Vrije Universiteit Brussel

FACULTEIT INGENIEURSWETENSCHAPPEN

# Introduction to wireless sensor networks with 6LoWPAN and Contiki

## Telecommunications IT-Networks

## Industrial Engineering


2015-09-10
Laurent Segers
Guest lecture taught at the Engineering School of Castres (ISIS) on June $30^{th}$ 2015, France

# Contents

# Abstract

Assisted ambient living plays a more important role in our modern society. One might think of monitoring the temperature, humidity, light intensity (etc.) and by taking the appropriate actions like venting, cooling, adapting the intensity of light (etc.) in order to optimize the living conditions inside buildings. In general, the monitoring and actuation happens through smart nodes which are connected to a central control system. In this course we will cover the basis of IP connectivity of smart embedded wireless devices with 6LoWPAN. 6LoWPAN (IPv6 for Low power Personal Area Network) enables the integration of wireless sensor network nodes into the regular IP networks, and is especially designed for low power and constraint devices which can possibly operate on battery. To facilitate the development of applications, some platforms and simulator tools have been made accessible, such as the Zolertia Z1 platform and the Cooja simulator. The Zolertia Z1 platform runs the Contiki operating system, which already integrates the 6LoWPAN stack. In this course, we will discover the Zolertia Z1 platform together with the basis functionalities of the Contiki operating system. In a first step, we will focus at the possibilities to enable the readings from sensors and the usage of actuators. In a second step, we will deploy a 6LoWPAN based network using the Cooja simulator, where the IP protocol plays an important role. Therefore, we will also verify our setup by monitoring the transmitted 6LoWPAN packets. In the last step, we will build a complete network using several Zolertia Z1 nodes, a border router and a remote application. Possibilities to monitor IP based packets back and forth the network will also be shown.

**Prerequisites**: basis in C/C++ and basis knowledge of IP-programming, basis of Linux (Ubuntu) and terminal commands.

**Technical requirements**: Internet connection, Linux Ubuntu 12.04 LTS (virtual machine) with the MSP430-GCC compiler installed /Instant Contiki Virtual Machine (http://www.contiki-os.org/download.html), Zolertia Z1 nodes (and USB-cables) if available.

# 1 Instant Contiki

The Instant Contiki virtual machine comes along with all the necessary tools for programming the Zolertia Z1 motes with the Contiki operating system. Also pre-installed in the Instant Contiki machine are the Cooja wireless network simulator and Wireshark network packet dissector. The virtual machine can be run by using either "Vmware Player" or "VirtualBox". The login password is "user". While running the virtual machine, one can see the appropriate icons (Wireshark and Cooja)on the desktop. The Contiki operating is copied into the home folder of the machine. There are 2 versions of Contiki available in the folders: ↪*Contiki* and ↪*Contiki-2.7*. In this course, we will use the ↪*Contiki* folder for development within the Cooja simulator, while the other folder will be used to program real motes. Each of the folders has some relevant subfolders for this course:

- ↪*tools*: This folder contains the tunslip6 application. This application will be used to bridge the network created in the simulator (real 6LoWPAN network) and applications residing on a remote application. Before using this application it needs to be compiled.
- ↪*examples*: This folder contains some examples from which one can start develop applications. We will start with the ↪*hello-world* subfolder.
- ↪*platform/z1/dev*: The folder where additional drivers can be written for devices not currently supported by Contiki. Some drivers are already provided by Contiki by default. Precautions must be taken while providing a new driver. The latter is especially the case when using interrupt routines, timers and other resources which can be used only once on the mote.
- ↪*platform/z1*: This folder contains the makefile "Makefile.common" which allows one to add custom *.c file in the list of source files. Once again, precautions must taken to avoid conflicting drivers.

Applications are written into C. Relevant example applications for this course are located in the folders ↪*examples/hello-world*, ↪*examples/ipv6/rpl-border-router*, ↪*examples/ipv6/rpl-udp* and ↪*examples/z1*. The last folder is only necessary when programming IPv6 applications on real Zolertia Z1 nodes. Each application is provided with its own makefile. This makefile is a receipt which allows the msp430-gcc compiler to determine which c-files to compile and which files to include. It contains the name of the project(s), which possible apps (from the folder ↪*apps/*) to include, which stack to include, etc. A general overview on how to create a makefile alongside with its application in its own folder can be found here http://anrg.usc.edu/contiki/index.php/Build_your_own_application_in_Contiki.

