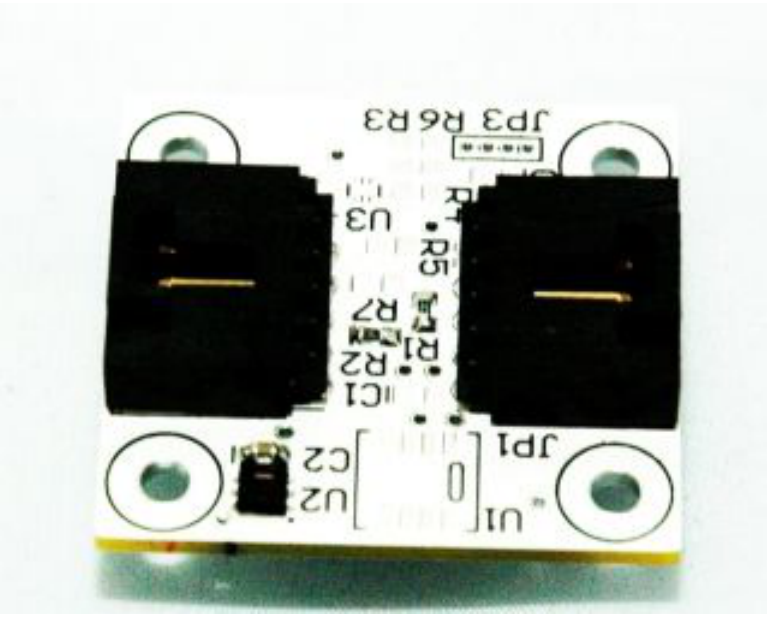

Figure 6.7. Temperature and humidity sensor

Parameter	Value
Sensor type	Temperature and Humidity
Supply Voltage [V]	2.1 - 3.6
Energy Consumption	3.2 uW (at 8 bit, 1 measurement / s)
Data range	0-100 % RH (humidity), -40-125ºC (temperature)
Max resolution	14 bits (temperature), 12 bits (humidity)
Max current consumption	**300 uA**

The ZIG-SHT25 sensor example is available in both `examples/zolertia/zoul/test-sht25.c` and `examples/zolertia/z1/test-sht25.c`, an output example is given below:

```
Starting 'SHT25 test'
Temperature 23.71 ºC
Humidity 42.95 % RH
Temperature 23.71 ºC
Humidity 42.95 % RH
Temperature 23.71 ºC
Humidity 42.95 % RH
```

```
Temperature 23.71 °C
Humidity 42.98  %RH
```

As usual you can check the driver implementation at the `platform/z1/dev` and `platform/zoul/dev` directories. Check out these locations for more available sensors, and examples to guide you to implement your own.

Exercise: convert the temperature to Fahrenheit. Try to get the temperature and humidity as high as possible (without damaging the mote!). Try to print only "possible" values (if you disconnect the sensor, you should not print anything, or print an error message!).

6.6. Emulate Contiki with Cooja

Cooja is the Contiki network emulator. Cooja allows large and small networks of Contiki motes to be emulated at the hardware level, which is slower but allows precise inspection of the system behavior, or at a less detailed level, which is faster and allows simulation of larger networks.

Cooja is a highly useful tool for Contiki development as it allows developers to test their code and systems long before running it on the target hardware. Developers regularly set up new simulations both to debug their software and to verify the behavior of their systems.

Most of the examples done in sections below can be emulated in Cooja as well (except the sensor ones).

To start Cooja, in the terminal window go to the Cooja directory:

```
cd contiki/tools/cooja
```

Start Cooja with the command:

```
ant run
```

When Cooja is compiled, it will start with a blue empty window. Now that Cooja is up and running, we can try it out with an example simulation.

Go to `File`, `Open Simulation`, `Browse` then navigate to the `examples/hello-world` and select the `hello-world-example.csc`. This will load the previously saved hello world example.

If you want to edit an example (i.e. to use a different type of platform), instead of choosing the `Browse` option, select `Open and Reconfigure`, this will walk you through the steps to configure the example.

6.7. Create a new simulation

Click the `File` menu and click `New simulation`. Cooja now opens up the `Create new simulation` dialog. In this dialog, we may choose to give our simulation a new name, but for this example, we'll just stick with `My simulation`. Leave the other options set as default. Click the `Create` button.

Cooja brings up the new simulation. You can choose what you want to visualize by using the `Tools` menu. The `Network` window shows all the motes in the simulated network, it should be empty now since we have no motes in our simulation. The `Timeline` window shows all communication events in the simulation over time - very handy for understanding what goes on in the network. The `Mote output` window shows all serial port printouts from all the motes. The `Notes` window is where we can put notes for our simulation. And the `Simulation control` window is where we start, pause, and reload our simulation.

6.8. Add motes to the simulation

Before we can simulate our network, we must add one or more motes. We do this via the `Motes` menu, where we click on `Add motes`. Since this is the first mote we add, we must first create a mote type to add. Click `Create new mote type` and select one of the available mote types. For this example, we click `Z1 mote`. to create an emulated Z1 mote type. Cooja opens the `Create Mote Type` dialog, in which we can choose a name for our mote type as well as the Contiki application that our mote type will run.

For this example, we stick with the suggested name, and instead click on the `Browse` button on the right hand side to choose our Contiki application.

Chapter 7. Wireless with Contiki

In the previous section we covered some of the core features of Contiki, basics of sensors and a general overview of how the applications are built, programmed and simulated in Contiki. This section introduces the wireless communication, details about radios, and basics ideas about configuring our platforms.

7.1. Preparing your device

The very first step is understanding how our platform is configured.

Each platform implements its own set of default values and configurations, to be used by underlying modules like the radio, serial port, etc.

The places to go for the Zolertia platform are the following:

- Specific hardware settings: parameters such as the default I2C pins, ADC channels, module-specific pin assignment and platform information can be found at `platform/zoul/remote/dev/board.h` and `platform/z1/platform-conf.h`.
- Specific Contiki settings: UART settings, MAC driver, radio channel, IPv6, RIME and network buffer configuration, among others, can be found at `platform/zoul/contiki-conf.h` and `platform/z1/contiki-conf.h`.

As a general good practice, user configurable parameters are normally allowed to be overridden by the applications, this also serves as a guideline to discern which values can be changed by the casual user, from those meant to be changed only if you really know what you are doing. Below is an example:

```
#ifndef UART0_CONF_BAUD_RATE
#define UART0_CONF_BAUD_RATE 115200
#endif
```

By defining `UART0_CONF_BAUD_RATE` in our application's `project-conf.h`, we can change the default 115200 bps baud rate. Notice that generally it is a good practice to add `CONF` to the user configurable parameters.

One of the most used tools is probably `grep`, a handy command to search for a text string in any document or location. One way to run this command is `grep -lr "alinan .`, if executed at the root of our

> Contiki installation, it will list recursively all the files authored by Antonio Linan. This is a good way to check the location of a specific definition, if you are not using an IDE like Eclipse. Check `man grep` for more information.
>
> For windows Astrogrep[1] is a good option.

In the next section we will review the most notable parameters to configure, but as usual depending on your application and setup, the best way to ensure everything is properly set is by reviewing the specific platform configuration files, and modify or redefine accordingly.

7.1.1. Device addressing

To start working you must first define the Node ID of each node, this will be used to generate the mote's MAC address and the IPv6 addresses (link-local and global).

RE-Mote addresses

The **RE-Mote** platform comes with two pre-loaded MAC addresses stored in its internal flash memory, but the user can instead choose a hardcoded one. The following switches at `platform/zoul/contiki-conf.h` selects the chosen one.

```
/**
  * \name IEEE address configuration
  *
  * Used to generate our RIME & IPv6 address
  * @{
  */
/**
  * \brief Location of the IEEE address
  * 0 => Read from InfoPage,
  * 1 => Use a hardcoded address, configured by IEEE_ADDR_CONF_ADDRESS
  */
#ifndef IEEE_ADDR_CONF_HARDCODED
#define IEEE_ADDR_CONF_HARDCODED              0
#endif

/**
 * \brief Location of the IEEE address in the InfoPage when
 * IEEE_ADDR_CONF_HARDCODED is defined as 0
 * 0 => Use the primary address location
```

[1] http://astrogrep.sourceforge.net/

```
 * 1 => Use the secondary address location
 */
#ifndef IEEE_ADDR_CONF_USE_SECONDARY_LOCATION
#define IEEE_ADDR_CONF_USE_SECONDARY_LOCATION 0
#endif
```

If using your own hardcoded address, the following define can be overridden by the application:

```
#ifndef IEEE_ADDR_CONF_ADDRESS
#define IEEE_ADDR_CONF_ADDRESS { 0x00, 0x12, 0x4B, 0x00, 0x89, 0xAB, 0xCD, 0xEF }
#endif
```

Z1 mote addresses

Let's use the ID from the mote list:

```
Reference  Device       Description
-------------------------------------------------
Z1RC3301   /dev/ttyUSB0 Silicon Labs Zolertia Z1
```

The node ID should be `3301` (decimal) if no previously saved node ID is found in the flash memory.

Let's see how Contiki uses this to derive a full IPv6 and MAC address. At `platforms/z1/contiki-z1-main.c`

```
#ifdef SERIALNUM
  if(!node_id) {
    PRINTF("Node id is not set, using Z1 product ID\n");
    node_id = SERIALNUM;
  }
#endif
node_mac[0] = 0xc1; /* Hardcoded for Z1 */
node_mac[1] = 0x0c; /* Hardcoded for Revision C */
node_mac[2] = 0x00; /* Hardcoded to arbitrary even number so that the 802.15.4 MAC
 address is compatible with an Ethernet MAC address - byte 0 (byte 2 in the DS ID)
 */
node_mac[3] = 0x00; /* Hardcoded */
node_mac[4] = 0x00; /* Hardcoded */
node_mac[5] = 0x00; /* Hardcoded */
node_mac[6] = node_id >> 8;
node_mac[7] = node_id & 0xff;
}
```

So the mote should have the following addresses:

```
MAC c1:0c:00:00:00:00:0c:e5
Node id is set to 3301.
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:0ce5
```

Where `0xce5` is the hex value corresponding to `3301`. The global address is only set when an IPv6 prefix is assigned (by now you should know this from earlier sections).

If instead you wish to have your own addressing scheme, you can edit the node_mac values at `contiki-z1-main.c` file. If you wish to replace the node id value obtained from the product id, you need to store a new one in the flash memory, luckily there is already an application to do so:

Go to `examples/zolertia/z1` location and replace the `158` for your own required value:

```
make clean && make burn-nodeid.upload nodeid=158 nodemac=158 && make z1-reset &&
 make login
```

You should see the following:

```
MAC c1:0c:00:00:00:00:0c:e5 Ref ID: 3301
Contiki-2.6-1803-g03f57ae started. Node id is set to 3301.
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:0ce5
Starting 'Burn node id'
Burning node id 158
Restored node id 158
```

As you can see, now the node ID has been changed to 158, when you restart the mote you should see that the changes have been applied:

```
MAC c1:0c:00:00:00:00:00:9e Ref ID: 3301
Contiki-2.6-1803-g03f57ae started. Node id is set to 158.
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:009e
```

7.1.2. Set the bandwidth and channel

The bandwidth and allowed channels depend on the operating frequency band. They will be determined by the spectrum regulation agency of the country, along with maximum transmitted power allowable. .**The IEEE 802.15.4 standard**

The IEEE 802.15.4 is a standard for wireless communication, it specifies the physical and media access control layers for low-rate wireless personal area networks (LR-WPANs).

The standard specifies the use of the 868-868.8 MHz (in Europe and many other countries), the 902-928 MHz (in United States, Canada, and some Latin America countries), or the more world-wide 2.400-2.4835 GHz band part of the Industrial Scientific and Medical applications (ISM).

OPTIONS FOR FREQUENCY ASSIGNMENTS			
Geographical regions	Europe	Americas	Worldwide
Frequency assignment	868 to 868.6 MHz	902 to 928 MHz	2.4 to 2.4835 GHz
Number of channels	1	10	16
Channel bandwidth	600 kHz	2 MHz	5 MHz
Symbol rate	20 ksymbols/s	40 ksymbols/s	62.5 ksymbols/s
Data rate	20 kbits/s	40 kbits/s	250 kbits/s
Modulation	BPSK	BPSK	Q-QPSK

Figure 7.1. IEEE 802.15.4 2.4 GHz regulation requirements (electronicdesign.com, 2013[2])

In practice the 2.4 GHz band is being heavily used due to its world-wide availability. The ZigBee proprietary protocol by the ZigBee alliance was one of the early adopters of IEEE 802.15.4. It did so leveraging the physical and MAC layer of IEEE 802.15.4, specifying on top additional routing and networking functionality to build mesh networks.

Quite recently the Thread Group[3] has proposed its own simplified IPv6-based mesh networking protocol for connecting products around the home to each other, to the Internet and to the cloud. This will surely boost the adoption of IEEE 802.15.4 and IPv6.

[2] http://electronicdesign.com/what-s-difference-between/what-s-difference-between-ieee-802154-and-zigbee-wireless
[3] http://threadgroup.org/

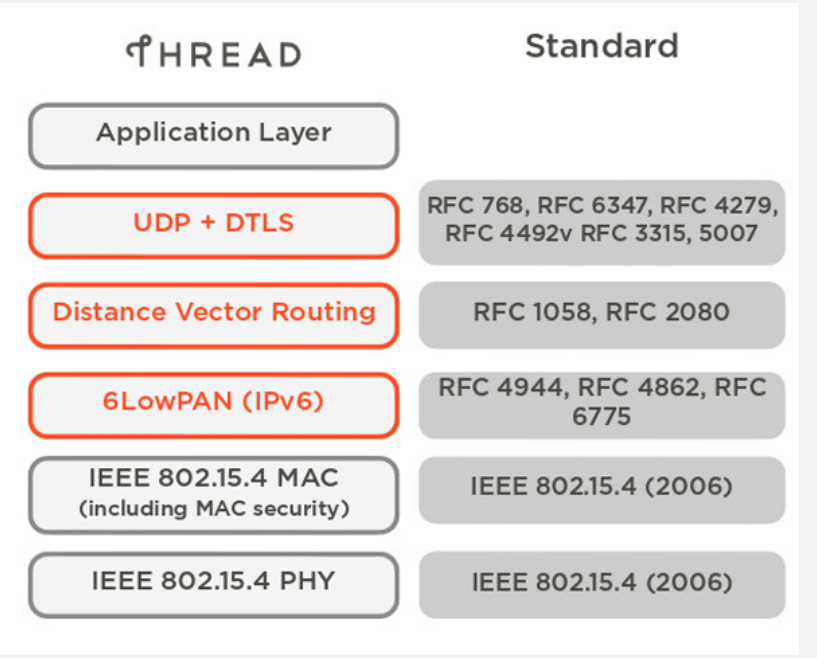

Figure 7.2. Thread layers and standards (Thread group, 2015[4])

Working at 2.4 GHz

As the 2.4 GHz band is also used by other technologies like WiFi and Bluetooth, this spectrum is shared and overlaps might occur. The Figure below shows the channel allocation of the 2.4 GHz IEEE 802.15.4, and the recommended channels to avoid interferences with other co-located devices. With the rise of the Bluetooth Low Energy, and the ubiquitous WiFi present in our lives, the selection of a proper operating channel is crucial in any deployment.

[4] http://threadgroup.org/

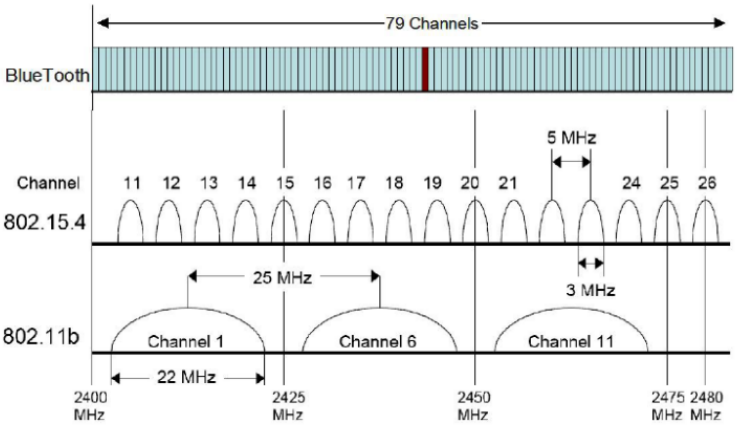

Figure 7.3. Channel assignment

The default channel of the **RE-Mote** is defined as follows:

```
#ifndef CC2538_RF_CONF_CHANNEL
#define CC2538_RF_CONF_CHANNEL              26
#endif /* CC2538_RF_CONF_CHANNEL */
```

The **Z1 mote** defines its default channel as:

```
#ifdef RF_CHANNEL
#define CC2420_CONF_CHANNEL RF_CHANNEL
#endif

#ifndef CC2420_CONF_CHANNEL
#define CC2420_CONF_CHANNEL              26
#endif /* CC2420_CONF_CHANNEL */
```

The radio channel can be defined from the application's `project-conf.h` or the `Makefile`.

