# Sleepy-nodes

Francesco Paolo Culcasi, Alessandro Martinelli, Nicola Messina

February 17, 2017

## Contents

# 1 Introduction

## 1.1 Definitions

Here's a list of terms that will be used frequently in the following report.

- *Sleepy Node:* device that, for energy saving purpose, spends most of its life in a power saving state.
- *Proxy:* any node that is configured to, or selected to, perform communication tasks on behalf of one or more Sleepy Nodes.
- *Regular Node:* any node in the network which is neither a Proxy nor a Sleepy Node.
- *Sleepy Proxy Resource (sp)*: the proxy's resource a sleepy node must send registration requests to. A proxy server lets the sleepy node know the URI of this resource during proxy discovery phase.
- *Container Resource*: a given proxy, for each sleepy node willing to delegate resources to it, must create a resource, the container resource, whose state represents the list of resources the considered sleepy node has delegated to it (one container resource for each delegating sleepy node). Each resource delegated by a sleepy node will have the container resource URI (*Location Path*) as prefix.
- *Location Path:* the URI of a Container Resource
- *Dirty Resource:* a resource that has been modified by an entity who is not the owner of the resource. The dirtiness is reset when the owner of the resource becomes aware of the new value and downloads it.

## 1.2 Low-power sensors: the reachability problem

The large diffusion of sensors happening at our days on one hand enables new concepts like smart houses, on the other hand carries with it new challenges due to their specific characteristic.

One of these challenges consist in being able to communicate with a device which, for energy saving purpose, spends most of its life in a low-power consumption state and periodically go back to an operative mode in order to perform their task, like measuring temperature, after which it goes to sleep again - from here, we call such a device a *sleepy node*; while the device is in a sleepy state, it is not possible to contact it, e.g. in order to obtain the measured temperature value, neither is possible to know a priori when it will wake up.

A solution has been proposed in draft RFC *Sleepy CoAP Nodes* [1].

## 1.3 A possible solution: Sleepy Coap Nodes

The solution proposed in draft RFC *Sleepy CoAP Nodes* make use of a Proxy, i.e. a node in charge of storing the measurements sleepy nodes send to it over time, and making them available to nodes interested in.

Figure 1 shows the main actors of this scenario: the **sleepy node**, the **proxy** and the **regular node**. The last one would be whatever node needs to exchange informations with a sleepy node and, in the general case, will achieve this exchange of informations by means of an intermediate node, the proxy.

Figure 1 also shows the interactions between the three actor just introduced, summarized under three interfaces: *synchronize*, *delegate* and *direct*. The *synchronize* interface is used for communications happening between sleepy nodes and proxies. These interaction are always initiated by the sleepy node, since the proxy has no means of knowing a priori when the sleepy node is active and when instead it is sleeping. In particular, this interface is used for the following exchanges:

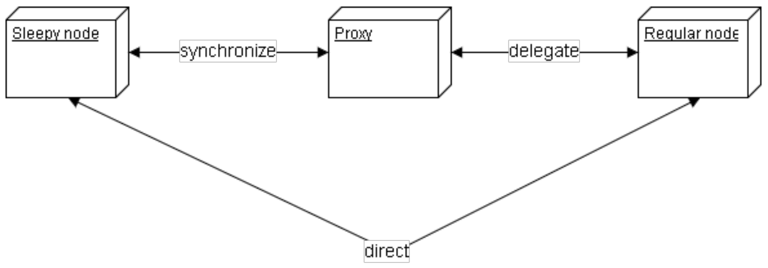


Figure 1: Overview of the actors involved

- a sleepy node wants to discover which proxies are available (if any) and the proxy's resource the sleepy node must send delegation request to. We call this resource *sleepy proxy resource* (sp).
- a sleepy node wants to delegate and initialize one or more of its resources to a proxy, so that the proxy is responsible for them and, when it receives requests for these resources, it will answer with his own copy of them.
- a sleepy node wants to update at proxy the state of a resource it has previously delegated to the it and/or the registration lifetime of the resource itself.
- a sleepy node wants to know if the proxy-local copy of one or more of the resources the sleepy node has delegated have been modified by some regular nodes. This may happen for example for configuration files.

The *delegate* interface allows a regular node to perform information exchanges with a proxy like:

- a regular node wants to discover which proxy are available (if any) and, for each one of them, which sleepy nodes have delegated at least one resource to that proxy and, for a given sleepy node, which resources it has delegated to the proxy.
- a regular node wants to retrieve the state of a resource delegated on a certain proxy.
- a regular node, in order to modify the state of a resource owned by a sleepy node, can modify the proxy-local copy of that resource. This may happen for example for configuration files.

The *direct* interface allows a sleepy node to exchange information directly with a regular node. This kind of interaction is always initiated by the sleepy node, since as we've seen the regular node has no means of knowing a priori when the sleepy node is active and when instead it is sleeping. Examples of such interaction are:

- let's say a sleepy node is a sensor in charge of revealing if someone enters in the room. When this happen, the sleepy node wakes-up and directly notify the interested regular node of such event.
- every time a sleepy node wakes-up, it may contact a regular node in order to request to it if some configuration resources of the sleepy node has to be modified.

Another actor this scenario may include is the Resource Directory. It represents a discovery server, i.e. an entity whose task is to enable nodes to discover all the devices in the network, including sleepy nodes, and retrieve their capabilities. In this paper we will not make use of a Resource Directory, as explained in the assumptions (section 2.1 on page 10).

Let's see which type of operations and message exchanges must by supported by implementations of sleepy nodes, proxies and regular nodes.

## 1.4 Interaction Model

Let's see more in detail the type of messages that are expected to be exchanged in a scenario reproducing the use cases illustrated in section 1.3 on the previous page

### 1.4.1 Synchronize interface

#### Proxy discovery and resource delegation

The first step a sleepy node has to perform is discovering presence of proxies in the network. This may be achieved sending a multicast request to the link-local *all nodes* multicast address, setting *.well-known/core* as Uri-path and resource type *rt=core.sp* as Uri-Query. Any answering proxy will include the *sp* resource a sleepy node must send registration request to. This information will be codified accordingly to [2]. Figure 2 shows this discovery procedure. Note: URI-Host and URI-Port options are omitted for shortness.

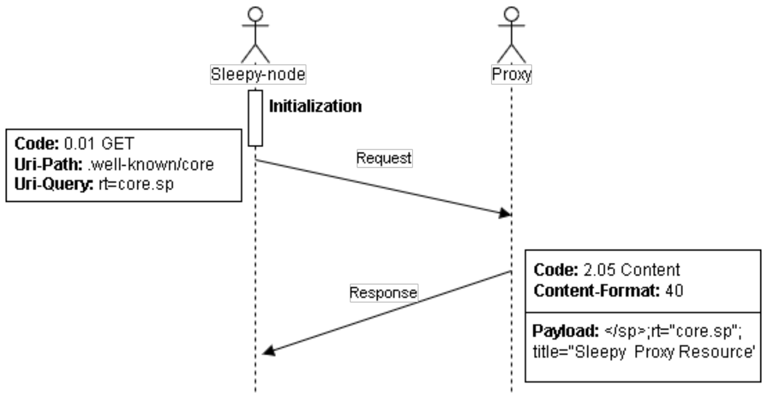


Figure 2: A sleepy node discovers a proxy

Once a sleepy node has discovered a proxy and its *sp* resource, it will presumably delegate some resources to the proxy using the received *sp* resource as Uri-Path. It will also include, in the Uri-Query field, a string whose value represent an identifier for the sleepy node itself. Since a sleepy node can delegate the same resource on different proxies, this option enable regular nodes to understand if resources offered by different proxies belong to the same sleepy node. The proxy will answer specifying in the *Location Path* field the location path of the container resource, i.e. the prefix for the proxy-local copies of the resources delegated by the registering sleepy node. The registration procedure must be implemented as idempotent and it is showed in figure 3.

