Ear-IT Network Qualification

Deliverable D1.1 " Network conditions for use of acoustic sensors"

Abstract

This document presents the SmartSantander (Santander) and HobNet (Geneva) test-bed network qualification. The Santander test-bed mainly consists in Libelium WaspMote motes that have high flexibility but limited capacity. We showed that the maximum realistic sending throughput is about 17kbps with 802.15.4 modules. When taking the reception side, a maximum of about 12kbps could be achieved without error. However, it is possible to increase the level of performance by appropriate modifications of programming API and higher baud rates to increase the data transfer rates. Regarding the HobNet test-bed, we found that AdvanticSys nodes based on a TelosB mote architecture have a much higher level of performance, resulting in faster sending rate and forwarding capabilities, than the WaspMote board. Maximum realistic sending throughput can reach a bit more than 30kbps. Reception throughput for AdvanticSys motes is limited by the sending throughput. Preliminary tests with a low-bit rate audio encoding scheme are quite promising regarding the possibilities of sending acoustic traffic on the SmartSantander and HobNet networks.

Project Number:	Project Acronym:	Project Title:

318381	EAR-IT	Experimenting Acoustics in Real environments using Innovative Test-beds

Instrument:	**Thematic Priority**
STREP	Future Internet Research and Experiment

Title
Ear-IT Network Qualification **Deliverable D1.1 " Network conditions for use of acoustic sensors"**

Contractual Delivery Date:	**Actual Delivery Date:**
1st July 2013	15 July 2013

Start date of project:	**Duration:**
October, 1st 2012	24 months

Organization name of lead contractor for this deliverable:	**Document version:**
EGM	V1.0

Dissemination level (Project co-funded by the European Commission within the Seventh Framework Programme)		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission	
RE	Restricted to a group defined by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Authors (organizations) :

Congduc Pham, Alexandre Berge, Philippe COUSIN (EGM)

Abstract :

This document presents the SmartSantander (Santander) and HobNet (Geneva) test-bed (transport) network qualification. The Santander test-bed mainly consists in Libelium WaspMote motes that have high flexibility but limited capacity. We showed that the maximum realistic sending throughput is about 17kbps with 802.15.4 modules. When taking the reception side, a maximum of about 12kbps could be achieved without error. However, it is possible to increase the level of performance by appropriate modifications of programming API and higher baud rates to increase the data transfer rates. Regarding the HobNet test-bed, we found that AdvanticSys nodes based on a TelosB mote architecture have a much higher level of performance, resulting in faster sending rate and forwarding capabilities, than the WaspMote board. Maximum realistic sending throughput can reach a bit more than 30kbps. Reception throughput for AdvanticSys motes is limited by the sending throughput. Preliminary tests with a low-bit rate audio encoding scheme are quite promising regarding the possibilities of sending acoustic traffic on the SmartSantander and HobNet networks.

Keywords :

Acoustic data, audio streaming, network qualification.

Disclaimer

Revision History

The following table describes the main changes done in the document since it was created.

Revision	Date	Description	Author (Organisation)
0.5	April	Main writing with Santander results	C.Pham
0.6	June, 28th	Add Hobnet part	C. Pham
0.7	4th July	Add some considerations for protocols qualification	Alex Berge, P.Cousin
0.9	16th July	Pre-final for internal review	P.Cousin
1.0	25th July	Finalisation V1	P.Cousin

Table of Content

1. EAR-IT, Test-bed Network Qualification

Review of the qualification objectives

Qualified (FIRE) Acoustic IoT Installations for Applications

This entire project is based on audio data processing. Because such data are collected by sensors mainly within WSNs (wireless sensors networks), which have their own constraints and complexity and where a lot of protocols underlying the technologies are very new ones, we face network validation challenges.

The challenges are to qualify first the network to assess under which conditions we can get the audio data without "losing the applications". Secondly once the conditions are known and described, it will be important to provide "benchmarks" in order to ensure qualification of a test bed or particular environment to meet the requirements and therefore ensure reproducibility and reproducibility.

The specific challenges are then:

- Be sure that the audio data can be processed without network perturbation in order to enable a smooth capture/processing of audio data for specific applications;
- Determine what are the minimum conditions / requirements to provide a stable transportation of audio data for specific services;
- Provide benchmark and associated methodology to assess existing and future environment for meeting requirements and conditions to deploy targeted services

Objective #1: QUALIFICATION
Qualify and Benchmark Test-beds for Acoustics in Deployment of Targeted Applications
• Put in place a clear testing methodology to support RTD in a well-defined environment using acoustic sensor networks • Develop benchmarks to qualify the readiness of test-beds for equivalents experiment using EAR-IT technologies • Support the validation of new and emerging technologies (e.g. protocols) for supporting audio data applications

WP1 of EART-IT is to ensure qualification of various test beds to be able to deliver audio data and to bring feedback to researchers on potential limitations.

Within WP1 there are three tasks:

T1.1 Transport Network Qualification (M01 - M09)

Objectives- This task will explore the network topology and environment to smoothly establish applications link allowing audio data stream. This task will investigate network performance at protocol and system levels. Robustness of new protocols still under deployment such as 6lowpan, COAP, etc. will be investigated

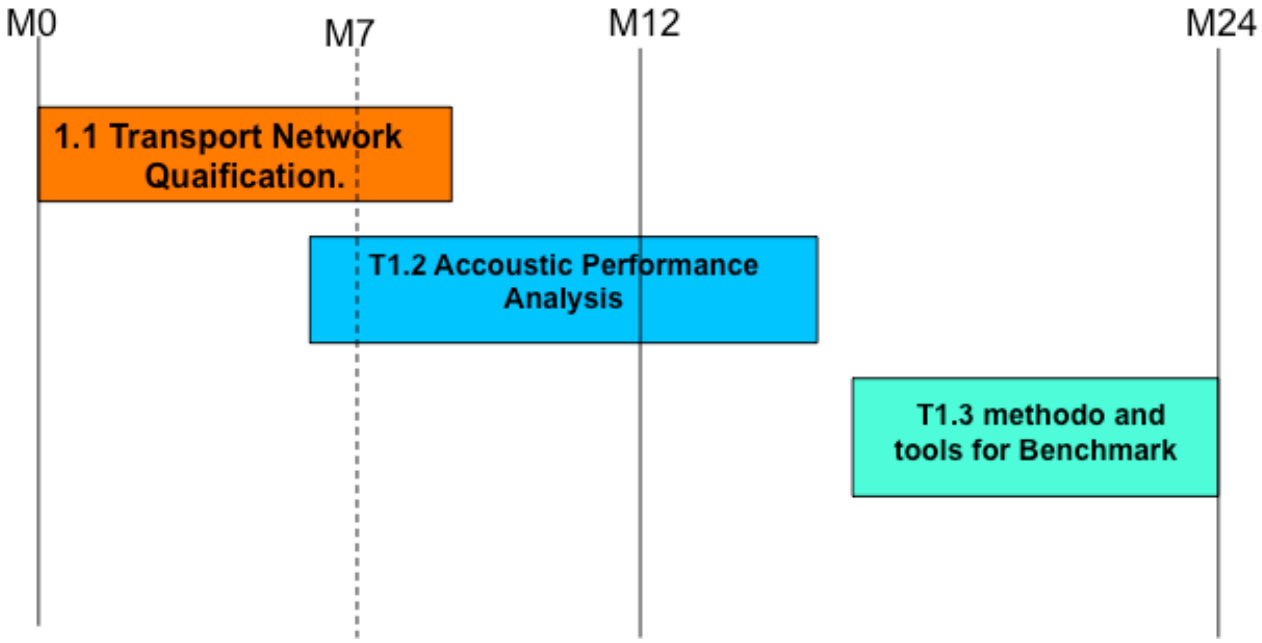
T1.2 Acoustic Performance Analysis (EGM, M06 - M15):
Objectives- While network condition well established under previous task and links established to allow audio data stream, this part will investigate the minimum requirements and quality necessary for the exploitation of audio data as well as repeatability of the experiments. This

will be done by specific audio measurement to qualify the environment and this will performed in close coordination with WP2 and WP3

T1.3 Methodology and Tools for Measurements and Benchmarking (M16 - M24):

Objectives- As soon as precise technical network and acoustic parameters will be qualified in previous tasks, it will be possible to define a methodology and tool for measurement and benchmarking allowing to qualified an entire test bed and sensor network for its use with acoustic sensors for specific usages. Such methodology will help to qualify other test beds or sensors network for specific applications. This will also allow ensuring quality, repeatability and reproducibility of the experiments or operational deployment in case of real operational services.

The tasks are carried out according the following schedule :



This deliverable D1.1 is the main result of the task T1.1 and will be realized on :

1. the Santander's SmartSantander **"**Static Environmental Monitoring**"** test-bed based on Libelium WaspMote hardware
2. the HOBNET UNIGE in-door test-bed mainly based on Advanticsys and TelosB sensor hardware

The main objective of the qualification process is to determine some upper bounds on the network performances that an end-user could get when deploying experimental applications. Therefore the main orientation of this study is towards best cases results.

As stated in the DoW the outcomes of T1.1 to be documented in D1.1 are to:

1. *"Precise the condition of network readiness to carry out audio data for the envisaged EAR-IT applications".* This is the main results of the D1.1 with the chapters 2 to 8.
2. "*Provide technical feedback to the worldwide community on new technologies (i.e. new protocols) and on their ability to be used for the audio-related applications*." This is done in Chapter 8 for the Hobnet test bed using 6lowpan and CoAP. "*Potentially provide new world-wide test cases and associated tools"* After the M6 review this part

was not considered a priority but status of protocols such as 6lowpan and CoAP are given in annex

2. Test-beds & network infrastructure

The SmartSantander test-bed at Santander

The SmartSantander project (www.smartsantander.eu) builds large-scale experimental testbeds consisting mainly in IEEE 802.15.4 devices (IoT nodes, Internet of Things) and gateways. Full details can be find on the project web page, especially additional details on the ressource reservation system and the dynamic code deployment system. In this document, we will only focus on the qualification of the communication infrastructure for IoT nodes and gateways in the context of the hardware deployed for the Environmental Monitoring use case in Santander city. The picture below illustrates such an IoT node deployment configuration in the centre of the Santander city.

Figure 1: Outdoor parking and Environmental Monitoring deployed architecture

Below is a brief description as found in the project document of the main nodes and network elements of the test-bed for Environmental Monitoring use cases.

1. **IoT node**: Responsible for sensing the corresponding parameter (temperature, CO, noise, light, car presence, soil temperature, soil humidity). The majority of them are integrated in the repeaters, whilst the others stand alone communicating wirelessly with the corresponding repeaters (it is the case for the parking sensor buried under the asphalt). For these devices, due to the impossibility of powering them with electricity, they must be fed with batteries.
2. **Repeaters**: These nodes are high-rise placed in street lights, semaphores, information panels, etc, in order to behave as forwarding nodes to transmit all the information associated to the different measured parameters. The communication between repeaters and IoT nodes performs through 802.15.4 protocol.
3. **Gateways**: Both IoT nodes and repeaters, are configured to send all the information (through 802.15.4 protocol), experiment-driven as well as service provision and network management to the gateway. Once information is received by this node, it can either store it in a database which can be placed in a web server to be directly accessed from internet, or send it to another machine (central server), through the different interfaces provided by it (WiFi, GPRS/UMTS or ethernet).

IoT nodes are WaspMote sensor boards and gateways are Meshlium gateways, both from

Libelium. Most of IoT nodes are also repeaters for muti-hops communication to the gateway. Here is a brief description of these 2 elements as found in the project document.

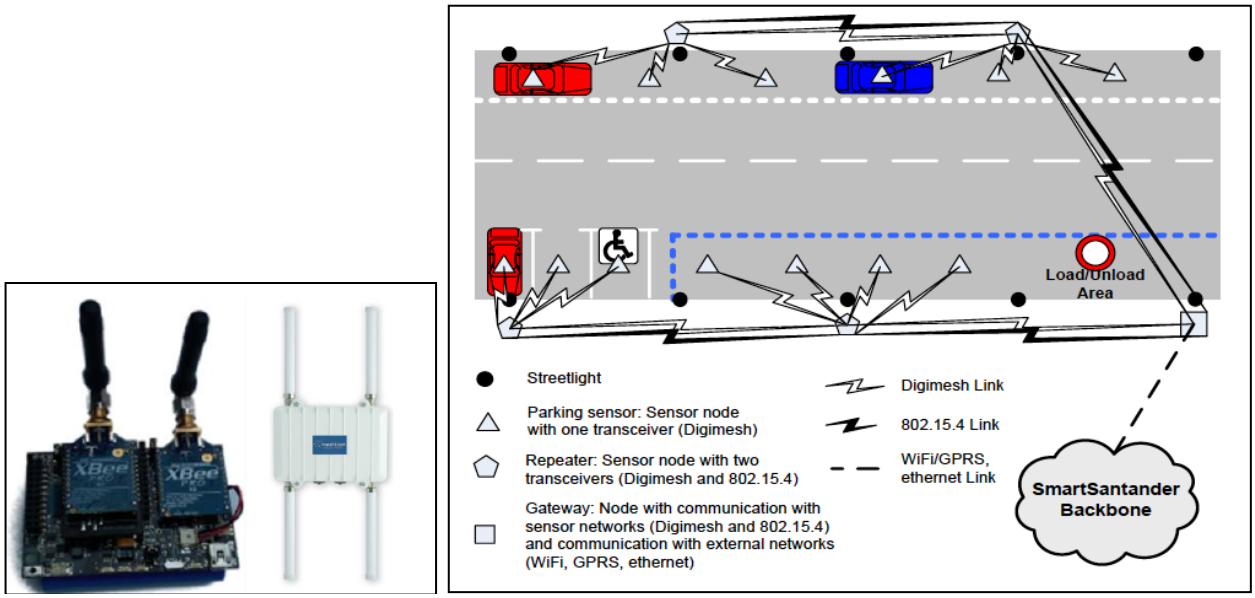


Figure 2: Left: Deployed IoT node and gateway. Right: Communications between IoT nodes and gateways

The IoT node depicted in figure 2 is composed of the following parts:

- **Main board**: This board (called Waspmote) is in charge of processing and memory issues, providing a set of interfaces for attaching different types of sensors (both analogue and digital), as well as to plug several radio modules to communicate with other nodes. The Waspmote comes with with a ATmega1281 microcontroller, and several types of memory, 8KB SRAM, 4KB EEPROM, 128KB FLASH and an extra storing SD memory with 2GB capacity. On the other hand, 7 analogue and 8 digital interfaces are available for external sensor connection, as well as 1 PWM, 2UART, 1 I2C and 1 USB interfaces for attaching different communication modules. All the development tools (libraries, API's, etc.) provided by Libelium are based on a pseudo-wiring solution which aims to promote the simplicity of the functioning of the micro-processor based on events and loops. Attached to the main board, they are placed the sensor boards with the corresponding sensing capabilities, such as temperature, luminosity, noise, parking, CO, soil temperature.
- **Two XBee-PRO radio modules**: Both modules manufactured by Digi company, run over 2.4 GHz frequency. One of the modules implements 802.15.4 protocol in a native way, and the other one runs 802.15.4 protocol modified with a proprietary routing protocol called Digimesh. This is a proprietary peer-to-peer networking topology protocol for use in wireless end-point connectivity solutions, allowing addressing in a simple way.

More details on the hardware will be given in the next section. Below is a typical communication scenario in the Environmental Monitoring use case that shows IoT nodes (attached on streetlights) with the 2 radio modules approach: 802.15.4 and Digimesh. The application/user traffic data associated to experimentation are handled by the 802.15.4 radio module while the Digimesh radio module is dedicated for network management and control traffic. The Meshlium gateway (also attached on some streetlights) also has the same 2 radio module interface with the same functionalities. In addition to the 802.15.4 and Digimesh interfaces, the gateway has external communication features such as WiFi, GPRS and Ethernet, depending on which technology is available.

SmartSantander hardware details

A/ The **Libelium** WaspMote

An IoT node in the Santander test-bed consists in a WaspMote sensor board shipped by Libelium. The WaspMote is built around an Atmel1281 microcontroller running at 8MHz with 128KB of flash memory available for the user application. Full specifications of WaspMote can be found in [WASP]. Below is a picture of the board, without any sensor board nor communication module.

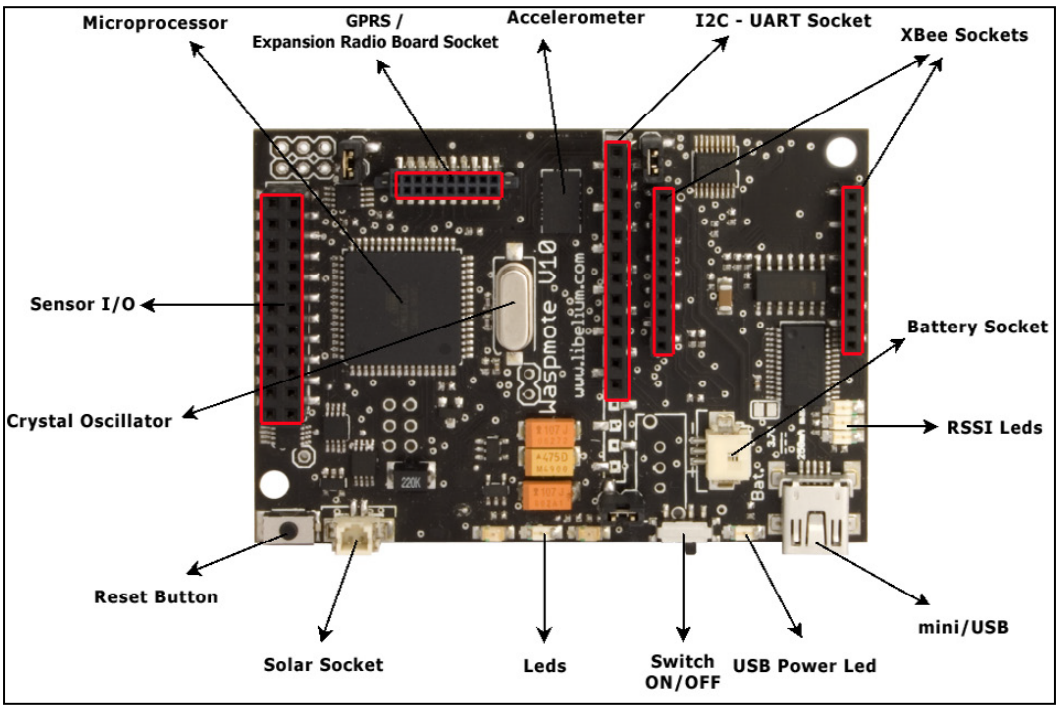


Figure 3: Libelium WaspMote

The WaspMote has a number of I/O interfaces: UARTs, SPI and I2C buses, analog and digital pins. There are 6 UARTs in the WaspMote that serve various purposes, the one that is relevant for our study is the UARTs which connects the microcontroller to the radio modules: UART0 and UART1 for the default XBee Socket and the Expansion Radio Board Socket of figure 3 respectively. The XBee socket can directly receive an XBee radio module from Digi International (see http://www.digi.com) that offers various connectivity technologies: 802.15.4, Digimesh/ZigBee, WiFi, 900 & 860MHz. The Radio Expansion Socket can receive a dedicated GSM/GPRS module (figure 4, left) or a radio expansion board that offers a second XBee connectivity board (figure 4, right). Various connectivity combinations can therefore be realized. Figure 5 shows a WaspMote with 2 XBee modules: (i) XBee 802.15.4 on a radio expansion board connected to the Radio Expansion socket and, (ii) XBee Digimesh on the XBee socket.

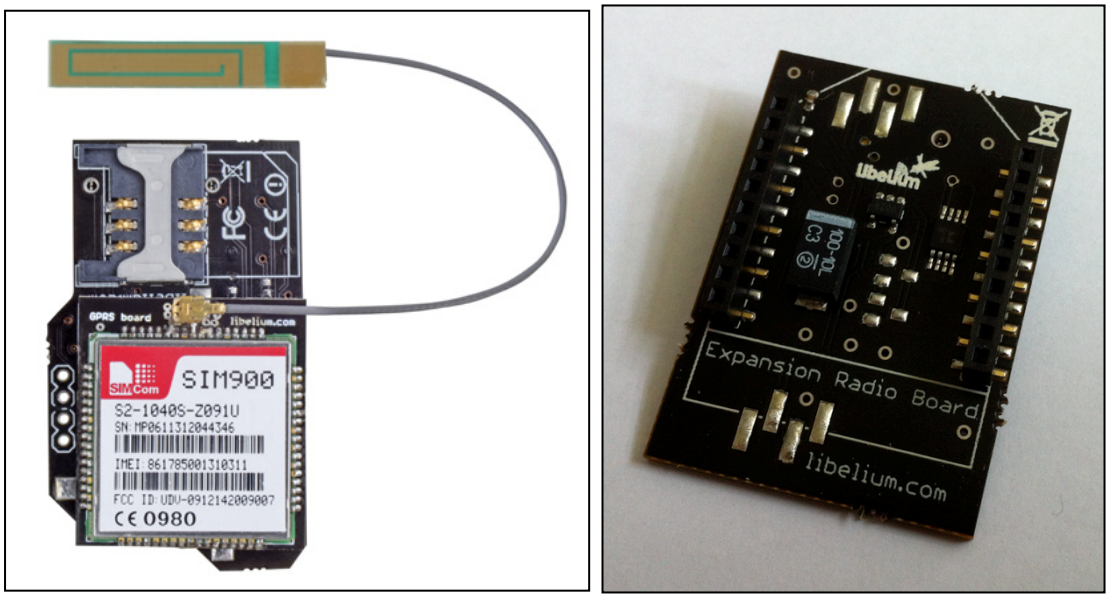


Figure 4: GSM/GPRS module (left), Radio Expansion Board (right)

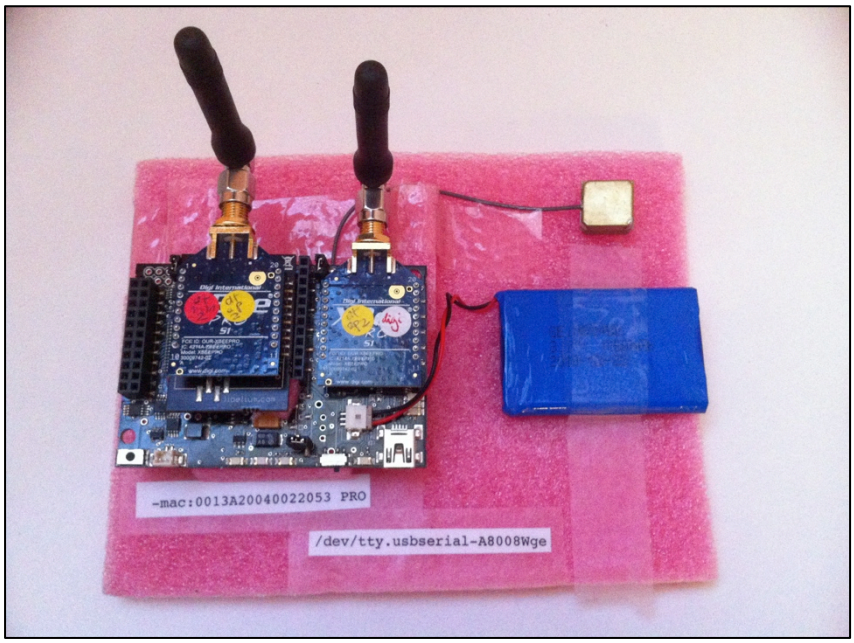


Figure 5: WaspMote with 2 radio modules that reproduces an IoT node in the SmartSantander network

Figure 6 below shows the data signal block diagram of the WaspMote.

