448.053 Embedded Internet, Labor
Exercise 6 - Building IoT Applications using ReST & CoAP

The sixth exercise of the Embedded Internet Laboratory course aims to develop a simple theft protection system using ReST and CoAP. Before diving into the exercise specifications, get acquainted with CoAP and ReST by completing the preparation task described below.

Preparation (i.e., make sure your VM works)

Basic principles of ReST and CoAP. It is nowadays common to use ReST APIs to interact with services in the cloud or on-site (e.g., Google services or Amazon Web services). While for Web applications HTTP is the ReST protocol of choice, in the IoT world a more efficient protocol is required. The Constrained Application Protocol (CoAP) was proposed by IETF as a ReST implementation for constrained devices. Unlike HTTP, CoAP employs UDP as underlying transport layer protocol, and uses a binary representation instead of HTTP's text-based communication. ReST relies on a limited set of primitives for interaction:

- **GET** is used to retrieve a resource or to list a collection;
- **POST** is used to create a new collection;
- **PUT** is used to replace an existing collection or resource (or to create it, if not existing);
- **DELETE** is used to delete a collection or resource.

Similarly to HTTP, CoAP requires an Uniform Resource Identifier (URI) to access a resource. A common form of a URI is `scheme://host:port/path?query`. The port is optional and does not need to be specified: if this is the case, `80` is assumed for HTTP and `5683` is assumed for CoAP. An example of CoAP URI is: `coap://[fe80::101]:5683/actuators/leds?color=r`, where the host portion is specified through its IPv6 address.

A regular browser is typically unable to access or present a CoAP URI. However, Mozilla Firefox can be extended with *Copper*: a plugin that presents CoAP resources as a tree view and allows basic GET, PUT, POST, and DELETE operations. Please note that the latest version of Firefox (Quantum) does not support plugins: therefore, please download our Firefox portable version with the Copper user-agent pre-installed from `https://gitlab.random-circuits.com/toolchains/embedded_internet_17/raw/master/FirefoxPortable.zip` and do not update it.

Copper user-agent. Using Copper it is possible to communicate with a CC2650 SensorTag running a CoAP server by simply entering the SensorTag's IPv6 address in the Web browser, as shown in Figure 1.

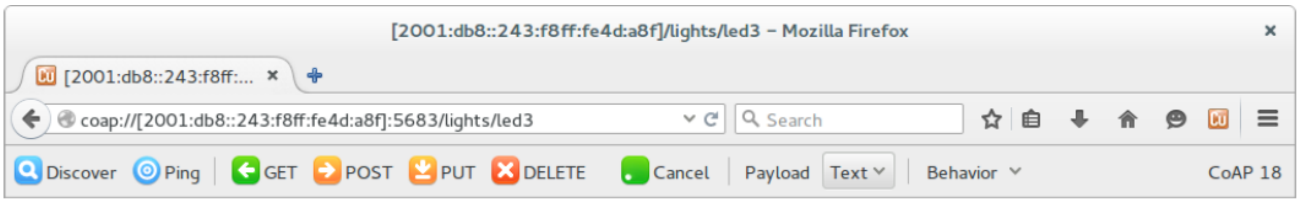

[2001:db8::243:f8ff:fe4d:a8f]/lights/led3 – Mozilla Firefox
[2001:db8::243:f8ff:...
coap://[2001:db8::243:f8ff:fe4d:a8f]:5683/lights/led3
Search
Discover
Ping
GET
POST
PUT
DELETE
Cancel
Payload
Text
Behavior
CoAP 18

Figure 1: Copper plugin for Mozilla Firefox.

To see Copper in action, take a look at the provided `coap_example.zip`. This file contains two programs that share a common `Makefile`:

- `coap-server.c` (that serves as a CoAP server and exposes a few resources);
- `coap-client.c` (that connects to a CoAP server and uses CoAP to read or write its resources).

Form a group. As a first step, please team up with another student and form a group of two people. In case no partner is available, you are exceptionally allowed to form a group of three.

Completed ✔

Setting up a border router. Set up a LoWPAN with a border router (BR) as root by following the steps shown in Figure 2. Assign one member of your group to be the BR, and download the Virtual Machine (VM) from `https://gitlab.random-circuits.com/toolchains/embedded_internet_17/raw/master/6LBR-SSIOT16-v1.3.ova`.

