Università di Pisa

FACOLTÀ DI INGEGNERIA
Corso di Laurea Specialistica in Ingegneria Informatica

A gateway-based scalable architecture for the Internet of Things

Candidato:
Giacomo Tanganelli

Relatori:
Prof. Enzo Mingozzi
Prof. Luciano Lenzini
Dott. Claudio Cicconetti

Anno Accademico 2011/2012

Contents

List of Figures

List of Tables

Listings

Abstract

The Internet of Things is a wide research field, and a lot of problems still have to be solved. This work aims to solve some of them, with particular regard to identifications and end point communication using only standard protocols. It consists of a network of gateways connected to each other by an overlay network. Clients and servers, on the other hand, are connected only to their local gateway. In this way distributed resources are seen, by clients, as logically connected to their local gateway. The entire work has its focus on a SOA view, where clients ask for services and obtain responses without necessarily knowing anything about the overall system. Aiming to use standard protocols only, clients use CoAP for interacting with the overall system while a p2p network, called XMHT, forms the overlay network and is responsible for localizing resources. CoAP is also used for communications between gateways. Moreover the proposed architecture implements a new way for the observing feature that aims to solve performance issues when CoAp servers are hosted in constrained devices.

For demonstration purposes a little prototype has been made. The prototype consists of two Alix, which operate as a gateway, and two different boards that are used to test a realistic scenario. The first board, in fact, can be used to remotely control a light bulb and also to detect the current consumption of the bulb itself. The second board, instead, has a light sensor and a water consumption sensor. In the testbed, clients send COAP requests only to their local gateway controlling, in this way, the whole environment.

Chapter 1

Introduction

Internet of Things (IoT) is a new concept that is growing more and more popular during the last years. The basic idea behind it is the pervasive presence around people of a variety of objects which are able to interact with each other to reach a common goal. On the other hand there are a lot of problems that are still open and have to be solved. One of the most important problem is the localization of a service, a resource or, in general, of an object.

Imagine a scenario where there are some smart objects, i.e. smartphones, smart plugs and sensors in general, that are connected through the Internet. To interact with them, an user has to know where they are and also how he can connect to them. This can become a great issue especially if resources can be created and deleted quickly and without any kind of prefixed rule. As [21] says, identifiers will play a critical role for retrieval of information from repositories and for lookup in global directory lookup services and discovery services, to discover the availability and find addresses of distributed resources.

Another problem could be how objects connect to the Internet. In fact, many smart objects are also constrained devices with battery power supply. Thus they can not always be switched on but instead they can go into a sleep mode silently. So an user can not know beforehand whether the object is turned on or off.

In the present time there are a lot of proprietary special purpose solutions that implement something like IoT but all use proprietary protocols that are not interoperable with each other. An user can interact with his devices but anyway devices from different manufacturers can not communicate with each other for exchanging information.

In [21] authors define some crucial behaviours that an IoT architecture should have. The architecture should have well-defined and granular layers, in order to foster a competitive marketplace of solutions, without locking any users into using

a monolithic stack from a single solution provider. Like the Internet, the IoT architecture should be designed to be resilient to disruption of the physical network and should also anticipate that many of the nodes will be mobile, they may have intermittent connectivity and may use various communication protocols at different times to connect to the IoT. IoT nodes may need to dynamically and autonomously form peer networks with other nodes, whether local or remote, and this should be supported through a decentralised, distributed approach.

This work aims to solve those problems. The idea is to develop a distributed upper layer that permits to an user to interact with a remote device without knowing its location. Moreover it is important to use only standard protocols that permit future works and interoperability between different implementations. In the future all objects will have a built-in device for remote interactions and all devices will be able to communicate with each other. The architecture proposed is totally hidden from an user's point of view and can work with the actual Internet architecture.

A typical future scenario is the smart home. The home will recognize its owner's face at the entrance and thanks to an electronic key in his pocket will be automatically detected. The home will also switch lights on and off when someone is in a room. Moreover the home will take care of temperature, combining data from outdoor and indoor temperature, weather forecast from the Internet, and user preferences. It will adjust the house energy consumption to the real needs of the family members, and most importantly it will help them save money. In fact the home will synchronise all the tasks performed by the domestic appliances with the cheaper pricing times of the utility companies. Displays around the home will indicate the current consumption, carbon emissions, and will send alarm signals during high peaks of consumption. All TV sets will be connected to each other, so when the owner moves from a room to another, the TV set in the first room will be switched off while the TV set in the second room will be switched on automatically. On the other hand an utility company will be able to monitor water/electricity consumption in a certain area. With this information the company could use some policy for optimizing water/electricity distribution service thus saving a lot of money. Furthermore, in the future big buildings will have complex systems where illumination, central heating and ventilated rooms will be controlled by taking into account the level of rooms occupancy, external light and temperature. This will be possible by introducing sensor networks that will work collaboratively to collect, share and disseminate information.

The thesis is organized as follows: Chapter 2 presents the state of art concerning the wireless sensor networks, with the main technologies developed in this field. Chapter 3 regards the software solution proposed by this work with an exhaustive

explanation and implementation details. There is also the infrastructure explanation that is the core part of this thesis. In Chapter 4 there is a performance evaluation that aims to present statistic results regarding distributed accesses to the overall system. Chapter 5 presents a prototype with the overall system made of sensors and the new infrastructure proposed in this work. Chapter 6 is about conclusions and future works.

Chapter 2

Technologies Overview

The Internet of Things is still growing up really quickly in the present time, a lot of new dedicated technologies are been developed with particular care to embedded systems and low power networks. In this section some technologies will be presented.

2.1 IP technologies

2.1.1 802.15.4 Physical Layer

802.15.4 is a simple, low-cost communication network that allows wireless connectivity in applications with limited power and relaxed throughput requirements. The main objectives of an 802.15.4 network are ease of installation, reliable data transfer, short-range operation, extremely low cost, and a reasonable battery life, while maintaining a simple and flexible protocol. For more information see [10]

Protocol Architecture

The architecture is based on two main layer: a Physical Layer, which contains the radio frequency (RF) transceiver along with its low-level control mechanism, and a MAC Layer that provides access to the physical channel for all types of transfer. The Figure 2.1 shows an overview of the architecture.

The Physical Layer can operate in one of the following frequencies bands:

- 868.0-868.6 MHz: Europe
- 902-928 MHz: North America
- 2400-2483.5 MHz: Worldwide

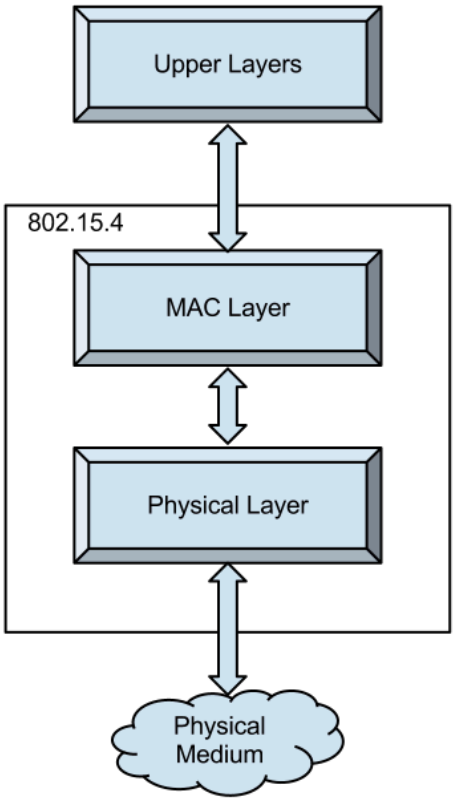


Figure 2.1: 802.15.4 Protocol architecture

The MAC Layer provides data transfers throw the Physical Layer and also manage the Physical Layer with a lot of features like: beacon management, channel access, frame validation, acknowledged frame delivery, association, and disassociation.

Upper Layers are not defined in this standard but exists specifications, such as 6LoWPAN (Section 2.1.2) and ZigBee, which build on this standard to propose integral solutions.

Network Model

The standard defines two device types; a full-function device (FFD) and a reduced-function device (RFD). The FFD can operate in three modes serving as a personal area network (PAN) coordinator, a coordinator, or a simple device. An FFD can talk to any other device and can also relay messages in star topology. An RFD is intended for applications that are extremely simple, so they can communicate only with a FFD and can never become a coordinator. All devices operating on a network shall have unique 64-bit addresses. This address may be used for direct communication within the PAN, or a short address (16-bit) may be allocated by

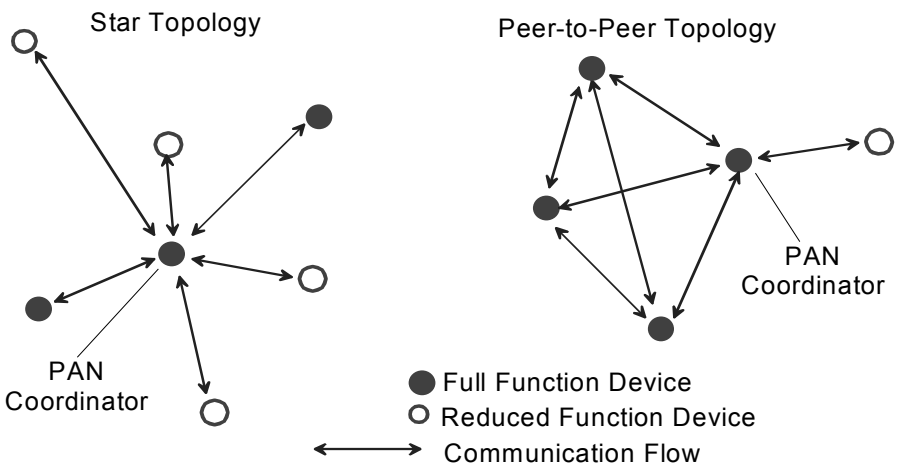


Figure 2.2: 802.15.4 Network Topologies

the PAN coordinator when the device associates and used instead.

Topologies There are two type of topology: Peer-to-peer topology and Star topology. In the Star topology the communication is established between devices and a single central controller, called the PAN coordinator. The PAN coordinator acts as a relay and interconnects FFDs and RFDs.

The peer-to-peer topology also has a PAN coordinator; however, it differs from the star topology in that any device may communicate with any other device as long as they are in range of one another. Peer-to-peer topology allows more complex network formations to be implemented, such as mesh networking topology. A peer-to-peer network can be ad hoc, self-organizing, and self-healing. It may also allow multiple hops to route messages from any device to any other device on the network. Such functions can be added at the higher layer, but are not part of the standard.

Physical Layer Access Method

There are two types of channel access mechanism, depending on the network configuration. Nonbeacon-enabled PANs use an unslotted CSMA-CA channel access mechanism. Each time a device wishes to transmit data frames or MAC commands, it waits for a random period. If the channel is found to be idle, following the random backoff, the device transmits its data. If the channel is found to be busy following the random backoff, the device waits for another random period before trying to access the channel again. Acknowledgment frames are sent without using a CSMA-CA mechanism.

Beacon-enabled PANs use a slotted CSMA-CA channel access mechanism, where the backoff slots are aligned with the start of the beacon transmission. The backoff slots of all devices within one PAN are aligned to the PAN coordinator. Each time a device wishes to transmit data frames during the Content Access Period (CAP), it locates the boundary of the next backoff slot and then waits for a random number of backoff slots. If the channel is busy, following this random backoff, the device waits for another random number of backoff slots before trying to access the channel again. If the channel is idle, the device begins transmitting on the next available backoff slot boundary. Acknowledgment and beacon frames are sent without using a CSMA-CA mechanism.

Data Model

Three types of data transfer transactions exist. The first one is the data transfer to a coordinator in which a device transmits the data. The second transaction is the data transfer from a coordinator in which the device receives the data. The third transaction is the data transfer between two peer devices. In star topology, only two of these transactions are used because data may be exchanged only between the coordinator and a device. In a peer-to-peer topology, data may be exchanged between any two devices on the network; consequently all three transactions may be used in this topology.

The mechanisms for each transfer type depend on whether the network supports the transmission of beacons. A beacon-enabled PAN is used in networks that either require synchronization or support for low- latency devices, such as PC peripherals. If the network does not need synchronization or support for low-latency devices, it can elect not to use the beacon for normal transfers. However, the beacon is still required for network discovery.

There are four fundamental type of frames: data, acknowledgement, bacon and MAC command frame. The data frame is used for all transfers of data, the acknowledgement frame used for confirming data reception, bacon frame is used by the coordinator for synchronizing scope and the MAC command frame is used for handling all MAC peer entity control transfers.

Optionally the coordinator can use a superframe. It consists of sixteen equal length slots which can be further divided into an active part and an inactive part. Each superframe is bounded by network beacons sent by the coordinator. The coordinator can assign the inactive period to some applications, this portions are called guaranteed time slots (GTSs) and are often used for low-latency applications.

2.1.2 6LoWPAN

6LoWPAN is an adaptation layer collocated at the bottom of IPv6 for low power networks (802.15.4 networks), it consists of an IPv6 stub network with three type of devices: Host, Routers and Border Routers. The Border Routers is intended for the communication to the Internet. 6LoWPAN lives over the underlying layer, 804.15.4 Physical Layer, it is possible to have bacon or non-bacon WPAN the only thing necessary is that each node, in the same 6LoWPAN network, share the same IPv6 prefix.

Addressing

IPv6 addresses are composed by a 64-bit prefix and a 64-bit field obtained from the underlying layer (802.15.4 address). The 64-bit field passed throw layers is defined both for IEEE 802.15.4 64-bit extended addresses, which are statically assigned to devices by the manufacturer, and for 16-bit short addresses chosen by the PAN coordinator as mentioned in 2.1.1.

Routing

6LoWPAN supports both layer-2 as well as layer-3 multi-hop routing and forwarding, referred to as mesh-under and route-over, respectively. The mesh-under routing mechanism is totally hidden by the IPv6 point of view and IPv6 see all nodes attached to the same IPv6 link. On the other hand route-over is accomplished based on the IPv6 addresses and is necessary a routing protocol like RPL (see 2.1.3 for more details).

Frame Format

6LoWPAN defines also a frame format for encapsulating message into the underlying layer, in details the 802.15.4 MTU is only 127 byte while the IPv6 minimum MTU is 1280 byte. It obviously necessary a method for fragmentation and reassembly of frames but 6LoWPAN moves forward and implements header compression mechanisms to reduce IPv6 overhead.

For more informations see [14].

2.1.3 RPL

RPL is a pure route-over solution used on the top of 6LoWPAN protocol for routing purposes. It is a distance vector multi hop protocol where each route is optimized for traffic delivery to a selected number of sink nodes. RPL uses

a Destination Oriented Directed Acyclic Graph (DODAG) for each sink node, where the sink node is the root of the DODAG. In this scenario RPL builds a Direct Acyclic Graph (DAG), made of a collection of DODAGs, that connects all nodes in the network. If more routing requirements are needed is possible to use more RPL instance, each one with a particular routing objective function, more instances can coexist inside the same network and operates independently. Each RPL node, however, may participate in more than one RPL instance, thus enabling to differentiate traffic forwarding in the same network. The traffic can be differentiated because each IPv6 packet transport also some RPL informations like the RPL instance to use for forwarding itself.

The metric inside a DODAG is named rank, each node has a rank proportionally to the distance from the root node.The rank must monotonically decrease on each path identified by the DODAG towards its root.