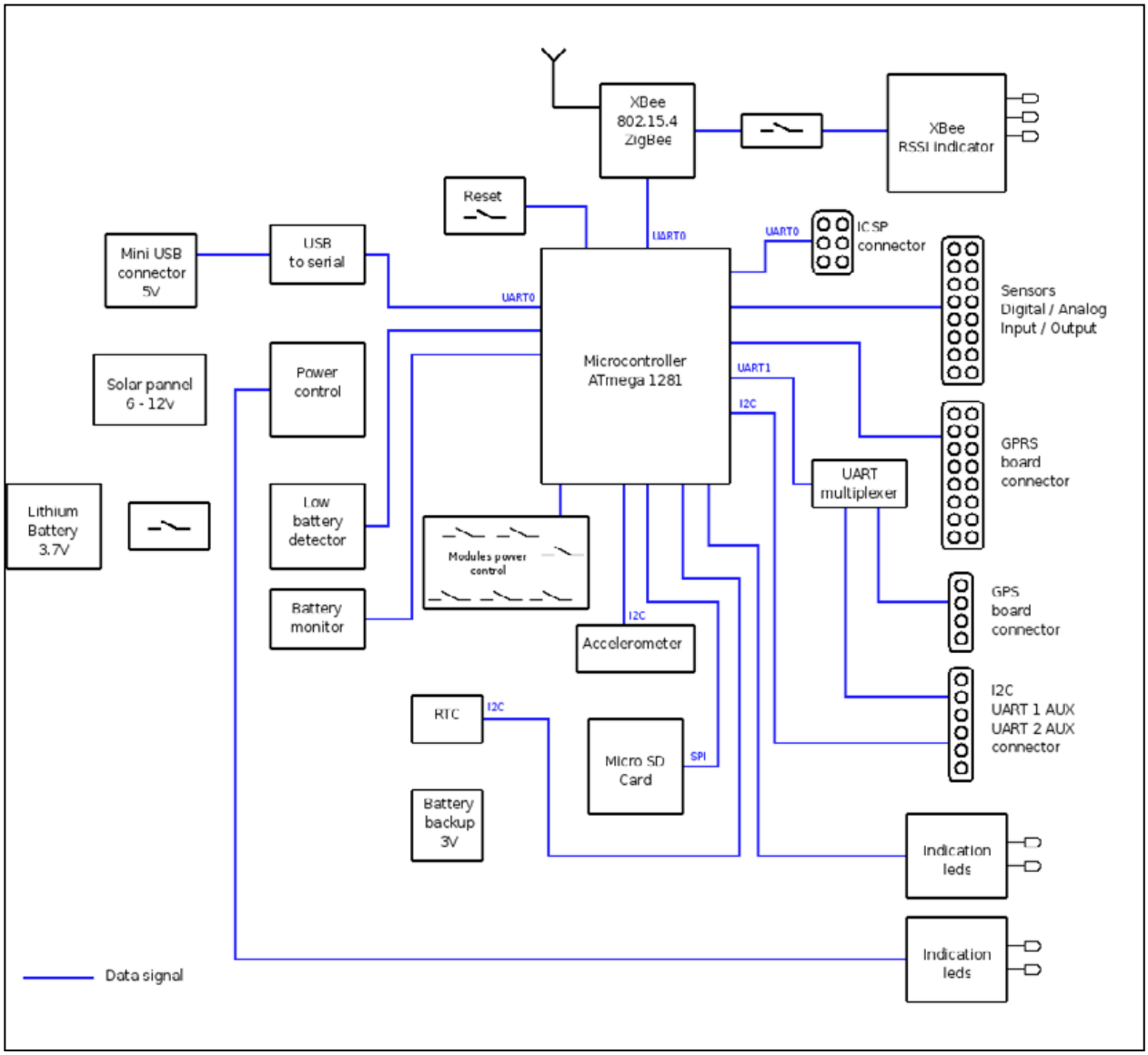


Figure 7: Data signal block diagram

As indicated previously, UART0 is connected to the XBee socket while UART1 is connected to the Radio Expansion socket allowing for a second radio communication module.

We will provide more details about the UART-XBee interactions, and limitations, in the Network Qualification section.

B/ The XBee 802.15.4 modules from Digi International

The Libelium Waspmote uses radio modules from Digi International. As said previously, IoT nodes have one XBee 802.15.4 module and one XBee Digimesh module. Differences between the 802.15.4 and the Digimesh version are that Digimesh implements a proprietary routing protocols along with more advanced coordination/node discovery functions. XBee 802.15.4 offers the basic 802.15.4 [802154] PHY and MAC layer service set in non-beacon mode. 802.15.4 and Digimesh can co-exist together but no direct communications are possible between the 2 variants. Both 802.15.4 and Digimesh are available from Digi in either "normal" or "pro" version. "pro" version uses a higher transmit power: maximum for "pro" is 63mW while maximum for "normal" is 1mW. Details on the XBee/XBee-PRO 802.15.4 modules can be found in [XBeeDigi][DMDigi]. Figure 7 shows a "normal" and a "pro" XBee 802.15.4 module.

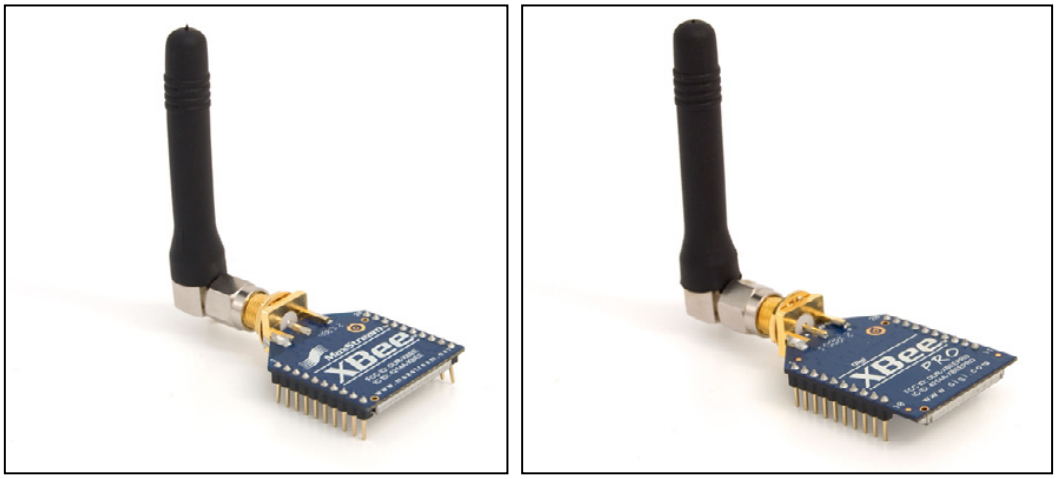


Figure 7: XBee 802.15.4 module (left), XBee 802.15.4 PRO module (right)

With the XBee-PRO, European regulations limit the maximum transmit power output to 10dBm therefore the maximum allowed power for the XBee-PRO is 10mW.

Figure 8 shows a table taken from [XBeeDigi] which summarizes important specifications of the XBee/XBee-PRO 802.15.4 module. Maximum range in Europe (therefore applicable to the SmartSantander testbed) is then the one of the International variant at 10mW.

Table 1-01. Specifications of the XBee®/XBee-PRO® RF Modules

Specification	XBee	XBee-PRO
Performance		
Indoor/Urban Range	Up to 100 ft (30 m)	Up to 300 ft. (90 m), up to 200 ft (60 m) International variant
Outdoor RF line-of-sight Range	Up to 300 ft (90 m)	Up to 1 mile (1600 m), up to 2500 ft (750 m) international variant
Transmit Power Output (software selectable)	1mW (0 dBm)	63mW (18dBm)* 10mW (10 dBm) for International variant
RF Data Rate	250,000 bps	250,000 bps
Serial Interface Data Rate (software selectable)	1200 bps - 250 kbps (non-standard baud rates also supported)	1200 bps - 250 kbps (non-standard baud rates also supported)
Receiver Sensitivity	-92 dBm (1% packet error rate)	-100 dBm (1% packet error rate)

Figure 8: XBee/XBee-PRO 802.15.4 module important specifications

Actually, the module hardware for 802.15.4 and Digimesh is the same. Then user can upload either 802.15.4 or Digimesh firmware with the XCTU [XCTU] utility tool provided by Digi.
An IoT node uses 2 XBee-PRO modules, one with the 802.15.4 firmware (on UART1) and the other with the Digimesh firmware (on UART0). As said previously, the 802.15.4 module is available for experimentations (peer-to-peer traffic can then be performed with this interface) while the management and service traffic are handled by the Digimesh module. Note that an IoT can send experimentation results to its associated gateway through the DigiMesh interface. With the Digimesh routing features, Over-The-Air (OTA) code deployment or communication in a multi-hop manner is natively possible whereas routing must be handled specifically by the application/user code with the 802.15.4 module.

Figure 9 shows a table taken from [DMDigi] which summarizes important specifications of the XBee/XBee-PRO Digimesh module.

Specifications of the XBee/XBee-PRO® DigiMesh 2.4 RF Module

Specification	XBee	XBee-PRO®
Performance		
Indoor/Urban Range	up to 100 ft (30 m)	up to300 ft (90 m), up to 200 ft (60 m) intl. variant
Outdoor RF line-of-sight Range	up to 300 ft (90 m)	up to1 mile (1.5 km) w/2.0 dB dipole antenna up to6 miles (10 km) w/high gain antenna
Transmit Power Output	1 mW (0 dBm)	63 mW (18 dBm)* 10 mW (10 dBm) for international variant
RF Data Rate	250 kbps	250 kbps
Serial Interface Data Rate (software selectable)	1200 bps - 250 kbps (non-standard baud rates also supported)	1200 bps - 250 kbps (non-standard baud rates also supported)
Receiver Sensitivity	-92 dBm (1% packet error rate)	-100 dBm (1% packet error rate)

Figure 9: XBee/XBee-PRO Digimesh module important specifications

We can see that 802.15.4 and Digimesh versions are quite equivalent at the physical level even though no direct communications are possible between them.

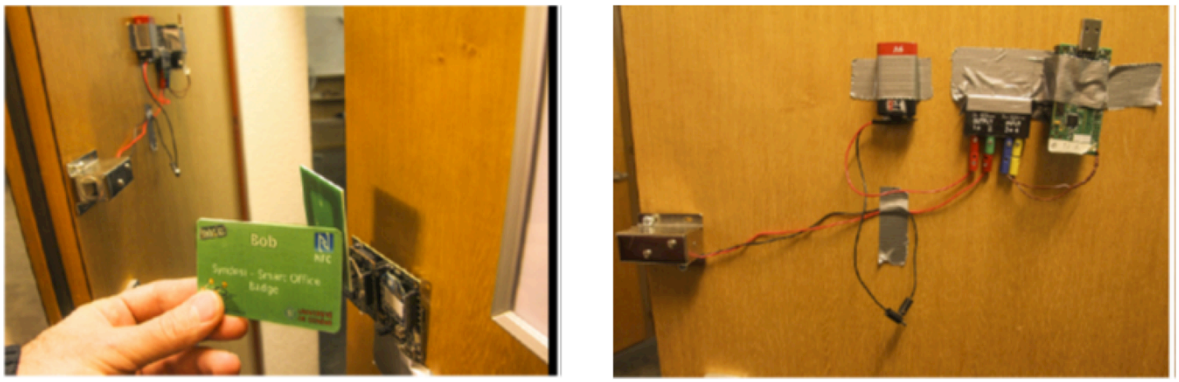
In the context of the EAR-IT project we will study the performance of both the 802.15.4 and DigiMesh XBee module and provide comparison between the 2 firmwares in terms of maximum sending throughput for instance. Therefore, the results could be used to determine which technology is most suitable for acoustic services.

The UNIGE HobNet test-bed

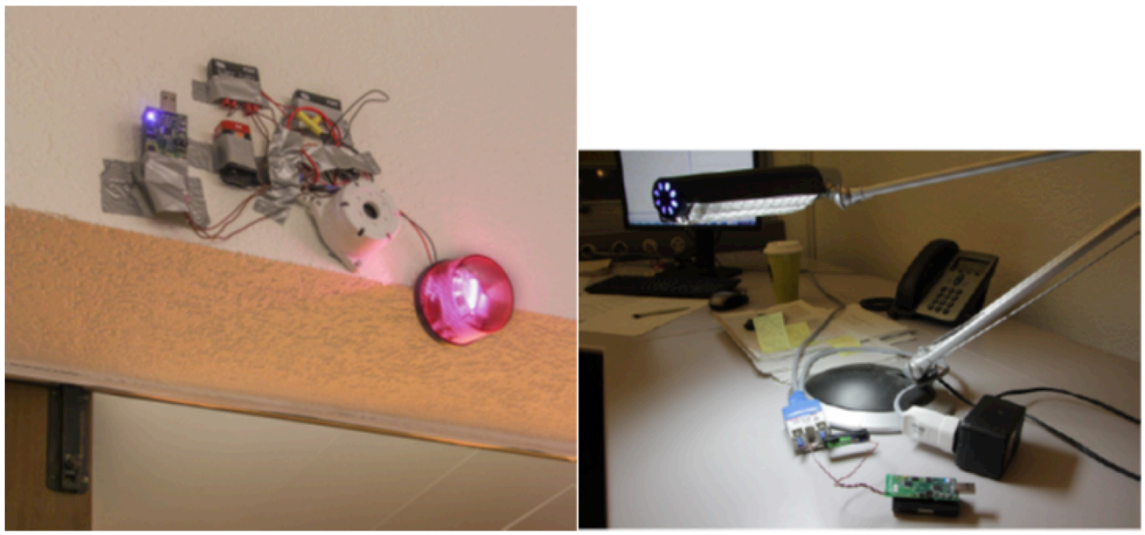
The HobNet test-bed (www.hobnet-project.eu) is a FIRE test-bed that focuses on Smart Building for the Future Internet. Although the HobNet test-bed has several sites, within the EAR-IT project only the UNIGE test-bed at the University of Geneva is concerned. The UNIGE test-bed consists in TelosB-based motes as described by the following descriptions from the HobNet web site.

The UNIGE test-bed. The testbed comprised of 20 TelosB motes is a fixed network running IPv6/6LoWPAN and COAP protocols. Some motes are used as mobile motes for measuring temperature and humidity in different places of our testbed. Specified scenarios defined in the WP1 have been implemented in the testbed such as: Local adaptation to presence, energy management (partially), electric device monitoring (partially), maintenance control (partially), user centric environment customization and mobile phone ID. The topology of the network is shown in the picture below. With a star symbol is represented a mote which acts as a sensor and with a cross a mote which acts as an actuator connected to an electronic device.

Furthermore, we have added to the testbed two libelium waspmotes with NFC chips, two wasmotes with WiFi antennas and two waspmotes with Bluetooth antennas, together with their expansion boards. Moreover electrical components such as relays, cables, motors, locks have been installed to the testbed, in order to be able to control and monitor electronic devices such as lamps, curtains, fans, heaters etc. The tesbed can be accessed through the internet and a website dedicated for it and also through an android application. Each node can be accessed both separately through their unique IPv6 address and as a group of sensors with the help of a COAP server which is running in the backbone of the system.



NFC Reader for User Identification (left). – TelosB mote activating/deactivating a door lock (right)



Fire alarm system including siren and light (left) – Desk lamp controlled by a TelosB mote (right).

The HobNet hardware details

Sensor nodes in the HobNet test-bed consist in Berkeley TelosB/TmoteSky motes and AdvanticSys motes (mainly CM5000 and CM3000, see figure 10) that are themselves based on the TelosB architecture. These motes are built around an TI MSP430 microcontroller with an embedded ChipCon CC2420 802.15.4 compatible radio module.

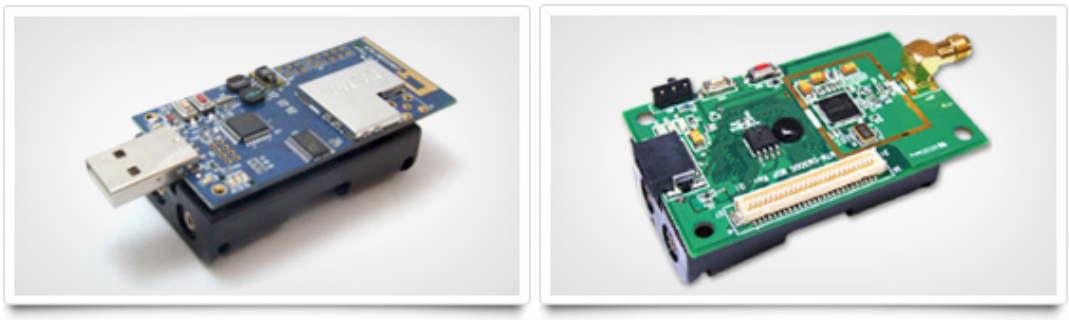


Figure 10: CM5000 (left) and CM3000 (right)

The TelosB description and datasheet can be found in [TELOSB]. Documentation on the AdvanticSys motes can be found in [ADVAN]. The CC2420 radio specification and documentation are described in [CC2420] but we will provide below a summary of its main characteristics. These motes have built-in temperature, humidity and luminosity sensors.

Figure 10a shows the schematic block diagram of a TelosB mote which is also valid for the AdvanticSys motes.

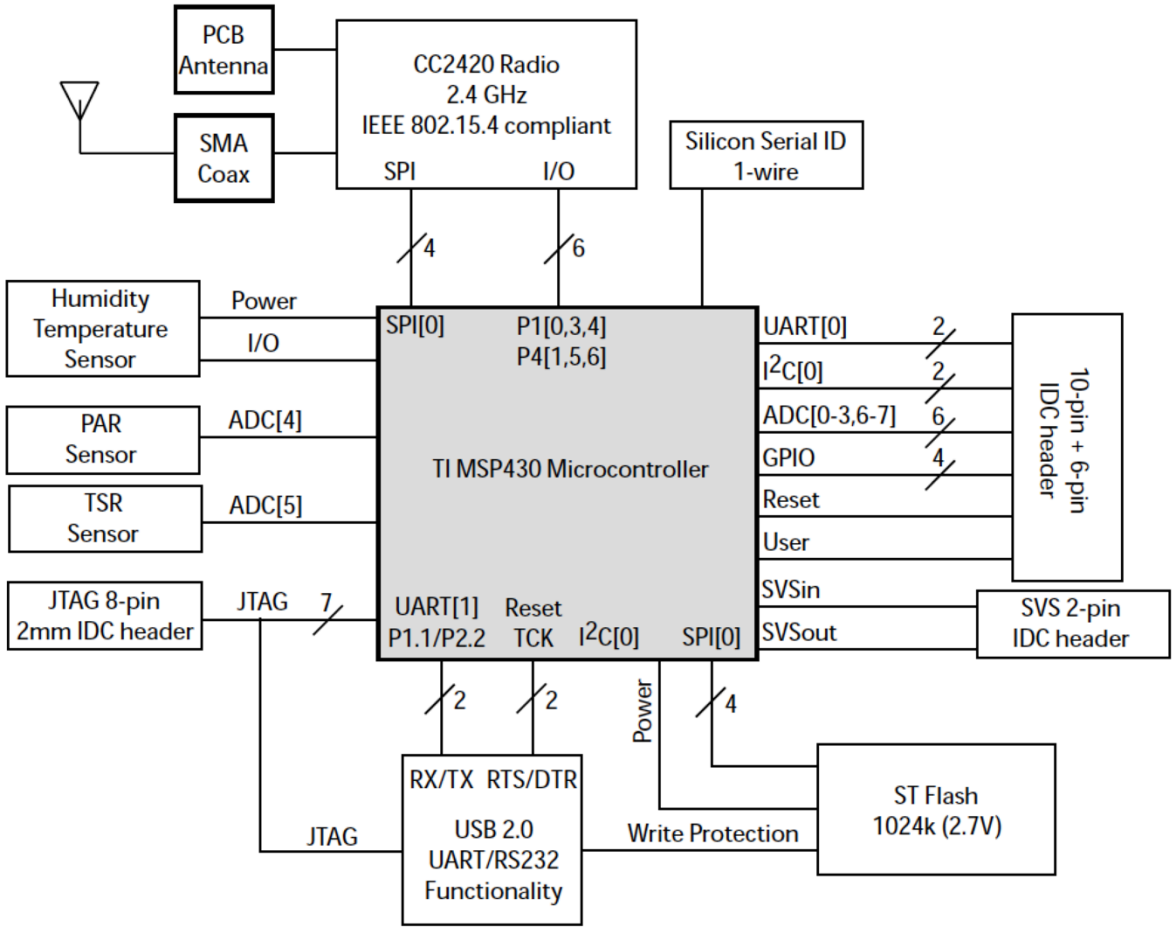


Figure 10a: Functional Block Diagram of the Telos Module, its components, and buses

The important difference compared to the previous Libelium WaspMote is that the radio module is connected to the microcontroller through an SPI bus instead of a serial UART component. This normally would allow for much faster data transfer rates.

3. Qualification tasks

Given the objective of the EAR-IT project, the network qualification process will be organized in 2 phases.

The first phase will address the following 3 main qualification objectives as explained in the EAR-IT project.

Objective #1: QUALIFICATION
Qualify and Benchmark Test-beds for Acoustics in Deployment of Targeted Applications
• Put in place a clear testing methodology to support RTD in a well-defined environment using acoustic sensor networks • Develop benchmarks to qualify the readiness of test-beds for equivalents experiment using EAR-IT technologies • Support the validation of new and emerging technologies (e.g. protocols) for supporting audio data applications

The second phase will address the following objective:

Defining benchmarks is one challenge; executing a benchmark in an experimenter-friendly way, sharing benchmarks with the research community, and/or building a sustainable framework is equally important and the EAR-IT project will work to progress state of the art in this domain.

Network qualification tasks

1. Qualification of the Libelium WaspMote and Advanticsys platforms: throughput, latency, reliability level and loss rate,
    a. Radio module to radio module
    b. Microcontroller to radio module
    c. Impact of software APIs
2. Identification & tests of important radio module parameters (radio and Medium Access Control level) and their impact on performances in a networked environment
    a. 1-hop and 2-hops
    b. prediction for k>2

3. Qualification of communications between IoT nodes, repeaters nodes and gateway nodes. Performance of the Digimesh routing.

a. impact of traffic load
b. impact of node density
c. impact of network topology
d. impact of routing mechanisms

4. Preliminary study of audio traffic on the SmartSantander infrastructure (Libelium WaspMote platform)
    a. Identification of the requirements of audio traffic in the context of the EAR-IT project
    b. Synthetic workloads for audio traffic

5. Deployment, execution of benchmarks and synthesis
    a. Deployment methods in the context of SmartSantander infrastructure (API, middleware,...)
    b. Test campaigns
    c. Identification of bottlenecks

4. SmartSantander network qualification: 1-hop

802.15.4 PHY Maximum application throughput

The latest standard for 805.15.4 is described in [802154]. The radio maximum throughput is 250kbs but it is useful to understand what effective throughput could be obtained at the application level. These bounds will be useful when comparing the maximum theoretical throughput to what could be measured on real platform with communication stacks between the user applications and the radio module. We will consider a non-beacon mode with CSMA/CA channel access as this mode theoretically offers the least overhead.

A/ Determining the maximum MAC payload

The IEEE 802.15.4 PSDU (PHY Service Data Unit) is depicted in figure 11 taken from [802154].

<table>
<tr><td>Octets: 2</td><td>1</td><td>0/2</td><td>0/2/8</td><td>0/2</td><td>0/2/8</td><td>0/5/6/10/ 14</td><td>variable</td><td>2</td></tr>
<tr><td rowspan="2">Frame Control</td><td rowspan="2">Sequence Number</td><td>Destination PAN Identifier</td><td>Destination Address</td><td>Source PAN Identifier</td><td>Source Address</td><td rowspan="2">Auxiliary Security Header</td><td rowspan="2">Frame Payload</td><td rowspan="2">FCS</td></tr>
<tr><td colspan="4">Addressing fields</td></tr>
<tr><td colspan="7">MHR</td><td>MAC Payload</td><td>MFR</td></tr>
</table>

Figure 11: IEEE 802.15.4 general frame format

In 802.15.4 networks, node addressing could be realized using 16-bit address or 64-bit address. Therefore the addressing field could be a minimum of 0 bytes and a maximum of 20 bytes which results in a header size between 5 bytes and 25 bytes in the case of no security header. As the maximum frame size at radio level is *aMaxPHYPacketSize*=127 bytes, the maximum payload is 127-5 = 122 bytes. This payload is the MAC payload which is also what is available for the application in case of no routing overhead (quite common in wireless sensor

networks where routing could be done at application layer).

However, realistic scenarios with non-zero size addressing field are as follows:

1. 4 bytes decomposed as follows : 16-bit address for SA and 16-bit identifier for SPAN. DA and DPAN have zero size for the particular case of sending to a PAN coordinator.
2. 8 bytes decomposed as follows : 16-bit address for both DA and SA, 16-bit identifier for both DPAN and SPAN.
3. 20 bytes decomposed as follows : 64-bit address for both DA and SA, 16-bit identifier for both DPAN and SPAN. **This is the scenario that is most likely to occur on the SmartSantander testbed.**

The maximum MAC payload is then 127-*minimumHeaderSize*(5)-*addressingFieldSize*(4, 8, 20) which give respectively 118, 114 and 102 bytes.

When an IEEE 802.15.4 radio sends a frame, it adds a preamble sequence on 4 bytes, a start of frame indicator on 1 byte and a frame length field on 1 byte. This is depicted in figure 12 where the SHR and PHR is put in front of the PHY PSDU to obtained the so-called PPDU (PHY Protocol data Unit).

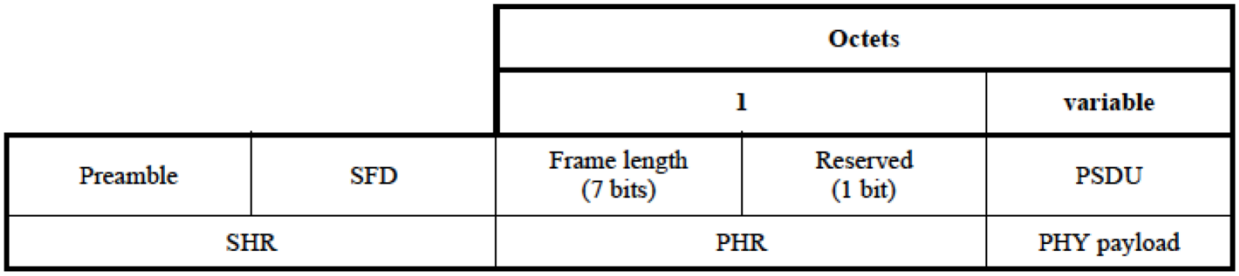


Figure 12: IEEE 802.15.4 PPDU

Therefore, the PPDU total size at the radio level for a frame transmission is *aMaxPHYPacketSize*+*SHR*+*PHR*=133 bytes.