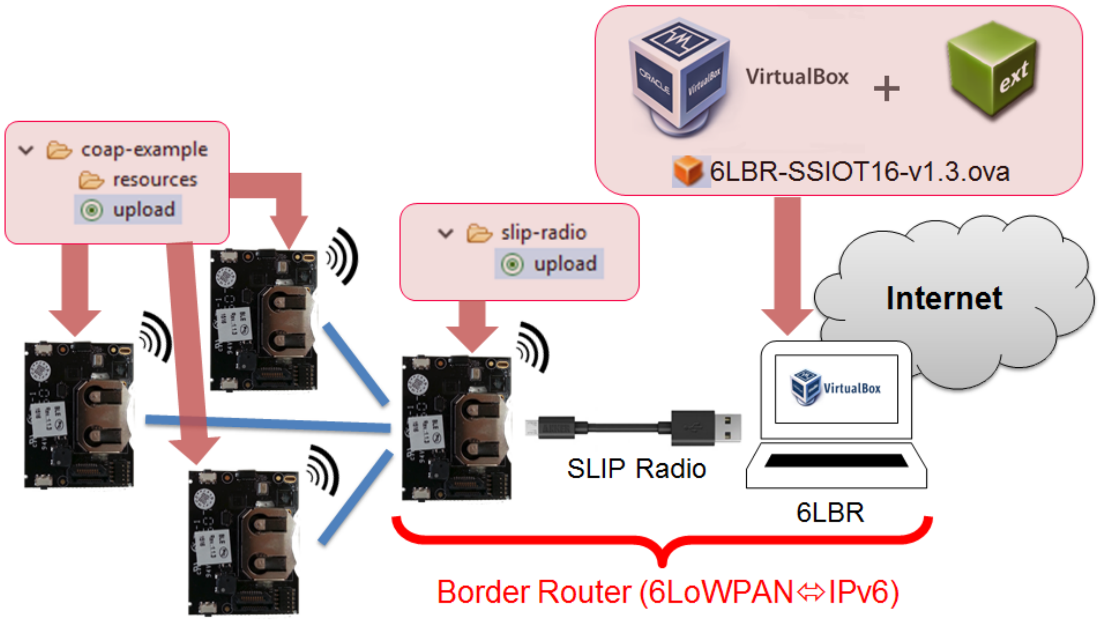


Figure 2: Network setup for the in-class tutorial with Copper.

The virtual machine contains a border router software called 6LBR by Cetic[1], which allows us to access IoT devices by using a SensorTag as a radio. The 6LBR code is also based on Contiki, but it is compiled for the native (x86) target and hence runs on top of Linux. Upon starting, 6LBR looks for any connected Serial Line IP (SLIP) radios and establishes a connection by decoding the serialized IP packets to a virtual network interface in the Linux kernel (a so called TUN device). As it is based on Contiki, it supports the same features we have previously explored in this course, including RPL, 6LoWPAN, and ContikiMAC.

6LBR also serves as a DAG-root and announces the Contiki's default prefix to all nodes. Contiki's default prefix, as configured in `contiki/core/net/ipv6/uip-ds6.h`, is:

```
#define UIP_DS6_DEFAULT_PREFIX   0xfd00
```

As the announced prefix `fd00::/64` is not a globally routable IPv6 prefix, network access translation (NAT) is needed. Furthermore, the university infrastructure has not yet been migrated to IPv6, therefore we also need to use NAT64. 6LBR is able to do this on its own: any outgoing IPv6 connections (e.g., to the IBM's Quickstart Platform) are first translated to IPv4 via NAT64, and then once more via NAT by the VirtualBox NAT interface. While this seems unnecessarily complicated, the advantage is that we do not have to bridge the VM's interface with any of the hosts.

[1] `https://github.com/cetic/6lbr/wiki`

6LBR also contains a Web interface for configuration and node management. It uses a second network interface in the VM that is configured to host-only mode. From this interface one can change, among others, the employed MAC protocol and prefixes, as well as trigger a global repair of the RPL DAG.

Flash the `contiki/examples/ipv6/slip-radio` firmware updated with your channel number on a CC2650 SensorTag by specifying "`TARGET=srf06-cc26xx BOARD=sensortag/cc2650 slip-radio.upload`" as `upload` target. This allows communication with the border router software embedded in the VM.

Completed ✓

Afterwards, use the provided `coap-server.c` program with the right channel number to flash a second SensorTag as a CoAP server, and verify that it correctly appears on the 6LBR Web interface. The `coap-server.c` can be flashed with a regular `upload` target.
In order to avoid mutual interference between groups, each LoWPAN will be operating on a different channel. Hence, define the `RF_CORE_CONF_CHANNEL` in your `project-conf.h` configuration file according to Table 1.

Group	Channel	Group	Channel	Group	Channel	Group	Channel
01	11	05	15	09	19	13	23
02	12	06	16	10	20	14	24
03	13	07	17	11	21	15	25
04	14	08	18	12	22	16	26

Table 1: `RF_CORE_CONF_CHANNEL` configuration for each group.

Before compiling the program, also comment the following lines from the `project-conf.h` file:

```
#undef UIP_CONF_BUFFER_SIZE
#define UIP_CONF_BUFFER_SIZE    140
```