Upwards Routes

DODAGs in a RPL instance are built according to a distributed algorithm. DODAG roots start advertising their presence by periodically sending DODAG Information Object (DIO) control packets to the IPv6 link-local multicast address. All RPL nodes listen for DIOs and forward them to the link-local multicast address to advertise their presence and contribute to disseminating this information throughout the network. When a RPL node receive a DIO message it can compute a set of nodes directly connected, from that set a RPL node chooses a set of neighbours and one or more parents, in this way a RPL node join a DODAG.

Downwards Routes

RPL is optimized for the upwards routes, anyway it supports also downwards routes with a particolar message called Destination Advertisement Object (DAO). After a RPL node join a DODAG it sends a DIO message including target IPv6 addresses or prefixes which are owned by that node. The DAO message is forwarded to the root in two different way: storing and non-storing. In the storing case the DAO message is sent unicast to the parents, each parent stores the received information in a downward routing table, and forwards the DAO message to its own parents towards the root. Traffic is then routed downstream based on the next-hop information stored in the routing table at each hop.
In the non-storing way the DAO message is addressed unicast to the DODAG root, and propagated by intermediate ancestors along the default route, as any other data packet. No information included in the DAO message is collected and stored

in a routing table at each node, but rather information on the path traversed to the root is stored in the DAO message. The root is therefore able to send data packets downstream to any advertised destination by means of IPv6 source routing and it is the only capable of that.

For more informations see [14].

2.1.4 Constrained Application Protocol (CoAP)

The main objective of the CoAP protocol is to develop a web protocol for devices with constrained resources using a subset of the HTTP functions and reducing also the overhead. CoAP is optimized for the Machine-to-Machine (M2M) communications with a lot of dedicated functions like: Built-in discovery, Multicast support, Asynchronous message exchange.

The paradigm is similar to the HTTP request/response paradigm. Clients sends requests to a server for a resource specifying a method code. Each resource, on each server, is identified by a URI, as in the HTTP protocol. Servers, on the other hand, send responses with a response code and optionally a resource representation. The exchange of requests/responses is totally asynchronous and the transport layer is UDP so there is not a connection phase.

By a logical point of view CoAP consists of two different layer: Request/Response and Messages. In details CoAP has four message types:

- Confirmable (CON)
- Non-Confirmable (NON)
- Acknowledgement (ACK)
- Reset (RST)

Each message can carry a request, a response or be empty hiding the message type to the request/response layer. CON messages must be acknowledged with an ACK message while NON messages are totally best-effort. RST messages are used to signal errors or unrecognised options. CON gives to CoAP a reliable behaviour without introducing too much overhead.

CoAP use two type of identifications one for each layer, Message ID is unique for each message and is used for duplicate detections, only an ACK message has the same Message ID of the CON message that has generated. A Token is used at the request/response layer and it is used for mapping a request to a response. Requests can be carried on CON or NON messages, Responses to CON messages can be carried in piggy-back on an ACK if available quickly (Figure 2.3a) or in a

separate CON message (Figure 2.3c). Responses to NON messages must be carried by NON messages (Figure 2.3b).

Concerning the Request/Response layer CoAP implements four method inherited from HTTP:

- GET requests
- POST requests
- PUT requests
- DELETE requests

The behaviour of this methods are as in the HTTP protocol and are explained in details in section 2.1.4. For more informations about CoAP protocol see [6].

Messages Layer

Message format is made by a fixed 4 byte header eventually followed by options, in the Type-Length-Value (TLV) format, and payload. A single CoAP message must be contained in a single IP datagram to avoid IP fragmentation.

CON CON messages are a simple reliability mechanism (stop-and-wait) with an exponential back-off, an end point sends a CON message until an ACK is received or a RST is received or reaches max retransmissions. A CON message always carries a request or a response and must not be empty.

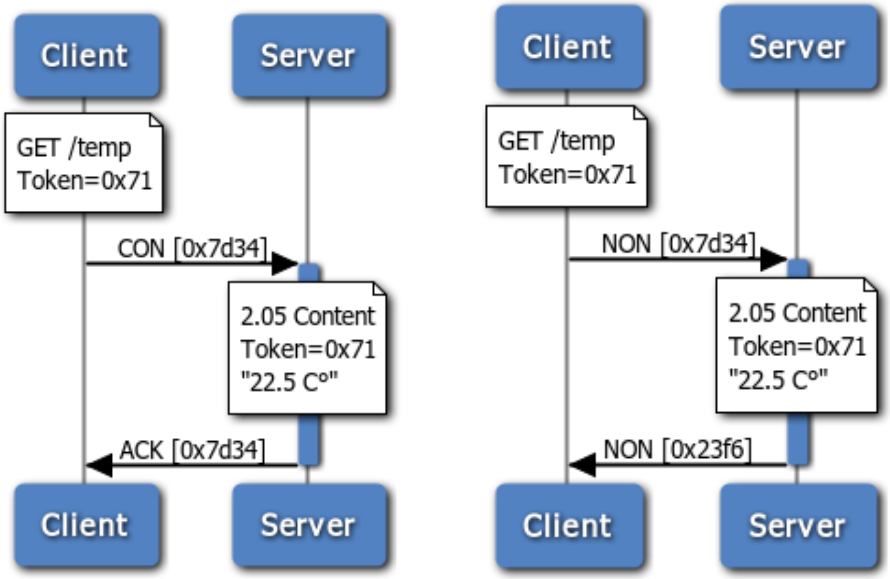
NON The receiver must not acknowledge a NON message. Like CON a NON message always carries a request or a response and must not be empty.

ACK An ACK message must have the same Message ID of the corresponding CON message. An ACK message must carry a response or be empty.

RST An RST message must have the same Message ID of the corresponding CON or NON message. An RST message must be empty.

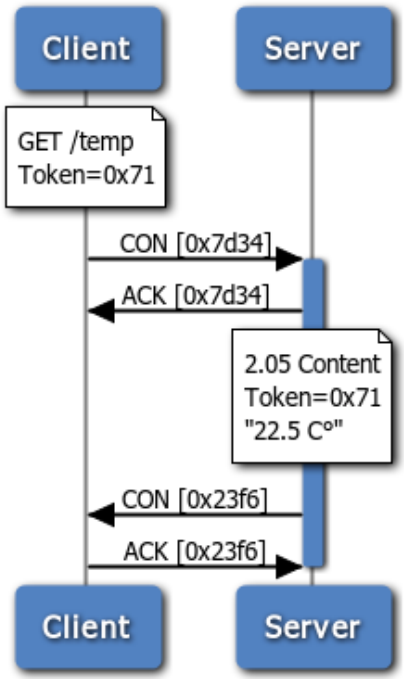
Request/Response Layer

Like in the HTTP protocol, CoAP uses concepts like client, server (or origin server), methods and, in general, the architecture is commonly knows as client-server architecture. As mentioned above CoAP inherits from HTTP the GET, POST, PUT and DELETE methods, the behaviour of each method is the same as in HTTP



(a) Response to a CON message in piggyback mode

(b) Response to a NON message



(c) Response to a CON message in separate mode

Figure 2.3: CoAP Messages

and other particular details are not needed. The matching between requests and responses are made using the token option that is mandatory for each request and has a default value, obviously the response incorporate the same token value. If a node receives a message with a token that it does not expect, the node should send

a RST message.

As in the HTTP protocol is important to notice that GET method is idempotent and also safe, PUT an DELETE methods are only idempotent and POST method is not safe nor idempotent.

Options Requests and Responses can have multiple options to specify themselves. Options are divided into two categories by them number: Critical and Elective. Critical options are identified by odd numbers and a non recognized critical option in a CON message must generate a response with code 4.02 (Bad Option). If there is an unrecognized critical option in a NON message the entire message must be silently ignored. Elective options are identified by even numbers and an unrecognized elective options must be silently ignored (only the option not the entire packet).

Caching

CoAP is designed for constrained networks, for this reason caching policies are really important. There are some options to accomplish this behaviour, Max-Age option is used, in a response message, to communicate how long the response is fresh. On the other hand a node can use the E-Tag option, when the node receives the reply it can store the response, with also the E-Tag value, and in the future can make GET requests with also the previous received E-Tag value. If the resource is not changes the node, that has to response, can reply with a 2.03 (Valid) response message, in this way it is not necessary to re-transmit the payload of the requested resource.

Proxy

CoAP uses also a proxy mechanism like HTTP. A client can contact a Proxy and asks to make request for itself, there are some options regarding this, like Proxy-Uri, for more informations see [6]. The most important thing to know is that proxying uses caching and validation with E-Tags.

Observing

The Observing mechanism permits to a client to register its interest in a particular resource. The server notifies each registered client when the resource status change (see Figure 2.4). Anyway only the server is responsible of the notifications message and the client can not influence this behaviour in the base scenario. On the other hand a server should send notifications periodically to any

registered client, because a client considers itself removed from an observation relationship if it dos not receive a notification message before Max-Age expire. For more informations about observing see [16].

Improvement to Observing The Condition option permits to a client to specify when receive notifications, i.e. Minimum period, Value over a certain value. See [5] for more informations about Condition option.

The Pledge option is used by a server to extend the Max-Age option of a previous response. See [15] for more informations about Pledge option.

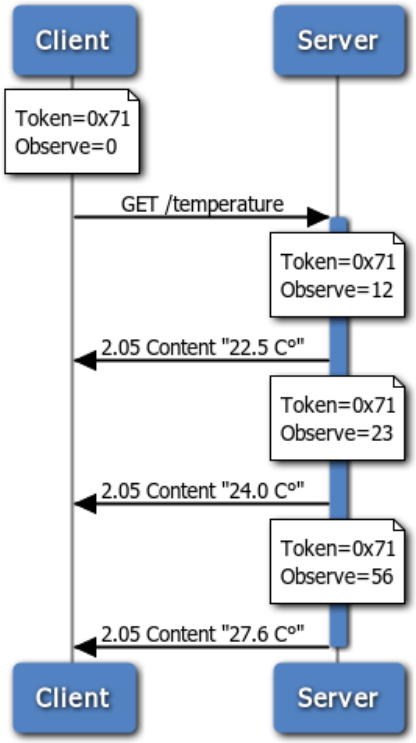


Figure 2.4: Observing a resource

Block-wise Transfers

There are two different options that can be used to split a Request or a Response message in more CoAP messages: Block1 and Block2. The Block1 option is used regarding the payload of a Request message (i.e. POST,PUT) thus the Block2 option is used regarding the payload of a Response message (i.e. GET). For more informations see [4].

URI format

A CoAP URI is something like:

$$coap :: //host[: port]/path[?query] \tag{2.1}$$

where:

- host: IP address or literal name if a DNS is in use
- port: UDP port (CoAP default is 5683)
- path: absolute path of a resource
- query: can influence a request behaviour (i.e. POST requests)

Differently from HTTP a CoAP URI is carried in four options (Uri-Host, Uri- Port, Uri-Path, Uri-Query) and also is important to notice that the path (or the query) is split in more Uri-Path options (or Uri-Query options) at the origin if it is necessary.

CoAP-HTTP and vice versa

CoAP inherits a lot of things from HTTP, so a CoAP-HTTP mapping and vice versa is quite a trivial task. If a CoAP client wants a HTTP resource, it contacts an intermediate Proxy specifying an HTTP Resource, the Proxy simply translate the request from CoAP to HTTP, then when the Proxy receives the response translates it in the CoAP domain and forwards it to the client.

If a HTTP Client wants a CoAP resource simply issue a message with a CoAP URI in the HTTP request line to the Proxy. The Proxy translates the message to CoAP and everything works as previous example. The only difference is that the HEAD method of HTTP is translate in a GET request and a CONNECT request may result in a 501 error (Not Implemented).

2.2 Overlay Networks

An overlay network is a computer network which is built on the top of another network. Nodes in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network. There are a lot of overlay networks kinds but in this section the Distributed Hash Tables (DHTs) will be investigated.

A traditional hash (key, value), table in programming languages is made of pairs where every value can be retrieved from its key at some cost depending on

the implementation. A distributed hash table is similar in principle, but the data are not stored in a single physical location in memory but, rather, are distributed as uniformly as possible among a set of nodes, called peers. The peers are all the nodes acting as the network access points for clients, and they are also assigned a key. The key is obtained by some function that maps a arbitrary length string of bit in a fixed length ID. Each IDs appertains to a key-space strictly defined and different between DHTs.

When a client wants to put a new resource in the network simply issues a publish request to its access point (anchor node). The DHT simply calculates the ID of the new resource and publish it in the overlay network. How this process is done depends on each DHT but at the end the new resource will be stored in the node (home node) with ID closest to the resource's ID previously calculated. To retrieve a resource a client has to know only the resource's ID, in fact a client can issues a get request with only the resource's ID and the overlay network will retrieve it. Again how this process is made depends on which DHT is used but, in general, any DHT can locate the home node in log(N) steps where N is the number of peers.

The main advantage of a DHT is the routing algorithm. Each peer does not have to know all others peers and can locate and retrieve resources without flooding.

2.2.1 XMHT

XMHT is a DHT overlay network based on Pastry. It use a circular key-space and unique identifiers (IDs) representing position in the circular key-space. Node IDs are chosen randomly and uniformly so peers who are adjacent in node ID can be geographically diverse. Another desirable property is that the number of hops for retrieving a resource is equal to $log_{2^B}(N)$, where B is the number of bits used to represent a digit of the key and N is the number of peers.

The XMHT's main aspect is the concept of locators. In order to use a given service, a client needs to know the ID of the service and the physical or logical location (locator) to reach the provider. Currently these two functions coexist within the IP address, which is inefficient for a variety of reasons, i.e. mobility issues. When a new resource enters the network, its ID is published into the DHT network, with also the locator of the resource outside the XMHT network i.e. ip address. This structure become a new object called enhanced locator. If the locator is changed, e.g., because the resource is mobile, an update is issued in the XMHT network, but the ID still remains the same. Therefore, when a client, at anytime from anywhere, requests the ID, the last updated e-locator will be returned, thus allowing a seamless access of the resource. Moreover an e-locator can have

some metadata fields that can contain pieces of information useful at either the application or the layers below for a more efficient reachability or exploitation of the service provided. Examples of such fields can be the geographical location of the object, the technology used for communication, or security data.

The basic function of the XMHT is to return e-locator to any user requesting the resource via its ID, by coping with possible mobility of both the physical objects and the users. Once the e-locator is obtained, an user can contacts directly the resource's owner outside XMHT network.

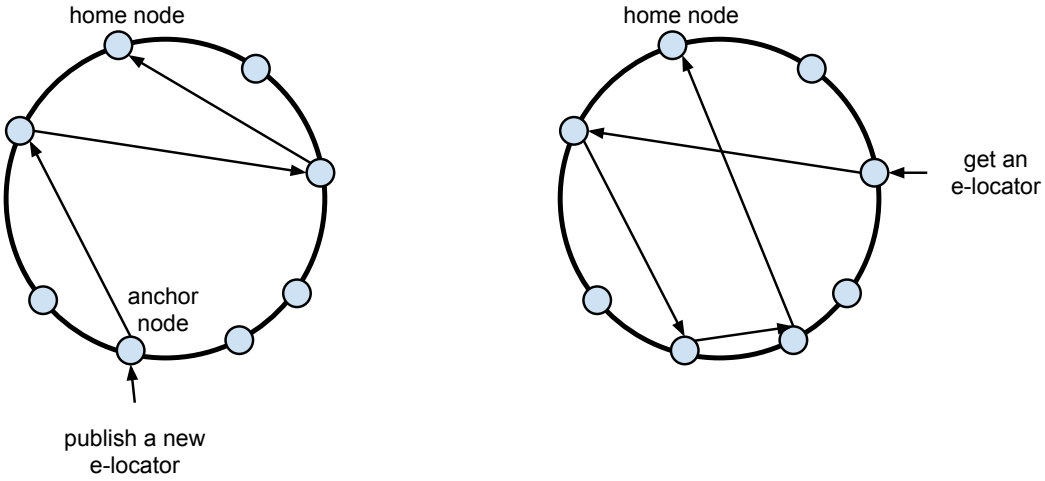


Figure 2.5: XMHT overlay network

2.3 Contiki

Contiki is an IPv6 ready, open source WSN lightweight operative system, design to be highly portable and memory efficient. Contiki is written in C programming language and has an event-driver kernel, but is also capable of handling per-process multi-threading and interprocess communication, achieved by combining the benefits of both event-driven systems and preemptive threads where preemptive multi-threading is implemented as an application library, that can only be linked with programs that require it.