B/ Channel Access Time (see [JENNIC])

Non-beacon enabled IEEE 802.15.4 networks use an unslotted CSMA-CA channel access mechanism. Each time a device needs to transmit, it waits for a random number of unit back-off periods in the range {0, 2^{BE}-1} before performing the Clear Channel Assessment (CCA).

Initially, the back-off exponent *BE* is set to *macMinBE*. Using the default value of 3 for *macMinBE* and assuming the channel is found to be free, the worst-case channel access time can be calculated as:

$$InitialbackoffPeriod + CCA = (2^3 - 1) \times aUnitBackoffPeriod + CCA$$
$$= 7 \times 320\ \mu s + 128\ \mu s$$
$$= 2.368\ ms$$

The CCA detection time is defined as 8 symbol periods. *aUnitBackoffPeriod* is defined as 20 symbol periods. 1 symbol period is equal to 16 µs.

However, *macMinBE* could be set to 0 to increase efficiency, in which case the CSMA/CA channel access time will default to the minimum Long Inter-Frame Spacing (LIFS) of 0.640 ms.

C/ Maximum throughput with no ACK

Normally, a unicast frame needs to be acknowledged by the receiving side. However, in

broadcast mode, there is no such acknowledgement and it is therefore possible to simply determine the maximum achievable throughput. We will show results for *macMinBE*=0 and *macMinBE*=3 (default IEEE 802.15.4 value).

no ack (broadcast) macMinBE=0			
max frame phy size (bytes)	133		
transmission time (ms)	4,256		
macMinBE	0		
back-off time (ms)	0,64		
worst case transmission time (ms)	4,896		
802.15.4 header size	9	13	25
max payload size (bytes)	118	114	102
max app throughtput (bps)	192810,46	186274,51	166666,67

no ack (broadcast) macMinBE=3			
max frame phy size (bytes)	133		
transmission time (ms)	4,256		
macMinBE	3		
back-off time (ms)	2,368		
worst case transmission time (ms)	6,624		
802.15.4 header size	9	13	25
max payload size (bytes)	118	114	102
max app throughtput (bps)	142512,08	137681,16	123188,41

D/ Maximum throughput with ACK, no error

When ACK is required (unicast communication) the IEEE 802.15.4 standard stipulates that the transmission of an acknowledgment frame (in a non-beacon enabled network) commences *aTurnaroundTime* symbols after the reception of the data frame, where *aTurnaroundTime* is equal to 192 µs. This allows the device enough time to switch between transmit and receive, or vice versa.

An acknowledgment frame consists of 11 bytes that can be transmitted in 0.352ms. The transmission of an acknowledgement does not use CSMA/CA. Once again, we will show results for *macMinBE*=0 and *macMinBE*=3 and we assume here that there are no error.

with ack, macMinBE=0			
max frame phy size (bytes)	133		
transmission time (ms)	4,256		
macMinBE	0		
back-off time (ms)	0,64		
ack transmission time (ms)	0,352		
turnaround time (ms)	0,192		
worst case transmission time (ms)	5,44		
802.15.4 header size	9	13	25
max payload size (bytes)	118	114	102
max app throughtput (bps)	173529,41	167647,06	150000,00

with ack, macMinBE=3			
max frame phy size (bytes)	133		
transmission time (ms)	4,256		
macMinBE	3		
back-off time (ms)	2,368		
ack transmission time (ms)	0,352		
turnaround time (ms)	0,192		
worst case transmission time (ms)	7,168		
802.15.4 header size	9	13	25
max payload size (bytes)	118	114	102
max app throughtput (bps)	131696,43	127232,14	113839,29

E/ Maximum throughput with ACK, 1 error

We can extend the previous study by considering the case of 1 error that need 1 retransmission by the sender. The default number of retransmission at the MAC level is 3. The transmitting node will wait *macAckWaitDuration* symbol periods for an acknowledgment before it attempts a retry, where *macAckWaitDuration* is equal to 54 symbol periods (0.864 ms). Once again, we will show results for *macMinBE*=0 and *macMinBE*=3.

with ack, macMinBE=0, 1 retry			
max frame phy size (bytes)	133		
transmission time (ms)	4,256		
macMinBE	0		
back-off time (ms)	0,64		
mac Ack Wait Duration time (ms)	0,864		
back-off time (ms)	0,64		
transmission time (ms)	4,256		
ack transmission time (ms)	0,352		
turnaround time (ms)	0,192		
worst case transmission time (ms)	11,2		
802.15.4 header size	9	13	25
max payload size (bytes)	118	114	102
max app throughtput (bps)	84285,71	81428,57	72857,14

with ack, macMinBE=3, 1 retry			
max frame phy size (bytes)	133		
transmission time (ms)	4,256		
macMinBE	3		
back-off time (ms)	2,368		
mac Ack Wait Duration time (ms)	0,864		
back-off time (ms)	2,368		
transmission time (ms)	4,256		
ack transmission time (ms)	0,352		
turnaround time (ms)	0,192		
worst case transmission time (ms)	14,656		
802.15.4 header size	9	13	25
max payload size (bytes)	118	114	102
max app throughtput (bps)	64410,48	62227,07	55676,86

F/ Maximum throughput under given Packet Error Rate (PER)

We could predict the maximum mean expected throughput by considering various value of PER. For instance, a PER of 25% means that 25% of transmitted data frames need 1 retry whereas 75% of transmitted data frames are successfull at the first try. In this case, we can use the following approximation to compute an average data frame transmission time:

Average data frame transmission time =
PER*worst_case_tr_time_error + (1-PER)*worst_case_tr_time_no_error

Once again, we will show results for *macMinBE*=0 and *macMinBE*=3.

macMinBE=0			max app throughput (bps)		
PER (%)	Correct pkt (%)	Average tr. Time	118	114	102
25,00%	75,00%	6,88	137209	132558	118605
30,00%	70,00%	7,168	131696	127232	113839
35,00%	65,00%	7,456	126609	122318	109442
40,00%	60,00%	7,744	121901	117769	105372
45,00%	55,00%	8,032	117530	113546	101594
50,00%	50,00%	8,32	113462	109615	98077
55,00%	45,00%	8,608	109665	105948	94796
60,00%	40,00%	8,896	106115	102518	91727

macMinBE=3			max app throughput (bps)		
PER (%)	Correct pkt (%)	Average tr. Time	118	114	102
25,00%	75,00%	9,04	104425	100885	90265
30,00%	70,00%	9,4144	100272	96873	86676
35,00%	65,00%	9,7888	96437	93168	83361
40,00%	60,00%	10,1632	92884	89736	80290
45,00%	55,00%	10,5376	89584	86547	77437
50,00%	50,00%	10,912	86510	83578	74780
55,00%	45,00%	11,2864	83640	80805	72299
60,00%	40,00%	11,6608	80955	78211	69978

More details could also be found in [LAT06].

The 802.15.4 XBee module from Digi International

The XBee module IoT nodes we consider is the XBee-PRO 802.15.4 radio module. This radio module is compliant with the IEEE 802.15.4 PHY layer and implements a non-beacon mode with CSMA/CA channel access. The XBee module has a number of networking parameters that will determine its behavior.

By default, *macMinBE*=0 (noted Random Delay Slots, ATRN command, on the XBee).

ACK mode can be controlled with the Mac Mode parameter (ATMM command). When MM=1, there is no ACK, even for unicast communications. With MM=2, we have the standard behavior of the IEEE 802.15.4 protocol: no ACK for broadcast communication and ACK for unicast communication. Note that it is the Mac Mode of the sender that determines the ACK behavior.

Regarding the maximum payload, the XBee radio module defines a maximum of 100 bytes for the application/MAC payload whatever addressing mode is used: 16-bit address or 64-bit address.

Doing so is simpler as with 64-bit address, the 802.15.4 standard limits the payload to 102 bytes. Therefore the XBee's 100 bytes maximum payload works in all cases. In addition, Digi reserve 2 bytes in the user payload to implement what they call "Digi Mac Mode 0" to enable a number of Digi specific operations such as node discovery, remote AT command,…, thus the 100 bytes instead of 102 bytes.

Communication stacks/APIs and their impacts on performance

Network qualification can not really make abstraction of the communication stack (with software communication API and libraries) that introduces large overheads for each packet transmission, thus reducing further the maximum achievable throughput.

We can summarize the various process of sending an application packet in figure 13 which also shows various time overheads introduced by memory copies, data transfers on various interfaces or buses (UART, I2C, SPI, …) , physical transmission, physical propagation, protocol-dependant overheads, …

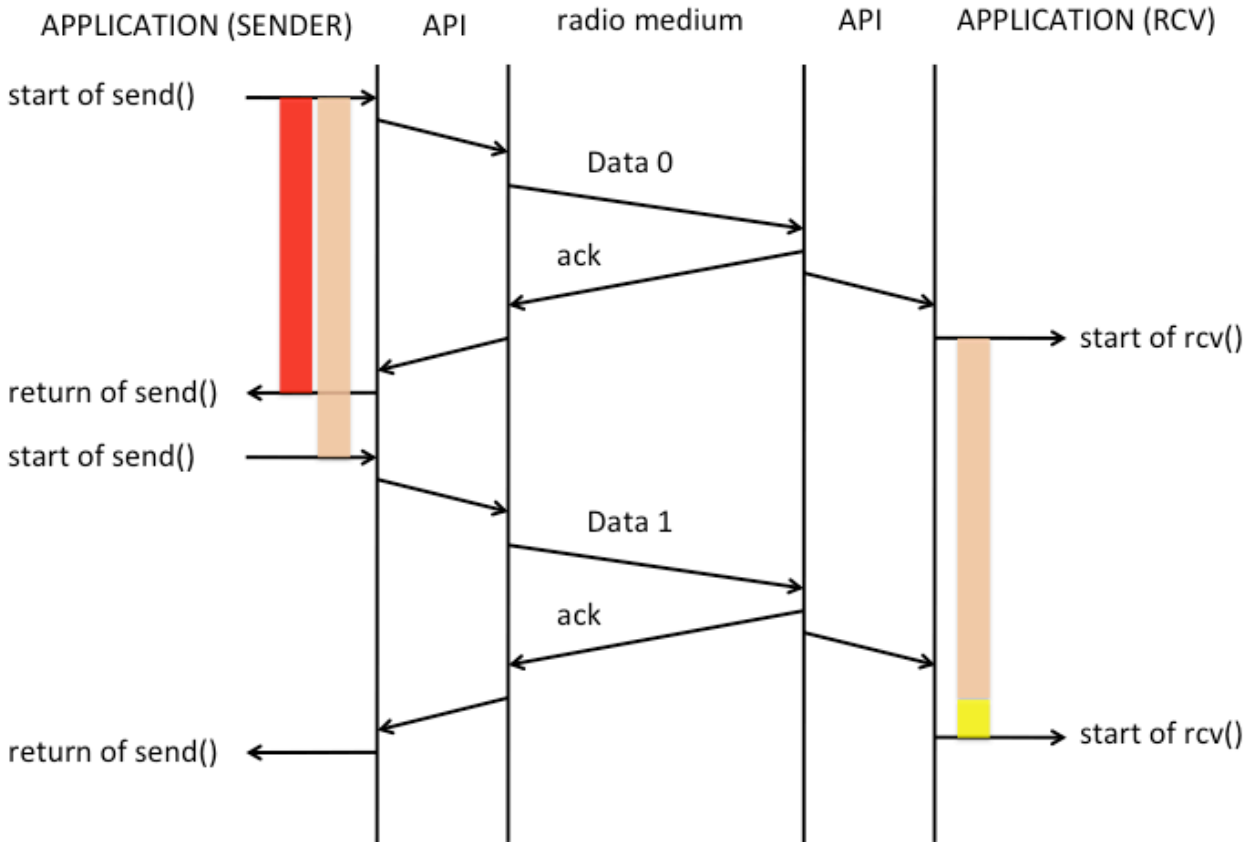


Figure 13: Timing of a sending process

At the application layer, sending packet back-to-back is strongly limited by the time needed to return from a generic send() function (red bar). Sometimes, this limitation is further increased (orange bar) when the next send() call cannot be performed just after the return. Assuming that the orange bar represents $T_{send()}$ seconds, then the maximum achievable sending throughput at the application level is :

$$TH_{App}\ (bps) = Payload(bytes) * 8 * (1/ T_{send()})$$

In what follows, we presents experimental measures performed on the test hardware depicted previously in figure 6. We use a Traffic Generator with timing functionalities to measure (up to 1 millisecond granularity) the overhead introduced at various stages of the sending process. We will show results for 2 cases: (i) with full Libelium sending API and (ii) with light Libelium sending API. We will use Libelium API v0.31 (v0.32 is available but it only improves the GPRS stack). Both modes have the following generic stages:

```
send(packet) {
      generate_frame; // construct frame to be sent to the radio
```

```
            write_to_radio(packet); // uses UART send to the XBee

            parse_message() {
                  tx_status_response; // get the response from XBee
                  process_answer;
            }
      }
```

A/ With full Libelium API

The full API version has advanced packet handling features that mask most of the complexity of data sending and receiving such as fragmentation and reassembling. It also adds an additional header as depicted by figure 14 taken from [WASP802].

Application ID	Fragment Number	First Fragment Indicator [#]	Source Type ID	Source ID	Data
0x52	1	#	0	0x12 0x21	A A A … A A (50 Bytes)

Application ID	Fragment Number	First Fragment Indicator [#]	Source Type ID	Source ID	Data
0x52	1	#	1	0x00 0x13 0xA2 0x00 0x40 0x30 0xF6 0x86	A A A … A A (50 Bytes)

Application ID	Fragment Number	First Fragment Indicator [#]	Source Type ID	Source ID	Data
0x52	1	#	2	forrestNode-01	A A A … A A (50 Bytes)

Figure 14: 3 variants of the Libelium API header

The Source Type Id field determine 3 methods of source node identification: 16-bit identifier, 64-bit identifier, and node string identifier. Therefore the extra header overhead is between 6 bytes and 24 bytes as the string identifier can be 20-byte long.

In our test platform, we use the following extra header that has a size of 9 bytes using the string identifier "WASP#". This approach is also a tradeoff between the 6 bytes of the 16-bit source id mode and the 12 bytes of the 64-bit source id mode. Both being more difficult to manage than the string identifier mode for a human operator.

1 byte	1 byte	1 byte	1 byte	5 bytes
0x00 – 0xFF	1	#	2	WASP#

Therefore, as the maximum application payload with the XBee is 100 bytes, the effective application maximum payload is 91 bytes.

B/ With light Libelium API

The light Libelium API is a very simple API that does not add any additional header and does not handle packet fragmentation nor reassembling. Therefore the maximum 100 bytes of the XBee payload is entirely available. It is very similar to other communication stacks that exist on similar platforms such as Arduino boards. Therefore, it could be considered as the minimal service set and overheads of existing or to come communication stack and library.

Synthetic workload with 802.15.4 Traffic Generator, sending side

A/ Sending with full Libelium API

Figure 15 shows the time spent in the **send() function with the full Libelium API** when the XBee payload is varied from 20 bytes to 100 bytes (application payload with full Libelium API is therefore from 11 to 91 bytes). Packets are sent back-to-back.

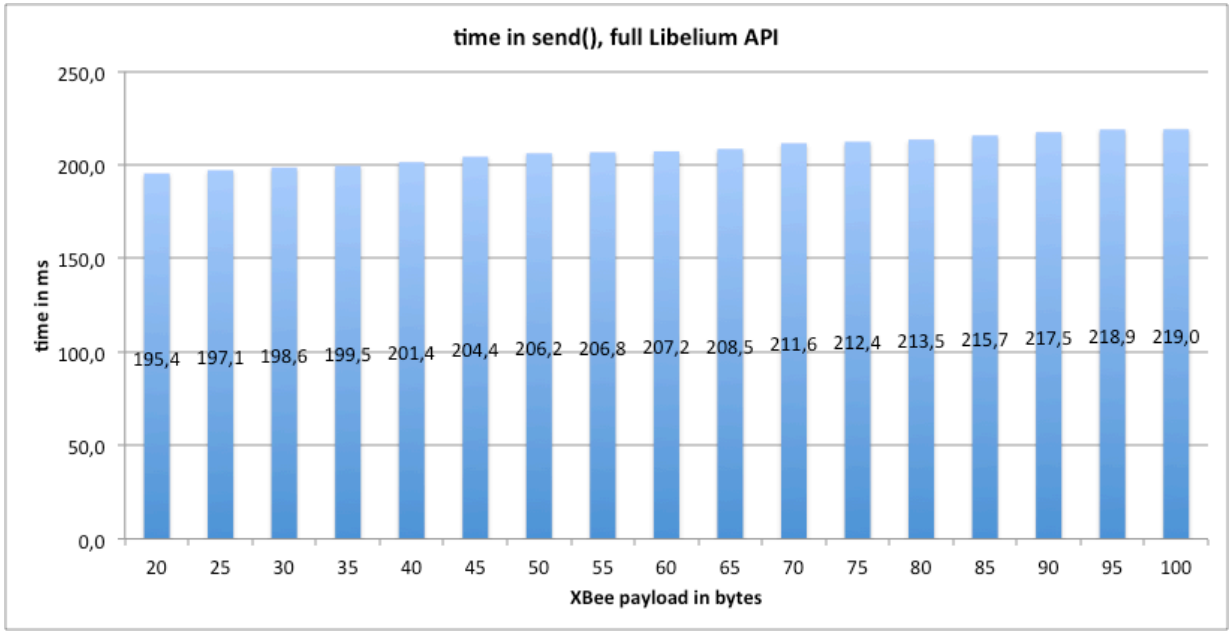


Figure 15: time in send(), full Libelium API

Figure 16 shows the breakout of the time spent in send(), highlighting the various overheads.

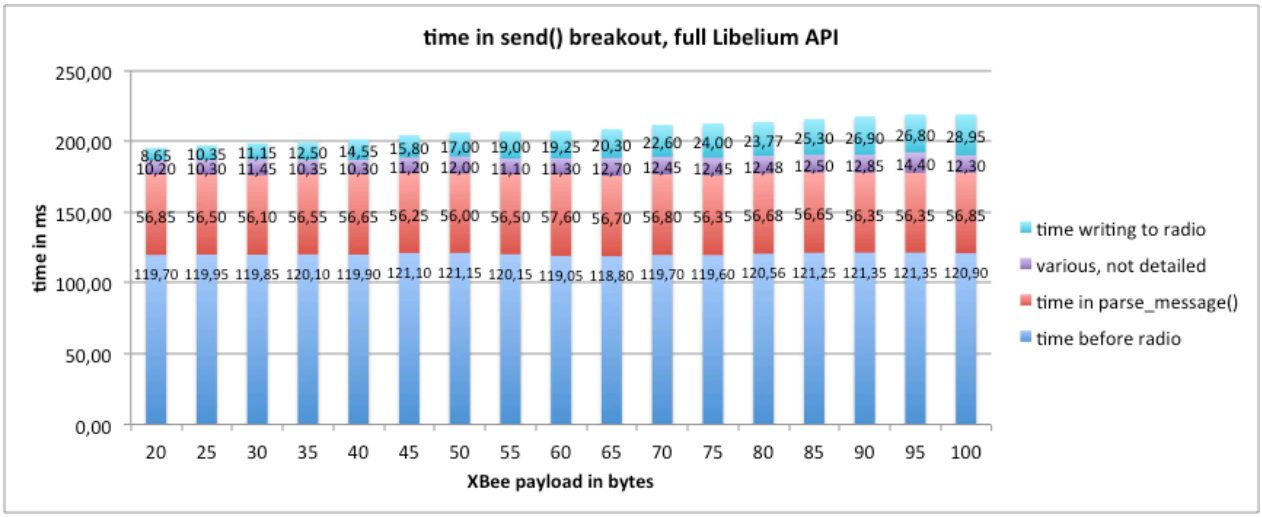


Figure 16: time in send() breakout, full Libelium API

time_before_radio is the amount of time spent in send() just before writing the frame to the radio module. We can see the time spent in parse_message() is quite constant while the time writing to radio directly depends on the payload size, which is a usual behaviour.

We have also a detailed timing of the time spent in parse_message() that differenciates the time waiting for the answer from the radio from the time of the other parse_message() processing tasks. Figure 17 shows the breakout.

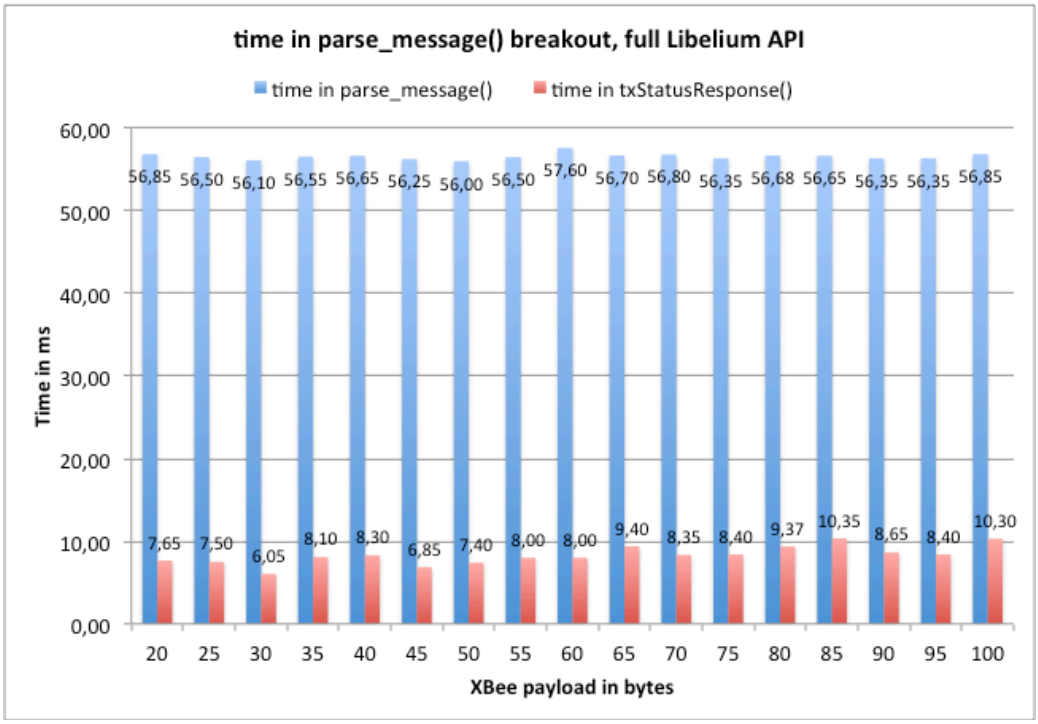


Figure 17: time in parse_message() breakout, full Libelium API

The main results is that the communication API, in full mode, introduces a non-compressible overhead of about 200ms per application packet. Therefore sending packets back to back is limited by this amount of time. We can therefore plot in figure 18 the maximum application level max sending throughput that could be achieved with the full Libelium API.

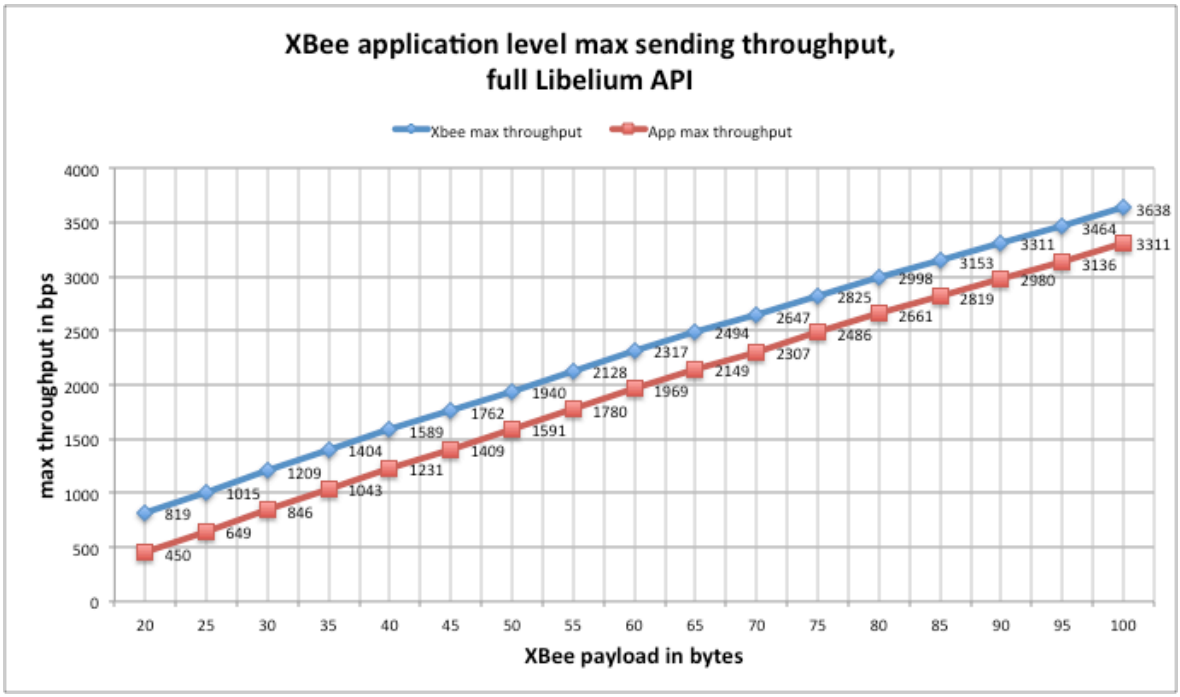


Figure 18: maximum application level max sending throughput, full Libelium API

The blue curve is for the total XBee payload while the red curve is the maximum effective throughput when considering and removing the full Libelium API header overhead of 9 bytes.