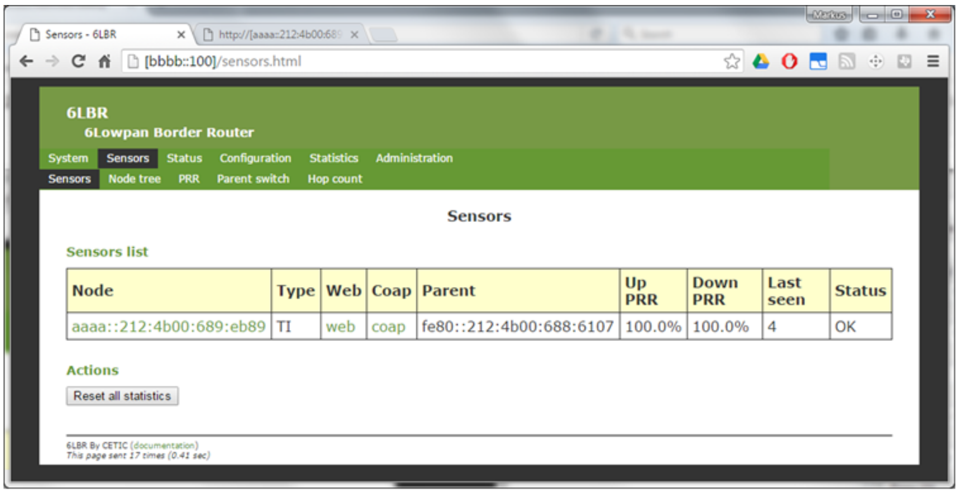
Completed ✓

Once these steps are completed, import the `6LBR-SSIOT16-v1.3.ova` file into VirtualBox and launch the VM (it is not needed to login, but you can do so using `user` as both username and password). Connect your CC2650 SensorTag flashed as slip radio to the notebook: an USB filter rule should add it automatically as a USB device in the VM. If this is not the case, please add it manually. Afterwards, use the Web browser of your notebook to access the Web interface of 6LBR running in the VM at `[http://bbbb::100]`. You should be able to see a Web interface similar to the one shown in Figure 3. If the Web interface is not displayed correctly, you can login into the VM and check the cause of the problem by doing `tail-f/var/log/6LBR.log`. If the device or the MAC address is not detected, you can simply disconnect and reconnect the cable of the SensorTag acting as slip radio.

Completed ✓

If your SensorTag correctly appears on the 6LBR Web interface, the interface will include a `coap://` link that should automatically start Copper. After pressing the "Discover" button in Copper, you should visualize a tree view of all resources available on the server, as depicted in Figure 4. Each resource is described by a URI similar to `coap://[IPv6Address]:5683/actuators/led` on which the ReST primitives GET, POST, PUT, and DELETE can be executed.

Completed ✓



6LBR
6Lowpan Border Router

Sensors

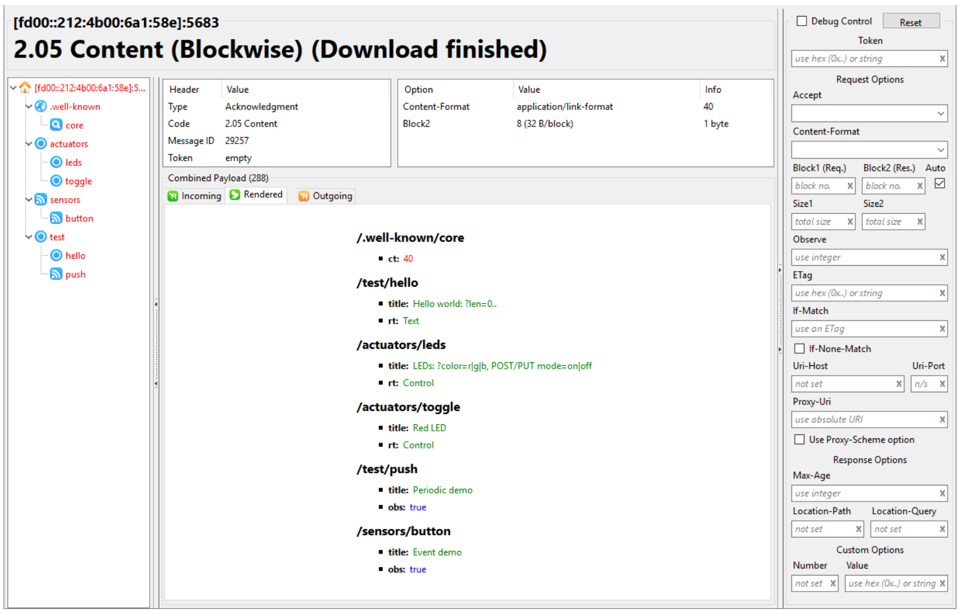
Sensors list

Node	Type	Web	Coap	Parent	Up PRR	Down PRR	Last seen	Status
aaaa::212:4b00:689:eb89	TI	web	coap	fe80::212:4b00:688:6107	100.0%	100.0%	4	OK

Actions

6LBR By CETIC (documentation)
This page sent 17 times (0.41 sec)

Figure 3: Web interface running 6LBR.



[fd00::212:4b00:6a1:58e]:5683

2.05 Content (Blockwise) (Download finished)

Header	Value
Type	Acknowledgment
Code	2.05 Content
Message ID	29257
Token	empty

Option	Value	Info
Content-Format	application/link-format	40
Block2	8 (32 B/block)	1 byte

Combined Payload (288)

/.well-known/core
- **ct:** 40

/test/hello
- **title:** Hello world: ?len=0..
- **rt:** Text

/actuators/leds
- **title:** LEDs: ?color=r|g|b, POST/PUT mode=on|off
- **rt:** Control

/actuators/toggle
- **title:** Red LED
- **rt:** Control

/test/push
- **title:** Periodic demo
- **obs:** true

/sensors/button
- **title:** Event demo
- **obs:** true

Figure 4: List of resources shown by Copper.

How does Copper know how to retrieve all the resources available on the server? Hint: check how resource discovery is implemented in CoAP.

