IoT in 5 days

Author's Name

Table of Contents

List of Figures

Chapter 1. Introduction to IoT

Chapter 2. Introduction to IoT

2.1. Introduction

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 ofinterconnecting 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.

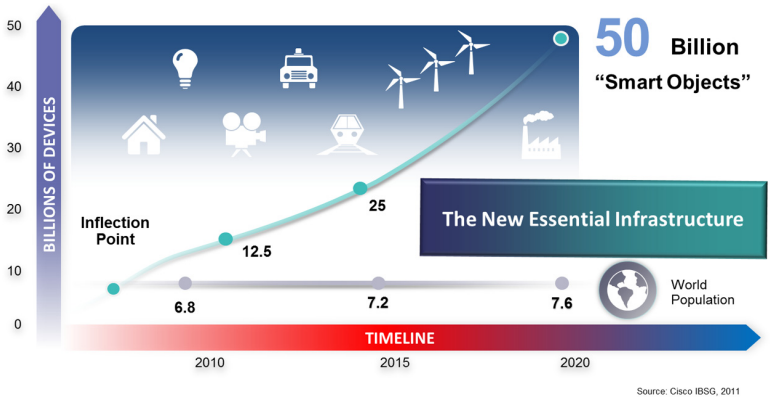


Figure 2.1. Internet-connected devices and the future evolution (Source: Cisco, 2011)

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.

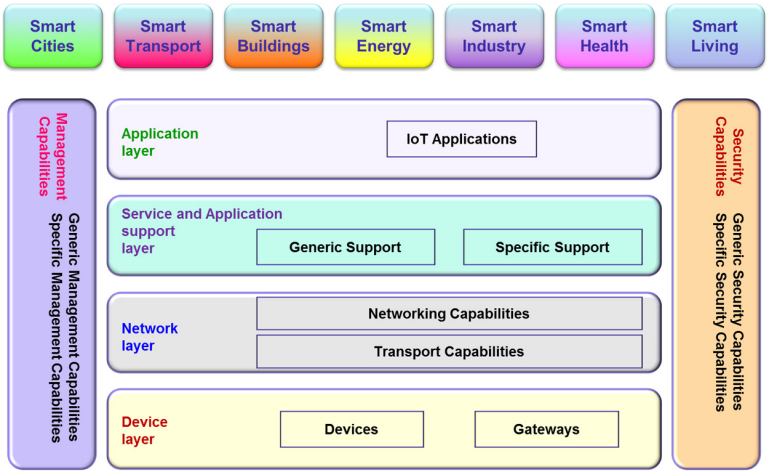


Figure 2.2. IoT Layered Architecture (Source: ITU-T)

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.

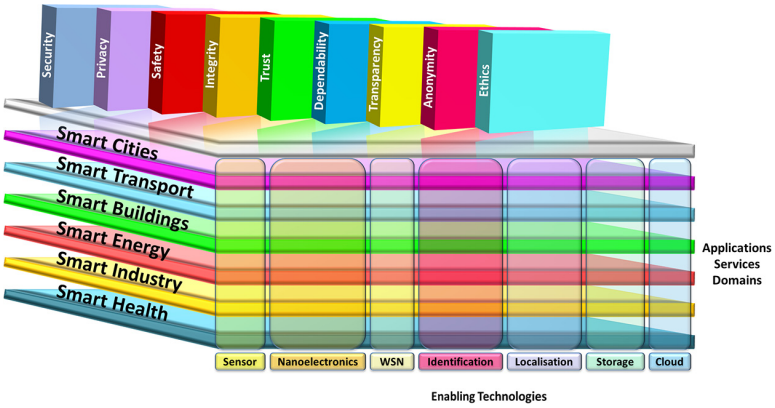


Figure 2.3. IoT-3Dimentional View (Source: [1])

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.

2.2. 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.

2.3. Applications

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) [2].
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.

2.4. 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. References.

[1] 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.

[2] 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

Chapter 4. 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.

4.1. A little bit of History

ARPANET in the early 1980's 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 completeley different systems. The first widely use 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

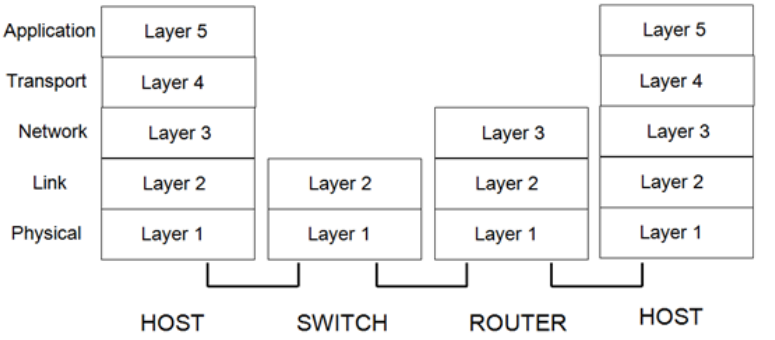
rapid depletion in the number of IP addresses available under IPv4, which was never designed to scale to these levels.

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.

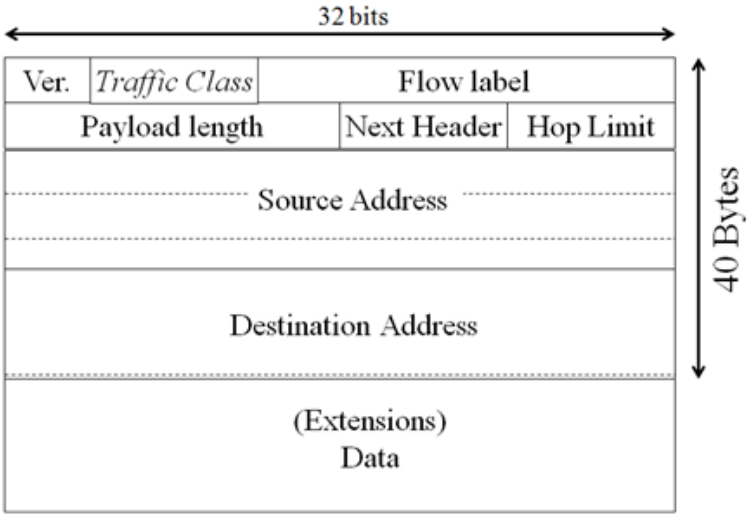
As you can see, the history of IP and Internet are almost the same, and because of this the growth of Internet is been hampered by the limitations of IPv4, and has led to the development of a new version of IP, IPv6, as the protocol to be used to interconnect all sorts of devices to send and/or receive 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.

4.2. 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. You need to have understood the concepts covered in the Networking Basics chapter, and be familiar with bits, bytes, networking stack, network layer, packets, IP header, etc. You should understand that IPv6 is a different protocol, non-compatiblewith to IPv4. In the following figure we represent the layered model used in the Internet.



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. The first thing you have to know is what an IPv6 packet looks like:



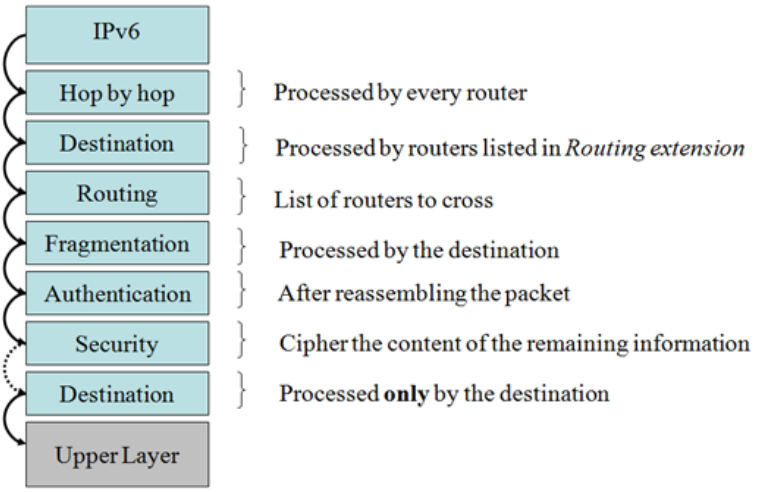
First you have the **basic IPv6 header** with a fixed size of 40 bytes, followed by upper layer data and optionally by some extension headers, that will be covered later. As you can see there are several fields in the packet header, with some improvements as compared with IPv4 header:

- 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.



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.

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.

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

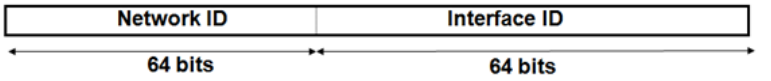
4) If we apply the fifth rule, allowing compression of one or more consecutive groups of zeroes using "::", we have: 2001:db8:4004:10::6543:ffd

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.

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 allways 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:



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.

The following are some types of unicast addresses:

- **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): 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.
- **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.

4.3. 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.

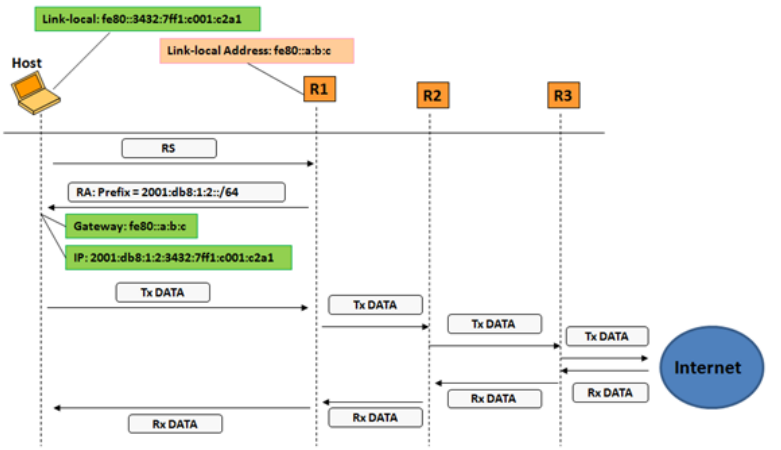
The availability of a huge amount of addresses has allowed a new mechanism called **stateless address autoconfiguration** (SLAAC) that didn't exist with IPv4. Following is a brief summary of the ways you can configure an address on an IPv6 interface:

- **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): 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 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:



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.

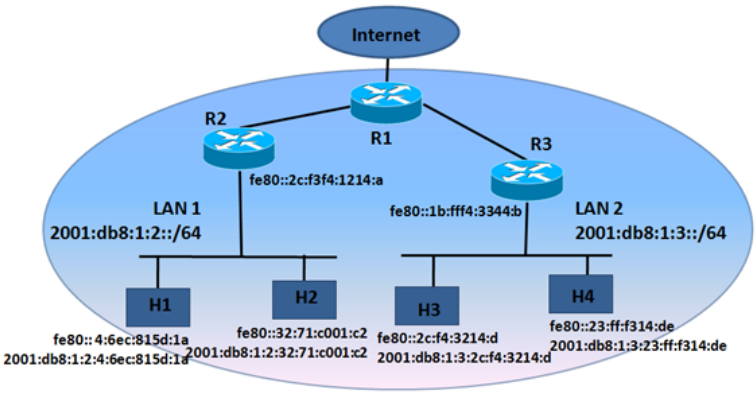
3) The hosts needs two basic things to be able to send packets to other networks: a global IPv6 address and the address of a gateway, i.e., a router to which to send the packets it wants to get routed outside its network.

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: 4.1) A global prefix of length 64 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 4.2) 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).

4.4. Network Example

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



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.

4.5. IPv6 Excercises

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

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)

4.6. Addressing Exercises

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

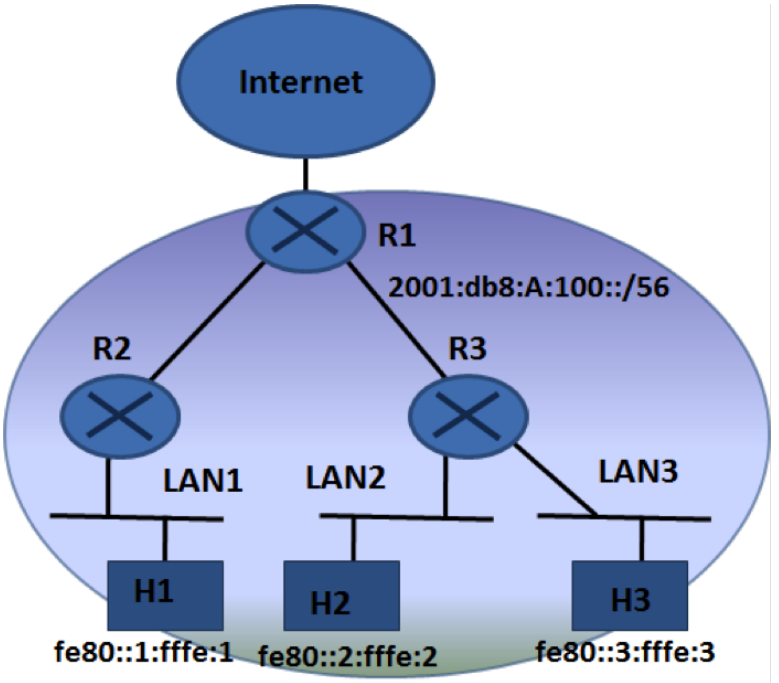
1. 2001:0db8:00A0:7200:0fe0:000B:0000:0005
2. 2001:0db8::DEFE:0000:C000
3. 2001:db8:DAC0:0FED:0000:0000:0B00:12

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:



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 5. Short introduction to Contiki

What is Contiki OS, how do I install it?, everything should be pre-installed in *Instant Contiki* already!

- Meet the Zolertia Z1 mote: identify sensors, connectors, antenna.
- Check the installation: test the toolchain installation with the *hello world* example

My first application: hello world with LEDs:

- Use the LEDs and printf to debug.
- Change timer values, triggers.

Adding sensors: analogue and digitals

- Difference between both, basics.
- How to connect and read: ADC, I2C.

Our first application: relay data over the serial port to Ubidots.

Chapter 6. What is Contiki OS?

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-power 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/

Chapter 7. How to install

7.1. Install VMWare for your platform

On Windows and Linux: 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

7.2. Download Instant Contiki:

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

Download the 23-bit version: `Instant_Contiki_Ubuntu_12.04_32-bit.vmdk`

7.3. Start Instant Contiki

Using VMWare just open the InstantContiki2.7.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.

7.4. Updating to the latest Contiki release

The Contiki OS git repository is hosted at Github, actively developed by a growing community of developers and enthusiasts, track the current work and don't miss the latest features and improvements at:

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

What is GIT

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. The main difference with previous change control tools like Subversion, 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. Making branches and merging between branches is really easy.