The radio drivers in Contiki are implemented to comply with the `struct radio_driver` in `core/dev/radio.h`. This abstraction allows to interact with the radio using a standardized API, independently of the radio hardware. The functions to set and read radio parameters are explained below.

```
/** Get a radio parameter value. */
radio_result_t (* get_value)(radio_param_t param, radio_value_t *value);
```

```
/** Set a radio parameter value. */
radio_result_t (* set_value)(radio_param_t param, radio_value_t value);
```

To change the channel from the application use `RADIO_PARAM_CHANNEL` as follows:

```
rd = NETSTACK_RADIO.set_value(RADIO_PARAM_CHANNEL, value);
```

Where `value` can be any value from 11 to 26, and rd will be either `RADIO_RESULT_INVALID_VALUE` or `RADIO_RESULT_OK`.

Working at 863-950 MHz

The **RE-Mote** has a dual 2.4 GHz and 863-950 MHz RF interface, which can be selected alternatively, or used simultaneously (the latter currently not supported in Contiki at the moment).

As default the RE-Mote uses the IEEE 802.15.4g mandatory mode for the 868 MHz band, configured for 2-GFSK modulation, 50 kbps data rate and with 33 channels available.

The RE-Mote uses the Texas Instruments CC1200 RF transceiver, referred to in Contiki as `dev/cc1200`. The default configuration file is located in `dev/cc1200/cc1200-802154g-863-870-fsk-50kbps.c`.

To change channels from the application we use the RF API:

```
rd = NETSTACK_RADIO.set_value(RADIO_PARAM_CHANNEL, value);
```

Where `value` can be any value from 11 to 26, and rd it will be either `RADIO_RESULT_INVALID_VALUE` or `RADIO_RESULT_OK`.

7.1.3. Set the transmission power

The radio frequency power transmission is that at the output of the transmitter before reaching the antenna. The higher the transmission power the higher the wireless range, but the power consumption usually increases as well. The range is heavily dependent on the frequency, the antenna used and its height above the ground.

Changing the transmission power for the Z1 (CC2420) and the RE-Mote (CC2538)

The **RE-Mote** platform uses the CC2538 built-in 2.4 GHz radio. As default the transmission power is set to 3 dBm (2 mW) in the `cpu/cc2538/cc2538-rf.h` header as shown below.

```
CC2538_RF_TX_POWER_RECOMMENDED 0xD5
```

This recommended value is taken from the SmartRF Studio[5].

Other values and its corresponding output power levels are shown in the next table.

Table 7.1. CC2538 Transmission power recommended values (from SmartRF Studio[6])

TX Power (dBm)	Value
+7	0xFF
+5	0xED
+3	0xD5
+1	0xC5
0	0xB6
-1	0xB0
-3	0xA1
-5	0x91
-7	0x88
-9	0x72
-11	0x62
-13	0x58
-15	0x42
-24	**0x00**

[5] http://www.ti.com/tool/smartrftm-studio&DCMP=hpa_rf_general&HQS=Other+OT+smartrfstudio
[6] http://www.ti.com/tool/smartrftm-studio&DCMP=hpa_rf_general&HQS=Other+OT+smartrfstudio

As illogical as it may sound, there might be some reasons to reduce transmission power:

- To reduce the power consumption.
- To test a multi-hop network without placing the nodes too far (too much power can saturate the receiver).
- To avoid interference among co-located networks in the same area.

The current consumption can go from 24 mA to 34 mA when changing the transmission power from 0dBm to 7dBm (AN125[7]).

The **Z1 mote** uses the Texas Instrument CC2420 RF transceiver. As default the transmission power is set to 0 dBm (1 mW), which is the maximum allowed by the radio.

The available output power and its corresponding configuration values are listed in the table below, as well as the current consumption at each level.

Table 7.2. CC2420 Transmission power (CC2420 datasheet, page 51[8])

TX Power (dBm)	Value	mA
0	31	17.4
-1	27	16.5
-3	23	15.2
-5	19	13.9
-7	15	12.5
-10	11	11.2
-15	7	9.9
-25	**3**	**8.5**

For both platforms the transmission power can be changed with:

```
rd = NETSTACK_RADIO.set_value(RADIO_PARAM_TXPOWER, value);
```

[7] http://www.ti.com/lit/an/swra437/swra437.pdf
[8] http://www.ti.com/lit/ds/symlink/cc2420.pdf

Where `value` can be any value from the above table, and rd it will be either `RADIO_RESULT_INVALID_VALUE` or `RADIO_RESULT_OK`.

Changing the transmission power for the RE-Mote (CC1200)

As mentioned earlier, the **RE-Mote** has an on-board sub-1 GHz interface based on the CC1120 radio transceiver, configured to operate in the 863-950 MHz bands. The maximum transmission power allowed depends on the specific band and regulations in place, which can also impose limits on the maximum antenna gain, be sure to check the local regulations before changing the output power.

The regulations are country specific and out of the scope of this section.

The following values are taken from the SmartRF Studio[9], using default IEEE 802.15.4g ETSI compliant configuration.

Table 7.3. CC1200 Transmission power recommended values (from SmartRF Studio[10])

TX Power (dBm)	Value
+14	0x7F
+13	0x7C
+12	0x7A
+11	0x78
+8	0x71
+6	0x6C
+4	0x68
+3	0x66
+2	0x63
+1	0x61
0	0x5F
-3	0x58
-40	**0x41**

[9] http://www.ti.com/tool/smartrftm-studio&DCMP=hpa_rf_general&HQS=Other+OT+smartrfstudio
[10] http://www.ti.com/tool/smartrftm-studio&DCMP=hpa_rf_general&HQS=Other+OT+smartrfstudio

TX Power (dBm)	Value
-6	0x51
-11	0x46
-24	0x42
-40	**0x41**

These values correspond to the `CC1200_PA_CFG1` register.

As default the CC1200 driver[11] in Contiki starts with the maximum transmission power, defined as follows:

```
/* The maximum output power in dBm */
#define RF_CFG_MAX_TXPOWER              CC1200_CONST_TX_POWER_MAX
```

The minimum and maximum allowed values are set in `dev/cc1200/cc1200-const.h` as shown below.

```
/* Output power in dBm */
/* Up to now we don't handle the special power levels PA_POWER_RAMP < 3, hence
 * the minimum tx power is -16. See update_txpower().
 */
#define CC1200_CONST_TX_POWER_MIN       (-16)
/*
 * Maximum output power will probably depend on the band we use due to
 * regulation issues
 */
#define CC1200_CONST_TX_POWER_MAX       14
```

The CC1200 driver calculates the proper `CC1200_PA_CFG1` register value, so we need to pass as `value` argument the required transmission power.

```
rd = NETSTACK_RADIO.set_value(RADIO_PARAM_TXPOWER, value);
```

Where `value` can be any value from -14 to 16, and rd will be either `RADIO_RESULT_INVALID_VALUE` or `RADIO_RESULT_OK`.

[11] https://github.com/contiki-os/contiki/blob/master/dev/cc1200/cc1200.c

7.1.4. Checking the wireless link

Due to the changing environment conditions that normally affect the wireless systems, such as rain, interferences, obstacles, etc., measuring the wireless medium and links quality is important.

Checking the wireless medium should be done in three stages: before deploying your network, at deployment phase and later at network runtime, to ensure that the nodes create and select the best available routes.

Link Quality Estimation

Link Quality Estimation is an integral part of assuring reliability in wireless networks. Various link estimation metrics have been proposed to effectively measure the quality of wireless links.

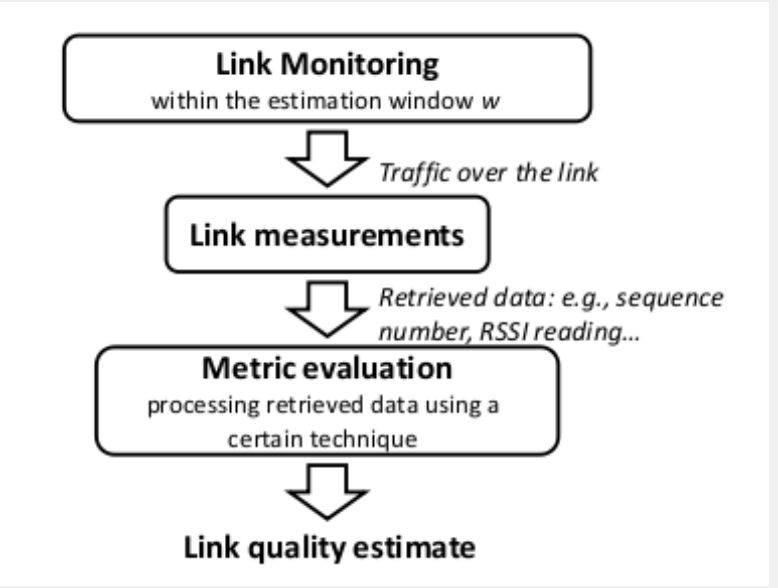

Figure 7.4. Link quality estimation process

The ETX metric, or expected transmission count, is a measure of the quality of a path between two nodes in a wireless packet data network. ETX is the number of expected transmissions of a packet necessary for it to be received without error at its destination. This number varies from one to infinity. An ETX of one indicates a perfect transmission medium, where an ETX of infinity represents a completely non-functional link. Note

that ETX is an expected transmission count for a future event, as opposed to an actual count of a past events. It is hence a real number, generally not an integer.

ETX can be used as the routing metric. Routes with a lower metric are preferred. In a route that includes multiple hops, the metric is the sum of the ETX of the individual hops.

Below we describe how to read the LQI and RSSI to have a first approximation of the link conditions.

What is RSSI?

RSSI (Received Signal Strenght Indicator) is a generic radio receiver technology metric used internally in a wireless networking device to determine the amount of radio energy received in a given channel. The end-user will likely observe an RSSI value when measuring the signal strength of a wireless network through the use of a wireless network monitoring tool like Wireshark, Kismet or Inssider.

The image below shows how the Packet Reception Rate (PRR) dramatically decreases as the CC2420 RSSI values worsen.

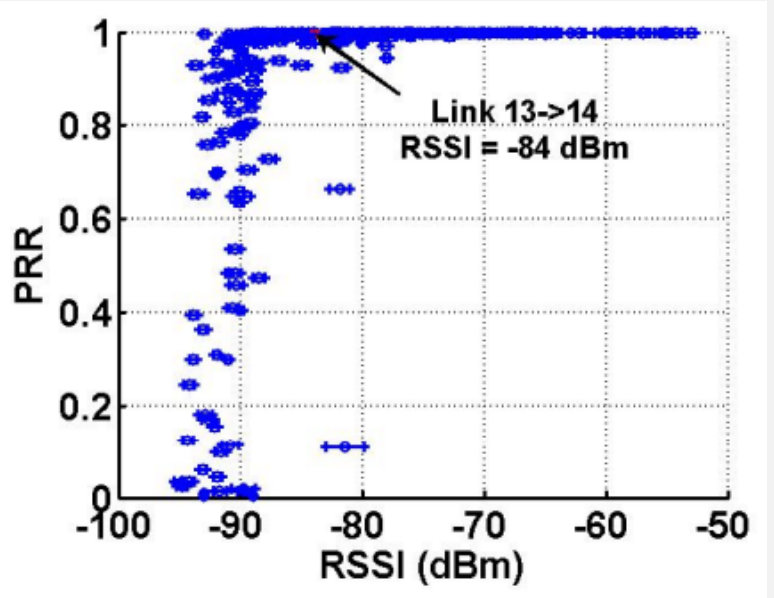

Figure 7.5. Packet rejection rate versus received signal strenght indicator

There is no standardized relationship of any particular physical parameter to the RSSI reading, Vendors and chipset makers provide their own accuracy, granularity, and range for the actual power (measured in mW or dBm) and their range of RSSI values.

There are 2 different types of RSSI readings available:

- The first one is an indication of the amount of power present in the wireless medium at the given frequency and at given time. In the absence of any packet in the air, this will be the noise floor. This measurement is also used to decide if the medium is free, and available to send a packet. A high value could be due to interference or to the presence of a packet in the air.
- The second measurement is performed only after a packet has been correctly decoded, and gives the strength of the packet received from a specific node.

The first measurement can be read using the radio API as follows:

```
rd = NETSTACK_RADIO.get_value(RADIO_PARAM_RSSI, value);
```

Where `value` is a variable passed as a pointer to store the RSSI value, and rd it will be either `RADIO_RESULT_INVALID_VALUE` or `RADIO_RESULT_OK`.

To read the RSSI value of a correctly decoded received packet, at the `receive` callback:

```
packetbuf_attr(PACKETBUF_ATTR_RSSI);
```

More information about the `packetbuf` attributes is available in `core/net/packetbuf.h`.

For the CC2420 radio frequency transceiver on the **Z1 mote**, the RSSI can range from 0 to -100, values close to 0 mean good links while values close to -100 are indicators of a bad link, which could be due to multiple factors such as distance, environment, obstacles, interferences, etc.

What is LQI?

LQI (Link Quality Indicator) is a digital value often provide by Chipset vendors as an indicator of how well a signal is demodulated, in terms of the strength and quality of the received packet, thus indicating a good or bad wireless medium.

The example below shows how the Packet Reception Rate decreases as the LQI decreases.

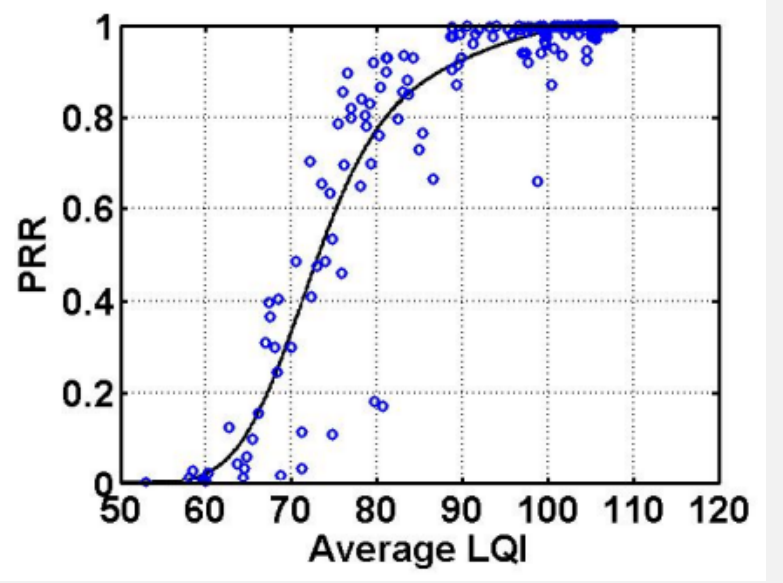

Figure 7.6. Packet rejection rate versus link quality indicator

To read the LQI value we use the Radio API:

```
rd = NETSTACK_RADIO.get_value(PACKETBUF_ATTR_LINK_QUALITY, value);
```

Where `value` is a variable passed as a pointer to store the LQI value, and rd it will be either `RADIO_RESULT_INVALID_VALUE` or `RADIO_RESULT_OK`.

The CC2420 radio used by the Z1 mote typically ranges from 110 (indicates a maximum quality frame) to 50 (typically the lowest quality frames detectable by the transceiver).

Detailed information about the CC2538 LQI calculation is found in the CC2538 user guide[12].

7.2. Configure the MAC layer

MAC protocols

Medium Access Control (MAC) protocols describe the medium access adopted in a network, by establishing the rules that specify when a given node is allowed to transmit packets.

Protocols can be classified as contention-based or reservation-based protocols.

The first are based on Carrier Sensing for detecting medium activity and are prone to collisions and lower efficiency at heavy loads, but are easy to implement. The second group is efficient in terms of throughput and energy, but require precise synchronization and is less adaptable to dynamic traffic.

The medium access implementation in Contiki has 3 different layers: Framer, Radio Duty-Cycle (RDC) and Medium Access Control (MAC).

[12] http://www.ti.com/lit/ug/swru319c/swru319c.pdf

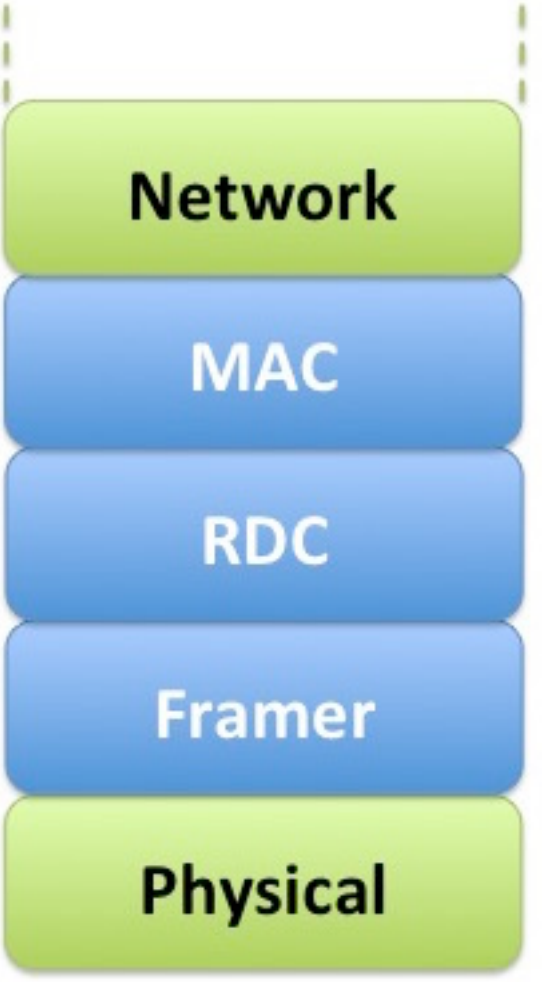

Figure 7.7. Contiki MAC stack[13]

The network layer can be accessed through the global variables `NETSTACK_FRAMER`, `NETSTACK_RDC` and `NETSTACK_MAC`, which are defined at compilation time.