Getting Acquainted with the CoAP Server and Client

Understanding the `coap-server.c` operation. The CoAP server running on your CC2650 SensorTag is using ReST to activate the five resources that you previously visualized in Copper. Practically, it instructs the ReST engine in `apps/rest-engine` to start accepting requests for these resources by doing:

```
/* The actual macro defining the resource is in the resource dir */
extern resource_t res_example;
/* Link the resources handlers to coap://[IP]:5683/path/to/example */
rest_activate_resource(&res_example, "path/to/example");
```

As you can see from this code snippet, the resources to be activated need to be imported through the `extern` keyword. These resources are indeed described in several `.c` files residing on a separate folder. Contiki's build system will take care of locating and compiling these files according to the instructions embedded in the `Makefile`, as described later on. In the provided code example, the `resources` folder contains the following resource descriptions:

- `res-hello.c` (resource that only contains static information accessible via a GET handler);
- `res-toggle.c` (resource that only contains a POST handler for toggling the red LED);
- `res-leds.c` (resource that uses a shared handler for POST and PUT to control the on-board LEDs);
- `res-event.c` (resource that notifies all subscribers when a external event occurs);
- `res-push.c` (resource that notifies all subscribers every 5 seconds).

Describing resources. To describe resources, Contiki uses the macros defined by the ReST engine in `apps/rest-engine`. Using these macros, one can easily create `resource_t` instances that can then be interacted with using a ReST API. Here follows a short description of the macros used in this assignment.

```
RESOURCE(NAME,
         ATTRIBUTES,
         GET_HANDLER,
         POST_HANDLER,
         PUT_HANDLER,
         DELETE_HANDLER);
```

The `RESOURCE` macro is for resources that behave in the same way as they would in a Web environment, and includes a handler for the typical GET, POST, PUT, and DELETE methods. The name of the resource is not the URI on the server, but instead the name of the resulting `resource_t` instance that is bound using `rest_activate_resource`. Please note that if a resource is activated twice, only the last URI is taken.

```
EVENT_RESOURCE(NAME,
         ATTRIBUTES,
         GET_HANDLER,
         POST_HANDLER,
         PUT_HANDLER,
         DELETE_HANDLER,
         EVENT_HANDLER);
```

```
PERIODIC_RESOURCE(NAME,
         ATTRIBUTES,
         GET_HANDLER,
         POST_HANDLER,
         PUT_HANDLER,
         DELETE_HANDLER,
         PERIOD,
         EVENT_HANDLER);
```

The EVENT_RESOURCE and PERIODIC_RESOURCE macros behave in the same way as the RESOURCE one, with the exception that they also implement an EVENT_HANDLER that can be used to allow clients to observe the resource and implement a subscription as for protocols based on publish/subscribe.
In the case of an EVENT_RESOURCE, the EVENT_HANDLER is called by the CoAP server through the function res_event.trigger (in the provided example code, this is done once a button has been pressed). In the case of a PERIODIC_RESOURCE, the EVENT_HANDLER is called periodically by the operating system (with the period of time specified in PERIOD).
The EVENT_HANDLER can be used to notify the subscribers. Unlike MQTT, CoAP does not rely on a dedicated broker (such as Mosquitto) for subscription management. This means that the CoAP server needs to keep a list of clients for each observable resource and notify them when necessary. In the provided examples, the registered observers are notified by calling the REST.notify_subscribers() function if a given condition is satisfied. In the res-event.c file (example of a EVENT_RESOURCE macro), this function is called in the res_event_handler that is invoked by a res_event.trigger in the server. In the res-push.c file (example of a PERIODIC_RESOURCE macro), the REST.notify_subscribers() function is called in the res_periodic_handler procedure that is periodically invoked by the operating system every 5 seconds.

Programming the handlers of a resource. Let's focus now on the GET, POST, PUT, and DELETE handlers of a resource. Open the resources/res-hello.c file from the coap-example application downloaded from the TeachCenter and focus on the RESOURCE declaration portion:

```
RESOURCE(res_hello,
         "title=\"Hello world: ?len=0..\";rt=\"Text\"",
         res_get_handler,
         NULL,
         NULL,
         NULL);
```

The declaration contains in the second line an attribute "title=\"Helloworld:?len=0..\";rt=\"Text\"" that is used as a description for well-known/.core. Whilst the third line specifies the function to be called on a GET request (res_get_handler), no handlers for PUT, POST, and DELETE are specified (the remaining lines are set to NULL). Hence, this is an example of a resource that can only be read using the GET method.
The GET request supports one parameter called len that represents the length of the returned string. This parameter is passed by adding ?len=x to the URL, with x in the range [0, REST_MAX_CHUNK_SIZE]. The default value for len is 12, hence, if no parameter is passed, the first 12 characters of "HelloWorld! ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxy" are returned. If the length exceeds the REST_MAX_CHUNK_SIZE, it is set to this maximum.

Extend the existing implementation of the resources/res-hello.c resource with a PUT handler that replaces the message that is returned by a GET message. The new string replacing the "HelloWorld! ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxy" is the one provided as the message variable in the payload of the PUT request (message=test). Get inspiration from resources/res-leds.c, use Copper to verify that it works correctly, and paste the relevant code of the implemented PUT callback in the space below.

```
char *message = "Hello World! ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxy";
 int rv;
 int length = 12;
static void res_put_handler(void *request, void *response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)
{
  char const *ptr = NULL;

  rv = REST.get_query_variable(request, "message", &ptr);
  if(rv)
  {
   length = rv;
   message = ptr;
   message[rv] = '\0';
  }
}
```

Adding a new resource. Extend the provided example by adding a new resource called `res-batmon`. This resource should provide a GET handler that returns the current battery voltage in mV. Bind this resource to the URI `sensors/battery`, and write a proper description that includes the resource type (`rt=`) "mV" and the title "Battery Voltage". Create a new resource file called `res-batmon.c` in the resources directory and use the regular `RESOURCE` macro to declare your resource. In the server portion, add this resource to the `extern resource_t` declaration, and create the corresponding `rest_activate_resource` function call. Don't forget to also add `SENSORS_ACTIVATE` in the server code to actually initialize and activate the battery sensor. In order to compute the mV from the raw sensor reading, you can reuse the equation derived in the first assignment.