To learn more about GIT there are some great tutorials online:

http://try.github.io

http://excess.org/article/2008/07/ogre-git-tutorial/

A nice graphical introduction to git: 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. Unlike Git, which is strictly a command-line tool, 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, and bug tracking and feature requests for every project.

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

To update the Instant Contiki source code to the latest commit available, open a terminal and write:

```
cd $HOME/contiki
git fetch origin master
git pull origin master
```

This will synchronize and download the latest code at the *master* branch, to visualize the commit history just type:

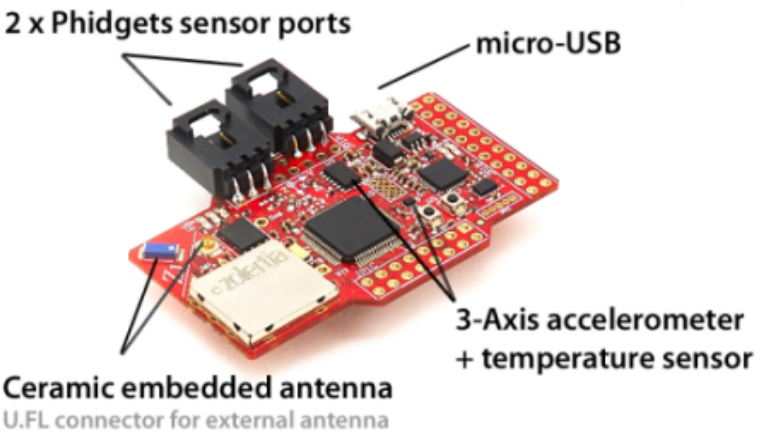
```
git log
```

As we will be working on Contiki OS and changing/adding our own code, let's create a *work* branch by typing:

```
git checkout -b work
```

7.5. Zolertia Z1 platform

For the remainder of the exercises we will use the **Zolertia Z1 mote** as the target development platform.



The Z1 mote features a second generation MSP430F2617 low power 16-bit RISC CPU @16MHz MCU, 8KB RAM and a 92KB Flash memory. Also includes the well known CC2420 transceiver, IEEE 802.15.4 compliant, which operates at 2.4GHz with an effective data rate of 250Kbps.

The Zolertia Z1 mote has been ported to TinyOS, Contiki OS, OpenWSN and RIOT, and has been used actively over 5 years now in Universities, Research and development centers and commercial products in more than 43 countries, featured in more than 50 scientific publications and

More information about can be found at: http://www.zolertia.io

7.6. Check the toolchain version and installation

Open a terminal and write:

```
msp430-gcc --version
```

The output should be the following:

```
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.
```

To install the toolchain yourself (not required if it is already installed), download it from here: http://sourceforge.net/projects/zolertia/files/Toolchain/

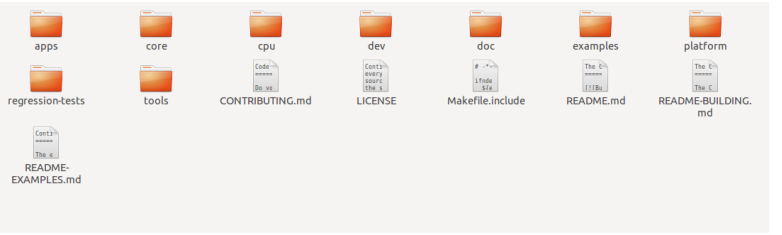
Afterwards just decompress (at home directory for example) and include its path by editing the profile session:

```
sudo gedit /home/user/.bashrc
```

And add at the end the following:

```
export PATH=$PATH:/home/user/msp430-47/bin
```

7.7. Contiki structure



- examples: Contiki's ready to build examples, at examples/z1 you will find specific Zolertia related examples.
- app: Contiki applications.
- cpu: specific MCU files, the /cpu/msp430/ folder contains the drivers of the MCU used by the Z1 mote.
- platform: specific device files and drivers, platform/z1/ has specific drivers and configuration for the Z1 mote.

- core: Contiki's filesystem and core components.
- tools: a variety of tools for debugging, simulating and enhancing your Z1 applications. in tools/z1 there are specific tools such as the BSL script for programming the devices. Also two simulation tools provided by Contiki: COOJA and MSPSim are also included.
- doc: self-generated Doxygen documentation.
- regression-tests: set of nightly regression tests that test important aspects of Contiki on a daily basis

7.8. Check installation: examples

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

```
cd examples/hello-world
make TARGET=z1 savetarget
```

So it know that when you compile you do so for the z1 mote. You need to do this only once per application.

```
make hello-world
```

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

```
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.

7.9. Check z1 connection to the virtual machine

Connect the Z1 mote via USB using a regular micro-USB cable, the VMWare instance should recognize the device as follows:

- In VM player: Player → Removable Devices → Signal Integrated Zolertia Z1 → Connect

- In VMWare Fusion: Devices → USB Devices → Silicon Labs Zolertia Z1

Write in the previous terminal:

```
make z1-motelist
../../tools/z1/motelist-z1
Reference  Device        Description
-----------------------------------------------
Z1RC0336   /dev/ttyUSB0   Silicon Labs Zolertia Z1
```

This will list the Z1 motes connected to the Virtual Machine (you can several at the same time).

Save the reference ID for next lab sessions (Z1RC0336). Each mote has a unique reference number. The port name is useful for programming and debugging.

Now upload the precompiled Hello World application to the Z1 mote.

```
make hello-world.upload  MOTES=/dev/ttyUSB0
```

if you don't use the *MOTES* argument, the system will install on the first device it finds. It is OK if you only have one device.

If you get this error:

```
serial.serialutil.SerialException: could not open port /dev/ttyUSB0: [Errno
 13] Permission denied: '/dev/ttyUSB0'
```

You need to add yourself to the *dialout* group of Linux. You can do it as follows:

```
sudo usermod -a -G dialout user
```

Enter the root password, which is **user** then reboot.

```
sudo reboot
```

To restart the Z1 mote and visualize the debug output in the terminal, let's chain the following commands:

```
make hello-world.upload && make z1-reset && make login
```

The Z1 mote prints to the console over the USB port the debug information from the application.

```
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-2.6-2067-g2734c97 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 'Hello world process'
Hello, world
```

Note that the node ID is displayed as well as the MAC address.

Quick Z1 cheat sheet

The Makefile.z1 provides useful commands platform/z1/Makefile.z1, which is included when TARGET=z1 is passed to the command line, use the command `make` with the following:

- `savetarget` : save the z1 platform as default target inside the Makefile.target file, from now on we can compile by just typing make without TARGET=z1
- `z1-motes` : list the devices of the z1 motes connected to the computer.
- `z1-motelist` : list the reference of the z1 motes connected to the computer, along with the device.
- `z1-reset` : reset the connected nodes.
- `z1-upload` : load a previously compiled tmpimage.ihex to the z1 mote.
- `linslip` : makes a serial connection over IP to the node.
- `serialdump` : print the timestamp and the output of the z1 mote.
- `serialview` : same as serialdump, but allows interaction with the z1 mote.
- `login` : prints the output of the connected node, without the timestamp.
- `MOTES=/dev/ttyUSBX` : points out the device to the bsl

Chapter 8. My first applications

8.1. Hello world with LEDs

Let’s see the main components of the Hello World example. View the code with:

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

When starting Contiki, you declare processes with a name. In each code you can have more 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.

The next step is adding a LED and the user button.

Let’s create a new file at:

```
cd examples/z1
```

Let's name the new file `test_led.c` with

```
gedit test_led.c.
```

You have to add the `dev/leds.h` which is the library to manage the LEDs (light emitting diodes). To check the available functions go to `/home/user/contiki/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);
```

Available LEDs:

```
LEDS_GREEN
LEDS_RED
LEDS_BLUE
LEDS_ALL
```

Now try to turn ON only the red LED and see what happens

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

We now need to add the project to the `Makefile` so when running the *make* command it will be compiled.

```
CONTIKI_PROJECT += test_led
```

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

```
make clean && make test_led.upload
```

The *make clean* command is used to erase previously compiled objects, this is **very important** as if you make changes to the source code, you must rebuilt the files, otherwise your change might not be pulled in.

Now the red LED should be ON!

> Exercise: try to switch on the other LEDs.

8.2. Printf

You can use prinf to visualize on the console what is happening in your application. It is really useful to debug your code, as you can print i.e 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.

```
#include "contiki.h"
#include "dev/leds.h"
#include <stdio.h>
char hello[] = "hello from the mote!";
/*---------------------------------------------------*/
PROCESS(led_process, "led process");
AUTOSTART_PROCESSES(&led_process);
/*---------------------------------------------------*/
PROCESS_THREAD(led_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 (LEDs are numbered 1, 2 and 4).

Exercise: what happens when you turn on more than one LED? What number do you get?

8.3. Button

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

Create a new file in `/home/user/contiki/examples/z1` called `test_button.c` The button in Contiki is considered as a sensor. We are going to use the `core/dev/button-sensor.h` library. It is a good practice to give the process a meaningful name so it reflects what the process is about. Here is the code to print the button status:

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

Let's modify the `Makefile` to add the new file.

```
CONTIKI_PROJECT += test_button
```

You can leave the previously created test_led in the makefile. 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.

8.4. Timers

Using timers will allow us to trigger events at a given time, fastening the transition from one state to another and making a given process or task automated, for example blinking a 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 have 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 post an event signaling 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.

Create a new file in `/home/user/contiki/examples/z1` called `test_timer.c`.

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.

```
#include "contiki.h"
#include "dev/leds.h"
#include "dev/button-sensor.h"
#include <stdio.h>
#define  SECONDS 2
/*---------------------------------------------------*/
PROCESS(hello_timer_process, "hello world with timer example");
AUTOSTART_PROCESSES(&hello_timer_process);
```

```
/*---------------------------------------------------*/
PROCESS_THREAD(hello_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.

Chapter 9. Sensors

The Z1 mote has **two built in digital sensors**: temperature and 3-axis accelerometer, as well as support to interface out-of-the-box most analogue sensors. The main difference between analog and digital sensors are the power consumption (lower in digital) and the protocol they use.

Analog sensors requires to typically being connected to an ADC (analog to digital converters) to translate the analog (continuous) reading to an equivalent digital value in millivolts. The quality and resolution of the measure depends on both the ADC (resolution is 12 bits in the Z1) and on the sampling frequency.

As a rule of thumb, you need to have double sampling frequency as the phenomena you are measuring. As an example, if you want to sample human sound (8 kHz) you need to sample at twice that frequency (16 kHz minimum).

Digitals 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.

9.1. Analog Sensors

The Z1 mote has one built-in analog sensor to get the battery level expressed in milliVolts.

There is an example in the Contiki example folder called `test-battery.c`. The example includes the battery level driver (`battery-sensor.h`). It activates the sensor and prints as fast as possible (with no delay) the battery level.

When working with the ADC you need to convert the ADC integers in milliVolts. This is done with the following formula:

```
float mv = (battery * 2.500 * 2) / 4096;
```

We multiply by the voltage reference and divide the raw value by 4096, as the precision of the ADC is 12 bits (1<<12).

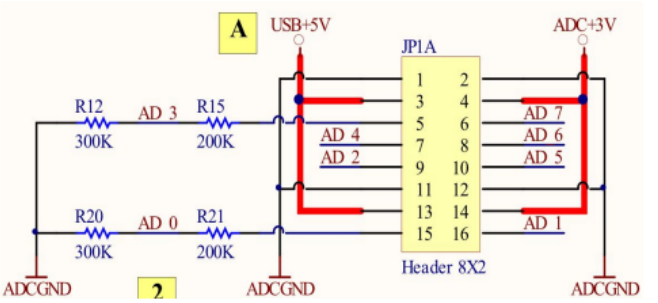
The Z1 is powered at 3.3V but it can be powered at 5V over the USB. There is an internal voltage divider that converts from 5V to 3.3V when using an analogue sensor in a specific

Phidget port, which are basically a connector with a given pin-out based on the commercially available *Phidget* 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



More information about the Z1 mote pinout can be browsed directly from its datasheet at:

http://zolertia.com/sites/default/files/Zolertia-Z1-Datasheet.pdf

9.2. External analog sensor:

We can connect an external analog sensor to any of the available *phidget* ports. As an example, let's connect the Phidget 1142 **precision light sensor**.

It is important to know the voltage required by each sensor. If the sensor can be powered at 3V, it should be connected to the *Phidgets* connector in the top row. If the sensor is powered at 5V it can be safely connected to the *Phidgets* bottom row. Only if the mote is powered by USB, then you can use the 5V sensor.

The Precision Light Sensor product information is available at:

http://www.phidgets.com/products.php?product_id=1142_0

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

Connect the light sensor to the 3V *Phidget* connector. As this is an official example, there is no need to add it to the Makefile (it is already there!). Let's compile the example code:

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

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 **Phidget 3V2 connector**, so the raw value is 2202. Try to illuminate the sensor with a flashlight (from your mobile phone, for example) and then to keep it in 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
Response time	2ms
Light level min	1 lux
Supply Voltage Min	2.4V
Supply Voltage Max	5.5V
Max current consumption	5mA
Light level max (3.3V)	660 lux
Light level max (5V)	**1000 lux**

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

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

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.

9.3. Internal digital sensor

As said earlier, the Z1 mote has a built-in ADXL345 3 axis accelerometer, and 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°.

More information here:

http://www.analog.com/en/products/mems/mems-accelerometers/adxl345.html

Parameter	**Value**
Sensor type	Accelerometer
Max data rate	3200 Hz
Max resolution	13 bits
Max current consumption	**140uA**

You don't need to add it to the Makefile. Once uploaded, this is the result:

```
[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 one. 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);
```

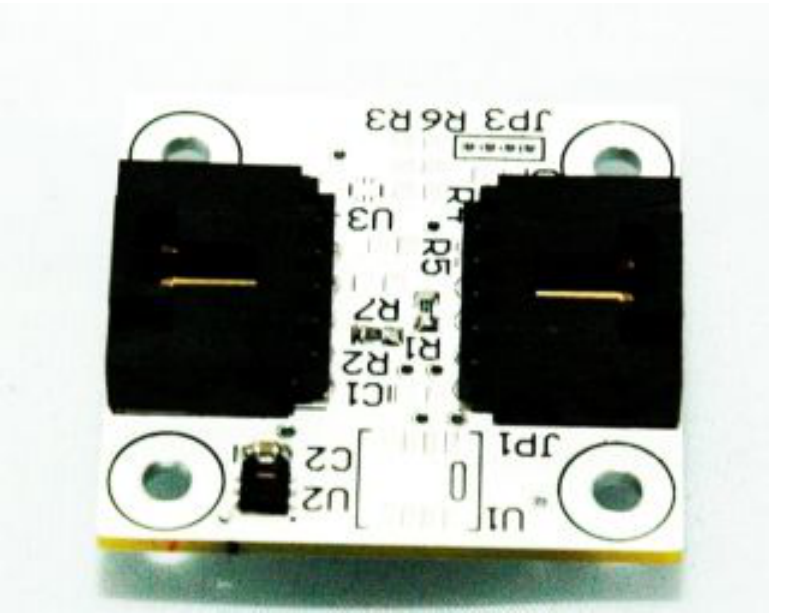
We then need to enable the interrupts like this:

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

In the while cycle 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 max acceleration by shaking the mote. Turn on and off the LED according to the acceleration on one axis.