The variables are located in `core/net/netstack.h`, and can be defined by each platform as default and overridden by applications.

7.2.1. MAC driver

Contiki provides two MAC drivers: CSMA and NullMAC

[13] http://anrg.usc.edu/contiki/index.php/MAC_protocols_in_ContikiOS

CSMA (Carrier-Sense Multiple Access) receives incoming packets from the RDC layer and uses the RDC layer to transmit packets. If the RDC layer or the radio layer detects that the medium is busy, the MAC layer may retransmit the packet at a later point in time. CSMA protocol keeps a list of packets sent to each of the neighbors and calculate statistics such as number of retransmissions, collisions, deferrals, etc. The medium access check is performed by the RDC driver.

NullMAC is a simple pass-through protocol. It calls the appropriate RDC functions.

As default both **Z1 mote** and **RE-Mote** uses the CSMA driver.

```
#ifndef NETSTACK_CONF_MAC
#define NETSTACK_CONF_MAC     csma_driver
#endif
```

Alternatively, a user can choose NullMAC as follow:

```
#define NETSTACK_CONF_MAC nullmac_driver
```

7.2.2. RDC driver

Radio Duty-Cycle (RDC) layer handles the sleep period of nodes. This layer decides when packets will be transmitted and ensures that nodes are awake when packets are to be received.

The implementation of Contiki's RDC protocols are available in `core/net/mac`. The following RDC drivers are implemented: `contikimac`, `xmac`, `lpp`, `nullrdc` and `sicslowmac`. The implementation and details of the aforementioned RDC drivers are out of the scope of this chapter. The most commonly used is ContikiMAC. NullRDC is a pass-through layer that never switches the radio off.

```
#ifndef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC contikimac_driver
#endif
```

RDC drivers trie to keep the radio off as much as possible, periodically checking the wireless medium for radio activity. When activity is detected, the radio is kept on to check if it has to receive the packet, or it can go back to sleep.

The channel check rate is given in Hz, specifying the number of time the channel is checked per second, and the default channel check rate is 8 Hz. Channel check rates are given in powers of two and typical settings are 2, 4, 8, and 16 Hz.

```
#ifndef NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE
#define NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE 8
#endif
```

A packet must generally be retransmitted or "strobed" until the receiver is on and receives it. This increments the power consumption of the transmitter, and increases the radio traffic, but the power savings when the receiver compensates for this and there is a net overall power saving..

One alternative to optimize the RDC is to enable "phase optimization", which delays strobing until just before the receiver is expected to be awake. This however requires a good time synchronization between the transmitter and the receiver (more details in RDC Phase Optimization[14]). To enable phase optimization change the 0 below to one.

```
#define CONTIKIMAC_CONF_WITH_PHASE_OPTIMIZATION 0
#define WITH_FAST_SLEEP 1
```

7.2.3. Framer driver

The Framer driver is actually a set of functions to frame the data to be transmitted, and to parse the received data. The Framer implementations are located in `core/net/mac`, of which the most noticeable ones are `framer-802154` and `framer-nullmac`.

In the **RE-Mote** platform the following configuration is the default:

```
#ifndef NETSTACK_CONF_FRAMER
#if NETSTACK_CONF_WITH_IPV6
#define NETSTACK_CONF_FRAMER  framer_802154
#else /* NETSTACK_CONF_WITH_IPV6 */
#define NETSTACK_CONF_FRAMER  contikimac_framer
#endif /* NETSTACK_CONF_WITH_IPV6 */
#endif /* NETSTACK_CONF_FRAMER */
```

Meaning that when IPv6 is used, the `framer-802154` is selected, else the `contikimac_framer` is used (default one for the `contikimac_driver`).

[14] https://github.com/contiki-os/contiki/wiki/RDC-Phase-optimization

The `framer-nullmac` framer should be used together with `nullmac_driver` (MAC layer). This simply fills in the 2 fields of `nullmac_hdr`, which are: receiver address and sender address.

The `framer-802154` is implemented in `core/net/mac/framer-802154.c`. The driver frames the data in compliance to the IEEE 802.15.4 (2003) standard. The framer insert and extracts the data to the `packetbuf` structure.

7.3. IPv6 and Routing

One of Contiki's most prominent feature is the support of IP protocols, being one of the first embedded operative systems to provide IPv6 support.

Alternatively Contiki also supports IPv4 and non-IP communication (Rime)[15], however the remainder of this book will focus in IPv6. There is a good set of rime examples available at `examples/rime`. The **RE-Mote** `zoul-demo.c` most basic example at `examples/zolertia/zoul` uses rime as well.

7.3.1. IPv6

The **uIP** is an Open Source TCP/IP stack designed to be used even with tiny 8 and 16 bit microcontrollers. It was initially developed by Adam Dunkels[16] while at the Swedish Institute of Computer Science (SICS)[17], licensed under a BSD style license, and further developed by a wide group of developers.

The implementation details of the uIP/uIPv6 is out of the scope of this section. The remainder of this section explains the basic configurations at the platform and application level.

To enable IPv6 the following has to be defined, either in the application's `Makefile` or in its `project-conf.h` file:

```
#define UIP_CONF_IPV6  1
```

```
#ifndef NBR_TABLE_CONF_MAX_NEIGHBORS
#define NBR_TABLE_CONF_MAX_NEIGHBORS        20
```

[15] https://github.com/alignan/contiki/tree/master/core/net/rime
[16] http://dunkels.com/adam/
[17] https://en.wikipedia.org/wiki/Swedish_Institute_of_Computer_Science

```
#endif
#ifndef UIP_CONF_MAX_ROUTES
#define UIP_CONF_MAX_ROUTES                   20
#endif

/* uIP */
#ifndef UIP_CONF_BUFFER_SIZE
#define UIP_CONF_BUFFER_SIZE                1300
#endif

#define UIP_CONF_IPV6_QUEUE_PKT               0
#define UIP_CONF_IPV6_CHECKS                  1
#define UIP_CONF_IPV6_REASSEMBLY              0
#define UIP_CONF_MAX_LISTENPORTS              8
```

7.3.2. RPL

There are several routing flavors to chose, but ultimately all do the same thing: ensure that packets arrive at the right destination. This is done in different ways depending on factors such as the routing metric (how a route is qualified as better than others), whether the routing is done dynamically or statically, etc.

In Contiki the default routing protocol is RPL. Other protocols such as Ad hoc On-Demand Distance Vector (AODV) are out of the scope of this section.

The specifics of the RPL implementation are out of the scope of this section, we merely describe the common configurations and provide a brief introduction to RPL. For more details, check the RPL implementation at `core/net/rpl`.

What is RPL?

RPL is IPv6 routing protocol for low power and lossy networks designed by the IETF Routing Over Low power and Lossy network (ROLL) group, used as the de facto routing protocol in Contiki. RPL is a proactive distance vector protocol, it starts finding the routes as soon as the RPL network is initialized.

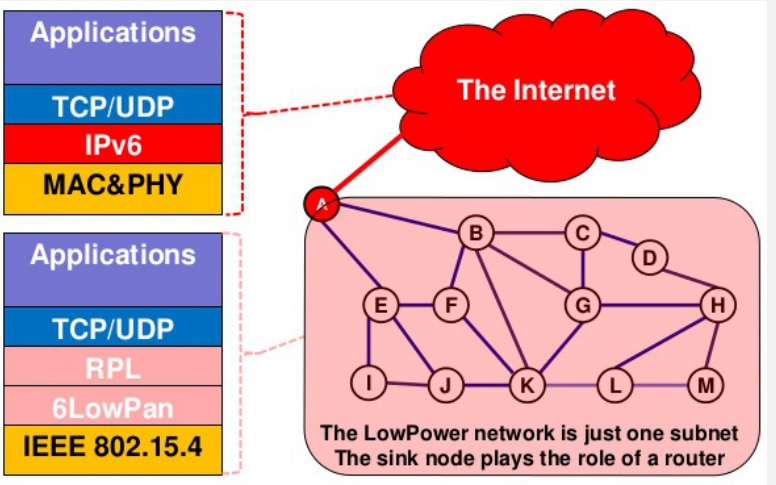

Figure 7.8. RPL in the protocol stack

It supports three traffic patterns:

- Multipoint-to-point (MP2P)
- Point-to-multipoint (P2MP)
- Point-to-point (P2P)

RPL builds a Destination Oriented DAGs (DODAGs) rooted towards one sink (DAG ROOT) identified by a unique identifier DODAGID. The DODAGs are optimized using an Objective Function (OF) metric identified by an Objective Code Point (OCP), which indicates the dynamic constraints and the metrics such as hop count, latency, expected transmission count, parents selection, energy consumption, etc. A rank number is assigned to each node which can be used to determine its relative position and distance to the root in the DODAG.

Within a given network, there may be multiple, logically independent RPL instances. An RPL node may belong to multiple RPL instances, and may act as a router in some and as a leaf in others. A set of multiple DODAGs can be in an RPL INSTANCE and a node can be a member of multiple RPL INSTANCEs, but can belong to at most one DODAG per DAG INSTANCE.

A trickle timer mechanism regulates DODAG Information Object (DIO) message transmissions, which are used to build and maintain upwards routes of the DODAG, advertising its RPL instance, DODAG ID, RANK and DODAG version number.

A node can request DODAG information by sending DODAG Information Solicitation messages (DIS), soliciting DIO messages from its neighborhoods to update its routing information and join an instance.

Nodes have to monitor DIO messages before joining a DODAG, and then join a DODAG by selecting a parent Node from its neighbors using its advertised latency, OF and RANK. Destination Advertisement Object (DAO) messages are used to maintain downward routes by selecting the preferred parent with lower rank and sending a packet to the DAG ROOT through each of the intermediate Nodes.

RPL has two mechanisms to repair the topology of the DODAG, one to avoid looping and allow nodes to join/rejoin, and other called global repair. Global repair is initiated at the DODAG ROOT by incrementing the DODAG Version Number to create a new DODAG Version.

More information about RPL can be found in RFC6550[18].

Routing support is enabled as default in the **Z1 mote** and **RE-Mote** platform. To enable routing the following has to be enabled:

```
#ifndef UIP_CONF_ROUTER
#define UIP_CONF_ROUTER  1
#endif
```

To enable RPL add the following to your application's `Makefile` or its `project-conf.h` file.

```
#define UIP_CONF_IPV6_RPL  1
```

The following is the default configuration done in the **RE-Mote**:

```
/* ND and Routing */
#define UIP_CONF_ND6_SEND_RA        0  ❶
#define UIP_CONF_IP_FORWARD         0  ❷
#define RPL_CONF_STATS              0  ❸
```

❶ Disable sending routing advertisements

[18] https://tools.ietf.org/html/rfc6550

❷ Disable IP forwarding

❸ RPL Configuration statistics are disabled

The `RPL_CONF_OF` parameter configures the RPL objective function. The Minimum Rank with Hysteresis Objective Function (MRHOF) uses ETX as routing metric and it also has stubs for energy metric.

```
#ifndef RPL_CONF_OF
#define RPL_CONF_OF rpl_mrhof
#endif
```

The Expected Transmissions metric (ETX) measure how many tries it takes to receive an acknowledgment (ACK) of a sent packet, keeping a moving average for each neighbor, computing the sum of all ETXs to build the routes.

As default Contiki uses `storing mode` for RPL downward routes. Basically all nodes store in a routing table the addresses of their child nodes.

7.3.3. Set up a sniffer

A packet sniffer is a must-have tool for any wireless network application, it allows to see what you are transmitting over the air, verifying both that the transmissions are taking place, the frames/packets are properly formatted, and that the communication is happening on a given channel.

There are commercial options available, such as the Texas Instruments SmartRF packet Sniffer[19], which can be used with a CC2531 USB dongle[20] to capture packets like the one below.

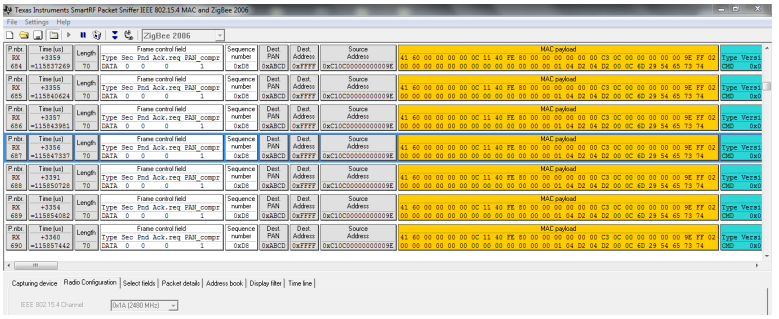

Figure 7.9. Sniffer packet capture

[19] http://www.ti.com/tool/packet-sniffer
[20] http://www.ti.com/tool/CC2531EMK

We will use for this exercise the SenSniff[21] application, paired with a **RE-Mote** and Wireshark (already installed in instant Contiki). This setup will allow us to understand how the wireless communication is done in Contiki.

To program the **RE-Mote** as a packet Sniffer:

```
cd examples/cc2538dk/sniffer
```

Compile and program:

```
make TARGET=zoul sniffer.upload
```

At the moment of writing this section the Z1 sniffer was not officially included in Contiki, however a branch with the implementation is available at: https://github.com/alignan/contiki/tree/z1_sniffer/examples/z1/sniffer

Open a new terminal, and clone the sensniff project in your home folder:

```
cd $HOME
git clone https://github.com/g-oikonomou/sensniff
cd sensniff/host
```

Then launch the sensniff application with the following command:

```
python sensniff.py --non-interactive -d /dev/ttyUSB0 -b 115200
```

Sensniff will read data from the mote over the serial port, dissect the frames and pipe to `/tmp/sensniff` by default, now we need to connect the other extreme of the pipe to wireshark, else you will get the following warning:

```
"Remote end not reading"
```

Which is not worrisome, it only means that the other pipe endpoint is not connected. You can also save the sniffed frames for later opening with wireshark, adding the following argument to the above command `-p name.pcap`, which will save the session output in a `name.pcap` file. Change the naming and location for storing the file accordingly.

[21] https://github.com/g-oikonomou/sensniff

At the moment of writing this tutorial changing channels from the Sensniff application was not implemented but proposed as a feature, check the Sensniff's `README.md` for changes and current status.

Open another terminal and launch wireshark with the following command, which will add the pipe as a capture interface:

```
sudo wireshark -i /tmp/sensniff
```

Select the `/tmp/sensniff` interface from the droplist and click `Start` just above.

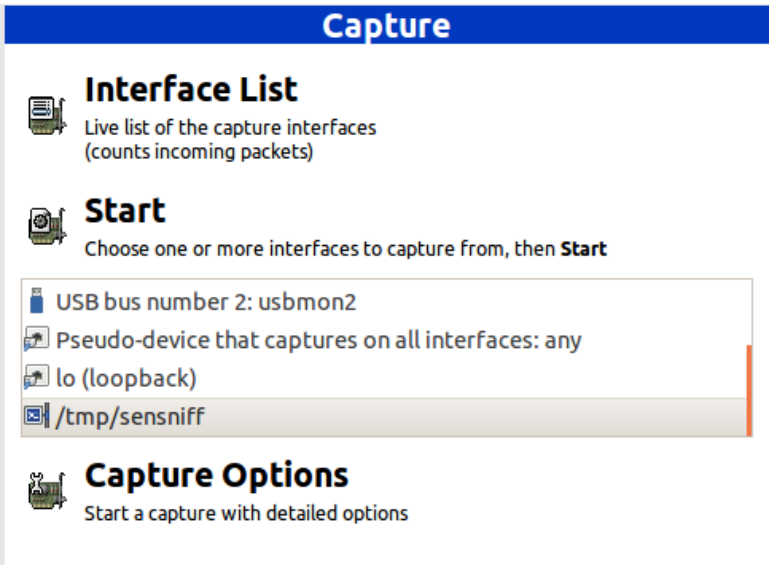

Figure 7.10. Capture options

Make sure that the pipe is configured to capture packets in promiscuous mode, if needed you can increase the buffer size, but 1 MB is normally enough.

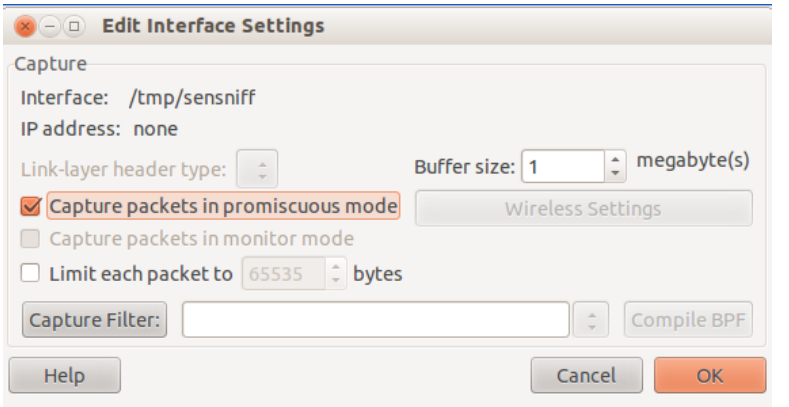

Figure 7.11. Interface settings

Now the captured frames should start to appear on screen.

