NOVIA
UNIVERSITY OF APPLIED SCIENCES

An Overview of Enabling Technologies for

THE INTERNET OF THINGS

Johan Westö & Dag Björklund

SERIES R: REPORTS, 10/2014

Abstract

The internet of the future is predicted to be an Internet of things. More and more devices will communicate directly with each other and, in time, this has the potential to make our environment "self aware", or provide something like an ambient intelligence. This drastic change will, however, require globally routable IP addresses for each device, standardized protocols, and wireless energy efficient and secure communication. This report summarizes several of the enabling technologies from both a theoretical and practical perspective. Focus is centred on IPv6, LLNs, 6LoWPAN, CoAP, REST, and how these protocol and architectures are or can be implemented in Contiki OS.

Sammanfattning

Framtidens Internet kommer att vara ett Internet av anslutna apparater. Fler och fler apparater kommer att kommunicera direkt med varandra och med tiden har detta potential att skapa en slags "självmedveten" omgivning eller ambient intelligens. Denna radikala förändring kommer att kräva globalt routningsbara IP adresser för varje apparat, standardiserade protokoll samt trådlös energieffektiv och säker kommunikation. Den här rapporten sammanfattar flera av de teknologier som kan möjliggöra förändringen ur både ett teoretiskt och praktiskt perspektiv. Fokus är lagt på IPv6, LLN, 6LoWPAN, CoAP, REST och hur dessa protokoll och arkitekturer är eller kan implementeras i Contiki OS.

Johan Westö, Novia Univesity of Applied Sciences, johan.westo@gmail.com
Dag Björklund, Comsel System Ltd, dag@iki.fi

Publisher: Novia University of Applied Sciences, Wolffskavägen 35 B, 65200 Vasa, Finland
© Johan Westö, Dag Björklund & Novia University of Applied Sciences
Novia Publications and Productions, series R: Reports 10/2014
ISSN: 978-952-5839-98-2,
ISBN: 1799-4179 (online)
Layout: Johan Westö

Contents

1

Introduction

The Internet is under continuous development, and today we stand before a new revolution. Different terms have been used to describe the coming change, but it can best be understood based on the Internet's history so far. Since its birth, the Internet of today has developed trough three different stages, and these stages are characterized by [Intel 2009]:

1. a network of mainly connected mainframes,
2. wide spread use of PCs, servers, e-mail, and, e-commerce, and
3. social services and connected personal appliances such as cellphones, smart TVs, and tablets.

Until today, the number of connected devices has been restricted by the number of humans. In the coming revolution, this will no longer be the case; *Machine-to-Machine* (M2M) communication will remove this constraint and the number of connected devices can therefore grow exponentially. The number of connected devices exceeded the number of humans on the planet already in 2008, and by 2050, the number is expected to have grown to 50 billion [Cisco]. In [Chui 2010], the authors expected that the increase will mainly be fuelled by new applications in the following six areas:

1. Continuous monitoring. Embedded sensors make it possible to continuously and automatically monitor the health of products, structures, animals, and humans.
2. Increased situation awareness. As more and more products become connected, it also becomes possible to receive more information and more feedback about the surroundings.
3. Sensor-driven decision analysis. Embedded M2M communication makes it possible for algorithms to make decisions without human intervention in a larger extent.
4. Process optimization. Better and more feedback from processes will make it possible to achieve a higher degree of control.

5. Optimized resource consumption. An increased information exchange between different peers is one of the corner stones for building smart grids.

6. Complex autonomous systems. Embedded sensors will make it possible for autonomous systems to operate together in a higher degree, an example would be collision avoidance systems.

As the above list suggests, the decreased cost of connected embedded systems will lead to increased M2M communication and more connected devices. These devices will be found everywhere in our environment and they will eventually, as described in [Ramos 2008], create something like an ambient intelligence, as household devices, clothes, and health monitoring devices become connected. As this happens, buildings too become self-aware, to some extent, and capable of self-diagnostics. Embedded sensors and actuators lead to an increased situation awareness that, in turn, can improve both energy efficiency and the indoor air quality.

An ambient intelligence will require that the number of connected devices far outnumber the amount of people on the planet, and hence, the Internet in the future will be an *Internet of Things* (IoT). The expected increase in connected devices will, however, require 1) an increased address space, 2) standardized protocols, 3) wireless energy efficient communication, and 4) use of cryptography for secure communications. Solutions for all these problems already exist as IPv6 can be used over Low-power and Lossy Networks. Furthermore, open source operating systems, such as Contiki, support IPv6, and hence, make it possible for anybody with suitable hardware (microcontroller and radio chip) to connect their own device.

This report strives to give a theoretical and practical introduction to some existing and emerging technologies that will make an ambient intelligence and the IoT a reality. The material is intended for people with a previous knowledge about computer networks and communication protocols, and emphasis has here been put on IPv6, LLNs, CoAP, and the Contiki OS.

The work has been done as part of a larger project named “Teori Möter Arbetslivet” (TEMA) at Novia University of Applied Sciences, in collaboration with Umeå University, the Country Council of Västerbotten, the Local and Regional Government Finland, and Comsel system Ltd. Funding has been obtained from EU/Botnia Atlantica, the Regional Council of Ostrobothnia, Region Västerbotten, and from the partners themselves.

2

IPv6

Version 6 of the IP protocol was introduce during the 1990s as an update for IPv4. The most noticeable change is an expansion of the address space from 32 bits to 128 bits. The 32 bits in IPv4 only allow for around 4.3 billion unique addresses (less than one per human), while the 128 bits in IPv6 allow for 3.4×10^{38} unique addresses. This is on the order of 4×10^{28} addresses per person, and it should be more than enough for building an ambient intelligence.

The lifetime of IPv4 has been extended greatly by the use of *Network Address Translation* (NAT). This mechanism makes it possible for several hosts, located behind a NAT router, to share a single globally routable IPv4 address. The drawback with this type of temporary solution is that hosts behind the NAT router are not directly accessible from the Internet. For clients, in client-server communication, this is not a problem since it is the clients that initiate communication, and it is therefore the server that needs to be accessible on the Internet. In todays Internet the clients greatly outnumber the servers, and most people do not run any servers at their homes. However, in a true IoT, where M2M communication is bidirectional (not only sensors reporting to a server, but true collaboration between things) it is fundamental that all nodes are truly a part of the Internet; they therefore need their very own IP address. This makes IPv6 an absolutely central enabling technology for the IoT.

On the other hand, NAT has provided a fairly secure environment for the IPv4 hosts behind a NAT router, as they are not directly reachable from the Internet. Security thus becomes an important part of the technologies required to enable the IoT, as all nodes will be truly connected and exposed. Security will, however, here only be touched upon briefly as the main purpose is to provide an introduction to IPv6.

2.1 Terminology

The following terminology has been defined in [RFC 2460] and it is also used here.

Node	A device with IPv6 implemented.
Router	A device capable of routing packages to other nodes than itself.
Host	Every node that is not a router.

Upper layer	A layer in the protocol suite that is situated right above IPv6, e.g. TCP, UDP, or ICMP.
Link	The medium over which nodes can communicate on the link layer, e.g. Ethernet, 802.11 (Wi-Fi), or 802.15.4.
Neighbours	Nodes connected to the same link.
Interface	The node's connection to a specific link.
Address	An IPv6 address for one or several interfaces.
Package	IPv6 header and accompanying data.
Link MTU	Maximum packet length in bytes for the current link.
Path MTU	Smallest maximum link MTU for the whole path from sender to receiver.

2.2 IPv6 addresses

The address hierarchy for IPv6 is specified in [RFC 4291] and what here follows is a short summary. An IPv6 address consists of 128 bits and depending on type it can be used to represent one or several interfaces. The three different types of addresses used are:

Unicast	Specific address.
Anycast	Reference to a group of interfaces where the message is delivered to the nearest group member based on the routing protocol's distance measure.
Multicast	Reference to a group of interfaces where the message is delivered to each member.

In text, addresses are written on the form x:x:x:x:x:x:x:x where each x represents a 16 bit hexadecimal number. Leading zeros do not have to be written out and subsequent zeros can be truncated with a double colon notation (can only be used once). A practical example of the previous statement is given by:

Original	FE80:0:0:0:0:1234:5678:9ABC
Simplification	FE80::1234:5678:9ABC

A part of the IP address is designated to be a prefix. This makes it possible to both route traffic and to recognize the address type. The bits in the prefix are calculated from the left, and the number of prefix bits is given with the notation $/n$, where n is the number of bits (just like the CIDR notation that also was adopted in IPv4). This notation makes it easy to describe different address spaces, and as an example, Table 2.1 presents some of the reserved address spaces.

Table 2.1: Reserved address spaces

Address type	Binary prefix	IPv6 notation
Unspecified	00...0	::/128
Loop back	00...1	::1/128
Multicast	11111111	FF00::/8
Link-local unicast	1111111010	FE80::/10
Global unicast	Everything else	

2.2.1 Unicast

Unicast addresses normally consist of a 64 bit subnet prefix and a 64 bit interface ID. The subnet prefix is handed out by the *Internet Service Provider* (ISP), while the interface ID can be obtained from a modified EUI-64 address , a *Dynamic Host Configuration Protocol* DHCPv6 server, a randomized process, or from manual configuration. For products with a 48 bit MAC address , such as Ethernet or Wi-Fi modules, this address can be converted into an EUI-64 address and further to an interface ID to be included in an IPv6 address. The two necessary steps for this conversion are:

1. Insert FF:FE between the third and the fourth byte (creation of an EUI-64 address).
2. Invert bit 7 counted from the left (creation of an interface ID).

As an example, the interface ID for the MAC address 00:11:22:33:44:55 is obtained as:

1. 00:11:22:FF:FE:33:44:55 (EUI-64 address).
2. 0211:22FF:FE33:4455 (interface ID).

Observe that the MAC and EUI-64 addresses are grouped into two byte parts, while in the notation used in IPv6 addresses, groups of four bytes are used.

Unfortunately there is a significant privacy problem with IPv6 addresses created from modified EUI-64 addresses. This originates from the fact that EUI-64 addresses are supposed to be unique, and hence, the IPv6 addresses becomes traceable. To solve this problem, [RFC 4941] describes how randomly generated interface IDs can complement those that have been generated from EUI-64 addresses. That is, a node can still be reached on its EUI-64 address; but for outgoing communication, it will use a randomly generated interface ID to avoid becoming traceable.

2.2.2 Anycast

Anycast addresses refer to more than one interface, and a package is delivered to the nearest interface determined by the routing protocol's distance measure. No specific implementation is defined, but possible uses include gathering a group of routers under one anycast address. This would make it possible to contact the nearest router in a network, or to direct traffic via a specific ISP. Anycast can also be used in IPv4, and perhaps the most typical case is the anycast address 192.88.99.1. This address is shared by a number of so called 6to4 relay

servers, and these can be used to connect IPv6 hosts trough the IPv4 Internet to other IPv6 hosts. More about this in section 2.8.

2.2.3 Multicast

Multicast addresses start of with the prefix FF (8 binary ones), followed by four flags, four additional bits to determine the scope, and finally 112 bits to indicate a group ID (Figure 2.1). The flags are normally all zero, and the scope field is normally set to 2. This configuration means that the address is limited to be link-local, and as a result, link-local multicast addresses start of with FF02.

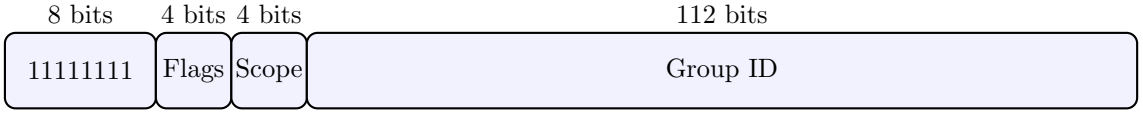

Figure 2.1: IPv6 multicast address.

Some multicast addresses are predefined; e.g the addresses given in Table 2.2. Of the presented addresses, the solicited-node address and the all-PRL-nodes address are a little bit special. The former is used for *Duplicate Address Discovery* (DAD) introduced in subsection 2.7.3, and the latter is used to address all nodes participating in an routing protocol presented in section 3.4. For DAD, the Xs in the address are replaced with the last 24 bits from the IPv6 address being checked.

Table 2.2: Predefined multicast addresses

Address	Definition
FF02::1	All nodes (link-local)
FF02::2	All routers (link-local)
FF02::1A	All-RPL-nodes (link-local)
FF02::1:FFXX:XXXX	Solicited-Node Address

Multicast, however, requires that the network's routers know to which multicast addresses the nodes in the network are connected to. This is achieved with the *Multicast Listener Discovery* (MLD) protocol, presented in section 2.6, in which all nodes are required to announce to which multicast groups they belong. As an example, all nodes have to belong to their own solicited-node address.

2.3 The IPv6 header

Each packet starts of with an header that indicates the sender, the receiver, and information about the package's content. This header is illustrated in Figure 2.2, and below follows a short description for each field [RFC 2460]:

Version IP version, in this case number 6 (4 bits).

Traffic class Used to give different priorities for different packages (8 bits).

Flow label	Indication to routers on how the package should be handled (20 bits).
Payload length	The length in bytes of everything following the header (16 bits).
Next header	Reference for indicating the next header (8 bits).
Hop limit	Maximum number of times that a package can be forwarded (8 bits).
Addresses	The source and destination IPv6 addresser (128 bits).

In order to avoid sending unnecessary information, IPv6 allows for extended headers. These can include information about routing, fragmentation, and security. If extension headers are used, the next header field will describe which extension header that follows. In other words, the next header field can identify either an extension header, a higher level protocol, or the header of an encapsulated IP package (tunnelling in section 2.8).

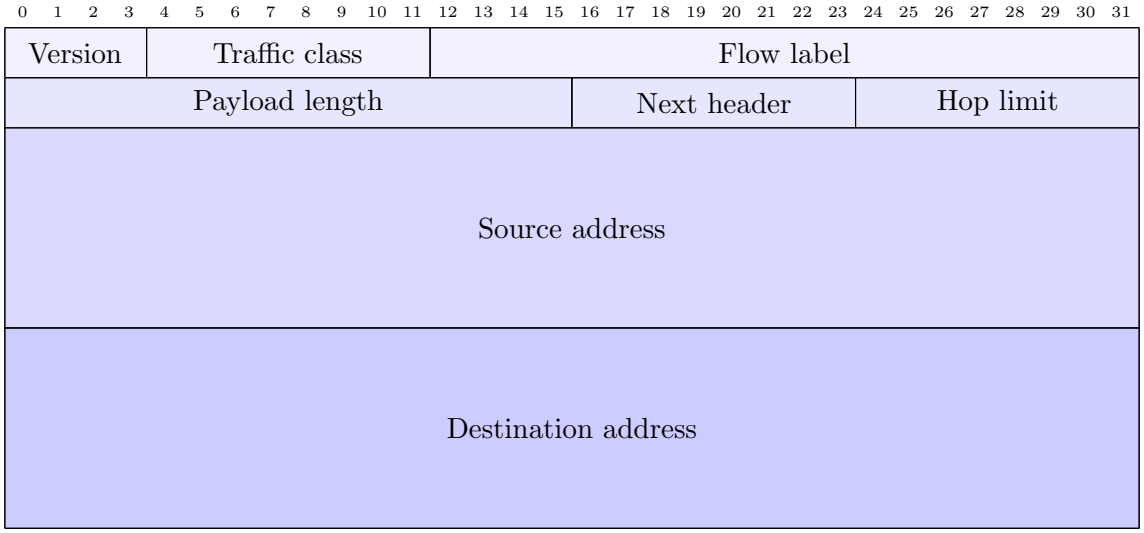

Figure 2.2: IPv6 header.

2.4 ICMPv6

Internet Control Message Protocol version 6 (ICMPv6) contributes with both error and information messages for IPv6 networks. These messages have a broader function in IPv6 than in IPv4, and important network protocols such as neighbour discovery (section 2.5), autoconfiguration (section 2.7), and RPL (section 3.4) are all dependent on ICMPv6 messages. The specification for the protocol is found in [RFC 4443], but additional messages can be specified in other *Internet Engineering Task Force* (IETF) documents. All ICMPv6 messages are identified with the next header value 58 in the IPv6 header, and messages always start off with the following three fields:

Type	Types 0–127 are error messages, while types 128–255 are information messages (8 bits).

Code	Adds additional options for each message type(8 bits).
Checksum	Check to avoid corrupt messages.

The contents of the rest of the message depends on the message type, and the *Internet Assigned Numbers Authority* (IANA) keeps a register of these. Some important message types, taken from [IANA ICMPv6], are given in Table 2.3, and the use of most of these messages will be explained in this document.

Table 2.3: Example of ICMPv6 messages

Type	Name	Reference
1	Destination Unreachable	[RFC 4443]
128	Echo Request (ping)	[RFC 4443]
129	Echo Reply (ping)	[RFC 4443]
130	Multicast Listener Query	[RFC 2710], Section 2.6
131	Multicast Listener Report	[RFC 2710], Section 2.6
132	Multicast Listener Done	[RFC 2710], Section 2.6
133	Router Solicitation	[RFC 4861], Section 2.5
134	Router Advertisement	[RFC 4861], Section 2.5
135	Neighbour Solicitation	[RFC 4861], Section 2.5
136	Neighbour Advertisement	[RFC 4861], Section 2.5
137	Redirect Message	[RFC 4861], Section 2.5
155	RPL Control Message	[RFC 6550], Section 3.4

2.5 Neighbor Discovery

The *Neighbour Discovery* (ND) protocol, as defined in [RFC 4861], describes the ICMPv6 messages for *Router Solicitation* (RS), *Router Advertisement* (RA), *Neighbor Solicitation* (NS), *Neighbor Advertisement* (NA), and *Redirect Message* (RM). These messages make it possible to perform address resolution (find the link-layer address given an IPv6 address), *Neighbour Unreachability Detection* (NUD), autoconfiguration, and *Duplicate Address Detection* (DAD). Below follows a short description of the different messages and how they are used. For a more detailed description regarding these processes see [RFC 4861].

2.5.1 Router Solicitation

Nodes can ask for RA messages directly by sending a RS message. The receiver of the message is normally the link-local all routers multicast address, while the sender is either the unspecified address or the real address of the sender, if it has one. If the sender has got an IPv6 address, it should also append its link-layer address so that replies can be sent as unicasts.

2.5.2 Router Advertisement

Routers send out RA messages periodically to all link-local nodes or as replies to RS messages. The messages contain flags indicating if DHCPv6 (see subsection 2.7.2) is used, and if additional information, such as *Domain Name System* (DNS) information, also can

be obtained trough DHCPv6. Depending on router configuration, the RA message may also include an MTU or a 64 bit network prefix.