Contiki provides a full IP network stack, with standard IP protocols such as UDP, TCP, and HTTP, in addition to the new low-power standards like 6LoWPAN, RPL, and CoAP. For devices that has an external flash memory chip, Contiki provides a lightweight flash file system, called Coffee. With Coffee, application programs can open, close, read from, write to, and append to files on the external

flash, without having to worry about flash sectors needing to be erased before writing or flash wear-leveling. For more informations about Contiki see [7].

2.3.1 Protothreads

A running Contiki system consists of the kernel, libraries, the program loader, and a set of processes. Contiki implements Protothread (for more informations see [8]), providing a sequential code structure and linear execution, a middle point between full multi-threading and pure event-driven systems by avoiding both the overhead of allocating multiple stacks required by multi-threading and making the implementation of blocking conditions more easily to the programmer by allowing to do it using while() loops and if() conditionals, without the explicit use of goto() calls and splitting up the code.

2.3.2 Processes

A process is still a protothread and may be either an application program or a service. A process is defined by an event handler function and an optional poll handler function. The process state is kept in its private memory and the kernel only keeps a pointer to the process state. All processes share the same address space and do not run in different protection domains. Interprocess communication is done by posting events. Events are not preemptive but can be interrupted by interrupts.

2.3.3 Services

A service is a process that implements functionality used by more than one application process becoming a sort of shared library. Typical examples of services includes communication protocol stacks, sensor device drivers, and higher level functionality such as sensor data handling algorithms. Programs call services through a service interface stub and need not be aware of the fact that a particular function is implemented as a service.

2.3.4 Events

The kernel supports two kind of events: asynchronous and synchronous events. Asynchronous events are a form of deferred procedure call: asynchronous events are enqueued by the kernel and are dispatched to the target process some time later. Synchronous events are similar to asynchronous but immediately causes the

target process to be scheduled. Control returns to the posting process only after the target has finished processing the event.
In addition to the events, the kernel provides a polling mechanism. Polling can be seen as high priority events that are scheduled in-between each asynchronous event, and then called in order of their priority.

2.3.5 Timers

Contiki provides three different kind of timer:

- Simple timer: The timer library provides functions for setting, resetting and restarting timers, and for checking if a timer has expired. An application must "manually" check if its timers have expired, meaning that this library does not post an event when the timer expires, so we must implement a routine that checks the timer for expiration.
- Callback timer: The callback timer library provides the same functions as above, but when the timer expires can callback a C function.
- Event timer: The same as above, with the difference that instead of calling a function, when the timer expires it post an event signalling the timer expiration.

Listing 2.1: An Example UDP server in Contiki

```
#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
#include "net/uip.h"
#include "net/netstack.h"
#include <string.h>
#include <stdio.h>

PROCESS(udp_server_process, "UDP server process");
AUTOSTART_PROCESSES(&udp_server_process);

static struct uip_udp_conn *server_conn;

static void tcpip_handler(void)
{
        char *appdata;
        if(uip_newdata()) {
```

```
                appdata = (char *)uip_appdata;
                appdata[uip_datalen()] = 0;
                printf("Server received: '%s' \n", appdata);
        }
}

PROCESS_THREAD(udp_server_process, ev, data)
{
        static struct etimer timer;

        PROCESS_BEGIN();

        // wait 3 second, in order to have the IP addresses
           well configured
        etimer_set(&timer, CLOCK_CONF_SECOND*3);

        // wait until the timer has expired
        PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);

        // set NULL and 0 as IP address and port to accept
           packet from any node and any srcport.
        server_conn = udp_new(NULL, UIP_HTONS(0), NULL);
        udp_bind(server_conn, UIP_HTONS(3000));

        printf("Server listening on UDP port %u\n",
           UIP_HTONS(server_conn->lport));

        while(1) {
                PROCESS_YIELD();
                if(ev == tcpip_event) {
                        tcpip_handler();
                }
        }
        PROCESS_END();
 }
```

Chapter 3

A gateway-based scalable architecture for the IoT

3.1 Overview

3.1.1 Goal

The main objective of this work is to aggregate IoT technologies in an enhanced way, realizing a distributed system where people can ask for resources or manipulate resources already present in the network, without knowing their location. In details the system is, as the name says, a gateway based system where people only know the location of their local gateway, the gateway masks everything from the user and works like a central server with all the resources connected. The architecture consists of many layers stacked together from the 802.15.4 physical layer to the application specific layer.

3.1.2 Proposed solution

The proposed solution aims to create a distributed system where resources can be created or deleted at running time without any kind of synchronization restrictions. An user sees the overall system as a centralized server with all the resources directly connected even if the resources are localized in different places. As mentioned in Chapter 1, the solution is based on standard protocols (or protocols that will become standard soon) and has a particular focus on interoperability issues.

The solution needs, as a prior building block, an overlay network above the normal Internet network. The choice is the XMHT overlay network (see Section

2.2) because it is a pastry-like[1] implementation that gives an abstraction layer for connecting devices, potentially dislocated everywhere in the world, in a scalable way. The overlay network is also necessary because resources, and in general servers, are created dynamically. Thus the solution needs a way to address new resources at run time, with this idea in mind a p2p[2] network such as XMHT is needed. On the other hand the overlay alone is not enough to accomplish the work goal. In fact overlay is used only to localize resources that are physically dislocated in different places. In details XMHT is used to store, for each resource, only the locators to gateways and servers that actually host the resource. A locator is, in fact, a metadata field with gateway IP address and server IP address. Once a locator is obtained, the associated resource can be retrieved in different ways that are explained below.

The choice of storing locators in the overlay network is motivated by the fact that servers can be in a sleepy mode for a period of time (as mentioned in chapter 1). The storing of a locator allows the upper layer to decide which policy to adopt. The easiest one is to contact the server that has the requested resource and to retrieve it. This approach however does not solve the sleepy server problem. A better approach is to make a connection to the gateway responsible for the requested resource. The gateway can request the resource to the server on behalf of the Client. If the server is in a sleepy mode, the gateway can reply with its cache or anyway the gateway can wait for the server to wake up.

Another reason for storing locators regards servers that can be on constrained networks. Servers can run on constrained networks attached to the Internet by a gateway. The gateway may manage a real big group of servers that probably sense the world in a particular area. A connection to a gateway, instead of one to each server, enables a user to obtain a sort of aggregated information about the overall sensor network attached to the gateway. This behaviour becomes more interesting if the client is also a constrained device with poor computation power. The gateway, in fact, can have a lot of computation power and can aggregate information better than a constrained device.

Over the overlay network it is necessary to have an application-layer that gives, to an user, a Service Oriented Architecture (SOA)[3]. Clients, on the other hand, have to know how to interact with the system and, to use only standard protocols,

[1]Pastry is an overlay and routing network for the implementation of a distributed hash table (DHT).

[2]A peer-to-peer (P2P) computer network is one in which each computer in the network can act as a client or server for the other computers in the network, allowing shared access to various resources such as files, peripherals, and sensors without the need for a central server.

[3]a service-oriented architecture (SOA) is a set of principles and methodologies for designing and developing software in the form of interoperable services.

CoAP seems to be the best choice (see Section 2.1.4 for more informations about CoAP).

Thus clients use CoAP for connecting to their gateways, but gateways have to make connections to each others. Gateways are already connected by the overlay network but, at application-layer, a different kind of connection is still needed. This is true because the nature of a p2p network does not permit to know all the p2p participants and also does not permit to make a connection to a specified gateway for retrieving a certain resource. To accomplish this behaviour there are a lot of possible choices, each one of them with pros and cons.

One choice could be to use a RPC-like system[4]. RPC systems are well documented and supported by a lot of projects. In this scenario a gateway has to invoke a method on another remote gateway and to wait for completion. It could be an easy interacting scheme but interoperability could become an issue. In fact many RPC-like systems are designed to operate only in a certain programming language, i.e. JAVA RMI[5] works very well but only with other JAVA programs. An RPC-like choice will probably force the programming language of the application-layer in too strong a way.

Another possible choice is to use Web Services[6]. Web Services support interoperability between different programming languages and can accomplish the desired behaviour without big issues. A gateway has to request a service to another remote gateway and to wait for completion. The possible issue in this scenario is the fact that it will need a Web Services enabled server running on each gateway. Gateways are not server machines and may be like a router with limited resources.

The last choice is a REST-full approach. A REST-full approach permits interoperability by its own nature and it is not so computational heavy as Web Services. Anyway in the REST-full family there are a lot of possible choices, i.e. HTTP, CoAP. HTTP could be an easy interaction model because there are a lot of lightweight HTTP server already developed. On the other hand CoAP is a new protocol specifically designed for constrained devices.

The final choice is to use the CoAP protocol because it permits to adopt different policies. For example the use of CoAP between gateways permits to have a single CoAP server, running on a gateway, that interfaces itself to servers on sensor nodes

[4]A remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another computer on a shared network without the programmer explicitly coding the details for this remote interaction.

[5]Java RMI is a Java API that performs the object-oriented equivalent of remote procedure calls (RPC).

[6]A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.

on the one hand, and to other gateways on the other hand. This permits a resource saving on gateways. Another reason concerns locators as mentioned above. There could be servers that are always awake and with resources that are time critical. The use of CoAP also between gateways permits to these special servers to be interrogated without passing through their gateways for a quick reply, if necessary.

A server could be a constrained device with limited computational resources. From this point of view observing features may become an issue (for more information about observing see Section 2.1.4). In fact embedded devices can not store too many observers because each observing relationship needs a lot of dedicated memory resource. The proposed solution aims to solve the problem by a delegating approach. Instead of storing the observing relationship on the real server, an observe request is memorized only at gateway level. Clients request an observe relationship on a resource to their gateway. A gateway stores the request and periodically interrogates the real server and gets requested resources. In this way the real server, that is on a constrained device, does not know anything about observing. From its point of view, every request is just a normal request. On the other hand gateways have more resources and can store more observing relationships without memory issues.

This solution permits, to an user, to interrogate constrained devices from everywhere without knowing their physical locations. The only thing necessary is that the client must be connected to a gateway that is already a node of the overlay network. Moreover, this solution permits also the use of observing even if the server is hosted on constrained device with few computational resources. Finally, the use of standard protocol permits the use of different implementations from different manufacturers without any problems.

On the other hand, by p2p nature the discovery of resources may be a problem. In Section 6 there are some possible approaches to solve this issue.

3.1.3 Terminology

In the following sections some terms will be used:

- Locators: meaning a struct of metafields used to find a resource.
- gateway/access-point: meaning a point where users, or servers, connect to the entire architecture.
- origin gateway/origin access-point: meaning a gateway where a client connects to the entire architecture.

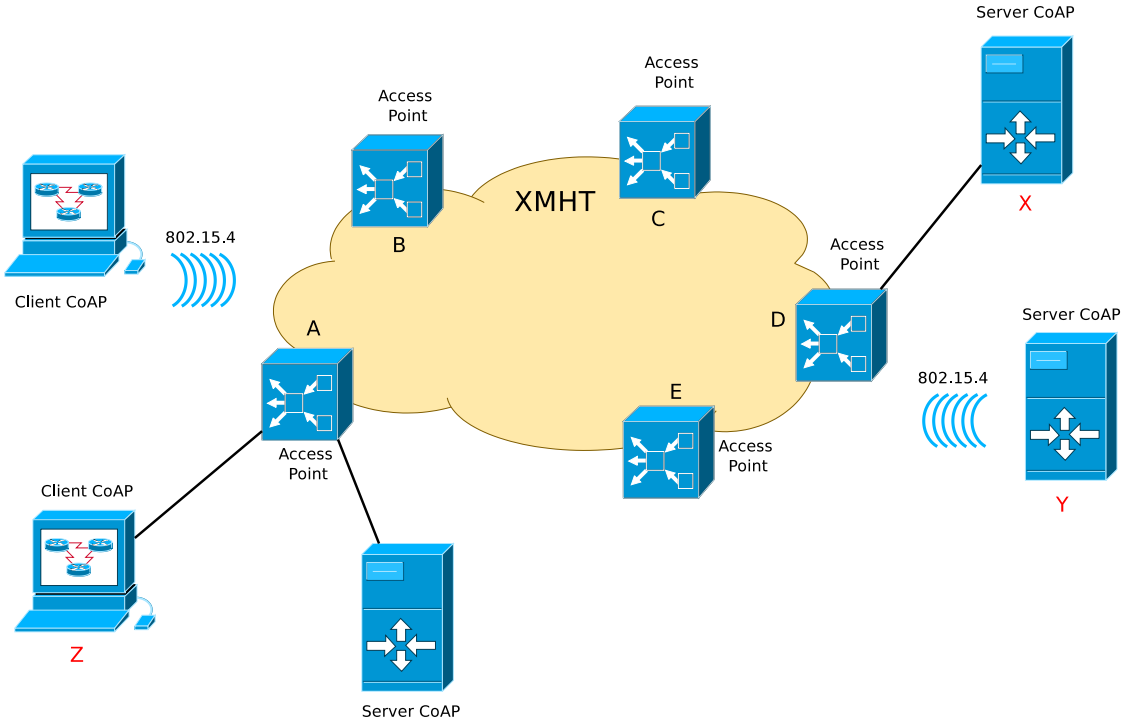


Figure 3.1: Architecture

- target gateway/target access-point: meaning a gateway where a server connects to the entire architecture.
- application-layer: meaning the upper software layer that gives to users the protocol abstraction.

3.1.4 Architecture

The Architecture consist in the XMHT network, used as an overlay network, plus the application-layer that gives to users services, thus implementing a Service Oriented Architecture (SOA). Each access-point is also a proxy between the XMHT network and the external network as shown in Figure 3.1. Clients connect to gateways with the CoAP protocol, and by their point of view, everything is just a unique CoAP server. For latency issues the application-layer interprets a CoAP server that always responds in separate mode (see Figure 3.2).

In detail is important to consider all layers and their functions. The XMHT network hosts only Locators to gateways which manage a group of servers as shown in Figure 3.1. The XMHT network is responsible to retrieve locators for any resource, once the Locators were obtained the application-layer gets and delivers the resource to the user who requested it. The connection between the origin access-point and the target access-point is established outside the XMHT network

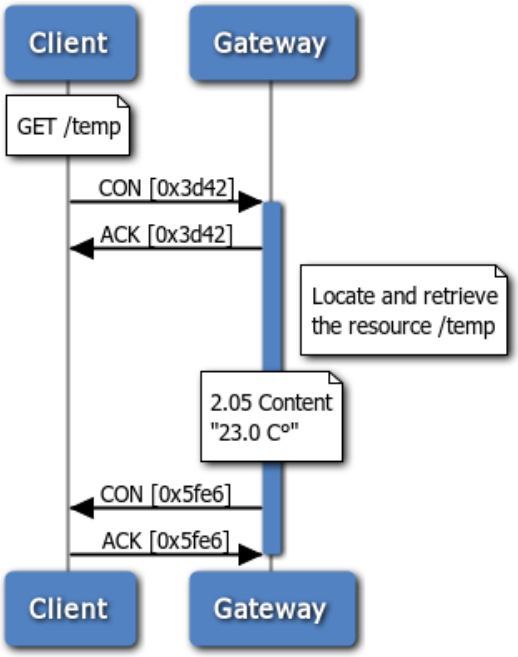


Figure 3.2: An interaction by the client point of view

and can potentially use any protocol or architecture RPC-like. In this work CoAP is used as protocol also between gateways. Every Gateway must have a also default CoAP server to use for publishing actions as described in 3.2.3 and in 3.2.5.