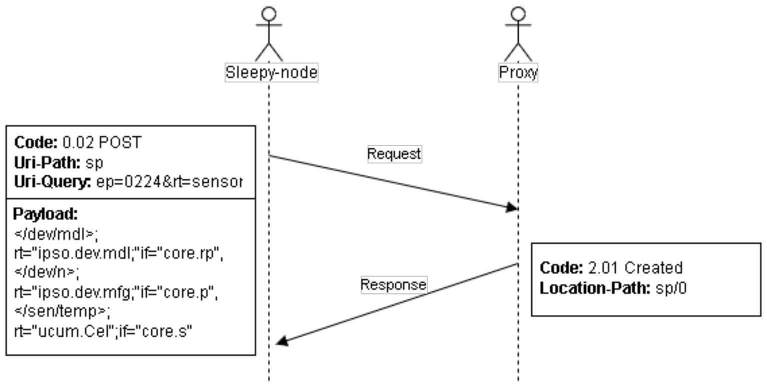


Figure 3: A sleepy node registers three resources with a proxy

### Resource initialization and updates

Registering a resource, however, is not enough for the resource to be reachable from regular nodes. A further step is required, the initialization, shown in 4. Initialization is achieved by means of a PUT operation, with the delegated resource to be initialized placed in the Uri-Path field. Please note the Uri-Path is composed by taking the location path for the specific sleepy node and concatenating it with the resource name, in the example respectively *sp/0* and *sen/temp*, producing *sp/0/sen/temp*.

Another information that may be sent along with this initialization message is the lifetime of the registration of this resource: in the example, by attaching a value of 3600 as lifetime, the sleepy node is indicating that it is supposed to send a temperature value at least every hour in order to keep the registration of this resource valid. When a resource expires, its value on the proxy is no longer valid, thus a NOT_FOUND response will be sent to anyone performing any operation on that resource. If the owner sleepy node happens to receive a NOT_FOUND in response to and update operation on a resource delegated by him, it is supposed to perform a new registration of that resource at the proxy.

Finally, the payload of the initialization request is supposed to contain the initial state of the resource. The proxy will answer with a *Created* message and, from now on until the expiration, the initialized resource is reachable from regular nodes.

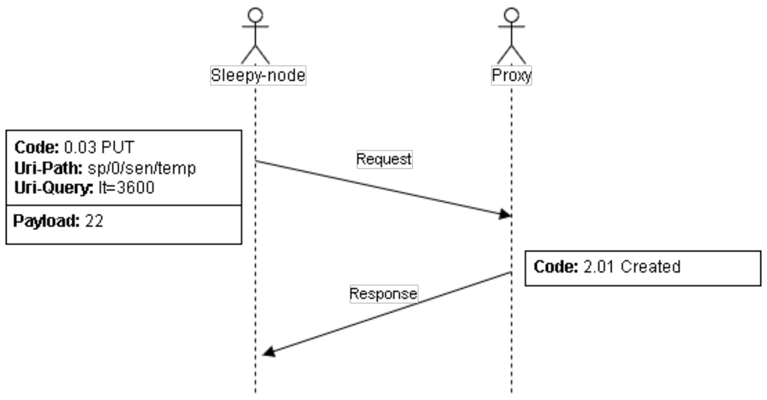


Figure 4: A sleepy node initializes a resource it has previously delegated

Every time a sleepy node wants to update the state of a delegated resource, it repeats the steps executed for initializing a resource, the only difference is Proxy answers with a *2.04 Changed* message, as shown in figure 5.

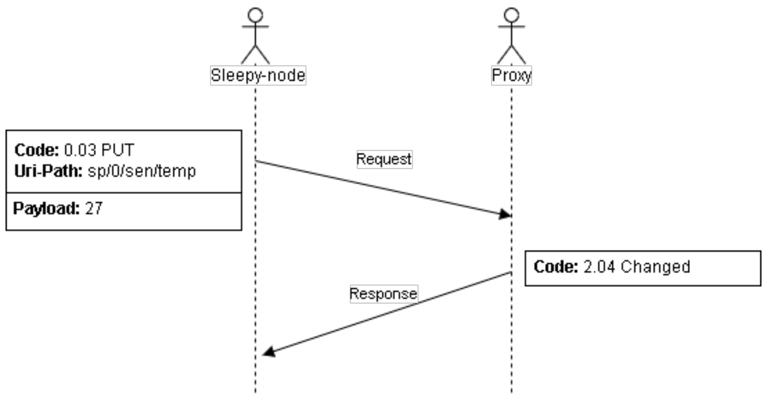


Figure 5: A sleepy node updates a resource it has previously initialized

### Retrieving of resources modified by regular nodes (dirty resources)

Another information the sleepy node may be interested in is the list of its delegated resource that have been modified by regular nodes. This may make sense for example for resources whose state represent a configuration information. There are two ways of obtaining this information: using a PUT [1, sec. 6.3] or a POST [1, sec. 5.6] request.

In the first case, shown in figure 6, the sleepy node simply issues an update request and, if the response coming from the Proxy includes a payload, this field is expected to contains the list in *application/link-format* format of the delegated resources that have been modified by regular nodes. The sleepy node, once have retrieved this list, is supposed to issue a GET request on each resource of the list, in order to retrieve the new values.

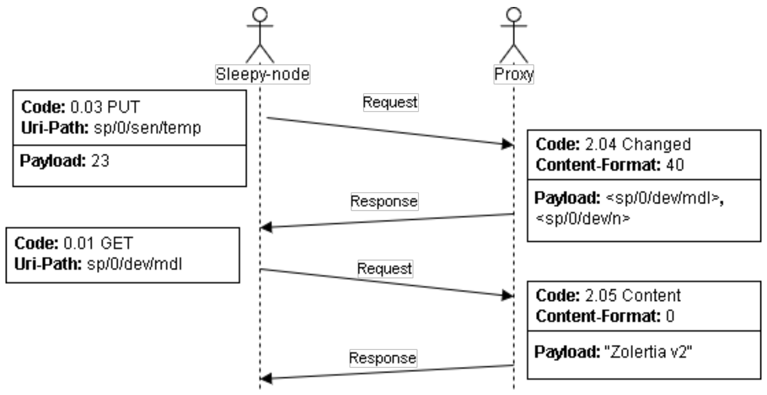


Figure 6: A sleepy node updates a resource it has previously initialized and retrieves the list of its delegated resource that have been modified by regular nodes

The second approach, shown in figure 7 and exploiting a POST request, is different from the previous approach in that the previous one limits the user forcing him to put in the URI-Path field the relative path of the resource to be updated by the sleepy nodes, while the second approach let the user put in the URI-Path the relative path of whatever delegated resources, hence an implementation may exploit this fact an have the proxy answering with different subset of the *dirty* resources basing of what has been putted in URI-Path. Furthermore, the URI-Query option may be used to filter the result.

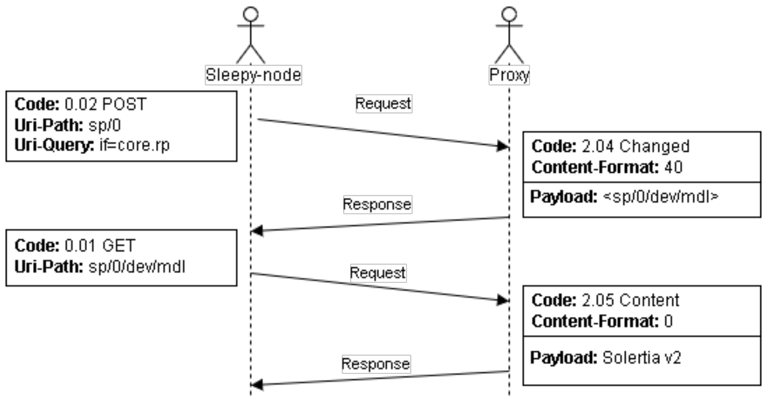


Figure 7: A sleepy node retrieves the list of its delegated resource that have been modified by regular nodes

### 1.4.2 Delegate interface

#### A regular node discovers proxies and the resources they expose

A regular node interested in retrieving the copy owned by the proxy of one or more resources delegated by some sleepy node is supposed to, as first thing, discovery the available proxies and which sleepy nodes have delegated something to them. This procedure is shown in figure 8 on the following page, where it is possible to observe that the regular node use *ep=** as URI-Query. This results in proxies answering with the list of Container Resources they have, and including the associated End Point identifier (sleepy node identifier), thus allowing the regular node to detect if e.g. a resource found on multiple proxy is actually the same resource, delegated by the same sleepy node on more proxies.