# 2
# Zolertia Z1 platform

The Zolertia Z1 platform is based on the MSP430F2617 microcontroller which has the following characteristics:

- 92kB of program flash,
- regular IO ports,
- IO ports with interrupt function (like the push-button),
- Digital IO communication to communicate with digital sensors (I2C) like the on board accelerometer, and with external sensors such as the light sensor,
- Serial port communication. One of these ports is connected to the cp2010 serial to USB chip which allows serial communication with a computer (programming, console printf).
- Wireless communication support through the antenna provided with Zolertia Z1 mote.
- Support for the Contiki operating system which provides the 6LoWPAN stack with UDP functionality.

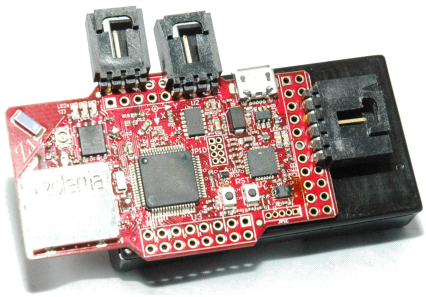

**Figure** 2.1: Zolertia Z1 mote. Picture taken from the Zolertia website.

### 2.0.1 Useful commands

Here we give a short overview of the most used commands (terminal) in this course.
**Commands when using real motes**.

- Compiling and uploading an application: *#: sudo make <app-name>.upload TARGET=z1*
- Cleaning a project: *#: sudo make clean TARGET=z1*

- Burning the node-id into the Zolertia Z1 (in the folder ↪*/examples/z1*): #: *sudo make burn-nodeid.upload nodeid=<value>nodemac=<value>TARGET=z1*. Do not forget to replace the value with a unique value (numeric) for the node for the complete network.
- Reading the serial port (console) output of a mote: #: *sudo make login*.
- Connecting the tunslip6 application to the edge-router (in the folder ↪*/tools*): #: *sudo ./tunslip6 aaaa::1/64 -v3*. If the application could not connect, reset the node and relaunch the application. The -v parameter determines the verbosity of the application. The higher the number, the more output (up to 5). Compiling the tunslip6 application can be done by: #: *gcc tunslip6.c -o tunslip6*.
- Each of the previous commands can be padded with the following command: #: *MOTES=/dev/ttyUSBx*, where the x in the ttyUSB device indicates which device to select (by number). An example of a command with this command padded would be: #: *sudo make <app-name>.upload TARGET=z1 MOTES=/dev/ttyUSBx*. If the MOTES command is omitted the compiler will try the operation on the first ttyUSB device available.

**Commands when using the Cooja simulator**.

- Connecting the tunslip6 application to the edge-router (in the folder /tools): #: *sudo ./tunslip6 aaaa::1/64 -a 127.0.0.1 -p 60001 -v3*. If the application could not connect, reset the node and relaunch the application. The -p argument determines the port on which the application will connect. This corresponds to the number given by the edge-router in the simulation tool. The -v parameter determines the verbosity of the application. The higher the number, the more output (up to 5). Compiling the tunslip6 application can be done by: #: *gcc tunslip6.c -o tunslip6*.
- The "MOTES" command is not valid in the context of the simulator.
- Compiling, programming the node ID launching the application and reading the console for each mote is done in through the simulator.

# 3 A basic Contiki Application

## 3.1 Demo applications

The Contiki operating systems comes along with a set of example applications. However, to understand these applications, a few notes are necessary. In the next code snippets provided in this course, we will discover the basics of Contiki. Contiki is a multi-threaded event driven operating system. This means that we run at least one thread and that interrupts (timers, push button, sensors) are made available through events. The first code snippet details the basic element in the Contiki application. In this example, only one thread is created that exits when the loop is executed for the first time (code snippet 3.1).

**Snippet** 3.1: Blank Contiki application.

```
1  //-- the include that is always used in a Contiki application
2  #include "contiki.h"
3  //-- Definition of the processes (actually a thread)
4  PROCESS(blink_LED, "blink_LED");
5  //-- load this process at boot
6  AUTOSTART_PROCESSES(&blink_LED);
7
8  //-- the process
9  PROCESS_THREAD(blink_LED,ev, data)
10 {
11    //-- Defines the exit statement
12    PROCESS_EXITHANDLER(goto exit);
13    //-- Defines where the application starts, must be set!
14    PROCESS_BEGIN();
15
16    //-- main loop of the application
17    while(1)
18    {
19       //-- do stuff here
20       //-- defines the end of the process
21       exit: PROCESS_END();
22    }
23 }
```