3.1.5 Assumptions

To reproduce this work same assumptions are needed. First of all, XMHT needs an IPv4 network between gateways. This constraint can be relaxed by a simply improvement to the XMHT architecture allowing so also IPv6 networks.

CoAP servers have to be used only with application-layer because there is not any kind of discovery process for server's resources. The application-layer, in fact, populates the XMHT network, with locators, before create resource on the CoAP servers. In Section 6 there are some possible approaches to relax this constraint.

A user need a CoAP client for interacting with the overall system because no HTTP-CoAP functionality are provided. This has been done because in the next future a lot of devices will have a CoAP client and, also, the proxy functionality adds nothing to this work.

Finally, this work has been tested only on Linux systems (especially Ubuntu and Voyage Linux). Because XMHT's implementation some packages are needed: Openssl, Protobuffer, Cryptopp, Tinyxml.

3.2 Detailed procedures

3.2.1 Configuration

For configure an access-point a config.xml file is needed. The structure is described in Listing 3.1. In Section 6 is described some possibles improvements.

Listing 3.1: config.xml

```
<info>
        <item>
                <ap default="true">
                        <ip>192.168.1.1</ip>
                        <port>5683</port>
                </ap>
                <server default="true">
                        <ip>192.168.1.1</ip>
                        <port>5684</port>
                </server>
        </item>
        <item>
                <ap>
                        <ip>192.168.1.1</ip>
                        <port>5683</port>
                </ap>
                <server>
                        <ip>aaaa::c30c:0:0:9e</ip>
                        <port>5683</port>
                </server>
        </item>
</info>
```

3.2.2 Retrieve a resource

A resource name could be unique or not. For unique resource name everything is almost easy because there is not a misunderstanding behaviour in what client expects. If a resource name is not unique the application-layer may use different behaviours.

A retrieve interaction is formed by:

- The Client asks a resource to its Gateway (GET).

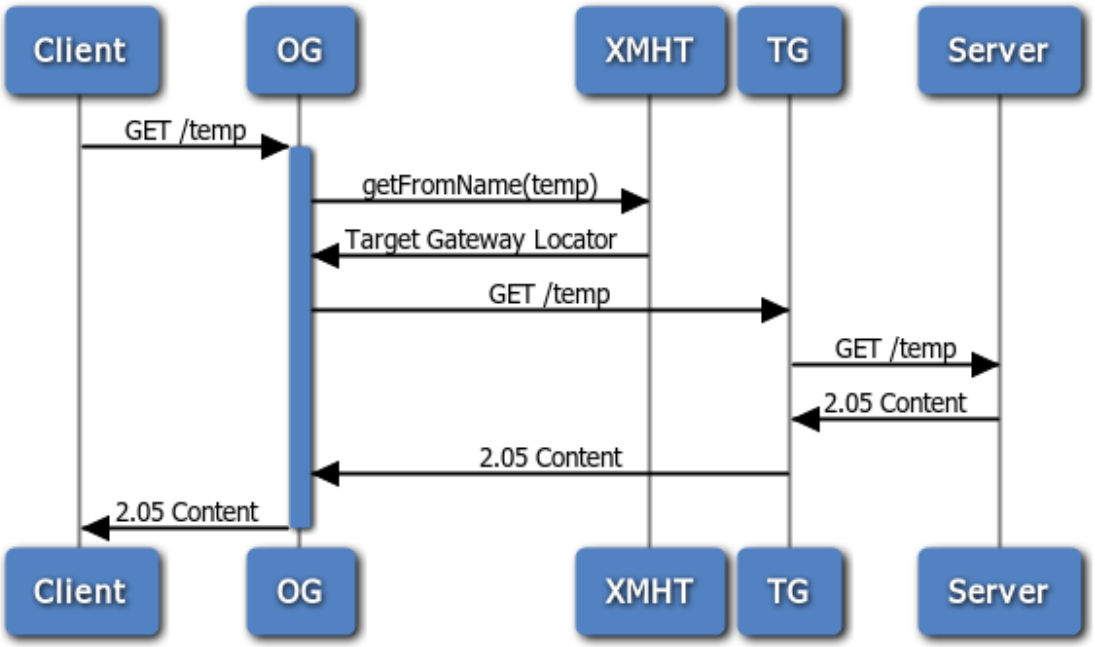


Figure 3.3: A retrieve interaction, OG=Origin Gateway, TG=Target Gateway

- The Origin Gateway (in details the application-layer running on the gateway) receives the request and asks, to the XMHT network (using the XMHT API), the Locator.
- The Origin Gateway parses the Locators retrieved and creates a set of Target Gateway.
- The Origin Gateway connects to each Target Gateway asking for the resource.
- The Target Gateway that has received the request connects to the CoAP server that really hosts the resource. The server is, in fact, directly connected to the Target Gateway.
- The Target Gateway retrieves the resource from the server and forwards it to the Origin Gateway.
- The Origin Gateway receives the resource from all Target Gateways in the set that was previous created. Than forwards to the Client the resource.
- The Client receives the resource who has asked.

If the application-layer obtains multiple locators it has to decide which policy adopt. A client can specify which policy use with a query string. The default behaviour is to send all resources to the client, in a unique message with resources payload appended.

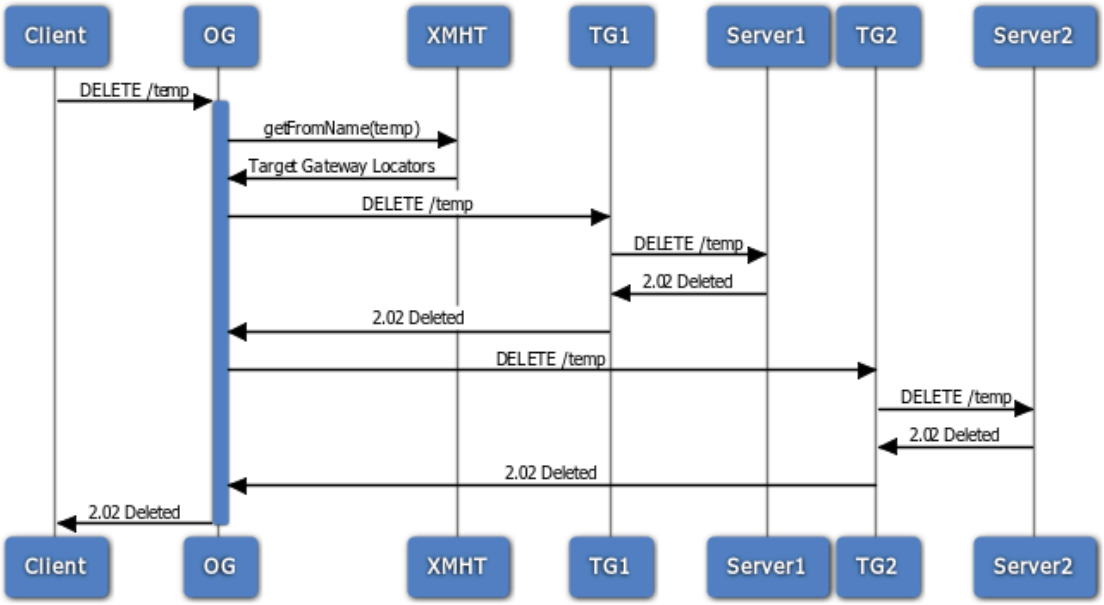


Figure 3.4: Delete operation on non unique resource name

3.2.3 Publish a new resource

A client issues a PUT/POST request to its access-point to publish a resource with the resource representation in the body of the message and optionally a query string saying which server outside the XMHT network has to actually host the new resource. If the query string is not present than the application-layer uses its default server. A correct query will be:

$$?serverip =< ... > \&serverport =< ... > \&apip =< ... > \&apport =< ... > \tag{3.1}$$

It means "Update or Create the resource on the server...".

3.2.4 Unpublish a resource

To unpublish a resource a client makes a DELETE request to its access-point. The application-layer contacts the Target gateway and asks to delete the resource on the real server that hosts it. Then if everything has gone well, the application-layer on the target gateway unpublishes the resource from the XMHT network. At the end the application-layer, on the target gateway, has to inform the origin application-layer the completion of the operation so it can acknowledge the client if it is necessary. If the resource name is not unique than the policy can be different, by default if no query was provided all resource with that name will be deleted. A possible query is as in query 3.1, it is meaning "delete the resource on the server...".

3.2.5 Update a resource

To Update a resource a client makes a PUT/POST request to its access-point. The application-layer retrieves the resource locator, contacts the target gateway and asks to update the resource. The target access-point updates the resource on the real server and inform the origin gateway the completion of the operation so it can acknowledge the client if it is necessary. If the resource name is not unique than the policy can be different, by default if no query was provided the resource will be modified or eventually created on the origin access-point's default server. A possible query is as in query 3.1, it is meaning "Update or Create the resource on the server...".

3.2.6 Observing a resource

The origin access-point memorizes the observer relationship with also the locator for the client who has requested it. The origin gateway is responsible of everything concerning observing relationships, when a notice has to be sent the origin access-point issue a GET request to the target gateway and when the response arrive, it sends the notification to subscribers. On the other hand, the target gateway does not know anything about the observing relationship, in fact by its point of view, each message is simply a GET request as in Section 3.2.2. The main issue is the notification period, in fact by CoAP draft a client considers itself excluded by an observe relationship if it does not receive a notification for a period longer than Max-Age (default:60 sec). The problem is that the origin gateway must consider the XMHT overlay, the target gateway and the real server, so the latency can be an issue. The problem also grows up if a resource is not unique because more target gateway has to be contacted.

3.3 Implementation Details

The start point of this work is the Intecs's implementation of the CoAP protocol. The architecture has already the main core and implements a lot of functions excepts Observing. The observing function is really necessary for this work and has been implemented during the thesis period. A Database abstraction was also important for mapping a resource in the distributed world created by XMHT and in general by application-layer.

3.3.1 Observing features

As mentioned in Section 2.1.4, the observing option permits to a client to register its interest in a certain resource. It has been necessary to develop a list of subscribers and a method for notify them. Because Intecs implementation is in C++, threads seems to be the perfect choose. When a client issues a GET request on a resource the server checks options, if an Observe option is found than the new mechanism start. Server save the client ip and the CoAP message type (CON or NON) in a list of subscriber for a certain resource. In parallel way a new thread periodically check subscribers lists and notify each client. The notification message is carried by the same message type of the observe request received previously.

This behaviour may generate some undesirable state, suppose a client made an observe request and then crash. If the request has been made by a CON message there are not particular issues, because server can detect the failure by the ACK mechanism. On the other hand if the client has made the request with a NON message the server can not detect failure. To solve this problem (that at point of write is not considered by the CoAP draft) a stateless mechanism has been developed. The thread responsible for notifications drops NON subscriptions in a probabilistic way, this mechanism is enough secure for the server and on the other hand a client can detect an un-subscription because it do not receive a notification message for a period grater than Max-Age. The Client, anyway, can safely register itself again as an observer without too much overhead.

3.3.2 Externalize Database

A new database interface is needed, in details the new implementation permits to use any kind of external database. For example a MySQL Database can be used, the only thing important is to implement an interface with public methods conforms to the one used in this work. The externalization of the database is a critical point in this work just because XMHT and external real server are mapped into a new database type called XMHTDatabase. Once a "get resource" is issued to the database class the interface implementation start the process described in details in the Section 3.2.2. It is the same for "put resource" or "delete resource".

Listing 3.2: XMHTDatabase

```
        ...
        std::set<XMHT::Metadata> result;
3       if (communicationApi.getFromName(xmhtName, result)
           == false) {
```

```
                INFO("Error when retrieving the resource");
                return Resource();
        }
        std::set<XMHT::Metadata>::iterator it = result.begin
            ();
        int counter = 0;
        std::string payload="";
        for(; it != result.end(); ++it){
                XMHT::Metadata meta=*it;
                std::string xmhtdata=meta.getMetadata();
                /*
                        Parse XMHT Locator to get the access
                           -point and server responsable for
                            the requested resource
                */
                XMHTLocator dst=parselocators(xmhtdata);

                std::string desturi;
                Message request;
                request.setType (Message::CON);
                request.setCode (Message::REQUEST_GET);
                std::string ip=dst.server_ip;
                std::string port=dst.server_port;
                /*
                        Verify if server is directly
                           connected to this gateway
                */
                if(isDirectlyConnected(dst))
                {
                        ip=dst.server_ip;
                        port=dst.server_port;
                        desturi = "coap://" + dst.server_ip
                           + ":" + dst.server_port+uri[0];
                }
                else
                {
                        ip=dst.access_point_ip;
                        port=dst.access_point_port;
                        desturi = "coap://" + dst.
                           access_point_ip + ":" + dst.
```

```
                access_point_port+uri[0];
38              request.addUriQuery("serverip="+dst.
                    server_ip);
                request.addUriQuery("serverport="+
                    dst.server_port);
                request.addUriQuery("apip="+dst.
                    access_point_ip);
                request.addUriQuery("apport="+dst.
                    access_point_port);
            }
43          /*
                Connect to the server or to the
                    gateway
            */
            Client client (ip,port);
            counter++;
48          request.parseURI (desturi);

            request.addEtag(etag);

            std::string token;
53          int ret = client.send (request);
            if(ret < 0) {
                INFO("ERROR: Cannot send request
                    message!");
                client.stop();
                client.wait();
58              return Resource();
            }
            Message response;
            ret = client.receive (token, response);
            if (ret < 0) {
63              INFO("ERROR: Cannot receive response
                     message!");
                client.stop();
                client.wait();
                return Resource();
            }
68
            std::string tmp;
```

```
                uint32_t size;
                size+=response.getPayload(tmp);
                payload+=tmp;
73
                client.stop();
                client.wait();
        }
        ...
```

Chapter 4

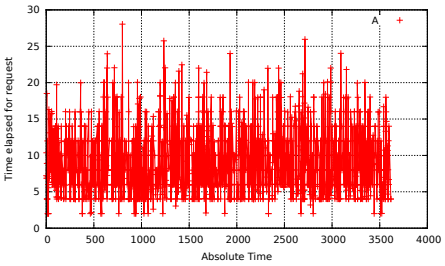
Performance Evaluation

To evaluate the overall system a small testbed has been created. The testbed is composed by five PCs with different hardware and operation systems. In this way the test can simulate a real scenario where devices from different manufacturers can be different in hardware and software. Each machine is a gateway, with all layers stacked together as mentioned in Chapter 3 and there is also a standard CoAP server used as default server. On each PC, four standard CoAP clients also run, each one connected to the application-layer on the hosting machine. Clients issue requests to their gateway (in this scenario the hosting PC) and randomly choose on which CoAP default server they want to store new resources.