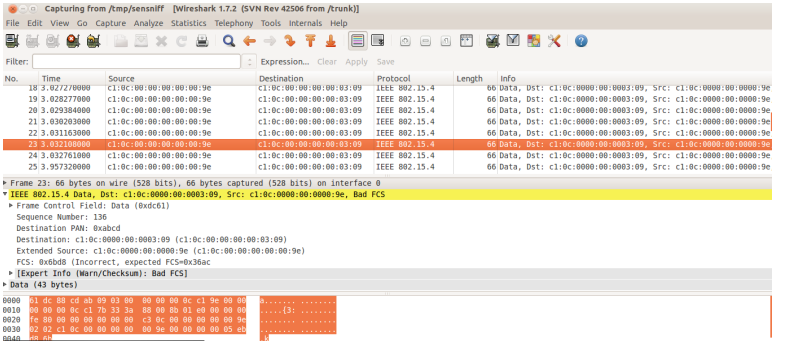

Figure 7.12. Captured frames

You can add specific filters to limit the frames being shown on screen, for this example click at the `Expression` button and a list of available attributes per protocol are listed, scroll down to IEEE 802.15.4 and check the available filters. You can also chain different filter arguments using the `Filter` box, in this case we only wanted to check the frames belonging to the `PAN 0xABCD` and coming from node `c1:0c::0309`, so we used the `wpan.dst_pan` and `wpan.src64` attributes.

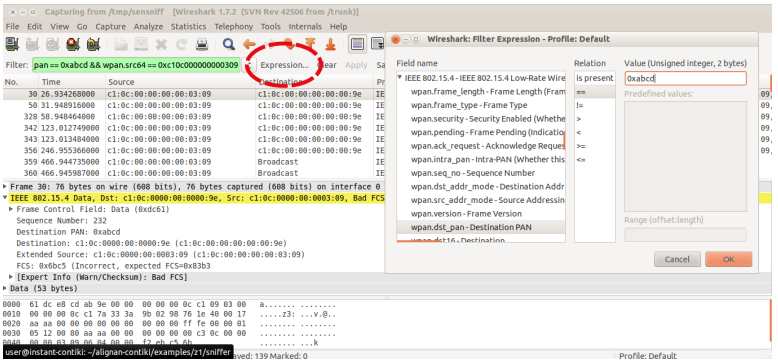

Figure 7.13. Wireshark filters

When closing the Sensniff python application, a session information is provided reporting the following statistics:

```
Frame Stats:
          Non-Frame: 6
          Not Piped: 377
     Dumped to PCAP: 8086
              Piped: 7709
           Captured: 8086
```

Exercise: sniff the traffic! try to filter outgoing and incoming data packets using your own rules.

7.3.4. The Border Router

The border router or edge router is typically a device sitting at the edge of our network, which allow us to talk to outside networks using its built-in network interfaces, such as WiFI, Ethernet, Serial, etc.

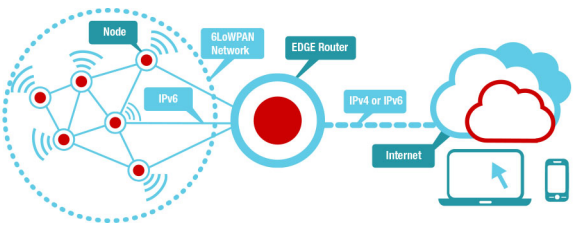

Figure 7.14. The border router

In Contiki the current and most used border router application implements a serial-based interface called SLIP, it allows to connect a given mote to a host using scripts like `tunslip6` in `tools/tunslip6` over the serial port, creating a tunnelled network interface, which can be given an IPv6 prefix to set the network global IPv6 addresses.

The border router application is located at `examples/ipv6/rpl-border-router`, the following code snippets are the most relevant:

```
/* Request prefix until it has been received */
while(!prefix_set) {
   etimer_set(&et, CLOCK_SECOND);
   request_prefix();
   PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
}

dag = rpl_set_root(RPL_DEFAULT_INSTANCE,(uip_ip6addr_t *)dag_id);
if(dag != NULL) {
   rpl_set_prefix(dag, &prefix, 64);
   PRINTF("created a new RPL dag\n");
}
```

The snippet above bootstraps until a valid prefix has been given. Once the prefix has been assigned, the node will set the prefix and convert itself as the root node (DODAG).

Normally is preferable to configure the border router as a non-sleeping device, so the radio receiver is always on. You can configure the border router settings using the `project-conf.h` file.

```
#undef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC       nullrdc_driver
```

By default the border-router applications includes a built-in web server, displaying information about the network, such as the immediate neighbors (1-hop located) and the known routes to nodes in their networks. To enable the web server, the `WITH_WEBSERVER` flag should be enabled, and by default it will add the `httpd-simple.c` application.

Hands on: installing the border router

The following assumes to use a **RE-Mote** platform, but the **Z1 mote** can be used as well.

```
make TARGET=zoul savetarget
```

To compile, flash the mote and connect the border router to your host; run:

```
make border-router.upload && make connect-router
```

By default it will try to connect to a mote at port `/dev/ttyUSB0` using the following serial settings: 115200 baudrate, 8 bits, No parity and 1 bit stop. If you do not state an IPv6 prefix it will use the default `aaaa::1/64`, to specify a different one run the tunslip tool instead using the following:

```
make connect-router PREFIX=2001:abcd:dead:beef::1/64
```

You can also compile and run the `tunslip6` tool directly from the tools location, to compile just type:

```
cd tools
cc tunslip6.c -o tunslip6
```

And to run with specific arguments, i.e. connect to a specific serial port, name your tunnel connection with a specific name, or proxify to a given address and port, use the following:

```
./tunslip -s /dev/ttyUSB0 -t tun0 2001:abcd:dead:beef::1/64
```

Run `tunslip -H` for more information.

6lbr[22] is a deployment-ready 6LoWPAN border router solution based on Contiki, it has support for the **Z1 mote** and in short time will have also for the **RE-Mote** platform (the CC2538dk is already supported, porting is trivial). To take your border router to the next level, this is the tool you have been looking for.

7.4. UDP and TCP basics

Now that we have covered the mote configurations and the MAC and routing layers, let us set up a UDP network.

[22] http://cetic.github.io/6lbr/

What is UDP?

UDP (User Datagram Protocol) is a communications protocol that offers a limited amount of services for messages exchanged among devices in a network that uses the Internet Protocol (IP).

UDP is an alternative to the Transmission Control Protocol (TCP) and, together with IP, is sometimes referred to as UDP/IP. Like the Transmission Control Protocol, UDP uses the Internet Protocol to actually get a data unit (called a datagram) from one computer to another.

Unlike TCP, UDP does not provide message fragmentation and reassembling at the other end, this means that the application must be able to make sure that the entire message has arrived and is in the right order.

Network applications that want to save processing time because they have very small data units to exchange (and therefore very little message reassembling to do) may prefer UDP to TCP

The UDP implementation is Contiki resides in `core/net/ip`. The remainder of the section will focus on describing the UDP available functions.

7.4.1. The UDP API

We need to create a socket for the connection, this is done using the `udp_socket` structure, which has the following elements:

```
struct udp_socket {
  udp_socket_input_callback_t input_callback;
  void *ptr;
  struct process *p;
  struct uip_udp_conn *udp_conn;
};
```

After creating the UDP socket structure, we need to register the UDP socket. This is done with the `udp_socket_register`.

```
/**
```

```
 * \brief      Register a UDP socket
 * \param c    A pointer to the struct udp_socket that should be registered
 * \param ptr  An opaque pointer that will be passed to callbacks
 * \param receive_callback A function pointer to the callback function that will be called
 when data arrives
 * \retval -1  The registration failed
 * \retval 1   The registration succeeded
 */
int udp_socket_register(struct udp_socket *c,
                        void *ptr,
                        udp_socket_input_callback_t receive_callback);
```

As the UDP socket has been created and registered, let us listen on a given port. The `udp_socket_bind` function binds the UDP socket to a local port so it will begin to receive data that arrives on the specified port. A UDP socket will receive data addressed to the specified port number on any IP address of the host. A UDP socket bound to a local port will use this port number as source port for outgoing UDP messages.

```
 * \brief      Bind a UDP socket to a local port
 * \param c    A pointer to the struct udp_socket that should be bound to a local
 port
 * \param local_port The UDP port number, in host byte order, to bind the UDP
 socket to
 * \retval -1  Binding the UDP socket to the local port failed
 * \retval 1   Binding the UDP socket to the local port succeeded
 */
int udp_socket_bind(struct udp_socket *c,
                    uint16_t local_port);
```

The `udp_socket_connect` function connects the UDP socket to a specific remote port and optional remote IP address. When a UDP socket is connected to a remote port and address, it will only receive packets that are sent from that remote port and address. When sending data over a connected UDP socket, the data will be sent to the connected remote address.

A UDP socket can be connected to a remote port, but not to a remote IP address, by providing a `NULL` parameter as the remote_addr parameter. This lets the UDP socket receive data from any IP address on the specified port.

```
/**
 * \brief      Bind a UDP socket to a remote address and port
 * \param c    A pointer to the struct udp_socket that should be connected
 * \param remote_addr The IP address of the remote host, or NULL if the UDP socket should
 only be connected to a specific port
```

```
 * \param remote_port The UDP port number, in host byte order, to which the UDP socket
 should be connected
 * \retval -1  Connecting the UDP socket failed
 * \retval 1   Connecting the UDP socket succeeded
 */
int udp_socket_connect(struct udp_socket *c,
                       uip_ipaddr_t *remote_addr,
                       uint16_t remote_port);
```

To send data over a connected UDP socket it must have been connected to a remote address and port with `udp_socket_connect`.

```
/**
 * \brief      Send data on a UDP socket
 * \param c    A pointer to the struct udp_socket on which the data should be sent
 * \param data A pointer to the data that should be sent
 * \param datalen The length of the data to be sent
 * \return     The number of bytes sent, or -1 if an error occurred
 */
int udp_socket_send(struct udp_socket *c,
                    const void *data, uint16_t datalen);
```

To send data over a UDP socket without being connected we use the function `udp_socket_sendto` instead.

```
/**
 * \brief      Send data on a UDP socket to a specific address and port
 * \param c    A pointer to the struct udp_socket on which the data should be sent
 * \param data A pointer to the data that should be sent
 * \param datalen The length of the data to be sent
 * \param addr The IP address to which the data should be sent
 * \param port The UDP port number, in host byte order, to which the data should be sent
 * \return     The number of bytes sent, or -1 if an error occurred
 */
int udp_socket_sendto(struct udp_socket *c,
                      const void *data, uint16_t datalen,
                      const uip_ipaddr_t *addr, uint16_t port);
```

To close a UDP socket previously registered with `udp_socket_register` the function below is used. All registered UDP sockets must be closed before exiting the process that registered them, or undefined behavior may occur.

```
/**
```

```
 * \brief      Close a UDP socket
 * \param c    A pointer to the struct udp_socket to be closed
 * \retval -1  If closing the UDP socket failed
 * \retval 1   If closing the UDP socket succeeded
 */
int udp_socket_close(struct udp_socket *c);
```

Each UDP socket has a callback function that is registered as part of the call to `udp_socket_register`. The callback function gets called every time a UDP packet is received.

```
/**
 * \brief      A UDP socket callback function
 * \param c    A pointer to the struct udp_socket that received the data
 * \param ptr  An opaque pointer that was specified when the UDP socket was registered with
 udp_socket_register()
 * \param source_addr The IP address from which the datagram was sent
 * \param source_port The UDP port number, in host byte order, from which the datagram was
 sent
 * \param dest_addr The IP address that this datagram was sent to
 * \param dest_port The UDP port number, in host byte order, that the datagram was sent to
 * \param data A pointer to the data contents of the UDP datagram
 * \param datalen The length of the data being pointed to by the data pointer
 */
typedef void (* udp_socket_input_callback_t)(struct udp_socket *c,
                                             void *ptr,
                                             const uip_ipaddr_t *source_addr,
                                             uint16_t source_port,
                                             const uip_ipaddr_t *dest_addr,
                                             uint16_t dest_port,
                                             const uint8_t *data,
                                             uint16_t datalen);
```

Alternatively there is another UDP library called `simple-udp`, which simplifies the UDP API to fewer functions. The library is located in `core/net/ip/simple-udp.c`. For the next example we are going to use the `simple-udp` library, to show how to create a very first basic broadcast example. In a later example we will come back to the full-fledged UDP API.

7.4.2. Hands on: UDP example

The objective of this example is to grasp the concepts shown in the preceding sections. We will create a UDP broadcast application using the `simple-udp`.

There is an existing UDP broadcast example which uses RPL, located at:

```
cd examples/ipv6/simple-udp-rpl
```

Open the `broadcast-example.c` and the `Makefile`. Let's see the contents of the `Makefile`:

```
UIP_CONF_IPV6=1
CFLAGS+= -DUIP_CONF_IPV6_RPL
```

The above adds the IPv6 stack and RPL routing protocol to our application.

The `broadcast-example.c` contains:

```
#include "net/ip/uip.h"
```

This is the main uIP library.

```
/* Network interface and stateless autoconfiguration */
#include "net/ipv6/uip-ds6.h"

/* Use simple-udp library, at core/net/ip/ */
/* The simple-udp module provides a significantly simpler API. */
#include "simple-udp.h"
static struct simple_udp_connection broadcast_connection;
```

This structure allows storing the UDP connection information and mapped callback in which to process any received message. It is initialized in the following call:

```
simple_udp_register(&broadcast_connection, UDP_PORT, NULL, UDP_PORT, receiver);
```

This passes to the simple-udp application the ports from/to handle the broadcasts, and the callback function to handle received broadcasts. We pass the NULL parameter as the destination address to allow packets from any address.

The receiver callback function is shown below:

```
receiver(struct simple_udp_connection *c,
       const uip_ipaddr_t *sender_addr,
       uint16_t sender_port,
       const uip_ipaddr_t *receiver_addr,
       uint16_t receiver_port,
       const uint8_t *data,
```

```
       uint16_t datalen);
```

This application first sets a timer and when the timer expires it sets a randomly generated new timer interval (between 1 and the sending interval) to avoid flooding the network. Then it sets the IP address to the link local all-nodes multicast address as follows:

```
uip_create_linklocal_allnodes_mcast(&addr);
```

And then use the `broadcast_connection` structure (with the values passed at register) and send our data over UDP.

```
simple_udp_sendto(&broadcast_connection, "Test", 4, &addr);
```

To extend the available address information, there is a library which allows to print the IPv6 addresses in a friendlier way, add this to the top of the file:

```
#include "debug.h"
#define DEBUG DEBUG_PRINT
#include "net/ip/uip-debug.h"
```

So we can now print the multicast address, add this before the `simple_udp_sendto(… )` call:

```
PRINT6ADDR(&addr);
printf("\n");
```

Now let's modify our receiver callback and print more information about the incoming message, replace the existing receiver code with the following:

```
static void
receiver(struct simple_udp_connection *c,
       const uip_ipaddr_t *sender_addr,
       uint16_t sender_port,
       const uip_ipaddr_t *receiver_addr,
       uint16_t receiver_port,
       const uint8_t *data,
       uint16_t datalen)
{
  /* Modified to print extended information */
  printf("\nData received from: ");
  PRINT6ADDR(sender_addr);
```

```
  printf("\nAt port %d from port %d with length %d\n",
        receiver_port, sender_port, datalen);
  printf("Data Rx: %s\n", data);
}
```

Before uploading your code, override the default target by writing in the terminal:

```
make TARGET=zoul savetarget
```

Remember you can also use the **Z1 mote** as target.

Now clean any previous compiled code, compile, upload your code and then restart the mote, and print the serial output to screen (all in one command!):

```
make clean && make broadcast-example.upload && make login
```

Upload this code to at least 2 motes and send/receive messages from their neighbors. If you have more than 1 mote connected in your PC, remember to use the `PORT=/dev/ttyUSBx` argument in the upload, reset and login commands!

You will see the following result:

```
Rime started with address 193.12.0.0.0.0.0.158
MAC c1:0c:00:00:00:00:00:9e Ref ID: 3301
Contiki-2.6-1803-g03f57ae started. Node id is set to 158.
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:009e
Starting 'UDP broadcast example process'
Sending broadcast to -> ff02::1

Data received from: fe80::c30c:0:0:309
At port 1234 from port 1234 with length 4
Data Rx: Test
Sending broadcast to -> ff02::1
```

Exercise: Write down the node ID of other motes. This will be useful later. At this point you should also use the Sniffer and capture data over Wireshark.