2.5.3 Neighbor Solicitation

NS messages are used for address resolution (equivalent to the *Address Resolution Protocol* (ARP) used in IPv4), NUD, and DAD. When the intent is to do address resolution or DAD, the message is sent as multicast; and when the intent is to check reachability, the message is sent as unicast. The sender normally uses one of its IPv6 addresses for the interface, unless DAD is performed. In this case, the sender instead uses the unspecified address. As additional options, NS messages include a target field which is used to indicate the targeted receiver. This is needed whenever a message is sent to a multicast address. Finally, in all cases when the sending nodes address is not the unspecified address, the sending nodes link-layer address should be appended in the message so that replying nodes do not have to perform address resolution before replying.

2.5.4 Neighbor Advertisement

A node will send a NA message in response to a received NS message, or to inform other nodes of its presence. The message includes flags informing the receiver of whether the sender is a router, responding to an NS message, or if it has a new link-layer address. When responding to an NS message, the receiver is the requesters unicast address, whereas it in all other cases it is the all nodes multicast address. Just as the NS message, the NA message also includes a target field. For solicited messages, this field contains the requesters address; and for unsolicited messages, it contains the address of the node whose link-layer address has changed. Unsolicited messages should further always include the sending nodes link-layer address.

2.5.5 Redirect Message

In cases where a router detects that there is a better route for a sender to use, it can forward this information to the sender using a redirect message. The message then includes the destination address, and the address of a better first-hop router.

2.6 Multicast Listener Discovery

As presented in Table 2.3, the multicast listener discovery protocol defines three ICMP messages in [RFC 2710]. These messages make it possible for a router to monitor which multicast addresses have listener on its different links. Slightly simplified, the protocol lets routers ask nodes to which multicast groups they belong using a query message, and similarly, nodes can register or unregister their presence in multicast groups to the router using a report/done message.

2.7 Autoconfiguration

Autoconfiguration can be achieved using two different methods in IPv6. The first of which lets nodes generate addresses (stateless), and the second which relies on DHCPv6 (stateful). Below follows an introduction to both methods.

2.7.1 Stateless Address Autoconfiguration

When routers include a network prefix in their RA messages, the only addition needed for obtaining a global IPv6 address is a unique interface ID for the link used. Hence, stateless address configuration, as described in [RFC 4862], works in two stages. In the first stage, a node generates an interface ID randomly (see [RFC 4941] and [RFC 3972]), or from an embedded IEEE Identifier (MAC address), and in the second stage, it check that the identifier is unique trough DAD, also specified in [RFC 4862]. In the absence of a router, the method can also be used for setting up link-local communications. This is accomplished by replacing the subnet prefix from the router with the link-local prefix from Table 2.1.

2.7.2 Stateful Address Autoconfiguration

Stateful address autoconfiguration, also known as DHCPv6, is defined in [RFC 3315]. This protocol designates UDP ports 546 and 547 for communication between servers (routers), relay agents[1], and nodes. In order to make it easy for nodes to find a DHCPv6 server, the special multicast address FF02::1:2 has also been defined for addressing both DHCPv6 servers and relay agents. Using this address, nodes can initiate communication and obtain settings for how to communicate within the network.

[RFC 3315] defines 12 different messages, but in typical situations, the configuration is set up using client-server exchanges involving two or four messages. In cases where a client already has an IPv6 address, additional information can be obtained in a single request–reply communication. In other cases, the client first sends a solicit message to locate servers. Available servers respond with an advertisement, including addresses and settings, after which the client selects one server and a IPv6 address to use. For verification, the client sends a confirmation to the selected server which in turn send back a reply. As the servers hand out addresses dynamically, each address has a lifetime. Once the configuration is done, additional renew messages will have to be sent periodically by the client to keep its address from expiring.

2.7.3 Duplicate Address Detection (DAD)

DAD should be performed regardless of whether stateless, stateful, or manual configuration has been used to assign each node an address. As long as this process is incomplete, the obtained address is said to be tentative. The DAD process is initiated when a node is signing up to its solicited multicast address for the tentative address. Once this is done, the node sends an NS message to the tentative address with the sender being the unspecified address. Based on these two measures, it is possible to detect a duplicate address if either another NS message is received (another node performing DAD); or if a NA response message is received for the previously sent NS message (another node is already using the tentative address). If DAD fails, the node can not assign the address to its interface and an error should be logged.

2.8 IPv6 Transition Mechanisms

It will take time before a full transition from IPv4 to IPv6 is complete. During this transition phase there are a number of mechanisms for connecting islands of IPv6 hosts to the emerging IPv6 Internet. These can, as noted in [RFC 4213], be divided into two

[1] **Relay agents** are here defined to be nodes that can relay traffic between a node and a DHCPv6 server on different links.

categories (1) **Dual IP stacks** and (2) **tunneling**. Dual stacks imply that both routers and nodes have a complete IPv4 and IPv6 stack implemented, which in time will make it possible to move over newer implementation to IPv6 and phase out IPv4.

Tunneling, on the other hand, is about transporting packets of some protocol inside packets of another (where the packet encapsulated is not a higher level protocol of the encapsulating one). The most common example of tunneling is *Virtual Private Network* (VPN) tunnels. In these, IPv4 packets with local addresses are transported inside IPv4 packets with global addresses over the Internet, hence connecting e.g. islands of corporate LANs. Similarly, tunneling can also be used to transport IPv6 packets trough the IPv4 Internet by encapsulating them in IPv4 packets. In order to indicate that an IPv4 package has an IPv6 package encapsulated, [RFC 4213] states that the next protocol field should be set to 41. At present, tunneling is the only viable alternative for connecting different IPv6 islands to each other, and for this reason three different alternatives (6to4, 6in4, and teredo) are presented below.

During the transition phase, it is also likely that there will be a need for letting IPv6 only nodes connect to IPv4 only servers. Under the assumption that a border router exists with both an IPv4 and an IPv6 interface, [RFC 6146] specifies a NAT64 protocol for the above situation specifically. This protocol's functionality is very similar to traditional NAT as it lets several IPv6 only nodes share the border router's IPv4 address for communication with IPv4 servers. NAT64 is also going to be released for Contiki within a near future [Dunkels 2014].

2.8.1 6to4

The most common tunneling protocol, specified in [RFC 3056], is called 6to4. This method uses a standardized way of creating IPv6 addresses from a global IPv4 address. The method relies on that a specific part, 2002::/16, of the IPv6 address space has been assigned to 6to4 border routers. These obtain a 48 bit IPv6 subnet prefix from a global IPv4 address by inserting the 32 bit IPv4 address on the right side of the 16 bit prefix 2002::/16. As an example, the IPv4 address 192.1.2.3 turns into c001:0203 when converting the bytes to hexadecimal and grouping them into 16-bit parts. Together with the 6to4 prefix 2002::/16, this gives the IPv6 subnet prefix 2002:c001:0203::/48. Thus, the 48 bit prefix then leaves an address space of 128-48=80 bits for local host or subnet addresses. A single global IPv4 address can therefore provide a subnet of 2^{80} globally routable IPv6 addresses.

Two scenarious can then occur when an IPv6 node, behind a 6to4 border router, wants to send a packet to another IPv6 node on the internet. First, if the destination address starts with 2002::/16. This means that the packet is addressed to a node located behind another 6to4 border router. In this case, the senders's border router simply extracts the IPv4 address of the receiver's border router using the rule explained above, encapsulates the message in an IPv4 packet, and sends it off over the Internet as normal IPv4 traffic. When the receiver's border router gets the packet, it extracts the original IPv6 package and sends it to the intended destination within its IPv6 network, see Figure 2.3.

For the second scenario, where the receiving node is not behind another 6to4 border router, the 6to4 protocol makes use of relay servers with the anycast address 192.89.99.1, as specified in [RFC 3068]. These servers function as borders for the native IPv6 Internet. More specifically, this means that once a packet reaches a relay server it can continue its path towards the destiantion as an IPv6 package. 6to4 border routers will therefore encapsulate packets destined for the native internet in an IPv4 packet with the destination address 192.89.99.1. As soon as such a package reaches a relay server, the original IPv6

package is extracted and sent away normally to its destination on the native IPv6 Interent. An example where the final destination is ipv6.google.com is provided in Figure 2.3.

Because of the above mentioned functionality, the 6to4 mechanism is an **automated tunneling** protocol. This originates from that there is no need to explicitly set up two two tunnel ends. Once one tunnel end is implemented, it can automatically communicate with any other 6to4 border router. The nodes within one 6to4 network can therefore communicate with a node on any other 6to4 network, or the native IPv6 Internet.

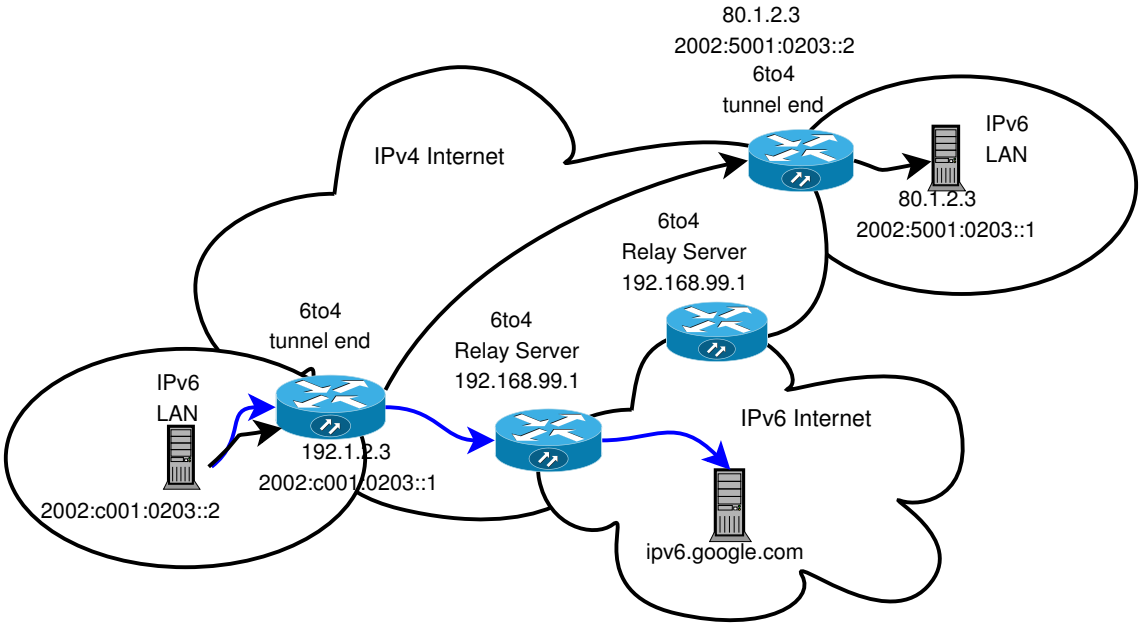

Figure 2.3: 6to4 Tunneling. Blue lines show 6to4 host to native IPv6 communication, black lines show a 6to4 to 6to4 path.

2.8.2 6in4

The name 6in4 is used to refer to the tunneling mechanism specified in [RFC 4213], despite that it is newer used in the specification. The functionality is very similar to 6to4, but contrary to 6to4, 6in4 is not an automated tunneling protocol. That is, 6in4 does not use a standardised way to infer a IPv6 address from an IPv4 address. It instead requires that two explicit tunnel ends are set up and manually configured for each separate case. Hence, 6in4 is said to be an **configured tunneling** protocol.

2.8.3 Teredo

The 6to4 protocol requires access to a global IPv4 address. That is, it can not be set up behind a NAT router. This restriction is lifted with 6in4, but with the added complexity of requiring routers to be manually configured. The Teredo protocol, on the other hand, encapsulatesthe IPv6 packets inside transport layer datagram (UDP/IPv4 datagrams). This allows use of the normal NAT mechanism for outbound traffic, and as a new feature, Teredo introduces a way to also allow for inbound traffic. To accomplish this, Teredo also makes use of relay servers, its own prefix 2001:0000::/32, and specific messages to keep paths open in both directions.

Teredo was developed by Microsoft, and a good introduction can be found at the protocols origins in “Teredo Overview” by [Microsoft 2003], but Teredo has also been

standardized by the IETF in [RFC 4380]. As a protocol, Teredo is much more complex than 6to4 or 6in4; and it is largely beyond the scope of this document. However, it is extreamly easy to set up; and it is certainly an **automated tunneling** mechanism.

2.8.4 Security

Tunneling, as described above, can have severe effects on security and some aspects of this therefore have to be covered. Teredo is perhaps the most dangerous protocol, as anyone can very easily use it to punch unexpected holes in many corporate NATs (not all, Teredo does not work with all types of NAT and a restrictive firewall also prevents Teredo from working). Similarly, both 6to4 and 6in4 can cause firewall problems as they require that packets with the next header field 41 are allowed to pass. When these messages in turn contain a complete IPv6 package, the firewall might end up letting every kind of package trough that is encapsulated if care is not taken.

3

Low-power and lossy networks

The term *Low-power and Lossy Networks* (LLNs) intends to highlight some significant differences between traditional IP networks and new smart sensor networks. In traditional networks, both routers and nodes are mains-powered and the links connecting them are highly stable. Wireless smart sensors on the other hand often have a very restricted power source. This could be due to a limited battery or limited energy harvesting capabilities. The facts that nodes can run out of power, be moved, and be placed in a noisy environment further gives rise to highly unstable links. Hence, the communication requirements for smart sensors are very different from traditional IP networks, especially since the sensors may also have to act as routers. Some adaptations are thus necessary in order to make it possible to connect sensors to the internet, and these adaptations will be discussed in this chapter. Despite that different physical-layer standards exist for LLNs, such as IEEE 802.15.4, IEEE 802.11 (Wi-Fi), and *Power Line Communication* (PLC), this report focuses on 802.15.4. The reason for this is the standards ability to facilitate low-power wireless communication, and the fact that significant work by IETF has already been done in adapting the 802.15.4 standard to IP based communication [Vasseur 2010].

3.1 IEEE 802.15.4

IEEE 802.15.4 is a radio technology standard specified by the *Institute of Electrical and Electronics Engineers* (IEEE) for *Personal Area Networks* (PAN). A PAN is defined to be a network with low-power nodes and with low-data-rate requirements, hence, the maximal data rate in 802.15.4 is limited to 250 kbits/s [IEEE 802.15.4].

The standard specifies both a physical layer and a *Medium Access Control* (MAC) layer. The physical layer can utilize some different frequency bands among which the 2.4 GHz band is available globally. This 83.5 Mhz wide band has been divided into 16 channels, numbered 11–26, while channels numbered 0–10 are used in the open subgigahertz band (Americas only). Unfortunately all 16 channels overlap with channels used by Wi-Fi in Europe, and these are therefore likely to be disturbed by stronger Wi-Fi signals. However, if Wi-Fi implementations only use the European non-overlapping channel set (Wi-Fi channels 1, 7, and 13), this leaves the 802.15.4 channels 15,16, 21, and 22 undisturbed [IEEE 802.11; IEEE 802.15.4]. As another solution, it is also possible to use channel hopping schemes and work on this exists for Contiki.

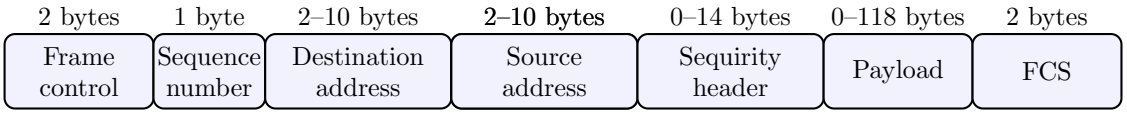

Figure 3.1: IEEE 802.15.4 frame structure, adapted from [IEEE 802.15.4].

The MAC layer is responsible for channel access, validation of received frames, and for acknowledging validated frames. This means that retransmissions of lost frames can be handled directly at the link layer. The maximal length of a frame is 127 bytes and its format is given in Figure 3.1. The functionality of the different fields are:

Frame control	Flags specifying how the rest of the frame should be interpreted, and if an acknowledgements is requested.
Seq. no.	Current sequence number.
Addresses	Addresses of both sender and receiver. An IEEE 802.15.4 address can be either 16 or 64 bits long, and additionally, each address can be accompanied by a 2 byte PAN id.
Security	Information regarding encryption and authentication if used.
FCS	Frame Check Sequence. A *Cyclic Redundancy Check* (CRC) code that can be used for detecting bit errors.

3.1.1 Topologies and beacons

Networks using the 802.15.4 standard can form either star topologies or peer-to-peer topologies (mesh networking requires higher layers on top). Both topologies, however, require that one node is selected to be a PAN coordinator. For star topologies, communication only occurs between the coordinator and connected devices (all traffic is routed trough the coordinator), whereas peer-to-peer topologies allow a device to communicate with any other device within radio range. 802.15.4 additionally supports two different types of PANs: 1) beacon enabled PANs and 2) non-beacon enabled PANs. If beacons are enabled, the coordinator periodically synchronizes the network by sending a beacon frame. If beacons are not enabled, devices can instead send messages at any moment by first performing *Carrier Sense Multiple Access with Collision Avoidance* (CSMA-CA) to check that the channel is free.

3.2 6LowPAN

Using IPv6 on top of IEEE 802.15.4 has one major drawback in that 802.15.4 has an MTU of only 127 bytes, whereas the MTU for IPv6 is 1280 bytes. As a consequence, IPv6 packets might have to be fragmented before they can be sent over a 802.15.4 link. Another problem with the small MTU is that encapsulated IPv6 packages, with their 128 bit addresses, take up a big portion of the available 127 bytes. A single IPv6 header is 40 bytes long (with no extended headers), and if TCP is used as well, this header adds another 20 bytes. The overhead for TCP/IP communication then adds up to 60 bytes. As presented in Figure 3.1, the header for 802.15.4 can add an additional 39 bytes in which case there is only 28 bytes left for data.

In order to find a solution to the above problems, a working group named 6LowPAN (IPv6 over Low power wireless Personal Area Networks) was appointed, by the IETF, to create an adaptation layer between IPv6 and IEEE 802.15.4. The fundamentals of this layer is specified in [RFC 4944], and the main purpose is to make communication over 802.15.4 links fulfil the requirements stated by IPv6. To this end, the dispatch header, the mesh addressing header, and the fragmentation header have all been defined, and these can be used in similar fashion to the extended headers in IPv6. For identification purposes, they have to be used in a specific order, and each one has to be preceded by an unique bit sequence. Examples of specified sequences are given in Table 3.1. As can be noted, it is the two left most bits that identify the header type. Sequences starting with 00 are not 6LowPAN frames and should be discarded by the receiver.