9.4. External digital sensor



The **ZIG-SHT25** is an I2C digital temperature and humidity sensor based on the SHT25 sensor from Sensirion.

Supply Voltage [V]: 2.1 - 3.6 Energy Consumption: 3.2uW (at 8 bit, 1 measurement / s) RH Operating Range: 0 - 100% RH Temp. Operating Range: -40 - +125°C (-40 - +257°F) RH Response Time: 8 sec (tau63%)

Parameter	**Value**
Sensor type	Temperature and Humidity
Data range	0-100%RH (humidity), -40-125ºC (temperature)
Max resolution	14 bits (temperature), 12 bits (humidity)
Max current consumption	**300uA**

More information available at:

http://webshop.zolertia.com/product_info.php/cPath/29_30/products_id/79

The advantage of using digital sensors is that you don't have to do calibration of your own, as sensors normally come factory-calibrated. Digital sensors often have a low power current consumption compared to their analog peers.

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.

The ZIG-SHT25 sensor example is available as `test-sht25.c`, an output example is given below:

```
Starting 'SHT25 test'
Temperature 23.71 °C
Humidity 42.95 %RH
Temperature 23.71 °C
Humidity 42.95 %RH
Temperature 23.71 °C
Humidity 42.95 %RH
Temperature 23.71 °C
Humidity 42.98 %RH
```

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!).

Chapter 10. Sending Data to Ubidots:

The objective of this practice is to familiarize with an IoT cloud platform and extend our Wireless Sensor Network to Internet, but before we cover the IP-based wireless communication, let's have some fun putting into practice the previously learned concepts and share our work.

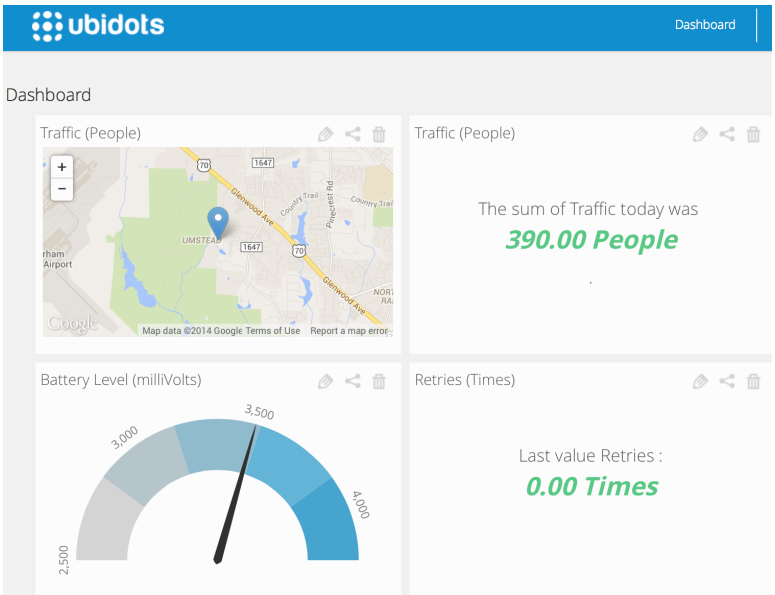
10.1. What is Ubidots

Ubidots is a cloud service to capture and make sense of sensor data, enabling both makers and professionals to easily deploy its application and connect things to the Internet.

The Dashboard feature allows to visualize the gathered data in a *human* way, enable to pre-process data (i.e convert from Celsius to Farenheit), as well as create events based on specific triggers, such as threshold limits, enabling notifications over email or SMS.

More information here:

http://www.ubidots.com

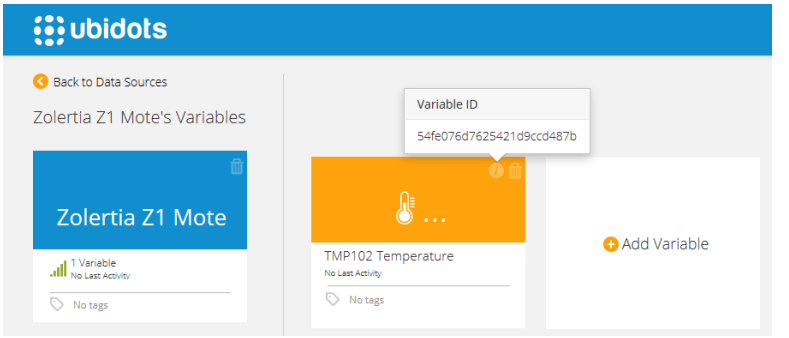


10.2. Get the API key and create your variables

The first step to start working is to register (don't worry is free!), click on `Sign Up`, and then at `My Profile` in the `API Keys tab` take note of your API Key or just create a `short Token`, which won't expire.

The next step is to create the Data Source, for this practice we will send Temperature readings from the built-in **TMP102 sensor**, so let's create a **Zolertia Z1 data source**. Click on `Sources` then click on `Add data Source` and select a Generic device. After naming and creating the source, let's add the temperature variable.

Click on the Data Source then click on `Add Variable`, after creating the variable click on the information button as shown below and take note of the Variable ID, it will be used to write and read data from the Variable.



10.3. Send data to Ubidots over the serial port

For this first example we need two software components: an application running on the Z1 mote sending data over the serial port, and a server-side application receiving, parsing and relaying the data to Ubidots.

We will create a simple Contiki application that handles the serial formatting to be sent to the server-side app (a python script with the Ubidots library). Contiki Apps provide extra features that can be used directly by other applications. Apps are placed in the apps folder of Contiki.

The Makefile of a Contiki App has the following naming convention:

```
Makefile.serial-ubidots
```

And inside you must specify which are the source codes that will be used, in this case:

```
serial-ubidots_src = serial-ubidots.c
```

This is what the serial-ubidots serial will look like:

```
#include "contiki.h"
#include <string.h>
#include "serial-ubidots.h"
void
send_to_ubidots(const char *api, const char *id, uint16_t *val)
{
  uint8_t i;
  unsigned char buf[6];
  printf("\r\n%s", api);
  printf("\t%s", id);
  for (i=0; i<UBIDOTS_MSG_16B_NUM; i++) {
    snprintf(buf, 6, "%04d", val[i]);
    printf("\t%s", buf);
  }
  printf("\r\n");
}
```

You need to declare a header file `serial-ubidots.h` as well:

```
/*-------------------------------------------------*/
#define UBIDOTS_MSG_APILEN 40
#define UBIDOTS_MSG_KEYLEN 24
#define UBIDOTS_MSG_16B_NUM 1
#define UBIDOTS_MSG_LEN (UBIDOTS_MSG_KEYLEN + UBIDOTS_MSG_APILEN + \
                         (UBIDOTS_MSG_16B_NUM*2) + 2)
/*-------------------------------------------------*/
struct ubidots_msg_t {
  uint8_t id;
  uint8_t len;
  uint16_t value[UBIDOTS_MSG_16B_NUM];
#ifdef UBIDOTS_MSG_APILEN
  char api_key[UBIDOTS_MSG_APILEN];
#endif
#ifdef UBIDOTS_MSG_KEYLEN
  char var_key[UBIDOTS_MSG_KEYLEN];
#endif
};
/*-------------------------------------------------*/
void send_to_ubidots(const char *api, const char *id, uint16_t *val);
/*-------------------------------------------------*/
```

We have also created a data structure which will simplify sending this data over a wireless link, we will talk about this a bit later.

Now that we have created this Contiki App, we should add it to our example code (that sends temperature to Ubidots), edit the `Makefile` at `examples/z1` and add serial-ubidots to the APPS argument:

```
APPS = serial-shell serial-ubidots
```

And now let's edit the `test-tmp102.c` example to include the serial-ubidots application, first add the serial-ubidots header as follows:

```
#include "serial-ubidots.h"
```

Then we should create 2 new constants with the API key and Variable ID, obtained at Ubidots site as follows:

```
static const char api_key[] = "XXXX";
static const char var_key[] = "XXXX";
```

It is a general good practice to declare constants values with as **const**, this will save some valuable bytes for the RAM memory. Change the polling interval to avoid flooding Ubidots.

```
#define TMP102_READ_INTERVAL (CLOCK_SECOND * 15)
```

Then we are ready to send our data to Ubidots, first change the call to the tmp102 sensor to have the value with 2 digits precision, and send it over to Ubidots, replace as follows:

```
PRINTFDEBUG("Reading Temp...\n");
raw = tmp102_read_temp_x100();
send_to_ubidots(api_key, var_key, &raw);
```

Upload the code to the Z1:

```
make test-tmp102.upload && make z1-reset && make login
```

This is what you will see on the screen (mockup values):

```
 kjfdkjg455g4jh54g5jh4g5jh4g54jhg54jh55jj
```

```
 545jh45jh5jh456jh546jh45    2718
```

Notice that you must divide by 100 to get the 27.18 ºC degree value, this can be done easily on Ubidots.

10.4. Ubidots Python API Client

The Ubidots Python API Client makes calls to the Ubidots API. The module is available on PyPI as *ubidots*. To follow this quickstart you'll need to have python 2.7 in your machine, available at http://www.python.org/download/.

You can install pip in Linux and Mac using this command:

```
$ sudo easy_install pip
```

Ubidots for python is available in PyPI and you can install it from the command line:

```
$ sudo pip install ubidots==1.6.1
```

Now let's create a Python script on the PC and name it `UbidotsPython.py`. Below is a simple code snippet that gets the job done (no error checking, just a sample code!):

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------------------------#
# Simple application to relay data to Ubidots from a Contiki serial-based conn
# ------------------------------------------------------------------------------#
import serial
from time import sleep
from ubidots import ApiClient

# Use as default
PORT = "/dev/ttyUSB0"

# ------------------------------------------------------------------------------#
# Create a serial object and connect to mote over USB
# ------------------------------------------------------------------------------#
def connectMote(port):
  try:
    ser = serial.Serial(port, 115200,timeout=0, parity=serial.PARITY_NONE,
                        stopbits=serial.STOPBITS_ONE, bytesize=serial.EIGHTBITS)
  except:
    sys.exit("Error connecting to the USB device, aborting")
```

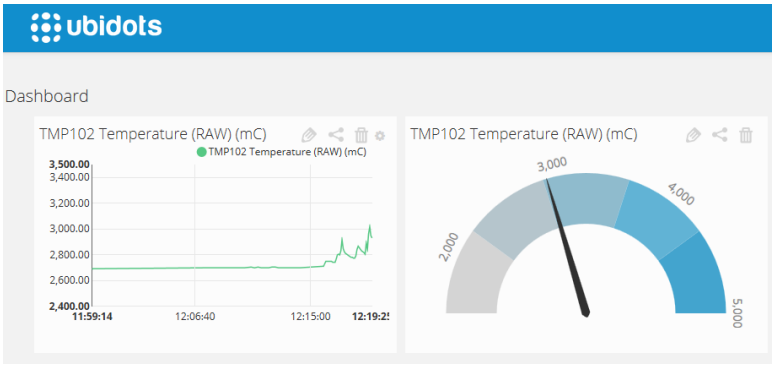
```
  return ser

# ------------------------------------------------------------------------------#
# Parse the serial data and publish to Ubidots, this assumes the following:
# \r\n API_KEY \t VAR_KEY \t VALUE \r\n
# ------------------------------------------------------------------------------#
def process_data(raw):
  # Search for start and end of frame and slice, discard incomplete
  if "\r\n" in raw:
    raw = raw[(raw.index("\r\n") + 2):]
    if "\r\n" in raw:
      raw = raw[:raw.index("\r\n")]
      # We should have a full frame, parse based on tab and create a list
      ubidots_raw = raw.split("\t")
      # Create a Ubidots client and get a pointer to the variable
      client = ApiClient(ubidots_raw[0])
      try:
        my_variable = client.get_variable(ubidots_raw[1])
      except Exception, e:
        print "Ubidots error: %s" % e
        return
      # Update the variable
      my_variable.save_value({'value':int(ubidots_raw[2])})

# ------------------------------------------------------------------------------#
# MAIN APP
# ------------------------------------------------------------------------------#
if __name__=='__main__':

  # Create the serial object and connect to the mode
  # Do not check the serial port object as the function already does it
  s = connectMote(PORT)

  # Loop forever and wait for serial data
  while True:
    queue = s.inWaiting()
    if queue > 0:
      data = s.read(1000)
      process_data(data)
    sleep(0.2)
```

The data is sent to Ubidots as long as the pyhton script is running. You can have it working in the background by adding `&` at the end of the script call.

While the python script is running, you cannot program the node as the resource is allocated, **close the Python script before doing so!**.

As the temperature sensor is located next to the USB connector, it tends to heat up. A realistic value is few degrees lower than the measured on. To get more reliable temperature measurements while connected to the USB, use an external temperature sensor.

To divide the incoming raw data by 100, you should name it as derived variable as follows: create a derived Variable by dividing our current temperature variable by 100.



Chapter 11. Wireless with Contiki:

This section goes as follows:

- Set up the Node ID and MAC address of the Z1 mote.
- Simple application: UDP broadcast.
- Simple application: UDP Server and client.
- Check mote to mote communication.
- Check ETX, LQI, RSSI.
- Change the Radio Frequency Channel and PAN ID.
- Debug: use a Packet sniffer and Wireshark.
- RSSI scanner example application.

Chapter 12. Set up the Node ID and MAC address of the Z1 mote.

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).

You can program and store to flash your own.

> ## Note
>
> Since commit `277dc8e1743cdcb253b13861044560464371e1c2` if you don't have stored a Node ID value in flash memory, upon programing the Z1 mote the Product Number (the one displayed earlier over the **make z1-motelist** command) is used instead.
>
> Newer Z1 motes have this number already flashed in the flash memory from factory.

Let's use the ID from the motelist:

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

The node ID should be 3301 (decimal) if not 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 node's addresses the mote should have will be :

```
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 c:e5 is the hex value corresponding to 3301. The global address is only set when an IPv6 prefix is assigned (more about this later).