The most important parts are already highlighted in the code with comments. Although this code-snippet is complete and compiles fine, this Contiki application does not do anything. In order to be able to interact with sensors/actuators, the programmer has to define the contents of the process. The first demo we will cover will handle the blinking of a LED. As a first step,

we need to define an event that will be triggered at regular time intervals: the timer. One can recycle code snippet 3.1. The timer can be used by including the ctimer library from Contiki, and by declaring a timer time period a and the timer variable. Code snippet 3.2 illustrates the use of a timer. The timer is fired every second ("TIMER_PERIOD"), and is handled as an event within the main loop of the application.

**Snippet** 3.2: Code to enable timer functions in Contiki. Note that some code has been omitted.

```
//-- include the library for the timer
#include "sys/ctimer.h"
//-- define the timerduration, multiplied
//-- by the systemclock (CLOCK_SECOND)
//-- the total duration corresponds here to 1 second
#define TIMER_PERIOD 1*CLOCK_SECOND

//-- use following variables/structs to define the timer
static struct etimer launchtimer;

//-- before the main loop of the application, launch the timer by:
etimer_set(&launchtimer,TIMER_PERIOD);
while(1)
{
   //-- in the main loop, declare following statemant to block
   //-- the execution of next statemants until an event occurs (replaces
       PROCESS_YIELD();)
   PROCESS_WAIT_EVENT();
   //-- the Process_Yield will resume when an event occured
   //-- such an event can be caused by the timer running out
   //-- and is given below
   if (etimer_expired(&launchtimer))
   {
      //-- reset the timer so it can be fired again
      etimer_reset(&launchtimer);
      //-- do stuff here, make LED blink for example
   }
}
```

To be able to turn on and off the available LEDs one the Zolertia, we will add some LED-functionality. Code-snippet 3.3 illustrates this. The place where the user puts the LED-functions should be obvious. In case of the LED-blink, the "leds_toggle" function should be used within the "etimer_expired" event-handling code-block.

**Snippet** 3.3: Using LEDS in Contiki.

```
//-- include the library for the LEDS
#include "dev/leds.h"
//-- toggling on the LEDS can be done in this way:
leds_on(LEDS_GREEN);
leds_on(LEDS_RED);
```

```
leds_on(LEDS_BLUE);
leds_on(LEDS_ALL);
//-- toggling off can be done like:
leds_off(LEDS_GREEN);
leds_off(LEDS_RED);
leds_off(LEDS_BLUE);
leds_off(LEDS_ALL);
//-- toggling the LEDS from one state to another:
leds_toggle(LEDS_GREEN);
leds_toggle(LEDS_RED);
leds_toggle(LEDS_BLUE);
leds_toggle(LEDS_ALL);
```

The Zolertia also enables the user to use the push-button to trigger an event. The way this event is handled within a Contiki application is quite similar to a timer event. The code for a button-event is shown below.

**Snippet** 3.4: Using the button in Contiki. Note that some code has been omitted.

```
//-- include the library for the push-button
#include "dev/button-sensor.h"
//-- enable the button-sensor in the code,
//-- generally before the main loop
SENSORS_ACTIVATE(button_sensor);
//-- in the main loop,
//-- provide event handler for the push-button, after the PROCESS_YIELD();
if(ev == sensors_event)
{
   if(data == &button_sensor)
   {
      //-- do stuff here (LED blink for example)
      //-- ev and data are both declare in the PROCESS_THREAD
   }
}
```

Another method compared to the "PROCESS_YIELD();" method is the "PROCESS_WAIT_EVENT();" method. This method blocks the current thread until and event occurred (button press, timer event,etc.). Note the use of the "Printf" method which enables us to output some data to the console. Depending if we are using the simulator or real motes, the console will be given in the simulator as the "motes output", or by using the login command in the terminal for real motes. The library "stdio.h" must be included in order to use "printf".