Figure 18 showed the maximum throughput derived from the time spend in send() function. Within a realistic application, the time between 2 packet generation is usually a bit higher due

to the additional time required to copy and perform various data manipulation (managing counter, statistic collection, display some data, …) before data could be passed to the send() function. With our traffic generator, with a minimal data handling and I/O display overheads we observed on the WaspMote an additional overhead of a few ms. Figure 19, below, shows the time between 2 packet generation, with the extra overhead of data manipulation, and the time spend in send(), previously shown in figure 15 so that the difference can be highlighted.

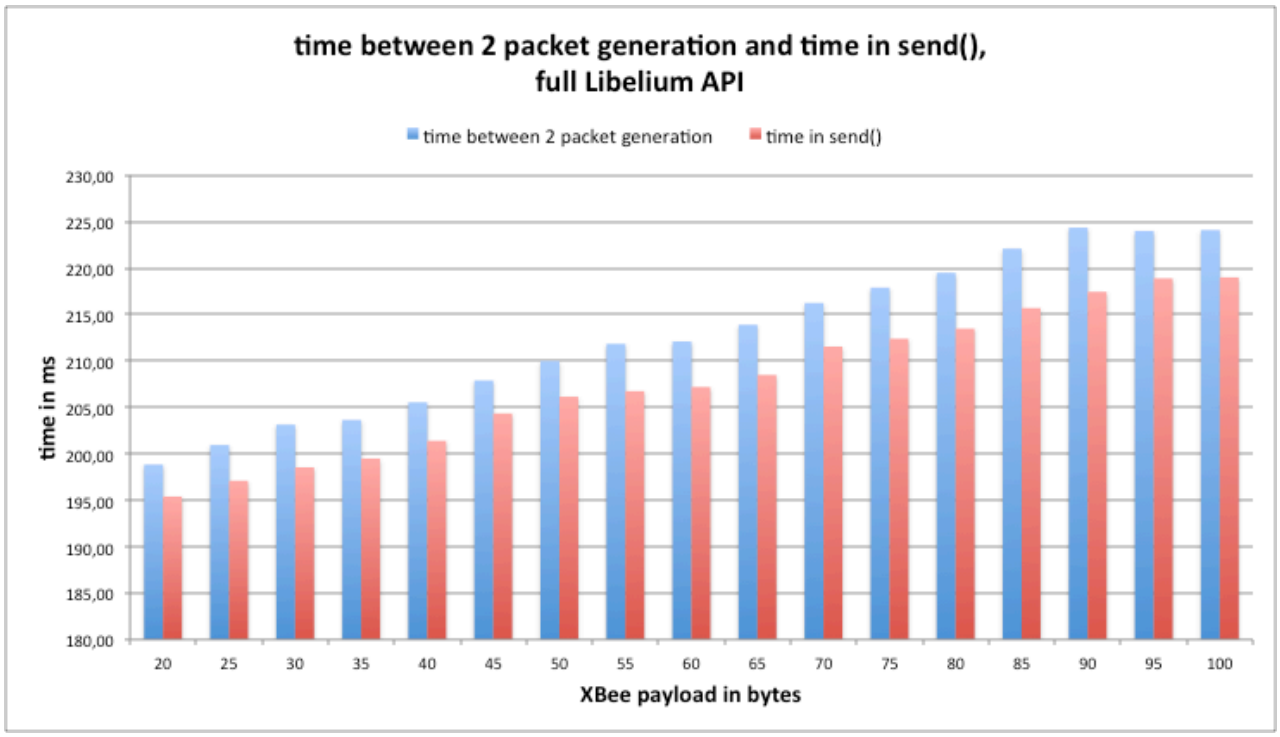


Figure 19: time between 2 packet generation and time in send(), full Libelium API

With this realistic overhead taken into account, figure 20 shows the maximum application level throughput in realistic traffic generation scenario.

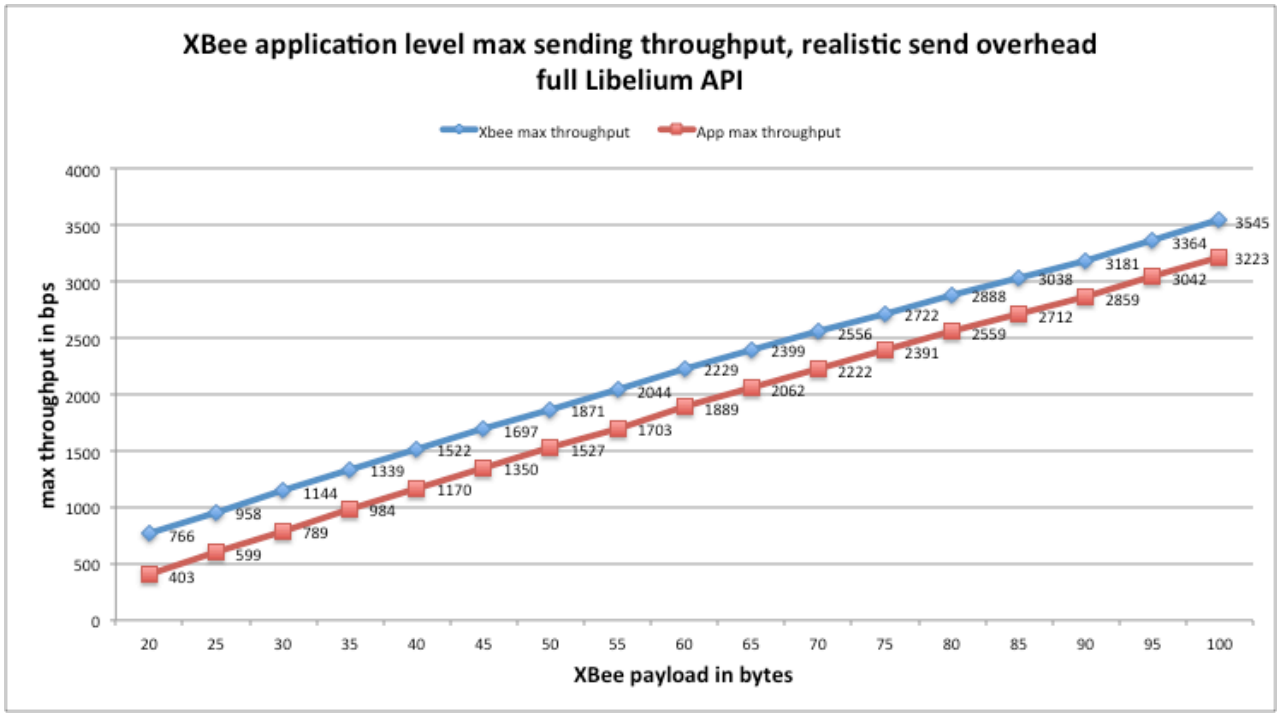


Figure 20: XBee app. level max sending throughout under realistic send overhead, full Libelium API

B/ Long application message support with full Libelium API

The Libelium API provides support for long messages at the application level. As the maximum radio payload is 100 bytes, long messages are fragmented and reassembled by the Libelum API. These features are enabled by the Libelium Application header which is between 5 and 24 bytes as explained previously.

Although the Libelium 802.15.4 programming manual [WASP802] states that messages could be 1500 bytes long at the application level, it appeared that the API only defines a maximum of 200 bytes. In the following tests, we increased this value to 400 bytes to reach 4 fragments.

Figure 21 shows the time spend in send() with long messages usng the full Libelium API features.

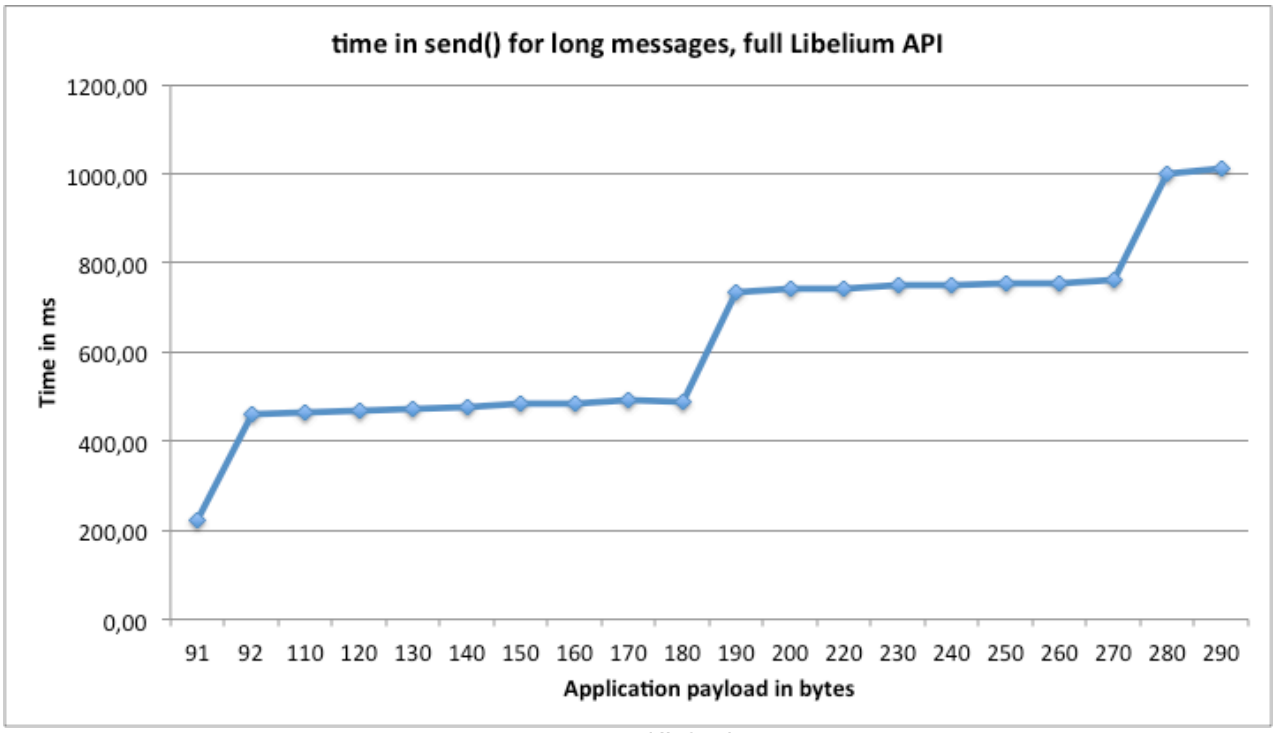


Figure 21: time in send() for long messages

Starting with an application payload of 91 bytes, we have an XBee payload of 100 bytes that fits in a single packet. Then, at 92 bytes for the application payload, we have 2 fragments as clearly shown in the figure where the time spend in send() is more than doubling. Then, for each new fragment we can easily see the impact on the time spend in send(). Although really necessary, it does not appear that using long messages is more efficient compared than sending several smaller packets.

C/ Sending with light Libelium API

Like in the full Libelium API case, figure 22 shows the breakout of the time spent in send() with the light Libelium API, highlighting the various overheads. XBee payload is varied from 10 bytes to 100 bytes. Packets are sent back-to-back. The first noticeable is the total time required for the send() : between 14ms and 44ms instead of between 195ms and 220ms !

If we look at the breakout, we can that the time spent in parse_message() is again quite constant with the payload but, most importantly, much smaller than the full Libelium API case. The time to write to radio is similar to the full Libelium API, which is an expected behavior. Overall, the time spent before sending to radio is largely decreased due to the much smaller complexity of the light Libelium API.

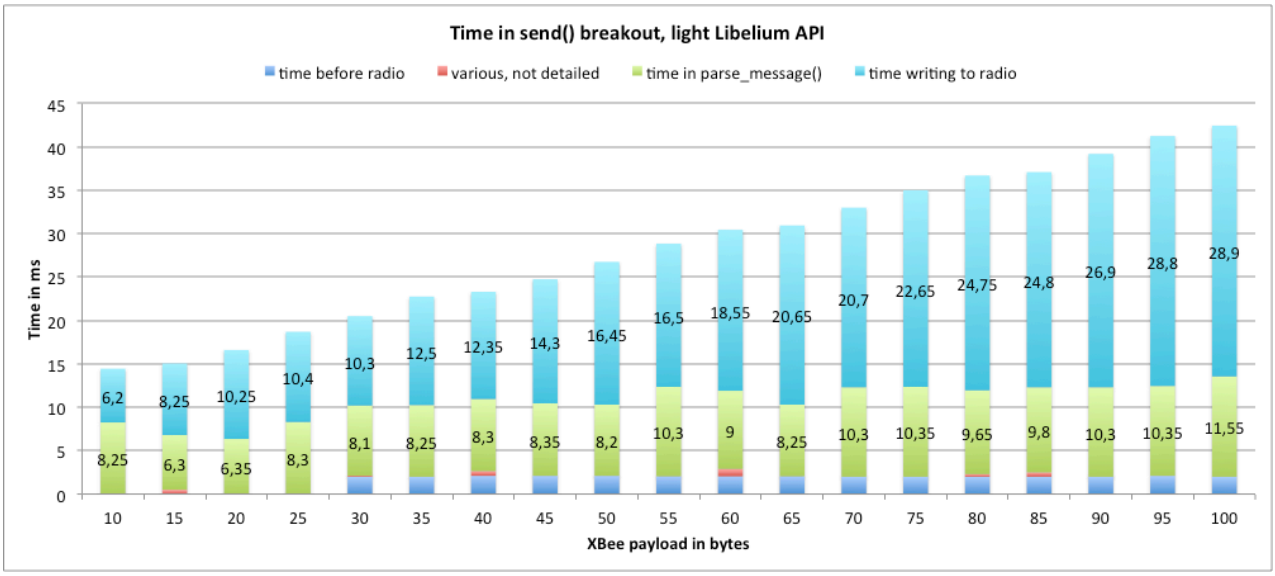


Figure 22: time in send() breakout, light Libelium API

We have also a detailed timing of the time spent in parse_message() that differenciates the time waiting for the answer from the radio from the time of the other parse_message() processing tasks. Figure 23 shows this breakout and we can see that most of the time spent in parse_message is to get and process the answer from the radio module.

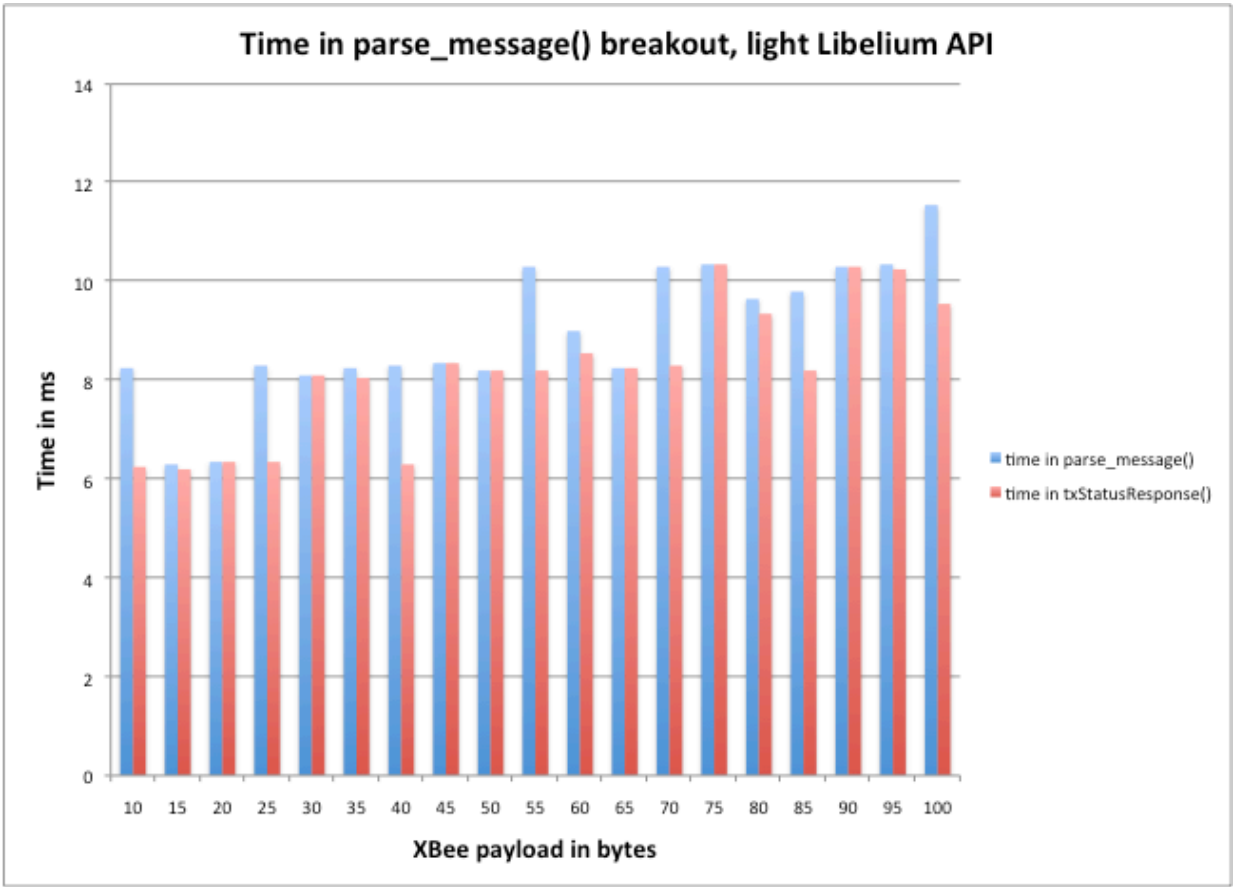


Figure 23: time in parse_message() breakout, light Libelium API

Figure 24 explicitly compares the full Libelium API case to the light Libelium API case in terms of time spent in send().

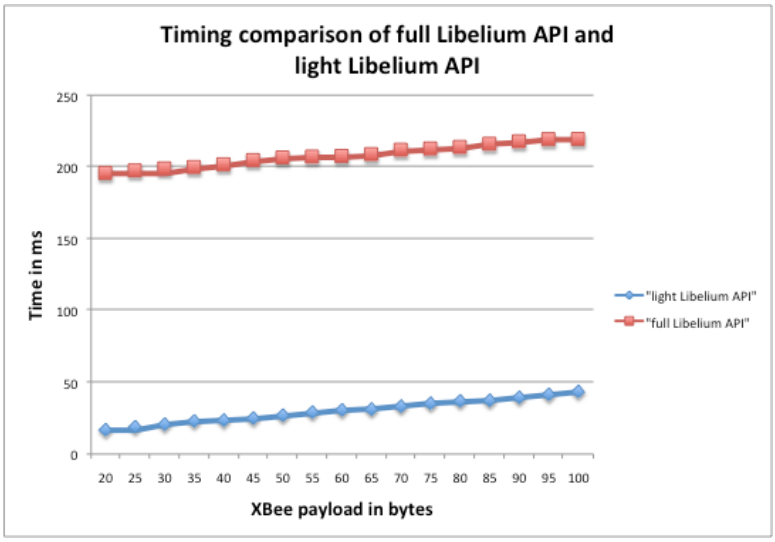


Figure 24: comparison between full Libelium API and light Libelium API

Figure 25 shows the maximum expected throughput at the application level when the light Libelium API is used.

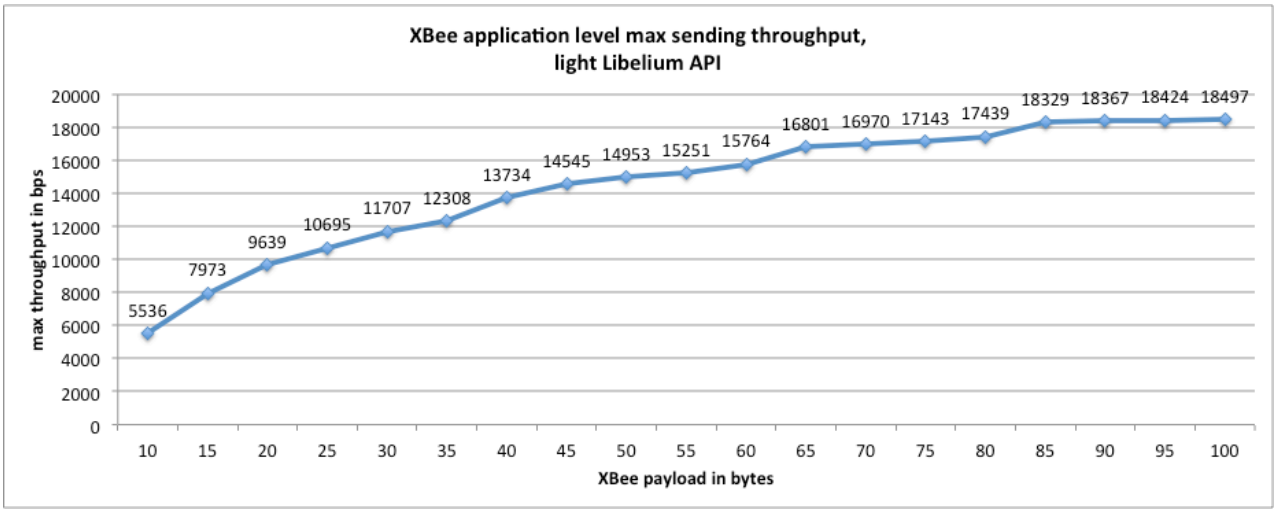


Figure 25: maximum expected throughput with light Libelium API

Figure 26 shows the throughput ratio light Libelium API / full Libelium API.

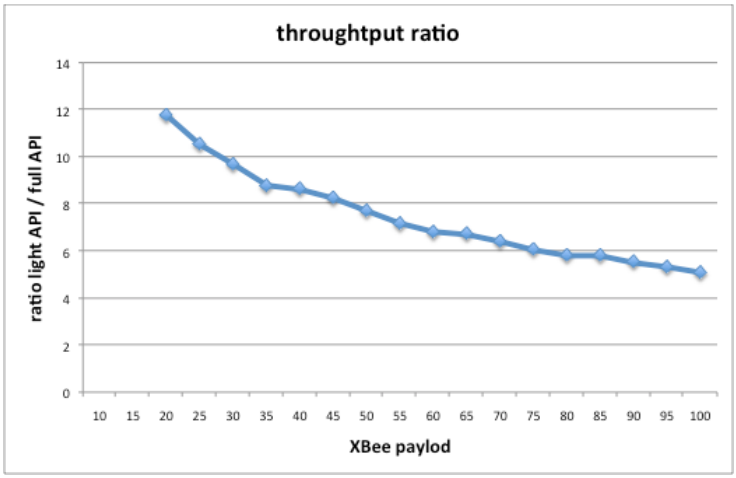


Figure 26: throughput ratio light Libelium API / full Libelium API

Once again, Figure 25 showed the maximum throughput derived from the time spend in send() function. Within a realistic application, the time between 2 packet generation is usually a bit higher due to the additional time required to copy and perform various data manipulation (managing counter, statistic collection, display some data, …) before data could be passed to the send() function. With our traffic generator, with a minimal data handling and I/O display overheads we observed on the WaspMote an additional overhead of a few ms. Figure 27, below, shows the time between 2 packet generation, with the extra overhead of data manipulation, and the time spend in send(), previously shown in figure 22 so that the difference can be highlighted.