Table 3.1: Bit sequences for identifying LowPAN headers

Pattern	Name	Description
00 xxxxxx	NALP	Not a LowPAN frame
01 000001	IPv6	Uncompressed IPv6 addresses
01 000010	LowPAN HC1	LowPAN HC1 compressed IPv6
01 1xxxxx	LowPAN IPHC	LowPAN IPHC compressed IPv6
10 xxxxxx	MESH	Mesh header
11 000xxx	FRAG1	Fragmentation header (first)
11 100xxx	FRAGN	Fragmentation header (subsequent)

Of the above headers, it is mainly the dispatch header and fragmentation header that are of interest. The mesh addressing header is only used when routing is performed at the network layer, and it is therefore not of interest if RPL (see Section 3.4) is implemented. Besides the adaptation layer, the 6LowPAN group has also developed a more effective method to perform neighbour discovery over 802.15.4. This modification is found in [RFC 6775], but Contiki still uses the original ND protocol for IPv6; the modified version will therefore not be presented here.

3.2.1 Fragmentation

IPv6 messages that are too big to fit within a single frame have to fragmented. This is accomplished by encapsulating the first fragmented part with FRAG1 and subsequent fragments with FRAGN (see Table 3.1). These headers include information about the IPv6 package's total length, its tag number, and an offset to indicate the fragments position (only in FRAGN).

3.2.2 Header compression

Two header compression schemes exist, one older version defined in [RFC 4944] and a newer one defined in [RFC 6282]. The older one, known as HC1 or HC2, is likely to be deprecated at some point and focus will therefore be put on the newer one called IPHC. This header is identified with the initial sequence 011 and has a total length of two or three bytes. The different fields in the IPHC header indicate which fields in the IPv6 header that are compressed and how. Uncompressed IPv6 headers fields follow the IPHC header in the normal order that they would occur in an uncompressed header. Under optimal circumstances, the IPHC compression technique can reduce the IPv6 header from 40 bytes (no extended headers) down to 4 bytes. Additionally, it also includes options for

compressing a few subsequent headers, e.g., some IPv6 extended headers and UDP. UDP can in the best case be compressed from 8 bytes down to 2 bytes, whenever both source and destination ports lie in the interval 61616–61531.

Implementing IPHC does not by itself guarantee a significant compression, although it provides very good compression in the best case. As the IPv6 addresses takes up most of the header space, the big question is if these can be compressed well or not. Good compression requires addresses that work well with IPHC's compression mechanism and for this there are two options. If the interface reference is determined from the link layer address, the IPv6 address can be elided completely. Similarly, if an address is of the form fe80::ff:fe00:XXXX then only the last 16 bits will be sent. These two cases assume that packets are sent as link-local unicast and that the subnet prefix then is fe80/10 padded with zeros. This is obviously a serious restriction, and for this purpose the IPHC header includes a *Context Identifier Extension* (CID) bit. If this bit is set, an additional byte will follow the normal two byte IPHC header that gives the possibility to indicate 16 different compression contexts for both the sender's and receiver's address. However, at present no mechanism has been defined for nodes to agree upon the meaning of a certain context. This means that compressions related to different contexts have to predefined and configured manually.

3.3 Radio duty cycling

Nodes operating on a restricted energy source, such as a battery, require that both hardware and software are designed for energy efficient operation. In the case of radio communication over IEEE 802.15.4, transmission of data is as energy demanding (15–20 mA without amplifier) as simply listening to the channel[2]. It therefore follows that the radio, both Rx and Tx, has to be put in sleep mode if energy is to be saved. This in turn creates new problems since a node will not know it somebody is trying to send it a frame while it is sleeping. A solution is to implement *Radio Duty Cycling* (RDC) that specifies a predetermined method for communication between sleeping nodes. This can be done either synchronously or asynchronously, where the former is more efficient but requires synchronized clocks [Vasseur 2010]. The 802.15.4 standard has built in support for synchronized RDC using beacons and this method will be presented briefly below. Contiki, however, does not yet support beacon enabled PANs and instead it relies on its own hybrid RDC method called ContikiMAC. This method too will be described in some detail below.

3.3.1 RDC using 802.15.4 beacon frames

In beacon enabled PANs, the coordinator defines a superframe that makes up the basis for how communication occurs within the PAN. This superframe is divided up into two regions: 1) an active period and 2) an inactive period, where the active period is further divided up into 16 time slots of equal duration. If a device wants to send a message, it will wait until the coordinator sends a beacon, whereupon the device uses CSMA-CA to try to send its message within one of the time slots (a device is required to stop transmitting when the time slot ends). Each sending device therefore "competes" with the others for channel access during the current time slot (end result determined by the random back-off time in CSMA-CA). If needed, it is possible to assign a subset of the 16 time slots for specific

[2] **Transmission** of the RF signal consumes almost an insignificant amount of energy compared to the base-band processing, i.e. modulation/demodulation, coding/decoding, spreading/despreading etc. The base-band processing again is about as costly in the transmitter as in the receiver

devices so that these are always guaranteed a time slot. As illustrated in Figure 3.2, this functionality divides the 16 time slots in the active period into a *Contention Access Period* (CAP) and a *Contention Free Period* (CFP).

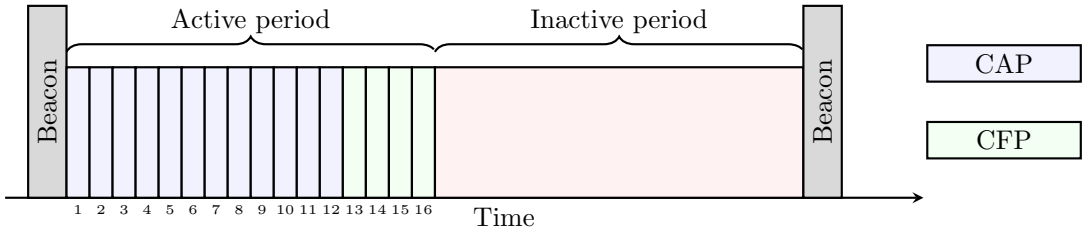

Figure 3.2: 802.15.4 superframe.

As coordinator beacons provide synchronization to the network, 802.15.4 PANs has a built in way to handle RDC for communication with devices that sleep most of their time. However, the defined mechanism only works for communication between a coordinator and a connected device. Whenever the device wants to send a message to the coordinator, it will wake up, wait for a beacon, and then try to send its message in one of the time slots. If the device can control its own sleep cycle, this can be used to ensure that wake up occurs just prior to an expected beacon, as these are transmitted periodically. Similarly, when the coordinator wants to send a message to a device, it will announce this in the transmitted beacon. Every sleeping device therefore has to wake up periodically, listen for a beacon, and send a query message to the coordinator in one of the time slots. Upon receiving a query, the coordinator will respond by transmitting the message to the device as it knows that it is now listening. As the coordinator can store messages, sleeping devices do not need to wake up prior to every expected beacon. They can instead sleep over several beacons and periodically wake up to check for pending messages in the next beacon. The coordinator is normally thought of as a node with main power, but it can also restrict its energy usage by sleeping trough the inactive period of the superframe [IEEE 802.15.4].

3.3.2 ContikiMAC

ContikiMAC was presented by [Dunkels 2011] as a suitable and energy efficient RDC mechanism for sensor networks running Contiki. By putting the majority of the communication load on the sender (which is ok, as Tx is as costly as Rx), ContikiMAC allows nodes to only periodically wake up and listen for incoming transmissions. This means that a sender is required to send repeated copies of a frame (strobes) throughout a complete sleep cycle or until an achknowledgement is obtained. There is no need for synchronization, but it is recommended as a sender can use such information to start sending strobes just before the receiver is expected to wake up. Figure 3.3 illustrates the basic process in more detail, and at the same time, the meaning of the following six functionally important time values.

$\tau_{\mathbf{cycle}}$	Wake up interval for sleeping nodes.
$\tau_{\mathbf{s}}$	Strobe transmitting time (dependent on frame length).
$\tau_{\mathbf{i}}$	Interval between strobes.
$\tau_{\mathbf{a}}$	Time needed before an acknowledgement to a sent frame can be detected by a sender.

$\tau_\mathbf{r}$	Time needed to perform a *Clear Channel Assessment* (CCA).
$\tau_\mathbf{c}$	Interval between CCA checks.

In short, ContikiMAC works by forcing every node to periodically wake up with an interval given by τ_{cycle}. Upon awakening, the node does two successive CCAs[3] separated by τ_c (two are needed in case the first one happens to occur in between two strobes), and if no signal is detected the node goes back to sleep. If on the contrary a signal is detected, the node will keep its radio on until the whole frame has been received, whereupon the node sends an acknowledgement. Unicast strobes are hence sent until acknowledgements are received or for maximum time of τ_{cycle}. As there is no way of knowing if all nodes received a broadcast[4] frame, strobes must be sent throughout the complete interval τ_{cycle} whenever IPv6 multicast packages are handled.

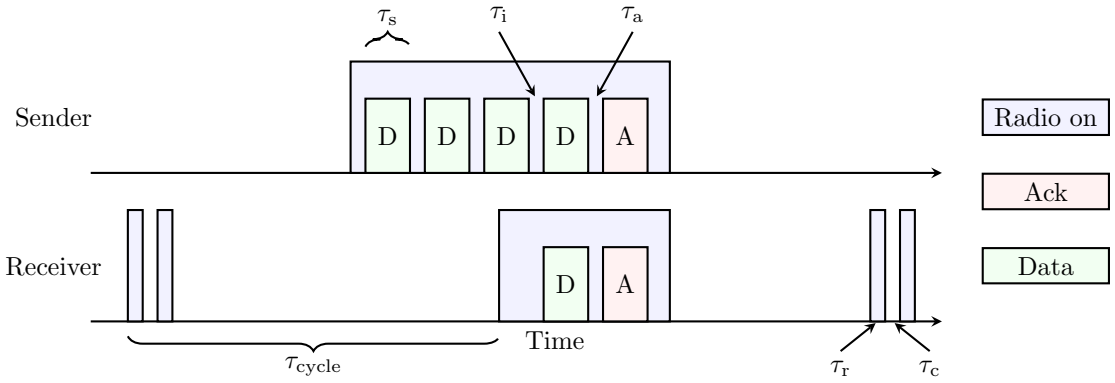

Figure 3.3: ContikiMAC RDC mechanism.

The above given functional description of ContikiMAC results in the following constraints for the different τ values.

$$\tau_\text{a} < \tau_\text{i} < \tau_\text{c} < \tau_\text{c} + 2\tau_\text{r} < \tau_\text{s} \tag{3.1}$$

In text, this means that the time needed to detect an acknowledgement has to be shorter than the interval between strobes. This interval in turn has to be shorter than the interval between CCAs. If not, two CCAs are not guaranteed to detect a sequence of strobes. The last two terms then finally state that successful strobe detection requires a strobe to be longer than the time it takes to perform two CCAs. Regarding numerical values, τ_a can be determined from the IEEE 802.15.4 standard to be 0.352 ms, τ_r is hardware dependent, and τ_i and τ_c can be specified by the user as long as they satisfy the inequality in Equation 3.1. Depending on choices done, the resulting minimum value for τ_p will reflect a lower bound on how short a frame can be (in contikiMAC data frames are used as strobes). In practise, IPv6 communication brings with it enough overhead for this issue to be ignored. Default numerical values in Contiki are $\tau_\text{i} = 0.4$ ms, $\tau_\text{c} = 0.5$ ms, and $\tau_\text{r} = 0.884$ ms. In contrast to the other time values, τ_{cycle} is free from constraints and can freely be selected for a specific implementation. Simulations have, however, shown that a value of 0.125 ms seems to be optimal for a sensor network and this is also the default value in Contiki [Dunkels 2011].

[3] A **CCA** is a *Received Signal Strength Indication* (RSSI) measurement that tries to assess in a simple way, i.e. only by measuring energy, if the channel is in used by another transmitter.

[4] **Broadcast** defines a transmission intended for all other nodes within radio range.

To further improve operation and make the process more energy efficient, the following two additional improvements are added: (1) **Phase-lock** states that each node should keep a record over different neigbors' phases so that strobes can be sent just prior to wake up. (2) **Fast sleep** lets the node go back to sleep quicker if there is an indication that it has detected background noise instead of a frame being transmitted. All in all, it is claimed that all the above would result in an RDC scheme that would let participating nodes keep their radios off 99 % of the time. In real life, the actual energy savings obtained are strongly dependent on hardware. It always takes time for a sensor to first wake up its microprocessor, wake up the radio, sleep the radio, wake up the radio again, sleep the radio again, and then finally put the microprocessor to sleep for another cycle. As this represents performing two CCAs, and since the microprocessor consumes energy throughout this process. The final energy consumption will be dependent on how fast both the MCU and the radio can switch between different states. Furthermore, the amount of noise and other traffic in the air will also affect how often false CCAs are triggered. The noise level may become higher than the RSSI threshold set in the CCA, which will force the node to do a complete reception of a frame which will end up in a CRC miss, discarding of the frame, and late sleep.

3.4 Routing with RPL

Due to both the restricted capabilities of participating nodes and the unreliability of the used links, LLNs encounter routing requirements not found in traditional networks. For this reason, the IETF created the work group *Routing over Low-power and Lossy networks* (ROLL) to specify and define a specialised routing protocol. The resulting protocol is called *Routing Protocol for Low-power and Lossy Networks* (RPL) and is specified in [RFC 6550].

3.4.1 DODAGs and modes of operation

RPL connects nodes in a tree like structure called a *Destination Oriented Directed Acyclic Graph* (DODAG)[5], illustrated with an example in Figure 3.4. The node at the top is called a DODAG root, and traffic moving towards the root is said to move in the upward direction, whereas traffic moving from the root is said to move in the downward direction. Communication within a DODAG is defined by four *Modes of Operation* (MOP), and the root is responsible for selecting which of the following modes the DODAG uses:

1. No downward routes maintained by RPL.
2. Non-storing mode of operation.
3. Storing mode of operation with no multicast support.
4. Storing mode of operation with multicast support.

In mode 1, only upward traffic is allowed; and nodes are restricted to send information to the root, to nodes along the path to the root, or to another network that the root is connected to. Mode 2 allows for downward traffic as well, but it relies on source routing by the DODAG root. That is, the root has to specify a complete path to the target node (made possible by [RFC 6554]). Non-root nodes, therefore, do not have to store any routing

[5] **A directed acyclic graph** is a set of nodes connected by directed edges in such a way that no loops are formed.

specific information, but traffic within the DODAG always has to go via the root as it is the only node capable of routing. Mode 3 and 4 lifts the previous constraint by requiring each node to store routing information. This means that traffic will either flow directly down the nodes own sub-DODAG,[6] or upwards until a common ancestor for both sender and receiver is found, whereupon the traffic again goes down the ancestors sub-DODAG.[7] This also means that packets lacking a routable destination address will gather at the DODAG root whereupon they are dropped. The difference between mode 3 and 4 lies in their support for multicast. Mode 4 requires each node to store both unicast and multicast addresses to nodes in its sub-DODAG, while mode 3 only requires storage of unicast addresses, under the assumption that multicast can be handled in some other unspecified way (e.g local flooding or several unicasts). Each node that joins the DODAG must be able to honour its mode of operation. If the node does not have sufficient capabilities for this, it can only join as a leaf node with no downward connections.

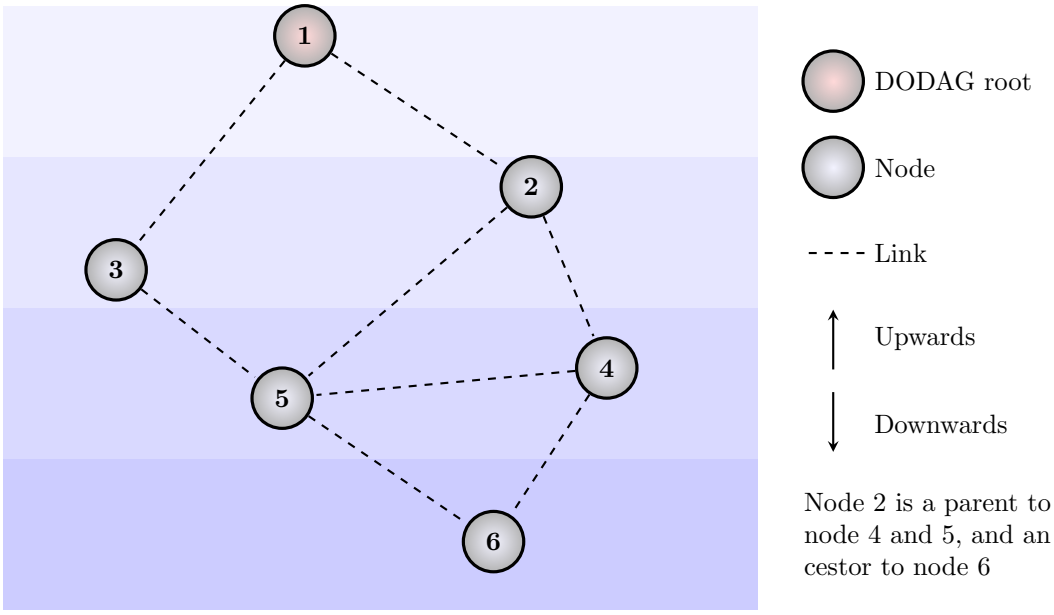

Figure 3.4: Example DODAG (a DAG would, strictly speaking, require directed edges, but these are left out from the figure as it does not include a specified sender/receiver path).

3.4.2 Hierarchy

The RPL protocol builds upon a hierarchy with four levels (instance, DODAG, version, and rank). Each level constitutes a set, and each set member has its own subset with members belonging to the next level in the hierarchy. At each level, different properties are determined and these are propagated downwards in the hierarchy. As illustrated in Figure 3.5, the instance level determines an *objective function* (OF),[8] the DODAG level specifies a root, the version level indicates the current DODAG version, and finally, the rank level indicates each node's position within the DODAG topology (distance to the root). It is possible for a node to participate in several DODAGs, but only if they belong

[6] **A sub-DODAG** refers to the DODAG part beneath a certain node, nodes in which are dependent on that node for access to the root.

[7] **Parents and ancestors** are a nodes immediate or subsequent successors on the path towards the root.

[8] **An objective function** defines how the DODAG topology is to be created.

to different instances. This then makes it possible for a node to use a different DODAG, built based on a different OF, for different types o traffic.