Use Copper to verify that the implemented resource works correctly and paste the relevant code of your implementation in the space below:

```
static void
res_get_handler(void *request, void *response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)
{
  int val = 0;
  int value_mv=0;
  char const *message = NULL;
  int length = sizeof(value_mv);
  printf("length: %d", length);
  val = batmon_sensor.value(BATMON_SENSOR_TYPE_VOLT);
  if(val == CC26XX_SENSOR_READING_ERROR) {
    message="BAR: BAR sensor Error\n";
    length=sizeof(message);
    memcpy(buffer, message, length);
   }
  else {
   value_mv=((val * 125) >> 5);
   snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "'Value_mV': %d", value_mv);
   }
  REST.set_header_content_type(response, REST.type.TEXT_PLAIN);
  REST.set_header_etag(response, (uint8_t *)&length, 1);
  REST.set_response_payload(response, buffer, strlen((char*)buffer));
```

Erbium CoAP implementation. In principle, one can use either HTTP or CoAP with the same resource descriptions: the ReST engine, indeed, does not describe the application protocol that needs to be used to interact with the specified resources. Since the release of Contiki 3.0, however, the HTTP support (called Helium) has been removed and is no longer supported. Contiki, instead, supports Erbium (Er): a comprehensive CoAP implementation for Contiki developed by Matthias Kovatsch (ETH Zürich) that can be found in `apps/er-coap`. Erbium sits on top of the `apps/rest-engine` ReST engine.

To use CoAP and ReST in a Contiki application, the following additions are required in the `Makefile`:

```
# REST Engine shall use Erbium CoAP implementation
APPS += er-coap
APPS += rest-engine
```

Also defined in the `Makefile` is the `REST_RESOURCES_DIR` directory, which tells Contiki's build system to add the `.c` files with the description of the ReSTful resources at this location to `PROJECT_SOURCEFILES` so that they can be built and linked. In the provided example application, the `.c` files are embedded in the `./resources` directory.

```
# automatically build RESTful resources
REST_RESOURCES_DIR = ./resources
REST_RESOURCES_FILES = $(notdir $(shell bash -c \
  "find $(REST_RESOURCES_DIR) -name '*.c'"))

PROJECTDIRS += $(REST_RESOURCES_DIR)
PROJECT_SOURCEFILES += $(REST_RESOURCES_FILES)
```

The `project-conf.h` configuration file also defines a set of parameters specific to the Erbium CoAP implementation that are commented out by default. When running the exemplary application, there is no need to change them, but please read and understand them, as you may need or want to edit them when building your own IoT application(s).

Understanding the `coap-client.c` operation. Instead of using Copper, one may also read data from a CoAP server directly using a CC2650 SensorTag as client. Flash one or more SensorTag as CoAP clients, and let them connect to a CoAP server as shown in Figure 5. To accomplish this task, a border router is not necessary: hence, shutdown the VM containing the border router, and flash the CC2650 SensorTag previously used as a slip radio with `coap-client.c` (you can create a new target `upload.client` in Eclipse). Important: make sure to change the `DAG_ROOT_ENABLE` definition in the `project-conf.h` from 0 to 1 and to re-flash with this new configuration also the SensorTag running the CoAP server.

Completed ✓

Client's mode of operation. The `coap-client.c` application works as follows. First, the IP address of the CoAP server (named `server_ipaddr`) is derived using mDNS (i.e., following the same principle used to discover the IPv6 address of the sink nodes in the previous assignments) by looking for `exercise6_sink.local`.
Afterwards, the application waits either for either an `etimer` to expire, or for an user button to be pressed. The `etimer` expires every 10 seconds, after which a POST request to the URI `actuators/toggle` on the CoAP server is triggered. The payload of the POST request is the string "Toggle!", which causes the red LED on the CoAP server to change its state.
The left user button of the CoAP client is connected to a counter that is used to read the n^{th} entry of the `char*service_urls` variable. When the button is pressed for the first time, the application reads