**Snippet** 3.5: Using the events in Contiki.

```
#include "contiki.h"
#include "dev/leds.h"
#include "dev/button-sensor.h"
#include <stdio.h> /* For printf() */

```

```
6  #define PERIOD CLOCK_SECOND*5
7  //---------------------------------------------------------------------
8  PROCESS(led_blink, "blink the red led with a timer");
9  AUTOSTART_PROCESSES(&led_blink);
10 //---------------------------------------------------------------------
11 PROCESS_THREAD(led_blink, ev, data)
12 {
13    PROCESS_EXITHANDLER(goto exit);
14    PROCESS_BEGIN();
15
16    /* Initialize timer and LEDS */
17
18    leds_off(LEDS_ALL); /* set them all off */
19    /* Print text to the console (UART over USB, Cooja console otherwise)*/
20    PRINTF("All leds are off\n");
21
22    static struct etimer et; // Define the timer
23
24    etimer_set(&et,PERIOD); // Set the timer
25    while(1)
26    {
27       /* wait until an event occurs */
28       PROCESS_WAIT_EVENT(); // Waiting for a event, don't care which event.
29       /* we can continue when an event has occurred */
30       if(etimer_expired(&et))
31       {
32          // If the event it's provoked by the timer expiration, then...
33          leds_toggle(LEDS_BLUE);
34          // reset the timer so we can catch the next timer event
35          etimer_reset(&et);
36       }
37       /* if we wanted the button event, place it here at the same level*/
38       /* if (ev==sensors_event) {...}*/
39    }
40    exit: PROCESS_END();
41 }
```

### 3.1.1 Exercises (Cooja)

- Write an application that enables to read events of both a the button and the timer. When the (user) button is pressed, the green LED is toggled. When the timer goes of, the red LED is toggled. Set the timer interval at 2 seconds Simulate this in cooja.
- Retake previous exercise, and write the message “Button has been pushed” when a button event occurred, or “The timer went off” when the timer goes of. Keep the LED toggling capabilities.

- Program an application that counts the amount timer events, and outputs it in a binary way to the LEDs. Since the Z1 mote has 3 leds, we can hold up to 8 states (0 up to 7).

### 3.1.2 Exercises (real motes)

- Write an application that enables to read events of both a the button and the timer. When the (user) button is pressed, the green LED is toggled. When the timer goes of, the red LED is toggled. Set the timer interval at 2 seconds. Program the node with this application.

- Retake previous exercise, and write the message "Button has been pushed" when a button event occurred, or "The timer went off" when the timer goes of. Keep the LED toggling capabilities. Use the login command to monitor the serial port for console messages from the mote.

- Program an application that counts the amount timer events, and outputs it in a binary way to the LEDs. Since the Z1 mote has 3 leds, we can hold up to 8 states (0 up to 7). Program the node with this application.

# 4 Introduction to a 6LoWPAN network

We previously demonstrated how we can build and compile a basic Contiki-application. In this section, we will review the necessary components of a 6LoWPAN-network.
To form a 6LoWPAN-network, one must have the following system components:

- the edge-router with the border-router software installed,
- the 6LoWPAN nodes, e.g. with UDP-server or UDP-client applications programmed,
- a computer on which the edge-router can be attached with a USB cable. This can be a regular computer, a BeagleBone, a Raspberry PI, etc.

The computer and the edge-router are what we call the gateway. Together they make it possible to transport packets from the outside world to the local 6LoWPAN-network, and backwards. The UDP-servers and UDP-clients ($Z_x$-nodes in figure 4.1) are inside the 6LoWPAN network.