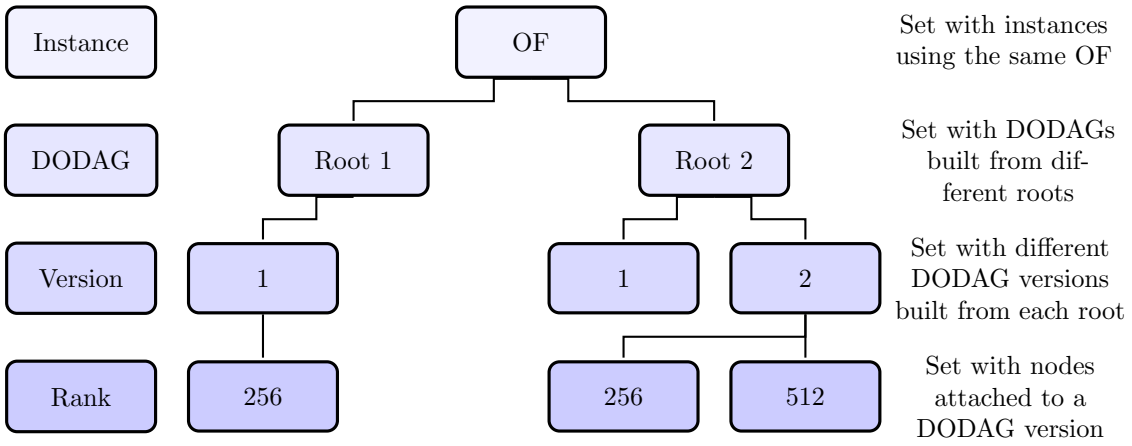

Figure 3.5: RPL hierarchy.

3.4.3 Objective function, rank, and topology

The creation of a DODAG's topology is guided so that the selected OF is minimized without violating any of its constraints. Variables for OFs are defined in [RFC 6551] and they consists of different link and node metrics such as throughput, latency, *Expected Transmissions* (ETX), energy (type of power source or remaining energy), and hop-count. The specification of an OF defines how the OF value is determined from available metrics, how this value is used to determine a nodes rank, and how it can be used to select a parent set and a preferred parent from a node's neighbours.[9] At present, only two different OFs have been defined: (1) OF0 in [RFC 6552] and (2) the *Minimum Rank with Hysteresis Objective Function* (MRHOF) in [RFC 6719]. Of these, OF0 is meant as a basic OF that does not require any metric to be measured and it will, using default configurations, end up minimizing hop-count. MRHOF is slightly more complicated and can compute a nodes rank based on the additive metrics hop-count, latency, and ETX (default). The RPL implementation in Contiki uses MRHOF as its default objective function, and it will be presented in more detail in subsection 3.4.5.

Unfortunately the concept of rank has a slightly different interpretation when viewed upon in the context of an OF or a DODAG topology. In its most basic form, rank is a 16 bit number that is computed by the OF. Within the DODAG topology, however, this raw rank value is divided by a number called MinHopRankIncrease,[10] whereupon the integer part of the result is interpreted as the rank. To separate between these two different values the latter is referred to as DAGrank. The implementation of MinHopRankIncrease ensures that rank will increase for every step downwards in the DODAG. Using the default value of 256 for MinHopRankIncrease, Figure 3.4 can be updated to Figure 3.6 by introducing hypothetical rank values.

Every root node is required to have a rank of MinHopRankIncrease which will, using the definition above, give them a DAGrank of 1. Similarly, every node beneath the root node will have DAGrank larger than 1. The rank and DAGrank of 0 is reserved for situations where an LLN has several border routers. In these cases, the coordination between different

[9] **Neighbours** are nodes within direct communication range.
[10] **MinHopRank-Increase** defines the minimum increase in rank for each step down the DODAG tree.

border routers over a backbone network gives the appearance of an existing virtual root with rank 0.

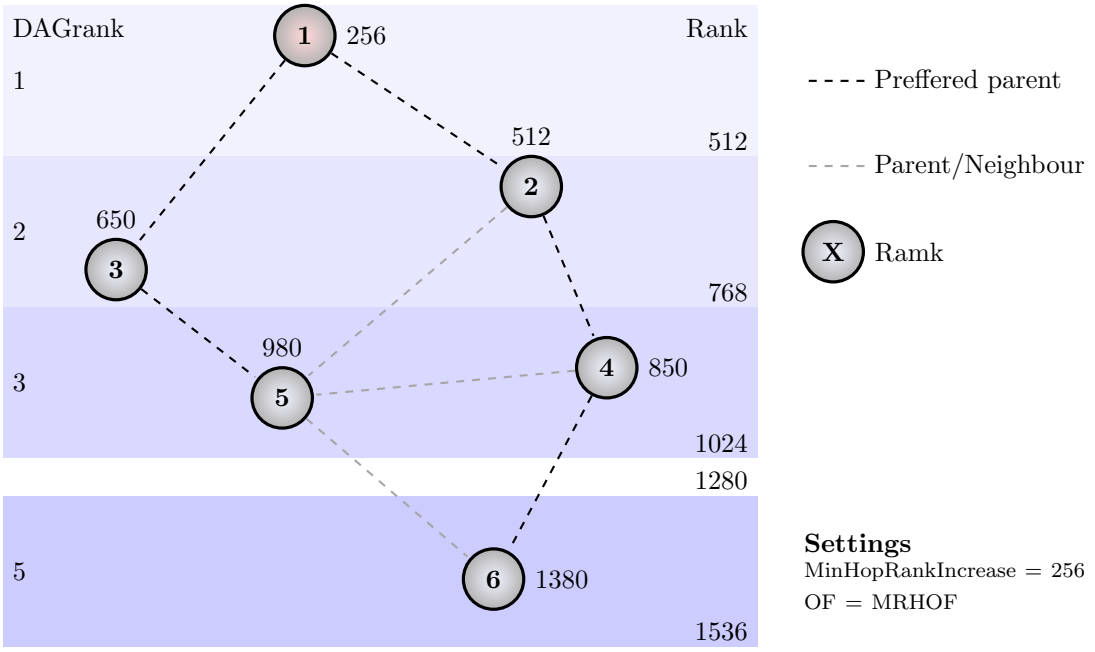

Figure 3.6: DODAG with ranks.

3.4.4 Defined ICMPv6 messages and their usage

[RFC 6550] defines a new ICMPv6 control message with the type number 155. This message is called a RPL control message, and depending on the code number, the message is defined as either a *DODAG Information Solicitation* (DIS), a *DODAG Information Object* (DIO), a *Destination Advertisement Object* (DAO), a *Destination Advertisement Object Acknowledgement* (DAO-ACK), or secure versions of the previous messages.

DIS and DIO messages are both used for building upward routes in the DODAG topology. A node can probe its neighbourhood to detect DODAGs in its vicinity by sending out DIS messages. If DODAGs exist, nodes already connected will respond with a DIO message carrying at least the following mandatory information: RPL instance ID, DODAG ID, DODAG version, rank, MOP, and a bit indicating if the DODAG is grounded.[11] Hence, a DIO message always informs the receiver of the mode of operation used, and it also carries information required to identify a set member on each level in the DODAG hierarchy (see Figure 3.5). Additionally, DIO messages can include options for distributing a network prefix, different metric values to be used by the OF, and different DODAG configuration parameters (e.g MinHopRankIncrease and which OF that is used).

Using received DIO messages, a node can create or maintain a set of neighbours. From this set, a parent set is selected as a subset, and the selection is based upon rules specified by the implemented OF. Similarly, the node within the parent set found to have the best path upwards, as determined by the OF, is selected to be a preferred parent; and traffic to destinations not found in the sending nodes neighbourhood or sub-DODAG is to be sent upwards trough the DODAG via this parent.

DIO messages are also continuously broadcasted in order to detect changes in the

[11] **Grounded** is used to indicate if the DODAG can satisfy an application dependent goal e.g if the root node is connected to the Internet.

DODAG topology. Both node movements and link failures lead to new rank values which will have to be propagated to current neighbours. In order to avoid unnecessary traffic, advertisement of DIO messages are controlled using a trickle timer (defined in [RFC 6206]). This timer increases the interval between messages exponentially to a maximum value when no changes are found. Similarly, when a new node is detected or other new information has to be propagated trough the network, the trickle timer resets and DIO messages are again sent with shorter intervals. Information for how to set up the trickle timer is distributed from the root, via DIO messages, and is later propagated throughout the network by DIO messages from nodes at lower and lower levels.

In contrast to DIO messages, DAO messages are used to create downward routes. From its parent set, each node selects a DAO parent subset that again is OF specific (normally identical to the parent set). DAO messages are sent to each member in the DAO parent set, and these convey information of which addresses that are found within the sending nodes sub-DODAG. This functionality renders DAO messages unnecessary and unused if the mode of operation only allows upward routes (mode 1). Similarly, all DAO messages have to be sent to the DODAG root when the DODAG operates in non-storing mode (mode 2). Only in storing mode (mode 3 and 4) are DAO messages addressed to DAO parents, and the parents will only send new DAO messages to their respective DAO parents if changes have occurred to which addresses that are accessible. To certify that a DAO message have been received, the sender can request the receiver to respond with an acknowledgement, in other words a DAO-ACK.

3.4.5 Minimum Rank with Hysteresis Objective Function

MRHOF, defined in [RFC 6719], computes a node's rank based on either hop count, latency, or ETX. The default is to use ETX, and in such cases the rank field in DIO messages is used to represent the cost for reaching the root. If other metrics are used, DIO messages have to include a metric option that carry information about the used metric. Based on the implemented metric, a node calculates a path cost, to the root, trough each neighbour. This is done by adding up the link/node cost for a neighbour with the metric value found in the DIO message from that neighbour (sent in the rank field or as a metric option). Important to note is that by default cost is calculated as ETX times 128. That is, if it on average takes two transmissions to deliver a frame to a neighbouring node, the cost associated with that link will be 256. A path cost trough a neighbour is therefore by default determined as the rank that the neighbour advertises, in its DIO message, plus 128 times the ETX value for the link to that neighbour.

The parent set is selected from available neighbours by picking out no more than PARENT_SET_SIZE (default value = 3) nodes with a link cost lower than MAX_LINK_METRIC (default value = 512). Of these, the parent with the lowest cost is selected as the nodes preferred parent. All path costs trough neighbours can be converted into rank values by increasing each value, if needed, so that it is higher or equal to advertised rank plus MinHopRankIncrease. To determine its own rank, a node selects the largest of the following values:

- The rank calculated trough its preferred parent.
- One DAGrank higher than the highest DAGrank found in the parent set. That is, MinHopRankIncrease times (highest DAGrank plus one).

- The largest calculated rank through all nodes in the parent set minus MaxRankIncrease (used for local repair see subsection 3.4.8).

Taken together, Figure 3.6 can once more be updated to Figure 3.7 using ETX values that correspond to the hypothetical ranks given earlier.

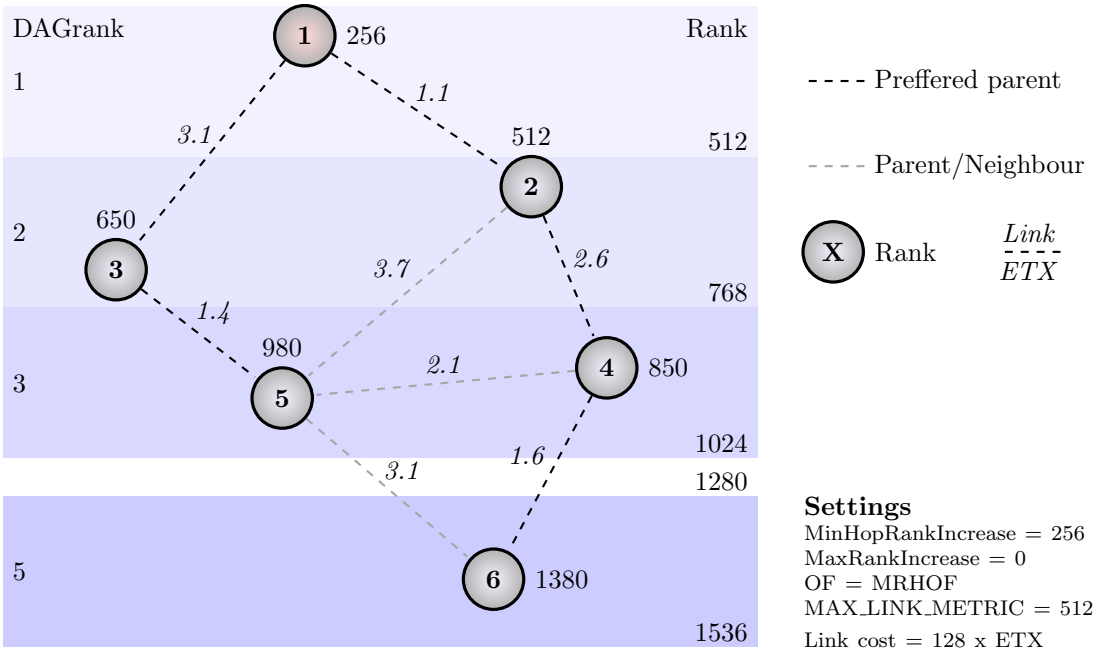

Figure 3.7: DODAG with ranks determined from ETX using the OF MRHOF.

3.4.6 DODAG creation

RPL nodes can be configured so that they, at power up, either (1) wait/probe for messages carrying information about existing DODAGs, or (2) create a new DODAG where the node itself is the DODAG root. A DODAG is hence created when a node configured to be a DODAG root is powered up. Once powered up, the root will start sending DIO messages periodically (triggered by the trickle timer) to announce its presence. From this point onwards, nearby nodes will use information from the DIO messages to determine their own rank, whereupon they in turn start sending out DIO messages. From a configuration perspective, the root is the only node that can change any of the following DIO fields: Grounded, MOP, Version, Instance ID, and DODAG ID. Hence, as joining nodes start sending out their own DIO messages, the previously mentioned fields have to stay the same. Depending on the mode of operation, nodes can select DAO parents and start updating these with DAO messages to inform them about reachable destinations. Initially several messages have to sent as more nodes join the DODAG, but eventually traffic settles down when a complete DODAG have formed.

3.4.7 Autoconfiguration

RPL supports stateless autoconfiguration as prefix information can be embedded into DIO messages. DIO and DIS messages together can therefore replace the functionality performed by NA, NS, RA, and RS messages, and these are therefore disabled in Contiki when RPL is used. There is, however, one exception to the above statement and that

relates to DAD which is not performed by DIO and DIS messages, but section 8.2 in [RFC 6775] states that DAD can be left out in cases where EUI-64 addresses are used to create interface IDs. So in theory this is not a problem, but if addresses are configured manually, care must be taken so that no duplicates are created as there is no mechanism that handles or detects this.

3.4.8 Repair mechanisms

Inconsistencies in a DODAG can be spotted using specific RPL information encoded into every data package sent. This mechanism uses the IPv6 hop-by-hop header extension (see [RFC 6553] for details) to equip each package with information regarding the sending nodes rank, and the intended direction within the DODAG (up or down). Hence, it is possible for each receiver to compare their rank with the sender's and decide if the observed direction of movement corresponds to the intended. Inconsistencies indicate that a loop may have formed within the DODAG and this therefore triggers a local repair mechanism. When this happens, a node will poison[12] its links and then temporally disconnect from the DODAG. Poisoning makes all nodes with a higher DAG rank remove the node from their parent set, if included. This is needed to prevent a new loop from forming if the node rejoins the DODAG under a parent that previously belonged to its sub-DODAG. Poisoning might also empty the parent set for some other nodes, and as a result, these nodes will have to perform local repair as well. Hence, local repair has the possibility to rebuild a part of the DODAG.

It is also possible to completely rebuilt the whole DODAG. This is called a global repair and it can only be triggered by the DODAG root if it increments the DODAG version. There is no specification for when a global repair should be triggered. The option is instead left to be implementation depended.

[12] **Poisoning** is accomplished by advertising an infinite rank which renders the node unusable as a parent.

4

Higher layer architectures and protocols

6LowPAN and IPv6 provide communication channels through the Internet, but in order to create useful applications and systems, some higher layers are needed. These will let connected nodes agree upon how transmitted data is to be formatted, how to find out what kind of services other nodes provide, and how data is retrieved or pushed. All these steps are necessary in order for a lamp to know if someone flipped a switch or not.

Security is an important topic, and IP based communication provides very good tools for e.g. end-to-end security. This chapter, therefore, starts with a brief summary of these technologies.

Bringing IPv6 to even the smallest resource constrained devices is a very good start for getting application developers to connect a diverse variety of things. Adding a large number of proprietary application level protocols on top of IP is on the contrary a step back. A strong trend is to move away from proprietary protocols, and instead extend the use of **web services** using e.g. the *Hypertext Transfer Protocol* (HTTP). This concept will be explored further later on in the chapter, and it will be followed up by an introduction of the *Constrained Application Protocol* (CoAP), which shares a lot of HTTP's features but with less overhead. CoAP can therefore be used to create RESTful systems containing resource constrained nodes. The REST paradigm is currently widely spoken for in the IoT community and it will also be briefly introduced.

Finally, an introduction is also provided for the ZigBee *Smart Energy profile 2* (SEP2). This is the most complete description of IP based communication between apparatus, in a given niche, and it is built upon ZigBee IP. ZigBee IP is not a new protocol definition but merely a specification of which existing protocols, e.g. 6LoWPAN, RPL, and TCP that products conforming to ZigBee IP should support and use.

4.1 End-to-End Security

One of the biggest benefits of running IP, and especially IPv6 with global addresses in each resource constrained node, is perhaps end-to-end security. A message encrypted in one node can be transported unaltered through the Internet to its final destination where it is decrypted. There is no need for secure tunnels, VPNs, or for routers to handle the message in any special way. Network security, including cryptography, key exchange etc., is

a huge topic and it is not covered in this text. Only a brief listing of the major technologies available follows.

There are two main protocols, or protocol families rather, used for encrypting TCP/IP packets on the Internet (and for exchanging keys and parameters needed for the encryption). These are *Transport Layer Security* (TLS) [RFC 5246] and the *IP Security Architecture* (IPSec) [RFC 4301]. The two provide very similar features; the most notable difference between them is that TLS, as the name states, operates on the transport layer (layer 4 in OSI), while IPSec operates on the Internet layer (layer 3 in OSI). IPSec thus protects protocols on top of IP, like TCP, UDP, ICMP etc. while TLS only protects protocols on top of TCP (TLS runs on top of TCP, but there is also DTLS [RFC 6347] for UDP). Usage of IPSec does not require any changes in the TCP/IP communication, whereas TLS requires some code modifications.

Unless a simple pre-shared key approach is used, the most complicated part of the cryptosystem is the key exchanged. For this purpose, IPsec uses IKEv2 [RFC 5996], and TLS the TLS Handshake Protocol.

For IPSec, there are proposals for header compression techniques that could make it more suitable for LLNs [Raza 2010; 6LoWPAN], and there also exists a lightweight version of IKEv2 [Raza 2012]. Similarly, for TLS, work has been done on e.g. a tiny DTLS implementations in [Raza 2013; Prelman 2012].