To change the sending interval you can also modify the values at:

```
#define SEND_INTERVAL  (20 * CLOCK_SECOND)
```

```
#define SEND_TIME      (random_rand() % (SEND_INTERVAL))
```

7.4.3. Hands on: connecting an IPv6 UDP network to our host

In the `udp-client.c` file at `examples/ipv6/rpl-udp`. set the server address to be `aaaa::1` (the host address), replace the options there (Mode 2 is default) and add:

```
uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);
```

To verify that we have set the address correctly let's print the server address, in the `print_local_addresses` function add this to the end:

```
PRINTF("Server address: ");
PRINT6ADDR(&server_ipaddr);
PRINTF("\n");
```

The UDP connection is created in the following block:

```
/* new connection with remote host */
client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
if(client_conn == NULL) {
  PRINTF("No UDP connection available, exiting the process!\n");
  PROCESS_EXIT();
}
udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));
```

And upon receiving a message the `tcpip_handler` is called to process the incoming data:

```
static void
tcpip_handler(void)
{
  char *str;

  if(uip_newdata()) {
    str = uip_appdata;
    str[uip_datalen()] = '\0';
    printf("DATA recv '%s'\n", str);
  }
}
```

Compile and program the mote:

```
cd examples/ipv6/rpl-udp
make TARGET=z1 savetarget
make udp-client.upload && make z1-reset && make login

Rime started with address 193.12.0.0.0.0.0.158
MAC c1:0c:00:00:00:00:00:9e Ref ID: 158
Contiki-2.6-2071-gc169b3e started. Node id is set to 158.
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:009e
Starting 'UDP client process'
UDP client process started
Client IPv6 addresses: aaaa::c30c:0:0:9e
fe80::c30c:0:0:9e
Server address: aaaa::1
Created a connection with the server :: local/remote port 8765/5678
DATA send to 1 'Hello 1'
DATA send to 1 'Hello 2'
DATA send to 1 'Hello 3'
DATA send to 1 'Hello 4'
```

Remember that you can also compile for the **RE-Mote** platform.

UDP Server

The UDP server is a python script that echoes any incoming data back to the client, useful to test the bi-directional communication between the host and the network.

The `UDP6.py` script can be executed as a single-shot UDP client or as a UDP Server bound to a specific address and port, for this example we are to bind to address `aaaa::1` and port `5678`.

The script content is below:

```
#! /usr/bin/env python

import sys
from socket import *
from socket import error

PORT      = 5678
BUFSIZE   = 1024

#---------------------------------------------------------------#
# Start a client or server application for testing
#---------------------------------------------------------------#
```

```
def main():
        if len(sys.argv) < 2:
                usage()
        if sys.argv[1] == '-s':
                server()
        elif sys.argv[1] == '-c':
                client()
        else:
                usage()

#--------------------------------------------------------------#
# Prints the instructions
#--------------------------------------------------------------#
def usage():
        sys.stdout = sys.stderr
        print 'Usage: udpecho -s [port]            (server)'
        print 'or:    udpecho -c host [port] <file (client)'
        sys.exit(2)

#--------------------------------------------------------------#
# Creates a server, echoes the message back to the client
#--------------------------------------------------------------#
def server():
    if len(sys.argv) > 2:
        port = eval(sys.argv[2])
    else:
        port = PORT

    try:
        s = socket(AF_INET6, SOCK_DGRAM)
        s.bind(('aaaa::1', port))
    except Exception:
        print "ERROR: Server Port Binding Failed"
        return
    print 'udp echo server ready: %s' % port
    while 1:
        data, addr = s.recvfrom(BUFSIZE)
        print 'server received', `data`, 'from', `addr`
        s.sendto(data, addr)

#--------------------------------------------------------------#
# Creates a client that sends an UDP message to a server
#--------------------------------------------------------------#
def client():
    if len(sys.argv) < 3:
        usage()
    host = sys.argv[2]
    if len(sys.argv) > 3:
```

```
        port = eval(sys.argv[3])
    else:
        port = PORT
    addr = host, port
    s = socket(AF_INET6, SOCK_DGRAM)
    s.bind(('', 0))
        print 'udp echo client ready, reading stdin'
    try:
        s.sendto("hello", addr)
    except error as msg:
        print msg
    data, fromaddr = s.recvfrom(BUFSIZE)
    print 'client received', `data`, 'from', `fromaddr`

#------------------------------------------------------------#
# MAIN APP
#------------------------------------------------------------#
main()
```

To execute the `UDP6.py` script just run:

```
python UDP6.py -s 5678
```

This is the expected output when running and receiving a UDP packet:

```
udp echo server ready: 5678
server received 'Hello 198 from the client' from ('aaaa::c30c:0:0:9e', 8765, 0, 0)
```

The Server then echoes back the message to the UDP client to the given `8765` port, this is the expected output from the mote:

```
DATA send to 1 'Hello 198'
DATA recv 'Hello 198 from the client'
```

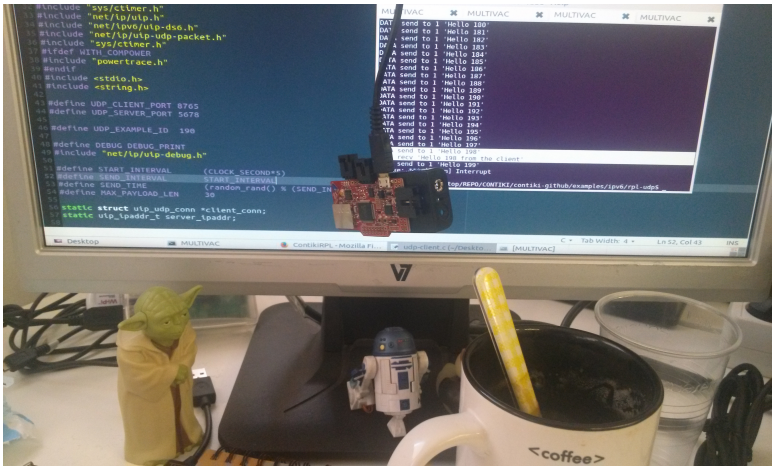

Figure 7.15. Z1 mote talking to the PC host

7.4.4. What is TCP?

What is TCP?

The Transmission Control Protocol (TCP) is a core protocol of the Internet Protocol (IP).

TCP is a reliable stream delivery service that ensures that all bytes received will be in the correct order. It uses a technique known as positive acknowledgment with retransmission to guarantee reliability of packet transfer. TCP handles the received fragments and reorders the data.

Applications that do not require reliable data stream service may use the User Datagram Protocol (UDP), which provides a connectionless datagram service that emphasizes reduced latency over reliability.

TCP is commonly used by HTTP, FTP, email and any connection-oriented service.

The TCP implementation is Contiki resides in `core/net/ip`. The remainder of the section will focus on describing the TCP available functions.

The TCP API

We need to create a socket for the connection, this is done using the `tcp_socket` structure, which has the following elements:

```
struct tcp_socket {
  struct tcp_socket *next;

  tcp_socket_data_callback_t input_callback;
  tcp_socket_event_callback_t event_callback;
  void *ptr;

  struct process *p;

  uint8_t *input_data_ptr;
  uint8_t *output_data_ptr;

  uint16_t input_data_maxlen;
  uint16_t input_data_len;
  uint16_t output_data_maxlen;
  uint16_t output_data_len;
  uint16_t output_data_send_nxt;
  uint16_t output_senddata_len;
  uint16_t output_data_max_seg;

  uint8_t flags;
  uint16_t listen_port;
  struct uip_conn *c;
};
```

Socket status:

```
enum {
  TCP_SOCKET_FLAGS_NONE      = 0x00,
  TCP_SOCKET_FLAGS_LISTENING = 0x01,
  TCP_SOCKET_FLAGS_CLOSING   = 0x02,
};
```

After creating the TCP socket structure, we need to register the TCP socket. This is done with the `tcp_socket_register`, which takes as arguments the TCP socket, and input/output buffers to use for sending and receiving data. Be sure to dimension these buffers according to the expected data to be sent and received.

```
/**
 * \brief      Register a TCP socket
 * \param s    A pointer to a TCP socket
 * \param ptr  A user-defined pointer that will be sent to callbacks for this socket
 * \param input_databuf A pointer to a memory area this socket will use for input data
 * \param input_databuf_len The size of the input data buffer
```

```
 * \param output_databuf A pointer to a memory area this socket will use for outgoing data
 * \param output_databuf_len The size of the output data buffer
 * \param data_callback A pointer to the data callback function for this socket
 * \param event_callback A pointer to the event callback function for this socket
 * \retval -1  If an error occurs
 * \retval 1   If the operation succeeds.
 */
int tcp_socket_register(struct tcp_socket *s, void *ptr,
                        uint8_t *input_databuf, int input_databuf_len,
                        uint8_t *output_databuf, int output_databuf_len,
                        tcp_socket_data_callback_t data_callback,
                        tcp_socket_event_callback_t event_callback);
```

As the TCP socket has been created and registered, let us listen on a given port. When a remote host connects to the port, the event callback will be called with the `TCP_SOCKET_CONNECTED` event. When the connection closes, the socket will go back to listening.

```
/**
 * \brief      Start listening on a specific port
 * \param s    A pointer to a TCP socket that must have been previously registered with
 tcp_socket_register()
 * \param port The TCP port number, in host byte order, of the remote host
 * \retval -1  If an error occurs
 * \retval 1   If the operation succeeds.
 */
int tcp_socket_listen(struct tcp_socket *s,
                      uint16_t port);
```

To stop listening on a given TCP port, just call the following function:

```
/**
 * \brief      Stop listening for new connections
 * \param s    A pointer to a TCP socket that must have been previously registered with
 tcp_socket_register()
 * \retval -1  If an error occurs
 * \retval 1   If the operation succeeds.
 */
int tcp_socket_unlisten(struct tcp_socket *s);
```

We can connect the TCP socket to a remote host. When the socket has connected, the event callback will get called with the `TCP_SOCKET_CONNECTED` event. If the remote host does not accept the connection, the `TCP_SOCKET_ABORTED` will be sent to the callback. If the

connection times out before conecting to the remote host, the `TCP_SOCKET_TIMEDOUT` event is sent to the callback.

```
/**
 * \brief      Connect a TCP socket to a remote host
 * \param s    A pointer to a TCP socket that must have been previously registered with
 tcp_socket_register()
 * \param ipaddr The IP address of the remote host
 * \param port The TCP port number, in host byte order, of the remote host
 * \retval -1  If an error occurs
 * \retval 1   If the operation succeeds.
 */
int tcp_socket_connect(struct tcp_socket *s,
                       const uip_ipaddr_t *ipaddr,
                       uint16_t port);
```

As we are using an output buffer to send data over the TCP socket, a good practice is to query the TCP socket and check the number of bytes available.

```
/**
 * \brief      The maximum amount of data that could currently be sent
 * \param s    A pointer to a TCP socket
 * \return     The number of bytes available in the output buffer
 */
int tcp_socket_max_sendlen(struct tcp_socket *s);
```

To send data over a connected TCP socket the data is placed in the output buffer. When the data has been acknowledged by the remote host, the event callback is sent with the `TCP_SOCKET_DATA_SENT` event.

```
/**
 * \brief      Send data on a connected TCP socket
 * \param s    A pointer to a TCP socket that must have been previously registered with
 tcp_socket_register()
 * \param dataptr A pointer to the data to be sent
 * \param datalen The length of the data to be sent
 * \retval -1  If an error occurs
 * \return     The number of bytes that were successfully sent
 */
int tcp_socket_send(struct tcp_socket *s,
                    const uint8_t *dataptr,
                    int datalen);
```

Alternatively we can send a string over a TCP socket as follows:

```
/**
 * \brief      Send a string on a connected TCP socket
 * \param s    A pointer to a TCP socket that must have been previously registered with
 tcp_socket_register()
 * \param strptr A pointer to the string to be sent
 * \retval -1  If an error occurs
 * \return     The number of bytes that were successfully sent
 */
int tcp_socket_send_str(struct tcp_socket *s,
                        const char *strptr);
```

To close a connected TCP socket the function below is used. The event callback is called with the `TCP_SOCKET_CLOSED` event.

```
/**
 * \brief      Close a connected TCP socket
 * \param s    A pointer to a TCP socket that must have been previously registered with
 tcp_socket_register()
 * \retval -1  If an error occurs
 * \retval 1   If the operation succeeds.
 */
int tcp_socket_close(struct tcp_socket *s);
```

And to unregister a TCP socket the `tpc_socket_unregister` function is used. This function can also be used to reset a connected TCP socket.

```
/**
 * \brief      Unregister a registered socket
 * \param s    A pointer to a TCP socket that must have been previously registered with
 tcp_socket_register()
 * \retval -1  If an error occurs
 * \retval 1   If the operation succeeds.
 *
 *             This function unregisters a previously registered
 *             socket. This must be done if the process will be
 *             unloaded from memory. If the TCP socket is connected,
 *             the connection will be reset.
 *
 */
int tcp_socket_unregister(struct tcp_socket *s);
```

The TCP socket event callback function gets called whenever there is an event on a socket, such as the socket getting connected or closed.

```
/**
 * \brief      TCP event callback function
 * \param s    A pointer to a TCP socket
 * \param ptr  A user-defined pointer
 * \param event The event number
 */
typedef void (* tcp_socket_event_callback_t)(struct tcp_socket *s,
                                             void *ptr,
                                             tcp_socket_event_t event);
```

The TCP data callback function has to be added to the application, it will get called whenever there is new data on the socket:

```
/**
 * \brief      TCP data callback function
 * \param s    A pointer to a TCP socket
 * \param ptr  A user-defined pointer
 * \param input_data_ptr A pointer to the incoming data
 * \param input_data_len The length of the incoming data
 * \return     The function should return the number of bytes to leave in the input buffer
 */
typedef int (* tcp_socket_data_callback_t)(struct tcp_socket *s,
                                           void *ptr,
                                           const uint8_t *input_data_ptr,
                                           int input_data_len);
```

Hands on: TCP example

Now let us put to practice the TCP API described before and browse a TCP application. The `tpc-socket` example is located in `examples/tcp-socket`. The TCP server simply echoes back the request done on port 80.

The `Makefile` enables as default the IPv4 stack, change it to IPv6:

```
UIP_CONF_IPV6=1
CFLAGS+= -DUIP_CONF_IPV6_RPL
```

Then let us open the `tcp-server.c` example and browse the implementation.

The port 80 will be used for the TCP server to receive remote connections. As shown earlier we need to create a `tcp_socket` structure, and use two separate input/output buffers to send and receive data.

```
#define SERVER_PORT 80

static struct tcp_socket socket;

#define INPUTBUFSIZE 400
static uint8_t inputbuf[INPUTBUFSIZE];

#define OUTPUTBUFSIZE 400
static uint8_t outputbuf[OUTPUTBUFSIZE];
```

These two variables will be used to count the number of bytes received, and the bytes to be sent.

```
static uint8_t get_received;
static int bytes_to_send;
```

As commented earlier, we need to include a `tcp_socket_event_callback_t` to handle events.

```
static void
event(struct tcp_socket *s, void *ptr,
      tcp_socket_event_t ev)
{
  printf("event %d\n", ev);
}
```

We register the TCP socket and pass as a pointer the `tcp_socket` structure, the data buffers, and our callback handlers. Next we start listening for connections on port 80.

```
tcp_socket_register(&socket, NULL,
                inputbuf, sizeof(inputbuf),
                outputbuf, sizeof(outputbuf),
                input, event);

tcp_socket_listen(&socket, SERVER_PORT);
```

The `input` callback handler receives the data, prints the string and its length, then if the received string is a complete request we save the number of bytes received into

`bytes_to_send` (the `atoi` function converts string numbers into integers). If the received string is not complete, we return the number of bytes received to the driver to keep the data in the input buffer.

```
static int
input(struct tcp_socket *s, void *ptr,
      const uint8_t *inputptr, int inputdatalen)
{
  printf("input %d bytes '%s'\n", inputdatalen, inputptr);
  if(!get_received) {
    /* See if we have a full GET request in the buffer. */
    if(strncmp((char *)inputptr, "GET /", 5) == 0 &&
       atoi((char *)&inputptr[5]) != 0) {
      bytes_to_send = atoi((char *)&inputptr[5]);
      printf("bytes_to_send %d\n", bytes_to_send);
      return 0;
    }
    printf("inputptr '%.*s'\n", inputdatalen, inputptr);
    /* Return the number of data bytes we received, to keep them all
       in the buffer. */
    return inputdatalen;
  } else {
    /* Discard everything */
    return 0; /* all data consumed */
  }
}
```

The application will wait for an event to happen, in this case the incoming connection from above. After the event is handled, the code inside the `while()` loop and after the `PROCESS_PAUSE()` will be executed.

```
while(1) {
  PROCESS_PAUSE();
```

If we have previously received a complete request, we echo it back over the TCP socket. We use the `tcp_socket_send_str` function to send the header of the response as a string. The remainder of the data is sent until the `bytes_to_send` counter is empty.

```
    if(bytes_to_send > 0) {
      /* Send header */
      printf("sending header\n");
      tcp_socket_send_str(&socket, "HTTP/1.0 200 ok\r\nServer: Contiki tcp-socket
 example\r\n\r\n");
```

```
      /* Send data */
      printf("sending data\n");
      while(bytes_to_send > 0) {
        PROCESS_PAUSE();
        int len, tosend;
        tosend = MIN(bytes_to_send, sizeof(outputbuf));
        len = tcp_socket_send(&socket, (uint8_t *)"", tosend);
        bytes_to_send -= len;
      }
      tcp_socket_close(&socket);
    }
  }
  PROCESS_END();
}
```

When all the data is echoed back, the TCP socket is closed.