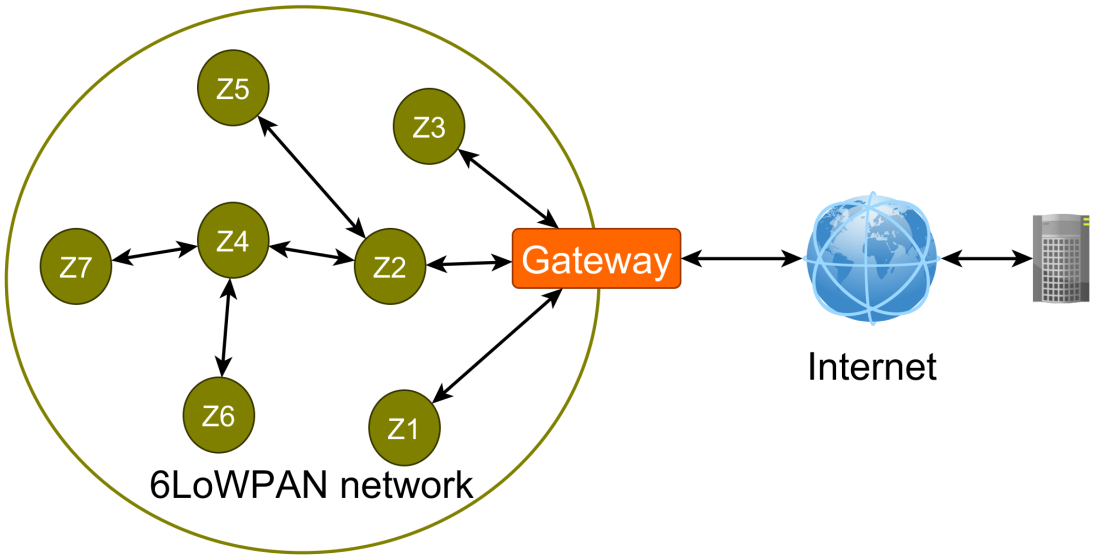


**Figure** 4.1: A typical 6LoWPAN topology.

### 4.0.3 Edge-router

To build up the network, one must first set up the edge-router sofware. This build-up is done in two steps:

- first, the edge-router needs the edge-router-Conikti application installed,
- the PC must run the tunslip6 application in order to tunnel all data from the 6LoWPAN-network to the normal IP-network.

The Contiki-application for the edge-router can be found in *↪/examples/ipv6/rpl-border-router/border-router.c*. When using real motes, the user needs to compile the application to the intended Zolertia Z1 node. This can be done using:

#: *sudo make border-router.upload TARGET=z1*

This application is compiled directly in the simulator when using Cooja.

After the edge-router application is installed onto the node, one can bridge the 6LoWPAN-network with the IP-network using the tunslip6 application. This application can be found in the folder /tools. One needs to compile it using:

#: *gcc tunslip6.c -o tunslip6*

When compiled, the "tunnel IP over serial line"-application can be ran using (real motes):

#: *sudo ./tunslip6 aaaa::1/64 -v3*

For use with the simulator, when can run this application with:

#: *sudo ./tunslip6 aaaa::1/64 -a 127.0.0.1 -p 60001 -v3*

In both cases, when needs to specify the local 6LoWPAN prefix (aaaa::1/64). This will be used by all the nodes inside the network to identify their network. If everything runs well, one must see the IPv6 address of the edge-router, along with its corresponding local link IPv6 address. If not, reset the edge-router (reset-button), and restart the tunslip6 application. The "-v3"-option indicates the verbosity of the application. The higher the verbosity (i.e. -v5), the more output this application will generate. Observe this application. When data is travelling from the 6LoWPAN-network, it is shown as: "from slip to tun". In the opposite way the tunnel application will output: "from tun to slip".

### 4.0.4 IP addresses

One can wonder which IP addresses are given to the different nodes. Each node inside the network will be assigned an IP address. Depending if the network simulated or when we use real motes, the IP address are generated in two different ways. The IP addresses consist out of 3 parts: the IPv6 prefix (64 bits), the IPv6 local network and the node ID. The first two are build in the same fashion in both the simulator and the real motes. The last one (the node ID), is done differently.
The IPv6 prefix is set by the prefix given at the edge-router ("aaaa::1/64"). It is the tunslip6 application (and thus the user) who determine this prefix. This will result in IP address in the

form of "aaaa:0:0:0:x:x:x:x" for each of the nodes in the network. One should note that the tunslip6 application creates a tunnel at the computer side, with IP address "aaaa::1". When nodes want to reach a remote application on the host machine, they need to use that IP address. Each node inside the network (including the edge-router), will have an IP adress assigned which starts with this prefix. The other part consists of a local address group. Generally, it is derived from the local link loop address from Contiki. The last part of address consists of the Node ID. When using the simulator, the first node put onto the map will have the first ID (i.e. '1'). The second node will have node ID '2', etc. For the real motes, the ID corresponds to the nodeID which can be burned beforehand onto the nodes (see usefull commands). The addresses (hexadecimal format) of the different component in our network will be as follows (with "aaaa::1/64" prefix):

- Remote application: aaaa:0:0:0:0:0:0:1 or aaaa::1
- Edge-router: aaaa:0:0:0:0xc30c:0:0:1 or aaaa::0xc30c:0:0:1
- Nodes inside the network: aaaa:0:0:0:0xc30c:0:0:123 or aaaa::0xc30c:0:0:123

### 4.0.5 Monitoring the network

Although Contiki has a nice set of IPv6 connectivity built in, it is good to trust it. It is even better if we can check the validity of the communication between the nodes. There are set of tools which can be used to test the interconnectivity of the nodes inside the network. One can monitor the network at several levels.