Once the regular node has discovered some proxies, it may either

- ask a proxy for the list of resources a certain sleepy node has delegated to it, by means of a GET request issued on the container resource associated to the sleepy node the regular node is interested in
- send a multicast request to multiple proxies asking for the resources compliant with some queries.

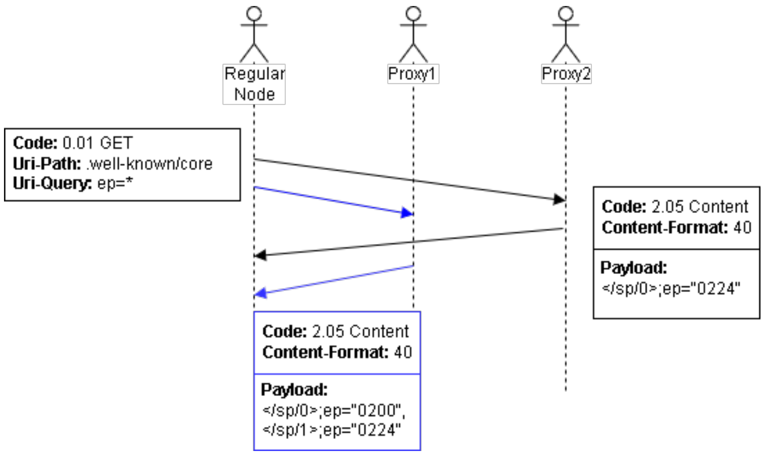


Figure 8: regular node discovers proxy

## Using a resource whose reference is known

Once the regular node knows the URI of the resource it is interested in, it may either read its state, with a simple GET request, that will be answered with a simple 2.05 Content message with the state of the resource as payload, or modify its state. This last option, shown in figure 9, make sense overall for configuration file. In the same figure we may notice that, after the regular node has modified a resource, the sleepy node, following the procedure illustrated in 1.4.1 on page 7, may ask for the modified value.

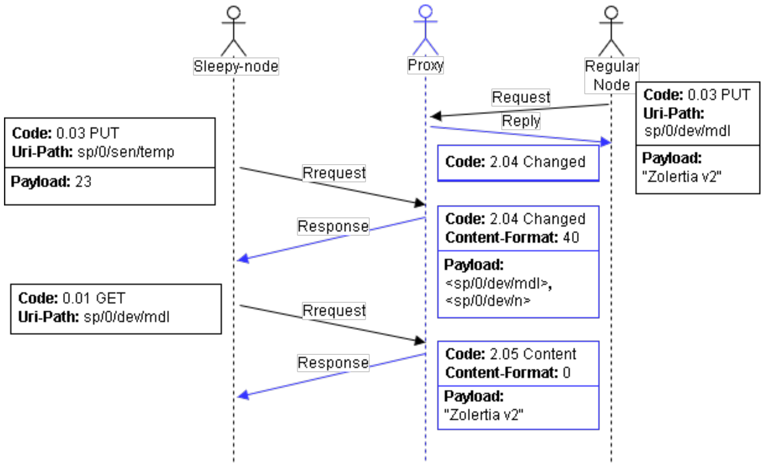


Figure 9: regular node discovers proxy

# 2 Implementation of Coap Sleepy Node Draft

In this section we describe which assumptions has been made in order to create and implement a working scenario aimed to reproduce the interactions between sleepy nodes, proxies and regular nodes as described in [1] (see 2.1). We will se which goals our implementation aimed to achieve, how we were able to achieve them, which implementation problems arose and how they have been solved (see 2.2 on page 13 and 2.3 on page 18).

## 2.1 Assumptions and Goals

In order to implement a working scenario aimed to reproduce the interactions between sleepy nodes, proxies and regular nodes as described in [1], the following considerations were made:

#### Devices and services

- Resource Directory (RD) implementation has been omitted (see 1.3 on page 3).
- In the scenario we are going to represent we will make use of one proxy only; however, the code will support the presence of multiple proxies.
- Even if theoretically the concepts of sleepy node and proxy do not exclude each other, in fact it seems a contradiction to have a sleepy node offering the proxy service, thus we are not going to implement the proxy service on a sleepy node.
- In our implementation the proxy service is supposed to be running on the same machine acting as a gateway.

#### Resource creation and initialization

- We allow resources with a path-like name (e.g. *"/a/b/c"*) to be delegated. From the user viewpoint this should be a mandatory requirement; on the implementation side this will bring to some difficulties, as explained in 2.3 on page 18.
- As we've seen, the procedure used in order to delegate a resource and make it accessible from the outside consist of two step: delegation and initialization. Only after a resource has been initialized it is visible (and thus reachable) from the outside.

  Thus, a regular node issuing a request to a resource that has been delegated but has not been initialized will receive a *NOT_FOUND* response. Instead, if the same request comes from the owner of the resource, it will receive *METHOD_NOT_ALLOWED* as response. An exception of this behavior, of course, is represented by the initialization request, since the sleepy node must be allowed to perform it.

### Proxy discovery

- Since as of now the multicast implementation in contiki is not reliable, Proxy discovery procedure is performed sending unicast messages to the address of the proxy (which is supposed to be known), instead of using multicast communicatiaon.

### Security considerations

- Security mechanisms has been omitted. An important example is: We assume sleepy nodes and regular nodes update different resources; generally speaking, a sleepy node should update the state of the resources representing his measurement, while a regular node should update configuration resource only. However, we don't perform any kind of supervision on this. Thus, if a sleepy node and a regular node happen to update the same resource, there is no guarantee that the value written by the regular node will persist.

### GET operation

- a GET issued to the .well-known/core resource of a certain proxy will be answered with the list of all the resources explicitly delegated at that proxy and all the container resources it exposes.

- as specified in 1.4.2 on page 8, a GET operation performed on a Container Resource associated with a sleepy node sn0 will be answered by the proxy with the list of resources delegated by sn0. Furthermore, we allow the node performing the GET request to specify some query using the URI-Query option in order to filter the resources included in the result.

### Resource deregistration

Draft [1] is not very clear in defining what lifetime is, since it's not clearly defined whether it is considered to be associated with the registration of a resource or with the freshness of its value. Thus, we have had to define the lifetime in a more precise way:

- Given a Delegated Resource which has been initialized, the lifetime is the maximum time interval allowed to elapse between two consecutive write on the initialized resource.

- When the lifetime expires, the value of the associated resource is considered no longer valid and thus the resource is removed from the proxy.

- A resource may or may not have an associated lifetime. In the second case, the resource never expires.

- Given a resource R with initial lifetime lt, every time his owner send an update request for R, the remaining lifetime of R is reset to lt.

- The owner of a delegated resource R may ask to change the remaining lifetime of R (with a shorter or longer one) specifying a new lifetime within an update request.

### Dirty resource retrieval

Draft [1] is not very clear in explaining the difference between PUT and POST operations behaviors regarding the returning of the list of dirty resources to the requesting sleepy-node. Some inconsistency in the interaction description shows up. Thus, we must better define the two behaviors:

- **PUT** besides changing a particular resource value on the proxy, the PUT has the collateral effect of triggering a link-format response containing **all** the dirty resources pertaining to the requesting sleepy node. No filtering for this response is possible.

- **POST** may be used to obtain only a specific part of the dirty resources list. In our implementation we have exploited the POST operation performed on a initialized resource (initialized by its owner) in order to return the list of "dirty" resources. Some filtering is possible:

  - the URI-Path option specifies the URI prefix the dirty resources should have in order to be included in the response;
  - the URI-Query option filters the proxy response by means of a detailed query string.

## 2.2 Sleepy-node

### 2.2.1 Introduction

The *sleepy node* is in general an hard constrained device: it has low computational capabilities (low memory, low processing power) and a small energy availability, since these devices are often battery-powered.