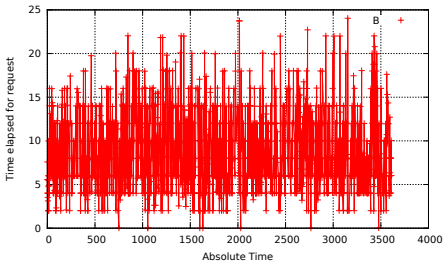
A lot of tests have been made to evaluate the overall system, by varying the Inter Request Time. The system reacts very well and delays are always limited to a small range. Anyway, it is important to consider that XMHT and also the CoAP implementation is still in a development phase. From this point of view there are certainly a lot of improvements that can be carried out for optimizing latency and reliability. Moreover CoAP protocol is, for the moment, in a design stage and its properties change really quickly with features that are added and removed almost on a weekly basis. The following graphs show requests elapsed time in varying Inter Request Time. On the other hand the differentiation by request type shows that there is not a real difference among messages. This behaviour is in compliance with the CoAP standard that tries to reduce overhead in all messages (Figure 4.5a, 4.5b, 4.5c and 4.5d).

Table 4.1: Testbed Software and Hardware details

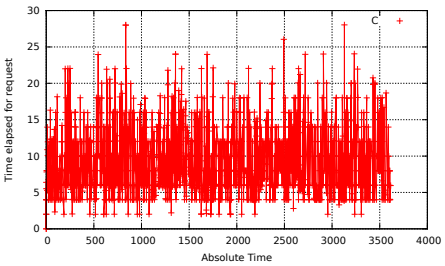
Gateway Name	Operation System	CPU
A	Ubuntu 11.04	Intel Core i5 3.20GHz
B	Ubuntu 12.04	Intel Core 2 2.13GHz
C	Ubuntu 10.04	Intel Core 2 2.40GHz
D	Ubuntu 10.04	Intel Core 2 2.40GHz
E	Ubuntu 10.10	Intel Pentium 4 3.00GHz



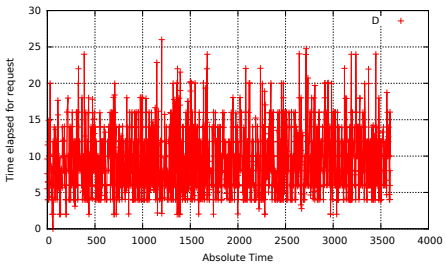
(a) Gateway A



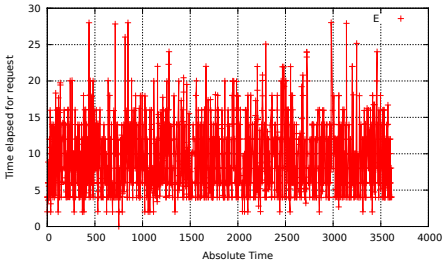
(b) Gateway B



(c) Gateway C



(d) Gateway D

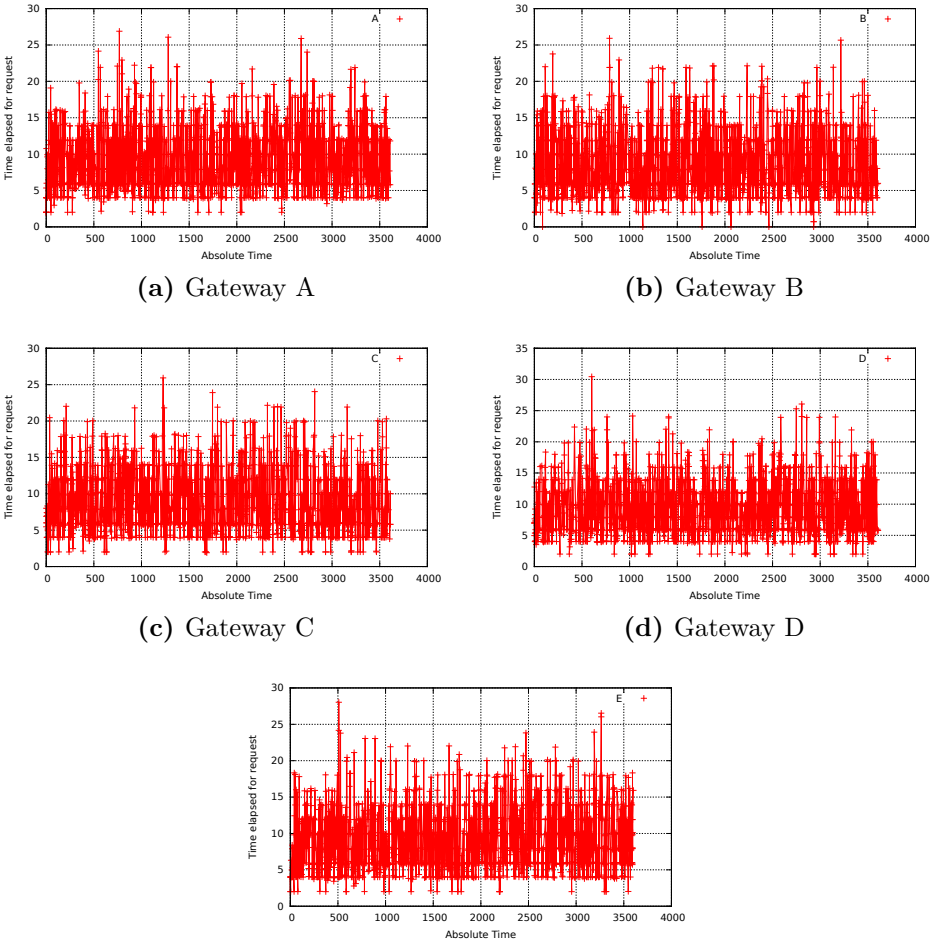


(e) Gateway E

Figure 4.1: Inter Request Time 10ms

4.1 Testbed Details

On each PC there are four clients that execute CoAP requests. The first client simply performs GET requests, the second issues POST requests, the third PUT requests and the fourth DELETE requests. Each request has been made for a resources that was randomly chosen, for this reason a lot of 4.04 NOT FOUND has been returned. For a better analysis, of the overall system, those requests has



(a) Gateway A

(b) Gateway B

(c) Gateway C

(d) Gateway D

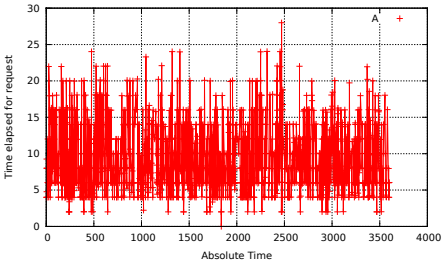
(e) Gateway E

Figure 4.2: Inter Request Time 100ms

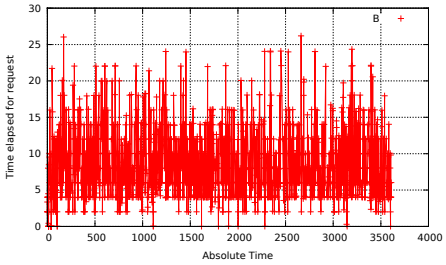
been deleted from the graphs because they are satisfied too quickly by XMHT layer without a real message exchange phase between application-layers. Moreover POST and PUT requests chose also the server for hosting the new resource in a random way. Once the choice has been done, clients uses a query approach (like in query 3.1) for selecting the CoAP server.

4.1.1 Stress test with more clients

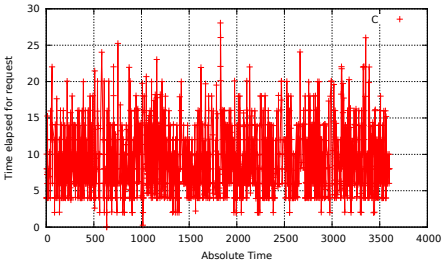
For better evaluate the entire system a short stress test has been made. Clients added simply performs POST requests for random resources and to random servers. Graphs below show an increment of the Time Elapsed per request, as expected. All graphs has been made with an Inter Request Time of 1000 ms.



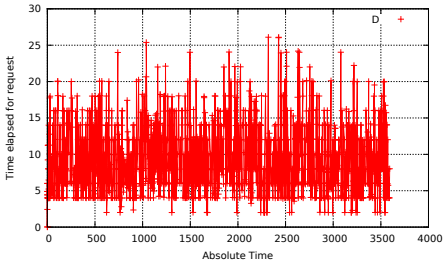
(a) Gateway A



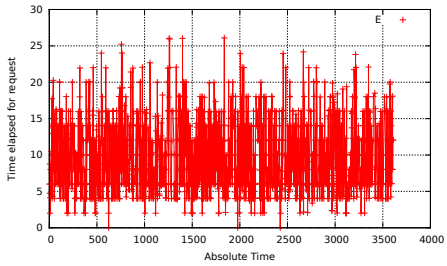
(b) Gateway B



(c) Gateway C

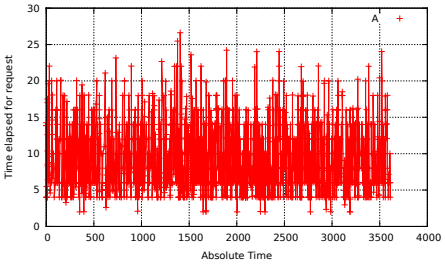


(d) Gateway D

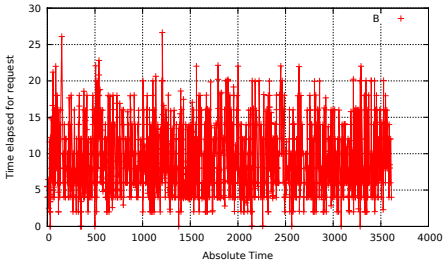


(e) Gateway E

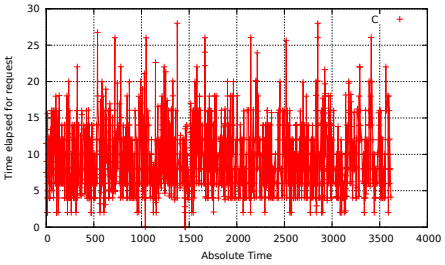
Figure 4.3: Inter Request Time 1000ms



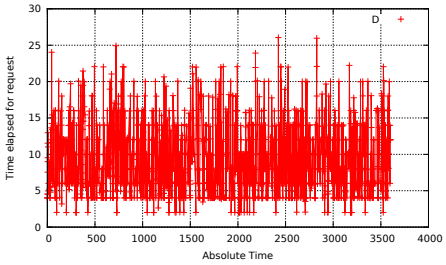
(a) Gateway A



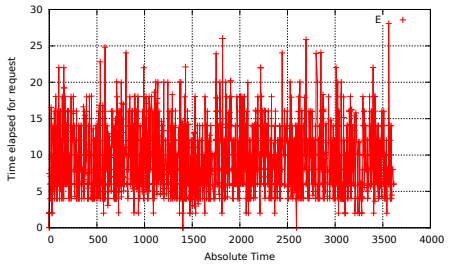
(b) Gateway B



(c) Gateway C

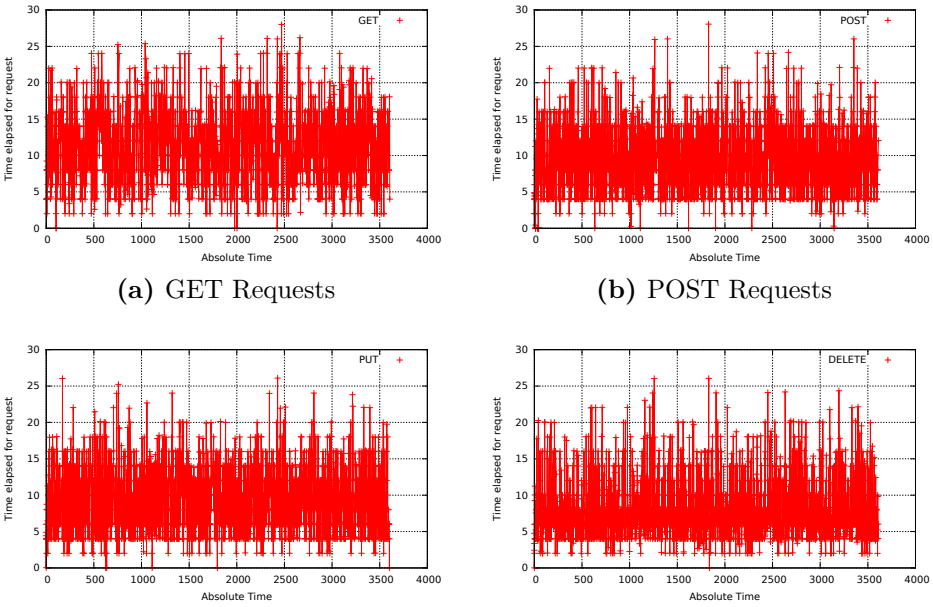


(d) Gateway D



(e) Gateway E

Figure 4.4: Inter Request Time 10000ms



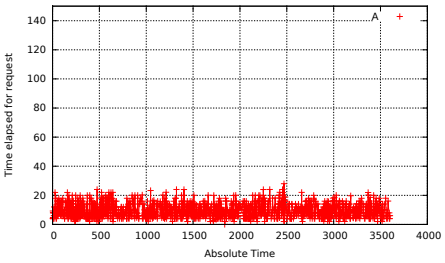
(a) GET Requests

(b) POST Requests

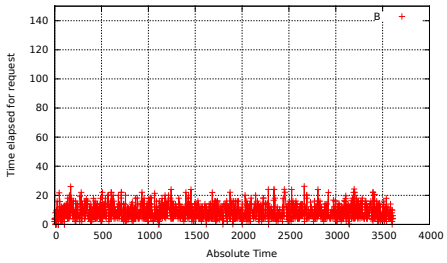
(c) PUT Requests

(d) DELETE Requests

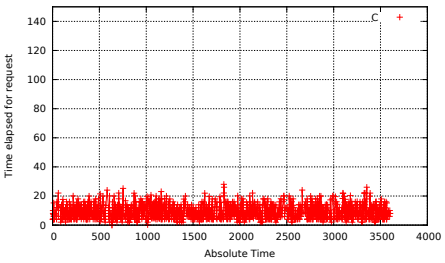
Figure 4.5: Different Requests Elapsed Time