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

Go to `examples/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
```

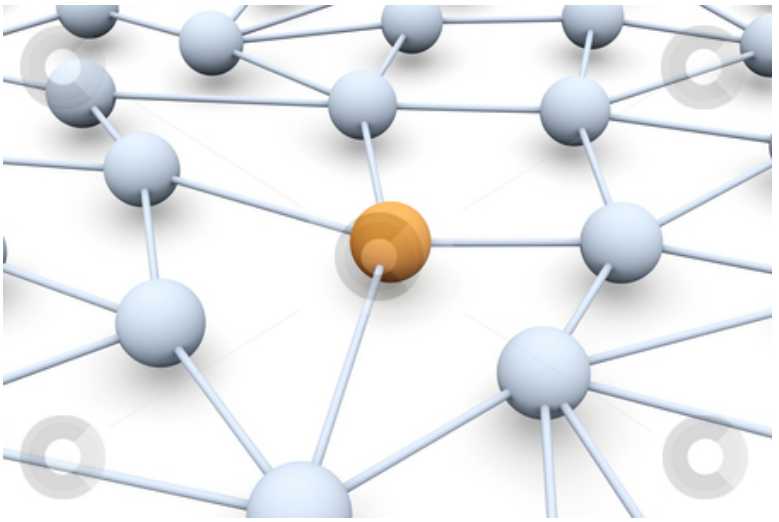
As you can see, now the node ID has been changed to 158, when you restart the mote you should now see that the changes are applied:

```
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
```

Chapter 13. UDP Broadcast

In this example, we will show how nodes can send data over the air and get to know the basics of Contiki IPv6/RPL implementation. This example contains a simple Contiki application that randomly broadcasts a UDP packet to its neighbors

We will use a simple version of UDP called simple-UDP.



UDP uses a simple connectionless transmission model with a minimum of protocol mechanism. It has no handshaking dialogues, and thus exposes any unreliability of the underlying network protocol to the user's program.

There is no guarantee of delivery, ordering, or duplicate protection. UDP is suitable for purposes where error checking and correction is either not necessary or it is performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.

Wireless sensor networks often use UDP because it is lighter and there are less transactions (which can be translated in less energy consumption). A protocols using UDP is COAP (see later).

Go to:

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

And 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 IP 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 to store the UDP connection information and mapped callback in which to process any received message. It is initialized below 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, theres a library which already 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=z1 savetarget
```

Now clean any previous compiled code, compile, upload your code and then restart the z1 mote, and print the serial output to screen (all in one command!):

```
make clean && make broadcast-example.upload && make z1-reset && make login
```

Upload this code to at least 2 motes and send/receive messages from neighbors. If you have more than 1 Z1 Mote connected in your PC, remember to use the MOTES=/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: replace the *Test* string with your group's name and try to identify others. Also write down the node ID of other motes. This will be useful for later.

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

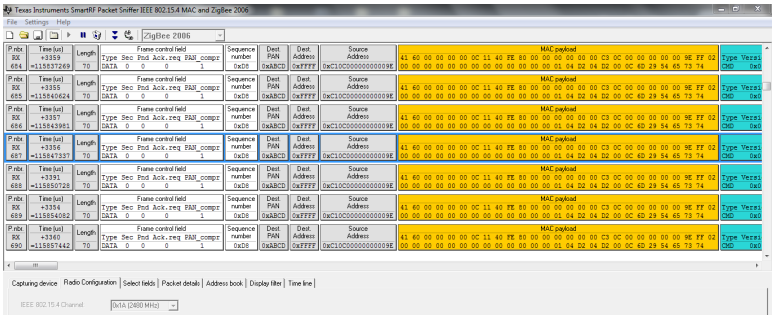
```
#define SEND_INTERVAL  (20 * CLOCK_SECOND)
#define SEND_TIME      (random_rand() % (SEND_INTERVAL))
```

Chapter 14. Setting up a sniffer

One of the must-have tools when developing wireless applications is a sniffer, which is basically a promiscuous wireless interface able to capture data and decode into a human-readable format.

A packet sniffer is a must-have tool for any wireless network application, a sniffer allows to actually see what are you 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 (http://www.ti.com/tool/packet-sniffer), which can be executed using a CC2531 USB dongle (http://www.ti.com/tool/CC2531EMK) which allows capturing outgoing packets like the one below.



For the remainder of this practice we will use Wireshark as our Packet analyzer, and we will learn about Open Source sniffers available.

14.1. Short intro to Wireshark

This example uses Wireshark to capture and examine a packet trace. More information and installation instructions are available at:

https://www.wireshark.org/

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 UI 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. An 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.

When you open Wireshark, there's a couple of toolbars at the top, an area called Filter, and a few boxes below in the main window. 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.

Lastly there is the Capture section. This will 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.

A hands-on session using a Z1 mote as a sniffer will help using Wireshark.

14.2. SenSniff IEEE 802.15.4 wireless sniffer

We will use for this practice the **SenSniff** application, freely available at: https://github.com/g-oikonomou/sensniff

Paired with a Z1 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 Z1 mote as a packet Sniffer go to the following location:

```
cd examples/z1/sniffer
```

In the `project-conf.h` select the channel to sniff, by changing the `RF_CHANNEL` and `CC2420_CONF_CHANNEL` definitions. 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.

Compile and program:

```
make sniffer.upload
```

Do not open a login session because the sniffer application uses the serial port to send its findings to the sensniff python script. 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:

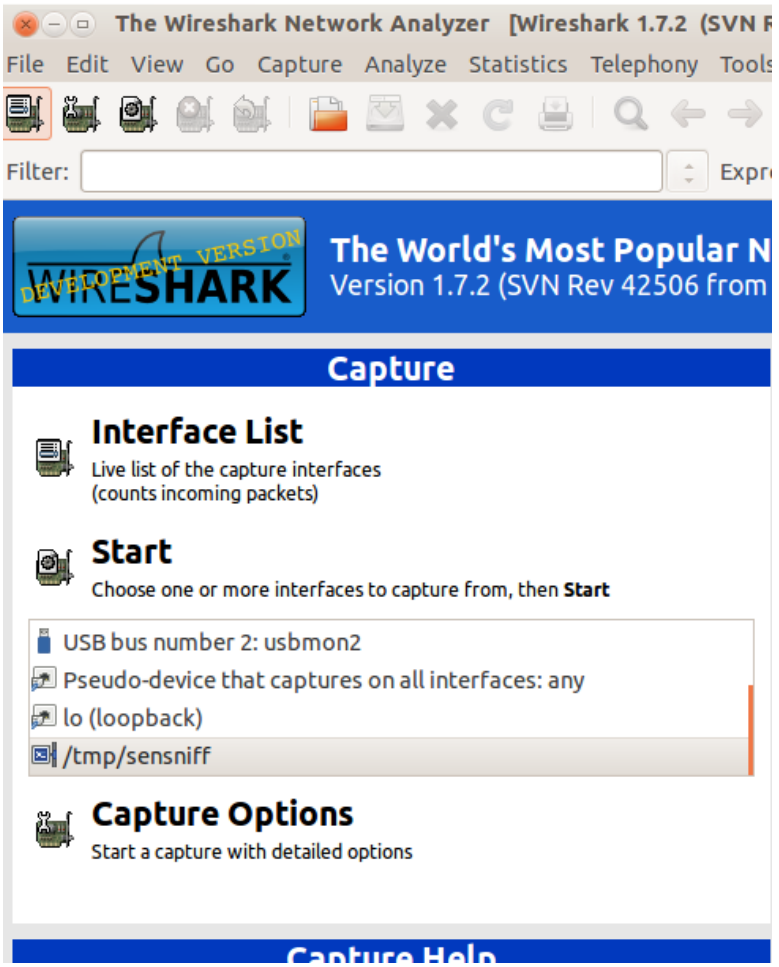
```
"Remote end not reading"
```

Which is not worrysome, it only means that the other pipe endpoint is not connected. You can also save the sniffed frames to open later 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 where to store the file accordingly.

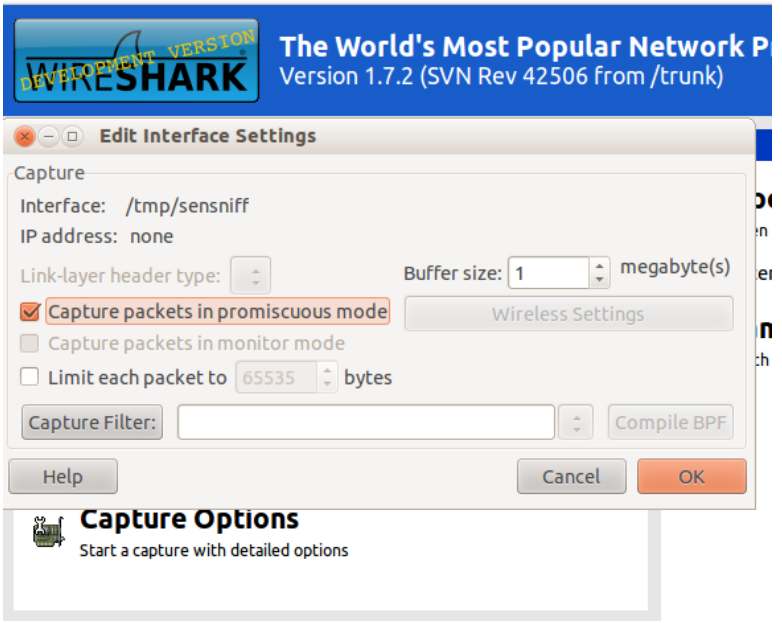
Open another terminal and launch wireshark with the following command, which will add the pipe as a capture interface:

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

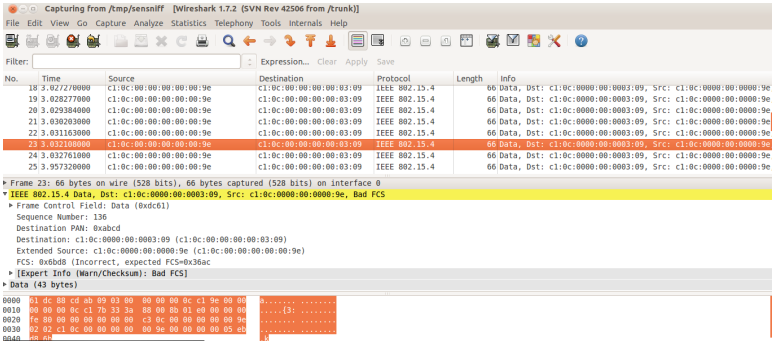
Select the `/tmp/sensniff` interface from the droplist and click `Start` just above.



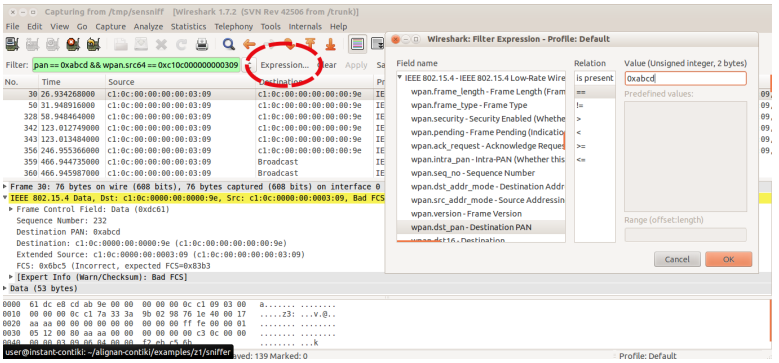
Make sure that the pipe is configured to capture packets in promiscuous mode, if needed you can increase the buffer size, but 1MB is normally enough.



Now the captured frames should start to appear on screen.



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 until 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.



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

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

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

14.3. Foren6

Another must-to-have tool for analyzing and debugging 6loWPAN/IPv6 networks is Foren6 http://cetic.github.io/foren6/, It uses a passive sniffer devices to reconstruct a visual and textual representation of network information, with a friendly graphical user interface and customizable layout, and also allows to rewind the packet capture history and replay a previous packet trace.

To install follow the instructions at http://cetic.github.io/foren6/install.html

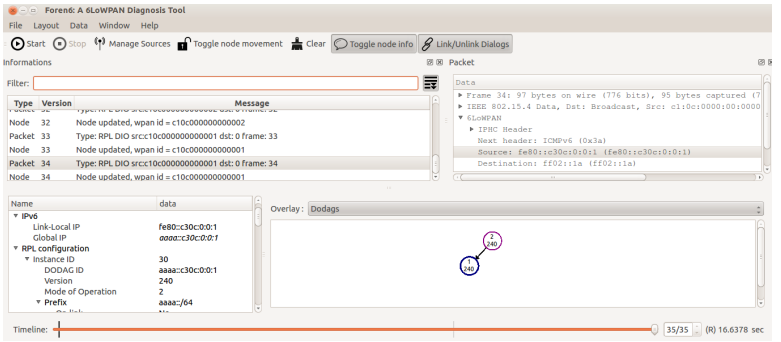
To program a Z1 mote as sniffer:

```
git clone https://github.com/cetic/Contiki
cd Contiki
git checkout sniffer
cd examples/sniffer
make TARGET=z1.upload
```

To connect to Foren6, a basic step-by-step guide for the Z1 mote is available at the link below:

http://cetic.github.io/foren6/example2.html

Open the `Manage Sources` dialog by clicking the Manage Sources button in the Toolbar or from the `File` menu, then click `Start` and visualize your network.


Foren6: A 6LoWPAN Diagnosis Tool
File Layout Data Window Help
Start Stop Manage Sources Toggle node movement Clear Toggle node info Link/Unlink Dialogs
Informations
Filter:
Type Version Message
Node 32 Node updated, wpan id = c10c000000000002
Packet 33 Type: RPL DIO src:c10c000000000001 dst: 0 frame: 33
Node 33 Node updated, wpan id = c10c000000000001
Packet 34 Type: RPL DIO src:c10c000000000001 dst: 0 frame: 34
Node 34 Node updated, wpan id = c10c000000000001
Packet
Data
Frame 34: 97 bytes on wire (776 bits), 95 bytes captured (7
IEEE 802.15.4 Data, Dst: Broadcast, Src: c1:0c:0000:00:0000
6LoWPAN
IPHC Header
Next header: ICMPv6 (0x3a)
Source: fe80::c30c:0:0:1 (fe80::c30c:0:0:1)
Destination: ff02::1a (ff02::1a)
Name data
IPv6
Link-Local IP fe80::c30c:0:0:1
Global IP aaaa::c30c:0:0:1
RPL configuration
Instance ID 30
DODAG ID aaaa::c30c:0:0:1
Version 240
Mode of Operation 2
Prefix aaaa::/64
Overlay : Dodags
2
240
1
240
Timeline:
35/35
(R) 16.6378 sec

Chapter 15. Simple application: UDP Server and client

Normal UDP or TCP transactions require a server-client model, in which the communication is made using a socket, which is an IP address and a port number. What we will do in this example is to forward to the receiver connected to a PC (via USB) temperature sensor data to be published to Ubidots.