Chapter 8. CoAP, MQTT and HTTP

In the previous section we covered some of the wireless basic, we should now have a good idea about working with Contiki. This section introduces two widely used protocols for the IoT: CoAP and MQTT. We will explain the basics and wrap up with ready to use examples.

8.1. CoAP example

The CoAP implementation in Contiki is based in Erbium (Er), a low-power REST Engine for Contiki. The REST Engine includes a comprehensive embedded CoAP implementation, which became the official one in Contiki.

More information about its implementation and author is available in the Erbium site[1].

What are REST and CoAP?

The **Representational State Transfer (REST)** relies on a stateless, client-server, cacheable communications protocol - and in virtually all cases, the HTTP protocol can be used.

The key abstraction of a RESTful web service is the resource, not a service. Sensors, actuators and control systems in general can be elegantly represented as resources and their service exposed through a RESTful web service.

RESTful applications use HTTP-like requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.

Despite being simple, REST is fully-featured; there's basically nothing you can do in Web Services that can't be done with a RESTful architecture. REST is not a standard.

The **Constrained Application Protocol (CoAP)** is a software protocol intended to be used in very simple electronics devices that allows them to communicate interactively over the Internet. It is particularly targeted for small low power sensors, switches, valves and similar components that need to be controlled or supervised remotely, through standard Internet networks. CoAP is an application layer protocol that is

[1] http://people.inf.ethz.ch/mkovatsc/erbium.php

intended for use in resource-constrained internet devices, such as WSN nodes. CoAP is designed to easily translate to HTTP for simplified integration with the web, while also meeting specialized requirements such as multicast support, very low overhead, and simplicity.

CoAP can run on most devices that support UDP. CoAP makes use of two message types, requests and responses, using a simple binary base header format. The base header may be followed by options in an optimized Type-Length-Value format. CoAP is by default bound to UDP and optionally to DTLS, providing a high level of communications security.

Any bytes after the headers in the packet are considered the message body (if any is present). The length of the message body is implied by the datagram length. When bound to UDP the entire message MUST fit within a single datagram. When used with 6LoWPAN as defined in RFC 4944, messages should fit into a single IEEE 802.15.4 frame to minimize fragmentation.

8.1.1. CoAP API

The CoAP implementation in Contiki is located in `apps/er-coap`. The Erbium REST engine is implemented in `apps/rest-engine`.

The coap engine (currently the CoAP-18 implementation) is implemented in `er-coap-engine.c`. The engine interface is provided by a structure as follows:

```
const struct rest_implementation coap_rest_implementation = {
  coap_init_engine,
  coap_set_service_callback,
  coap_get_header_uri_path,
(...)
}
```

It is possible then to invoke the CoAP engine as follows:

```
REST.get_query_variable();
```

Web services are viewed as resources, and can be uniquely identified by their URLs. The basic REST design uses the HTTP or COAP protocol methods for typical `CRUD` operations (create, read, update, delete):

- POST: Create a resource
- GET: Retrieve a resource
- PUT: Update a resource
- DELETE: Delete a resource

There are various resources that are available at the server. Each resource at the server has a handler function which the REST layer calls to serve the request by the client. The REST server sends the response back to the client with the contents of the resource requested.

The following macros are available in `apps/rest-engine`, recommended when creating a new CoAP resource.

A **normal resource** is defined by a static Uri-Path that is associated with a resource handler function. This is the basis for all other resource types.

```
#define RESOURCE(name, attributes, get_handler, post_handler, put_handler,
 delete_handler) \
  resource_t name = { NULL, NULL, NO_FLAGS, attributes, get_handler, post_handler,
 put_handler, delete_handler, { NULL } }
```

A **parent resource** manages several sub-resources by evaluating the Uri-Path, which may be longer than the parent resource.

```
#define PARENT_RESOURCE(name, attributes, get_handler, post_handler, put_handler,
 delete_handler) \
  resource_t name = { NULL, NULL, HAS_SUB_RESOURCES, attributes, get_handler,
 post_handler, put_handler, delete_handler, { NULL } }
```

If the server is not able to respond immediately to a `CON` request , it simply responds with an Empty ACK message so that the client can stop re-transmitting the request. After a while, when the server is ready with the response, it sends the response as a `CON` message. The following macro allows to create a CoAP resource with **separate response**:

```
#define SEPARATE_RESOURCE(name, attributes, get_handler, post_handler, put_handler,
 delete_handler, resume_handler) \
  resource_t name = { NULL, NULL, IS_SEPARATE, attributes, get_handler,
 post_handler, put_handler, delete_handler, { .resume = resume_handler } }
```

An **event resource** is similar to an periodic resource, only that the second handler is called by an irregular event such as a button.

```
#define EVENT_RESOURCE(name, attributes, get_handler, post_handler, put_handler,
 delete_handler, event_handler) \
  resource_t name = { NULL, NULL, IS_OBSERVABLE, attributes, get_handler,
 post_handler, put_handler, delete_handler, { .trigger = event_handler } }
```

If we need to declare a **periodic resource**, for example to poll a sensor and publish a changed value to subscribed clients, then we should use:

```
#define PERIODIC_RESOURCE(name, attributes, get_handler, post_handler, put_handler,
 delete_handler, period, periodic_handler) \
  periodic_resource_t periodic_##name; \
  resource_t name = { NULL, NULL, IS_OBSERVABLE | IS_PERIODIC, attributes,
 get_handler, post_handler, put_handler, delete_handler, { .periodic =
 &periodic_##name } }; \
  periodic_resource_t periodic_##name = { NULL, &name, period, { { 0 } },
 periodic_handler };
```

Notice the `PERIODIC_RESOURCE` and `EVENT_RESOURCE` can be observable, meaning a client can be notified for any change in a given resource.

Once we declare and implement the resources (we will get to that in the Hands on section), we need to initializes REST framework and start the HTTP or CoAP process. This is done using:

```
void rest_init_engine(void);
```

Then for each declared resource we need want to be accessible, we need to call:

```
void rest_activate_resource(resource_t *resource, char *path);
```

So assume we have created a `hello-world` resource in `res-hello.c`, and declared as follows:

```
RESOURCE(res_hello,
         "title=\"Hello world: ?len=0..\";rt=\"Text\"",
         res_get_handler,
         NULL,
         NULL,
         NULL);
```

To enable the resource we would do:

```
rest_activate_resource(&res_hello, "test/hello");
```

Which means the resource would be available at `test/hello` uri-Path.

The function above stores the resources into a list. To list the available resources, the `rest_get_resources` function is used. This will return a list with the resources with the following:

```
rest_get_resources();
```

Remember the mandatory CoAP port is `5683`.

Now let us put the above to work in the following hands on example.

8.1.2. Hands on: CoAP server and Copper

First get the Copper (Cu) CoAP user-agent from:

https://addons.mozilla.org/en-US/firefox/addon/copper-270430/

Copper is a generic browser for the Internet of Things based on the Constrained Application Protocol (CoAP), a user-friendly management tool for networked embedded devices. As it is integrated into web browsers, it allows an intuitive interaction and with the presentation layer making easier to debug existing CoAP devices.

More information available at:

http://people.inf.ethz.ch/mkovatsc/copper.php

For this practice we will use 2 motes: a Border Router and a CoAP server.

If you are using the **Z1 motes**, Ensure that the 2 motes you will be using to test this (border router, client, server) have flashed a Node ID to generate the MAC/IPv6 addresses as done in previous sessions, be sure to write down the addresses! Another thing, if you get an error like the following, go to `platform/z1/contiki-conf.h` and change `UIP_CONF_BUFFER_SIZE` to 240:

```
#error "UIP_CONF_BUFFER_SIZE too small for REST_MAX_CHUNK_SIZE"
make: *** [obj_z1/er-coap-07-engine.o] Error 1
```

In the `Makefile` we can notice two things: the `resources` folder is included as a project directory, and all the resources files are added in the compilation.

```
REST_RESOURCES_DIR = ./resources
REST_RESOURCES_FILES = $(notdir $(shell find $(REST_RESOURCES_DIR) -name '*.c' ! -
name 'res-plugtest*'))
PROJECTDIRS += $(REST_RESOURCES_DIR)
PROJECT_SOURCEFILES += $(REST_RESOURCES_FILES)
```

The second thing is we are including the `er-coap` and `rest-engine` applications.

```
# REST Engine shall use Erbium CoAP implementation
APPS += er-coap
APPS += rest-engine
```

Remove the following as we want to avoid collisions as much as possible:

```
#undef NETSTACK_CONF_MAC
#define NETSTACK_CONF_MAC     nullmac_driver
```

Next let us check the `project-conf.h` relevant configuration. First we make sure TCP is disabled, as CoAP is based on UDP.

```
/* Disabling TCP on CoAP nodes. */
#undef UIP_CONF_TCP
#define UIP_CONF_TCP                    0
```

The `REST_MAX_CHUNK_SIZE` is the maximum buffer size that is provided for resource responses. Larger data should be handled by the resource and be sent in CoAP blocks. The `COAP_MAX_OPEN_TRANSACTIONS` is the number of maximum open transactions the node is able to handle.

```
/* Increase rpl-border-router IP-buffer when using more than 64. */
#undef REST_MAX_CHUNK_SIZE
#define REST_MAX_CHUNK_SIZE             48

/* Multiplies with chunk size, be aware of memory constraints. */
#undef COAP_MAX_OPEN_TRANSACTIONS
#define COAP_MAX_OPEN_TRANSACTIONS      4
```

```
/* Filtering .well-known/core per query can be disabled to save space. */
#undef COAP_LINK_FORMAT_FILTERING
#define COAP_LINK_FORMAT_FILTERING     0
#undef COAP_PROXY_OPTION_PROCESSING
#define COAP_PROXY_OPTION_PROCESSING   0

/* Enable client-side support for COAP observe */
#define COAP_OBSERVE_CLIENT 1
```

CoAP Server:

Let us walkthrough the `er-example-server.c` example and understand its implementation. The first noticeable thing is that the folder example has a folder called resources, this is fairly simple: the resources are implemented in a different file, this makes easier to debug and maintain the example.

The resources to be included in the CoAP server are defined in the following declaration:

```
extern resource_t
  res_hello,
  res_mirror,
  res_chunks,
  res_separate,
  res_push,
  res_event,
  res_sub,
  res_b1_sep_b2;
#if PLATFORM_HAS_LEDS
extern resource_t res_leds, res_toggle;
#endif
#if PLATFORM_HAS_BATTERY
#include "dev/battery-sensor.h"
extern resource_t res_battery;
#endif
#if PLATFORM_HAS_RADIO
#include "dev/radio-sensor.h"
extern resource_t res_radio;
#endif
```

The resources wrapped inside the `PLATFORM_HAS_X` defines are dependant on the target platform, and will get pulled-in if the platform has those enabled.

Then the REST engine is initialized by calling the `rest_init_engine()`, and the enabled resources are binded:

```
/* Initialize the REST engine. */
rest_init_engine();

/*
 * Bind the resources to their Uri-Path.
 * WARNING: Activating twice only means alternate path, not two instances!
 * All static variables are the same for each URI path.
 */
rest_activate_resource(&res_hello, "test/hello");
rest_activate_resource(&res_push, "test/push");
rest_activate_resource(&res_event, "sensors/button"); */
#if PLATFORM_HAS_LEDS
  rest_activate_resource(&res_toggle, "actuators/toggle");
#endif
(...)
```

Now let us take a look at the `res-hello.c` resource, which implements a "hello world" resource for testing.

As shown before resources are defined using the `RESOURCE` macro, for this particular implementation we specify the resource name as `res_hello`, the link-formatted atributtes and the `GET` callback handler. The `POST`, `PUT`, and `DELETE` methods are not supported by this resource, so a `NULL` parameter is used as argument.

```
RESOURCE(res_hello,
         "title=\"Hello world: ?len=0..\";rt=\"Text\"",
         res_get_handler,
         NULL,
         NULL,
         NULL);
```

The `res_get_handler` is the event callback for `GET` requests, its implementation

```
static void
res_get_handler(void *request, void *response, uint8_t *buffer, uint16_t
 preferred_size, int32_t *offset)
{
  const char *len = NULL;
  /* Some data that has the length up to REST_MAX_CHUNK_SIZE. For more, see the
 chunk resource. */
  char const *const message = "Hello World!
 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxy";
  int length = 12;
```

```
  /* The query string can be retrieved by rest_get_query() or parsed for its key-
value pairs. */
  if(REST.get_query_variable(request, "len", &len)) {                    ❷
    length = atoi(len);
    if(length < 0) {                                                     ❸
      length = 0;
    }
    if(length > REST_MAX_CHUNK_SIZE) {                                   ❹
      length = REST_MAX_CHUNK_SIZE;
    }
    memcpy(buffer, message, length);
  } else {
    memcpy(buffer, message, length);                                     ❺

    /* text/plain is the default, hence this option could be omitted. */
  } REST.set_header_content_type(response, REST.type.TEXT_PLAIN);        ❻

  REST.set_header_etag(response, (uint8_t *)&length, 1);                 ❼
  REST.set_response_payload(response, buffer, length);                   ❽
}
```

❶ The default lenght of the reply, in this case from the complete string, only `Hello world!` will be sent

❷ If the `len` option is specified, then a number of `len` bytes of the `message` string will be sent

❸ If the value is a negative one, send end an empty string

❹ If `len` is higher than the maximum allowed, then we only send the maximum lenght value

❺ Copy the default

❻ Set the response content type as `Content-Type:text/plain`

❼ Attach the header to the response, set the payload lenght field

❽ Attach the payload to the response

> Be sure the settings are consistent with the ones of the border router,

In the `project-conf.h` file add the following for this test purpose:

```
#undef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC     nullrdc_driver
```

Then compile and upload:

```
cd examples/er-rest-example/
make TARGET=zoul savetarget
make er-example-server.upload && make login
```

Write down the IPv6 server address, disconnect the mote and connect another one to be used as client.

Disconnect the mote, connect another one to be used as border-router:

Border-Router:

```
cd ../ipv6/rpl-border-router/
make TARGET=z1 savetarget
make border-router.upload && make connect-router
```

Or use also:

```
make TARGET=zoul savetarget && make border-router.upload && make connect-router
```

Don't close this window! leave the mote connected, now you will be watching something like this:

```
SLIP started on ``/dev/ttyUSB0''
opened tun device ``/dev/tun0''
ifconfig tun0 inet `hostname` up
ifconfig tun0 add aaaa::1/64
ifconfig tun0 add fe80::0:0:0:1/64
ifconfig tun0

tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
     inet addr:127.0.1.1  P-t-P:127.0.1.1  Mask:255.255.255.255
     inet6 addr: fe80::1/64 Scope:Link
     inet6 addr: aaaa::1/64 Scope:Global
     UP POINTOPOINT RUNNING NOARP 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 txqueuelen:500
     RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

Rime started with address 193.12.0.0.0.0.3.229
MAC c1:0c:00:00:00:00:03:e5 Contiki-2.5-release-681-gc5e9d68 started. Node id
 is set to 997.
```

```
CSMA nullrdc, channel check rate 128 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:03e5
Starting 'Border router process' 'Web server'
Address:aaaa::1 => aaaa:0000:0000:0000
Got configuration message of type P
Setting prefix aaaa::
Server IPv6 addresses:
aaaa::c30c:0:0:3e5
fe80::c30c:0:0:3e5
```

Let's ping the border-router:

```
ping6 aaaa:0000:0000:0000:c30c:0000:0000:03e5
PING aaaa:0000:0000:0000:c30c:0000:0000:03e5(aaaa::c30c:0:0:3e5) 56 data bytes
64 bytes from aaaa::c30c:0:0:3e5: icmp_seq=1 ttl=64 time=21.0 ms
64 bytes from aaaa::c30c:0:0:3e5: icmp_seq=2 ttl=64 time=19.8 ms
64 bytes from aaaa::c30c:0:0:3e5: icmp_seq=3 ttl=64 time=22.2 ms
64 bytes from aaaa::c30c:0:0:3e5: icmp_seq=4 ttl=64 time=20.7 ms
```

Now connect the server mote, ping it too:

```
ping6 aaaa:0000:0000:0000:c30c:0000:0000:0001
PING aaaa:0000:0000:0000:c30c:0000:0000:0001(aaaa::c30c:0:0:1) 56 data bytes
64 bytes from aaaa::c30c:0:0:1: icmp_seq=1 ttl=63 time=40.3 ms
64 bytes from aaaa::c30c:0:0:1: icmp_seq=2 ttl=63 time=34.2 ms
64 bytes from aaaa::c30c:0:0:1: icmp_seq=3 ttl=63 time=35.7 ms
```

Now we can start discovering the Server resources, Open Firefox and type the server address:

```
coap://[aaaa::c30c:0000:0000:0001]:5683/
```

And began by discovering the available resources, Press `DISCOVER` and the page will be populated in the left side:

If you select the `toggle` resource and use `POST` you can see how the RED led of the server mote will toggle:

If you do the same with the `Hello` resource, the server will answer you back with a neighbourly well-known message:

And finally if you observe the `Sensors` → `Button` events by selecting it and clicking `OBSERVE`, each time you press the user button an event will be triggered and reported back:

Finally if you go to the `er-example-server.c` file and enable the following defines, you should have more resources available:

```
#define REST_RES_HELLO    1
#define REST_RES_SEPARATE 1
#define REST_RES_PUSHING  1
#define REST_RES_EVENT    1
#define REST_RES_SUB      1
#define REST_RES_LEDS     1
#define REST_RES_TOGGLE   1
#define REST_RES_BATTERY  1
#define REST_RES_RADIO    1
```

And now to get the current RSSI level on the transceiver:

```
coap://[aaaa::c30c:0000:0000:0001]:5683/sensor/radio?p=rssi
```

Do the same to get the battery level readings:

```
coap://[aaaa::c30c:0000:0000:0001]:5683/sensors/battery
```

This last case returns the ADC units when the mote is connected to the USB, the actual value in millivolts would be:

```
V [mV] = (units * 5000)/4096
```

Let's say you want to turn the green LED ON, in the URL type:

```
coap://[aaaa::c30c:0000:0000:0001]:5683/actuators/leds?color=g
```

And then in the payload (the ongoing tab) write:

```
mode="on"
```

And press `POST` or `PUT` (hover with the mouse over the actuators/leds to see the description and allowed methods).