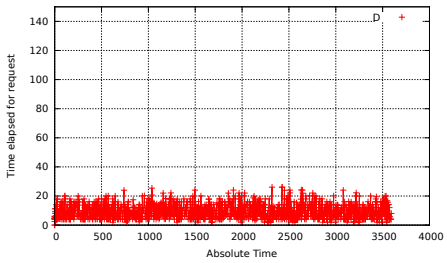
(a) Gateway A



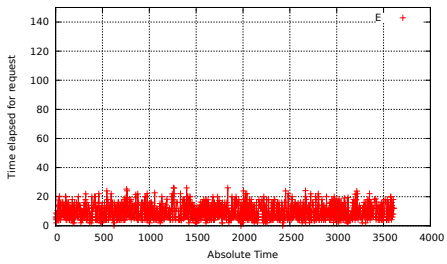
(b) Gateway B



(c) Gateway C

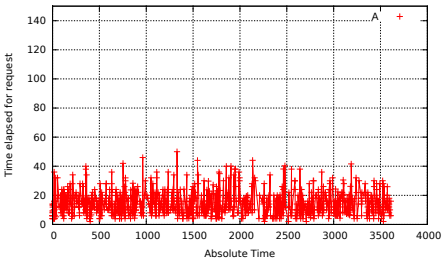


(d) Gateway D

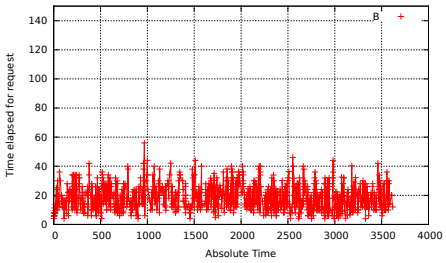


(e) Gateway E

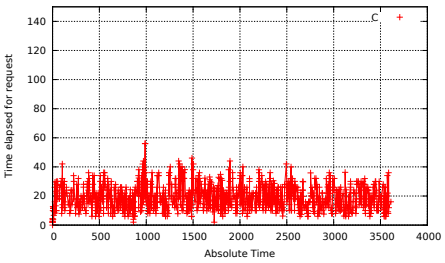
Figure 4.6: 20 Clients



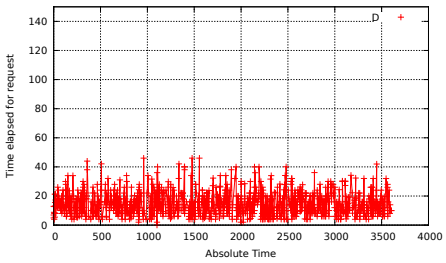
(a) Gateway A



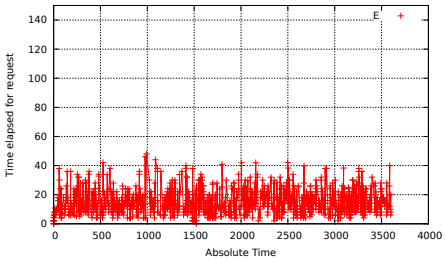
(b) Gateway B



(c) Gateway C

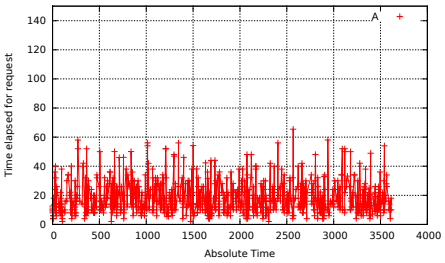


(d) Gateway D

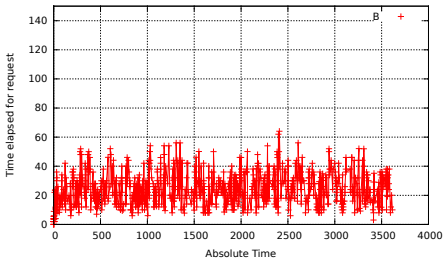


(e) Gateway E

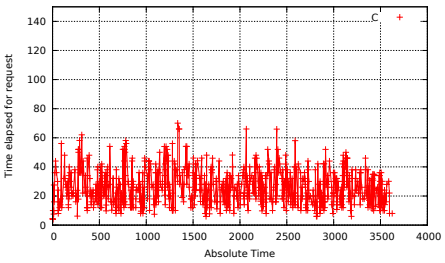
Figure 4.7: 24 Clients



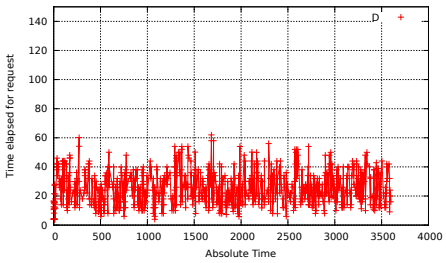
(a) Gateway A



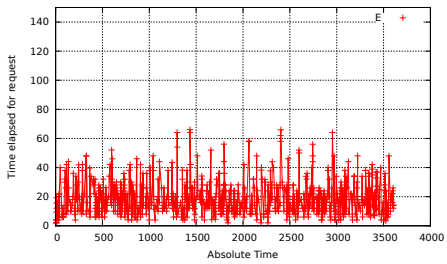
(b) Gateway B



(c) Gateway C

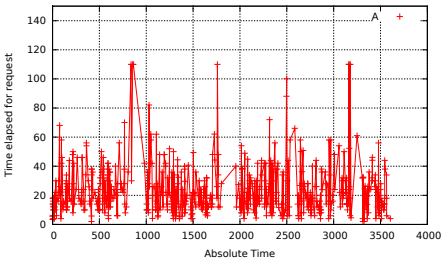


(d) Gateway D

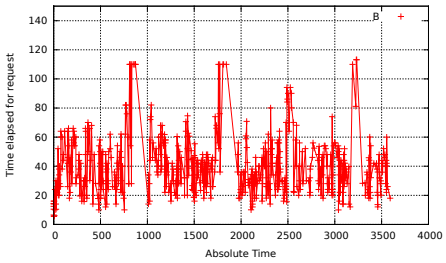


(e) Gateway E

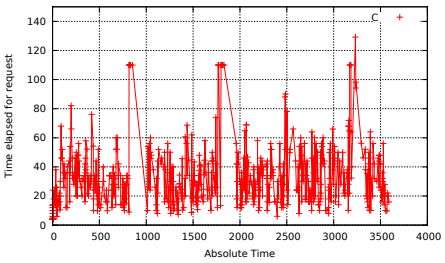
Figure 4.8: 26 Clients



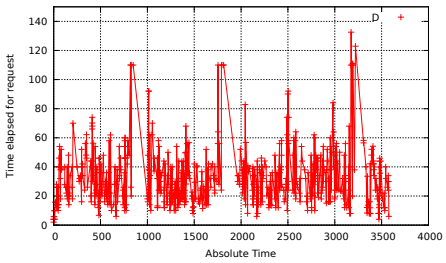
(a) Gateway A



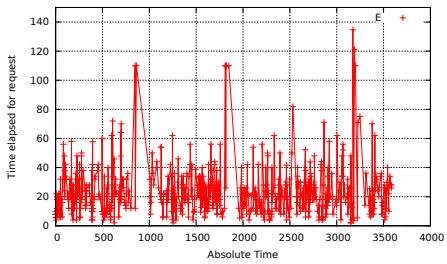
(b) Gateway B



(c) Gateway C



(d) Gateway D



(e) Gateway E

Figure 4.9: 28 Clients

Chapter 5

Prototype

5.1 Overview

For demonstration purposes a demo system has been made. It made of different components from sensors to Android device used to control all the system. It is necessary to choose hardware for building the prototype with particular care to costs and performances.

5.2 Possibles Gateway solutions

The project need a gateway between 802.15.4 world and Internet. In this section there are a lot of possible approach to the problem with advantages and disadvantages.

5.2.1 All on board

The main advantage is that all the computational load is on the board with a general purpose OS. While the main disadvantage is that there is not a 6LoWPAN opensource implementation for general purpose system. There are same project active in this field [22], [12], [3], [13] but, at the moment of writing, no one is mature. At the end there are not a lot of board with a 802.15.4 transceiver that support a general purpose OS.

Commercial Device

As mentioned above there is not a board with a general purpose system and a 802.15.4 transceiver, the only exception is the iMote2 [11].

The iMote2 is composed of:

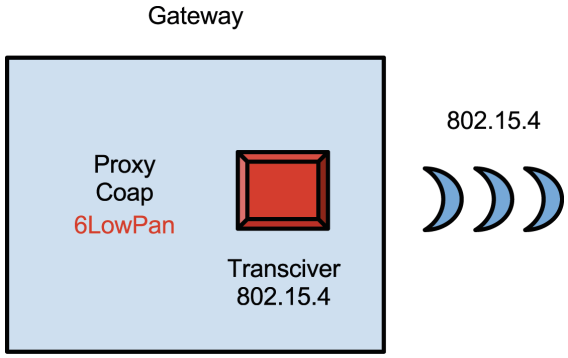


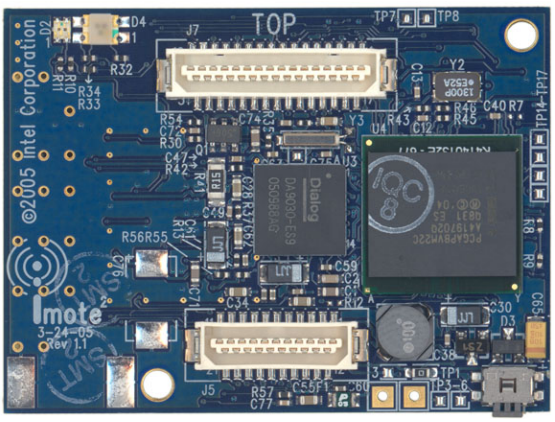
Figure 5.1: All on board

- 320/416/520MHz PXA271 XScale Processor
- 32MB Flash on-board
- 32MB SDRAM on-board
- 256 KB SRAM
- Mini-USB Client (slave), multiplexed with RS232 console over USB, power
- CC2420 (Transciver 802.15.4 - 2GHz)

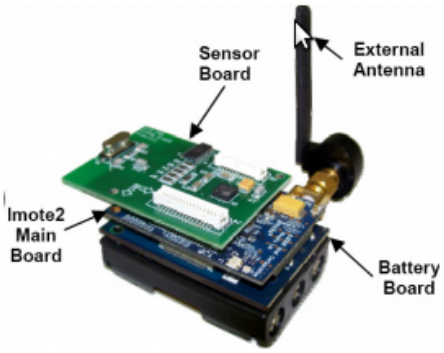
It support a dedicated Linux system that has been moved inside the openembedded project [17], anyway the documentations about this board is a bit confused because the transceiver has a lot of integrations problems inside the Linux kernel. A possible approach is to use the tos_mac driver, it is a porting of the TinyOS 6loWPAN driver. The problem in this scenario regarding the fact that the TinyOS driver is conform to the first 6LoWPAN draft, thus the interoperability with modern 6LoWPAN implementations (like the one present in Contiki) is not assured. On the other hand, another possible approach is to use the Linux-Zigbee [12] but documentations, especially in this scenario, is really confused.

5.2.2 External Transceiver

The main advantage is that all the computational load is on the board with a general purpose OS. While the main disadvantage is that there is not a 6LoWPAN opensource implementation for general purpose system. Another possible disadvantage is the bottle-neck between transceiver and board, the transceiver use a serial protocol so more studies in this behaviour are necessary.



(a) iMote2 standalone



(b) iMote2 and sensor board

Figure 5.2: iMote2

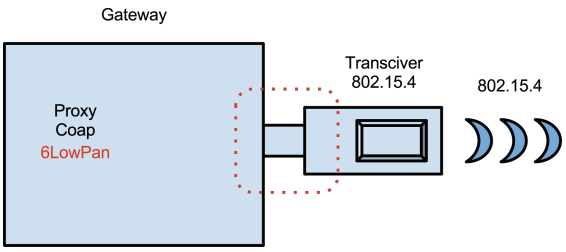


Figure 5.3: External Transceiver

Commercial Device

Texas Instruments proposes the CC2531 USB Evaluation Module Kit [20]. It is an USB Dongle that communicates with a PC throw a serial protocol. The main disadvantage is the needed of a JTAG programmer for programming the stick.

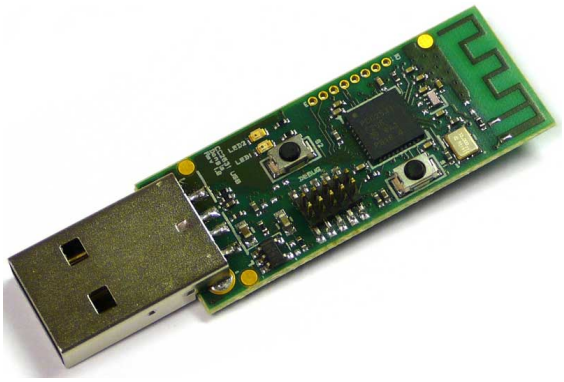
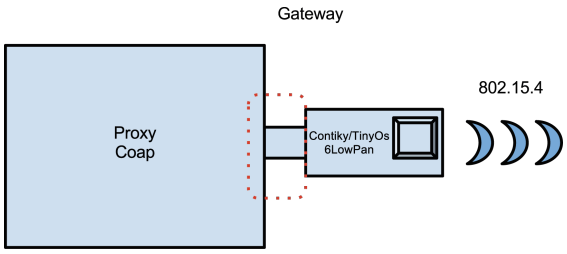


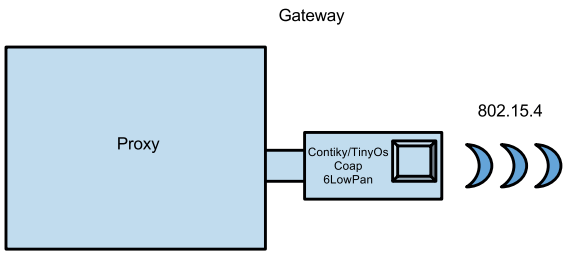
Figure 5.4: CC2531 USB Evaluation Module Kit

5.2.3 External Sensor

The main advantage of this architecture is that it does not need a 6LoWPAN implementation in the Linux Kernel. In details the approach displayed in Figure 5.5a permits the use of a simply program on the sensor that actually only encapsulate informations into 6LoWPAN frames. On the other hand, the architecture showed in Figure 5.5b delegates all the work to the external sensor. Thus the sensor need a CoAP implementation that is well supported by Contiki. The main disadvantage, instead, is the possible bottle-neck between USB and CoAP.



(a) Sensor with 6LoWPAN



(b) Sensor with 6LoWPAN and CoAP

Figure 5.5: External Sensor

Commercial Device

Atmel proposes the RZRAVEN Kit [2]. The Kit consists of:

- RZUSBSTICK
- 2x AVRRAVEN Board

There are a lot of academic projects regarding the RZUSBSTICK ([1]) that use the stick as the building block for realizing a gateway. The board, in fact, supports Contiki and TinyOS but the programming phase still need a JTAG programmer.

Figure 5.6: RZRAVEN Kit

ST MicroEletronics sells a STM32W RF Control Kit [18] that consists of:

- USB Dongle
- Remote Board

The kit supports Contiki and can be programmed without a JTAG programmer (USB cable is enough). On the other hand, especially the Dongle, does not have too much memory for programs upload.

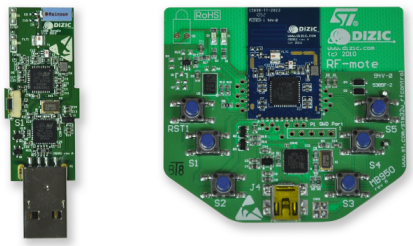


Figure 5.7: STM32W Kit

5.2.4 Gateway Conclusions

The better solution is, probably, the External Sensor solution (Section 5.2.3). There are a lot of devices that can be used and a Linux 6LoWPAN implementation is not needed. All sensors supports Contiki and, as mentioned in Section 2.3, it supports 6LoWPAN and CoAP natively. The Atmel RZRAVEN is more famous but a JTAG programmer is needed, on the other hand the STM32W is less known and also it is a really constrained device but for gateway connections can be a great solution.

5.3 Possibles Sensor Nodes

A Sensor Node is a device with an embedded operation system and a radio interface for communicating. In this work the sensor nodes considered are compliant with Contiki and also have a 802.15.4 transceiver on the 2.4 GHz. All devices are quite similar and differs for physics sensor connections, RAM, and CPU.

5.3.1 TelosB

Figure 5.8: TelosB

Developed by Berkley University, the TelosB [19] is one of the most famous and used sensor node. It is composed by:

- MSP430 processor with 10KB of RAM
- 1MB external Flash
- Light Sensor, Humidity Sensor and Temperature Sensor integrated on board
- GPIO expansion slot
- USB programming

It is well documented especially because it is the most used sensor node in academic projects. On the other hand all projects focus regard the integrated sensors and there are not a lot of documents about external physic sensor connection.