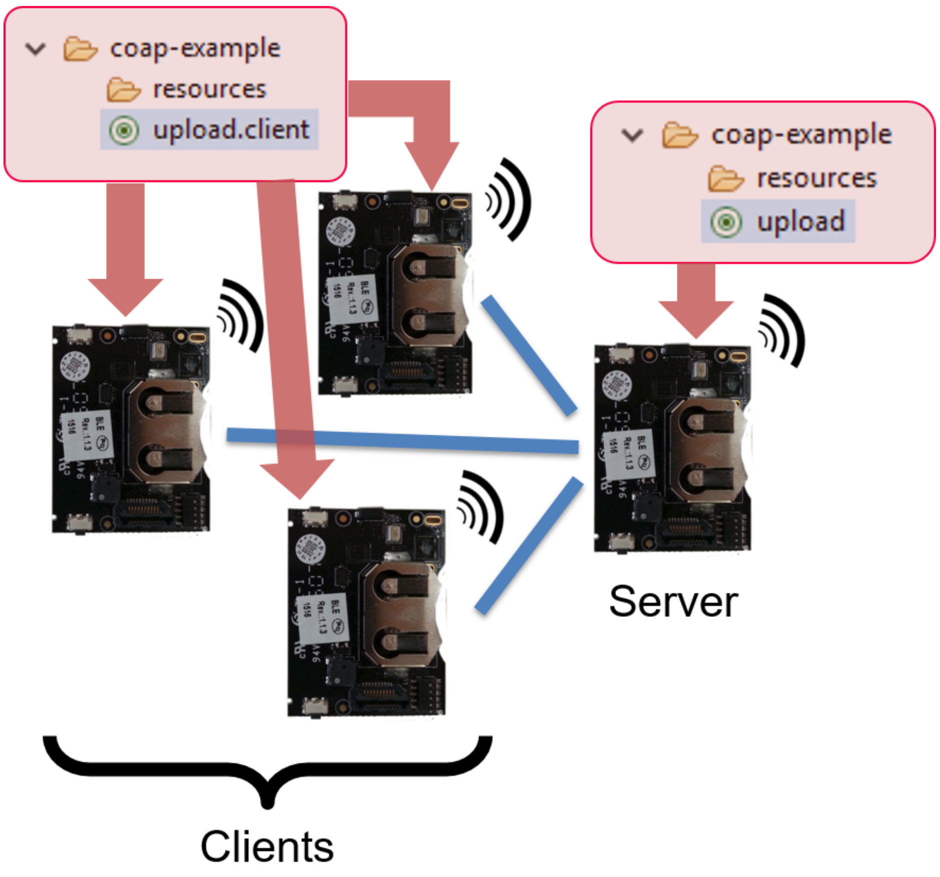


Figure 5: Network setup for the in-class tutorial with an embedded client.

.well-known/core on the server using a GET request and prints the response to the serial output. When pressed a second and a third time, the application will read actuators/toggle and sensors/battery, respectively. If the button is pressed further, the counter exceeds the specified NUMBER_OF_URLS and is reset to 0, such that the application reads again .well-known/core.
The right button is used to toggle the subscription of the resource sensors/button. When pressed for the first time, the client subscribes to the resource. When pressed for a second time, the client cancels its subscription. This resource is connected to the left button on the server: whenever the latter is pressed, an update is sent to all subscribed clients. The client uses then GET on the same URI to retrieve the current event count from the server. This GET request is executed in the client's notification_callback function that is called by the ReST-engine whenever it is notified about a change in the server's resource.

Programming the client. To read a resource, the client first needs to initialize the CoAP engine by calling once coap_init_engine(). Thereafter, one needs to construct and send a CoAP request as follows:

```
#define LOCAL_PORT       UIP_HTONS(COAP_DEFAULT_PORT + 1)
#define REMOTE_PORT      UIP_HTONS(COAP_DEFAULT_PORT)
coap_init_engine();
coap_init_message(request, COAP_TYPE_CON, COAP_GET, 0);
coap_set_header_uri_path(request, "test/hello");
const char msg[] = "";
coap_set_payload(request, (uint8_t *)msg, sizeof(msg) - 1);
COAP_BLOCKING_REQUEST(&server_ipaddr, REMOTE_PORT, request,
    client_chunk_handler);
```

The client_chunk_handler function prints all data received in the response via the serial port. Note that larger messages such as .well-known/core may get transmitted in multiple chunks.

```
void client_chunk_handler(void *response)
{
  const uint8_t *chunk;
  int len = coap_get_payload(response, &chunk);
  printf("%.*s", len, (char *)chunk);
}
```

Extending the CoAP client application. While setting up your nodes as shown in Figure 5 and running the application, you should have noticed that the GET requests for .well-known/core and sensors/battery (if properly implemented in the previous task) can be correctly read by the CoAP client. However, reading actuators/toggle does not results in a LED being toggled on the server side. What is the reason for this behavior, and how can this be fixed? Explain your answer in the space below and paste the relevant code used to fix the problem:

Change further the coap-client.c application such that:

- if the left button is pressed, only the URI test/hello is read and its output is shown to the console;
- if the right button is pressed, instead of subscribing to the resource sensors/button, a PUT request to test/hello is issued. This request should call the previously extended implementation of the resource such that the message returned by the server becomes *"[your name] was here!"*.

You may need to change the length passed to the GET command to read back the entire message. In Contiki, instead of adding ?len=[x] to the URI argument of coap_set_header_uri_path, you need to use coap_set_header_uri_query(request,"len=[x]") before calling COAP_BLOCKING_REQUEST. Verify that your code works properly and paste the relevant code of your implementation in the space below:

Implementing a Theft Detection System

Use a CoAP server and Copper to implement a simple theft protection system following these specifications.