- At node level: using the "Printf" statement to debug a node.
- Inside the 6LoWPAN network (using the simulator): one can use the monitor tool provided by the Cooja simulator. Better is to export the radio transmissions to Wireshark. This can be done by activating the Radio messages (under tools in Cooja) and by setting the analyzer to "6LoWPAN with PCAP". This tool will export the communication to a file located in $\hookrightarrow$*Contiki/tools/cooja/radiologxxxxx.pcap*. The filename is always different. Read this file into Wireshark, and a list of communications will appear. The format is 6LoWPAN.
- Inside the 6LoWPAN network (real motes): Some sniffer devices exist that enables to overhear the radiocommunications between nodes. An example of such is the sniffer provided by Texas Instruments. The format is 6LoWPAN.
- At the tunslip6 application (tunnel). Point the Wireshark application towards the tunnel devices created by tunslip6 (generally "tun0"). This will allow to overhear IPv6 communication between the 6LoWPAN and regular IPv6 networks. The format is IPv6.

## 4.1 Setting up UDP applications within the network

Contiki is developed with IPv6 connectivity in mind. Contiki provides the compressed version of IPv6, which is called 6LoWPAN. The developers of Contiki only implemented the UDP

functionalities, since TCP would demand too much overhead from battery powered constraint devices. Although it might seem that the names provided by those developers resembles to the TCP communication method, only UDP has been fully implemented.

### 4.1.1 UDP-client

The code from which we start, is located in the folder *↪/examples/ipv6/rpl-udp*. For the UDP-client we start with the udp-client.c file. The code is subdivided in several subtasks. Of course, the UDP-client has also the PROCESS_THREAD. The execution of the code also starts at that place. The first important function to be mentioned is the "set_global_address();". This function is detailed in code-snippet 4.1.

**Snippet** 4.1: Setting addresses in Contiki UDP-client.

```
1  //-- struct for the ipaddress of our node
2  uip_ipaddr_t ipaddr;
3  //-- struct for the remote ip address (server),
4  //-- should be declared as global
5  uip_ipaddr_t server_ipaddr;
6  //-- initialize our client stack, the client will adapt
7  //-- its ip address according to the network
8  //-- here the local IP address is defined by the MAC and network
9  //-- addresses
10 uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
11 uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
12 uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
13 //-- init the server ip address, here aaaa::1 -> (outside the 6LoWPAN network)
14 uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);
```

This function is originally written with other code, just keep these four lines of code! Bear in mind that the "server_ipaddr" is already declared as a global variable, on top of the file (no need to redeclare here).
After this function, Contiki will print the addresses with the function "print_local_addresses();". This function only prints the address of the server and the client.
The next step is done by opening a connection to the remote server. Therefore the client opens a local socket through which it can send data. To do so, some functions are involved (code-snippet 4.2).

**Snippet** 4.2: Setup of the socket at the client side.

```
1  //-- globally declared client-connection (pointer)
2  static struct uip_udp_conn *client_conn;
3  //-- make new socket with the portnumber of the server
4  client_conn = udp_new(&ipaddr, UIP_HTONS(UDP_SERVER_PORT), NULL);
5  //-- bind the socket in our system with client connection port
6  udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));
```

The server and client can only communicate if the sockets are set up in the right way. A socket is defined by two parameters:

- The IP-addresses: client and server should know the IP address of their partner, usually the client initiates the connection, and the server replies.
- Port numbers: both server and client can only communicate if their listening and writing ports match.

After the socket has been set up, the client is able to send data to the server and to receive data back. Sending data is done by using for example the timer. This method ensures a regular sending interval. The sender method is called through a callback method of the timer. To send information to the server, the method "send_packet(void *ptr)" is used (code-snippet 4.3).

**Snippet** 4.3: Sending a UDP-packet at client side.

```
//-- declare both the sequence ID and the
//-- buffer for the payload (message)
//-- the max mpayload length has been declared elsewhere
static int seq_id;
char buf[MAX_PAYLOAD_LEN];
//-- increment the message number
seq_id++;
//-- print to the console (when issuing 'sudo make login')
PRINTF("DATA send to %d 'Hello %d'\n",
   server_ipaddr.u8[sizeof(server_ipaddr.u8) - 1], seq_id);
//-- put our desired message as a string, where the ID is also included
//-- to put an integer with a string, we use
//-- the String Print Formatted function
//-- this function behaves like the printf function
sprintf(buf, "Hello %d from the client", seq_id);
//-- now, send everything to the remote server
uip_udp_packet_sendto(client_conn, buf, strlen(buf),
   &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
```

To receive packets from a remote device (server) the other function "tcpip_handler(void)" is used. This function is called when the "tcpip_event" is raisen in the main loop(code-snippet 4.4).