More in detail, conforming to detailed definition presented in [1, sec 1.3], a sleepy-node is a battery-powered node that can switch off its communication interface in order to save energy.

The sleepy-node needs to implement some particular functionalities in order to make available some of its resources to regular nodes. In particular, the sleepy-node can delegate a subset of its resources to the proxy node, so that they are available even if the sleepy-node is sleeping.

In other words, the sleepy-node must implement the *synchronize* interface, as stated in [1, sec 5].

### 2.2.2 High level synchronize interface

The code is organized as a library of functions that implement the functionalities by the *synchronize* interface. The sleepy-node programmer can use these functions in order to communicate with the proxy.

In particular, the programmer has been left with very few things to carry about. The programmer should:

- initialize the proxy informations to correctly perform the discovery operation;
- set up the sleep-awake cycle and handle the entering in low power mode;
- inside the cycle, calling the appropriate methods to retrieve updated informations and/or put new informations on the proxy before going sleeping again.

#### Unicast discovery

Due to multicast implementation instabilities in Contiki, the proxy discovery is a "fake" discovery operation: it is performed on a-priori known proxy IP. It is for that reason that the user is supposed to initialize the proxy with its IPv6 address, as shown in the user code organization, section ( 2.2.2).

This is the result from the assumption explained in 2.1 on page 10.

#### User code organization

The user code should look like the following:

```
<CoAP resource declaration>
<sleepy-node resource declaration>
<buffer declaration for sleepy-node resource value>
```

```
PROCESS_THREAD(sleepy_node, ev, data){
    proxy#0 = proxy_add(IPv6 address of the proxy)

    <initialize CoAP resource>
    <initialize sleepy-node resource with buffer>

    PROXY_REGISTER(resource, proxy#0)
    PROXY_INITIALIZE(resource, proxy#0, 3600s lifetime)
    etimer_set(&et, SLEEP_PERIOD);

    while(true){
        PROCESS_YIELD();
        if(etimer_expired(&et)){
            <turn on radio interface>

            PROXY_CHECK_UPDATES(proxy#0)
            <if needed, work on updated value>
            <read new value from sensor>
            PROXY_PUT(resource, new_sensor_value)
            if(expired){
                <possibly retry PROXY_PUT>
            }

            <turn off radio interface>
            etimer_reset(&et);
        }
    }

    PROCESS_END();
}
```

The above code shows a simple use-case with only one proxy connection and a single resource. It is possible to associate multiple proxies to the interface (giving them different IDs) and multiple resources to every proxy. As a particular use-case, there is the possibility to register the same resource with different proxies, in order to add some redundancy for better resources availability.

Notice that the user is asked to initialize the *standard* CoAP resource, that could be used to implement the *direct* interface (not objective of this project); after that, the user must create a *sleepy-node* resource, that is an extension to the basic CoAP resource introducing additional functionalities useful for sleepy-node operation (more on this topic in section 2.2.3 on the next page).

### Useful functions/macros

The user can use these functions/macros to easily handle sleepy node - proxy interaction:

- **phandler = proxy_add(proxy_ipv6)**: adds a proxy with the specified ipv6 address. Returns an handler that will be used to identify the proxy during the sleepy-node / proxy interactions;
- **PROXY_DISCOVERY(phandler)**: performs proxy discovery ([1, sec 5.1]) on the sleepy node identified by the proxy handler parameter;
- **PROXY_RESOURCE_REGISTRATION(phandler, resource)**: performs the registration of a sleepy-node resource to a given proxy ([1, sec 5.2]);
- **PROXY_RESOURCE_PUT_LT(phandler, resource, lifetime)**: Initializes or updates an already registered resource on a given proxy, with a maximum specified lifetime. After lifetime expiration, the resource will be no more available on the proxy. This macro performs both the operations described in pseudocode in the previous section: PROXY_INITIALIZE() and PROXY_PUT() ([1, sec 5.3, 5.4, 5.5]);
- **PROXY_RESOURCE_PUT(phandler, resource)**: same as before, but with no lifetime. If the resource was already initialized with a certain lifetime, the proxy will reset the timer to the original lifetime; otherwise the resource will never expire ([1, sec 5.4, 5.5]);
- **PROXY_ASK_UPDATES(phandler, local_path_prefix, query)**: explicitly asks the proxy for updates of delegated resources whose URI has a particular prefix. Also, a specific query can be passed in URI-query format as parameter, in order for the proxy to filter out its response ([1, sec 5.6]).

### 2.2.3 Internal implementation

#### Resources handling

*Sleepy-node* resources add some useful functionality to the basic CoAP resource. In fact, the standard coap resource doesn't carry the value for the resource, since the standard resource is designed to respond to requests from a client (the sensor works as a server).

In this case, instead, the sensor works as a client with respect to the proxy. Hence, the sleepy node must keep in some buffer the value of the resources delegated to the proxy, so that synchronization operations can be performed on pre-defined per-resource buffers.

More in detail, the sleepy-node resource object adds the following functionalities:

- it mantains a pointer to a buffer where the value for this resource could be stored: this simplifies update operations, since they can now be performed on the desired buffer in a transparent way to respect the user (the user that issues a CHECK_UPDATES() can find updated values for the modified resources directly in the prepared buffer);

- it can be organized in a list of sleepy-nodes resources (basically, a delegated resources list) so that resources could be easily retrieved by their URI.

This way:

- the resource updates resulting from GET request to the proxy (for example, triggered by the request by the user to download updates from the proxy), directly affect the local buffer of the specified resource.

- a resource in the delegated resources list could be easily found when the proxy returns a list of resources in a link-format payload: the payload is parsed and the URI of each resource is used as search key in the resources list. This way, the buffer for that resource could be retrieved and directly updated.

**Handling the program flow**

Interaction with proxy is performed through a sequence of requests/responses, where requests are initiated by the sleepy node. The sleepy node cannot proceed until the response from the proxy has been elaborated. Thus, in this scenario, a code organization that makes use of a *blocking* send primitive is needed. Without a blocking send implementation, relying only on the Contiki event driven core, the program flow would be very complex, since a relative tricky state machine would be needed.

Blocking functions in Contiki can be implemented and their behavior is similar to the one found in blocking functions present inside any general purpose OS: they block the current *protothread* until an event (in this case, a response from the proxy) is fired and the control is consequently returned to the previously blocked protothread.

Protothreads, however, are implemented using a conceptually very simple stackless mechanism, called *local continuation*, that makes use of simple C syntax tricks: the thread abstraction is performed through ad-hoc generated code enclosed in C macros.

It is for that reason that blocking functions in Contiki are not highly manageable: they cannot be implemented as pure functions; instead they have to be part of the main process function and the only way possible to write a readable, reusable code is to enclose them in macros as well.

The core for the blocking mechanism is implemented by a predefined macro, called COAP_BLOCKING_REQUEST(), whose purpose is to send a CoAP packet to a given destination. In the sleepy-node implementation, this macro is wrapped by SN_BLOCKING_SEND() macro, that tries to keep the code better structured: it takes as argument the following parameters:

- a function *f* that is supposed to construct the request CoAP packet;

- the handler of the involved proxy;

- a list of variable parameters for the function *f*.

And performs the following:

- it construct a CoAP packet using the function passed as parameter;
- it sends the CoAP packet using COAP_BLOCKING_REQUEST().

The response will be handled by a particular callback function that puts the CoAP response packet in a field in the sleepy-node state variable, so that the response content can be accessed inside the normal program flow.

The final interface resulting from this organization has been described in section 2.2.2 on page 14.

## 2.3 Proxy-node

With the term *Proxy* we indicate *"any node in the network which is configured to, or selected to, perform communication tasks on behalf of one or more Sleepy Node"*[1].

In section 1.3 on page 3 of this report we have illustrated the main interactions between Proxy and other nodes.

For the realization of this project we assume a *Proxy* cannot acts as *Sleepy Node* and as *Proxy* at the same time.

Therefore we consider the *Proxy* as a service, implemented by Californium[3] framework, running on the WSN gateway.