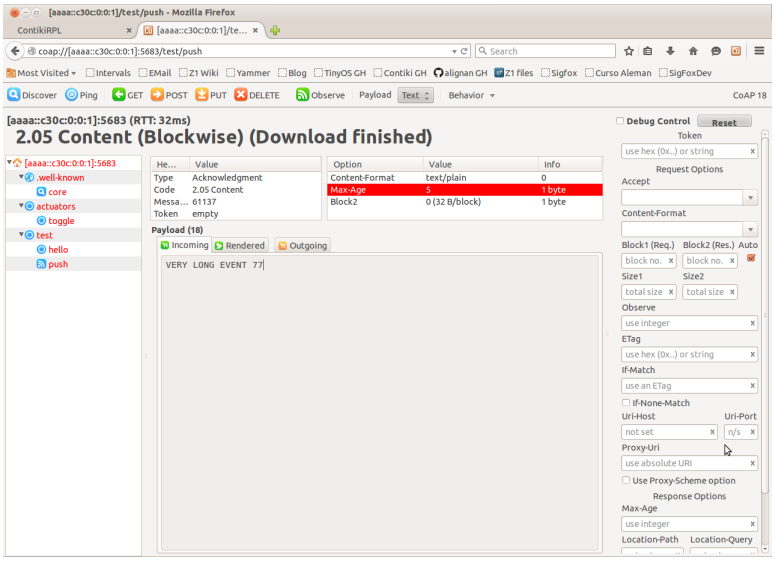

Figure 8.1. Copper CoAP plugin Screenshot

8.2. MQTT example

What is MQTT?

MQTT (formerly MQ Telemetry Transport) is a publish-subscribe based messaging protocol on top of the TCP/IP protocol. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited.

The publish-subscribe messaging pattern requires a message broker. The broker is responsible for distributing messages to interested clients based on the topic of a message.

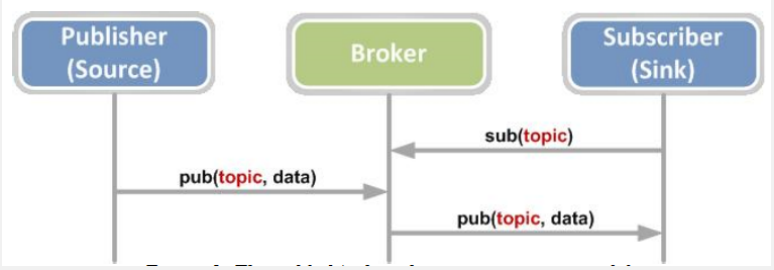

Figure 8.2. MQTT publish/suscribe

MQTT has defined three levels of Quality of Service (QoS):

- QoS 0: The broker/client will deliver the message once, with no confirmation (fire and forget)
- QoS 1: The broker/client will deliver the message at least once, with confirmation required.
- QoS 3: The broker/client will deliver the message exactly once by using a four-step handshake.

Other features of MQTT are:

- Keep-Alive message (PINGREQ, PINGRESP).
- A broker can detect client disconnection, even without an explicit `DISCONNECT` message.
- "Last Will and Testament" message: specified in `CONNECT` message with topic, QoS and retain. On unexpected client disconnection, it the "Last Will and Testament" message is sent to to subscribed clients.
- "Retain" message: a `PUBLISH` message on a topic is kept on the broker which allows a new connected subscriber on the same topic to receive this message (last known good message).
- "Durable" subscription: on client disconnection, all subscriptions are kept on the broker and recovered on client reconnection.

MQTT reserves TCP/IP port 1883 and 8883, the later for using MQTT over SSL.

For more complete information on MQTT, see http://mqtt.org/.

8.2.1. MQTT API

MQTT is implemented in Contiki in `apps/mqtt`. It makes use of the `tcp-socket` library discussed in an early section.

The current MQTT version implemented in Contiki supports QoS levels 0 and 1.

The MQTT available functions are described next. A hands on example in the next section will help to clarify its use and suggest a tate-machine approach to maintain the connection state and device operation.

This function initializes the MQTT engine and shall be called before any other MQTT function.

```
/**
 * \brief Initializes the MQTT engine.
 * \param conn A pointer to the MQTT connection.
 * \param app_process A pointer to the application process handling the MQTT
 *        connection.
 * \param client_id A pointer to the MQTT client ID.
 * \param event_callback Callback function responsible for handling the
 *        callback from MQTT engine.
 * \param max_segment_size The TCP segment size to use for this MQTT/TCP
 *        connection.
 * \return MQTT_STATUS_OK or MQTT_STATUS_INVALID_ARGS_ERROR
 */
mqtt_status_t mqtt_register(struct mqtt_connection *conn,
                            struct process *app_process,
                            char *client_id,
                            mqtt_event_callback_t event_callback,
                            uint16_t max_segment_size);
```

This function connects to a MQTT broker.

```
/**
 * \brief Connects to a MQTT broker.
 * \param conn A pointer to the MQTT connection.
 * \param host IP address of the broker to connect to.
 * \param port Port of the broker to connect to, default is MQTT port is 1883.
 * \param keep_alive Keep alive timer in seconds. Used by broker to handle
 *        client disc. Defines the maximum time interval between two messages
 *        from the client. Shall be min 1.5 x report interval.
 * \return MQTT_STATUS_OK or an error status
 */
mqtt_status_t mqtt_connect(struct mqtt_connection *conn,
                           char *host,
                           uint16_t port,
                           uint16_t keep_alive);
```

This function disconnects from a MQTT broker.

```
/**
 * \brief Disconnects from a MQTT broker.
 * \param conn A pointer to the MQTT connection.
 */
void mqtt_disconnect(struct mqtt_connection *conn);
```

This function subscribes to a topic on a MQTT broker.

```
/**
 * \brief Subscribes to a MQTT topic.
 * \param conn A pointer to the MQTT connection.
 * \param mid A pointer to message ID.
 * \param topic A pointer to the topic to subscribe to.
 * \param qos_level Quality Of Service level to use. Currently supports 0, 1.
 * \return MQTT_STATUS_OK or some error status
 */
mqtt_status_t mqtt_subscribe(struct mqtt_connection *conn,
                             uint16_t *mid,
                             char *topic,
                             mqtt_qos_level_t qos_level);
```

This function unsubscribes from a topic on a MQTT broker.

```
/**
 * \brief Unsubscribes from a MQTT topic.
 * \param conn A pointer to the MQTT connection.
 * \param mid A pointer to message ID.
 * \param topic A pointer to the topic to unsubscribe from.
 * \return MQTT_STATUS_OK or some error status
 */
mqtt_status_t mqtt_unsubscribe(struct mqtt_connection *conn,
                               uint16_t *mid,
                               char *topic);
```

This function publishes to a topic on a MQTT broker.

```
/**
 * \brief Publish to a MQTT topic.
 * \param conn A pointer to the MQTT connection.
 * \param mid A pointer to message ID.
 * \param topic A pointer to the topic to subscribe to.
 * \param payload A pointer to the topic payload.
 * \param payload_size Payload size.
 * \param qos_level Quality Of Service level to use. Currently supports 0, 1.
 * \param retain If the RETAIN flag is set to 1, in a PUBLISH Packet sent by a
 *        Client to a Server, the Server MUST store the Application Message
 *        and its QoS, so that it can be delivered to future subscribers whose
 *        subscriptions match its topic name
 * \return MQTT_STATUS_OK or some error status
 */
mqtt_status_t mqtt_publish(struct mqtt_connection *conn,
                           uint16_t *mid,
```

```
                            char *topic,
                            uint8_t *payload,
                            uint32_t payload_size,
                            mqtt_qos_level_t qos_level,
                            mqtt_retain_t retain);
```

This function sets clients user name and password to use when connecting to a MQTT broker.

```
/**
 * \brief Set the user name and password for a MQTT client.
 * \param conn A pointer to the MQTT connection.
 * \param username A pointer to the user name.
 * \param password A pointer to the password.
 */
void mqtt_set_username_password(struct mqtt_connection *conn,
                                char *username,
                                char *password);
```

This function sets clients Last Will topic and message (payload). If the Will Flag is set to 1 (using the function) this indicates that, if the Connect request is accepted, a Will Message MUST be stored on the Server and associated with the Network Connection. The Will Message MUST be published when the Network Connection is subsequently closed

This functionality can be used to get notified that a device has disconnected from the broker.

```
/**
 * \brief Set the last will topic and message for a MQTT client.
 * \param conn A pointer to the MQTT connection.
 * \param topic A pointer to the Last Will topic.
 * \param message A pointer to the Last Will message (payload).
 * \param qos The desired QoS level.
 */
void mqtt_set_last_will(struct mqtt_connection *conn,
                        char *topic,
                        char *message,
                        mqtt_qos_level_t qos);
```

The following helper functions can be ussed to assert the MQTT connection status, to check if the mote is connected to the broker with `mqtt_connected`, and with `mqtt_ready` if the connection is estabished and there is space in the buffer to publish.

```
#define mqtt_connected(conn) \
```

```
  ((conn)->state == MQTT_CONN_STATE_CONNECTED_TO_BROKER ? 1 : 0)

#define mqtt_ready(conn) \
  (!(conn)->out_queue_full && mqtt_connected((conn)))
```

8.2.2. Hands on: MQTT and mosquitto

Now let's build a simple example using mosquitto[2] MQTT broker v.3.1 running in our host as a MQTT Broker, in the following setup:

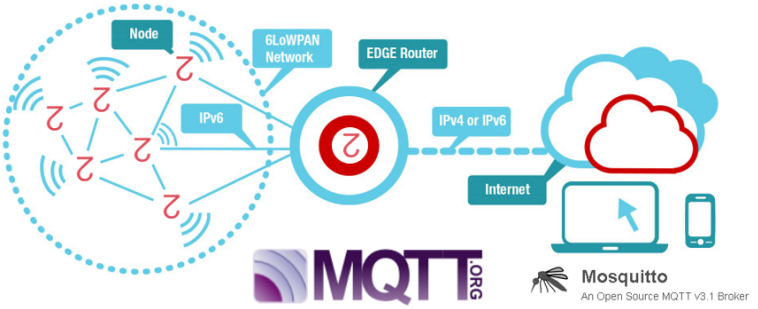

Figure 8.3. MQTT with Mosquitto

The mosquitto installation is quite straightforward, the installation instructions are available in the mosquitto website[3]. For this example we use the default configurations.

Instructions to compile the Border Router are available in previous sections.

We will use the example available at `examples/cc2538dk/mqtt-demo`. With minor tweaks (like removing the cc2538-specific code) it should also work for the **Z1 mote**.

The example is heavily based on the IBM quickstart format, alternatively it can be built to be used with IBM quickstart. For more information about this check out the `README` file, or read the IBM MQTT doc[4].

In the `project-conf.h` file the IPv6 broker address is defined as follows:

```
#define MQTT_DEMO_BROKER_IP_ADDR "aaaa::1"
```

[2] http://mosquitto.org/
[3] http://mosquitto.org/2013/01/mosquitto-debian-repository/
[4] https://docs.internetofthings.ibmcloud.com/messaging/applications.html

As default the `mosquitto` broker binds to the IPv4//IPv6 address of the host, if using the `tunslip6` script with the `aaaa::1/64` address, it should match the `MQTT_DEMO_BROKER_IP_ADDR` definition.

Remember that some examples are meant as platform specific, so you should check if there is a `Makefile.target` predefined, or if the `TARGET` is defined elsewhere.

The example does the following:

- Publish information to an MQTT broker.
- Subscribe to a topic and receive commands from an MQTT broker.

Note also the `PUBLISH_TRIGGER` is mapped to the user button, it can be used to trigger a publish event.

The data structure for the MQTT client configuration is declared as:

```
typedef struct mqtt_client_config {
  char org_id[CONFIG_ORG_ID_LEN];                ❶
  char type_id[CONFIG_TYPE_ID_LEN];              ❷
  char auth_token[CONFIG_AUTH_TOKEN_LEN];        ❸
  char event_type_id[CONFIG_EVENT_TYPE_ID_LEN];  ❹
  char broker_ip[CONFIG_IP_ADDR_STR_LEN];        ❺
  char cmd_type[CONFIG_CMD_TYPE_LEN];            ❻
  clock_time_t pub_interval;                     ❼
  int def_rt_ping_interval;                      ❽
  uint16_t broker_port;                          ❾
} mqtt_client_config_t;
```

❶ Unique organization ID
❷ Device type
❸ Authorization token (if required)
❹ Default event type
❺ Broker IPv6 address
❻ Default command type
❼ Publish interval period
❽ Periodically ping the parent and retrieve RSSI value
❾ Broker default port, default is `1883`

Later defined and populate as follows:

```
static mqtt_client_config_t conf;

static int
init_config()
{
  /* Populate configuration with default values */
  memset(&conf, 0, sizeof(mqtt_client_config_t));
  memcpy(conf.org_id, DEFAULT_ORG_ID, strlen(DEFAULT_ORG_ID));
  memcpy(conf.type_id, DEFAULT_TYPE_ID, strlen(DEFAULT_TYPE_ID));
  memcpy(conf.auth_token, DEFAULT_AUTH_TOKEN, strlen(DEFAULT_AUTH_TOKEN));
  memcpy(conf.event_type_id, DEFAULT_EVENT_TYPE_ID,
         strlen(DEFAULT_EVENT_TYPE_ID));
  memcpy(conf.broker_ip, broker_ip, strlen(broker_ip));
  memcpy(conf.cmd_type, DEFAULT_SUBSCRIBE_CMD_TYPE, 1);
  conf.broker_port = DEFAULT_BROKER_PORT;
  conf.pub_interval = DEFAULT_PUBLISH_INTERVAL;
  conf.def_rt_ping_interval = DEFAULT_RSSI_MEAS_INTERVAL;
  return 1;
}
```

The application example itself can be understood as a finite state machine, despise it seems complicated it is actually very straightforward. The `mqtt_demo_process` starts as follows:

```
PROCESS_THREAD(mqtt_demo_process, ev, data)
{
  PROCESS_BEGIN();

  if(init_config() != 1) {                                          ❶
    PROCESS_EXIT();
  }

  update_config();                                                  ❷

  uip_icmp6_echo_reply_callback_add(&echo_reply_notification,       ❸
                                    echo_reply_handler);
  etimer_set(&echo_request_timer, conf.def_rt_ping_interval);       ❹

  while(1) {

    PROCESS_YIELD();

    if(ev == sensors_event && data == PUBLISH_TRIGGER) {            ❺
      if(state == STATE_ERROR) {
```

```
        connect_attempt = 1;
        state = STATE_REGISTERED;
      }
    }

    if((ev == PROCESS_EVENT_TIMER && data == &publish_periodic_timer) ||
       ev == PROCESS_EVENT_POLL ||
       (ev == sensors_event && data == PUBLISH_TRIGGER)) {          ❻
      state_machine();
    }

    if(ev == PROCESS_EVENT_TIMER && data == &echo_request_timer) {  ❼
      ping_parent();
      etimer_set(&echo_request_timer, conf.def_rt_ping_interval);
    }
  }

  PROCESS_END();
}
```

❶ Initial configuration values, as described earlier

❷ Creates the client ID, publish and subscribe topics. The initial state `STATE_INIT` is set and the `publish_periodic_timer` event is scheduled

❸ Registers the event callback used when a ping event occurs

❹ Starts the periodic ping timer

❺ Allows to try and recover from `STATE_ERROR` by pressing the user button

❻ Handles the `publish_periodic_timer` and button events, this is where the application actually starts