In addition to end-to-end security that can be provided by the above mentioned technologies, the link layer (layer 2 in OSI) can also be secured separately. Obviously, link layer security is mostly needed for wireless networks, common examples are WEP and WPA for WiFi networks, but it is also possible to secure IEEE 802.15.4 links. This might seem unnecessary if end-to-end security is already achieved, but it will be needed to restrict access to LLNs. Luckily, most devices have hardware acceleration for calculating crypto algorithms that can be used equally well for end-to-end security as for securing links. Despite of this, secure communication will always require additional energy and this might cause trouble for energy constraint devices.

4.2 Web Services

As stated above, usage of freely available unified application layer protocols is likely to ease the spread of the IoT, especially if it is integrated with the world wide web at the same time. When nodes are not only connected to the Internet but also to the web, one can talk about the *Web of Things* (WoT). A web service is a web based interface to e.g. the data, the computational services, or the actuators of some system. Where a web site is designed for machine-man interaction, a web service is instead designed for machine-to-machine interaction. However, there is often a web site front end connected to a web service system for human interaction with the system. As an example, a flight booking system that can find the most suitable flights from a number of airlines is a web service that interacts with web services of the different airlines. There is no human in the loop as the system is fully automatic. The booking system does of course then also offer a web site that a human can use to interact with the web services and search for and book a flight. The web site can essentially be a front end to the underlying web service.

Three major paradigms for web service implementation exist: The *Simple Object Access Protocol* (SOAP), *Representational State Transfer* (REST), and XML with *Remote Procedure Call* (XML-RPC). Among these, SOAP can actually be considered an evolution of XML-RPC [Castillo 2011]. All three normally use HTTP for transport, and at least SOAP and XML-RPC, but often also REST, use the *Extensible Markup Language* (XML)

for representation. All are also application level protocols, architectures, or paradigms (SOAP and RPC are more like protocols, whereas REST is more a way of thinking), and as such, they can replace a large number of legacy protocols thus unifying the IoT.

As the IoT is often about automating things, suitable higher layer protocols for connecting things could also be chosen from the ones already adopted within industry, such as Modbus, Profibus, CAN bus, HART, and KNX. However, web service based systems are much more platform independent, scalable, and easier to integrate with the Internet and the web. Furthermore, these also support service discovery without having to incorporate a human in the loop. For these reasons, implementations relying on web services are here thought to show greater promise for developing the IoT. Of the three web service architectures, REST is the simplest, and thus believed to be the prevailing paradigm for realizing the WoT, and it is thus introduced next.

4.2.1 Representational State Transfer (REST)

The REST architecture was originally introduced by [Fielding 2000]; The aim was to make possible applications built upon loosely coupled services that could be shared among many users, and as an example, it could be noted that the World Wide Web is based on REST. A **resource** in a REST architecture is an abstraction controlled by a server and identified by a *Universal Resource Identifier* (URI).[13] Each resource has a **representation** that is a content, value, or state of that resource at a given time. A REST architecture is built around **requests** and **responses** that transfer representations of resources.

REST states that communication should be based on a client/server architecture where the servers are stateless, which means they should not "remember" anything about what the clients have asked in previous requests. Each request from a client must thus contain all information needed by the server to complete the transaction with a response. This makes the servers very scalable, as they do not need to maintain open dialogues with numerous clients.

Resources in REST are accessed and manipulated using a protocol on the application layer based on client/server request/response dialogues. The REST model does not specify which protocol should be used, but it requires that the protocol uses a small set of uniform commands, like the ones in HTTP (GET, POST, PUT, and DELETE). Systems that fulfil these requirements are said to be RESTful. As HTTP fulfils the requirements, a vast majority of applications actually use HTTP (on the Internet, different sites often state that REST is defined as basically anything that uses HTTP requests, but this is not quite the case).

When applying REST in the context of the IoT and M2M applications, the nodes and their sensors and actuators become abstract resources identified by URIs [Colitti 2011]. Typical URIs for a sensor node could look something like this [Shelby 2009]:

```
http://sensor10.example.com/sensors/temp
http://sensor10.example.com/sensors/light
http://sensor10.example.com/sensors/enabled
```

Reading the temperature by a GET command for the first URI above could then return the value as plain text, or perhaps an XML document as below (a human can also do GET operation to the URI by writing the URI into a web browser):

[13] A URL is an address to a resource, while a URN is the name of the resource. **URI** is a common term for both.

```
<temp value="277.18" min="275.93" max="278.71" unit="kelvin"/>
```

Instead of HTTP and XML as above, another possibility is to use CoAP + JSON (CoAP is introduced in section 4.3, and JSON is a compact format that mostly can replace XML).

In order for RESTful systems to interact, there should be some level of agreement on what resources should exist, under which paths, which methods different resources should support, and how the resources and their parameters can be discovered given some base URI. There are some proposed standards that try to unify the design of RESTful interfaces for IoT systems by e.g. the *Internet Protocol for Smart Objects* (IPSO) Alliance, the IETF's *Constrained RESTful Environments* (CoRE) working group, and the ZigBee Alliance. Luckily these more or less build upon each other, instead of working in different directions. In [RFC 6690] and [CoRE Interface], a format for describing and discovering resources is proposed, which is used in defining proposed standard function sets (explained later) and resources in [IPSO]. [CoRE RD] also introduces a way of hosting descriptions of resources on sleepy nodes, on other nodes. Furthermore, ZigBee SEP2, which is briefly introduced in section 4.5, is RESTful and it defines, among other things, required or optional resources, their paths, and types.

The IPSO Application Framework and the CoRE Interfaces

Lets have a look at one of the above mentioned REST based interface design proposals. The IPSO Application Framework defines, in [IPSO], a RESTful design for use in IP based smart object systems, such as Home Automation, Building Automation, and other M2M applications. It is intended to be complementary to SEP2 and, as mentioned, it also builds upon the CoRE Interfaces. It defines a number of **function sets** that all have a recommended **root path**, under which their sub-resources are organized. The function sets and their root paths proposed by IPSO are given in Table 4.1.

Table 4.1: IPSO application framework function sets and recommended root paths

Function set	**Root path**
Device	/dev
General Purpose IO	/gpio
Power	/pwr
Load Control	/load
Sensors	/sen
Light Control	/lt
Message	/msg
Location	/loc
Configuration	/cfg

The IPSO Application framework further suggests paths for the sub-resources (the actual resources) and more importantly also for their *Resource Types* (RT), *Interface descriptions* (IF), data types, and units (the concepts of RT, IF, and also Function Sets are defined in [CoRE Interface]). An example will clarify: IPSO proposes the sub-resources in Table 4.2 with corresponding parameters for the Light Control function set.

Table 4.2: The IPSO application framework Light Control function set

Function	Path	RT	IF	Type	Unit
Light Control	`/lt{#}/on`	`ipso.lt.on`	a	b	
Light Dimmer	`/lt{#}/dim`	`ipso.lt.din`	a	i	0–100 %

According to the table, a light control should have IF="a", which means actuator. An actuator should support methods GET, PUT, and POST, according to the suggestions by the CoRE working group listed in Table 4.3 (there the interface descriptions are put inside the namespace `core`, i.e. `core.a` for actuator). The RT field further reveals that the resource is a light control (usually also the path of the resource reveals this to a human at least), and the `{#}` in the paths can be an arbitrary string or index e.g. a "bedroom light 1" could have the URI: `/lt/bed1/on`.

Table 4.3: Suggested Interface description (IF) attributes and the methods they should support

Interface	IF	Methods
Link List	`core.ll`	GET
Batch	`core.b`	GET, PUT, POST (where applicable)
Linked Batch	`core.lb`	GET, PUT, POST, DELETE (where applicable)
Sensor	`core.s`	GET
Parameter	`core.p`	GET, PUT
Read-only parameter	`core.rp`	GET
Actuator	`core.a`	GET, PUT, POST
Binding	`core.bnd`	GET, POST, DELETE

Adapted from [CoRE Interface].

4.2.2 Resource Discovery

In M2M applications there is often a need for clients and servers to find and interact with each other without a human in the loop. A client then needs to find out e.g. the Interface Descriptions (IF), Resource Types (RT) and the actual URIs of the resources residing on a given server (e.g. a sensor node), or on an unspecified server. This process is called resource discovery. The resources and their properties can e.g. be obtained by retrieving them in the *CoRE Link Format* specified in [RFC 6690], by doing a GET to a standard resource. In [CoRE Interface], the path of this is defined to be `/.well-known/core`. As an example, the request:

```
REQ: GET /.well-known/core
```

To a node containing a light switch at uri `/lt/bed1/on` and also a temperature sensor at uri `/sen/temp` could return the following representation, in the CoRE link format:

```
</lt/bed1/on>;if="core.a";rt="ipso.lt.on",
</sen/temp>;if="core.s";rt="ipso.sen.temp">
```

Resource Discovery can be performed using either unicast or multicast. That is, a client can search for resources on nearby servers without knowing on which node a certain type

of resource might exist. Multicast discovery would be performed by sending a GET request for `/.well-known/core` to the appropriate multicast address in the given network. The request could contain query parameters, e.g. for obtaining only certain resource types, in order to reduce the possibly large and numerous responses. Some information, like existing function sets on a node, can also be exported to DNS based messages. This is used e.g. in the SEP2 profile.

A node or an application, say a wireless light switch, could perform resource discovery in its neighbourhood, by sending a multicast GET with query parameter for getting only resources with resource type `ipso.lt.on`, i.e. lights it could control. Any responses would further contain the interface type etc. in the that would reveal that the light is in fact an actuator, and thus we know we can GET the binary state of the switch, or control it by PUT or POST.

4.3 Constrained Application Protocol (CoAP)

The problem with HTTP in resource constrained systems is, apart from the protocol overhead, the lack of support for multicast messages and built in subscription mechanisms [Colitti 2011]. In order to address the overhead and the shortcomings of HTTP, the IETF's CoRE working group has developed the *Constrained Application Protocol* (CoAP) [CoRE CoAP]. CoAP includes a subset of the functionality of HTTP that is redesigned to suite resource constrained systems, and it also adds other capabilities to address special needs of M2M applications and the IoT.

4.3.1 UDP as Transport Layer

As illustrated in Figure 4.1, UDP is used in place of TCP as the transport layer protocol. This because the flow control and retransmission mechanisms, and other overheads of TCP, are not considered acceptable for LLN networks by the CoRE group (and the authors of this document). Note still that HTTP+TCP is used e.g. in ZigBee SEP2 (section 4.5), and also preferred by other distinguished IoT developers.

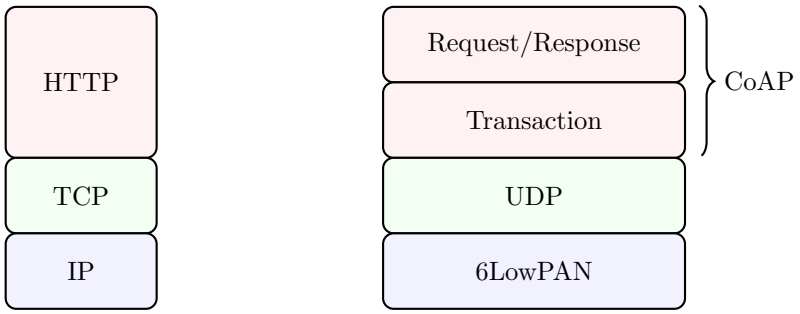

Figure 4.1: HTTP and CoAP protocol stacks

The UDP protocol is very simple and only adds a header with source and destination ports, in contrast to TCP that adds additional handshakes, flow controls, and error handling. UDP, therefore, does not provide for error free communication by error detection and retransmission as TCP, but if needed, CoAP provides an optional simple retransmission mechanism.

Figure 4.1 also illustrates that CoAP is divided into two sub-layers, where the actual RESTful communication takes place in the Request/Response layer, while the transaction

layer handles individual messages between nodes. These messages can be of four different types:

Confirmable	Confirmation i.e. an acknowledgement is required from the receiver.
Non-confirmable	No acknowledgement is required from the receiver.
Acknowledgement	An acknowledgement message.
Reset	Response when a confirmable message has been received, but discarded.

So, messages of type "Confirmable" are retransmitted if a packet is corrupted, and error free transmission over UDP is hence possible.

4.3.2 Message Format

CoAP messages, illustrated in Figure 4.2, have a compact binary format and a minimum length of four bytes. In addition to the fixed fields (Ver, T, TKL, and Code), the message can also carry a token, options, or a payload; below follows a description of each field:

Ver	CoAP version, currently 1 (2 bits).
T	Indicates message type as 00) confirmable, 01) non-confirmable, 10) acknowledgement, or 11) reset (2 bits).
TKL	Token length in bytes (4 bits).
Code	Specifies if the message is a request or a response and also what type of request/response message (e.g. get, put, post, or delete) (8 bits).
Message ID	Used for pairing acknowledgement/reset messages with confirmable/non-confirable messages (16 bits).
Token	Used for matching requests and responses. (0–8 bytes).
Options	Can be used with both requests or responses to indicate e.g. the content format or a location path.

It might seem unnecessary to have both a token and a message ID, but as will be explained in subsection 4.3.4 below, a resource can be observed, which means that a request does not get only the immediate response but keeps getting notifications when the resource changes. These notifications are really multiple responses to the request and thus carry the same original token used by the client in its initial request, whereas the message id for each transaction changes. This also makes the token quite an important concept needed by application programmers to trigger correct handlers for responses.

Even if the header length can be as small as 4 bytes, it typically is around 10 bytes. This, in addition to using UDP instead of TCP, reduces the data transmitted in a CoAP transaction to one tenth of that in a HTTP transaction (the TCP header is 20 bytes, without options, and a HTTP header can be hundreds of bytes, compared to 8 bytes for UDP and $\approx$ 10B for CoAP). This, in turn, could double the battery life of a device

according to studies in [Colitti 2011]. Certainly everything depends on the communication intervals. If the device is idle almost all the time, the efficiency of the radio duty cycling discussed earlier in this text will dictate the batter life almost completely.

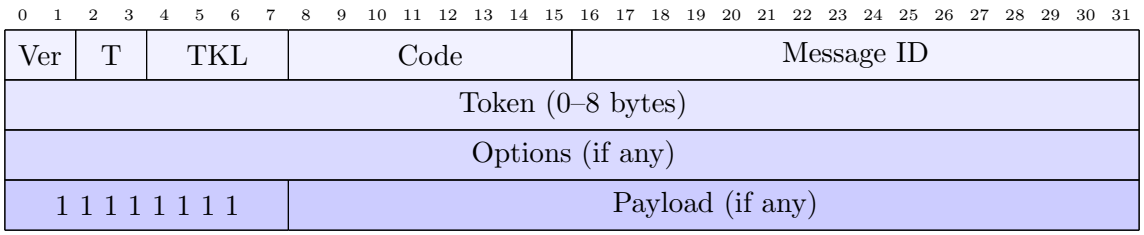

Figure 4.2: CoAP Message Format, adapted from [CoRE CoAP].

4.3.3 HTTP vs. CoAP Transactions

The reduced protocol overhead due to the compact CoAP header is only part of the story. Perhaps an even bigger benefit of CoAP over HTTP for LLNs is the simplified transaction, which is illustrated in Figure 4.3.

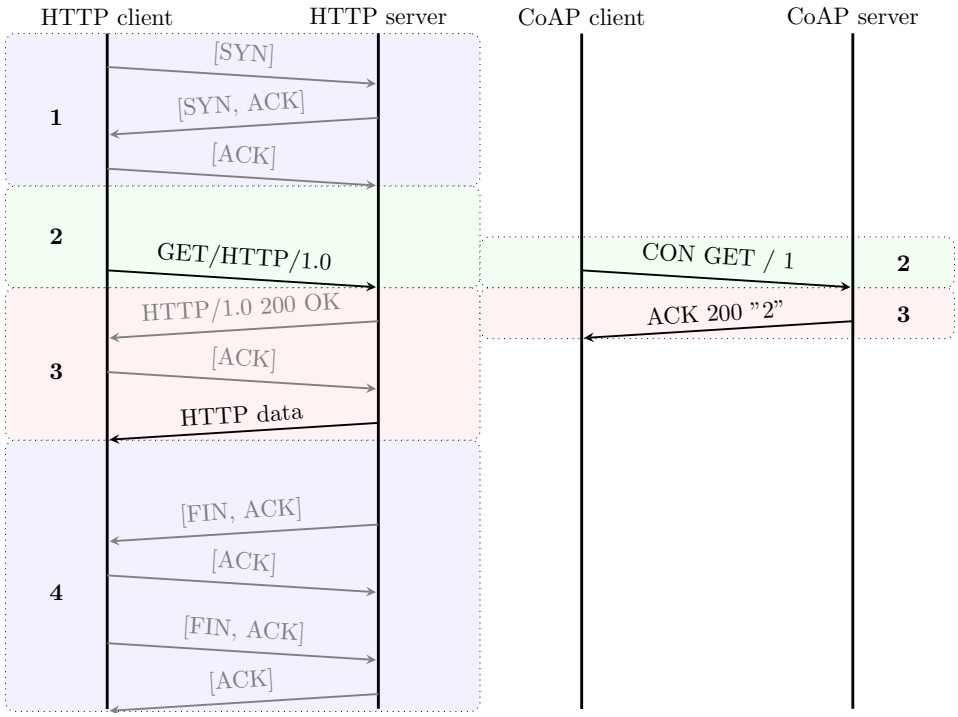

Figure 4.3: An HTTP transaction versus a CoAP transaction. The numbered fields represent 1) a three-way handshake, 2) a request, 3) a response, and 4) a three-way handshake (termination). Adapted from [Pötsch 2011].

Retrieving the representation of a resource on a CoAP server is as simple as sending the GET request and retrieving the ACK, with the data piggy backed in return. In contrast, a HTTP transaction involves 1) a three-way handshake to open a TCP connection, 2) the actual request (GET in this case), 3) a HTTP acknowledgement followed by a client acknowledgement and the data transfer, and finally 4) a three-way handshake to close the TCP connection. In a sleepy network doing radio duty cycling, each of these messages can take a long time and many router nodes can be repeatedly woken up on the way, with all the additional strobing and CCA detections this involves (see chapter 3).

4.3.4 Observing Resources with CoAP

In IoT and M2M applications, it is common that clients want to be notified about changes in resources on some server. Normally in a client/server system, like REST based ones, clients do requests to servers, but certainly a lamp should not be polling a switch, instead the switch needs to inform the lamp only when its state changes. Unfortunately, HTTP has no direct support for this (though it can be accomplished as they have done in the ZigBee SEP2 profile, introduced in section 4.5), but CoAP has a special addition for this purpose, defined in [CoRE OR], which provides a mechanism for clients to **observe** resources on a server. This allows a client to be continuously notified about changes in the representation of the observed resource.