**Snippet** 4.4: Processing the event of a receiving packet.

```
if(ev == tcpip_event)
{
   tcpip_handler();
}
```

The code for decoding a received packet is shown in code-snippet 4.5.

**Snippet** 4.5: Processing the event of a receiving packet.

```
//-- define a pointer to an array of characters
char *str;
//-- if we have a valid incoming packet, then do:
if(uip_newdata())
```

```
{
   //-- our pointer becomes valid, and is
   //-- pointing to the incoming data
   str = uip_appdata;
   //-- since our data is a string (example of Contiki)
   //-- always terminate a string with a zero-character
   //-- both '\0' and 0 are valid
   str[uip_datalen()] = '\0';
   //-- print the data (string)
   printf("DATA recv '%s'\n", str);
}
```

Instead of decoding the incoming data as a string, one can decode it as binary or numeric data. An example is shown in code-snippet 4.6.

**Snippet** 4.6: Processing the event of a receiving packet.

```
//-- define a pointer to an array of characters
char *str;
//-- if we have a valid incoming packet, then do:
if(uip_newdata())
{
   //-- our pointer becomes valid, and is
   //-- pointing to the incoming data
   str = uip_appdata;
   //-- make the green LED blink if the first element is 10
   //-- otherwise put all LEDS off.
   //-- a char* can have values of characters, which can
   //-- be represented as integers between -128 to +127 (1 byte)
   if (str[0]==10)
   {
      leds_on(LEDS_GREEN);
   }
   else
   {
      leds_off(LEDS_ALL);
   }
}
```

### 4.1.2 UDP-server in Contiki

The UDP-server of Contiki follows more or less the same rules as the UDP-client. There are however some major differences compared to the client.

- The server only binds itself to a socket. When the socket is created, the server will wait until an incoming connection arises. At that moment, a tcpip_event will be arisen and the server parses the incoming data.

- Since the server only binds to its own socket, it will not create a clientsocket beforehand. In order to respond back to a client, the server will only create a socket for the client when data comes in from a client. The socket is then used to reply back to that particular client.

Some additional details should be noted before proceeding to the programming of such a node. First of all, during our labs, the servers' IP address will be given by the nodes nodeid and nodemac (in case of real motes). Therefore, we will compile the application in mode 3 (nothing special to be done). This results in the IP-address of the server being set up as follows:

**Snippet** 4.7: Set up IP address of server.

```
1 //-- Mode 3 - derived from link local (MAC) address
2 uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
3 uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
```

Remember that the IP address of the server is given by the nodeid and nodemac, combined with the IPv6-prefix of the network. E.g. if the nodeid and nodemac are equal to 6, and the IP address of the border-router equals aaaa::c30c:0:0:1, then the IP address of the UDP-server will be: aaaa::c30c:0:0:6. This address should be used in UDP-client applications in order to reach the server.
Also, the UDP-port of the server should be the same as the one used in the UDP-client.
In order for the server to reply back to a clients request, the server implements following method (code-snippet 4.8):

**Snippet** 4.8: Server replying back to a client.

```
1  char *appdata;
2
3  //-- normal reception, as seen in the client
4  if(uip_newdata())
5  {
6      appdata = (char *)uip_appdata;
7      appdata[uip_datalen()] = 0;
8      PRINTF("DATA recv '%s' from ", appdata);
9      PRINTF("%d",
10           UIP_IP_BUF->srcipaddr.u8[sizeof(UIP_IP_BUF->srcipaddr.u8)
11           - 1]);
12     PRINTF("\n");
13    //-- reply back to a client request
14     PRINTF("DATA sending reply\n");
15     //-- copy the IP address of the client
16     uip_ipaddr_copy(&server_conn->ripaddr, &UIP_IP_BUF->srcipaddr);
17     //-- reply back to the client
18     uip_udp_packet_send(server_conn, "Reply", sizeof("Reply"));
19     uip_create_unspecified(&server_conn->ripaddr);
20
21 }
```

### 4.1.3 Exercises (Cooja)

- Demo: Write an UDP-client application that sends a text message to a UDP-server in the network every 5 seconds. Make sure the IP address of the server matches with the client application and both send and receive ports of both motes correspond. Print the received message at the server, and respond back to the client with same message. Set up the network by using the edge-router, a UDP-client and a UDP-server. Monitor the network traffic with the available tools (Cooja and Wireshark). **Hint**: put the UDP-client as last node onto the simulator to know the other nodes their IP addresses.