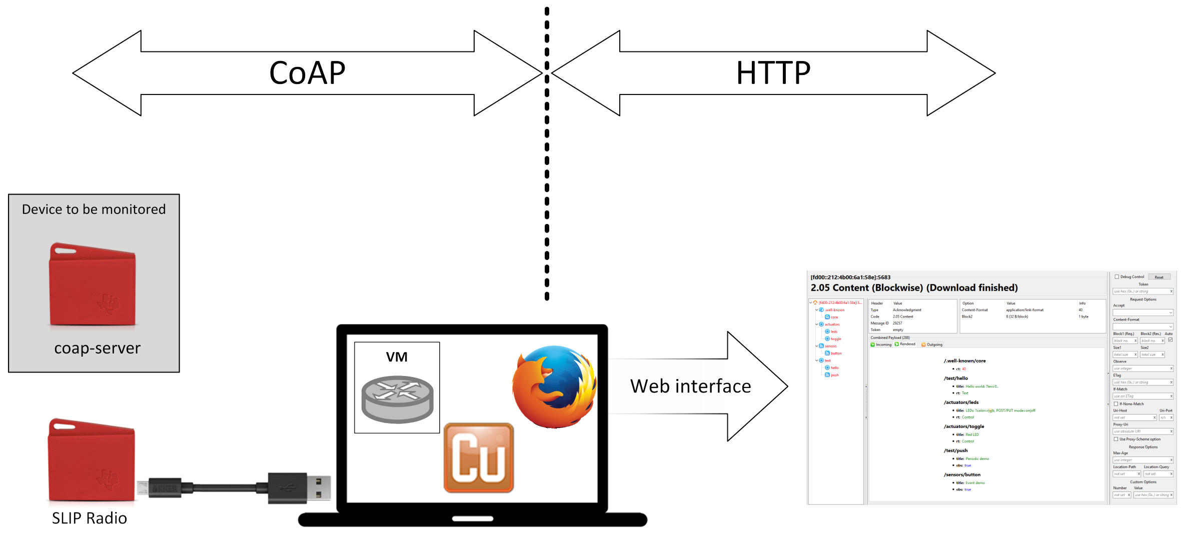


Figure 6: Architecture used for the simple theft protection IoT application.

Architecture. The application should use two SensorTags as shown in Figure 6: one contains your application code, while the other one serves as `slip-radio` for the VM. Use Copper to make sure that your program adheres to the API specified in this document. Unlike the system designed in the previous assignment, this time the entire logic will reside in the device itself. Using a well defined API, the user can authenticate with the device by sending a CoAP GET with a given PIN as payload to `/auth/login`. When you first start the device, the PIN is set to *1234*. The server responds by returning a secret that is valid for 5 minutes. For all other commands, such as arming or disarming the device, this secret must be supplied as part of the URI and acts as an access token to the resources. Using CoAP, the user can also change the

PIN by using PUT on /auth/change?secret=[SECRETfromlogin] with the new PIN embedded in the payload. This change will be persistent throughout a reboot of the SensorTag by using the flash for storage. If an invalid secret is supplied, the alarm is triggered immediately, i.e., the buzzer is turned on and the red LED toggles continuously every 0.5s.

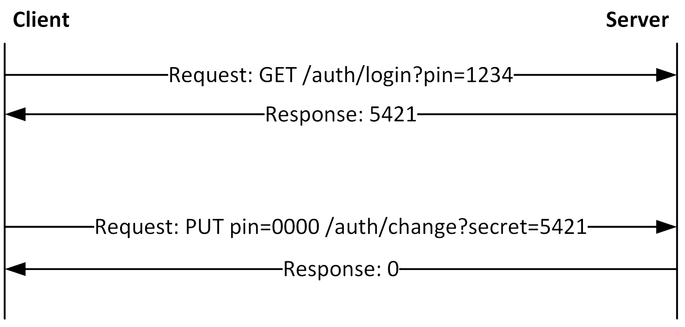

Client
Server
Request: GET /auth/login?pin=1234
Response: 5421
Request: PUT pin=0000 /auth/change?secret=5421
Response: 0

Figure 7: An example for the login sequence between server and client.

In order to save the PIN code, you need permanent storage on your CC2650 SensorTag. To do so, take a look at the use of the following functions in cc26xx/cc26xx-web-demo.c (the header file platform/srf06-cc26xx/common/ext-flash.h contains detailed information as well):

```
ext_flash_open();                      //Initialize storage driver.
ext_flash_close();                     //Close the storage driver.
ext_flash_erase(offset,size);          //Erase storage sectors.
ext_flash_write(offset,size,*data);    //Write to storage sectors.
ext_flash_read(offset, size, *data);   //Read storage sectors.
```

WARNING: If the CC-DEVPACK debugger is attached to the CC2650 SensorTag, the Flash is not accessible. If this is the case, use temporary defaults and do not persistently store them. You can easily check if the flash is available by using:

```
#include "board-peripherals.h"
...
rv = ext_flash_open();

if(!rv) {
  printf("Could not open flash to save config\n");
  ext_flash_close();
  return;
}
...
```

Server. The theft protection system stores the current X,Y and Z values of the build-in accelerometer when a POST or PUT request is sent to /theft/arm. Only the green LED is kept on when the device is not armed. On the contrary, when the system is getting armed, the red LED is toggled every 0.5s six times. Once the system is armed, the red LED blinks for 0.5s once every 10s. Whenever a configurable threshold in G (via /theft/threshold) is exceeded, the device produces an alarm sound using the buzzer, and starts blinking the red LED continuously every 0.5s. The threshold is to be stored in flash and has a default value of 0.1G. The alarm is persistent and can only be stopped by sending a DELETE request to /theft/arm. The latter silences the alarm and also disarms the device (turning on the green LED). The current status of the alarm can be read from /theft/alarm. The device also has a /keepalive resource that needs to be updated using PUT every 60 seconds when the device is armed, otherwise the alarm is triggered immediately. Please note that this resource does not require authentication. The device may be disarmed by sending a DELETE request to `/theft/arm`, or be re-armed overriding the values already in memory by sending a PUT request to `/theft/arm`. Most operations require a prior login via `/auth/login` to obtain a secret that is passed via the URI (with `?secret=[SECRET]`). If an invalid secret is supplied, the alarm is triggered immediately. For a detailed list of the commands relying on prior authentication using a secret, please refer to the API below.