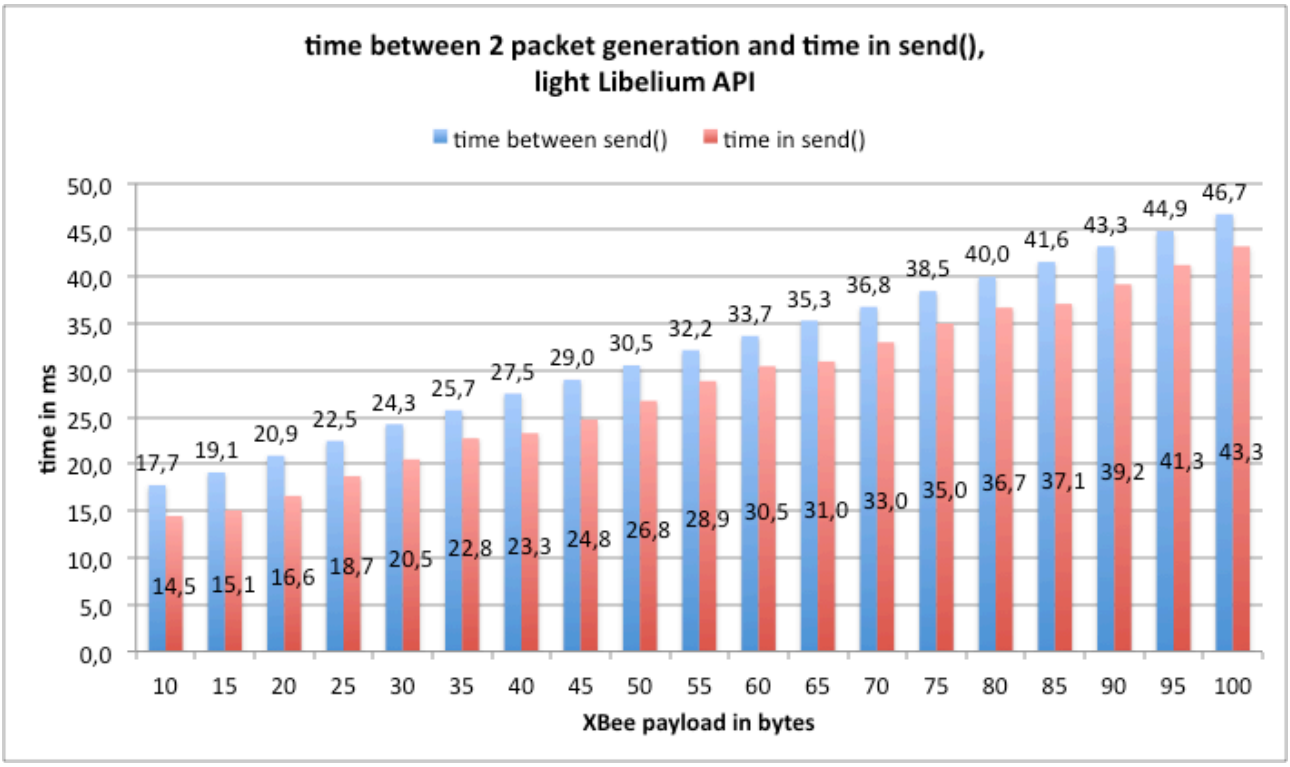


Figure 27: time between 2 packet generation and time in send(), light Libelium API

With this realistic overhead taken into account, figure 28 shows the maximum application level throughput in realistic traffic generation scenario with the light Libelium API.

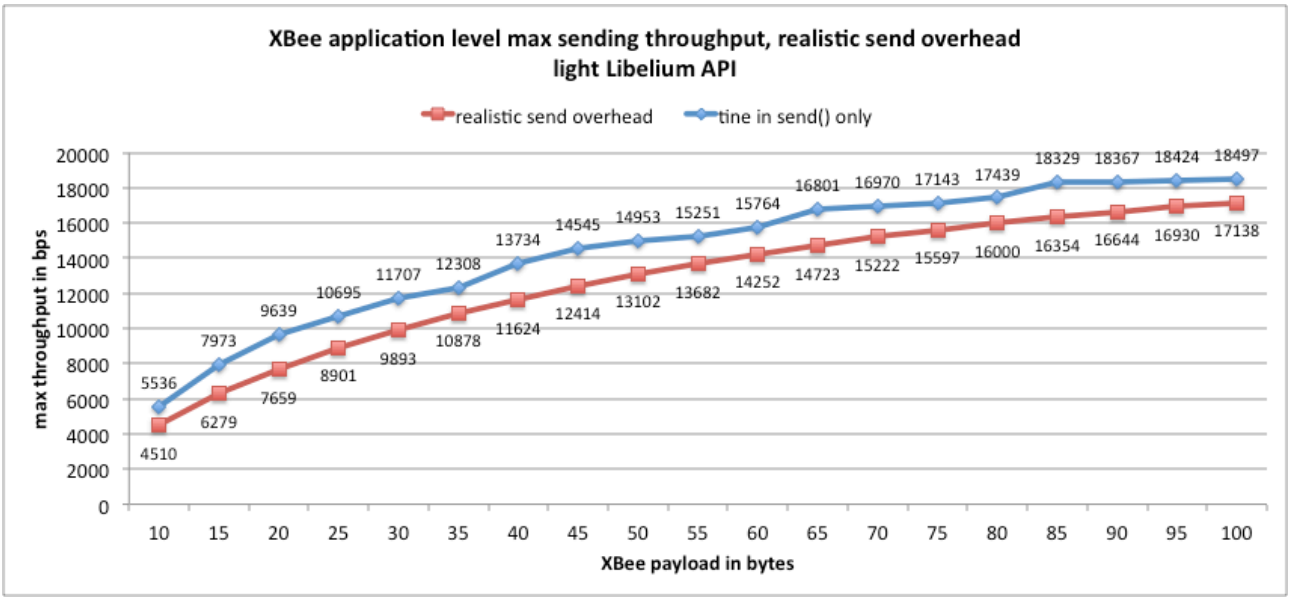


Figure 28: XBee app. level max sending throughout under realistic send overhead, light Libelium API

Synthetic workload with DigiMesh Traffic Generator, sending side

An XBee-PRO module under the DigiMesh firmware adds the DigiMesh multi-hop (AODV based) routing features. For this purpose, the XBee DigiMesh module reserves some bytes in the XBee application payload (which is 100 bytes maximum) to leave a maximum of 73 bytes per packets for the application.

Once again, the Libelium API provides support for DigiMesh modules with a full and light API version. With the full API version, we use the same application header overhead of 9 bytes, thus leaving 73-9=64 bytes for the end application. With the light Libelium API, all 73 bytes are available for the end application.

As opposed to the 802.15.4 where broadcast and unicast traffic are quite similar in performance from the sender perspective, the DigiMesh firmware uses a multiple transmission mechanism for a broadcast packet acting as a substitute to ACK mechanism. The XBee DigiMesh parameter MT (Multi-Transmit, AT-MT) which default value is 3 generates MT+1 transmissions per broadcast packet. Therefore it is expected that a broadcast packet would take longer to transmit, leading to a lower maximum throughput at the sender side. The impact on performances on a multi-hop environment can be very heavy, as relay nodes will also re-transmit MT+1 times the packet, and this will be studied later one in task 3 of the qualification process.

The experimental results presented in this section uses the hardware configuration depicted previously in figure 6, where the XBee DigiMesh module is used to send packets.

A/ With full Libelium API - Broadcast traffic

Figure 29 shows the time in send() breakout for a broadcast packet when the application payload is varied.

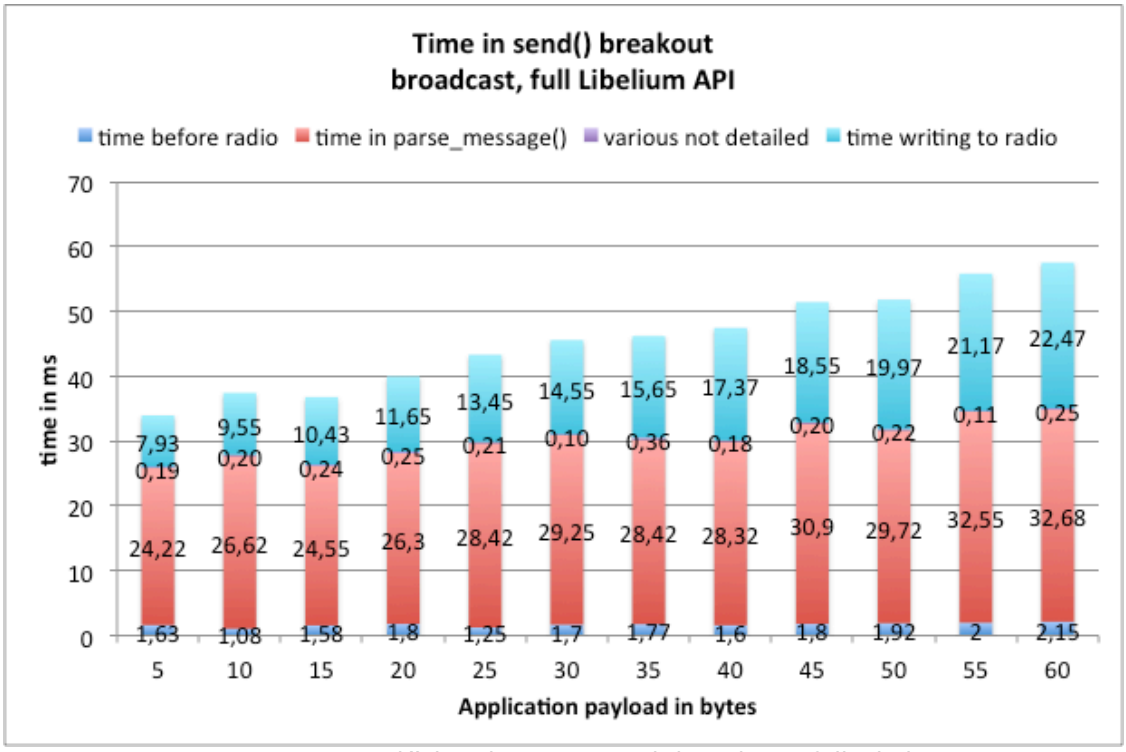


Figure 29: time in send() breakout, DigiMesh broadcast, full Libelium API

Recall that as the DigiMesh firmware adds at the MAC level an overhead of 27 bytes, only 73 bytes is available. As the full Libelium API header size is 9 bytes, the breakout only shows the timing for application payload up tp 60 bytes (60+9+27=96 bytes) as 65 bytes would make the packet longer than 100 bytes.

Figure 30 shows a detailed breakout of time in send() that shows the various stages of the sending process.

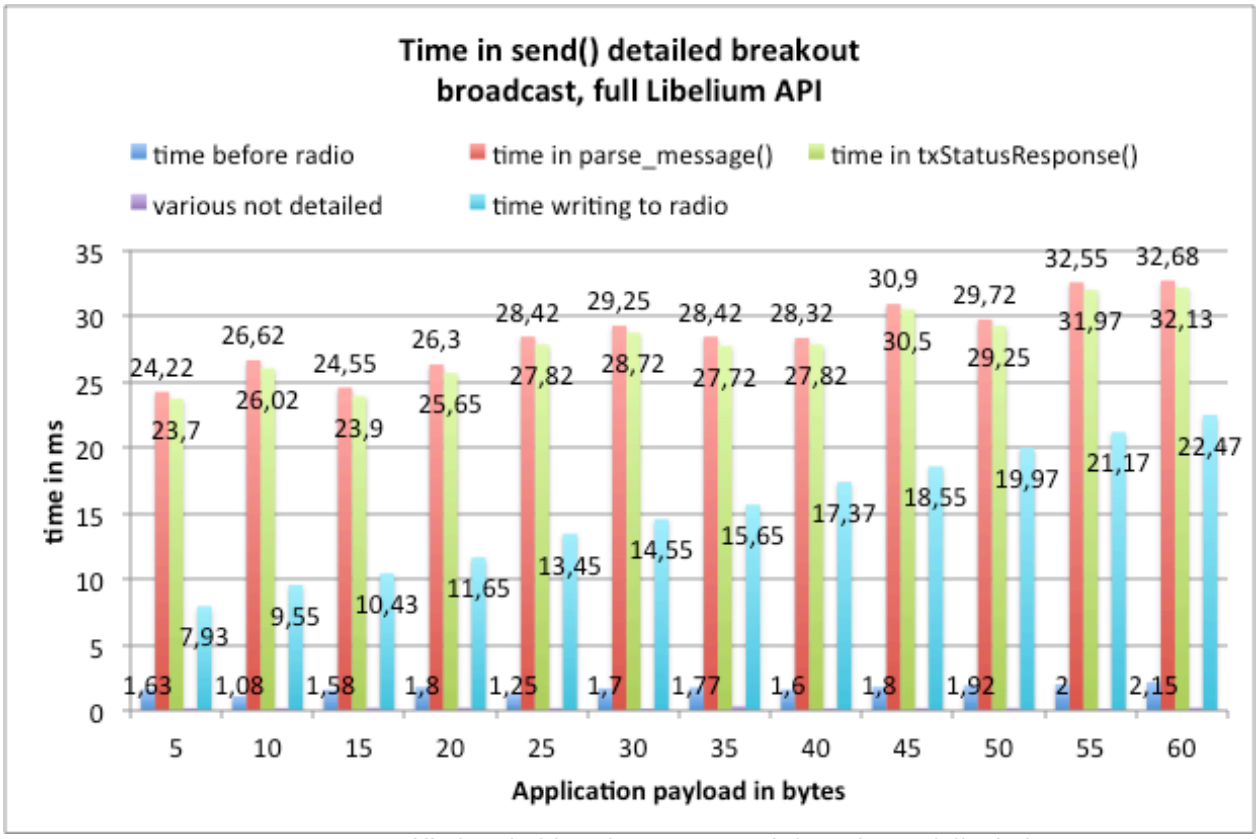


Figure 30: time in send() detailed breakout, DigiMesh broadcast, full Libelium API

We can see that the time to write to radio is very similar to what have been measured previously: for instance 60 bytes of application payload is 69 bytes to be transmitted to the radio. As the DigiMesh header is added at the MAC level, it is not transmitted from the application layer, thus the 69 bytes instead of the 60+9+27 bytes. If we look back at figures 16 or 22, we can see that for 70 bytes, the time writing to radio is very similar which is an expected result.

However, if we look at figure 31 that shows the time between 2 packet generation and the time in send(), we can see that compared to the 802.15.4 full Libelium API case, the time between 2 packet generation is much lower: we decrease from around 200ms to around 50ms with the full Libelium API. In figure 31, we show application payloads up to 90 bytes which give a 126 bytes packet that will be fragmented by the full Libelium API. We can see the increase in time at 65 bytes for the application payload. Once again, with 1 fragmentation, the 802.15.4 full Libelium API roughly needed more than 400ms (see figure 21) while the DigiMesh full Libelium needs about 150ms.

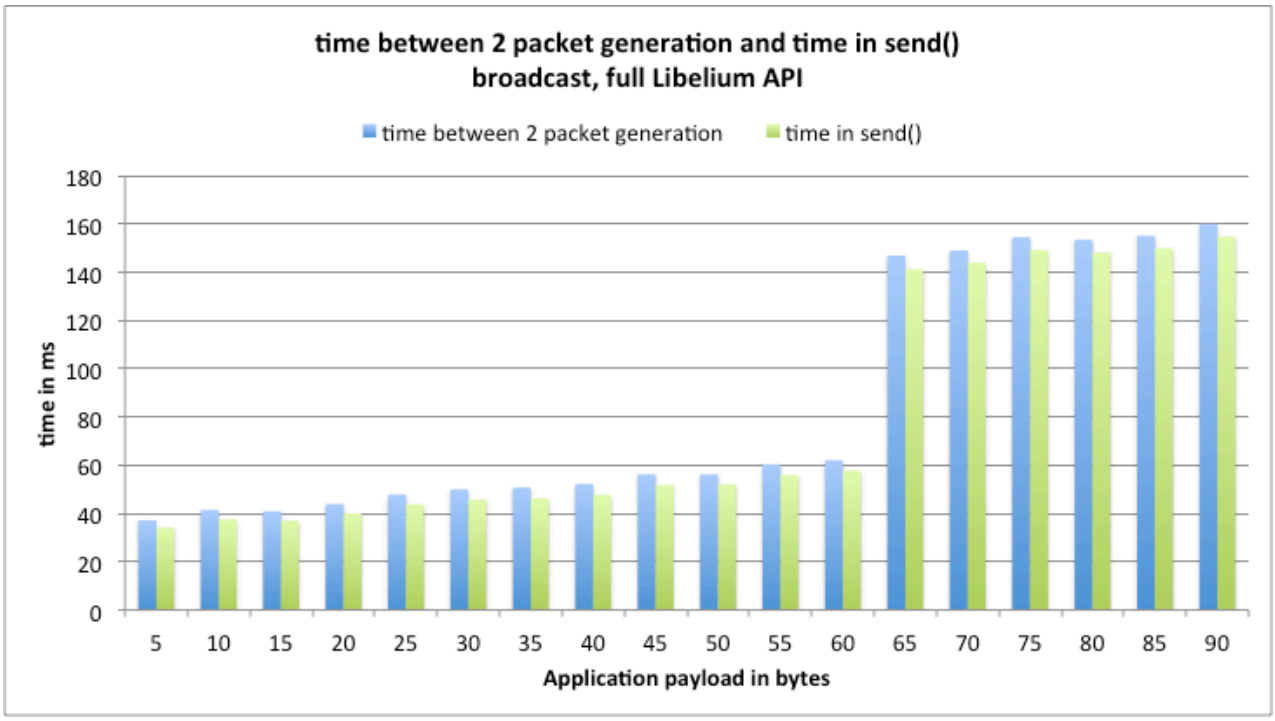


Figure 31: time between 2 packet generation and time in send(), DigiMesh broadcast, full Libelium API

B/ With full Libelium API - Unicast traffic

With unicast packet, the multi-transmit mechanism is disabled and we could expect a smaller time spent in send() function. Figure 32 shows the time in send() breakout for the unicast case, always with the full Libelium API. Figure 33 shows the detailed time in send() breakout.

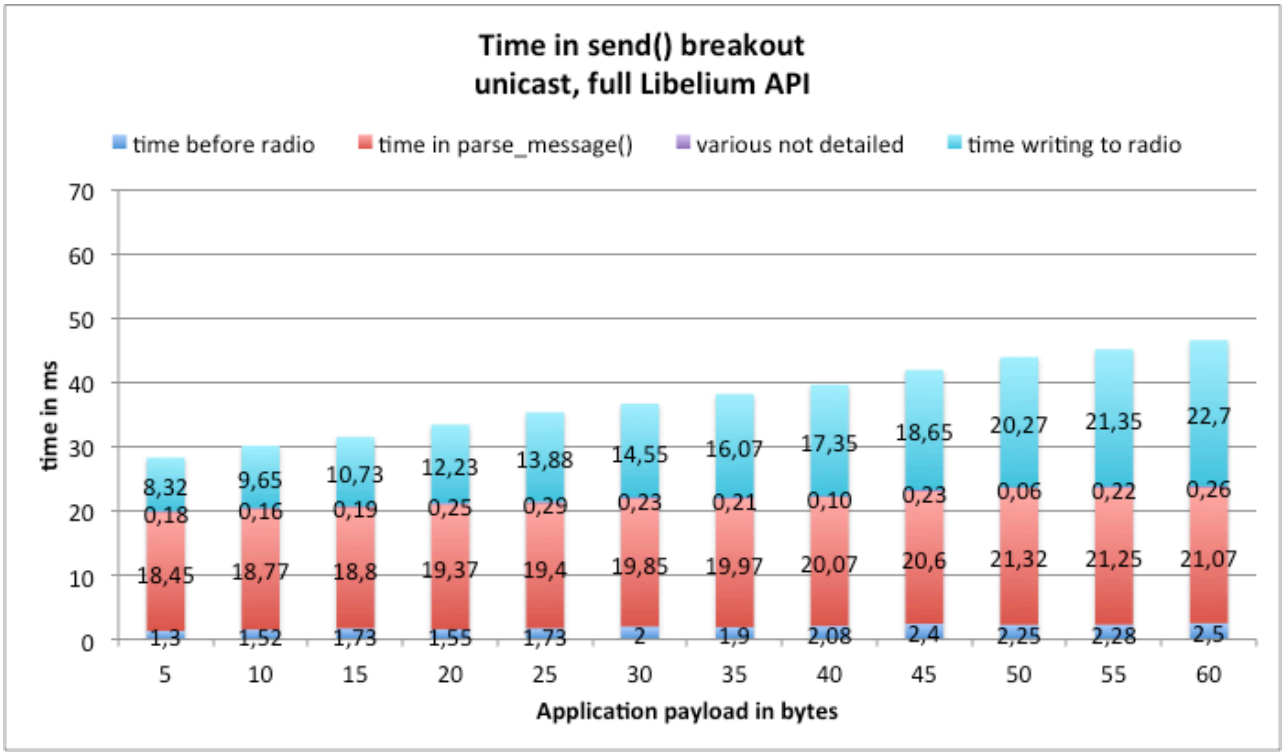


Figure 32: time in send() breakout, DigiMesh unicast, full Libelium API

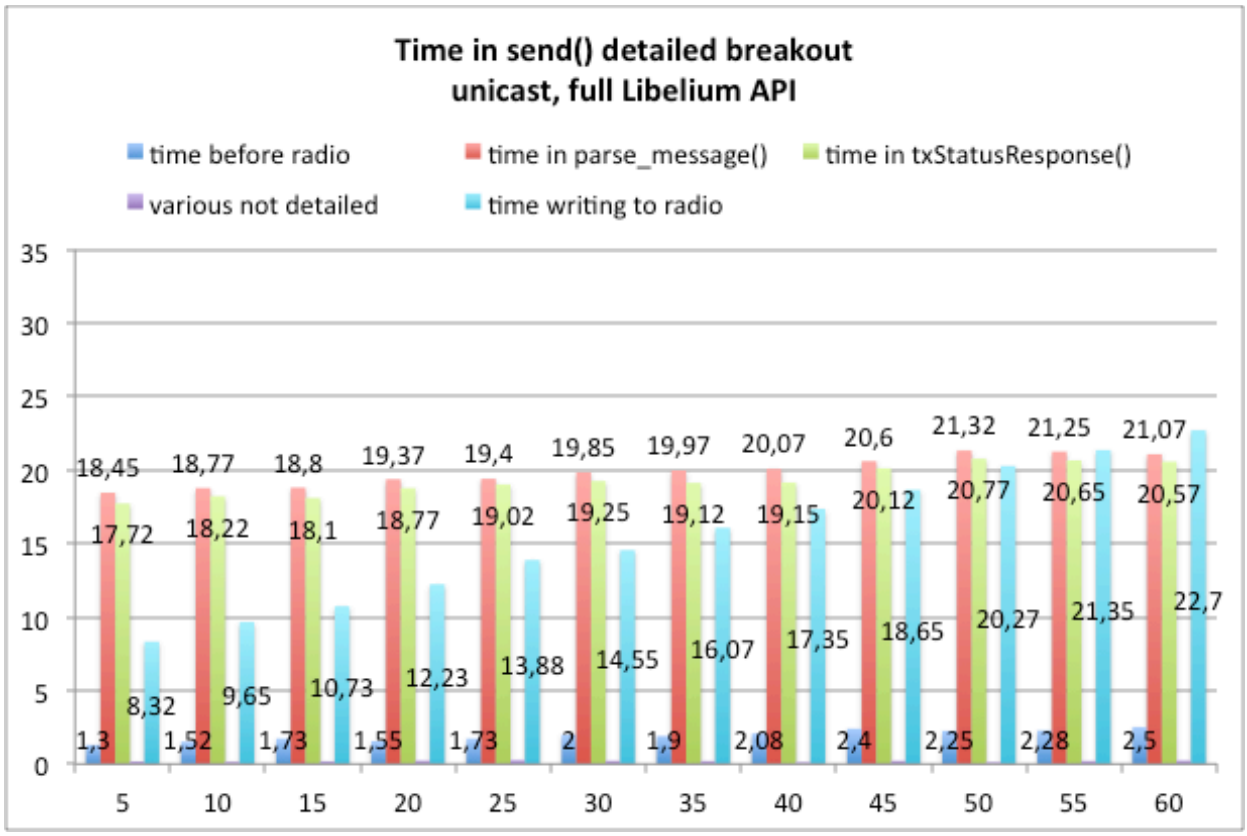


Figure 33: time in send() detailed breakout, DigiMesh unicast, full Libelium API

What can be seen is that the time spent in parse_message (mainly spent for getting the answer from the XBee module) is greatly decreased because mutlti-transmit is not used for unicast traffic. The time between 2 packet generation can therefore be smaller as shown in figure 34.