To instantiate a *Proxy* we create a new class called *Proxy* extending *CoapServer* (An execution environment for CoAP Resources) [4].

All the needed for the proper functioning of *Proxy* class is an integer *counter* accessed atomically to generate a ContainerResource (resource child of *sp* in the tree of resources) identifier for a new delegating node, and a *map* to connect every Sleepy Node with its respective ContainerResource.

### 2.3.1 Proxy initialization

As already mentioned in previous chapter, every *Proxy* at the beginning has to expose at least a resource with Resource-Type *rt=core.sp*. This is because every Sleepy Node discovers proxies with a GET request to *.well-known/core* of the destination node (**unicast** requests, under the assumptions made in section 2.1 on page 10) with Uri-Query: *rt="core.sp"*.

Adding the resource

$$</sp>; rt = \text{``}core.sp\text{''}; title = \text{``}SleepyProxyResource\text{''}$$

to Proxy, every GET request will return the list of links about resources hosted by a server, also containing *sp*.

As every resource in Californium, the *sp* resource is a class that extends *CoapResource* [4] and overrides its methods in order to handle CoAP requests for the registration of delegated resources as described in Figure 3 on page 5.

### 2.3.2 "Path-like"resource name problem

Let's examine the case of resources with "path-like"name (e.g. "dev/mfg"). At search time an error arises, and the resource is unreachable from external node performing requests on it.

This problem is caused by the current implementation of the resource adding operation, that uses to store resources in a generic tree structure, adding a resource as child of another. The search operation instead takes the resource's name, splits it over the slash character and for every sub-string goes down by a level in the resource hierarchy.

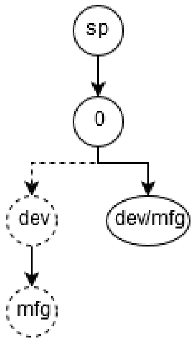


Figure 10: A resource with "path-like"name: how Californium adds it and how it searches for it (dashed arrows)

**Example** : The resource "dev/mfg"has been added as child of "/sp/0/". This is realized with a single resource with name "dev/mfg"(straight arrow in Figure 10).
Sending a request to "/sp/0/dev/mfg"the server searches in the resource tree for following resources:

- *sp* child of / (root)
- *0* child of *sp*
- *dev* child of *0* (not present, dashed in Figure 10)
- *mfg* child of *dev* (not present, dashed in Figure 10)

Since it can't find the last 2 resources, server cannot find a tree node that handles the request and responses with NOT_FOUND.
This leads to some conflicts in resource declaration and management.

For this reason two concepts for the same word *resource* has been introduced:

- external resource: a resource which has in the general case a path-like name, e.g. "dev/mfg"on previous example;
- internal resource: every node in the path-like tree, e.g. "dev"or "mfg"on previous example.

On one hand external resources correspond to the vision an external entity has of the system; on the other hand internal resources are used by the internal implementation of the server, that uses a tree like data structure.

### 2.3.3 Resource Tree handling

In order to hide the actual implementation of the tree data structure from an external user we introduce the concept of Active/Inactive resources and we take advantage of the Visibility attribute.

So we define the following orthogonal concepts:

- **Active:** an internal resource is said to be active if it's the last piece of an external resource path-like name; it inherits all the parameters of the corresponding external resource (e.g. "mfg"in Figure 10 on the previous page). Otherwise the resource is said to be **Inactive** (e.g. "dev"). Inactive resources are introduced just to create a branch toward the Active resource.

- **Visible:** inherits the meaning from the Californium *CoapResource* class. In particular a visible resource is included in the list of links in response to a CoAP GET request to *.well-known/core*.

### ActiveCoapResource

This distinction takes form under the declaration of the *ActiveCoapResource* class, that extends CoapResource and implements attributes and methods to manage Active/Inactive resources. If the flag **active** is set the ActiveCoapResource represents an active resource, otherwise it specifies an inactive resource. This class also distinguishes **dirty** resources (used to generate the list of modified resources seen in Figure 7 on page 8).

The class overrides the *handleRequest()* method in order to respond with code **4.04 Not Found** to every request sent to an inactive resource.

### CoapTreeBuilder

For the purpose of creating the abstraction of external resources, a support class is introduced, *CoapTreeBuilder* class, responsible for the correct creation and removal of internal resources in a tree that takes an internal resource as root (e.g. in our use case the root is a location-path like "sp/0/"). This root together with the default visibility policy are parameters of a CoapTreeBuilder instance.

Every internal resource in the tree is at least an ActiveCoapResource.

When the creation of an external resource is issued, the `add()` method is called with as arguments:

- A new resource declared with the parameters of the external resource;

- A path corresponding to the path-like name of the external resource to be created;

- (optional) A visibility policy that specifies the visibility to be assigned to every internal inactive resources created by this invocation.

`add()` builds a subtree of internal resources from the given path, according to the given - or default, if not specified - *VisibilityPolicy* (an enumerator that can take as value ALL_VISIBLE, that means that every resource in the specified path is set as visible, or ALL_INVISIBLE that sets only the newly created resources as invisible), and adds the subtree as child of the root. The last internal resource is not created inside the `add()` method but must be created outside by the user (since it has to be initialized with external resource's parameters) and passed as `add()`'s attribute.

The removal of a resource is much easier: by invoking the `remove()` method a resource is substituted with a newly created inactive one (it is no more reachable by any request). Only if the resource has no more children it is deleted and the `remove()` is recursively called on the parent if the latter is inactive. If the root is reached `remove()` stops without deleting it, since the root is created outside the CoapTreeBuilder, so it must not be deleted inside the CoapTreeBuilder's `remove()` method.

### 2.3.4 DelegatedResource

For the use case of our project, active and visible concepts can be easily extended in the following way:

- **Active:** a internal resource is created as active if it represents the last part of the path-like name of an external resource that a Sleepy Node registers to a Proxy;
- **Visible:** besides the already explained semantics, an internal resource is visible if it is initialized, while it is invisible if it is registered but uninitialized.

To implement an active resource delegated by the Sleepy Node the *DelegatedResource* class is used. This class extends ActiveCoapResource, and its *active* flag is always set as true. The main task of DelegatedResource is to handle the resource value initialization/update and lifetime expiration.

As seen in section 2.1 on page 10, a delegated resource, in order to be reachable from the outside, needs to be initialized. The initialization is triggered by a CoAP PUT request coming from the owner, that sets the DelegatedResource as visible and observable.

This behavior is introduced in the *handlePUT()* method, that treats the first PUT request from the owner as an initialization operation and any further PUT from whichever node as an update operation for the delegated resource value.

Every update request, beyond updating the value, have different additional behaviors depending on the sender's identity:

- When the PUT request comes from the owner Sleepy Node, handlePUT() must also retrieve the list of all delegated resources updated by nodes different from the owner since the last update/read;
- If instead the PUT request comes from a node different from the owner, handlePUT() must also set the resource as *dirty*, in order to be able to build the list of dirty resources described in the previous point.

After its initialization a DelegatedResource also exposes the handler for a POST method.
This can be used by the owner of the resource to retrieve a list of dirty resources,

in a similar way to respect the PUT method, but with some improvements: this method allows the owner to specify in the Uri-Query a possible attributes list in order to filter out the proxy response.

### Lifetime

Another feature of the delegated resource update requests coming from the owner is the ability to set a lifetime (as defined in 2.1 on page 11) for the sent value. Updates from configuring nodes does not affect the lifetime expiration.
This mechanism is implemented with a *Timer* object which is initialized with the resource lifetime and, as soon as the internal counter expires, triggers a *TimerTask* that removes the resource from the tree if the owner has not updated its value before the expiration.

Every resource whose lifetime has never been set has no Timer instances to run. If the owner of a resource issues a PUT on it specifying a lifetime in the Uri-Query, *handlePUT()* has to stop the running Timer (if any) and to start a new one with the specified expiration period for the task.

Obviously update and lifetime handling bring critical races problem that has been solved executing both in mutual exclusion with the help of *lock* synchronization mechanisms.