> You will need two nodes. The one sending the temperature data is the server, while the one connected to the PC via USB is the client.

This example relies on a service ID, which allows registering, disseminating, and looking up services. A service is identified by an 8-bit integer between 1 and 255. Integers below 128 are reserved for system services. When setting up the example, we need to decide a service ID for the temperature data. The advantage is that the servers (senders of data) don't need to know the address of the receiver. It is a subscription model where we only need to agree on the service number ID.

We have three groups:

Group 1 hosts the client that received the data from Groups 2 and 3.

Group 2 and 3 are the servers that transmit data. Group 2 sends temperature data and has service ID number 190. Group 3 sends acceleration data and has service ID number 191.

Server side:

Open `examples/ipv6/simple-udp-rpl/unicast-sender.c`

First we are going to add

```
#include "serial-ubidots.h"
#include "dev/i2cmaster.h"
```

Group 2:

```
#include "dev/tmp102.h"
```

```
#define SERVICE_ID 190
#define UDP_PORT 1234
```

Group 3:

```
#include "dev/adxl345.h"
#define SERVICE_ID 190
#define UDP_PORT 5678
```

Change the poll rate to something faster:

```
#define SEND_INTERVAL  (15 * CLOCK_SECOND)
```

We have declared a structure at `apps/serial-ubidots.h` to store the Variable ID and data to be pushed to Ubidots, this will be helpful when sending data wirelessly to the receiver. This is already declared at `serial-ubidots.h`, do not add this to the example.

```
struct ubidots_msg_t {
  char var_key[VAR_LEN];
  uint8_t value[2];
};
```

Declare a structure in our code and a pointer to this structure as below:

```
static struct ubidots_msg_t msg;
static struct ubidots_msg_t *msgPtr = &msg;
```

These structures are used to send Ubidots specific information.

In this application we are going to use global IPv6 addresses besides the link-local ones, the function set_global_address initializes our IPv6 address with the prefix aaaa::, and generates also the link local addressing based on the MAC address.

```
static void
set_global_address(void)
{
  uip_ipaddr_t ipaddr;
  int i;
  uint8_t state;
  /* Initialize the IPv6 address as below */
  uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
```

```
  /* Set the last 64 bits of an IP address based on the MAC address */
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  /* Add to our list addresses */
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
  printf("IPv6 addresses: ");
  for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
    state = uip_ds6_if.addr_list[i].state;
    if(uip_ds6_if.addr_list[i].isused &&
      (state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
      uip_debug_ipaddr_print(&uip_ds6_if.addr_list[i].ipaddr);
      printf("\n");
    }
  }
}
```

Now inside the `PROCESS_THREAD(unicast_sender_process, ev, data)`, right after the `set_global_address()` call, we initialize our sensors:

Group 2:

```
int16_t temp;
tmp102_init();
```

Group 3:

```
accm_init();
```

And we pass our variable ID obtained at Ubidots to the ubidots message structure as follows:

```
memcpy(msg.var_key, "545a202b76254223b5ffa65f", VAR_LEN);
printf("VAR %s\n", msg.var_key);
```

This function returns the address of the node offering a specific service. If the service is not known, the function returns *NULL*. If there is more than one node offering the service, this function returns the address of the node that announced its service most recently.

```
addr = servreg_hack_lookup(SERVICE_ID);
```

If we have the receiver node in our services list, then we take a measure from the sensor, pack it into the byte buffer, and send the information to the receiver node by passing the structure as an array using the pointer to the structure, specifying the size in bytes.

The `UBIDOTS_MSG_LEN` is the sum of the Variable ID string length (24 bytes) plus the sensor reading size (2 bytes).

Replace the existing `if (addr != NULL)` block with the following:

Group 2:

```
if (addr != NULL) {
    temp = tmp102_read_temp_x100();
    msg.value[0] = (uint8_t)((temp & 0xFF00) >> 8);
    msg.value[1] = (uint8_t)(temp & 0x00FF);
    printf("Sending temperature reading -> %d via unicast to ", temp);
    uip_debug_ipaddr_print(addr);
    printf("\n");
    simple_udp_sendto(&unicast_connection, msgPtr, UBIDOTS_MSG_LEN, addr);
} else {
    printf("Service %d not found\n", SERVICE_ID);
}
```

Group 3:

Replace inside the `if (addr != NULL)` conditional with the following:

```
msg.value[0] = accm_read_axis(X_AXIS);
msg.value[1] = accm_read_axis(Y_AXIS);
printf("Sending temperature reading -> %d via unicast to ", temp);
uip_debug_ipaddr_print(addr);
printf("\n");
simple_udp_sendto(&unicast_connection, msgPtr, UBIDOTS_MSG_LEN, addr);
```

And finally add the serial-ubidots app to our `Makefile`:

```
APPS = servreg-hack serial-ubidots
```

If the address is NULL it can mean that the receiver node is not present yet.

```
connecting to /dev/ttyUSB0 (115200) [OK]
Rime started with address 193.12.0.0.0.0.3.9
MAC c1:0c:00:00:00:00:03:09 Ref ID: 255
Contiki-2.6-1796-ga50bc08 started. Node id is set to 377.
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:0309
```

```
Starting 'Unicast sender example process'
IPv6 addresses: aaaa::c30c:0:0:309
fe80::c30c:0:0:309
VAR 545a202b76254223b5ffa65f
Service 190 not found
```

Client side:

Open `examples/ipv6/simple-udp-rpl/unicast-receiver.c` and add the Ubidots app:

```
#include "serial-ubidots.h"
```

Add the services we are interested in, each to be received in a different UDP port:

```
#define SERVICE_ID 190
#define UDP_PORT_TEMP  1234
#define UDP_PORT_ACCEL 5678
```

You can delete the `SERVICE_ID`, `SEND_INTERVAL` and `SEND_TIME` definitions.

A quick RPL intro

RPL is on the IETF standards track for routing in low-power and lossy networks. The protocol is tree-oriented in the sense that one or more root nodes in a network may generate a topology that trickles downward to leaf nodes.

In each RPL instance multiple Directed Acyclic Graphs (DAGs) may exist, each having a different DAG root. A node may join multiple RPL instances, but must only belong to one DAG within each instance.

The receiver creates the RPL DAG and becomes the network root with the same prefix as the servers:

```
static void
create_rpl_dag(uip_ipaddr_t *ipaddr)
{
  struct uip_ds6_addr *root_if;
```

```
  root_if = uip_ds6_addr_lookup(ipaddr);
  if(root_if != NULL) {
  rpl_dag_t *dag;
  uip_ipaddr_t prefix;

  rpl_set_root(RPL_DEFAULT_INSTANCE, ipaddr);
  dag = rpl_get_any_dag();
  uip_ip6addr(&prefix, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
  rpl_set_prefix(dag, &prefix, 64);
  PRINTF("created a new RPL dag\n");
  } else {
  PRINTF("failed to create a new RPL DAG\n");
  }
}
```

We should now subscribe to both services (temperature and acceleration), let's replace the `simple_udp_register` call inside the `PROCESS_THREAD` block, after the `servreg_hack_register(…)` call with the following:

```
simple_udp_register(&unicast_connection, UDP_PORT_TEMP,
                    NULL, UDP_PORT_TEMP, receiver);
simple_udp_register(&unicast_connection, UDP_PORT_ACCEL,
                    NULL, UDP_PORT_ACCEL, receiver);
```

And at the receiver callback, replace 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)
{
  char var_key[VAR_LEN];
  int16_t value;
  printf("Data received from ");
  uip_debug_ipaddr_print(sender_addr);
  printf(" on port %d from port %d\n",
       receiver_port, sender_port);
  if ((receiver_port == UDP_PORT_TEMP) || (receiver_port == UDP_PORT_ACCEL)){
    /* Copy the data and send to ubidots, restore missing null termination char */
    memcpy(var_key, data, VAR_LEN);
    var_key[VAR_LEN] = "\0";
```

```
    value =  data[VAR_LEN] << 8;
    value += data[VAR_LEN + 1];
    printf("Variable -> %s : %d\n", var_key, value);
    send_to_ubidots("fd6c3eb63433221e0a6840633edb21f9ec398d6a", var_key, value);
  }
}
```

Once the sender and the receivers have started, the following messages are shown on the screen of the receiver:

```
Starting 'Unicast receiver example process'
IPv6 addresses: aaaa::c30c:0:0:2
fe80::c30c:0:0:2
Data received from aaaa::c30c:0:0:309 on port 1234 from port 1234
Variable -> 545a202b76254223b5ffa65f : 2712

fd6c3eb63433221e0a6840633edb21f9ec398d6a    545b43f776254256ebbef0a6   2712
```

15.1. IEEE 802.15.4 channels and PAN ID

The IEEE 802.15.4 standard is intended to abide to established radio frequency regulations and defines specific physical (PHY) layers according to the country's regulations, 2.4 GHz is available almost everywhere, but in the lower band some countries use 868 MHz while others use 915 MHz as unlicensed frequencies.

The Z1 motes operate on the unlicensed and worldwide available 2.4 GHz band, The modulation scheme used is Direct Sequence Spread Spectrum (DSSS) with up to 250 kbps data rate, allowing a wireless range of 50-100 meters.

A total of 16 channels are available in the 2.4 GHz band, numbered 11 to 26, each with a bandwidth of 2 MHz and a channel separation of 5 MHz. As other technologies also share this band, such as WiFi based on IEEE 802.11 and Bluetooth based on IEEE 802.15, we should strive to choose channels that are not currently used by other devices.

As shown above the channels 15, 20, 25 and 26 are not overlapping WiFi assigned channels, so typically most IEEE 802.15.4 based devices tend to operate on this frequencies. One handy tool to have is a spectrum analyser to scan the wireless medium, which shows the wireless activity on a given band. A spectrum analyzer will show you the received power at a certain frequency, so you will not know if the power comes from another node, a WiFi device or even a microwave oven! We can use the Z1 mote as a simple spectrum analyser, which sweeps across the list of supported channels and shows the current received power.

To install the spectrum analyser application in the Z1 mote go to the following directory:

```
user@instant-Contiki:~/Contiki$ cd examples/z1/rssi_scanner
```

Compile, upload and execute the Java application to visualize the received power across channels:

```
make rssi-scanner.upload && make viewrssi
```

The result are shown below.

You can change the default 26 radio channel in Contiki by changing or redefining the following defines: `RF_CHANNEL`

But, where are this constants declared? Let's use a handy command line utility that allows to search for files and content within files, most useful when you need to find a declaration, definition, a file in which an error/warning message is printed, etc. To find where this definition is used by the Z1 platform use this command:

```
user@instant-Contiki:~/Contiki/platform/z1$ grep -lr "RF_CHANNEL" .
```

Which gives the following result:

```
./Contiki-conf.h
```

Basically grep as used above uses the following arguments: `-lr` instructs the utility to search recursively through the directories for the required content between the quotes, from our current location (noted by the dot at the end of the command) traversing the directories structure.

The `platform/z1/Contiki-conf.h` shows the following information regarding the `RF_CHANNEL`

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

#ifndef CC2420_CONF_CHANNEL
#define CC2420_CONF_CHANNEL           26
```

```
#endif /* CC2420_CONF_CHANNEL */
```

So we could either change the channel value directly in this file, but this change would affect other applications that perhaps need to operate on a different channel, so we could just override the `RF_CHANNEL` instead by adding the following to our applications `Makefile`:

```
CFLAGS += -DRF_CHANNEL=26
```

Or adding the following argument at compilation time:

```
DEFINES=RF_CHANNEL=26
```

The `PAN ID` is a unique Personal Area Network identifier that distinguishes our network from others in the same channel, thus allowing to subdivide a given channel into sub-networks, each having its own network traffic. By default in Contiki and for the Z1 mote the `PAN ID` is defined as`0xABCD`.

Exercise: Search where the PAN_ID is declared (hint: it has the 0xABCD value) and change to something different, then use the Z1 Sniffer and Wireshark to check if the changes were applied. Keep in mind that for 2 devices to talk to each other, they must have the same PAN ID. You can also program the Z1 Sniffer and your test application on a channel other than 26.

15.2. ETX, LQI, RSSI.

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

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 event. 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.

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 CC2420 radio frequency transceiver 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). The example below shows how the Packet Reception Rate decreases as the LQI decreases.

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.

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 as mW or dBm) and their range of RSSI values (from 0 to RSSI_Max), in the case of the CC2420 radio frequency transceiver on the Z1 mote, the RSSI can range from 0 to -100, values close to 0 are related to good links and values close to -100 are closely related to a bad link, due to multiple factors such as distance, environmental, obstacles, interferences, etc. The image below shows how the Packet Reception Rate (PRR) dramatically decreases as the CC2420 RSSI values worsen.

To print the current channel, RSSI and LQI of the last received packet (the relevant attributes of the link between the node and the sender), we are going to revisit the `unicast-receiver.c` example; open the file and include the following:

```
#include "dev/cc2420/cc2420.h"
```

Add the following print statement in the receiver block. The external variables `cc2420_last_rssi` and `cc2420_last_correlation` (LQI) are updated on a new incoming packet, so it should match our received packet.

```
printf("CH: &u RSSI: %d LQI %u\n", cc2420_get_channel(), cc2420_last_rssi,
 cc2420_last_correlation);
```

We should see something like the following:

```
Data received from aaaa::c30c:0:0:309 on port 1234 from port 1234
CH: 26 RSSI: -27 LQI 105
```

```
Variable -> 545b43f776254256ebbef0a6 : 2650
```

Exercise: Z1 motes come in two models: one with an integrated antenna and another with an external antenna. The integrated antenna is a ceramic antenna from Yageo/Phycomp, connected to the CC2420. The external antenna can be connected via a u.FL connector. Try to move away from the receiver and check the received signal on your laptop. What is the max distance at which the trasmission is successful? What is the nominal value of RSSI at 50 m with line of sight? Build an application that blinks a green LED when the RSSI is above -55 and a red LED when the RSSI is lower than -55. Does changing the node height and orientation change the RSSI value? If you have one, test the RSSI with an external directional antenna.