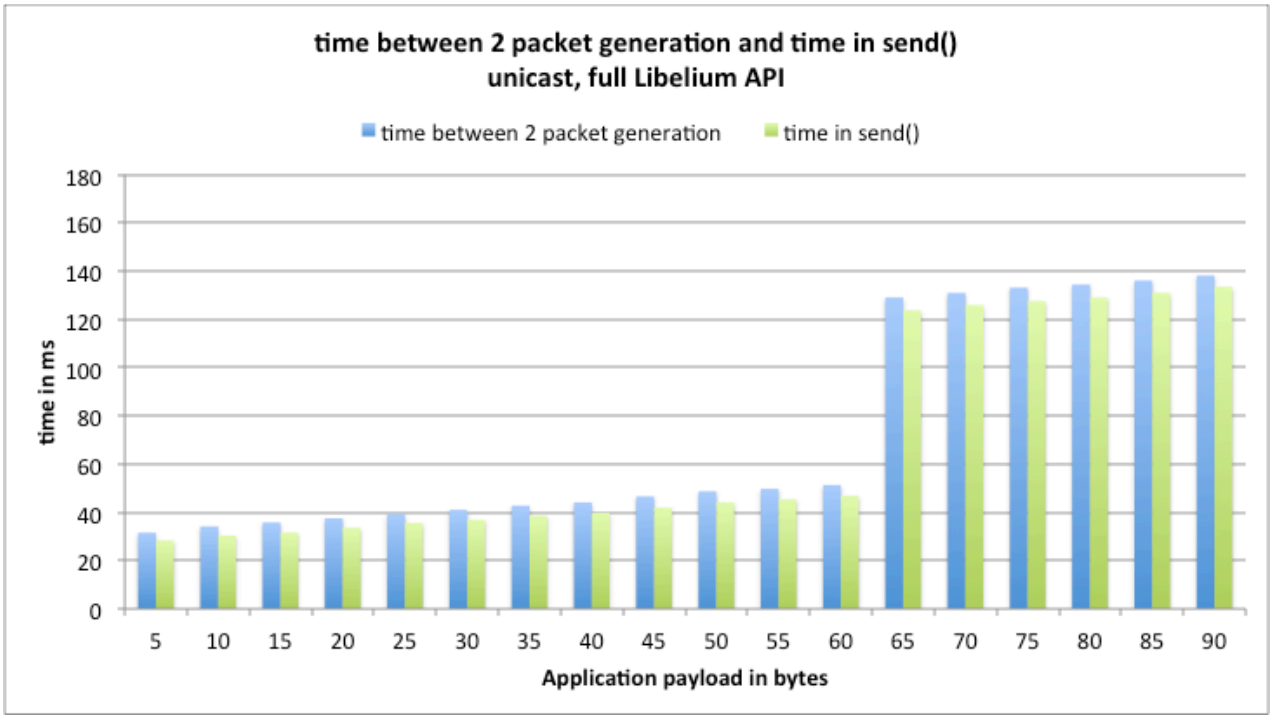


Figure 34: time between 2 packet generation and time in send(), DigiMesh unicast, full Libelium API

C/ With light Libelium API - Broadcast traffic and Unicast traffic

We show in figure 35 and 36 the time in send() breakout for the broadcast and unicast traffic respectively, under the light Libelium API. Here, there is no fragmentation support therefore the maximum application payload is 70 bytes (70+27=97 bytes for the XBee payload).

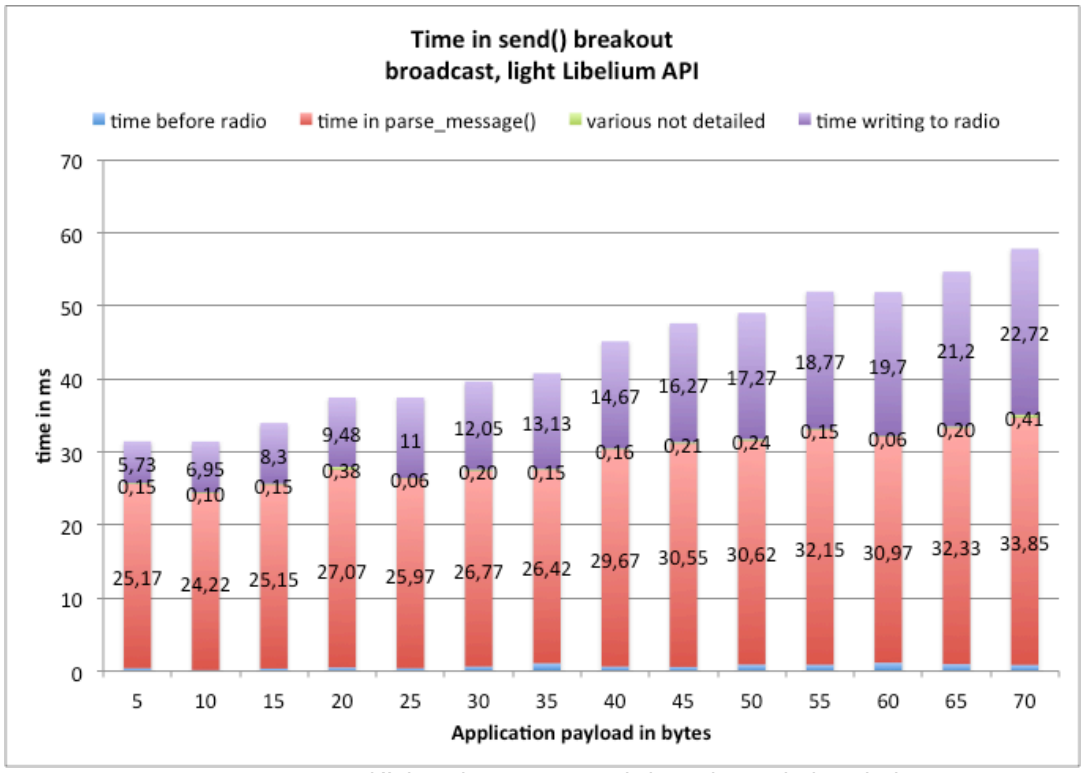


Figure 35: time in send() breakout, DigiMesh broadcast, light Libelium API

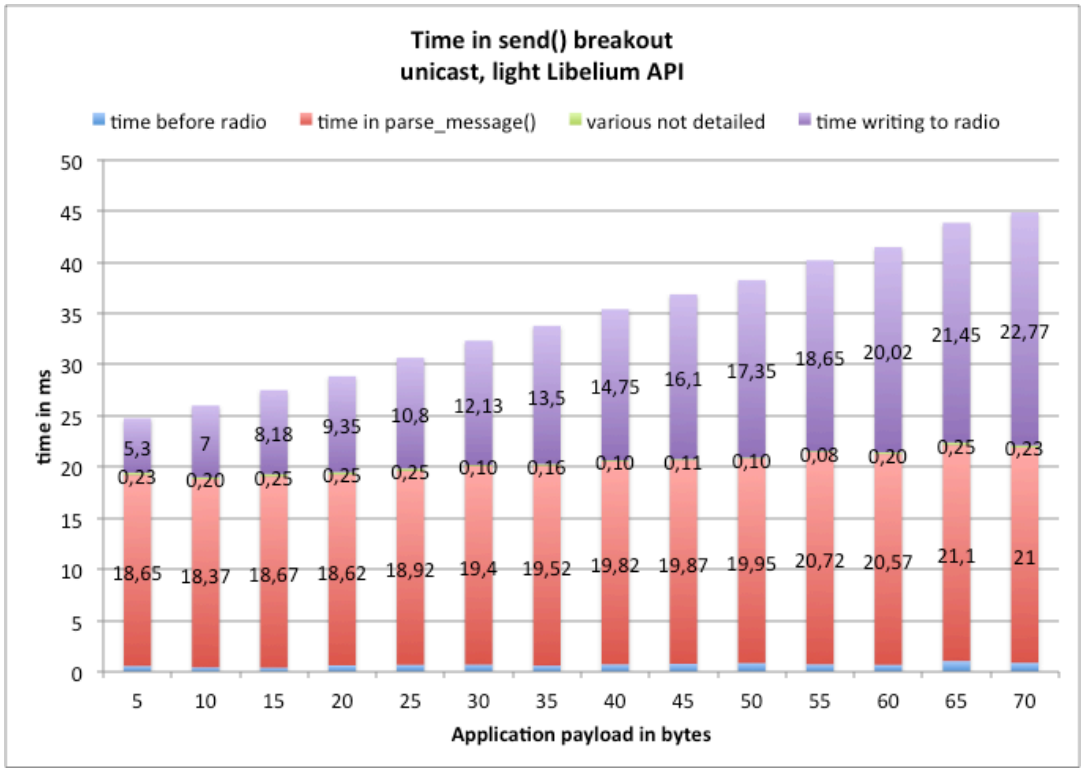


Figure 36: time in send() breakout, DigiMesh unicast, light Libelium API

The results are not very different from the full Libelium API and we can conclude from these experimentations that using the full or the light Libelium API have less performance impacts with the DigiMesh firmware than with the 802.15.4 module. However, if we look at the application level max sending throughput, shown in figure 37, we can see the DigiMesh header size of 27 bytes is very penalizing compared to the 802.15.4 version with light Libelium API.

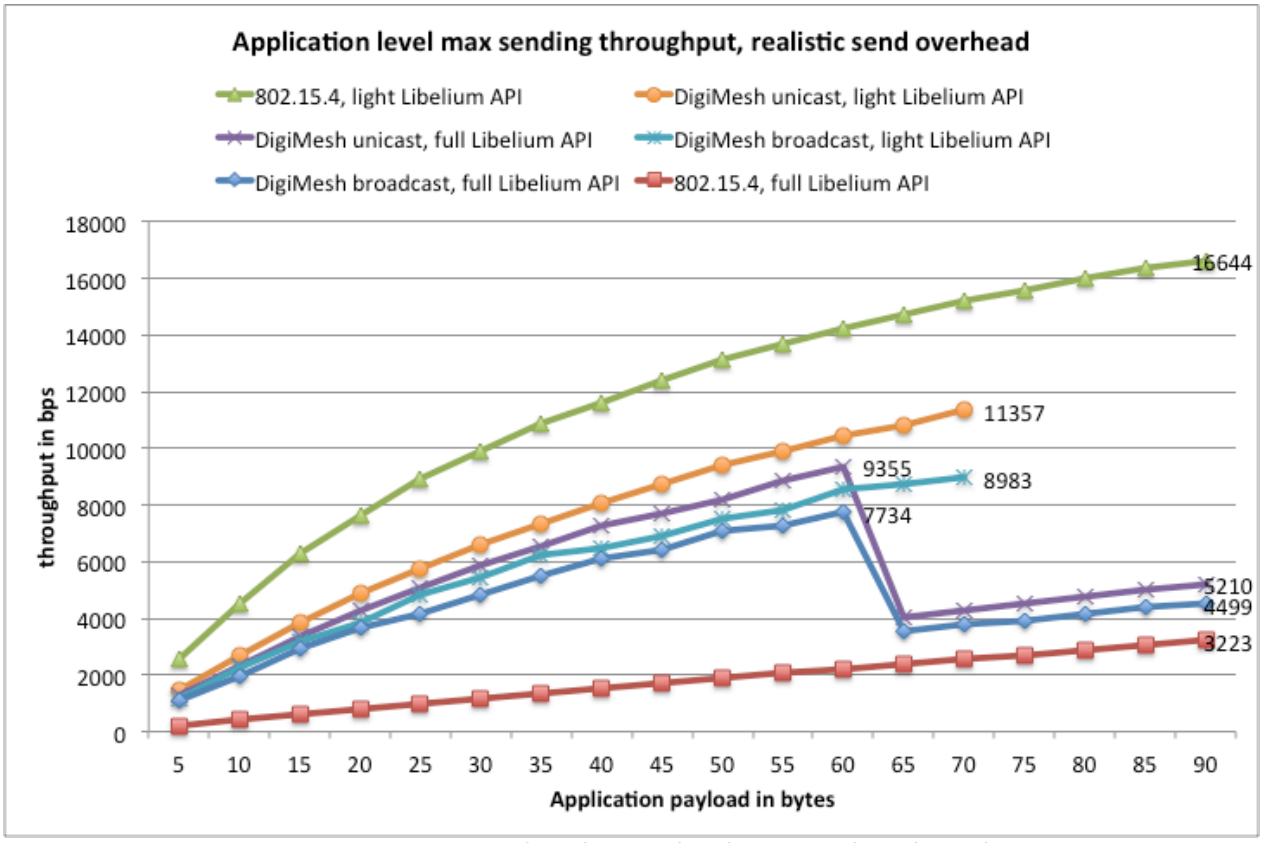


Figure 37: comparison of application level max sending throughput

The table below summarizes the application level max sending throughput for the DigiMesh firmware.

	Comparison of application level max sending throughput DigiMesh full/light Libelium API					
	full Libelium API			light Libelium API		
AppPayload	Xbee payload	broadcast	unicast	Xbee payload	broadcast	unicast
5	41	1076	1267	32	1147	1440
10	46	1932	2341	37	2253	2679
15	51	2939	3352	42	3167	3822
20	56	3645	4266	47	3868	4887
25	61	4184	5076	52	4818	5757
30	66	4803	5854	57	5437	6574
35	71	5518	6547	62	6231	7336
40	76	6130	7258	67	6494	8062
45	81	6402	7725	72	6911	8732
50	86	7114	8212	77	7486	9396
55	91	7285	8851	82	7793	9872
60	96	7734	9355	87	8524	10473
65	101	3540	4029	92	8763	10802
70	106	3758	4275	97	8983	11357
75	111	3882	4505	102		
80	116	4170	4759	107		
85	121	4383	4997	112		
90	126	4499	5210	117		

Explaining differences of full Libelium API with 802.15.4 and DigiMesh

If we compare figure 16 to figure 32, we can see that using DigiMesh with the full Libelium API is much faster that with 802.15.4 with the same full Libelium API, especially with a very small time before radio for the DigiMesh case. As the hardwares are similar, the differences come from how the full Libelium API handles the DigiMesh module compared to the 802.15.4 module.

First, all the previous tests, 802.15.4 and DigiMesh, were performed using transmit requests with a 64-bit destination address (the so-called Medium Access Control address). With the 802.15.4 module, it is possible to use 16-bit destination address and the 802.15.4 module from Digi does differenciate whether sending and reception are done with 64-bit or with 16-bit addresses.

Each 802.15.4 module has a 16-bit source address that, if not set to 0xFFFF, will trigger at the reception side a reception event for a 16-bit address. The full Libelium API avoids 16-bit address reception event when a 64-bit destination address is used by (a) saving the current values of the 16-bit source address of the sender, (b) setting this 16-bit source address to 0xFFFF and (c) by restoring the saved values. All these steps are performed in the send() and take a lot of time. DigiMesh modules do not have 16-bit address features and only use 64-bit addresses, therefore there is no need to perform the additional steps described above.

We will show in this section the performance of the full Libelium API when instead of using a 64-bit destination address, we use the 16-bit addressing scheme. In this way, the full Libelium API does not perform the additional steps required for a 64-bit destination address[1]. Figure 38 shows the time in send() breakout for the 802.15.4 module with the full Libelium API when the 16-bit addressing scheme is used.

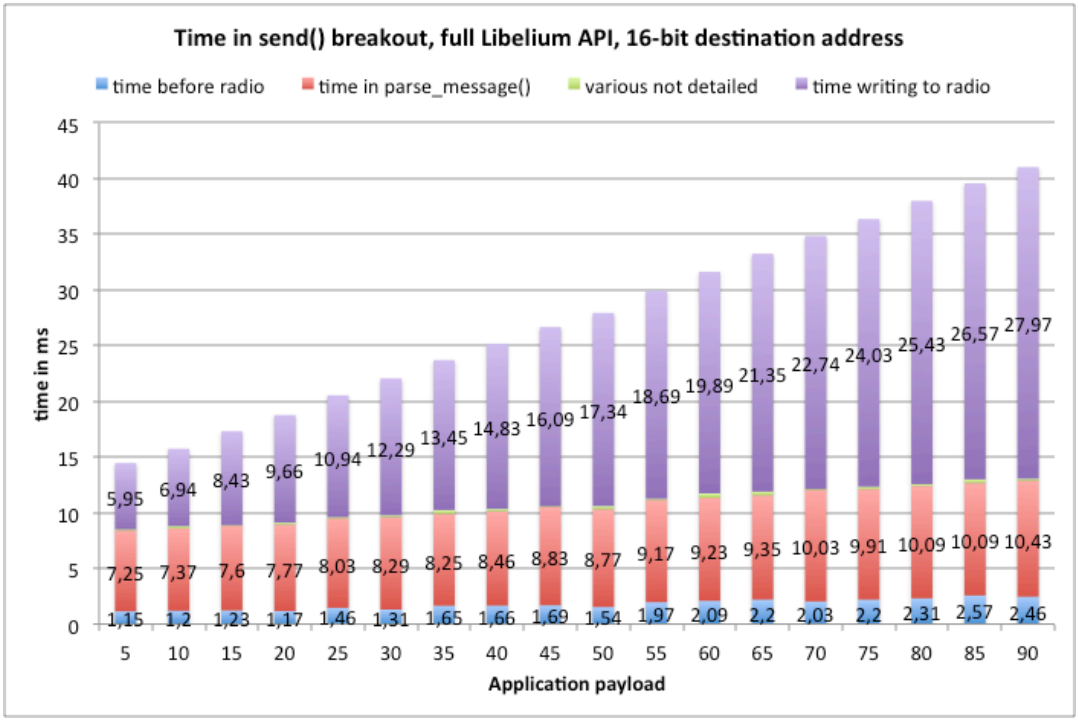


Figure 38: time in send() breakout, full Libelium API, 16-bit destination address

[1] *The objective of the qualification process is not to change the existing programming API but to qualify its performances. This is the reason we did not change the 64-bit address sending procedure of the full Libelium API but use instead the 16-bit destination address send version that is provided by the full Libelium API.*

We can clearly see the reduction in the time before radio and the time in parse_message (which we can not explain at the current stage of the qualification process). The final result is that the full Libelium API with 16-bit destination address can have the same level of performance than the light Libelium API version which does not care about 16-bit address. Figure 39 compares the time between 2 packet generation and the time in send() for both the full Libelium API 16-bit address and the light Libelium API.

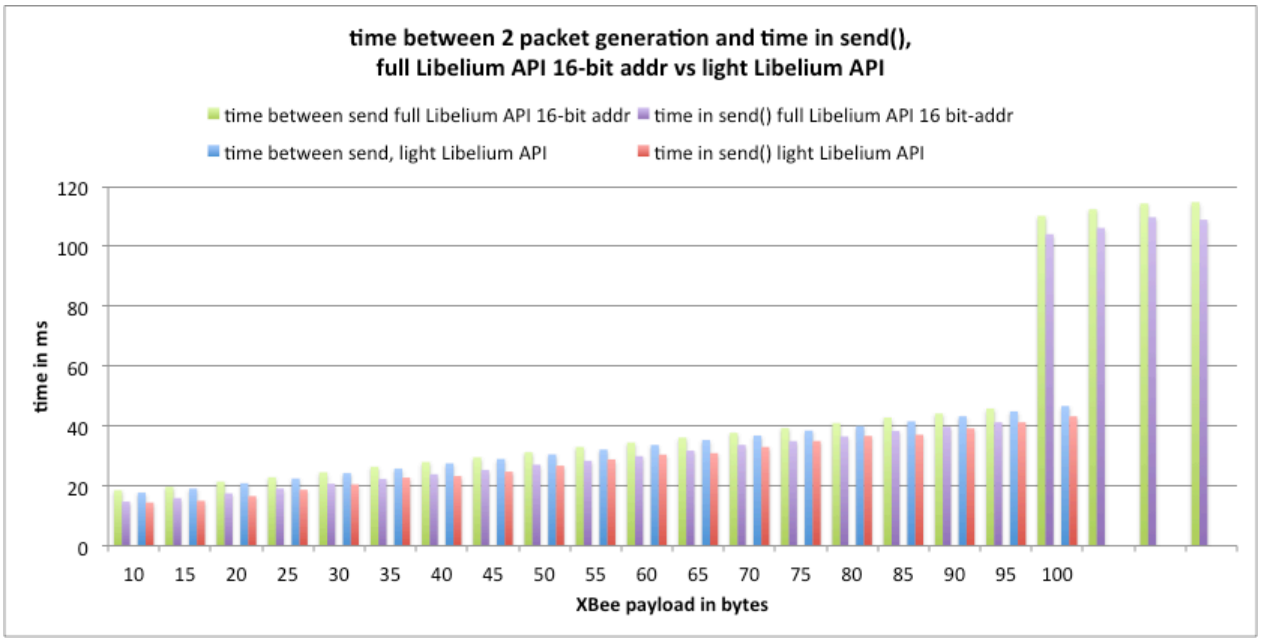


Figure 39: time between 2 packet generation and time in send() for full Libelium API 16-bit address and light Libelium API

We can see that the timing are very close when the full Libelium API does not have to perform additional steps for saving, setting and restoring the 16-bit source address. One advantage of the full Libelium API in 16-bit addressing scheme is to provide the fragmentation and reassembly feature for long message. However, as depicted in figure 39, the cost is quite high.

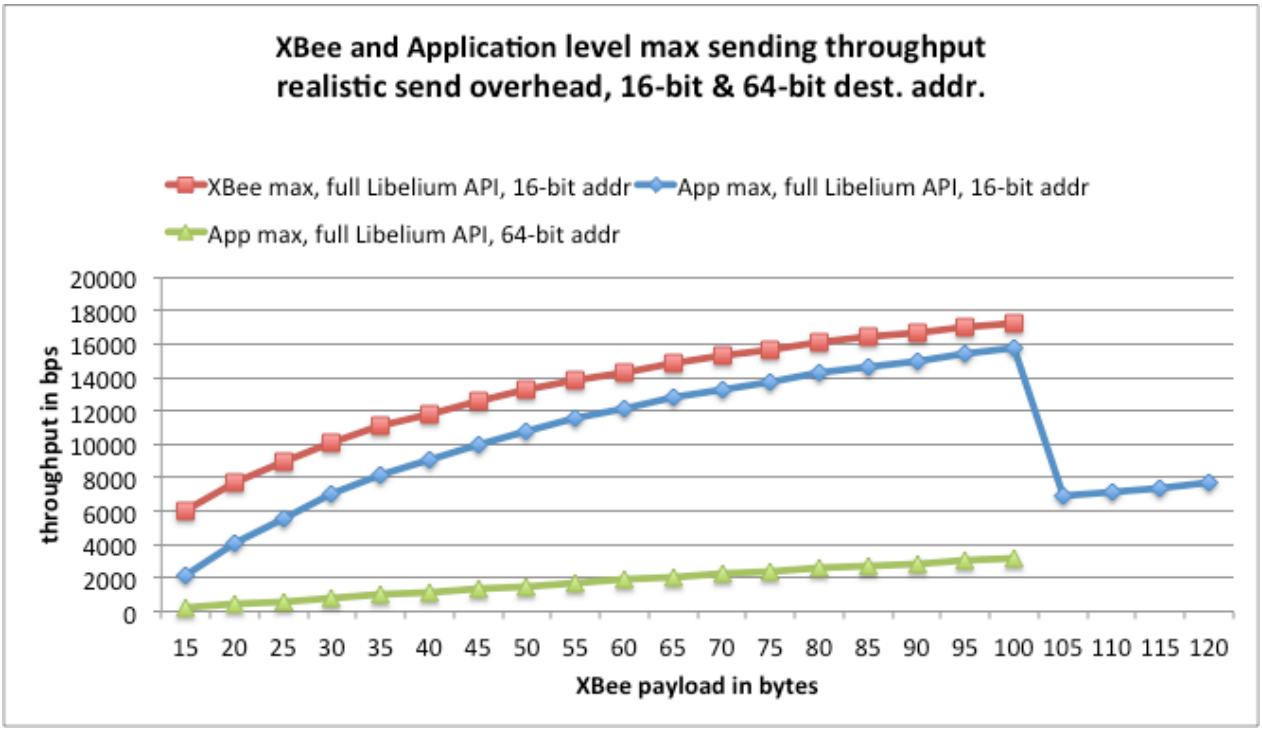


Figure 40: XBee and app. level max. sending throughput, full Libelium 16-bit addr. and light Libelium

Figure 40 shows the Xbee and application level maximum sending throughput under realistic sending overhead (taking the time between 2 packet generation and not the time to return from send()). The red curves (XBee max throughput) takes into account the additional header of the full Libelium API. If we compare this red curve to the red curve of figure 28 that showed the XBee level throughput (which is also the application level throughput) with the ligh Libelium API, we can see that the 2 curves are very close.

The blue curve in figure 40 shows the application level maximum sending throughput. For comparison purpose, we replot with the green curve the application level maximum sending throughput with the full Libelium API in 64-bit address mode.

Although it is beyond the scope of this document, is it possible to improve the performances of the 64-bit address full Libelium API by setting with an AT command the 16-bit source address to 0xFFFF (ATMY FFFF) and remove the additional steps in the full Libelium API for 64-bit address.

Performance of the SmartSantander communication library

On the SmartSantander test-bed there are operational constraints that limit what an experimenter could do. The reason is to ensure that an IoT node will not become inaccessible for the SmartSantander management system.

Therefore the SmartSantander research team provides to the end-users a specific communication library that offers 2 communication classes, `smartComm802` and `smartCommDM` to handle the 802.15.4 and the DigiMesh interface respectively. A class function `sendPacket()` sends the data. For the DigiMesh interface, a specific function `sendLogDM()` allows the end-user to send logs to the associated Meshlium gateway. According to the SmartSantander developpers, `sendLogDM()` is a wrapper for `smartCommDM.sendPackets()`.