5.3.2 Zolertia Z1

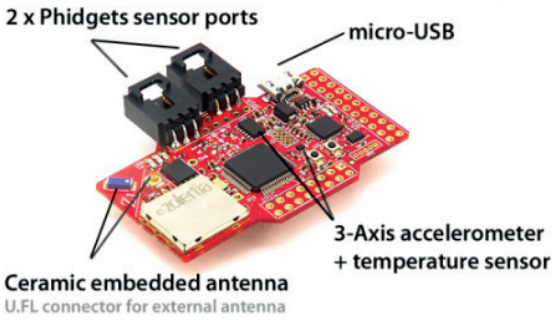


Figure 5.9: Zolertia Z1

Probably it is the most all-around sensor node in commercial [23]. Its design includes:

- MSP430 processor
- External Antenna connector (optionally)
- Accelerometer Sensor and Temperature Sensor integrated on board
- USB Programming
- Standard Analogue Input Connectors
- Standard Digital Input Connectors

The Zolertia Z1 is really well documented and moreover the standard connectors permits to use a great variety of physic sensor.

5.3.3 Econotag

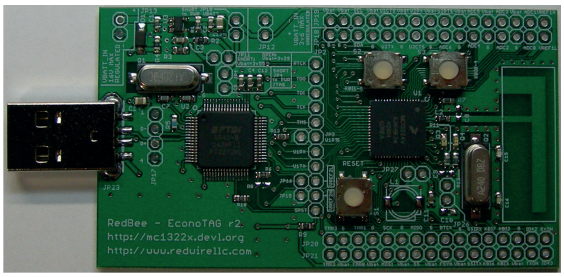


Figure 5.10: Econotag

It is the most cheap sensor node because it is a really constrained device [9]. It is made of:

- MC13224v processor
- USB programming
- USB power supply
- GPIO expansion slot

The USB power supply may be a limitation and also there are not a lot of documentations regarding external sensors.

5.3.4 Sensor Node Conclusions

Probably the best choice is the Zolertia Z1 because it is well supported by constructors and by community, moreover the Zolertia can be connected to a lot of physic sensors in a plug and play way.

5.4 Choices

The overall system is shown in Figure 5.15. All components are COX and cheaper.

5.4.1 Sensors

When someone talks about sensors network he is probably talking about sensor node's network, but in reality the two concepts are different. Sensor is the board that real sense the world while sensor node is a board needed for interact with sensors. In this work four sensors has been used.

Light Sensor

The light sensor, as the name says, simply sense light intensity in Lux. In this work the Phidgets Precision Light Sensor has been used. Its measurement range is from 1 lux (Moonlight) to 1000 lux (TV studio lighting).

The human eye is less sensitive to changes in light intensity than the Precision Light Sensor, but is able to see a wider range. The Human eye range is from 50 microlux (starlight) to 100 klx (extremely bright sunny day). The Precision Light Sensor, on the other hand, is able to measure from 1 lux (Moonlight) to 1000 lux (TV studio lighting). The sensor is designed to respond to visible light, and it can sense light from concentrated sources like laser pointers. It will also have a very muted response to IR light that is close to the visible spectrum (700-800

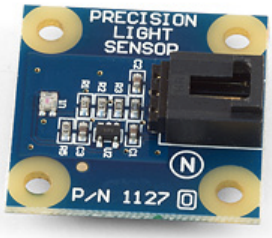


Figure 5.11: Phidgets Precision Light Sensor

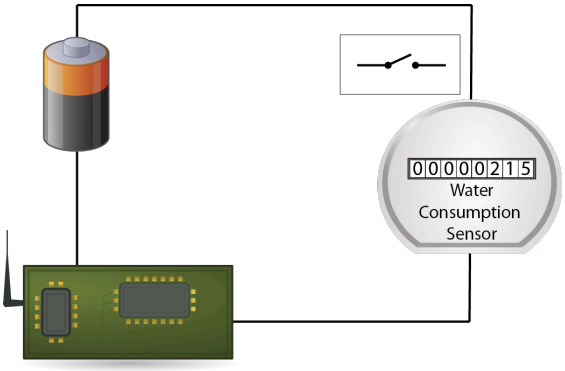


Figure 5.12: Water Consumption Sensor Schema

nm). The Precision Light Sensor is Non-Ratiometric so it is necessary to interpret SensorValue over 950 as saturated, with the true light level being unknown.

Water Consumption Sensor

It is a sensor that measure the consumption of water, in details the sensor notify, with a relay switch, every litre. For using the sensor a simple circuit has to be made as show in Figure 5.12. In details it is an analogue sensor that simply sense the water flow and close a circuit for each litre.

Current Sensor

Measure the current in a wave in Amp. In this work the Phidgets 30 Amp Current Sensor AC/DC has been used. The Phidgets Current Sensor should be wired in series with the circuit under test.

The 30 Amp Sensor measures alternating current (AC) up to 30 Amps and direct current (DC) between –30 and +30 Amps. The AC output will give the RMS (Root Mean Square) value of an alternating current assuming the current is sinusoidal, and the sine wave is varying equally across the zero point. The AC

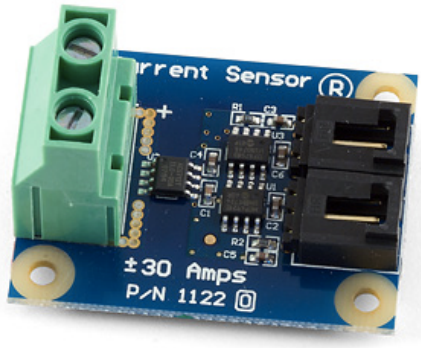


Figure 5.13: Phidgets 30 Amp Current Sensor

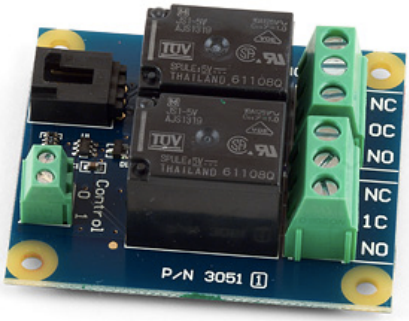


Figure 5.14: Phidgets Dual Relay Board

output can also be used for signals that are not varying evenly around the zero point but the value will be the RMS plus a DC component. If a DC signal is being measured, the AC output will produce a signal that can be used to calculate the current but without the value representing direction of current flow.

Relay

To control the load a relay is necessary. In this work the Phidgets Dual Relay Board has been used. It allows controlling of larger loads and devices like AC or DC motors, electromagnets, solenoids, and incandescent light bulbs.

The relays used in the Dual Relay Board are SPDT relays: single pole, double throw. In an SPDT relay, one of the throw terminals is labelled Normally Closed (NC), and the other is labelled Normally Open (NO). As the name indicates, the normally closed terminal is the terminal connected to common when the relay coil is not powered. When the relay coil is energized by the relay control circuit, the electromagnetic field of the coil forces the switch element inside the relay to break

its contact with the normally closed terminal and make contact with the normally open terminal. The switch element would then connect the normally open terminal and the common terminal.

5.4.2 Sensor Node

The sensor nodes choose for this work are the Zolertia Z1. Reasons are the power, memory dimension, easy sensors connections and good support. In fact the Z1 is sold with two Phidgets connectors already soldered on the board and attached to the ADC. The Z1 also has a 802.15.4 transceiver with an embedded antenna. At the end the use of Contiki with Zolertia is a trivial problem.

An ST MB951 is used as RPL border router, the MB951 is a small board with a 802.15.4 interface and an USB connector. The system use a tunslip interface to communicate on the 802.15.4 world via USB.

5.4.3 Gateways

For gateways two Alix has been used. The Alix has been equipped with the current Voyage Linux release.

The first Alix use a tunslip [1] interface mapped on an USB port for connecting to the ST MB951. Over the tunslip interface there are three processes running: XMHT, CoAP default server and application-layer. The first gateway has also an Ethernet port configured for connecting on an IPv4 LAN.

The second gateway has the same three running processes but no tunslip interface. Instead the gateway has two interface configured: an Ethernet interface for communicate with the first gateway and a Wifi interface for connecting to the rest of the world. The Wifi interface can be used for example by an Android device for controlling the overall system.

5.5 Contiki Implementations

The two Zolertia has been configured with simple programs that sense the Phidgets interfaces and transmit the read value via CoAP once a request is received. The first Z1 is equipped with the current sensor and the relay as shown in Figure 5.16.

[1] TUN is a virtual network kernel device that is supported entirely in software, which is different from ordinary network devices that are backed up by hardware network adapters. TUN simulates a network layer device and it operates with layer 3 packets such as IP packets so TUN is used for routing.

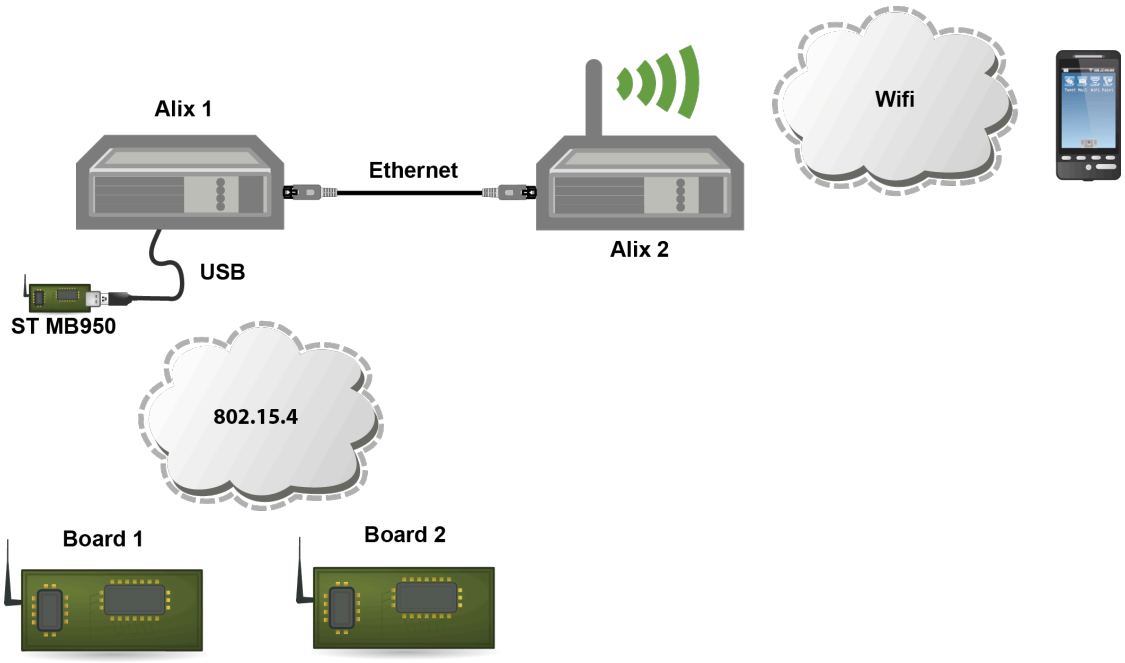


Figure 5.15: Overall System

The second Zolertia has the light sensor and the water consumption sensor as shown in Figure 5.17.

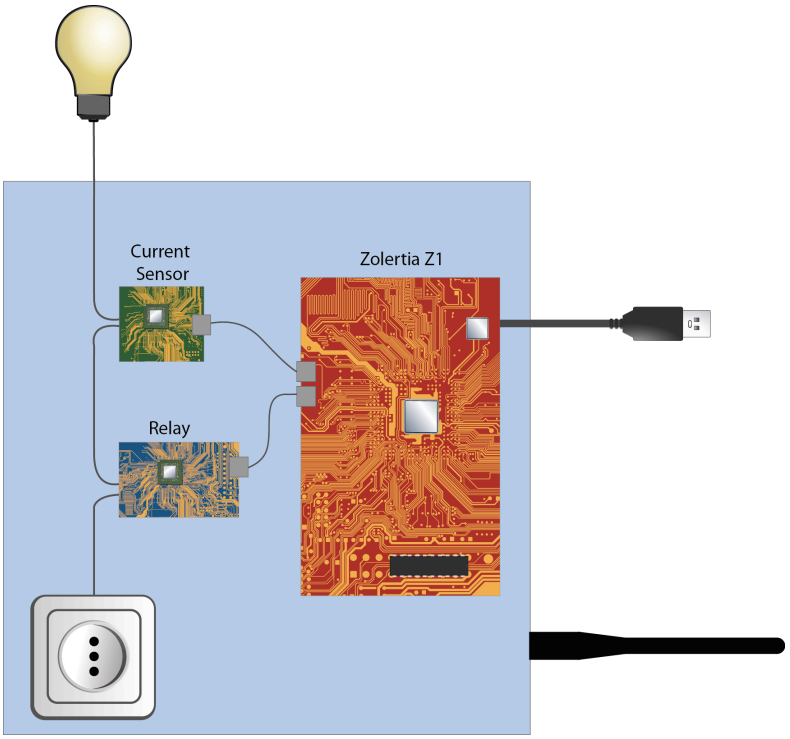


Figure 5.16: First Z1 with the current sensor and the relay

The ST MB951 has been configured with RPL border router, it is a simple program already presents in the Contiki distribution. Anyway the ST MB951 does

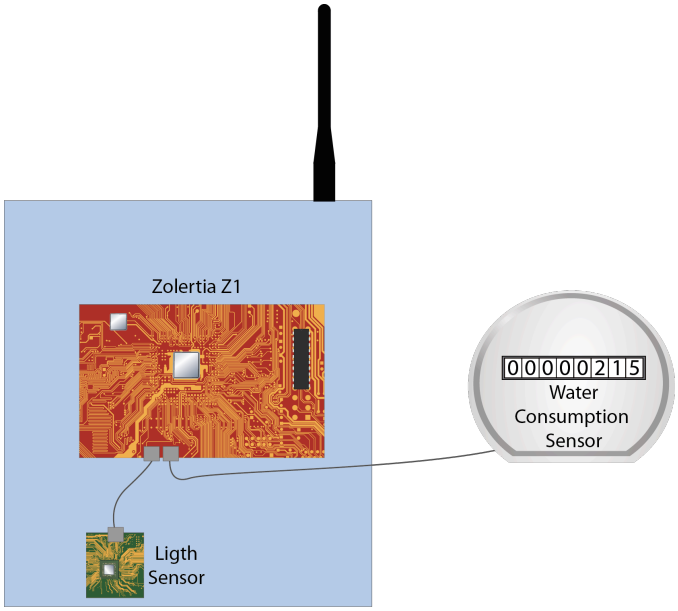


Figure 5.17: Second Z1 with the light sensor and the water consumption sensor

not have a lot on RAM, so a simple MAC algorithm has to be used (nullmac). If ST is substituted by a more enhanced device also more enhanced MAC protocol can be used.