## 3 Results

### 3.1 Scenario

In order to test our implementation we set up the following scenario: we deployed two sleepy nodes simulated in Cooja. It was not posible to use Skymote due to their limited ROM size, so we used Z1 motes. We have then deployed an rpl-border-router in cooja, linked to the proxy via Tunslip interface. the global prefix for accessing the capillary sensor network is aaaa::/64.

Border router exposes to the outside of the WSN, through Tunslip, an interface having an IPv6 address of `aaaa::1/64`. This interface is virtualized in the host through a `tun0` interface.

Proxy endpoint should listen on the same `tun0` interface, under the assumption we made on sleepy nodes and proxy running on the same machine.

Also the Regular node simulated by Copper plug-in connects to address `aaaa::1/64` (`tun0` interface).

In Figure 11 Sleepy node networks shows a possible scenario in Cooja, where green node is a border router and yellow nodes are sleepy nodes.

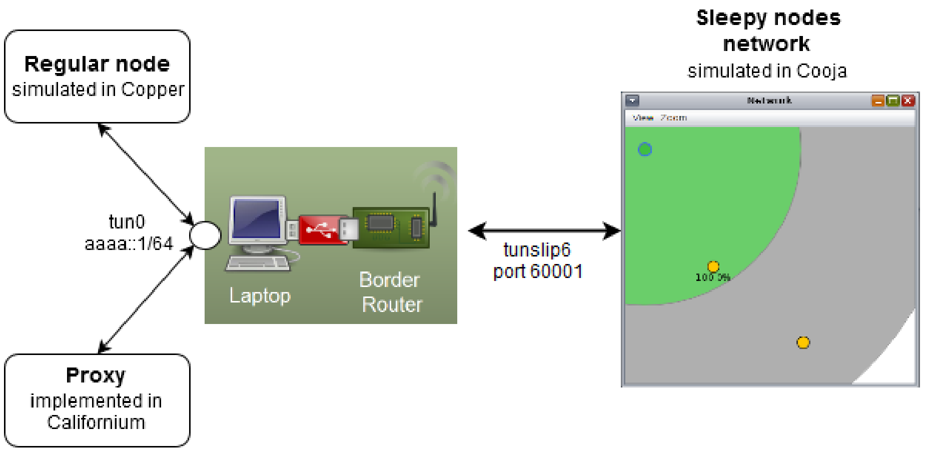


Figure 11: Scenario set up on Cooja

### 3.2 Tests

For the test phase we prepare a pair of sleepy nodes carrying the same firmware. They handle the following resources:

- **dev/n**: a name configurable by a Regular node;
- **vsen/button**: the value ON/OFF of a button used to turn on/off the green led. A lifetime of 100$s$ is asociated to this resource;

- **vsen/counter**: a integer counter incremented by the value stored in /vsen/counter/incr resource every time sleepy node awake. A lifetime of 50$s$ is associated to this resource.

- **vsen/counter/incr**: the increment value for the counter vsen/counter.

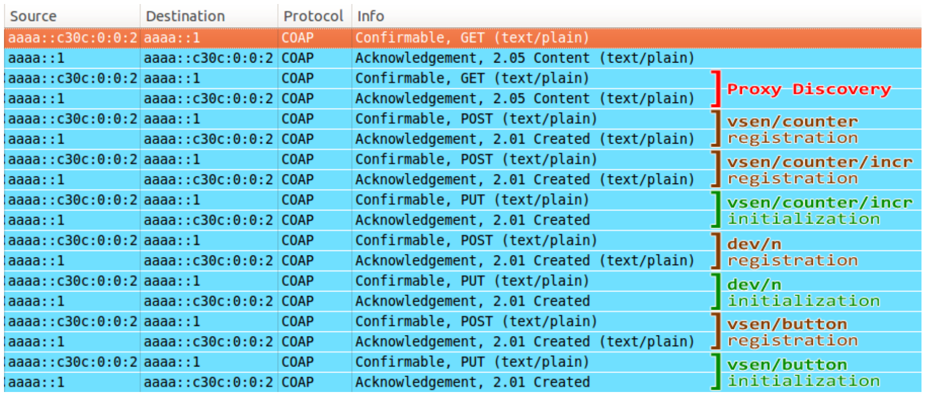
Resources *dev/n* and *vsen/counter/incr* have no lifetime because they carry no time sensible informations. In fact, these ones are classified as configuration resources: they are modified by a regular node in order to instill a particular behavior to the sleepy nodes or to change their internal state. Hence, they must be always reachable by a regular node.

At the beginning of the simulation, the messages shown in figure 12 are exchanged. From the image, we may see the operation performed by the sleepy node before entering the cyclic behavior:

- Proxy discovery by means of GET;

- Resource registration by means of POST;

- Resource initialization by means of PUT.



```
Source           Destination      Protocol Info
aaaa::c30c:0:0:2 aaaa::1          COAP     Confirmable, GET (text/plain)
aaaa::1          aaaa::c30c:0:0:2 COAP     Acknowledgement, 2.05 Content (text/plain)
aaaa::c30c:0:0:2 aaaa::1          COAP     Confirmable, GET (text/plain)
aaaa::1          aaaa::c30c:0:0:2 COAP     Acknowledgement, 2.05 Content (text/plain)
aaaa::c30c:0:0:2 aaaa::1          COAP     Confirmable, POST (text/plain)
aaaa::1          aaaa::c30c:0:0:2 COAP     Acknowledgement, 2.01 Created (text/plain)
aaaa::c30c:0:0:2 aaaa::1          COAP     Confirmable, POST (text/plain)
aaaa::1          aaaa::c30c:0:0:2 COAP     Acknowledgement, 2.01 Created (text/plain)
aaaa::c30c:0:0:2 aaaa::1          COAP     Confirmable, PUT (text/plain)
aaaa::1          aaaa::c30c:0:0:2 COAP     Acknowledgement, 2.01 Created
aaaa::c30c:0:0:2 aaaa::1          COAP     Confirmable, POST (text/plain)
aaaa::1          aaaa::c30c:0:0:2 COAP     Acknowledgement, 2.01 Created (text/plain)
aaaa::c30c:0:0:2 aaaa::1          COAP     Confirmable, PUT (text/plain)
aaaa::1          aaaa::c30c:0:0:2 COAP     Acknowledgement, 2.01 Created
aaaa::c30c:0:0:2 aaaa::1          COAP     Confirmable, POST (text/plain)
aaaa::1          aaaa::c30c:0:0:2 COAP     Acknowledgement, 2.01 Created (text/plain)
aaaa::c30c:0:0:2 aaaa::1          COAP     Confirmable, PUT (text/plain)
aaaa::1          aaaa::c30c:0:0:2 COAP     Acknowledgement, 2.01 Created
```

Proxy Discovery
vsen/counter registration
vsen/counter/incr registration
vsen/counter/incr initialization
dev/n registration
dev/n initialization
vsen/button registration
vsen/button initialization

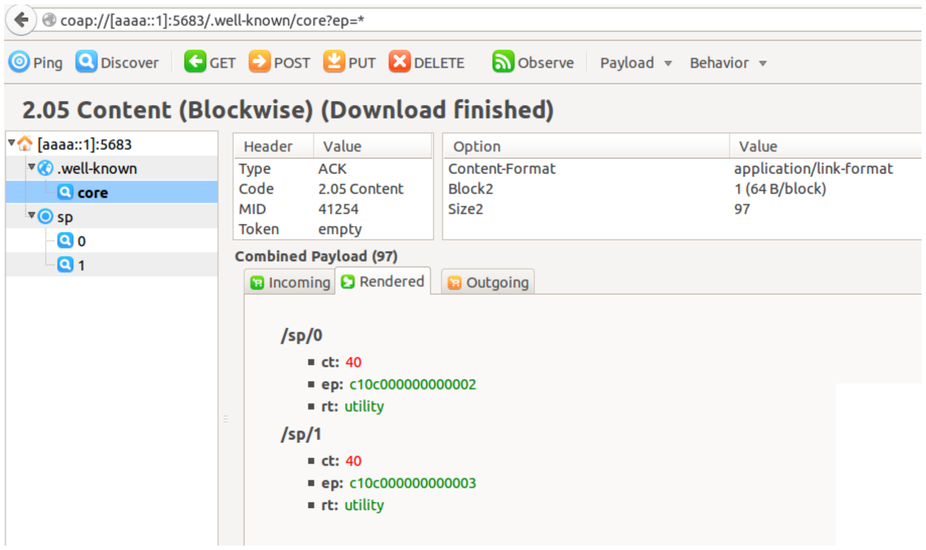
Figure 12: A sleepy node performs proxy discovery, resource registration and initialization

The contents of exchanged packets is available as Wireshark pcap *1_registration_initialization.pcap* file distributed together with this report.