Figure 4.4 illustrates how a client registers as an observer of the resource `temperature` on a server by sending a GET request with the `Observe` option set. When the representation of the resource, the temperature, subsequently changes, the server sends notifications to the client.

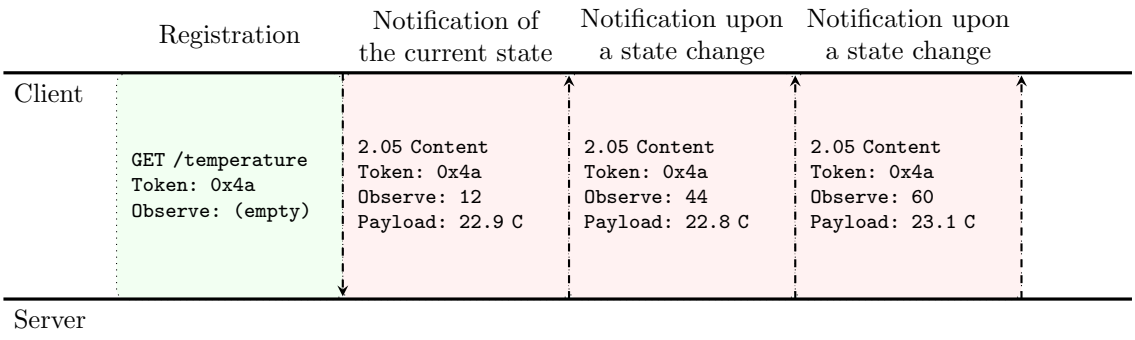

Figure 4.4: Observing a resource using CoAP, adopted from [CoRE OR].

As already discussed when dissecting the CoAP message format, the figure shows how the notifications carry the same Token as the original request. Notifications, however, do not necessarily arrive in the same order as they were sent. The `Observe` field is therefore used to allow a client to discard a notification that is older than one it has already received. See [CoRE OR] for details about e.g. how the field is incremented.

It is up to the server to decide when a resource has in fact changed. Should for instance the observers be notified when the temperature changes by 0.01 degrees, or only when it changes a full degree? The protocol does not provide a mechanism for setting these thresholds, but quite naturally, the server would present this parameter as a resource in its RESTful API, or as query parameters of the resource in question. In [CoRE Interface], the latter is advocated. There, a set of query string parameters are defined to allow a client to control e.g. how much a resource value should change for the new representation to be interesting. For example the following request could set the change step of a temperature sensor to two degrees.

```
PUT /s/temperature?st=2
```

4.3.5 HTTP/CoAP gateways

Gateways are certainly something that the authors of this report do not want, and if they have to be used, they should be transparent. In the present context, gateways here

represent some form of protocol translator, like ZigBee <–> IP gateways or KNX[14] <-> IP gateways. Certainly IPv6 <-> 6LoWPAN gateways are needed (border routers), but these still implement IPv6 all the way.

Unfortunately CoAP is not the language of the Internet, albeit very similar to HTTP which is. A gateway is therefore needed to connect the RESTful webservices on sensor nodes using CoAP with the webservices on the big Internet. This gateway will, however, be very simple compared to e.g. some ZigBee to IP translator. Just like 6LoWPAN is as close to IPv6 as reasonably possible for LLNs, CoAP is as close to HTTP as reasonably possible.

In [CoRE CoAP], the authors talk about CoAP-HTTP Proxying, and specify how it should be done. Essentially, a CoAP node, without HTTP, can send a CoAP request for a HTTP resource on a remote server using a proxy. The CoAP-HTTP proxy quite simply translates the CoAP request to an equivalent HTTP request and passes it on to the actual HTTP based node.

4.4 ZigBee IP (ZIP)

ZigBee is a well marketed protocol which, like 6LoWPAN, is mainly developed as a higher layer protocol on top of IEEE 802.15.4. Originally, ZigBee was a protocol family on layer three to five,[15] with additional profiles on top of that. The protocols specified adhoc routing (like RPL in 6LoWPAN), MAC layer mechanisms, and beacon based radio duty cycling (discussed in subsection 3.3.1). The protocol family was not based on any IETF standards, and fairly complex gateways were thus needed for connecting ZigBee networks to the Internet. This in turn resulted in that no end-to-end security or web based interfaces could be directly used.

Now, however, the ZigBee organization has also reached the conclusion that IP (IPv6) is the way to go. This has resulted in the *ZigBee IP Specification* (ZIP) by the ZigBee Alliance [ZigBee IP]. ZIP uses 6LoWPAN, RPL, and other familiar protocols defined by the IETF in its RFC documents. So ZIP is not much different from Contiki with its uIP stack, and ZIP could perhaps be implemented in Contiki. Currently though, at least the beacon mode is not implemented in Contiki, and this makes things more difficult as Contiki and ZIP uses different RDC mechanisms.

So ZIP is something quite different from the former ZigBee protocol, and it precisely lists the IETF standards that devices shall support. As such, it is quite useful in allowing multivendor devices to communicate with each other. Without strict standards there would be too many different configurations that 6LoWPAN based nodes could use.

The ZIP layers are illustrated in Figure 4.5; all of the protocols are IETF or IEEE standards, except for the *Smart Energy Profile* (SEP2) which currently is the only ZigBee profile built on ZIP (introduced in section 4.5). The layer directly below SEP2 is called the session layer, and it handles security, service discovery, and *Mesh Link Establishments* (MLE). Security is obtained using the TLS and PANA protocols[16]

For service discovery[17] the *DNS-Based Service Discovery* (DNS-SD) and the *Multicast DNS* (mDNS) protocols, specified in [RFC 6763] and [RFC 6762] respectively, shall be supported by ZIP nodes (there is some existing work on implementing these protocols

[14] **KNX** is a popular protocol for home automation.

[15] Based on the 5-layer TCP/IP protocol suite, layer **three** corresponds to the network layer and layer **five** to the application layer [Kurose 2010].

[16] **PANA** is an IETF protocol for authenticating clients for Internet/intranet access, it falls out of the scope of this document

[17] The terms **service discovery** and resource discovery are often used interchangeably

in Contiki e.g. [Klauck 2012]). These protocols are used together with e.g. the resource discovery mechanisms defined in SEP2, a bit but more about this in section 4.5 (the DNS-SD and mDNS protocols are not covered in this text).

The Mesh Link Establishment (MLE) protocol is used for establishing and configuring links in the ad hoc mesh network. The MLE is actually a layer two protocol, but MLE messages are send using UDP which resides on layer four. This partly makes MLE a layer five protocol, as in the figure [MLE].

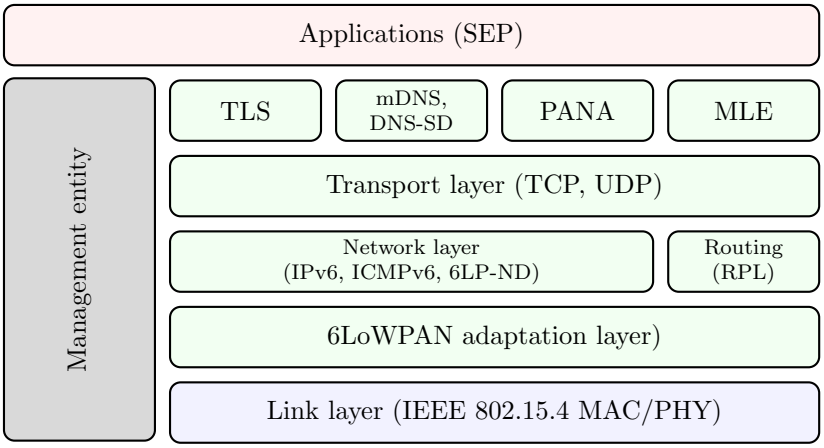

Figure 4.5: ZigBee protocol layers, adapted from ZigBee Alliance [ZigBee IP].

4.5 ZigBee Smart Energy Profile 2 (SEP2)

ZIP provides for secure (TLS, PANA), and reliable (if needed, with TCP) transport of data over LLNs. It also provides for some resource discovery and network configuration using mDNS and DNS-SD, but this is not enough for building IoT applications. One or many upper application layer protocols or profiles that e.g. state what kind of resources shall exist (like in the IPSO or IETF proposed standards we touched upon earlier) are still needed. This is where SEP2 comes in, and yes, it follows a RESTful architecture and is thus built around the core actions of GET, HEAD, PUT, POST, and DELETE. The profile also includes a lightweight subscription mechanism presented in subsection 4.5.2 [ZigBee SEP2].

As in general with REST, any application protocol that can implement a RESTful command set could be used underneath SEP2, but SEP2 actually requires HTTP. Usually SEP2 would sit on top of ZIP, at least in in LLNs based on 802.15.4, but it should be possibly to utilize SEP2 in any any device with an IP stack (including other protocols like DNS-SD etc.).

4.5.1 Resources and Resource Discovery

The SEP2 recommended URI structures and HTTP methods associated with these objects are defined in the SEP2 WADL.[18] SEP2 specifies a number of default function sets, resources, and their sample (recommended) URIs, of which some examples are listed in Table 4.4. As presented earlier in subsection 4.2.1, further function sets are proposed by IPSO, and those are intended to be complementary to the function sets defined in SEP2.

[18]The *Web Application Description Language* (**WADL**) is a machine processable format for describing the interface of REST systems

Table 4.4: Some SEP2 function sets and recommended root paths

Function set	Recommended root path
Device Capability	`/dcap`
Power Status	`/edev/{id1}/ps`
Network Status	`/edev/{id1}/ns`
Metering	`/upt`

The Device Capability function set contains one resource, named `DeviceCapability` (at the base uri of the function set: `/dcap`), which enumerates the function sets supported by a device and can be used by clients to discover the location of available function sets. This is quite similar to the `/.well-known/core` resource used in the CoRE Interfaces.

The Power Status function set also contains only one resource, `PowerStatus` at the base URI of the function set, which provides information about a device's power source and possible battery status.

The Network status function set contains a number of resources that contain information about a nodes IP addresses, RPL status, neighbours etc. For example, the `IPAddr` resource at `/edev/{id1}/ns/{id2}/addr/{id3}` which is a list of IP addresses, or the `NeighborList` at uri: `/edev/{id1}/ns/{id2}/ll/{id3}/nbh` which contains a list of a nodes 802.15.4 neighbours.

The Metering function set contains contains e.g. the `MeterReadingList` resource at `/upt/id1/mr` which contains URIs for each meter reading resource, so that GETing this URI gives links to available `MeterReading` resources.

It must be noted that devices shall not assume that these URIs and their structures are actually used, they are just recommendations. All resources shall be self-describing as it is expected that resources and URIs will change in the future.

All resources shall also contain links to their subordinate resources to allow for discovery and extensibility. This means that actual URIs, presently used by a node, must be retrieved through resource discovery and links within the resources. For network efficiency, devices may assume some URIs exist on a server, but if a URI returns an unexpected result, the client should execute resource discovery to determine the new URI value. All devices must be prepared for changes in other nodes and nothing shall be taken for granted.

The text above, paraphrased from the SEP 2 specification, could have been taken from a description of the REST architecture, as this, in all essence, is REST. Using this approach, RESTful systems are supposed to live and scale for decades.

As previously presented, SEP2 uses DNS-based methods for service discovery, resource discovery, and hostname to IP address resolution; and the process for discovering a resource is summarized below [ZigBee SEP2]:

1. Use mDNS/DNS-SD to locate the servers with the function sets of interest.
2. For each server do the following:
   (a) Establish TLS session if required
   (b) GET the `DeviceCapabilities` resource
   (c) Look for the desired function set in the `DeviceCapabilities` resource.
   (d) If there is an entry in `DeviceCapabilities`, it will contain a URI of the entry point for that function set.

3. Determine which of the discovered resources are of interest. This depends on the resource and other outside factors.

4.5.2 Subscription/Notification Mechanism

CoAP includes a subscription mechanism, while HTTP does not. ZIP does not add anything of the sort either, but SEP2 does introduce an optional possibility for devices to subscribe to changes in resources on others, which is interesting to compare to the CoAP mechanism.

Ech node that has a subscribable resource presents a **SubscriptionList Resource**. The recommended URI for this resource is: `/edev/{id1}/sub`. And it has sub-resources for each active subscription, that is the actual **Subscription Resources** e.g `/edev/{id1}/sub/{id2}`. A client wishing to subscribe to changes of some resource on a node would then POST an XML document to the SubscriptionList Resource URI. An example, from [ZigBee SEP2], is given below:

Subscription example

```
POST /edev/8/sub HTTP/1.1
Host: {hostname}
Content-Type: application/sep+xml
Content-Length: {contentLength}
<Subscription xmlns="http://zigbee.org/sep">
<subscribedResource>/upt/0/mr/4/r</subscribedResource>
<Condition>
<attributeIdentifier>0</attributeIdentifier>
<lowerThreshold>10</lowerThreshold>
<upperThreshold>1000</upperThreshold>
</Condition>
<encoding>0</encoding>
<level>+S0</level>
<limit>1</limit>
<notificationURI>/note</notificationURI>
</Subscription>
```

Here a client subscribes to the resource at `/upt/0/mr/4/r` on the server. There are also some subscription parameters included in the post, e.g. the lower and upper threshold the client wishes to set for the subscription, i.e. the client here wants to be notified when the representation of the resource drops below 10 or rises over 1000. Remember that [CoRE Interface] proposed query parameters for setting these parameters for a resource. Note that here the parameters are set to a subscription object, not the resource itself, i.e. each subscriber can set the parameters differently. If the actual resource is configured, as in the CoRE approach, all subscribers would get the same subscription settings.

The SEP2 server will notify the subscribing client by POSTing to its **Notification Resource**. In the example, this is found at the URI `/note` on the subscriber as seen in the XML document above. An active subscription is exposed in the REST API both at the client and the server device.

In the SEP2 documentation, it is stated that battery-powered sleepy devices should use a pull mechanism (the client polls the server) instead of the subscription/notification mechanism, as they say it may not be reliable, but it is not explained why this is.

5

Contiki

Contiki is an open source *Operating System* (OS) with an emphasis on connecting memory constrained devices to the IoT. It was created by Adan Dunkels in 2002, and it has since been further developed by people and companies from around the world. The small memory footprint is made possible by executing processes[19] using a light-weight multithreading[20] mechanism where every thread[21] shares a single stack.[22]

Contiki supports network communication based upon IPv4, IPv6, and its own lightweight communication protocol RIME. Furthermore, the IPv6 implementation also includes support for 6LowPAN fragmentation, 6LowPAN header compression, and RPL. Implementations of security mechanisms like IPSec with rudimentary IKEv2 key exchange [Raza 2010; Raza 2012] and DTLS exist, but these are not yet part of the Contiki main code base. Today, the most recent version of Contiki is 2.7 (released on the 15th of November, 2013), and a list of supported hardware can be found in [Hardware].

This chapter will provide information on how to get started, how Contiki works, and how to configure Contiki in light of the previously presented material.

5.1 Getting started

Contiki can be freely downloaded from its official git repository on GitHub, and it comes with several examples to get started. Each example includes a general makefile for compiling the code, and one or a few c-files containing the example process. However, before taking a closer look at these examples, it is a good idea to look closer at how Contiki handles processes and process scheduling.

[19] **A process** is a collection of threads that together constitute an application [Stallings 2009].

[20] **Multithreading** is a technique in which a process, executing an application, is divided into threads that can run concurrently [Stallings 2009].

[21] **A thread** is an interruptible unit of code [Stallings 2009].

[22] **A stack** is an abstract memory type, similar to heap of paper, that operates according to the last-in-first-out principle [Stallings 2009].

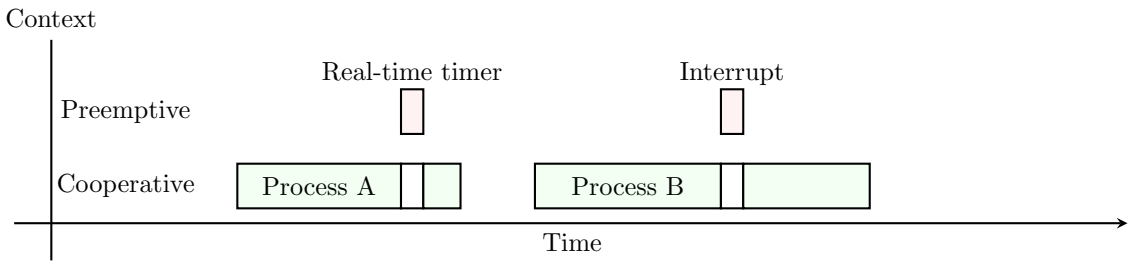

Figure 5.1: Processes in Contiki.

5.2 Processes and process scheduling

Contiki supports multitasking by running code in either **preemptive mode** or **cooperative mode** , see Figure 5.1. Preemptive mode essentially means that the kernel[23] can force a process to be interrupted (to yield), whereas in cooperative mode each each process is responsible for periodically yielding back control to the kernel [Stallings 2009]. Cooperative processes are, in Contiki, realized with an abstract structure called **protothreads**. A complete explanation of these are out of scope here but an excellent explanation can be found in [Protothread]. It can, however, shortly be mentioned that these protothreads make extensive use of macros. Hence, processes within Contiki also rely on macros to a great extent; this is best illustrated with an example process.

Example process

```
1  #include "contiki.h"
2
3  PROCESS(example_process, "Example process");
4  AUTOSTART_PROCESSES(&example_process);
5
6  PROCESS_THREAD(example_process, ev, data)
7  {
8
9    /* Code here is always executed */
10
11   PROCESS_BEGIN();
12
13   /* Code here is only executed if the process
14      has not yielded */
15
16   while(1) {
17
18     PROCESS_WAIT_EVENT();
19
20     printf("Got event number %d\n", ev);
21   }
22
23   PROCESS_END();
24 }
```

In the above example, needed Contiki functions and definitions are initially included on line 1. The process itself is declared on line 3 using the PROCESS macro, and it

[23]The **kernel** is here defined to be a computer program that manages hardware resources between different processes.

is further added to the kernel's list of processes to start scheduling on boot using the AUTOSTART_PROCESSES macro on line 4. The definition of the process (or thread) is done within the PROCESS_THREAD block. A running process will execute the code within PROCESS_THREAD, and this region is required to include at least the macros PROCESS_BEGIN and PROCESS_END.

Expansion of the different protothread macros (not shown) reveals that PROCESS_THREAD is just a function that the kernel scheduler (basically a while loop) repeatedly calls when e.g. an event[24] has been posted to the process. A similar expansion of the PROCESS_BEGIN and PROCESS_END macros reveal that these are nothing more than a switch case statement that is used in an intriguing way[25] to do jumps into a line of code between the PROCESS_BEGIN and PROCESS_END statements, where the process previously yielded. Macros like PROCESS_WAIT_EVENT then, finally, mark out where in the code the process will yield control back to the scheduler. More specifically, this macro actually performs a return from the function, but by first taking note of the state, i.e. the current line in the code.