Chapter 16. Intro 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 [6lowpan], 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 [6lo]. 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.

16.1. 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 (2.4 GHz, 915 MHz, and 868 MHz, 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.

16.2. 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:

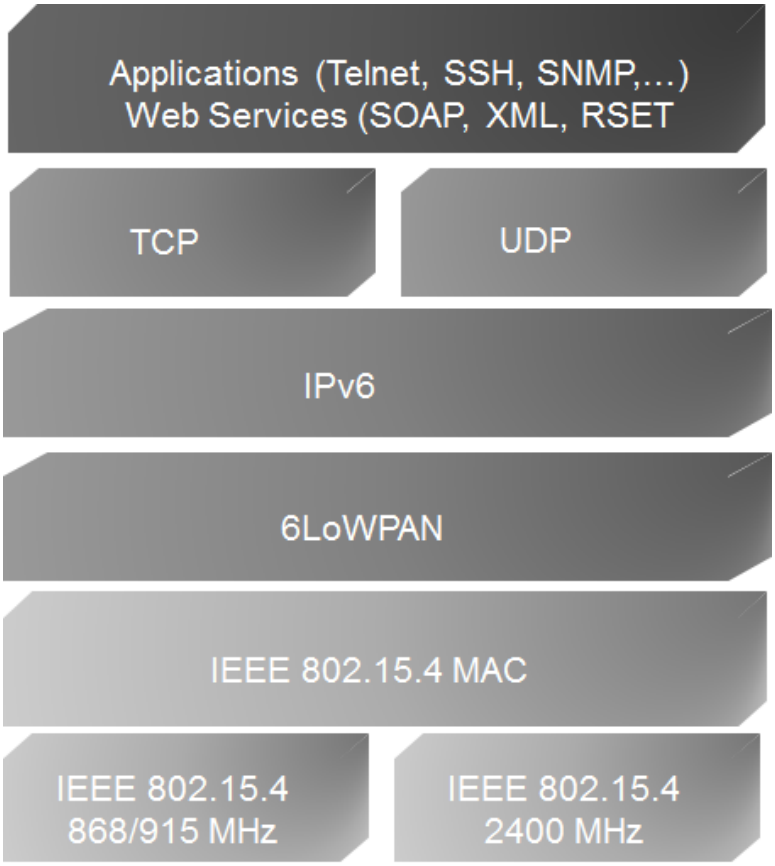
a. IP Connectivity: One of the characteristics of 6LoWPANs is the limited packet size, which implies that headers for IPv6 and layers above must be compressed whenever possible.
b. Topologies: LoWPANs must support various topologies including mesh and star: Mesh topologies imply multi-hop routing to 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.

16.3. 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.



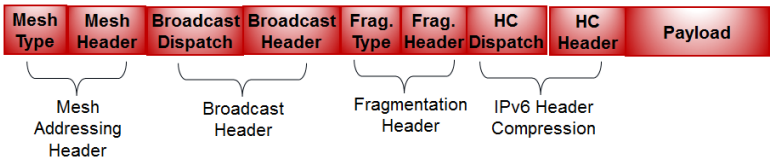
The main goals of 6lowpan are:

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:

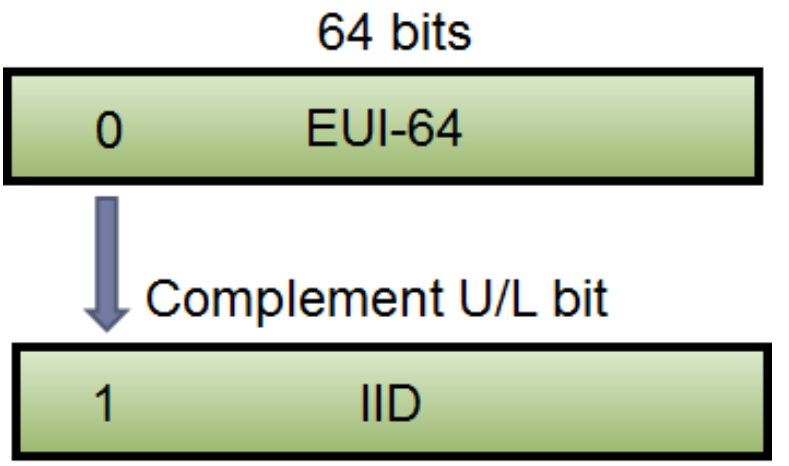
1. A Mesh Addressing header to support sub-IP forwarding.
2. A Fragmentation header to support the IPv6 minimum MTU requirement.
3. A Broadcast Header to be used when IPv6 multicast packets must be sent over the IEEE 802.15.4 network.
4. Stateless header compression for IPv6 datagrams to reduce the relatively 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.



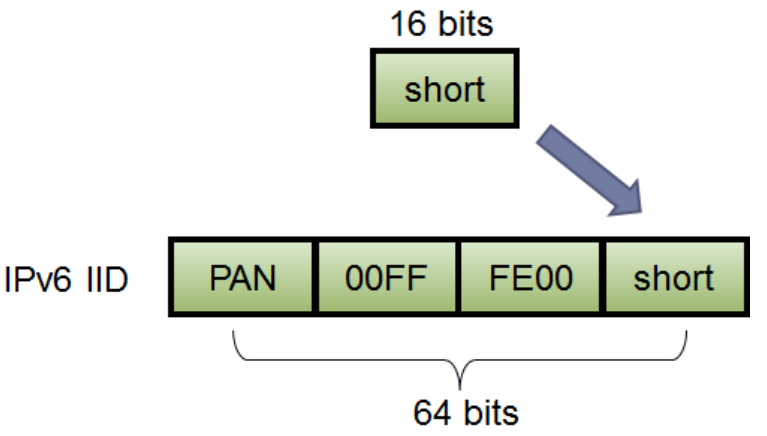
16.4. IPv6 Interface Identifier (IID)

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.



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.

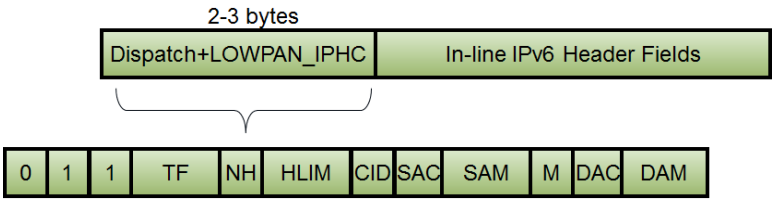


16.5. Header Compression

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.



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.

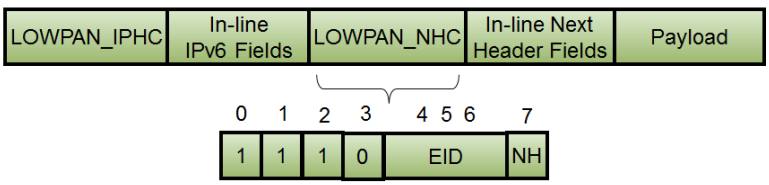
This additional octet identifies the pair of contexts to be used when the IPv6 source and/or destination address is compressed. The context identifier is 4 bits for each address, supporting up to 16 contexts. Context 0 is the default context. The two fields on the Context Identifier Extension are:

- 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.



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.

16.6. NDP optimization

IEEE 802.15.4 and other similar link technologies have limited or no usage of multicast signaling 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)

16.7. References

[6lo] 6lo IETF WG: http://datatracker.ietf.org/wg/6lo/charter/ [6lowpan] 6lowpan IETF WG: http://datatracker.ietf.org/wg/6lowpan/charter/ [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 [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 [roll] roll IETF WG: http://datatracker.ietf.org/wg/roll/charter/

Chapter 17. IoT Simulation (Cooja)

Cooja is the Contiki network simulator. Cooja allows large and small networks of Contiki motes to be simulated. Motes can 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.

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.

17.1. 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 is 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.

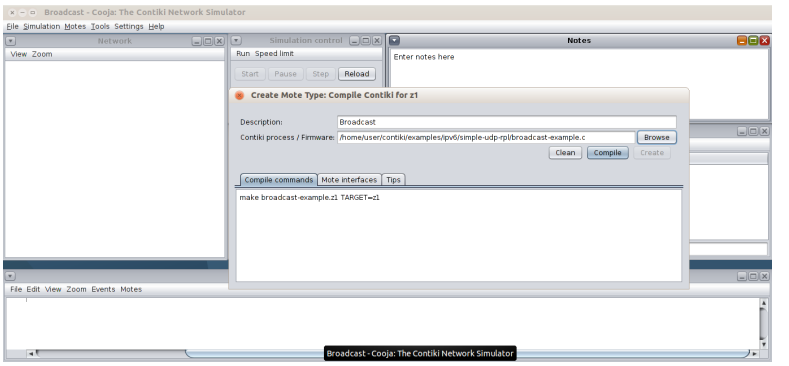
17.2. Add motes

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.

17.3. Revisiting broadcast-example in Cooja

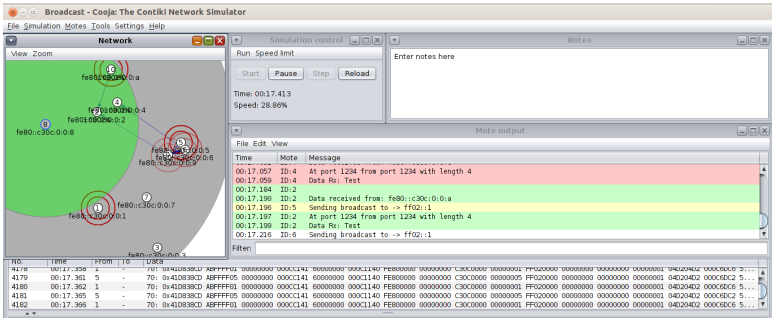
Setting up large networks on physical nodes can pose a challenge, in terms of instrumenting properly to measure all network elements and having enough equipment available, that's where **Cooja** comes handy.

Create a new simulation and define a new Z1 mote-based type of mote called `Broadcast`, do so by clicking on `Motes`, `Add Motes`, `Create new mote type` and select the `Z1 mote`. Use the example at `examples/ipv6/simple-udp-rpl/broadcast-example.c`



Press `Compile` and wait for the compilation to end, if there are no errors press `Create` and it will take you to the next screen. Now add 10 motes using `Random positioning`. Next in the `Network Panel`, click on `View` and add the following: `Radio Environment, Mote ID and addresses`. Now if you click on a Mote, it will display the Effective Radio coverage (green zone) and interference zone (grey zone) according to the selected medium model.

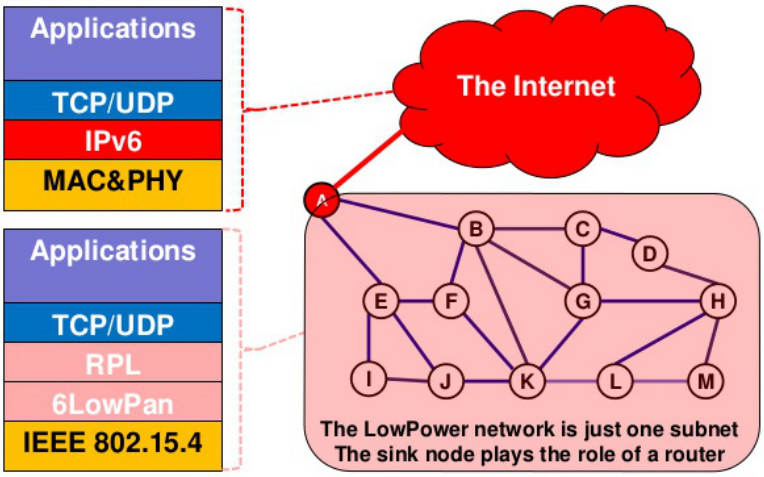
Be sure to also have the `Network, Mote Output and Radio Messages` panels in your layout, available at the `Tools` menu.



You now should be able to see the network traffic, the messages and the Motes console output, You can pause and start again the simulation to inspect the generated information at your own pace.

17.4. Routing Protocol for Low Power Networks (RPL)

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



It supports three traffic patterns: MP2P, point-to-multipoint (P2MP) and 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 a 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, 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. A 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 a 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.

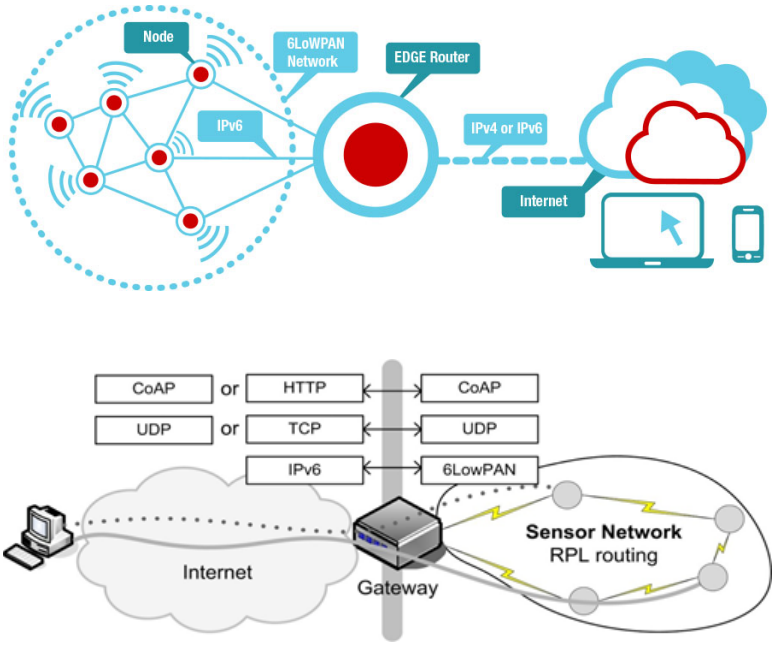
RPL has two mechanisms to repair the topology of the DODAG, one to to avoid looping and allow nodes to join/rejoin, and other called global repair. Global repair is initiated at the DODAG ROOT by incrementing the DODAG Version Number to create a new DODAG Version.

Exercise: Go to `core/net/rpl` and navigate through the C files, look for DEBUG defines and change its value to DEBUG_PRINT, this will print out to screen useful information allowing to better understand the RPL mechanics.

Chapter 18. Connecting our network to the world

We now want to go entirely IPv6, so that our nodes can be reached by the Internet!

But first let's test locally our IPv6-based network, it is often useful to debug any possible problem before going public. In the next section we will learn about setting up a Border Router and simulate a network using **cooja**.



18.1. 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.