- Write an client application that sends a value between 0 and 9 to the server at regular time intervals (5 seconds). The value is incremented before sending. When the server reads a 7, the red led goes on. When a 9 is read, the blue led goes on. In all other circumstances, the leds are off. Monitor the network as well.

- Write an client application that sends a '1' when the user button has been pressed. The server outputs the event in the console.

### 4.1.4 Exercises (real motes)

- Demo: Write an UDP-client application that sends a text message to a UDP-server in the network every 5 seconds. Make sure the IP address of the server matches with the client application and both send and receive ports of both motes correspond. Print the received message at the server, and respond back to the client with same message. Set up the network by using the edge-router, a UDP-client and a UDP-server. Monitor the network traffic with the available tools (printf). **Hint**: put the UDP-client as last node onto the simulator to know the other nodes their IP addresses. **Hint**: Do not forget to set nodeID and nodeMac to each node.

- Write an client application that sends a value between 0 and 9 to the server at regular time intervals (5 seconds). The value is incremented before sending. When the server reads a 7, the red led goes on. When a 9 is read, the blue led goes on. In all other circumstances, the leds are off. Monitor the network as well (printf).

- Write an client application that sends a '1' when the user button has been pressed. The server outputs the event in the console.

### 4.1.5 Exercises (Cooja with remote server)

A starting application for the UDP-remote application server can be downloaded from the Rapptor-website at rapptor.vub.ac.be Note that for the stability of the network, only one edge-router should be considered.

- Demo: Write an UDP-application that sends data to a remote UDP-server. The remote server prints the received messages. Monitor the network traffic. Make sure the tunslip6 application runs in order to let the traffic pass.

- Write an client application that sends a value between 0 and 9 to the remote UDP-server at regular time intervals (5 seconds). The value is incremented before sending. The remote server writes the received value to the console.
- Write an client application that sends a '1' when the user button has been pressed. The remote server outputs the event in the console.

### 4.1.6 Exercises (real motes with remote server)

- Demo: Write an UDP-application that sends data to a remote UDP-server. The remote server prints the received messages. Monitor the network traffic. Make sure the tunslip6 application runs in order to let the traffic pass.
- Write an client application that sends a value between 0 and 9 to the remote server at regular time intervals (5 seconds). The value is incremented before sending. The remote server writes the received value to the console.
- Write an client application that sends a '1' when the user button has been pressed. The remote server outputs the event in the console.

# 5 Additional hardware: sensors and actuators

The Zolertia Z1 motes have been designed to enable interfacing with other hardware like sensors and actuators. With this course are provided some switches, a light sensor and a few LED boards. The Zolertia Z1 also has an accelerometer and a temperature sensor build in. Contiki also provides ways to implement custom driver for the operating system. The next few exercises will guide you through the usage of such external devices. A lot of information is available on the internet (Contiki mainpage). A few links are given below in order to get a starting point for the next exercises. One should always carefully read the posted code. Generally speaking: sensors and actuators first need to be initialized before use. Some useful links are:

- Example about reading the light sensor: http://wiki.zolertia.com/wiki/index.php/Mainpage:Contiki_drivers, scroll down to the "ZIG002 Light Sensor driver" section.
- Zolertia Z1 datasheet,in case of developping your own driver: http://zolertia.sourceforge.net/wiki/images/e/e8/Z1_RevC_Datasheet.pdf.
- PIR motion sensor: the PIR motion sensor can be interfaced through the available driver at the rapptor website rapptor.vub.ac.be. Look at the datasheet to get information to connect the PIR to the Zolertia Z1 port (ask if necessary).
- Activating an external LED: this can be done by using the already build in relay-phidget code (section "Relay sensors (phidget-like connectors)") on http://wiki.zolertia.com/wiki/index.php/Mainpage:Contiki_drivers.

### 5.0.7 Exercise

Build one 6LoWPAN network (real motes) with the complete group. In the network are following elements available:

- One edge-router: the edge-router is connected via a USB cable to a Linux machine. A UDP-server runs on that machine and processes the incoming requests from the clients (Zolertia Z1 motes).
- The remaining nodes are used for sensing and actuation. Use the possible available sensors and actuators.
- Monitor the messages at the tunslip6 application and use the printf statement to verify the execution of the program code.