When the example process above is executed for the first time, each line of code will be processed all the way to PROCESS_WAIT_EVENT, where control is returned to the kernel (storing the state). When an event later occurs, the process is invoked again, but this time only the code up to PROCESS_BEGIN is executed, whereupon the PROCESS_BEGIN (the switch statement) does a jump back to the PROCESS_WAIT_EVENT line. At this point, the printf is executed and the while loop does its second iteration, leading up to the PROCESS_WAIT_EVENT with the resulting return of control to the scheduler again.

In short, a protothread is then simply a function, repeatedly called by the scheduler that can remember the location in the code where it returned from in the previous cycle. As each process essentially is a protothread (function), process variables will not be remembered between successive calls. Hence, all variables that need to retain their value between calls should be defined as static.

For a more comprehensive review of Contiki processes, please see [Processes].

5.3 Building (Compiling) Contiki

Contiki is developed under Linux and the buildsystem is based solely on GNU make[26] (no other tools like gnu autotools, cmake etc. are used), but it is quite advanced. A simple way of getting started is to download Instant Contiki, which is a Ubuntu Linux virtual machine (that can e.g. be run from a Windows machine) with all required tools for building Contiki installed, though it is usually no problem to install the tools to a running Linux system either.

Contiki comes with a bunch of examples, all of which include a Makefile that, in turn, includes the Makefile.include file in the Contiki root folder. As an example, the hello-world example provides a generic Makefile which look like:

[24] **Events** are identified with identifications numbers. Numbers below 127 can be freely used for user applications, whereas the numbers above 128 are reserved for kernel operations.

[25] the **switch** does jumps into the while loop, which is quite special but allowed by most compilers

[26] Background knowledge with **make** and Makefiles is very useful, but it is possible to still get away with minimal knowledge

Generic Makefile

```
# Project name
CONTIKI_PROJECT = hello-world
all: $(CONTIKI_PROJECT)

# Path to the contiki root folder
CONTIKI = ../..
include $(CONTIKI)/Makefile.include
```

Writing make in the console window, when located in the same folder as the Makefile, initializes the build (compiles and links all required files in the example as well as Contiki itself). This, however, compiles for a native platform (a Linux PC) as no target[27] was provided. To provide a target, in this case avr-zigbit, write:

```
/contiki/examples/hello-world make TARGET = avr-zigbit
```

The Makefile can be further customized to handle both compilation and programming of the microcontroller; an example of a more comprehensive Makefile can be found in Appendix A.

5.4 The Contiki tree

Structurally, the Contiki tree is organized so that each platform folder always includes a configuration file and main file. These fulfil the following purposes:

Conf file	Specification of platform specific drivers, communication settings and modules, and memory usage. This file is normally named contiki-conf.h.
Main file	Contains the c-main function running the process scheduler, and an init function for initializing selected drivers, and communication modules.

Platform specific drivers are located under the CPU folder, whereas core Contiki functions are found inside the core folder. The hierarchy states that flags (macros) affecting the configuration file can be set already in the makefile, whereas all macros affecting driver selection and network configuration should be restricted to the configuration file. To separate between configurations macros and macros used within drivers and modules, the macros assigned in the configuration file or makefile will have the notation conf added to their names. As an example, usage of the IPv6 protocol is made possible by setting the macro UIP_CONF_IPV6 in the makefile (CFLAGS += -DUIP_CONF_IPV6). As can be seen below, this further sets macros in contiki-conf.h for including ICMPv6, TCP, and UDP in the project.

contiki-conf.c (avr-zigbit)

```
#if UIP_CONF_IPV6
#define UIP_CONF_ICMP6          1
```

[27] **Target** is equivalent to platform, and available targets are therefore the folder names found inside the platform folder.

```
#define UIP_CONF_UDP              1
#define UIP_CONF_TCP              1
#endif
```

For other examples where the macro sets a parameter used by functions in loaded modules, the macro set in the configuration file have a name identical to the parameter except for the conf addition. This can e.g. be illustrated by the macro NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE that determines the sleep cycle for ContikiMAC. From contiki-conf.h, netstack.h, and contikimac.c, the following code snippets can be found:

contiki-conf.c (avr-zigbit)

```
/* Default is two CCA separated by 125 usec */
#define NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE   8
```

netstack.h

```
#ifndef NETSTACK_RDC_CHANNEL_CHECK_RATE
#ifdef NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE
#define NETSTACK_RDC_CHANNEL_CHECK_RATE
NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE
#else /* NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE */
#define NETSTACK_RDC_CHANNEL_CHECK_RATE 8
#endif /* NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE */
#endif /* NETSTACK_RDC_CHANNEL_CHECK_RATE */
```

contikimac.c

```
/* CYCLE_TIME for channel cca checks, in rtimer ticks. */
#ifdef CONTIKIMAC_CONF_CYCLE_TIME
#define CYCLE_TIME (CONTIKIMAC_CONF_CYCLE_TIME)
#else
#define CYCLE_TIME
(RTIMER_ARCH_SECOND / NETSTACK_RDC_CHANNEL_CHECK_RATE)
#endif
```

Hence, the macro NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE, set in contiki-conf.h, is used in netstack.h to set the macro NETSTACK_RDC_CHANNEL_CHECK_RATE. If the previous macro is not defined in contiki-conf.h, a default value will be chosen instead (in this case 8). In contikimac.c, the defined macro from netstack.h is further used to determine CYCLE_TIME that finally defines the sleep interval in ticks. A similar tracing from a definition, in contiki-conf.h, to a parameter value in a function can be done for all macros, and Appendix B explains how a wide variety of macros set in contiki-conf.h relate to the material presented in previous chapters.

References

URLs have been provided for all references that are freely accessible online. These might change over time, but all were valid when tested in March 2014.

[6LoWPAN] S. Raza, S. Duquennoy, and G. Selander. *Compression of IPsec AH and ESP Headers for Constrained Environments draft-raza-6lowpan-ipsec-01*. Internet-Draft. 2013. URL: http://tools.ietf.org/html/draft-raza-6lowpan-ipsec-01.

[Castillo 2011] Pedro A Castillo et al. *SOAP vs REST: Comparing a master-slave GA implementation*. 2011. arXiv: 1105.4978 [cs.NE]. URL: http://arxiv.org/abs/1105.4978.

[Chui 2010] Michael Chui, Markus Löffler, and Roger Roberts. "The Internet of Things". In: *McKinsey Quartley* (Mar. 2010). URL: http://www.mckinsey.com/insights/high_tech_telecoms_internet/the_internet_of_things (visited on 10/09/2013).

[Cisco] Cisco. *The Internet of Things*. Infographic. URL: http://share.cisco.com/internet-of-things.html (visited on 10/09/2013).

[Colitti 2011] Walter Colitti, Kris Steenhaut, and Niccolò De Caro. "Integrating Wireless Sensor Networks with the Web". In: *Extending the Internet to Low powerand Lossy Networks (IP+ SN 2011)*. 2011. URL: http://hinrg.cs.jhu.edu/joomla/images/stories/IPSN_2011_koliti.pdf.

[CoRE CoAP] Z. Shelby et al. *Constrained Application Protocol (CoAP) draft-ietf-core-coap-18*. Internet-Draft. 2013. URL: https://ietf.org/doc/draft-ietf-core-coap/.

[CoRE Interface] Z. Shelby and M.V Vial. *CoRE Interfaces draft-ietf-core-interfaces-01*. Internet-Draft. 2013. URL: http://tools.ietf.org/html/draft-ietf-core-interfaces-01.

[CoRE OR] K Hartke. *Observing Resources in CoAP draft-ietf-core-observer-08*. Internet-Draft. 2014. URL: http://tools.ietf.org/pdf/draft-ietf-core-observe-12.pdf.

[CoRE RD] Z. Shelby and S. Krco. *CoRE Resource Directory draft-ietf-core-resource-directory-01*. Internet-Draft. 2013. URL: http://tools.ietf.org/html/draft-ietf-core-resource-directory-01.

[Dunkels 2011] Adam Dunkels. *The ContikiMAC Radio Duty Cycling Protocol*. Tech. rep. T2011:13. Swedish Institute of Computer Science, 2011. URL: http://dunkels.com/adam/dunkels11contikimac.pdf.

[Dunkels 2014] Adam Dunkels. *Information on ip64*. Contiki-Developers forum comment. Feb. 14, 2014. URL: http://sourceforge.net/p/contiki/mailman/message/31940441/.

[Fielding 2000] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, 2000. URL: http://jpkc.fudan.edu.cn/picture/article/216/35/4b/22598d594e3d93239700ce79bce1/7ed3ec2a-03c2-49cb-8bf8-5a90ea42f523.pdf.

[Hardware] Thingsquare. *Contiki Hardware*. Contiki webpage. URL: http://www.contiki-os.org/hardware.html.

[IANA ICMPv6] Internet Assigned Numbers Authority. *Internet Control Message Protocol version 6 (ICMPv6) Parameters*. URL: http://www.iana.org/assignments/icmpv6-parameters/icmpv6-parameters.xml.

[IEEE 802.11] IEEE. *IEEE Std 802.11 TM 2007*. IEEE standard. 2007. URL: http://standards.ieee.org/getieee802/download/802.11-2007.pdf.

[IEEE 802.15.4] IEEE. *IEEE Std 802.15.4 TM 2011*. IEEE standard. 2011. URL: http://standards.ieee.org/getieee802/download/802.15.4-2011.pdf.

[Intel 2009] Intel. *Rise of the Embedded Internet*. White paper. intel, 2009. URL: http://download.intel.com/newsroom/kits/embedded/pdfs/ECG_WhitePaper.pdf.

[IPSO] Z. Shelby and C. Chauvenet. *The IPSO Application Framework draft-ipso-app-framework-04*. IPSO draft. 2012. URL: http://www.ipso-alliance.org/wp-content/media/draft-ipso-app-framework-04.pdf.

[Klauck 2012] Ronny Klauck and Michael Kirsche. "Bonjour Contiki: A Case Study of a DNS-Based Discovery Service for the Internet of Things". In: *Ad-hoc, Mobile, and Wireless Networks*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 316–329. URL: http://dx.doi.org/10.1007/978-3-642-31638-8_24.

[Kurose 2010] James F. Kurose and Keith W. Ross. *Computer Networking a top-down approach, 5/E*. Pearson Education, 2010.

[Microsoft 2003] Microsoft. *Teredo Overview*. 2003. URL: http://technet.microsoft.com/en-us/library/bb457011.aspx (visited on 11/01/2013).

[MLE] R. Kelsey. *Mesh Link Establishment draft-kelsey-intarea-mesh-link-establishment-05*. Internet-Draft. 2013. URL: http://tools.ietf.org/html/draft-kelsey-intarea-mesh-link-establishment-05.

[Pötsch 2011] Thomas Pötsch. *Performance of the Constrained Application Protocol for Wireless Sensor Networks*. Presentation. 2011. URL: http://www.comnets.uni-bremen.de/itg/itgfg521/aktuelles/fg-workshop-29092011/ITG_HH_thomas_poetsch.pdf.

[Prelman 2012] Vladislav Prelman. "Security in IPv6-enabled Wireless Sensor Networks: An Implementation of TLS/DTLS for he Contiki Operating System". Master thesis. Jacobs University, 2012. URL: http://cnds.eecs.jacobs-university.de/archive/msc-2012-vperelman.pdf.

[Processes] Ronnie Bajwa. *Processes*. Contiki GitHub wiki. URL: https://github.com/contiki-os/contiki/wiki/Processes.

[Protothread] Adam Dunkels. *How protothreads really work*. Homepage. URL: http://dunkels.com/adam/pt/expansion.html.

[Ramos 2008] Carlos Ramos, Juan Carlos Augusto, and Daniel Shapiro. "Ambient intelligence—The next step for artificial intelligence". In: *Intelligent Systems, IEEE* 23.2 (2008), pp. 15–18.

[Raza 2010] Shahid Raza. *Securing Internet of Things with Lightweight IPsec*. Tech. rep. T2010:08. Swedish Institute of Computer Science, 2010. URL: http://soda.swedish-ict.se/4052/2/reportRevised.pdf.

[Raza 2012] Shahid Raza, Thiemo Voigt, and Vilhelm Jutvik. "Lightweight IKEv2: A Key Management Solution for both Compressed IPsec and IEEE 802.15.4 Security". In: *Proceedings of the IETF Workshop on Smart Object Security*. 2012.

[Raza 2013] S. Raza et al. "Lithe: Lightweight Secure CoAP for the Internet of Things". In: *Sensors Journal, IEEE* 13.10 (2013), pp. 3711–3720.

[RFC 2460] S. Deering and R. Hinden. *RFC 2460 Internet Protocol, Version 6 (IPv6) Specification*. RFC. 1998. URL: http://tools.ietf.org/html/rfc2460.

[RFC 2710] S. Deering, W. Fenner, and B. Haberman. *Multicast Listener Discovery (MLD) for IPv6*. RFC. 1999. URL: http://tools.ietf.org/html/rfc2710.

[RFC 3056] B. Carpenter and K. Moore. *Connection of IPv6 Domains via IPv4 Clouds*. RFC. 2001. URL: http://tools.ietf.org/html/rfc3056.

[RFC 3068] C. Huitema. *An Anycast Prefix for 6to4 Relay Routers*. RFC. 2001. URL: http://tools.ietf.org/html/rfc3068.

[RFC 3315] J. Bound et al. *Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*. RFC. 2003. URL: http://tools.ietf.org/html/rfc3315.

[RFC 3972] T. Aura. *Cryptographically Generated Addresses (CGA)*. RFC. 2005. URL: http://tools.ietf.org/html/rfc3972.

[RFC 4213] E. Nordmark and R. Gilligan. *Basic Transition Mechanisms for IPv6 Hosts and Routers*. RFC. 2005. URL: http://tools.ietf.org/html/rfc4213.

[RFC 4291] R. Hinden and S. Deering. *RFC 4291 IP Version 6 Addressing Architecture*. RFC. 2006. URL: http://tools.ietf.org/html/rfc4291.

[RFC 4301] S. Kent and K. Seo. *RFC 4301 Security Architecture for the Internet Protocol*. RFC. 2005. URL: http://tools.ietf.org/html/rfc4301.

[RFC 4380] C. Huitema. *Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs)*. RFC. 2006. URL: http://tools.ietf.org/html/rfc4380.

[RFC 4443] A. Conta, S. Deering, and M. Gupta. *RFC 4443 Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. RFC. 2006. URL: http://tools.ietf.org/html/rfc4443.

[RFC 4861] T. Narten et al. *RFC 4861 Neighbor Discovery for IP version 6 (IPv6)*. RFC. 2007. URL: http://tools.ietf.org/html/rfc4861.

[RFC 4862] S. Thomson, T. Narten, and T. Jinmei. *RFC 4862 IPv6 Stateless Address Autoconfiguration*. RFC. 2007. URL: http://tools.ietf.org/html/rfc4862.

[RFC 4941] T. Narten, R. Draves, and S. Krishnan. *RFC 4941 Privacy Extensions for Stateless Address Autoconfiguration in IPv6*. RFC. 2007. URL: http://tools.ietf.org/html/rfc4941.

[RFC 4944] G. Montenegro et al. *RFC 4944 Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC. 2007. URL: http://tools.ietf.org/html/rfc4944.

[RFC 5246] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC. 2008. URL: http://tools.ietf.org/html/rfc5246.

[RFC 5996] C. Kaufman et al. *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC. 2010. URL: http://tools.ietf.org/html/rfc5996.

[RFC 6146] M. Bagnulo, P. Matthews, and I. van Beijnum. *Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers*. RFC. 2011. URL: http://tools.ietf.org/html/rfc6146.

[RFC 6206] P. Levis et al. *The Trickle Algorithm*. RFC. 2011. URL: http://tools.ietf.org/html/rfc6206.

[RFC 6282] J. Hui and P. Thubert. *RFC 6282 Compression Format for IPv6 Datagram over IEEE 802.15.4-Based Networks*. RFC. 2011. URL: http://tools.ietf.org/html/rfc6282.

[RFC 6347] E. Rescorla and N. Modadugu. *Datagram Trasnport Layer Security Version 1.2*. RFC. 2012. URL: http://tools.ietf.org/html/rfc6347.

[RFC 6550] T. Winter et al. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC. 2012. URL: http://tools.ietf.org/html/rfc6550.

[RFC 6551] JP. Vasseur et al. *Routing Metrics Used for Path Calculation in Low-Power and Lossy Networks*. RFC. 2012. URL: http://tools.ietf.org/html/rfc6551.

[RFC 6552] P. Thubert. *Objective Function Zero for the Routing Protocol for Low-Power and Lossy Networks (RPL)*. RFC. 2012. URL: http://tools.ietf.org/html/rfc6552.

[RFC 6553] J. Hui and JP. Vasseur. *The Routing Protocol for Low-Power and Lossy Networks (RPL) Option for Carrying RPL Information in Data-Plane Datagrams*. RFC. 2012. URL: http://tools.ietf.org/html/rfc6553.

[RFC 6554] J. Hui et al. *An IPv6 Routing Header for Source Routes with the Routing Protocol for Low-Power and Lossy Networks (RPL)*. RFC. 2012. URL: http://tools.ietf.org/html/rfc6554.

[RFC 6690] Z. Shelby. *Constrained RESTful Environments (CoRE) Link Format*. RFC. 2012. URL: http://tools.ietf.org/html/rfc6690.

[RFC 6719] O. Gnawali and P. Levis. *The Minimum Rank with Hysteresis Objective Function*. RFC. 2012. URL: http://tools.ietf.org/html/rfc6719.

[RFC 6762] S. Cheshire and M. Krochmal. *Multicast DNS*. RFC. 2013. URL: http://tools.ietf.org/html/rfc6762.

[RFC 6763] Cheshire S. and Krochmal M. *DNS-Based Service Discovery*. RFC. 2013. URL: http://datatracker.ietf.org/doc/rfc6763/.

[RFC 6775] Z. Shelby et al. *Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)*. RFC. 2012. URL: http://tools.ietf.org/html/rfc6775.

[Shelby 2009] Zach Shelby and Carsten Bormann. *6LoWPAN, The Wireless Embedded Internet*. Wiley, 2009.

[Stallings 2009] William Stallings. *Operating Systems: Internals and Design Principles, 6/E*. Pearson Education India, 2009.

[Vasseur 2010] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting Smart Objects with IP*. Elsevier, 2010.