Now, in order to verify the correct delegation and initialization of resources we simulate a regular node through Copper. It will perform the following operations we have documented:

- The discovery of which sleepy node have delegated at proxy, documented in 13 on the next page

- The retrieval of the list of resources delegated by a chosen sleepy node, documented in 14 on the following page. From the same figure we may see that the separation between the internal implementation of the resource tree and the viewpoint of an external observer, as mentioned in 2.3.2 on page 18 has been achieved: the external observer is only aware of the resource it has explicitly delegated.



2.05 Content (Blockwise) (Download finished)

```
[aaaa::1]:5683
  .well-known
    core
  sp
    0
    1
```

| Header | Value |
|---|---|
| Type | ACK |
| Code | 2.05 Content |
| MID | 41254 |
| Token | empty |

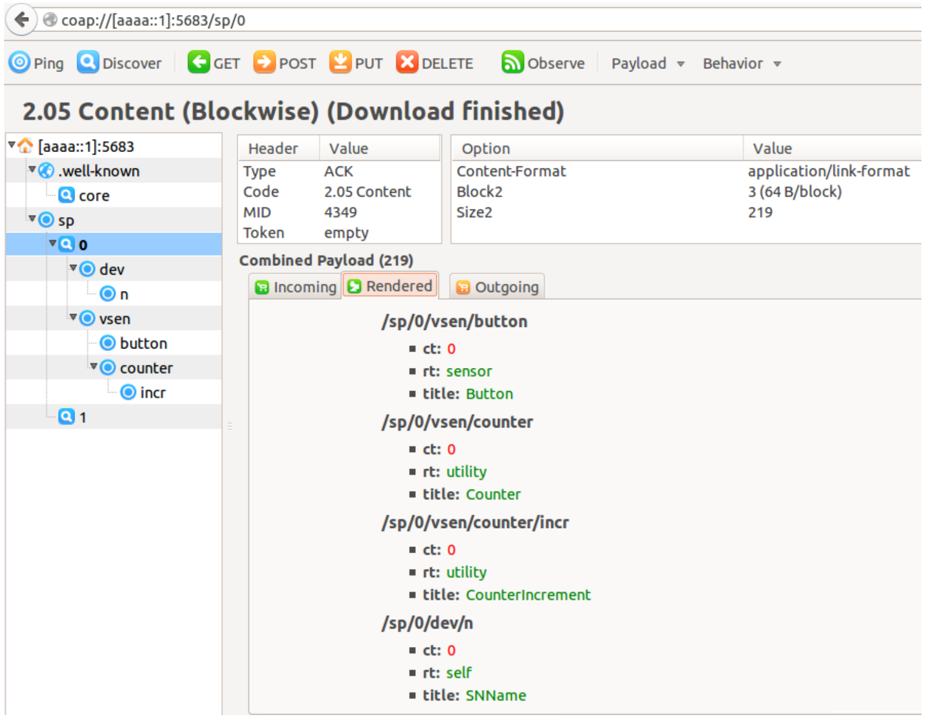
| Option | Value |
|---|---|
| Content-Format | application/link-format |
| Block2 | 1 (64 B/block) |
| Size2 | 97 |

Combined Payload (97)

**/sp/0**
- **ct:** 40
- **ep:** c10c000000000002
- **rt:** utility

**/sp/1**
- **ct:** 40
- **ep:** c10c000000000003
- **rt:** utility

Figure 13: A regular node discovers sleepy nodes who have delegated at proxy



2.05 Content (Blockwise) (Download finished)

| Header | Value |
|---|---|
| Type | ACK |
| Code | 2.05 Content |
| MID | 4349 |
| Token | empty |

| Option | Value |
|---|---|
| Content-Format | application/link-format |
| Block2 | 3 (64 B/block) |
| Size2 | 219 |

Combined Payload (219)

/sp/0/vsen/button
- ct: 0
- rt: sensor
- title: Button

/sp/0/vsen/counter
- ct: 0
- rt: utility
- title: Counter

/sp/0/vsen/counter/incr
- ct: 0
- rt: utility
- title: CounterIncrement

/sp/0/dev/n
- ct: 0
- rt: self
- title: SNName

Figure 14: A regular node retrieves the list of resources delegated by a chosen sleepy node

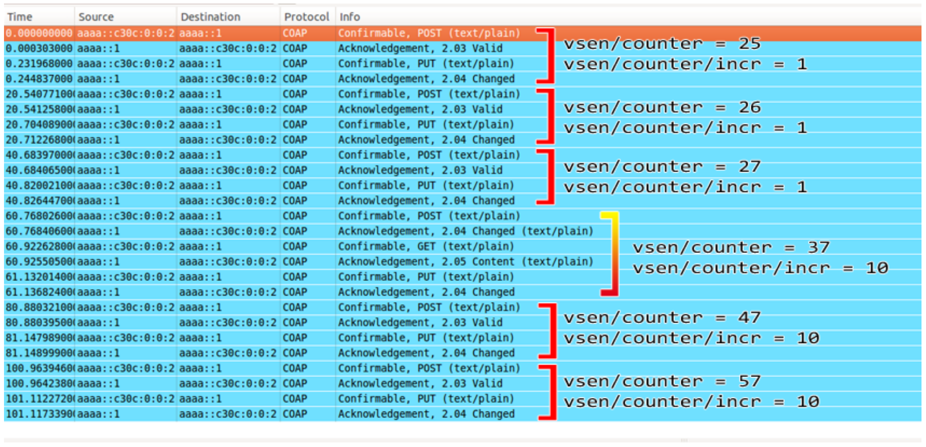
After initial operations, the sleepy node enters in the main cycle, where

- it goes to sleep and wakes-up after 20 seconds;
- at wake-up
  - it checks for dirty resources;
  - if there are any, it retrieves the new value;
  - it updates *vsen/counter* incrementing its value by *vsen/counter/incr*.

At this point, if we try to perform observing on the *vsen/counter* resource we may be able to see the value of *vsen/counter* advancing by 1 unit every 20 seconds.

Let's now suppose a regular node modifies the value of *vsen/counter/incr*. The sleepy node will be informed of this change as soon as it wakes-up and checks for dirty resources. Being *vsen/counter/incr* dirty resource, its value will be retrieved by the sleepy node and stored in its local resource buffer. From now on, the *vsen/counter* notifications received by an observing regular node will present an increment in *vsen/counter* value by the new *vsen/counter/incr*.

What we've just exposed is shown in figure 15. Notice sleepy node performs communication tasks about every 20 seconds because, as previously stated, during these 20 seconds it is sleeping.