❼ When the periodic ping timer expires, pings the parent

When the `construct_client_id` is first called with the `STATE_INIT`, the `state_machine` is called. A brief walkthrough of the state machine is shown next. Notice how in some events (like `STATE_INIT`) immediately the driver jumps to the next events as there is not a `break` statement.

```
static void
state_machine(void)
{
  switch(state) {

  case STATE_INIT:                                    ❶
    /* If we have just been configured register MQTT connection */
    mqtt_register(&conn, &mqtt_demo_process, client_id, mqtt_event,
```

```
                  MAX_TCP_SEGMENT_SIZE);
    state = STATE_REGISTERED;

  case STATE_REGISTERED:                    ❷
    if(uip_ds6_get_global(ADDR_PREFERRED) != NULL) {
      connect_to_broker();
    } else {
      leds_on(STATUS_LED);
      ctimer_set(&ct, NO_NET_LED_DURATION, publish_led_off, NULL);
    }
    etimer_set(&publish_periodic_timer, NET_CONNECT_PERIODIC);
    return;
    break;

  case STATE_CONNECTING:                    ❸
    leds_on(STATUS_LED);
    ctimer_set(&ct, CONNECTING_LED_DURATION, publish_led_off, NULL);
    /* Not connected yet. Wait */
    DBG("Connecting (%u)\n", connect_attempt);
    break;

  case STATE_CONNECTED:                     ❹
    /* Don't subscribe unless we are a registered device */
    if(strncasecmp(conf.org_id, QUICKSTART, strlen(conf.org_id)) == 0) {
      DBG("Using 'quickstart': Skipping subscribe\n");
      state = STATE_PUBLISHING;
    }

  case STATE_PUBLISHING:                    ❺
    /* If the timer expired, the connection is stable. */
    if(timer_expired(&connection_life)) {
      /*
       * Intentionally using 0 here instead of 1: We want RECONNECT_ATTEMPTS
       * attempts if we disconnect after a successful connect
       */
      connect_attempt = 0;
    }

    if(mqtt_ready(&conn) && conn.out_buffer_sent) {
      /* Connected. Publish */
      if(state == STATE_CONNECTED) {
        subscribe();
        state = STATE_PUBLISHING;
      } else {
        leds_on(STATUS_LED);
        ctimer_set(&ct, PUBLISH_LED_ON_DURATION, publish_led_off, NULL);
        publish();
```

```
    }
    etimer_set(&publish_periodic_timer, conf.pub_interval);
    return;
  } else {
    /*
     * Our publish timer fired, but some MQTT packet is already in flight
     * (either not sent at all, or sent but not fully ACKd).
     *
     * This can mean that we have lost connectivity to our broker or that
     * simply there is some network delay. In both cases, we refuse to
     * trigger a new message and we wait for TCP to either ACK the entire
     * packet after retries, or to timeout and notify us.
     */
  }
  break;

case STATE_DISCONNECTED:                    ❻
  DBG("Disconnected\n");
  if(connect_attempt < RECONNECT_ATTEMPTS ||
     RECONNECT_ATTEMPTS == RETRY_FOREVER) {
    /* Disconnect and backoff */
    clock_time_t interval;
    mqtt_disconnect(&conn);
    connect_attempt++;

    interval = connect_attempt < 3 ? RECONNECT_INTERVAL << connect_attempt :
      RECONNECT_INTERVAL << 3;

    DBG("Disconnected. Attempt %u in %lu ticks\n", connect_attempt, interval);

    etimer_set(&publish_periodic_timer, interval);

    state = STATE_REGISTERED;
    return;
  } else {
    /* Max reconnect attempts reached. Enter error state */
    state = STATE_ERROR;
    DBG("Aborting connection after %u attempts\n", connect_attempt - 1);
  }
  break;

case STATE_CONFIG_ERROR:                    ❼
  /* Idle away. The only way out is a new config */
  printf("Bad configuration.\n");
  return;

case STATE_ERROR:                           ❽
```

```
  default:
    leds_on(STATUS_LED);
    /*
     * 'default' should never happen.
     *
     * If we enter here it's because of some error. Stop timers. The only thing
     * that can bring us out is a new config event
     */
    printf("Default case: State=0x%02x\n", state);
    return;
  }

  /* If we didn't return so far, reschedule ourselves */
  etimer_set(&publish_periodic_timer, STATE_MACHINE_PERIODIC);
}
```

❶ Entry point, register the mqtt connection and move to the `STATE_REGISTERED` event

❷ Attempts to connect to the broker. If the node has not joined the network (doesn't have a valid IPv6 global address) it retries later. If the node has a valid address then calls the `mqtt_connect` function and sets the state to `STATE_CONNECTING`, then sets the `publish_periodic_timer` with a faster pace

❸ This event just informs the user about the connection attemps. When the MQTT connection to the broker has been made, the `MQTT_EVENT_CONNECTED` is triggered at the `mqtt_event` callback handler

❹ As we are connected now, proceed and publish. As we should not be using IBM's quickstart, we skip changing the state to `STATE_PUBLISHING` and just go ahead

❺ Checks if the MQTT connection is OK in `mqtt_ready`, then subscribe and publish

❻ Handles any disconnection event triggered from `MQTT_EVENT_DISCONNECTED`

❼ Halts the application, only allows valid configuration values

❽ Default event handler, stops the timer and do nothing

The `publish` function create the string data to be published. Below is a snippet of the function highlighting only the most relevant parts. The example publish periodically the following:

- Device name.
- An incremental sequence number.
- Device uptime (in seconds).
- On-chip temperature.
- Battery value.

```
static void
publish(void)
{
  len = snprintf(buf_ptr, remaining,
                 "{"
                 "\"d\":{"
                 "\"myName\":\"%s\","
                 "\"Seq #\":%d,"
                 "\"Uptime (sec)\":%lu",
                 BOARD_STRING, seq_nr_value, clock_seconds());

  len = snprintf(buf_ptr, remaining, ",\"Def Route\":\"%s\",\"RSSI (dBm)\":%d",
                 def_rt_str, def_rt_rssi);

  len = snprintf(buf_ptr, remaining, ",\"On-Chip Temp (mC)\":%d",
                 cc2538_temp_sensor.value(CC2538_SENSORS_VALUE_TYPE_CONVERTED));

  len = snprintf(buf_ptr, remaining, ",\"VDD3 (mV)\":%d",
                 vdd3_sensor.value(CC2538_SENSORS_VALUE_TYPE_CONVERTED));

  mqtt_publish(&conn, NULL, pub_topic, (uint8_t *)app_buffer,
               strlen(app_buffer), MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);

  DBG("APP - Publish!\n");
}
```

The `mqtt_publish` updates the MQTT broker with the new values, publishing to the specified `pub_topic`. The default topic is `iot-2/evt/status/fmt/json` as done in the `construct_pub_topic` function. This topic follows IBM's format but it can be otherwise changed accordingly.

When receiving an event from a topic we are subscribed to, the `MQTT_EVENT_PUBLISH` event is triggered and the `pub_handler` is called. The example allows to turn the red LED on and off alternatively.

The default topic the example subscribe to is `iot-2/cmd/+/fmt/json`, specifically to change the LED status we would need to publish to the `iot-2/cmd/leds/fmt/json` topic with value `1` to turn the LED on, and `0` otherwise.

```
static void
pub_handler(const char *topic, uint16_t topic_len, const uint8_t *chunk,
            uint16_t chunk_len)
{
  if(strncmp(&topic[10], "leds", 4) == 0) {
```

```
    if(chunk[0] == '1') {
      leds_on(LEDS_RED);
    } else if(chunk[0] == '0') {
      leds_off(LEDS_RED);
    }
    return;
  }
}
```

Note the `+` in the subscribed topic, this allows to have different subsets of topics, such as `leds`.

8.3. Hands on: connecting to a real world IoT platform (HTTP-based)

Ubidots (www.ubidots.com) is an IoT cloud platform Ubidots that helps you create applications that capture real-world data and turn it into meaningful actions and insights.

8.4. Ubidots IPv6 example in native Contiki

The example will demonstrate the basic functionality of Contiki's Ubidots library:

- How to use the library to POST to a variable.
- How to use the library to POST to a collection.
- How to receive (parts of) the HTTP reply.

At the present time the Ubidots example was to be merged to Contiki, however the functional example can be browsed and forked from the following:

https://github.com/g-oikonomou/contiki/tree/ubidots-demo

The Contiki's Ubidots Library was written by George Oikonomou.

The Ubidots example is located at `examples/ipv6/ubidots`.

Ubidots application is implemented at `apps/ubidots`.

Ubidots application uses TCP sockets to connect to the host `things.ubidots.com`, which has the following IPv4 and IPv6 endpoints:

Figure 8.4. Ubidots endpoint IPv4/IPv6 addresses

To check what's going on enable the debug print statements in the `ubidots.c` file, search for `#define DEBUG DEBUG_NONE` and replace with:

```
#define DEBUG DEBUG_PRINT
```

As default the `ubidots` application uses the `things.ubidots.com` remote host, however if no NAT/NAT64 service is available to resolve the host address, the IPv6 endpoint can be explicitly defined as:

```
#define UBIDOTS_CONF_REMOTE_HOST "2607:f0d0:2101:39::2"
```

If you don't have a local IPv6 connection, services like gogo6[5] and hurricane electric[6] provides IPv6 tunnels over IPv4 connections. Other options like wrapsix[7] allows to have NAT64 to translate IPv6/IPv4 addresses.

The Ubidots demo posts every 30 seconds the Z1 mote's **uptime** and **sequence number**, so as done before in the past sections we need to create these two variables at Ubidots. Create

[5] http://www.gogo6.com/
[6] https://ipv6.he.net/
[7] http://www.wrapsix.org/

the data source, its variables and then open `project-conf.h` file and replace the following accordingly:

```
#define UBIDOTS_DEMO_CONF_UPTIME       "XXXX"
#define UBIDOTS_DEMO_CONF_SEQUENCE     "XXXX"
```

The last step is to assign an Ubidot's fixed Short Token so we don't have to request one from time to time when it expires, get one and add this to the `Makefile`, the file should look like this:

```
DEFINES+=PROJECT_CONF_H=\"project-conf.h\"
CONTIKI_PROJECT = ubidots-demo
APPS = ubidots
UBIDOTS_WITH_AUTH_TOKEN=XXXXXXXX
ifdef UBIDOTS_WITH_AUTH_TOKEN
  DEFINES+=UBIDOTS_CONF_AUTH_TOKEN=\"$(UBIDOTS_WITH_AUTH_TOKEN)\"
endif
all: $(CONTIKI_PROJECT)
CONTIKI_WITH_IPV6 = 1
CONTIKI = ../../..
include $(CONTIKI)/Makefile.include
```

Note that you should replace the `UBIDOTS_WITH_AUTH_TOKEN` **without using ""** quotes.

Now everything should be set, let's compile and program a Z1 mote!

```
make TARGET=z1 savetarget
make clean && make ubidots-demo.upload && make z1-reset && make login
```

You should see the following output:

```
connecting to /dev/ttyUSB0 (115200) [OK]
Rime started with address 193.12.0.0.0.0.0.158
MAC c1:0c:00:00:00:00:00:9e Ref ID: 158
Contiki-d368451 started. Node id is set to 158.
nullmac nullrdc, channel check rate 128 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:009e
Starting 'Ubidots demo process'
Ubidots client: STATE_ERROR_NO_NET
Ubidots client: STATE_ERROR_NO_NET
Ubidots client: STATE_ERROR_NO_NET
Ubidots client: STATE_STARTING
Ubidots client: Checking 64:ff9b::3217:7c44
```

```
Ubidots client: 'Host: [64:ff9b::3217:7c44]' (remaining 44)
Ubidots client: STATE_TCP_CONNECT (1)
Ubidots client: Connect 64:ff9b::3217:7c44 port 80
event_callback: connected
Ubidots client: STATE_TCP_CONNECTED
Ubidots client: Prepare POST: Buffer at 199
Ubidots client: Enqueue value: Buffer at 210
Ubidots client: POST: Buffer at 211, content-length 13 (2), at 143
Ubidots client: POST: Buffer at 208
Ubidots client: STATE_POSTING (176)
Ubidots client: STATE_POSTING (176)
Ubidots client: STATE_POSTING (144)
Ubidots client: STATE_POSTING (112)
Ubidots client: STATE_POSTING (80)
Ubidots client: STATE_POSTING (48)
Ubidots client: STATE_POSTING (16)
Ubidots client: STATE_POSTING (0)
Ubidots client: HTTP Reply 200
HTTP Status: 200
Ubidots client: New header: <Server: nginx>
Ubidots client: New header: <Date: Fri, 13 Mar 2015 09:35:08 GMT>
Ubidots client: New header: <Content-Type: application/json>
Ubidots client: New header: <Transfer-Encoding: chunked>
Ubidots client: New header: <Connection: keep-alive>
Ubidots client: New header: <Vary: Accept-Encoding>
Ubidots client: Client wants header 'Vary'
H: 'Vary: Accept-Encoding'
Ubidots client: New header: <Vary: Accept>
Ubidots client: Client wants header 'Vary'
H: 'Vary: Accept'
Ubidots client: New header: <Allow: GET, POST, HEAD, OPTIONS>
Ubidots client: Chunk, len 22: <[{"status_code": 201}]> (counter = 22)
Ubidots client: Chunk, len 0: <(End of Reply)> (Payload Length 22 bytes)
P: '[{"status_code": 201}]'
```

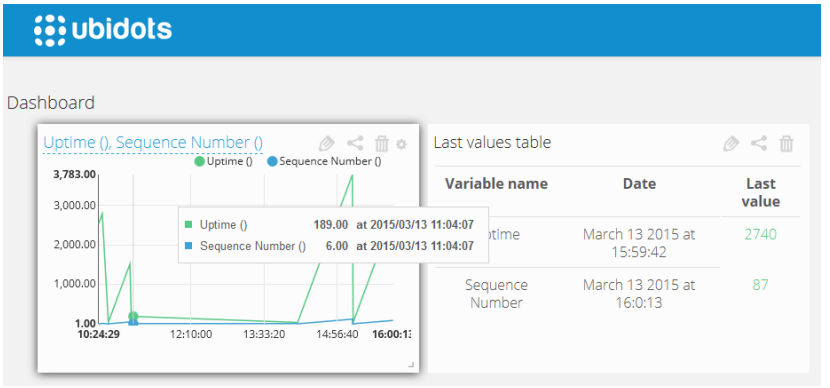

Figure 8.5. Ubidots graphs

The values are displayed using a **Multi-line chart** and a **Table-Values** dashboard.

Abbreviations and definitions

CoAP
Constrained Application Protocol

DHCPv6
Dynamic Host Configuration Protocol for IPv6

IANA
Internet Assigned Numbers Authority

IETF
Internet Engineering Task Force

IID
Interface Identifier (in an IPv6 address)

IP
Internet Protocol

IPv4
Internet Protocol version 4

IPv6
Internet Protocol version 6

LAN
Local Area Network

MAC
Medium Access Control

MQTT
Message Queuing Telemetry Transport

NAT
Network Address Translation

SLAAC
Stateless Address Autoconfiguration

TCP
Transmission Control Protocol

UDP
User Datagram Protocol

WSN

Wireless Sensor Network

Bibliography

References

[IoT] Ovidiu Vermesan & Peter Fress, “Internet of Things –From Research and Innovation to Market Deployment”, River Publishers Series in Communication, ISBN: 87-93102-94-1, 2014.

[SusAgri] Rodriguez de la Concepcion, A.; Stefanelli, R.; Trinchero, D. Adaptive wireless sensor networks for high-definition monitoring in sustainable agriculture, Wireless Sensors and Sensor Networks (WiSNet), 2014

[sixlo] IETF WG 6Lo: http://datatracker.ietf.org/wg/6lo/charter/

[sixlowpan] IETF WG 6LoWPAN: http://datatracker.ietf.org/wg/6lowpan/charter/

[IANA-IPV6-SPEC] IANA IPv6 Special-Purpose Address Registry: http://www.iana.org/assignments/iana-ipv6-special-registry/

[IEEE802.15.4] IEEE Computer Society, "IEEE Std. 802.15.4-2003", October 2003

[RFC2460] S. Deering, R. Hinden, “Internet Protocol, Version 6 (IPv6) Specification”, December 1998, RFC 2460, Draft Standard

[RFC3315] Droms, R., Bound, J., Volz, B., Lemon, T., Perkins, C., and M. Carney, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", July 2003

[RFC4193] R. Hinden, B. Haberman, "Unique Local IPv6 Unicast Addresses", RFC4193, October 2005

[RFC4291] R. Hinden, S. Deering, “IP Version 6 Addressing Architecture”, RFC 4291, February 2006

[RFC4862] S. Thomson, T. Narten, T. Jinmei, "IPv6 Stateless Address Autoconfiguration", September 2007

[RFC6775] Z. Shelby, Ed., S. Chakrabarti, E. Nordmark, C. Bormann, “Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)”, November 2012, RFC 6775, Proposed Standard

[RFC6890] M. Cotton, L. Vegoda, R. Bonica, Ed., B. Haberman, "Special-Purpose IP Address Registries", RFC 6890 / BCP 153, April 2013

[roll] roll IETF WG: http://datatracker.ietf.org/wg/roll/charter

[l802154r] L. Frenzel, "What's The Difference Between IEEE 802.15.4 And ZigBee Wireless?" [Online] Available: http://electronicdesign.com/what-s-difference-between/what-s-difference-between-ieee-802154-and-zigbee-wireless.


RESPICE PROSPICE
UNIVERSITY of the
WESTERN CAPE

ICTP
The Abdus Salam
International Centre
for Theoretical Physics

NODO6
zolertia™