Listing 5.1: Current and Relay Zolertia

```
...

struct phidgets_value
{
  int p5v1;
  int p5v2;
};
typedef struct phidgets_value phidgets_value_t;
phidgets_value_t sm;

...

RESOURCE(current, METHOD_GET, "sensors/current", "title=\"
   Current Sensor\"");
void
current_handler(void* request, void* response, uint8_t *
   buffer, uint16_t preferred_size, int32_t *offset)
{
  char buf[100];
```

```
  float mv = 0;

  printf("Receive request for sensors/current\n");
  /* Calculate Current */
  if(sm.p5v2>13)
    mv = (sm.p5v2 * 0.01142) - 0.118;
  sprintf(buf,"Ampere: %ld.%02d\n", (long) mv, (unsigned) ((
     mv - floor(mv)) * 100));
  int length=strlen(buf);
  memcpy(buffer, buf, length);

  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, length);
}

RESOURCE(relay, METHOD_GET | METHOD_POST | METHOD_PUT , "
   actuators/relay", "title=\"Relay: Status=ON|OFF\"");
void
relay_handler(void* request, void* response, uint8_t *buffer
   , uint16_t preferred_size, int32_t *offset)
{
  size_t len = 0;
  const char *status = NULL;
  int success = 1;
  char buf[100];
  uint8_t method = REST.get_method_type(request);

  if (method & METHOD_GET){
    printf("Receive GET request for actuators/relay\n");
    /* Read bit 4 of P6OUT registry */
    int value = P6OUT;
    value &= 0x10;
        if(value==0)
                sprintf(buf,"Status=OFF");
        else
                sprintf(buf,"Status=ON");
        int length=strlen(buf);
        memcpy(buffer, buf, length);
```

```
        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, length);
  }
  else if ((method & METHOD_POST)||(method & METHOD_PUT)){
        printf("Receive request for actuators/relay\n");
        /* Read payload and search "Status" variable */
        if (success && (len=REST.get_post_variable(request,
           "Status", &status))) {
          if (strncmp(status, "ON", len)==0) {
            P6OUT |= 0x10;
          } else if (strncmp(status, "OFF", len)==0) {
                P6OUT &= 0xEF;
          } else {
                success = 0;
          }
        } else {
          success = 0;
        }
        if (!success) {
          REST.set_response_status(response, REST.status.
             BAD_REQUEST);
        }
        REST.set_response_status(response, REST.status.
           CHANGED);
  }
}

PROCESS(voltage_relay_server, "Current Relay Server");
AUTOSTART_PROCESSES(&voltage_relay_server);

PROCESS_THREAD(voltage_relay_server, ev, data)
{
  static struct etimer periodic_timer;
  PROCESS_BEGIN();
  SENSORS_ACTIVATE(phidgets);

```

```
  printf("RF channel: %u\n", RF_CHANNEL);
  printf("uIP buffer: %u\n", UIP_BUFSIZE);
  printf("LL header: %u\n", UIP_LLH_LEN);
  printf("IP+UDP header: %u\n", UIP_IPUDPH_LEN);
  printf("REST max chunk: %u\n", REST_MAX_CHUNK_SIZE);
  /* Initialize the REST engine. */
  rest_init_engine();
  /* Activate the application-specific resources. */
  rest_activate_resource(&resource_current);
  rest_activate_resource(&resource_relay);
  /* Configure pin 6.4 as output for switching the relay. */
  P6SEL &= 0xEF;
  P6REN &= 0xEF;
  P6DIR |= 0x10;
  P6OUT &= 0xEF;

  printf("Server is waiting\n");
  etimer_set(&periodic_timer, SEND_INTERVAL);
  while(1){
        /* Sense Phidgets values every 3 seconds */
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&periodic_timer)
       );
    sm.p5v1=phidgets.value(PHIDGET5V_1);
    sm.p5v2=phidgets.value(PHIDGET5V_2);
    etimer_reset(&periodic_timer);
  }
  PROCESS_END();
}
```

Listing 5.2: Light and Water Zolertia

```
...

struct datas
{
  int p5v1;
  int count;
};

typedef struct datas datas_t;
datas_t sm;
```

```
int check;

...

RESOURCE(light, METHOD_GET, "sensors/light", "title=\"Light Sensor\"");
void
light_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)
{
  char buf[100];
  printf("Receive request for sensors/light\n");
  /* Calculate Lux (it is not a ratiometric sensor) */
  int lux = (sm.p5v1*0.253152)-8.878637;
  if(lux < 0) lux=0;
  if(lux > 1000) lux=1000;
  sprintf(buf,"Lux: %d", lux);
  int length=strlen(buf);
  memcpy(buffer, buf, length);

  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, length);
}

RESOURCE(water, METHOD_GET|METHOD_POST|METHOD_PUT, "sensors/water", "title=\"Water Sensor POST/PUT value=int\"");
void
water_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)
{
  size_t len = 0;
  const char *value = NULL;
  uint32_t val = 0;
  int success = 1;
  int i=0;
  char buf[100];
  uint8_t method = REST.get_method_type(request);

```

```
  if (method & METHOD_GET)
  {
        printf("Receive GET request for sensors/water\n");
        sprintf(buf,"Litri: %d", sm.count);
        int length=strlen(buf);
        memcpy(buffer, buf, length);

        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, length);
  }
  else if ((method & METHOD_POST)||(method & METHOD_PUT)){
        /* Read payload and search "value" variable */
        if (success && (len=REST.get_post_variable(request,
           "value", &value))) {
          for(i=0;i<len;i++){
            if(!isdigit(value[i]))
                  success=0;
                }
                if(success){
                  val=atoi(value);
                  sm.count=val;
                } else {
                  success = 0;
                }
        } else {
          success = 0;
        }
        if (!success) {
          REST.set_response_status(response, REST.status.
             BAD_REQUEST);
        }
        REST.set_response_status(response, REST.status.
           CHANGED);
  }
}

PROCESS(light_server, "Light Server");
```

```
AUTOSTART_PROCESSES(&light_server);

PROCESS_THREAD(light_server, ev, data)
{
  static struct etimer periodic_timer;
  PROCESS_BEGIN();
  SENSORS_ACTIVATE(phidgets);

  printf("RF channel: %u\n", RF_CHANNEL);
  printf("uIP buffer: %u\n", UIP_BUFSIZE);
  printf("LL header: %u\n", UIP_LLH_LEN);
  printf("IP+UDP header: %u\n", UIP_IPUDPH_LEN);
  printf("REST max chunk: %u\n", REST_MAX_CHUNK_SIZE);
  /* Initialize the REST engine. */
  rest_init_engine();
  /* Activate the application-specific resources. */
  rest_activate_resource(&resource_light);
  rest_activate_resource(&resource_water);

  printf("Server is waiting\n");
  etimer_set(&periodic_timer, SEND_INTERVAL);
  sm.count=0;
  check=0;
  while(1){
        /* Sense Phidgets values every 500 milliseconds */
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&periodic_timer)
       );
        /* Get Light sensor's value */
    sm.p5v1=phidgets.value(PHIDGET5V_1);
        /* Get Water Consumption sensor's value */
    int value=phidgets.value(PHIDGET5V_2);
    if(value>3500 && check==0){
        check=1;
        sm.count++;
    }
    if(value<3500)
        check=0;
    etimer_reset(&periodic_timer);
  }
  PROCESS_END();
```

```
120 }
```

5.6 Client interactions

The first Zolertia Z1 has the current sensor and the relay. A client can interact with them by a CoAP URI. For reading the current sensor value is necessary a GET request to the URI:

$$coap : // < gatewayip > [:< gatewayport >]/sensors/current \tag{5.1}$$

For switching the relay a client has to issue a POST/PUT request with status=on as payload (or status=off for switch off the bulb). The request has to been made with a query that specify the real server where update the resource as mentioned in Section 3.2.3. Anyway for demonstration purpose another query string can be used. In details if the query is:

$$?update \tag{5.2}$$

the application-layer locate the resource already present in the network and update it excluding so the CoAP default server running on the gateway. Thus the complete URI for switching the relay is:

$$coap : // < gatewayip > [:< gatewayport >]/actuators/relay?update \tag{5.3}$$

with the payload set to on or off.

The second Zolertia Z1 has been equipped with the light sensor and the water consumption sensor. For reading the light sensor the URI is:

$$coap : // < gatewayip > [:< gatewayport >]/sensors/light \tag{5.4}$$

while for the water sensor is:

$$coap : // < gatewayip > [:< gatewayport >]/sensors/water \tag{5.5}$$

The water sensor can be also set to a start value. In fact the sensors gives only a signal for each litre, so the Zolertia has to count litres continuously. Because water consumption sensor is analogue, it is important to set start value of Zolertia counter as the one displayed on the sensor. For set the start value a POST/PUT request is needed with payload set to value=<...>.

Chapter 6

Conclusions

This works aims to solve some problems that are interesting in the Internet of Thing view. The proposed solution solves the problem of retrieving resources identifiers in a scalable way and without single points of failure. Furthermore, the solution uses standard protocols only, defined by international organizations, so a lot of manufacturers will be able to explore by themselves this new market. The proposed solution, in fact, can be a great help for costumers and producers by decentralizing control facility directly in homes. This can potentially reduce costs and also offer the users a way to simplify their lives.

The entire work, in fact, creates a totally Service Oriented Architecture where the users ask the network for services. As you can see in Chapter 4 this work is still a prototype and more improvement is necessary before thinking about developing some commercial software. Anyway, it is a good starting point for future works in the IoT field.

Another significant improvement is a new observing concept. A constrained device can not support a lot of observers. This problem can be solved by clients who, instead of registering themselves as observers on a certain resource, can periodically ask for the status of the resource of interest. Anyway, this approach can be a problem if the client is also a constrained device. This work solves the problem in a different way. The infrastructure is responsible for all observing relationships without interfering with clients or servers. If we use this approach, on one hand only gateways are responsible for storing the observing relationships and act like a client who requests a resource. On the other hand, they act like a server with an observing relationship. Gateways can also implement enhanced policies on both sides. For example, on the server side a gateway can periodically send requests with option such as If-None-Match, and the server can respond with "Valid", thus saving bandwidth. On the client's side a gateway can send notifications only if the

requested resource change, thus saving the client's bandwidth.

Moreover the solution proposed in this work permits a lot of other interesting features, such as some information aggregations among services. Imagine an entire building with sensors in all the rooms,. An administrator can ask for a service/resource specifying some parameters he is interested in. These parameters can be used by the application-layer for filtering purposes. Suppose there is a light sensor in each room, each sensor is connected to a gateway and publishes the resource *sensors/light*, so a GET operation on this URI gives the list of all light sensors values. The administrator can issue a GET request with some parameters such as WHERE MINVALUE=<value>. The application-layer on the origin gateway can collect all the resources in the network and then filter values. This behaviour needs that sensors use a standard way for encoding values before the publishing process.

In the next year IoT will become one of the most important fields for research. There are still a lot of open problems that need to be solved. Also this work can be quickly improved by different points of view. First of all the resource discovery process in a p2p network can be an issue. In literature there are a lot of possible approaches to solve the problem. Moreover, commercial software such as the Kad network [1] already solved the problem in a quite scalable way.

Another possible improvement is the CoAP server discovery. In this work a static configuration file has been used but a lot of other "plug'n play" solutions can be used. For example something such as UPnP [2] or, as mentioned in [6], clients (or gateways) can use Multicast CoAP messages and the "All CoAP Nodes" multicast address to find CoAP servers.

[1] Kad is a distributed hash table for decentralized peer-to-peer computer networks designed by Petar Maymounkov and David Mazières in 2002. It specifies the structure of the network and the exchange of information through node lookups. Kad nodes communicate among themselves using UDP. A virtual or overlay network is formed by the participant nodes. Searches are implemented using keywords. The resource name is divided into its constituent words. Each of these keywords is hashed and stored in the network. A search involves choosing one of the keywords, contacting the node with an ID closest to that keyword hash, and retrieving the list of resource names that contain the keyword.

[2] Universal Plug and Play (UPnP) is a set of networking protocols that permits networked devices, such as personal computers, printers, Internet gateways, Wi-Fi access points and mobile devices to seamlessly discover each other's presence on the network and establish functional network services for data sharing, communications, and entertainment.

Bibliography

[1] *Anuj Sehgal, Jurgen Schonwalder - 6lowpan with uIPv6 in Contiki.* URL: http://dak664.github.com/contiki-doxygen/a01682.html (cit. on p. 49).

[2] *Atmel RZRAVEN.* URL: http://www.atmel.com/tools/RZRAVEN.aspx (cit. on p. 48).

[3] *Ben WPAN.* URL: http://en.qi-hardware.com/wiki/BenWPAN (cit. on p. 45).

[4] *Blockwise transfers in CoAP - draft-ietf-core-block-09.* URL: http://datatracker.ietf.org/doc/draft-ietf-core-block/ (cit. on p. 14).

[5] *Conditional observe in CoAP - draft-li-core-conditional-observe-02.* URL: http://datatracker.ietf.org/doc/draft-li-core-conditional-observe/ (cit. on p. 14).

[6] *Constrained Application Protocol (CoAP) - draft-ietf-core-coap-11.* URL: http://datatracker.ietf.org/doc/draft-ietf-core-coap/ (cit. on pp. 11, 13, 66).

[7] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors". In: *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks.* LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462. ISBN: 0-7695-2260-2. DOI: 10.1109/LCN.2004.38. URL: http://dx.doi.org/10.1109/LCN.2004.38 (cit. on p. 18).

[8] Adam Dunkels et al. "Protothreads: simplifying event-driven programming of memory-constrained embedded systems". In: *Proceedings of the 4th international conference on Embedded networked sensor systems.* SenSys '06. Boulder, Colorado, USA: ACM, 2006, pp. 29–42. ISBN: 1-59593-343-3. DOI: 10.1145/1182807.1182811. URL: http://doi.acm.org/10.1145/1182807.1182811 (cit. on p. 18).

[9] *Econotag.* URL: http://redwirellc.com/store/node/1 (cit. on p. 51).

[10] *IEEE Standard Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*. IEEE Computer Society (cit. on p. 4).

[11] *iMote2*. URL: http://www.instrumentationdevices.it/images/upload/20101124115356imote2datasheetid.pdf.pdf (cit. on p. 45).

[12] *Linux-Zigbee*. URL: http://sourceforge.net/apps/trac/linux-zigbee/ (cit. on pp. 45, 46).

[13] *LNX-6LoWPAN*. URL: http://code.google.com/p/lnx-6lowpan/ (cit. on p. 45).

[14] Enzo Mingozzi and Claudio Cicconetti. "Enabling technologies and standards for mobile multi-hop wireless networking" (cit. on pp. 8, 10).

[15] *Miscellaneous additions to CoAP - draft-bormann-coap-misc-20*. URL: http://datatracker.ietf.org/doc/draft-bormann-coap-misc/ (cit. on p. 14).

[16] *Observing Resources in CoAP - draft-ietf-core-observe-05*. URL: http://datatracker.ietf.org/doc/draft-ietf-core-observe/ (cit. on p. 14).

[17] *OpenEmbedded*. URL: http://www.openembedded.org/wiki/MainPage (cit. on p. 46).

[18] *STM32W Low-cost RF control kit Active*. URL: http://www.st.com/internet/evalboard/product/251361.jsp (cit. on p. 49).

[19] *TelosB*. URL: http://www.instrumentationdevices.it/images/upload/2010_11_24_12_19_43_telosb_datasheet_id.pdf.pdf (cit. on p. 50).

[20] *Texas Instruments USB Evaluation Module Kit*. URL: http://www.ti.com/tool/cc2531emk (cit. on p. 47).

[21] Dr. Ovidiu Vermesan et al. "Internet of Things Strategic Research Roadmap". In: (). URL: http://www.internet-of-things-research.eu/pdf/IoT_Cluster_Strategic_Research_Agenda_2011.pdf (cit. on p. 1).

[22] *ZigBuzz*. URL: http://zigbuzz.sourceforge.net/ (cit. on p. 45).

[23] *Zolertia Z1*. URL: http://zolertia.sourceforge.net/wiki/index.php/MainPage (cit. on p. 51).