Figure 40a shows the "time between send()" for the SmartSantander communication library, using the `smartComm802.sendPacket()` function to have access to the 802.15.4 interface. We compare with the light Libelium API performance shown previously in Figure 27.

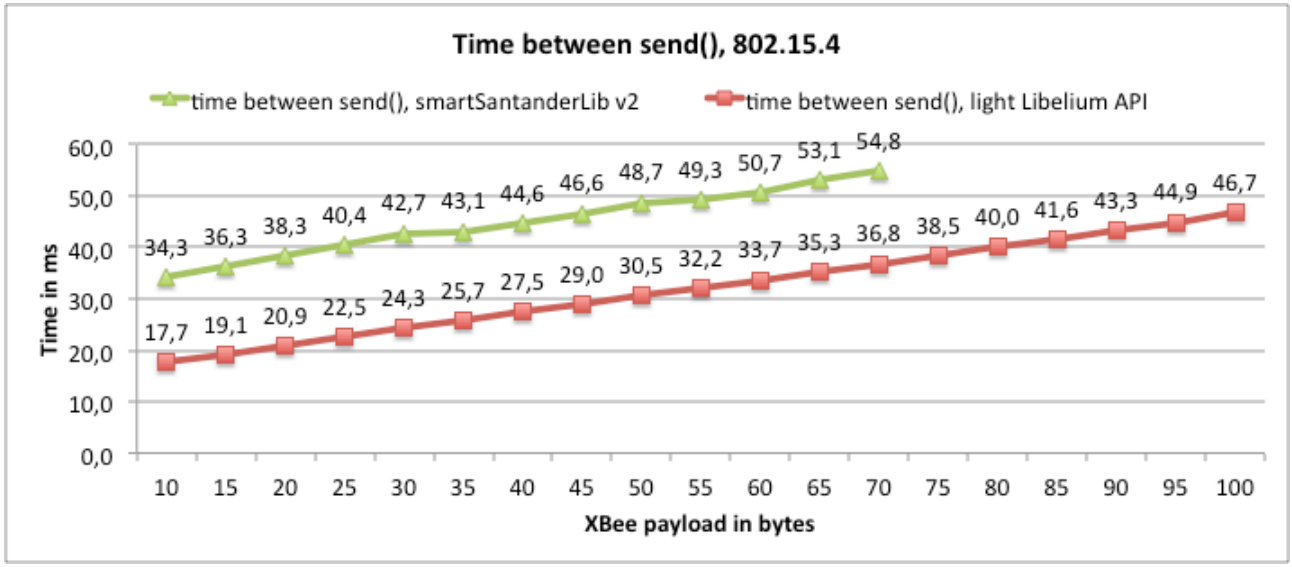


Figure 40a: time between send() for the SmartSantander library and light Libelium API

The `sendPacket()` function takes a bit more time to return compared to the light Libelium API. Also, the payload is currently limited to 70 bytes, even on the 802.15.4 interface, to share

the same limitations than the DigiMesh interface. However, as this behaviour will not necessarily be true for future versions of the library, the important information is that compared to the light Libelium API, the `smartComm802.sendPacket()` function adds about 17ms of overhead. Figure 40b shows the maximum sending throughout derived from these "time between send()" measures.

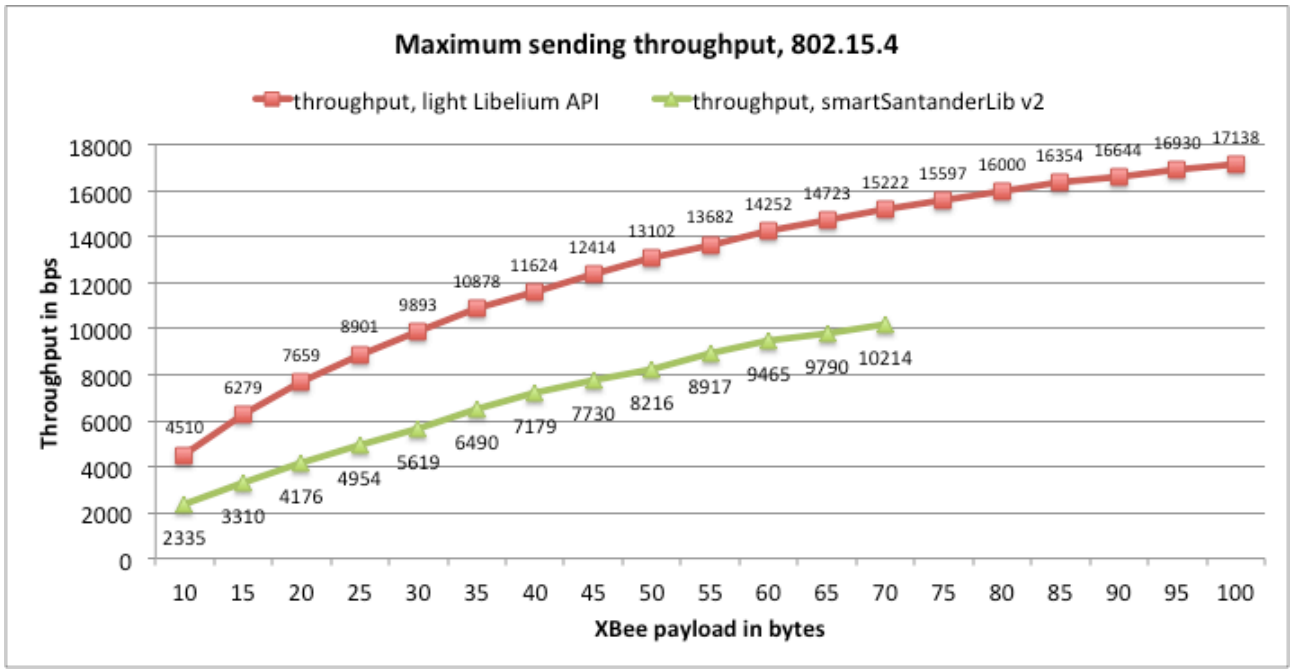


Figure 40b: Maximum sending throughput: SmartSantander library and light Libelium API

It is possible to use the DigiMesh interface to send data and logs (text data). We will differentiate between broadcast and unicast traffic as explained previously. Figure 40c shows the "time between send()" when using the `smartCommDM.sendPackets()` function for broadcast traffic. Performance of unicast traffic is shown in Figure 40d.

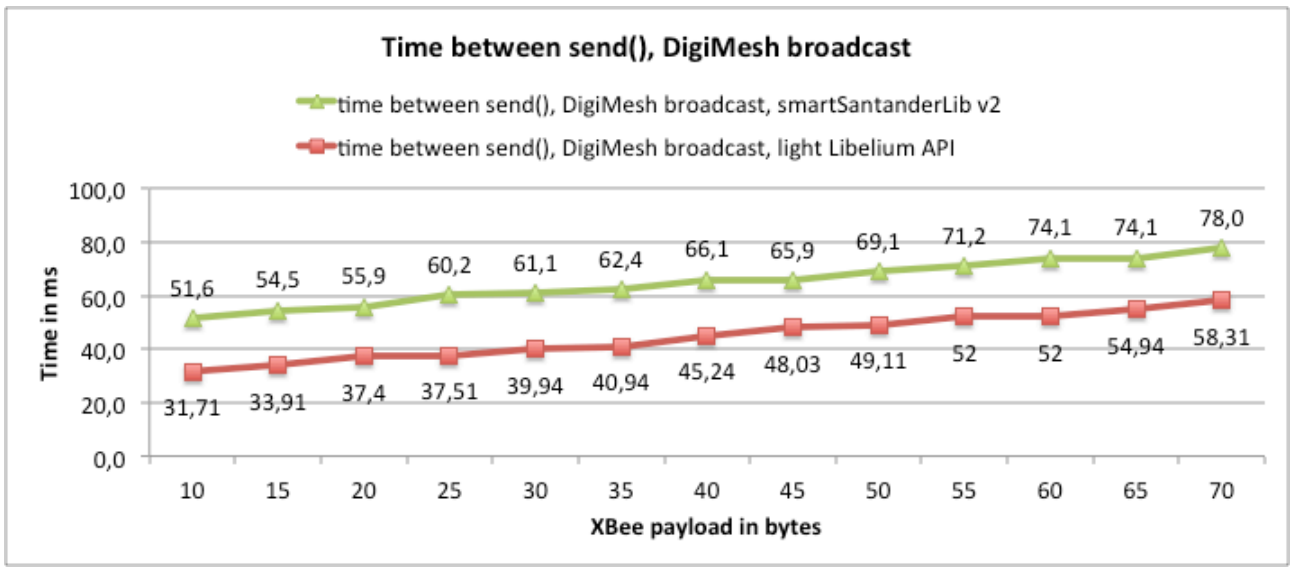


Figure 40c: Time between send(), DigiMesh broadcast: SmartSantander library and light Libelium API

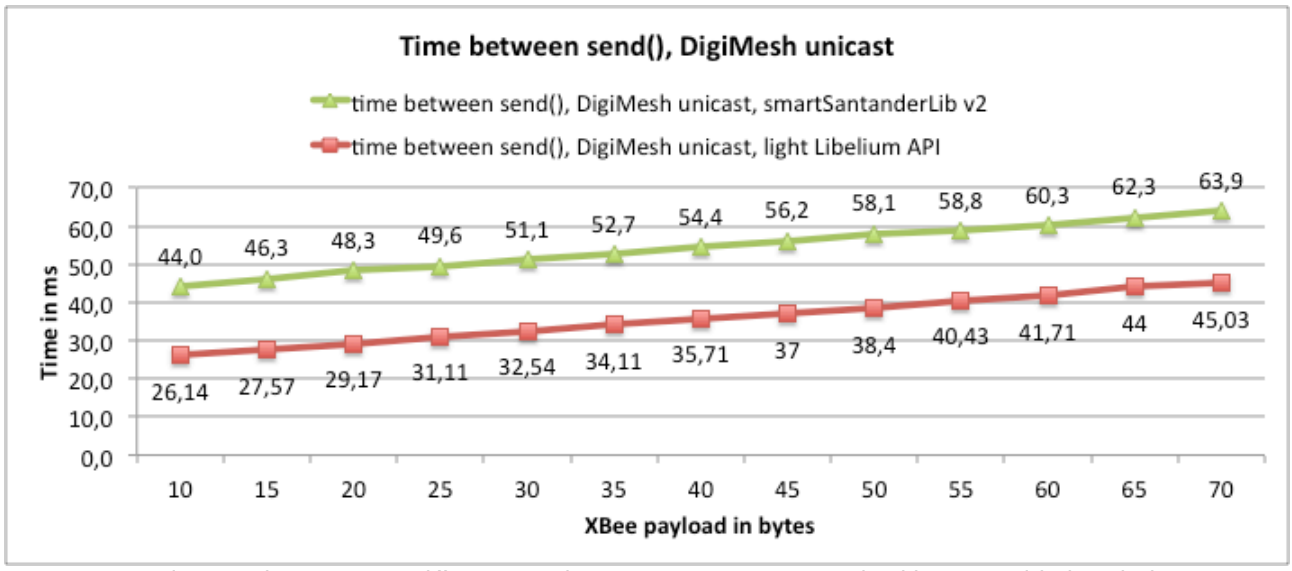


Figure 40d: Time between send(), DigiMesh unicast: SmartSantander library and light Libelium API

Figure 40e shows and summarizes the maximum sending throughput for the DigiMesh interface when using the SmartSantander library, and provides a comparison with the light Libelium API.

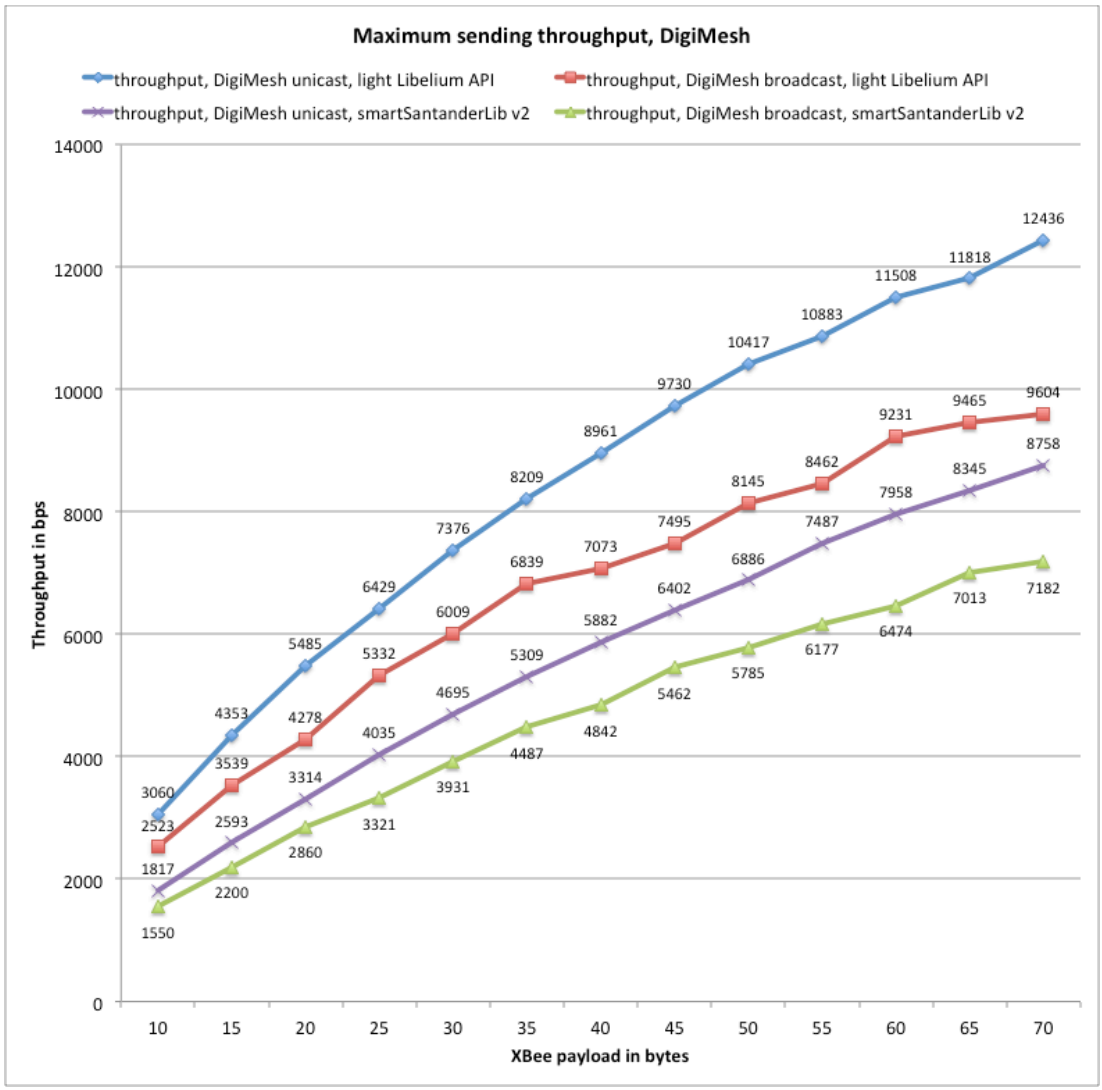


Figure 40e: Maximum sending throughput, DigiMesh: SmartSantander library and light Libelium API

Synthetic workload with 802.15.4 Traffic Generator, receiving side

At the receiving side, one can also choose to have either the full Libelium reception API, when the sender is using the full Libelum API of course, or a lighter version that directly read from the radio module byte by byte. Actually, when the payload is smaller than 100 bytes, meaning that there have not been packet fragmentations at the sending side, both approaches are quite similar in terms of overheads. We will look at the particular case of packet fragmentation and reassembly later on.

The following experimentation tests in this section have been performed with the sender using either the full Libelium API or the light Libelium API and the receiver using exclusively the light version of the reception API in order to get access to all the bytes of the payload.

The receiving throughput is directly linked to the sending throughout. We will show the additional overhead added by the reception and will measure the receiving throughput at the application level.

A/ Receiver throughput when sending with full Libelium API

Figure 41 shows the measured and estimated mean inter-arrival time of packets at the receiving side. Here packets are sent back-to-back with the full Libelium API at the sender side.

We measured this inter-arrival time for XBee payload size of 30 bytes, 60 bytes and 100 bytes. Then, we use a linear estimation method to deduce the mean inter-arrival time for the other payload size. Figure 41 explicitly shows the 3 measured inter-arrival time. The purpose of doing so is to validate the linear behaviour because we will later on measure the real reception throughput and will compare it to the estimated one.

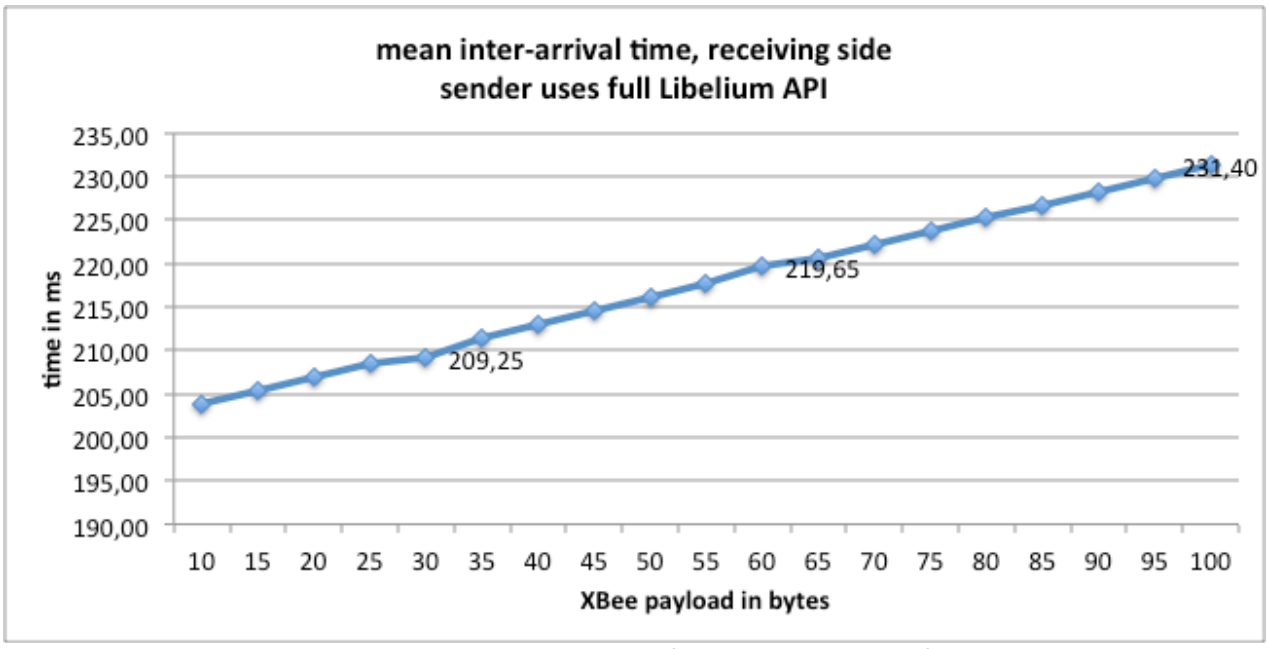


Figure 41: mean inter-arrival time at receiving side

Given the mean inter-arrival time at the receiving side, we can determine the maximum expected reception throughput with the following relation:

$$TH_{AppRcv}\ (bps) = Payload(bytes) * 8 * (1/\ mean\_inter\_arrival\_time)$$

Figure 42 shows the results of the maximum estimated reception throughput (computed with the relation above) when some inter-arrival time are estimated ("estimated" curve) and compares it with the real maximum reception throughput.

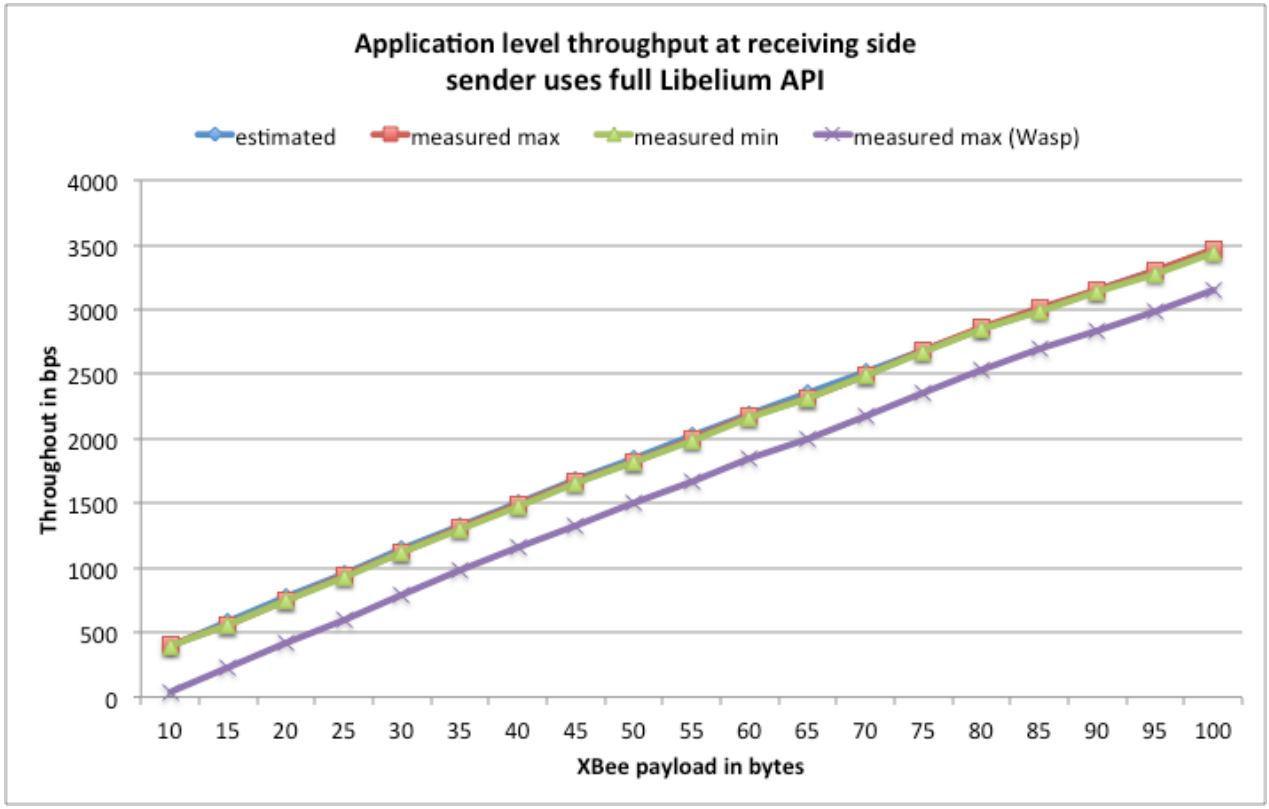


Figure 42: Application level throughput at receiving side, sender uses full Libelium API

The real reception throughput is computed every 10 packets and we take 10 measures to extract the maximum and the minimum reception throughput observed ("measured max" and "measured min" curves respectively).

The "measured max (Wasp)" curve is the application level throughput when removing the additional 9 bytes per packets introduced at the sending side by the full Libelium API.

Figure 43 compares the sender throughput previously shown in figure 20 (the XBee throughput curve) to the receiver throughput of figure 41. In this case, instead of plotting the "measured min" and "measured max", we plot instead the average of these 2 curves. We can see in the figure that both throughputs are very close to each other when the full Libelium API is used at the sender side. The explanation is because the time between 2 send() with the full Libelium API cannot be reduced enough to overflow the receiver. We will see in the next section that with the light Libelium API, this is not the case anymore.