Client. Once you have verified the proper behavior of the server using Copper, also implement an embedded CoAP client that replaces the CC2650 SensorTag running the `slip-radio`. The client uses the default PIN of *1234* to connect to the server. The left button obtains the secret by sending a GET requests to `/auth/login` and then arms the server by sending a POST request to `/theft/arm`. Once the arm command has been sent, the Copper client also needs to send a periodic keep-alive every 30 seconds. Pressing the right button sends a GET requests to `/auth/login` and then disarms the server by sending a DELETE request to `/theft/arm`.

Create a small application that fulfills the aforementioned requirements for both server (theft protection system) and client (the user interface), and that uses the API provided below.

Each group should submit one filled PDF and **all** source files to the TeachCenter within **Monday, 11.12.2017, 23:59 CET**.

Group number:	15
Student 1 (name + ID):	Maja Ovcarik(11775894)
Student 2 (name + ID):	

Finalize Form

API:

Table 2: URI: `/auth/login`

ReST Method	Arguments	Return value
GET	*pin*: the PIN code used for authentication.	A secret that is valid for 5 minutes if the PIN is correct. If an invalid PIN is supplied the alarm is triggered immediately.
POST	-	-
PUT	-	-
DELETE	-	-

Examples:

```
GET /auth/login?pin=1234
```

Table 3: URI: `/auth/change`

ReST Method	Arguments	Return value
GET	-	-
POST	-	-
PUT	*secret,pin*: change the PIN to the one supplied in the payload if the secret is valid.	0 if successful, otherwise -1. If an invalid secret is supplied the alarm is triggered immediately and an empty string (” ”) is returned. If the secret is correct, but no new PIN is supplied, -1 is returned and no alarm is triggered.
DELETE	*secret*: resets the PIN to the default 1234 if the secret is valid.	0 if successful, -1 otherwise. If an invalid secret is supplied the alarm is triggered immediately.

Examples:

```
PUT pin=4321 /auth/change?secret=0451
DELETE /auth/change?secret=0451
```

Table 4: URI: `/theft/arm`

ReST Method	Arguments	Return value
GET	-	0 if not armed, 1 if armed.
POST	*secret*: store the current X,Y,Z from the accelerometer and arm the device if the secret is correct.	0 if successful, -1 otherwise. Turns off the green LED. The red LED flashes 3 times every 0.5s afterwards every 10s for 0.5s. A POST to a armed device or an invalid secret triggers the alarm.
PUT	*secret*: store the current X,Y,Z from the accelerometer and arm the device if not armed and the secret is correct.	0 if successful, -1 otherwise. Turns off the green LED. The red LED flashes 3 times every 0.5s afterwards every 10s. A invalid secret triggers the alarm.
DELETE	*secret*: Disarms the device if the secret is correct. If an alarm is active it is silenced.	0 if successful, -1 otherwise. Turn off the blinking of the red LED and turn on the green LED permanently. If an invalid secret is supplied the alarm is triggered immediately.

Examples:

```
GET /theft/arm?secret=0451
POST /theft/arm?secret=0451
PUT /theft/arm?secret=0451
DELETE /theft/arm?secret=0451
```

Table 5: URI: `/theft/threshold`

ReST Method	Arguments	Return value
GET	*secret*: valid secret to be used.	The current threshold. If an invalid secret is supplied the alarm is triggered immediately and -1 is returned.
POST	-	-
PUT	*secret, threshold*: change the threshold (in hundreds of G) to the one supplied in the payload if the secret is valid. A value of 100 equals 1G.	0 if successful, otherwise -1. If an invalid secret is supplied sound the alarm. If the secret is correct but no new threshold is supplied -1 is returned and no alarm is triggered.
DELETE	-	-

Examples:

```
GET /theft/threshold?secret=0451
PUT thresold=20 /theft/threshold?secret=0451
```

Table 6: URI: `/theft/alarm`

ReST Method	Arguments	Return value
GET	*secret*: valid secret to be used.	0 if the alarm is currently not active, 1 if the alarm is currently active. If an invalid secret is supplied the alarm is triggered immediately.
POST	-	-
PUT	-	-
DELETE	-	-

Examples:

```
GET /theft/alarm?secret=0451
```

Table 7: URI: `/theft/sensor`

ReST Method	Arguments	Return value
GET	*axis*: The axis (x,y or z) of which the acceleration is inquired.	The current acceleration of the specified axis. If no axis is specified, an empty string (" ") is returned instead.
POST	-	-
PUT	-	-
DELETE	-	-

Examples:

```
GET /theft/sensor?axis=x
GET /theft/sensor?axis=y
GET /theft/sensor?axis=z
```

Table 8: URI: `/keepalive`

ReST Method	Arguments	Return value
GET	-	-
POST	-	-
PUT	-	Reset the keepalive timer to 60s. 0 if successful, otherwise -1.
DELETE	-	-

Examples:

```
PUT /keepalive
```