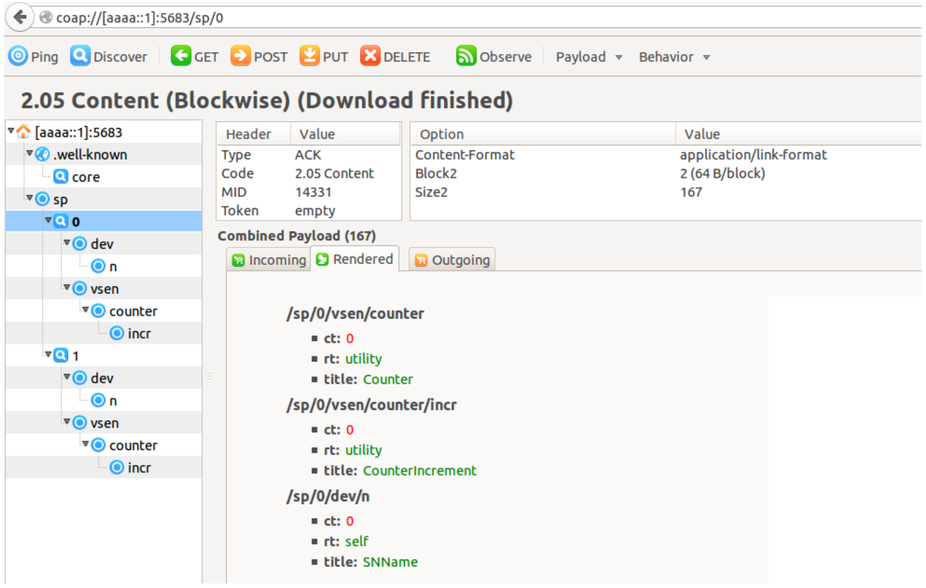


| Time | Source | Destination | Protocol | Info |
|---|---|---|---|---|
| 0.000000000 | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, POST (text/plain) |
| 0.000303000 | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.03 Valid |
| 0.231968000 | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, PUT (text/plain) |
| 0.244837000 | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.04 Changed |
| 20.54077100( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, POST (text/plain) |
| 20.54125800( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.03 Valid |
| 20.70408900( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, PUT (text/plain) |
| 20.71226800( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.04 Changed |
| 40.68397000( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, POST (text/plain) |
| 40.68406500( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.03 Valid |
| 40.82002100( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, PUT (text/plain) |
| 40.82644700( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.04 Changed |
| 60.76802600( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, POST (text/plain) |
| 60.76840600( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.04 Changed (text/plain) |
| 60.92262800( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, GET (text/plain) |
| 60.92550500( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.05 Content (text/plain) |
| 61.13201400( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, PUT (text/plain) |
| 61.13682400( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.04 Changed |
| 80.88032100( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, POST (text/plain) |
| 80.88039500( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.03 Valid |
| 81.14798900( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, PUT (text/plain) |
| 81.14899900( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.04 Changed |
| 100.9639460( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, POST (text/plain) |
| 100.9642380( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.03 Valid |
| 101.1122720( | aaaa::c30c:0:0:2 | aaaa::1 | COAP | Confirmable, PUT (text/plain) |
| 101.1173390( | aaaa::1 | aaaa::c30c:0:0:2 | COAP | Acknowledgement, 2.04 Changed |

Figure 15: The sleepy node cycle during the update of vsen/counter/incr

Let's now suppose we don't press the sensor button for more than 100 seconds, thus letting the lifetime of button resource to expire. After the expiration, if we issue a GET on */sp/0*, we obtain the result in figure 16 on the next page, from which we are able to notice the absence of */vsen/button* in the returned list. Furthermore, this example proofs delete operation is performed in the correct way: internal resource "button"is correctly removed, while his parent "vsen"is not deleted since it is needed in order for "counter"and "incr "to be reachable.

If a delegated resource expires on the proxy, the sleepy node will realize the expiration event when it will wake up from sleeping: when the sleepy node will try a PUT operation on the expired resource, the proxy will respond with a NOT_FOUND message. Thus, the sleepy node will re-register the same resource on the proxy and possibly it will retry the PUT operation. Sleepy node and proxy operations generated in this scenario are reported in figure 17 on the following page.



2.05 Content (Blockwise) (Download finished)

```
[aaaa::1]:5683
  .well-known
    core
  sp
    0
      dev
        n
      vsen
        counter
          incr
    1
      dev
        n
      vsen
        counter
          incr
```

| Header | Value |
|---|---|
| Type | ACK |
| Code | 2.05 Content |
| MID | 14331 |
| Token | empty |

| Option | Value |
|---|---|
| Content-Format | application/link-format |
| Block2 | 2 (64 B/block) |
| Size2 | 167 |

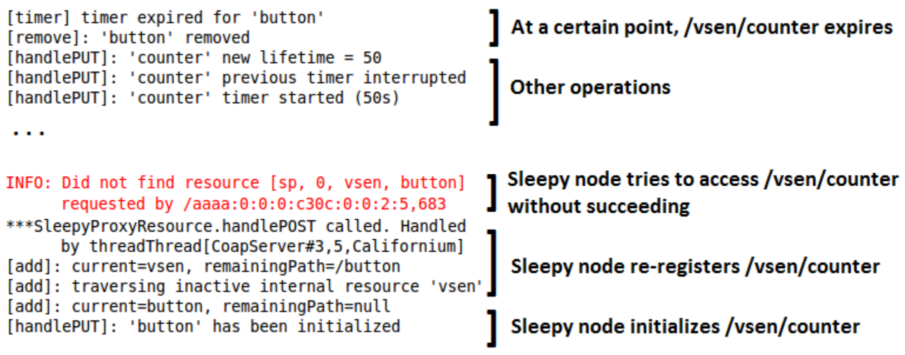
Combined Payload (167)

/sp/0/vsen/counter
- ct: 0
- rt: utility
- title: Counter

/sp/0/vsen/counter/incr
- ct: 0
- rt: utility
- title: CounterIncrement

/sp/0/dev/n
- ct: 0
- rt: self
- title: SNName

Figure 16: After button resource expiration, resources with a common prefix with button are still visible

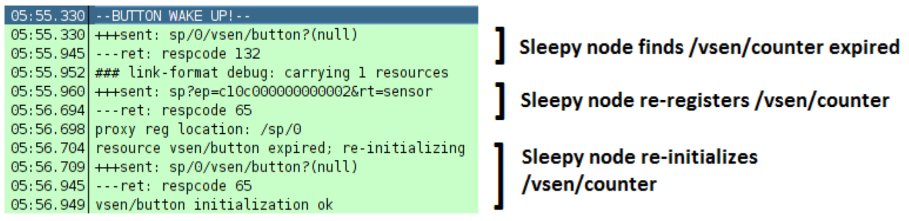


```
[timer] timer expired for 'button'
[remove]: 'button' removed
[handlePUT]: 'counter' new lifetime = 50
[handlePUT]: 'counter' previous timer interrupted
[handlePUT]: 'counter' timer started (50s)
 ...

INFO: Did not find resource [sp, 0, vsen, button]
      requested by /aaaa:0:0:0:c30c:0:0:2:5,683
***SleepyProxyResource.handlePOST called. Handled
      by threadThread[CoapServer#3,5,Californium]
[add]: current=vsen, remainingPath=/button
[add]: traversing inactive internal resource 'vsen'
[add]: current=button, remainingPath=null
[handlePUT]: 'button' has been initialized
```

At a certain point, /vsen/counter expires
Other operations
Sleepy node tries to access /vsen/counter without succeeding
Sleepy node re-registers /vsen/counter
Sleepy node initializes /vsen/counter

(a) Proxy resource expiration and sleepy node re-registration sequence



```
05:55.330 --BUTTON WAKE UP!--
05:55.330 +++sent: sp/0/vsen/button?(null)
05:55.945 ---ret: respcode 132
05:55.952 ### link-format debug: carrying 1 resources
05:55.960 +++sent: sp?ep=c10c000000000002&rt=sensor
05:56.694 ---ret: respcode 65
05:56.698 proxy reg location: /sp/0
05:56.704 resource vsen/button expired; re-initializing
05:56.709 +++sent: sp/0/vsen/button?(null)
05:56.945 ---ret: respcode 65
05:56.949 vsen/button initialization ok
```

Sleepy node finds /vsen/counter expired
Sleepy node re-registers /vsen/counter
Sleepy node re-initializes /vsen/counter

(b) Sleepy-node re-registration sequence

Figure 17: Resource re-registering and re-initialization

## References

[1] T. Zotti, E. Dijk, *CoAP Sleepy Node draft, rev. 4*

[2] Z. Shelby, *Constrained RESTful Environments (CoRE) Link Format*

[3] *Californium* github repository

[4] *LDP-CoAP Framework*

[5] *Contiki* OS documentation