[ZigBee IP] ZigBee Alliance. *ZigBee IP Specification*. Tech. rep. 13-002r00. ZigBee Alliance, 2013. URL: http://www.zigbee.org/Specifications/ZigBeeIP/Download.aspx.

[ZigBee SEP2] ZigBee Alliance. *Smart Energy Profile 2 Application Protocol Standard*. Tech. rep. 13-0200-00. ZigBee Alliance, 2013. URL: http://www.csee.org.cn/Portal/zh-cn/Publications/atm/docs-13-0200-00-sep2-smart-energy-profile-2.pdf.pdf.

Index

A Makefile (AVR)

Example makefile for compiling and flashing the Contiki hello world example to an AVR zigbit module.

Makefile

```
# Default platform
ifndef TARGET
TARGET = avr-zigbit
endif

# Used MCU
MCU = atmega1281v

# Project name
CONTIKI_PROJECT = hello-world

# Including IPv6
CFLAGS+= -DUIP_CONF_IPV6

# Name of created file
OUTFILE = hello_world.avr-zigbit

# Compiling
all: $(CONTIKI_PROJECT)
        avr-objcopy -O ihex -R .eeprom -R .fuse
        -R .signature $(OUTFILE) $(OUTFILE).hex
        avr-size -C --mcu=$(MCU) $(OUTFILE)

# Flash program to MCU
flash:
        avrdude -c dragon_jtag -P usb -p $(MCU)
        -U flash:w:$(OUTFILE).hex

# Reading fuses
# extendend
read_efuse:
        avrdude -c dragon_jtag -P usb -p $(MCU)
        -U efuse:r:efuse.hex:i
# high
read_hfuse:
        avrdude -c dragon_jtag -P usb -p $(MCU)
        -U hfuse:r:hfuse.hex:i
# low
read_lfuse:
        avrdude -c dragon_jtag -P usb -p $(MCU)
        -U lfuse:r:lfuse.hex:i

# Writing fuses
# extendend
write_efuse:
        avrdude -c dragon_jtag -P usb -p $(MCU)
        -U efuse:w:0xFE:m
# high
write_hfuse:
```

```
50          avrdude -c dragon_jtag -P usb -p $(MCU)
51          -U hfuse:w:0x19:m
52  # low
53  write_lfuse:
54          avrdude -c dragon_jtag -P usb -p $(MCU)
55          -U lfuse:w:0xE2:m
56
57  # Read EEPROM
58  read_eeprom:
59          avrdude -c dragon_jtag -P usb -p $(MCU)
60          -U eeprom:r:eepromm.out:i
61
62  # Write EEPROM
63  write_eeprom:
64          avrdude -c dragon_jtag -P usb -p $(MCU)
65          s-U eeprom:w:eepromm.out:i
66
67  # Contiki's main directory
68  CONTIKI = ../..
69  include $(CONTIKI)/Makefile.include
70
71  # Erase created files
72  clean:
73          rm -rf *.hex $(OUTFILE)
74          rm -rf obj_$(TARGET)
75          rm -rf contiki-$(TARGET).a
76          rm -rf contiki-$(TARGET).map
```

B Contiki macros

B.1 uIPv6

Below follows a summary of different macros used to configure IPv6 in Contiki. Mandatory settings for a proper IPv6 implementation are: UIP_CONF_IPV6 = 1, UIP_CONF_ICMP6 = 1, and UIP_CONF_LL_802154 = 1. UIP_CONF_IPV6 is preferably set from the makefile, whereas the others should be set in contiki-conf.h. Besides these mandatory macros, the list below contains additional configuration macros that can be used to constrain memory usage.

uIPv6 configuration macros

UIP_CONF_IPV6
Default value 0 (contiki-default-conf.h).
Specifies if IPv6 or IPv4 is used (0 = IPv4 / 1 = IPv6).

UIP_CONF_BUFFER_SIZE
Default value 128 (contiki-default-conf.h).
Determines how much memory that is reserved for the uIP packet buffer, and it hence puts an upper limit on the largest IP packet that can be received.

UIP_CONF_ICMP6
No default value (not included).
Specifies if ICMPv6 is included or not (0 = not included / 1 = included).

UIP_CONF_TCP
Default value 1 (contiki-default-conf.h).
Specifies if TCP is included or not (0 = not included / 1 = included).

UIP_CONF_MAX_CONNECTIONS
Default value 8 (contiki-default-conf.h).
Determines how many active TCP connections that can be operated concurrently. Each active connections requires an additional memory allocation of approximately 30 bytes.

UIP_CONF_MAX_LISTENPORTS
Default value 20 (uipopt.h).
Determines how many ports TCP can concurrently listen at. Each port requires 2 additional bytes of memory to be allocated.

UIP_CONF_UDP
Default value 1 (contiki-default-conf.h).
Specifies if UDP is included or not (0 = not included / 1 = included).

UIP_CONF_UDP_CONNS
Default value 10 (uipopt.h).
Determines the maximum number of concurrent UDP connections.

UIP_CONF_ROUTER
Default value 1 (contiki-default-conf.h).
Specifies if the node is a router or not (0 = not a router / 1 = router).

UIP_CONF_MAX_ROUTES
Default value 20 (contiki-default-conf.h).
Determines the maximum size of the routing table.

NBR_TABLE_CONF_MAX_NEIGHBORS
Default value 8 (contiki-default-conf.h).
Determines the maximum number of neighbours that each node can store in memory.

UIP_CONF_ND6_SEND_RA
Default value 1 (uip-nd6.h).
0 = disables / 1 = enables router advertisements.

UIP_CONF_ND6_SEND_NA
Default value 1 (uip-nd6.h).
0 = disables / 1 = enables neighbour advertisements.

UIP_CONF_ND6_MAX_UNICAST_SOLICIT
Default value 3 (uip-nd6.h).
Specifies the maximum number of neighbour solicitation messages that should be sent when a node performs NUD.

UIP_CONF_ND6_DEF_MAXDADNS
Default value 0 (uip-nd6.h).
Specifies the maximum number of neighbour solicitation messages that should be sent when a node performs DAD. This value can be 0 as section 8.2 in [RFC 6775] specifies that DAD is not needed if IPv6 addresses are derived from EUI-64 addresses.

UIP_CONF_LL_802154
No default value
Should be set to 1 in contiki-conf.h if 802.15.4 is used at the link layer. The macro informs uIPv6 that the length of the link layer address is 64 bits (Contiki does not yet support 16 bit 802.15.4 addresses).

B.2 6LoWPAN

6LowPAN is enabled by setting the NETSTACK_CONF_NETWORK macro to sicslowpan_driver in contiki-conf.h. This is the only mandatory setting, and the rest of the macros in the list below can be used to configure different 6LowPAN features.

6LoWPAN configuration macros

NETSTACK_CONF_NETWORK
Default selection rime_driver (contiki-default-conf.h).
Selection between RIME and 6LowPAN. For 6LowPAN, the correct driver is sicslowpan_driver.

SICSLOWPAN_CONF_FRAG

Default value 1 (contiki-default-conf.h).
0 = disables / 1 = enables 6LoWPAN fragmentation.

SICSLOWPAN_CONF_MAC_MAX_PAYLOAD
Default value 102 (contiki-default-conf.h).
Determines the maximum packet size in bytes before packets gets fragmented. The default value 102 is obtained as: the maximum frame size in 802.15.4 (127 bytes) minus the 25 bytes long MAC layer header.

SICSLOWPAN_CONF_COMPRESSION_THRESHOLD
Default value 0 (contiki-default-conf.h).
Adds a lower threshold for when packets should not be fragmented, it could for example be used to avoid getting packets that are smaller than the minimum level set by ContikiMAC.

SICSLOWPAN_CONF_MAX_MAC_TRANSMISSIONS
Default value 4 (contiki-default-conf.h).
Specifies how many times the MAC layer should resend packets if no link-layer ACK was received. This setting only makes sense with the csma_driver (see subsection B.5).

SICSLOWPAN_CONF_COMPRESSION
Default selection SICSLOWPAN_COMPRESSION_HC06 (contiki-default-conf.h).
Specifies if 6LoWPAN header compression is to be used. Alternatives are: SICSLOWPAN_COMPRESSION_IPV6, SICSLOWPAN_COMPRESSION_HC1, and SICSLOWPAN_COMPRESSION_HC06 (workin version for IPHC).

SICSLOWPAN_CONF_MAX_ADDR_CONTEXTS
Default value 1 (uipopt.h).
If IPHC is used, this macro defines for how many address contexts memory should be reserved.

SICSLOWPAN_CONF_ADDR_CONTEXT_0
Default prefix aaaa:0:0:0 (sicslowpan.c).
Assigns a network prefix for the first address context, if atleast one context is allowed. Assignments are done using:
#define SICSLOWPAN_CONF_ADDR_CONTEXT_0
{addr_contexts[0].prefix[0]=0xaa;addr_contexts[0].prefix[1]=0xaa;}

B.3 RPL

As IPv6, RPL should also be selected from the Makefile; this is done by setting the macro UIP_CONF_IPV6_RPL. The other macros can be changed in contiki-conf.h, but this is not normally necessary.

RPL configuration macros

UIP_CONF_IPV6_RPL
Default value 0 (contiki-default-conf.h).
0 = disables / 1 = enables RPL routing.

RPL_CONF_STATS
Default value 0 (rpl-conf.h).
0 = disables / 1 = enables RPL statistics.

RPL_CONF_OF
Default selection rpl_mrhof (rpl-conf.h).
Sets the OF to either rpl_of0 (OF0) or rpl_mrhof (MRHOF).

RPL_CONF_DEFAULT_INSTANCE
Default value 0x1e (rpl-conf.h).
Sets the Instance ID for the default DODAG.

RPL_CONF_LEAF_ONLY
Default value 0 (rpl-conf.h).
Determines if the node can join the DODAG as a normal node (0) or only as a leaf node (1).

RPL_CONF_DIO_INTERVAL_MIN
Default value 12 (rpl-conf.h).
Minimum DIO advertisement interval in ms calculated as 2^{MIN}. The default minimum interval is therefore 2^{12} ms which gives ≈ 4 s.

RPL_CONF_DIO_INTERVAL_DOUBLINGS
Default value 8 (rpl-conf.h).
Maximum DIO advertisement interval in ms calculated as $2^{\text{MIN+DOUBLINGS}}$.
Default values hence give 2^{12+8} which is ≈ 1048 s.

RPL_CONF_DIO_REDUNDANCY
Default value 10 (rpl-conf.h).
Defines the parameter k for the trikle algorithm in [RFC 6206].

RPL_CONF_GROUNDED
Default value 0 (rpl-conf.h).
Indicates if the DODAG is ground (0 = not grounded / 1 = grounded).

RPL_CONF_MIN_HOPRANKINCREASE
Default value 256 (rpl-conf.h).
Sets the DODAG parameter MIN_HOP_RANK_INCREASE.

RPL_CONF_MOP
Default selection RPL_MOP_STORING_NO_MULTICAST (rpl-private.h).
RPL mode of operation selection. Alternatives are RPL_MOP_NO_DOWNWARD_ROUTES, RPL_MOP_NON_STORING, RPL_MOP_STORING_NO_MULTICAST, and RPL_MOP_STORING_MULTICAST.

B.4 ContikiMAC and RDC

ContikiMAC is enabled by setting the macro NETSTACK_CONF_RDC to contikimac_driver. However, the implementation of ContikiMAC depends heavily on the properties of the used hardware. An oscilloscope is therefore necessary for both adjusting time parameters and for verifying that the implementation works as intended. On top of this, ContikiMAC will keep the MCU awake every 16th sleep interval. This eats up the battery for battery driven nodes but a possible fix is presented in Appendix C.

ContikiMAC configuration macros

NETSTACK_CONF_RDC
Default selection nullrdc_driver (contiki-default-conf.h).
Selects the RDC driver. For ContikiMAC, the correct driver is contikimac_driver.

NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE
Default value 8 (contiki-default-conf.h).
Specifies the wake up frequency for nodes implementing ContikiMAC.

CONTIKIMAC_CONF_WITH_PHASE_OPTIMIZATION
Default value 0 (contiki-default-conf.h).
0 = disables / 1 = enables phase optimization.

CONTIKIMAC_CONF_CCA_COUNT_MAX
Default value 2 (contikimac.c).
Specifies the number CCAs that a node performs, upon awakening, in order to check if anybody is sending anything.

CONTIKIMAC_CONF_CCA_CHECK_TIME
Default value RTIMER_ARCH_SECOND / 8192 (contikimac.c).
The time it takes to perform a CCA check. The value should be zero if the hardware blocks until a check is completed.

CONTIKIMAC_CONF_INTER_PACKET_INTERVAL
Default value RTIMER_ARCH_SECOND / 2500 (contikimac.c).
Specifies the time between successive packets when strobing.

RDC_CONF_HARDWARE_CSMA
Default value 0 (contikimac.c).
Indicates if software needs to implement CSMA (0) or if it can be done automatically by the hardware (1). AVR radios do this automatically and this value should therefore be 1.

RDC_CONF_HARDWARE_ACK
Default value 0 (contikimac.c).
Indicates if software needs to send acknowledgements explicitly (0) or if it can be done automatically by the hardware (1). AVR radios do this automatically and this value should therefore be 1.

RDC_CONF_MCU_SLEEP
Default value 0 (contikimac.c).
Determines if the MCU also will be put to sleep between the periodic ContikiMAC channel checks (0 no sleep / 1 = sleep). The standard implementation

in Contiki for this macro is no good as it will let the MCU stay awake every 16th cycle. A possible fix for this is presented in Appendix C.

B.5 Netstack

The netstack macros selects proper drivers for low level functions such as radio handling and link layer properties. In principle, these macros should include a way to set the used 802.15.4 channel, but they do not. Channel selection is unfortunately done in different ways for different platforms, but normally the default channel is set to be channel 26.

Link layer configuration macros

NETSTACK_CONF_RADIO
Default selection nullradio_driver (contiki-default-conf.h).
Selects radio driver. For AVR, the correct driver is rf230_driver.

NETSTACK_CONF_FRAMER
Default selection framer_nullmac (contiki-default-conf.h).
Selects the link-layer frame type. For 802.15.4, the correct framer is framer_802154.

NETSTACK_CONF_MAC
Default selection nullmac_driver (contiki-default-conf.h).
Selects the MAC driver. For CSMA, the correct driver is csma_driver.

QUEUEBUF_CONF_NUM
Default value 8 (contiki-default-conf.h).
Specifies the number of queue buffers.

C Sleeping from main

When ContikiMAC sets the MCU to sleep, it has no way of knowing if there are any other processes running. For this reason, it keeps the MCU awake every 16th interval so that other running processes will get time to complete. In order to fix the problem, the code that sleeps the MCU have to be moved to a position in the code where knowledge about other running processes exists. This information is found in the main while loop (the function process_run returns an integer indicating the number of currently running processes), and a solution is therefore to sleep the MCU from this file instead. This, however, requires changes in multiple places and these are summarized below.

Sleeping the MCU from the main function

```
// Changes to contikiMAC.c //
// Make ContikiMAC variables global
-static volatile unsigned char we_are_sending = 0;
-static volatile unsigned char radio_is_on = 0;
-static volatile rtimer_clock_t cycle_start;
+volatile unsigned char we_are_sending = 0;
+volatile unsigned char radio_is_on = 0;
+volatile unsigned char contikiMAC_ready = 0;
+volatile rtimer_clock_t cycle_start;
// Make this function global
-static void schedule_powercycle_fixed
  (struct rtimer *t, rtimer_clock_t fixed_time)
+void schedule_powercycle_fixed
  (struct rtimer *t, rtimer_clock_t fixed_time)
// Replace everything within #if RDC_CONF_MCU_SLEEP with
+if(!we_are_sending && !radio_is_on) {
+  schedule_powercycle_fixed(t, CYCLE_TIME + cycle_start);
+  contikiMAC_ready = 1;
+  PT_YIELD(&pt);
+}
// New function for scheduling an imediate rtimer interupt
+static void
+set_interrupt(void)
+{
+  if(contikimac_is_on) {
+    rtimer_reset(&rt, RTIMER_NOW() + 1, 1,
+      (void (*)(struct rtimer *, void *))powercycle, NULL);
+  }
+}
// Adding the above function to the contikimac_driver
const struct rdc_driver contikimac_driver = {
  "ContikiMAC",
  init,
+ set_interrupt,
  qsend_packet,
  qsend_list,
  input_packet,
  turn_on,
  turn_off,
  duty_cycle,
};

```

```
// Changes to rdc.h //
+void (*set_interrupt)(void);

// Changes to rtimer.c //
// Add function for resetting the rtimer
+int
+rtimer_reset(struct rtimer *rtimer, rtimer_clock_t time,
+       rtimer_clock_t duration,
+       rtimer_callback_t func, void *ptr)
+{
+  int first = 0;
+  rtimer->func = func;
+  rtimer->ptr = ptr;
+  rtimer->time = time;
+  rtimer_arch_schedule(time);
+   return RTIMER_OK;
+}

// Changes to the main file //
// Definitions and external variables
+#define CYCLE_TIME (RTIMER_ARCH_SECOND /
  NETSTACK_RDC_CHANNEL_CHECK_RATE)
+#include "sys/rtimer.h"
+#include "rtimer-arch.h"
+extern rtimer_clock_t cycle_start;
+extern unsigned char contikiMAC_ready;
// Additions to the main function
// Add before while(1)
+int nProcesses;
+short timeToSleep;
// Changes within while(1)
-process_run();
+nProcesses = process_run();
+#if RDC_CONF_MCU_SLEEP
+if (nProcesses == 0 && contikiMAC_ready){
+  timeToSleep = CYCLE_TIME - (RTIMER_NOW() - cycle_start);
+  if (timeToSleep > 0){
+    rtimer_arch_sleep(timeToSleep);
+  }
+  contikiMAC_ready = 0;
+  NETSTACK_RDC.set_interrupt();
+}
+#endif
```

NOVIA
UNIVERSITY OF APPLIED SCIENCES

Novia University of Applied Sciences is the largest Swedish-speaking UAS in Finland. Novia UAS has about 4000 students and a staff workforce of 360 people. Novia has five educational units or campuses in Vaasa (Seriegatan and Wolffskavägen), Jakobstad, Raseborg and Turku. High-class and state-of-the-art degree programs provide students with a proper platform for their future careers.

NOVIA UNIVERSITY OF APPLIED SCIENCES
Tel +358 (0)6 328 5000
Fax +358 (0)6 328 5110
www.novia.fi

ADMISSIONS OFFICE
PO BOX 6
FI-65201 Vaasa, Finland
Tel +358 (0)6 328 5055
Fax +358 (0)6 328 5117
admissions@novia.fi

ISBN 978-952-5839-98-2

Read our latest publication at www.novia.fi/FoU/publikation-och-produktion