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` (tools/tunslip6) over the serial port, creating a tunneled 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");
}
```

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 its network. To enable the web server, the WITH_WEBSERVER flag should be enabled, and by default it will add the httpd-simple.c application.

To compile and upload the border router to the Mote just type:

```
make TARGET=z1 border-router.upload
```

And to connect the border router to your host run:

```
make TARGET=z1 connect-router
```

By default it will try to connect to a mote at port `/dev/ttyUSB0` using the following serial settings: 115200 baudrate 8N1. If you do not specify an IPv6 prefix it will use the default `aaaa::1/64`, to specify a specific one run the tunslip tool instead using the following:

```
make TARGET=z1 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:

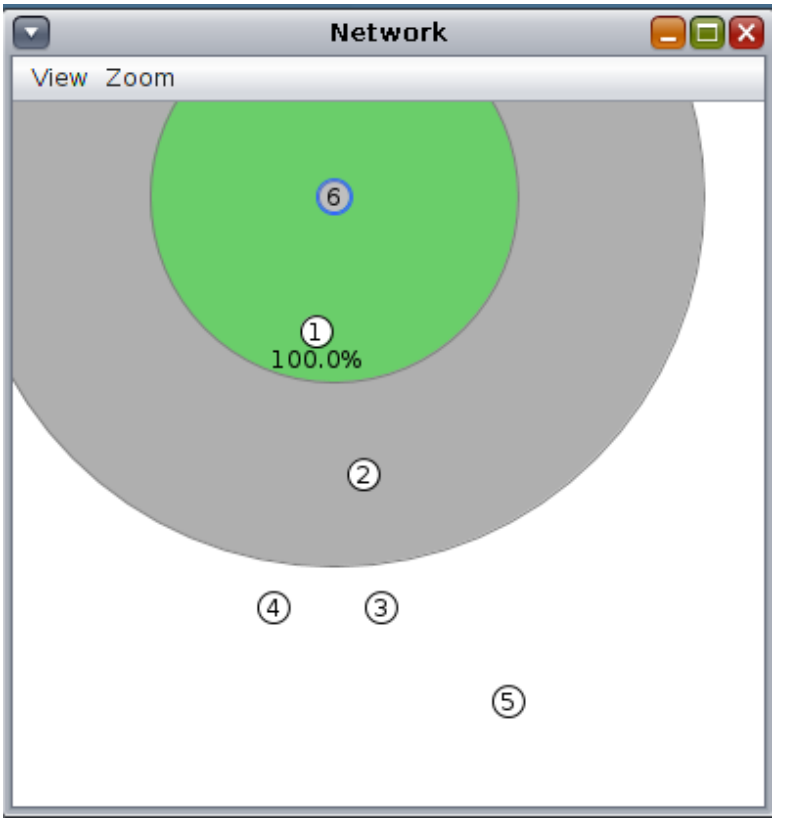
```
cd tools
cc tunslip6.c -o tunslip6
```

And to run with specific arguments, if you are required to use a Z1 mote connected to a specific serial port, or require to name your tunnel connection with a specific naming, or proxify to a given address and port. Run ./tunslip -H for more information.

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

The next example is going to be executed using Cooja, the main objective behind is to deploy a multi-hop network, reachable world-wide using IPv6, effectively connecting our simulation to real-word devices.

Open Cooja and load the file at `examples/z1/ipv6/z1-websense/example-z1-websense.csc`.



The first noticeable thing about the example is the topology at the Network panel, the **Node 6** corresponds to the **Border Router** Mote, which has only 1 neighbor in its wireless coverage area, and downwards we can see the Border Router being 4-hops apart from the Mote 5. The example contain a step-by-step guide in the Notes panel for you to also follow.

Motes 1 to 5 have been programed with an example called `z1-websense`, which is a small built-in webserver displaying a history chart with the battery and temperature readings of the Mote, accessible from our web browsers.

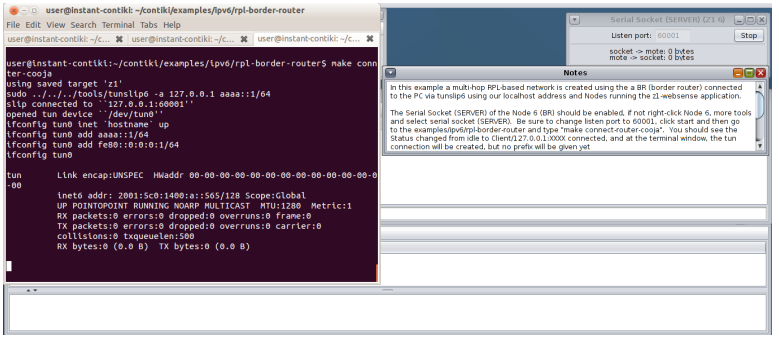
Now enable the Border Router (from now on BR) to connect over the serial socket to our host, right-click the BR and select the `Serial Socket (SERVER)` option from the `More tools` panel. This will allow us to connect to the simulated BR via the tunslip6 script, run the tool by typing:

```
sudo ./tunslip6 -a 127.0.0.1 -p 60001 aaaa::1/64
```

Or from the `examples/ipv6/rpl-border-router` location you can run instead:

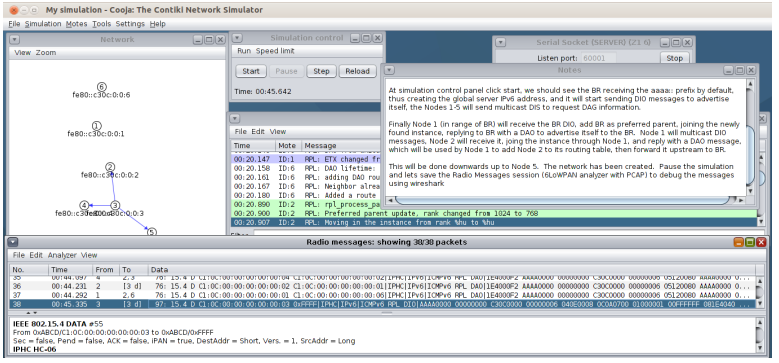
```
make connect-router-cooja
```

You can replace the IPv6 prefix with your own, the output is shown below. The connection status is shown in the Serial Socket panel in your Cooja layout, it should have changed from listening to Connected.



Notice the tunnel connection has been created but no prefix has been given yet. At the Simulation control panel click `Start`, we should see the BR receiving the `aaaa::` prefix by default, thus creating the global server IPv6 address, and it will start sending DIO messages to advertise itself, the Nodes 1-5 will send multicast DIS to request DAG information.

Finally Node 1 (in range of BR) will receive the BR DIO, add BR as preferred parent, joining the newly found instance, replying to BR with a DAO to advertise itself to the BR. Node 1 will multicast DIO messages, Node 2 will receive it, join the instance through Node 1, and reply with a DAO message, which will be used by Node 1 to add Node 2 to its routing table, then forward it upstream to BR.

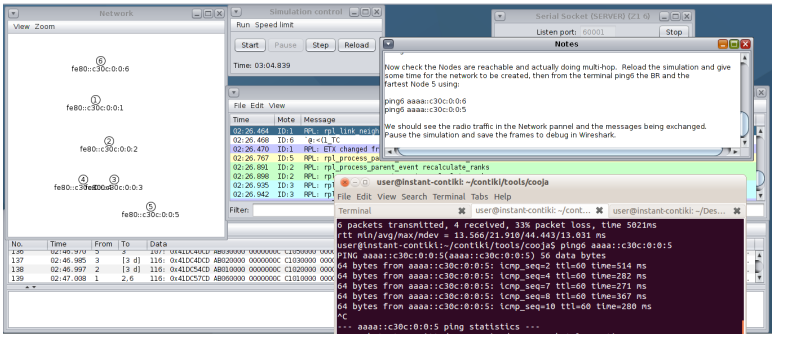


You can pause the simulation and examine the packets and console output at your own pace, simply click the Pause button at the Simulation Control panel. When you are done click Restart.

To check we have connectivity from our host to the simulated Motes, open a terminal console and try to ping the devices like follows:

```
ping6 aaaa::c30c:0:0:5
```

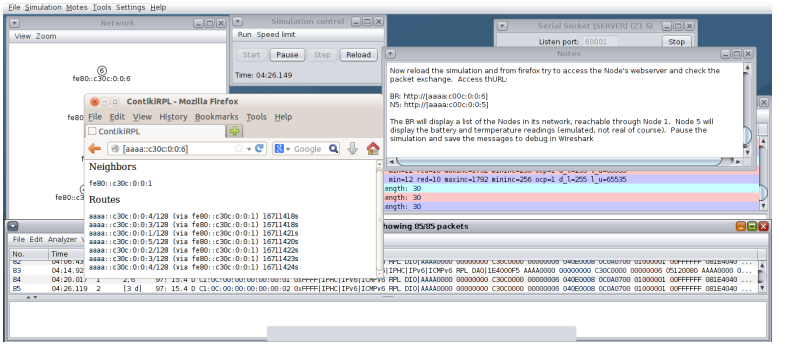
Replace the prefix with your own.



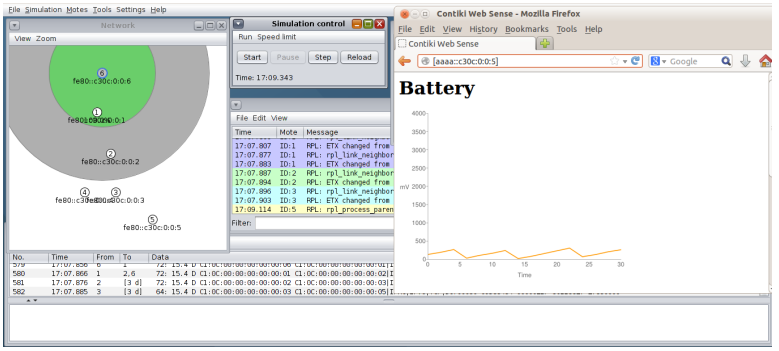
Now open a web browser (Firefox in our example) and type in the URL panel the BR IPv6 public address as follows:

```
http://[aaaa::c30c:0:0:6]
```

It will display the BR built-in webserver, showing the immediate neighbors and the known routes to the Motes in its instance. We can notice the next-hop neighbor to reach Motes 2 to 5 is Mote 1 as expected.



Now to access the webserver running on the nodes, type the global address of any of the Motes in the network. The battery and temperature reading diplayed are emulated values.



You can examine the packets being sent and received by going through the Radio Messages panel, click on the Analyzer menu and select the 6LoWPAN analyzer with PCAP to format the messages with PCAP to open later using wireshark, click on the File menu and Save To File.

Exercise: connect your simulation and browse the simulation results, try moving the motes out of range and see how the network heals itself.

18.2. Setting up IPv6 using gogo6.

In networking, a tunneling protocol enables new networking functions while still preserving the underlying network as it is. IPv6 tunneling enables IPv6 hosts and routers to connect with other IPv6 hosts and routers over the existing IPv4 Internet.

The main purpose of IPv6 tunneling is to deploy IPv6 as well as maintain compatibility with the existing base of IPv4 hosts and routers. IPv6 tunneling encapsulates IPv6 datagrams within IPv4 packets. The encapsulated packets travel across an IPv4 Internet until they reach their destination host or router. The IPv6-aware host or router decapsulates the IPv6 datagrams, forwarding them as needed.

Several tunnel brokers have been developed along with a Tunnel Setup Protocol (TSP). TSP allows IPv4 or IPv6 packets to be encapsulated and carried over IPv4, IPv6 or IPv4 NATs. TSP sets up the tunnel parameters between a user and a server. It handles authentication, encapsulation, IP address assignment and DNS functionality.

One of these TSP providers is gogo6, which provides the gogoCLIENT. The gogoCLIENT connected to the Freenet6 service provides IPv6 connectivity so you can test your v6 network, service or app.

To use the free gogo6 service, setup an account at http://www.gogo6.com/profile/gogoCLIENT

To set up the client on Ubuntu, first install the client with:

```
sudo apt-get install gogoc
```

Modify the config file from `/etc/gogoc/gogoc.conf` with

```
sudo nano -w /etc/gogoc/gogoc.conf.
```

Locate and modify the following lines:

```
userid= your_freenet6_id
passwd= your_password
server= broker.freenet6.net
```

Start the gogo client with

```
sudo /etc/init.d/gogoc start
```

And you are ready to go.

Try a ping to ipv6.google.com with

```
ping6 ipv6.google.com
```

If you can ping google via IPv6, you are ready to go!

You can use http://lg.as6453.net/bin/lg.cgi to check if your machine is visible from the Internet

18.3. Setting up IPv6 using Hurricane Electric

Hurricane Electric is another IPv6 service provider. Most tunnels use IPv4 protocol 41 encapsulation (6in4), where the data payload is just the IPv6 packet itself. Not all firewalls and NATs can properly pass protocol, so you will need to check this with your ISP (the fastest way is setting up a DMZ).

You will need to provide a public IPv4 address to create the tunnel, if you are behind a router or a firewall check if the public IP you are assigned is static or could be mapped to your local IP statically, else you will need other arrangements outside the scope of this guide.

Once you got this checked, register at Hurricane Electric (we will be using HE for our example), they have a lot of documentation available and certification programs if interested. You will get the following parameters after creating your tunnel:

```
IPv6 Tunnel Endpoints
Server IPv4 Address:216.66.XXX.XXXX
Server IPv6 Address:2001:470:XXXX:XXXX::1/64
Client IPv4 Address:213.151.XXX.XXX
Client IPv6 Address:2001:470:XXXX:XXXX::2/64
Available DNS Resolvers
Anycasted IPv6 Caching Nameserver:2001:470:20::2
Anycasted IPv4 Caching Nameserver:74.82.42.42
Routed IPv6 Prefixes
Routed /64:2001:470:XXXX:XXXX::/64
```

Next edit the following file and uncomment:

```
sudo nano gedit /etc/sysctl.conf
# net.ipv6.conf.all.forwarding=1
```

And use the following commands (you can copy and paste into a script to run at one):

```
modprobe ipv6
ip tunnel add he-ipv6 mode sit remote 216.66.XXX.XXX local 192.168.XXX.XXX ttl 255
ip link set he-ipv6 up
ip addr add 2001:470:XXXX:XXXX::2/64 dev he-ipv6
ip route add ::/0 dev he-ipv6
ip -f inet6 addr
```

Note: if you are behind a Firewall or a router providing DHCP access, you will need to use the LOCAL IP rather that the public one.

Check if everything is correct:

```
ifconfig he-ipv6
he-ipv6   Link encap:IPv6-in-IPv4
          inet6 addr: fe80::c0a8:461d/128 Scope:Link
          inet6 addr: 2001:470:XXXX:XXXX::2/64 Scope:Global
          UP POINTOPOINT RUNNING NOARP  MTU:1480  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:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

And now ping ipv6.google.com:

```
ping6 ipv6.google.com
PING ipv6.google.com(par03s02-in-x12.1e100.net) 56 data bytes
64 bytes from par03s02-in-x12.1e100.net: icmp_seq=1 ttl=56 time=37.4 ms
64 bytes from par03s02-in-x12.1e100.net: icmp_seq=2 ttl=56 time=37.8 ms
64 bytes from par03s02-in-x12.1e100.net: icmp_seq=3 ttl=56 time=37.9 ms
64 bytes from par03s02-in-x12.1e100.net: icmp_seq=4 ttl=56 time=37.3 ms

--- ipv6.google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3009ms
rtt min/avg/max/mdev = 37.383/37.662/37.964/0.271 ms
```

Chapter 19. IPv6 communication in Contiki and IoT/M2M protocols

The aim of this section is to introduce the reader to the most used protocols in IoT, already present in Contiki, most from a M2M background, now making its way to most common IoT applications and services.

This section goes as follow:

- Revisiting the Z1 Websense application on Z1 Motes.
- UDP communication between network and host.
- CoAP example and Firefox Copper plugin.
- RESTfull HTTP example with curl.
- MQTT example.

Chapter 20. Revisiting the Z1 Websense application on Z1 Motes.

We expect your machine to have an IPv6 address, either via the existing cabled network or via a tunnel (such as Gogo6 or Hurricane Electric or others).

In the past examples we have deployed and analyzed link-local networks only, now we will take a step further and connect our networks to external ones, allowing them to be reachable from outside and communicate globally to other networks using 6loWPAN and IPv6.

This small example will allow to add a working webserver to your IPv6 network in Contiki, this serves as a starting point to implement more complete applications based on the webserver functionalities.

As we did in previously, we will be working with the `examples/z1/ipv6/z1-websense` application. The sensors enabled for this test are temperature and battery.

Border-Router:

```
cd examples/ipv6/rpl-border-router/
make TARGET=z1 savetarget
make border-router.upload && make connect-router
```

Webserver

To compile and program a mote just execute:

```
cd examples/z1/ipv6/z1-websense
make z1-websense.upload
```

When the script finish uploading the image to the mote, verify the application is running properly and check the IPv6 address:

```
make z1-reset && make login
connecting to /dev/ttyUSB0 (115200) [OK]
Node id is not set, using Z1 product ID
Rime started with address 193.12.0.0.0.0.18.233
MAC c1:0c:00:00:00:00:12:e9 Ref ID: 4841
Contiki-2.6-2071-gc169b3e started. Node id is set to 4841.
```

```
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:12e9
Starting 'Sense Web Demo'
```

Then go to the border-router (see in section above how to access the border router via web browser) and verify the webserver is in the network:

```
Neighbors
fe80::c30c:0:0:12e9

Routes
aaaa::c30c:0:0:12e9/128 (via fe80::c30c:0:0:12e9)
```

We should be able to ping both the Border Router and the devices in the network.

Consult the webserver by browsing in your web browser http://[aaaa::c30c:0:0:12e9] and see the available readings.

Launch Wireshark and capture traffic of tun0 interface, browse the webserver and see what happens.

Chapter 21. UDP communication between network and host.

What is UDP?

UDP (User Datagram Protocol) is a communications protocol that offers a limited amount of service when messages are exchanged between 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

In this exercise we are going to send UDP messages from a Z1 mote in a network to an UDP Server running on the host, being UDP the simplest way to send short messages without the overhead TCP poses.

UDP client

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, 1);
```

To verify we have set the address correctly let's print the server address, at 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'
```

UDP Server

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

The `UDP6.py` script can be executed as a single-shot UDP client or as a UDP Server binded 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:

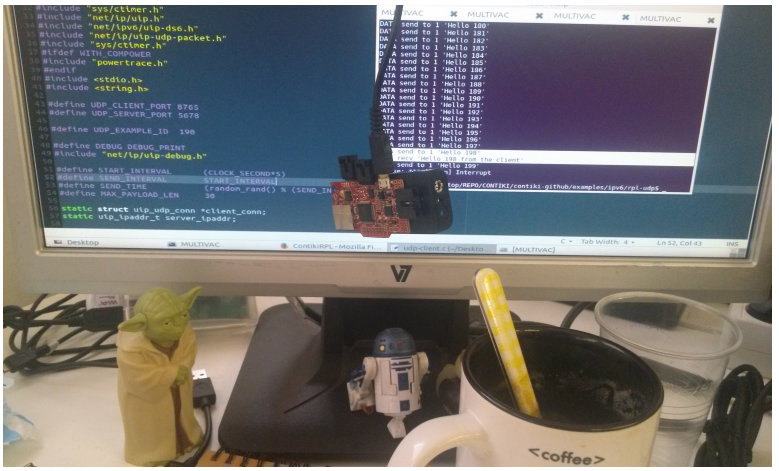
```
python UDP6.py -s 5678
```

This is the expected output when running and receiving an 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'
```



Chapter 22. CoAP example and Firefox Copper plug-in.

What is CoAP?

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 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.

Multicast, low overhead, and simplicity are extremely important for Internet of Things (IoT) and Machine-to-Machine (M2M) devices, which tend to be deeply embedded and have much less memory and power supply than traditional internet devices have. Therefore, efficiency is very important. 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. 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.

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 to Web browsers, allows an intuitive interaction and a presentation layer making easier to debug and interact with existing CoAP devices.

More information available at:

http://people.inf.ethz.ch/mkovatsc/copper.php

22.1. Preparing the setup

For this practice we will use 3 motes: a BR, server and client.

Ensure the 3 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
```

Be sure the settings are consistent, at the `examples/ipv6/rpl-border-router` in the `project-conf.h` file add the following:

```
#undef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC     nullrdc_driver

#undef NETSTACK_CONF_MAC
#define NETSTACK_CONF_MAC     nullmac_driver
```

Server:

```
cd examples/er-rest-example/
make TARGET=z1 savetarget
make er-example-server.upload && make z1-reset && make login
```

Annotate the IPv6 server address, disconnect the mote and connect another one to be used as client…

Client:

Remember you annotate the server address? edit the er-example-client.c and replace in this line:

```
SERVER_NODE(ipaddr)
 uip_ip6addr(ipaddr, 0xaaaa, 0, 0, 0, 0xc30c, 0x0000, 0x0000, 0x039c)
```

Then compile and flash the client:

```
make er-example-client.upload
```

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
```

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
```

And connect the client mote, assuming it is connected in /dev/ttyUSB2 port:

```
make login MOTES=/dev/ttyUSB2
../../tools/sky/serialdump-linux -b115200 /dev/ttyUSB2
connecting to /dev/ttyUSB2 (115200) [OK]
Rime started with address 193.12.0.0.0.0.0.158
MAC c1:0c:00:00:00:00:00:9e Contiki-2.5-release-681-gc5e9d68 started. Node id is set
 to 158.
CSMA nullrdc, channel check rate 128 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:009e
Starting 'COAP Client Example'
Press a button to request .well-known/core
--Requesting .well-known/core--
|</.well-known/core>;ct=40,</hello>;title="Hello world: ?len=0.."|;rt="Text",</
tes|t/push>;title="P| demo";obs,</tes|itle="Sub-resour||g|b, POST/PUT m|le="Red
 LED";rt=|BlockOutOfScope
--Done--
```

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 CC2420 transceiver:

```
coap://[aaaa::c30c:0000:0000:0001]:5683/sensor/radio?p=rssi
```

Do the same to get the battery 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 mv value would be:

V [mV] = (units * 5000)/4096

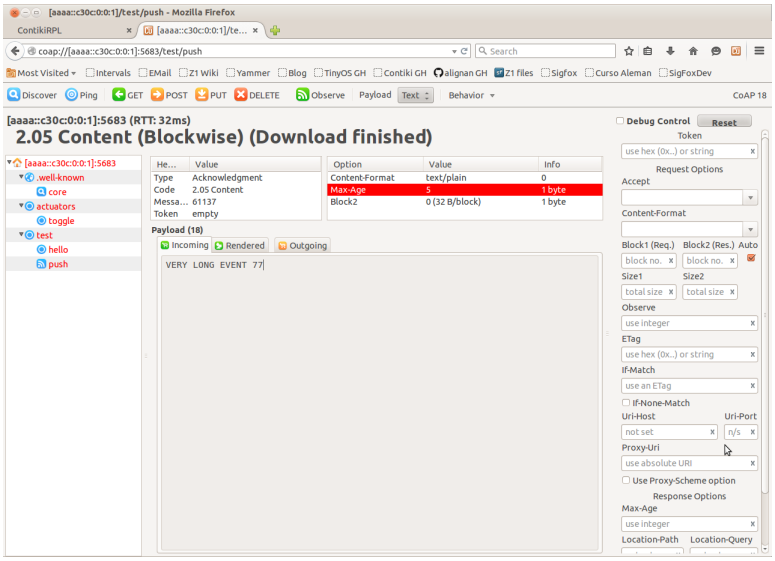
Let's say you want to turn the green LED ON, in the URL type:

```
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).


[aaaa::c30c:0:0:1]/test/push - Mozilla Firefox
ContikiRPL
[aaaa::c30c:0:0:1]/te...
coap://[aaaa::c30c:0:0:1]:5683/test/push
Search
Most Visited
Intervals
EMail
Z1 Wiki
Yammer
Blog
TinyOS GH
Contiki GH
alignan GH
Z1 files
Sigfox
Curso Aleman
SigFoxDev
Discover
Ping
GET
POST
PUT
DELETE
Observe
Payload
Text
Behavior
CoAP 18
[aaaa::c30c:0:0:1]:5683 (RTT: 32ms)
2.05 Content (Blockwise) (Download finished)
[aaaa::c30c:0:0:1]:5683
.well-known
core
actuators
toggle
test
hello
push
He...
Value
Type
Acknowledgment
Code
2.05 Content
Messa...
61137
Token
empty
Option
Value
Info
Content-Format
text/plain
0
Max-Age
5
1 byte
Block2
0 (32 B/block)
1 byte
Payload (18)
Incoming
Rendered
Outgoing
VERY LONG EVENT 77
Debug Control
Reset
Token
use hex (0x..) or string
Request Options
Accept
Content-Format
Block1 (Req.)
Block2 (Res.)
Auto
block no.
Size1
Size2
total size
Observe
use integer
ETag
use hex (0x..) or string
If-Match
use an ETag
If-None-Match
Uri-Host
Uri-Port
not set
n/s
Proxy-Uri
use absolute URI
Use Proxy-Scheme option
Response Options
Max-Age
use integer
Location-Path
Location-Query

Chapter 23. RESTfull HTTP example with curl.

What is REST?

REST stands for Representational State Transfer. (It is sometimes spelled "ReST".) It relies on a stateless, client-server, cacheable communications protocol - and in virtually all cases, the HTTP protocol is used.

REST and the Internet of Things (and Services) can be an excellent match. REST implementations are lightweight: HTTP clients and servers are now available even on the smallest, IP-enabled platforms.

The key abstraction of a RESTful web service is the resource, not a service. Sensors, actors and control systems in general can be elegantly represented as resources and their service exposed through a RESTful web service.

RESTful applications use HTTP 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.

http://www.restapitutorial.com/

Install curl

```
sudo apt-get install curl
```

In `/examples/rest-example/Makefile` switch to a HTTP build by disabling CoAP as follow `WITH_COAP = 0`.

Ensure the 2 motes you will be using to test this (Border Router, HTTP server) have flashed a MAC/IPv6 address, be sure to write down the addresses! If you get an error like the following,

Go to `platform/z1/contiki-conf.h` and change `UIP_CONF_BUFFER_SIZE` to 240, or just add this to a `project-conf.h file`.

```
#error "UIP_CONF_BUFFER_SIZE too small for REST_MAX_CHUNK_SIZE"
make: *** [obj_z1/er-coap-07-engine.o] Error 1
```

Server:

```
cd /examples/rest-example/
make TARGET=z1 savetarget
make rest-server-example.upload && make z1-reset && make login
```

Annotate the address, Press Ctrl + C to stop the serialdump script.

Border-Router:

```
cd ../ipv6/rpl-border-router/
make TARGET=z1 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:

```
make connect-router
using saved target 'z1'
sudo ../../../tools/tunslip6 aaaa::1/64
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)
```

```
*** Address:aaaa::1 => aaaa:0000:0000:0000
Got configuration message of type P
Setting prefix aaaa::
Server IPv6 addresses:
aaaa::c30c:0:0:0
fe80::c30c:0:0:0
```

Ping both motes (the http-server in this example has aaaa::c30c:0:0:97):

```
ping6 aaaa::c30c:0:0:97
PING aaaa::c30c:0:0:97(aaaa::c30c:0:0:97) 56 data bytes
64 bytes from aaaa::c30c:0:0:97: icmp_seq=1 ttl=63 time=41.6 ms
64 bytes from aaaa::c30c:0:0:97: icmp_seq=2 ttl=63 time=44.0 ms
64 bytes from aaaa::c30c:0:0:97: icmp_seq=3 ttl=63 time=42.4 ms

--- aaaa::c30c:0:0:97 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 41.641/42.706/44.023/1.016 ms
```

Discover the available resources:

```
curl -H "User-Agent: curl" aaaa::c30c:0:0:0097:8080/.well-known/core
</helloworld>;n="HelloWorld",</led>;n="LedControl"
```

Now let's use curl (http client) to get information from the mote and send commands:

```
curl -H "User-Agent: curl" aaaa::c30c:0:0:0097:8080/helloworld
Hello World!

curl -H "User-Agent: curl" aaaa::c30c:0:0:0097:8080/led?color=green -d mode=on -i -v
* About to connect() to aaaa::c30c:0:0:0097 port 8080 (#0)
*   Trying aaaa::c30c:0:0:97... connected
* Connected to aaaa::c30c:0:0:0097 (aaaa::c30c:0:0:97) port 8080 (#0)
> POST /led?color=green HTTP/1.1
> Host: aaaa::c30c:0:0:0097:8080
> Accept: */*
> User-Agent: curl
> Content-Length: 7
> Content-Type: application/x-www-form-urlencoded
>
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< Server: Contiki
Server: Contiki
```

```
< Connection: close
Connection: close

<
* Closing connection #0

$ curl -H "User-Agent: curl" aaaa::c30c:0:0:0097:8080/led?color=green -d mode=off -i
HTTP/1.1 200 OK
Server: Contiki
Connection: close
```