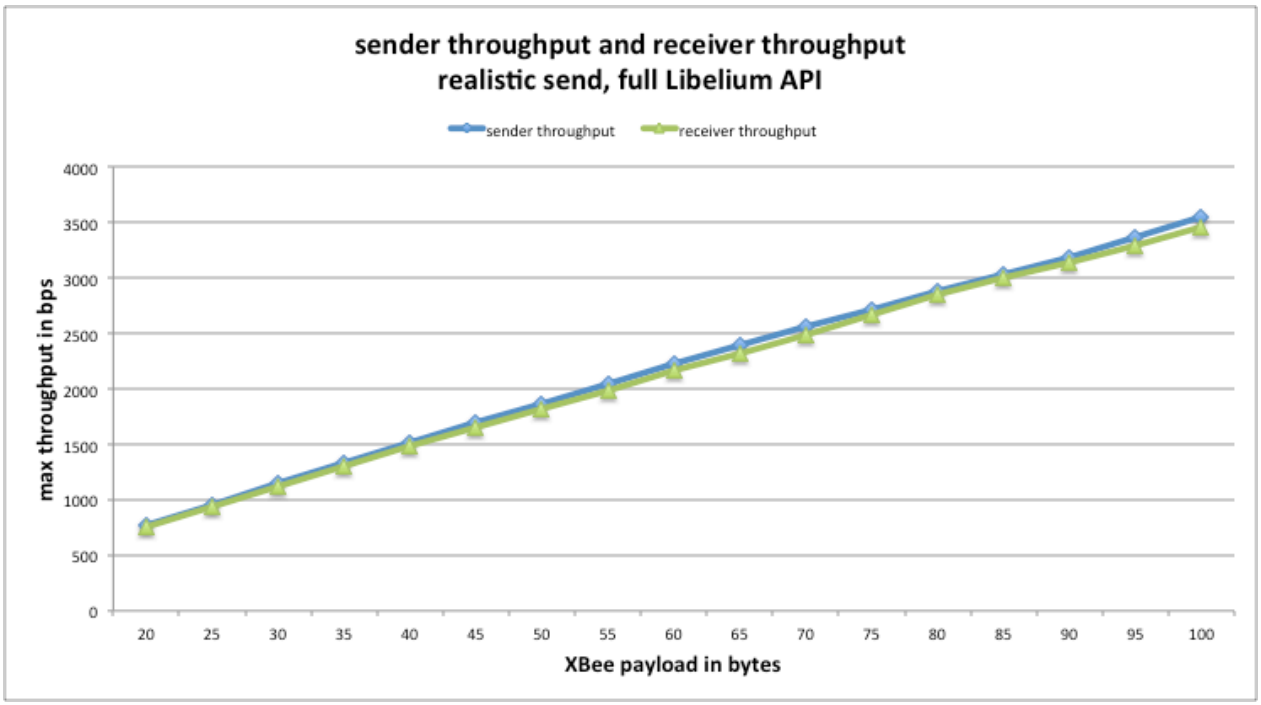


Figure 43: sender throughput and receiver throughput, sender uses full Libelium API

B/ Receiver throughput when sending with light Libelium API

When the sender uses the light Libelium API, the time between 2 packet generation could be greatly decreased as shown previously when comparing figure 27 to figure 19. Therefore, we could expect a much higher throughput at receiving side than those illustrated in figure 42.

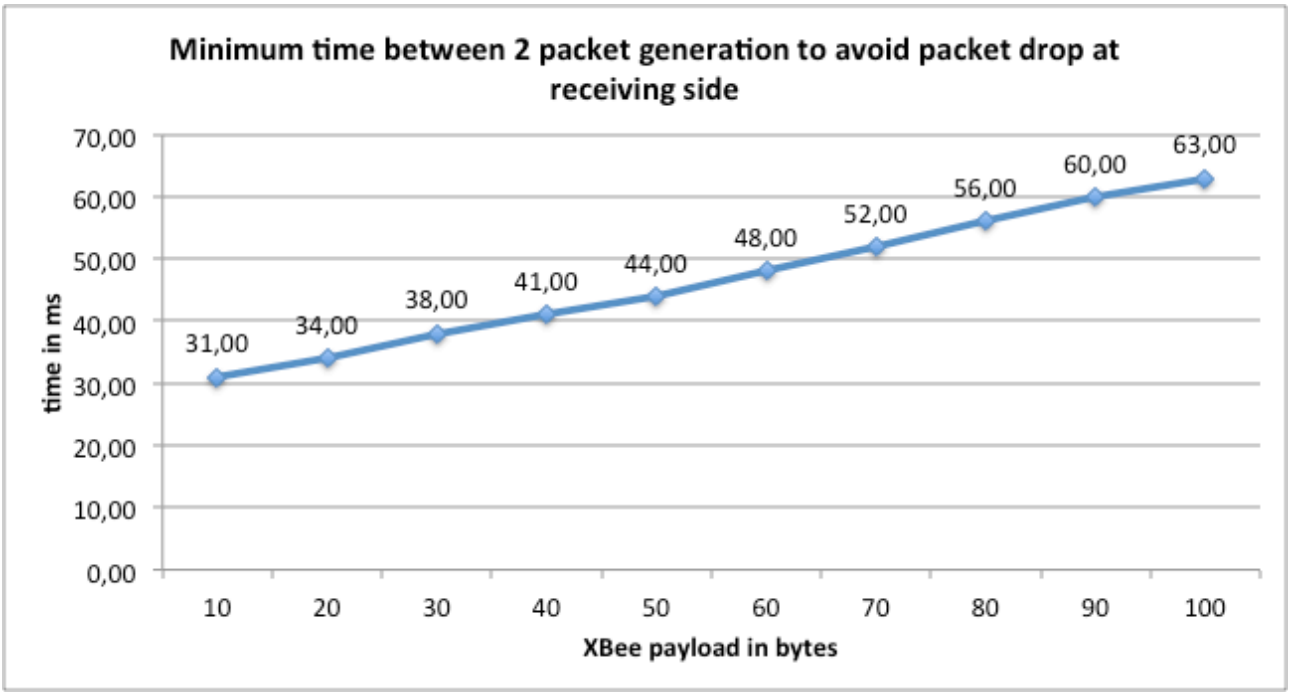


Figure 44: Minimum time between 2 packet generation to avoid packet drop at receiving side

However, at higher sending rate, packets may arrive faster at the receiver than it can read them from the radio module. Therefore, in order to have consistent throughput measures, packets are sent back to back at the sender side at the maximum rate that does not overflow or provoke truncated packets in the receiver radio buffer. Figure 44 shows the minimum time

between 2 packet generation, at the sender, to avoid packet drop at receiving side. Note that for the tests depicted in figure 44, the receiver spends all its time reading data from the radio module without any processing tasks on the input data to avoid radio buffer overflows.

Figure 45 compares the sender throughput shown previously in figure 28, referred to as back-to-back sender throughput (red curve in figure 45), to the sender throughput that avoids packet drops at the receiver side. This throughput is therefore the maximum throughput achieved at the receiver side. We can clearly see that, unlike the previous case with the full Libelium API, when the sender uses the light Libelium API the receiver cannot follow the sender's pace and the receiver throughput diverges when the payload increases. **Therefore, the throughput shown in figure 45, blue curve, really represents the maximum throughput that could be achieved at the receiving side.**

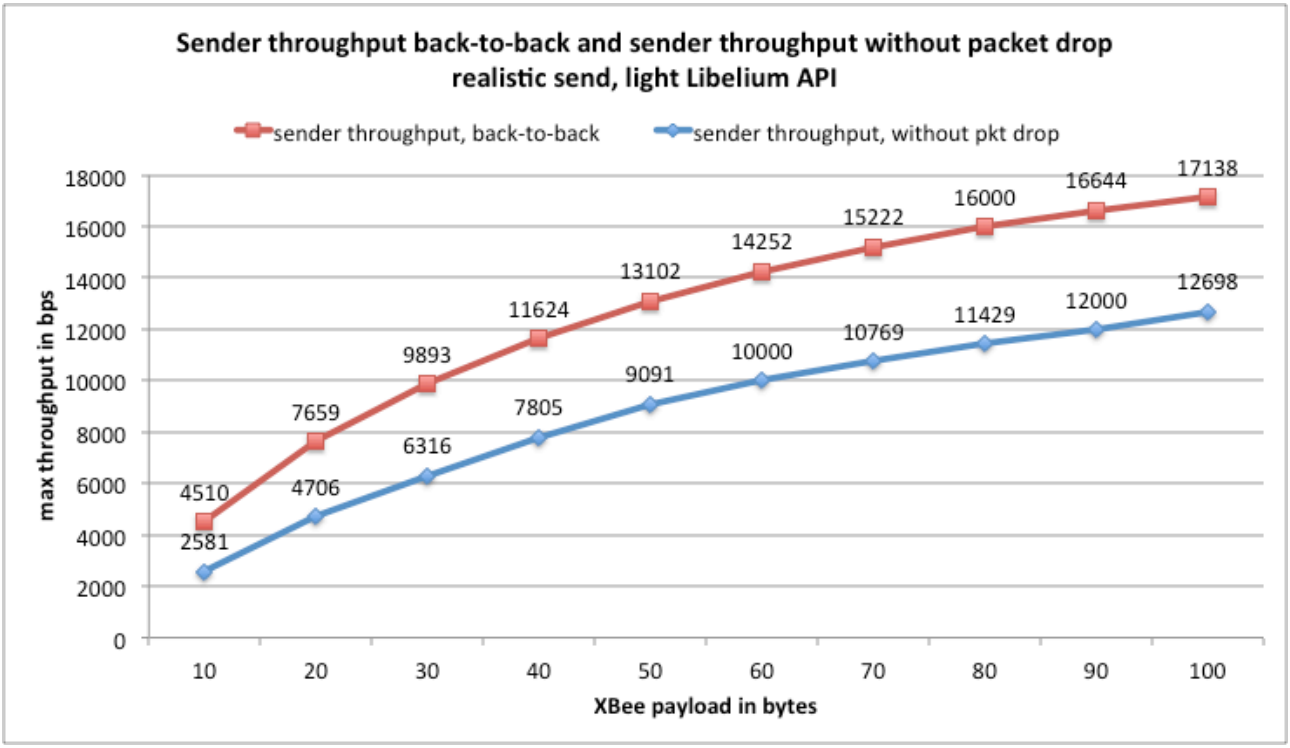


Figure 45: sender throughput back-to-back and receiver throughput, sender uses light Libelium API

Limitations on throughput

When using the full Libelium API (64-bit address), the bottleneck in getting more throughput is clearly on the API library (see figure 16). If the light Libelium API is used or the full Libelium API with 16-bit address, figure 18 and 38 showed that the bottleneck has moved to the time required to write to the radio. In this case, it makes sense to try to reduce this amount of time. As depicted in figure 7, the WaspMote microcontroller communicates with the XBee module through a serial line configured at 38400 bauds. This communication speed directly determines the time required to write to the radio the data to be transmitted.

A/ XBee module in API mode

An XBee module receives data frames in a so-called API mode is which data and required addressing information are passed to the radio module in a structured way. Figure 46 from [XBeeDigi] shows the API frame for a transmit request.

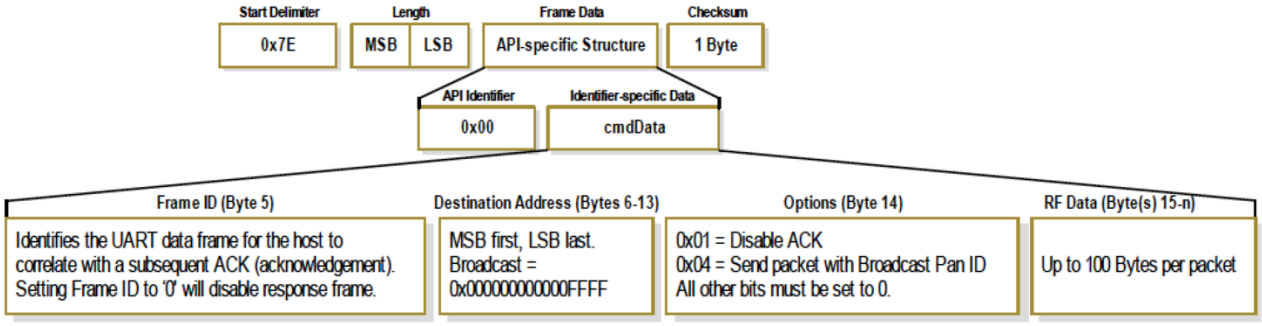


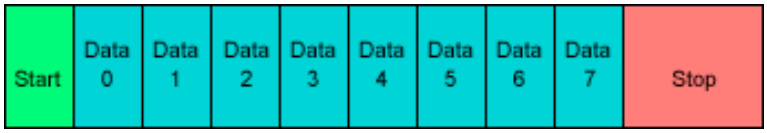
Figure 46: API frame for a transmit request

The important information from this figure is that for any data packet from the application, the API frame format adds 15 bytes (when 64-bit addresses are used) that need to be passed to the radio module in addition to the user payload.

B/ Time to write to radio at 38400 bauds

The configuration of the serial communication between the microcontroller and the XBee is as follows, which is usually noted as 38400/8N1:

1. speed rate is 38400 bps
2. 1 start bit
3. 8 bits of data
4. 1 stop bit
5. no parity bit



Therefore, for each byte that need to be sent to the radio, we have actually 8+1+1=10 bits to be sent through the UART. The following table illustrates for various payload values the total number of bits to be sent to the radio. The last two columns show the theoretical and actual measured time needed to write to the radio at 38400bps respectively (previously shown in figure 22 for the light Libelium API case).

user payload in bytes	Xbee frame size in bytes	physical # of bits	theoretical time at 38400bps in ms	measured time at 38400bps in ms
10	25	250	6,51	6,2
15	30	300	7,81	8,25
20	35	350	9,11	10,25
25	40	400	10,42	10,4

30	45	450	11,72	10,3
35	50	500	13,02	12,5
40	55	550	14,32	12,35
45	60	600	15,63	14,3
50	65	650	16,93	16,45
55	70	700	18,23	16,5
60	75	750	19,53	18,55
65	80	800	20,83	20,65
70	85	850	22,14	20,7
75	90	900	23,44	22,65
80	95	950	24,74	24,75
85	100	1000	26,04	24,8
90	105	1050	27,34	26,9
95	110	1100	28,65	28,8
100	115	1150	29,95	28,9

C/ Time to write to radio at various baud rates

Figure 47 shows the estimated time to write to radio at various baud rates, greater than 38400. We can clearly see the theoretical reduction in time needed to write to the radio module and figure 48 shows the corresponding maximum expected sending throughput when the time to write to radio can be reduced in the send() function.

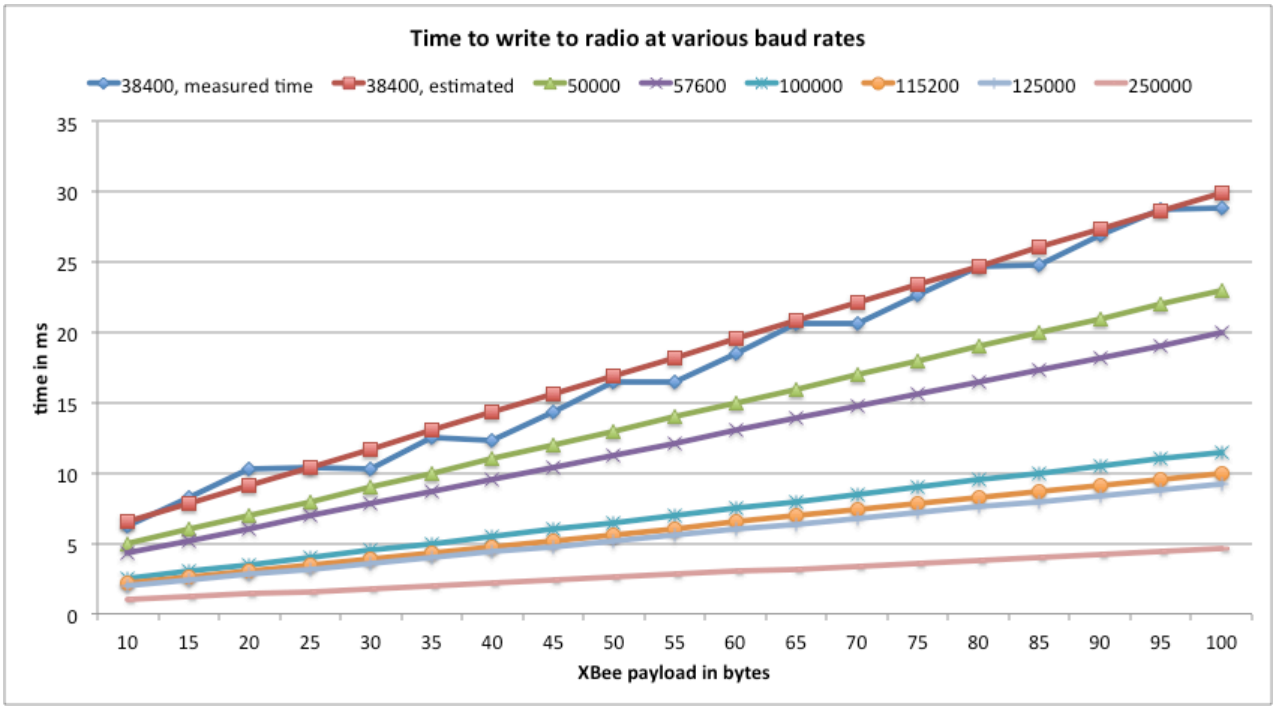


Figure 47: time to write to radio at various baud rates

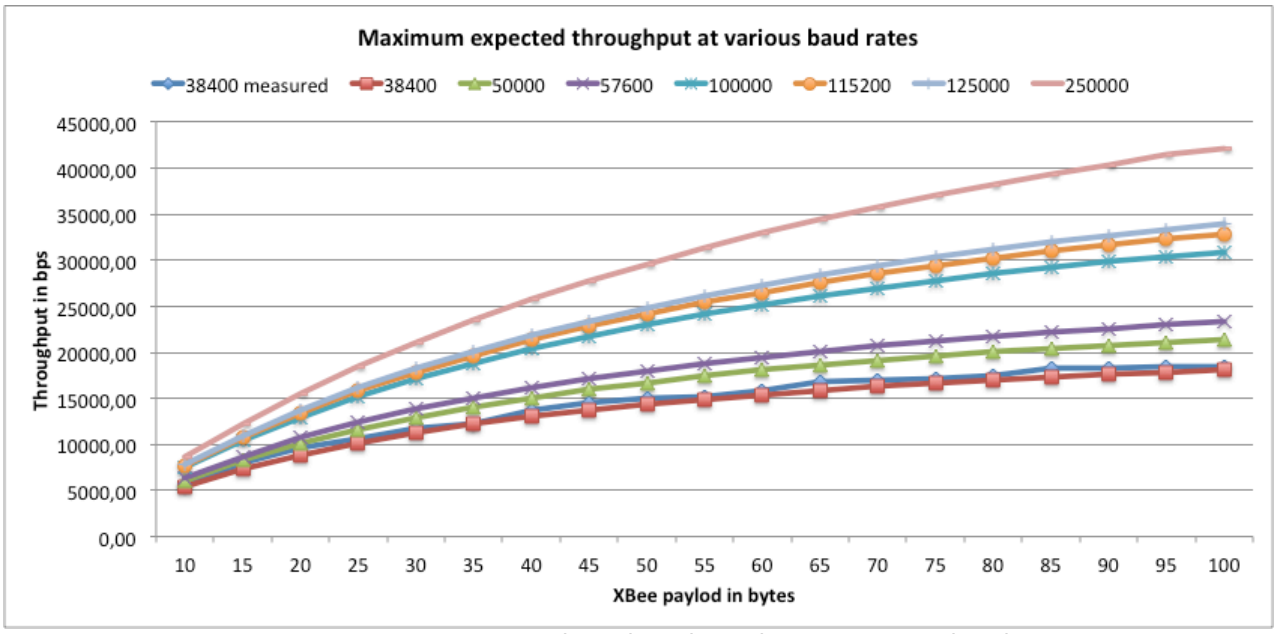


Figure 48: maximum expected sending throughput at various baud rates

The "38400 measured" curve is the same than the one shown previously in figure 25 where the maximum throughput with a payload of 100 bytes was 18497bps, using the light Libelium API. We can see that if the time to write to radio could be reduced by using a higher baud rate, the maximum expected sending throughput could be increased, but still the time spend in send() cannot be reduced further because of the other overheads as shown previously in figure 22.

To produce the curves depicted in figure 48, we performed a linear interpolation for the measured time writing to radio and for the measured time in send(). The reason for this manipulation is to remove randomness in the measures as the estimation process for higher baud rates amplifies any small measure uncertaincy. Figure 49 shows this interpolation.

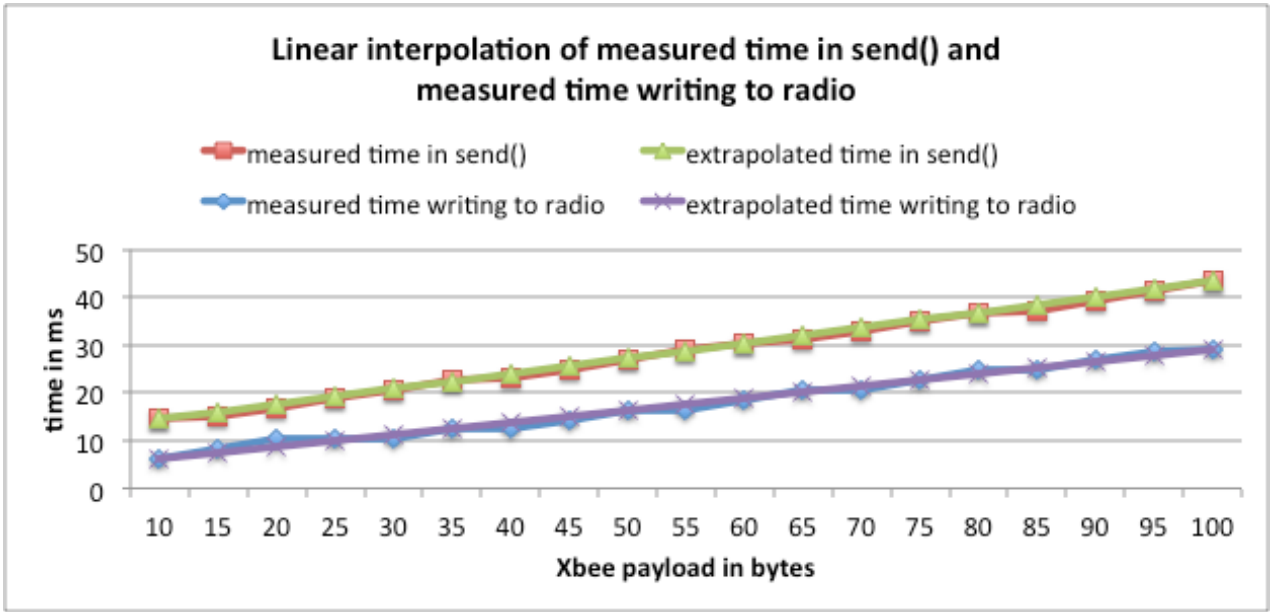


Figure 49: linear interpolation to avoid amplifying measure uncertaincy

D/ Reliability at various baud rates

Increasing the baud rate cannot be done without taking into account some timing constraints that may make the serial communication unreliable [FOS11]. The WaspMote microcontroller runs at 8MHz while the XBee module has an 16MHz clock and requires that the frequency is 16 times the baud rate. It means that for a baud rate of 38400, the actual operating frequency need to be 16*38000 = 614400Hz. For reliable communication, the WaspMote clock should also produce a frequency close to 614000Hz. Since it runs at 8MHz, the dividing factor is 8000000/614000 = 13.020833. Using the nearest integer dividing factor of 13, the actual baud rate is 8000000/16/13 = 38461,54 which is 1.0016026 times greater than the target baud rate. The error is about 0.1602% which allows for reliable communication between the microcontroller and the XBee module. The table below summarizes for standard target baud rates the actual baud rate between the microcontroller and the XBee module.

Baud rate	frequency	dividing factor	nearest	actual baud rate	ratio	% error
1200	19200	416,6666667	416	1201,92	1,001602564	0,16025641
2400	38400	208,3333333	208	2403,85	1,001602564	0,16025641
4800	76800	104,1666667	104	4807,69	1,001602564	0,16025641
9600	153600	52,08333333	52	9615,38	1,001602564	0,16025641
14400	230400	34,72222222	34	14705,88	1,02124183	2,124183007
19200	307200	26,04166667	26	19230,77	1,001602564	0,16025641
38400	614400	13,02083333	13	38461,54	1,001602564	0,16025641
57600	921600	8,680555556	8	62500,00	1,085069444	8,506944444
115200	1843200	4,340277778	4	125000,00	1,085069444	8,506944444

What can be seen is that 38400, which is the value chosen by the Libelium API, is actually the fastest standard baud rate that provides acceptable errors between the target baud rate and the actual baud rate. Using 57600 or 115200 baud rates would generate too many errors, making the communication very unreliable and therefore not functioning at all.

Using these constraints, the perfect dividing factors are 10, 5, 4, 2 and 1 that correspond to 50000, 100000, 125000, 250000 and 500000 baud rates respectively. As we showed that the maximum 802.15.4 effective throughput is roughly 166666bps in broadcast mode when there are no errors, there is no point to consider 500000 baud rate. We can see in figure 48 the theoretical throughput for baud rates up to 250000. **We have performed experiments with XBee module set at 250000 baud and modified the Libelium API to also run at 250000 baud and the measured results are consistent with the theoretical results, allowing much faster sending rates.**

Figure 50 shows the estimated and measured time between 2 packet generation for data transfer rates of 125000 and 250000 with the WaspMote. Figure 51 shows the estimated and measured sending for data transfer rates of 125000 and 250000. We can see that the estimated and the measured curves are very close each other, thus validating our estimation method of the time to write to radio and the constant overheads of the communication API.

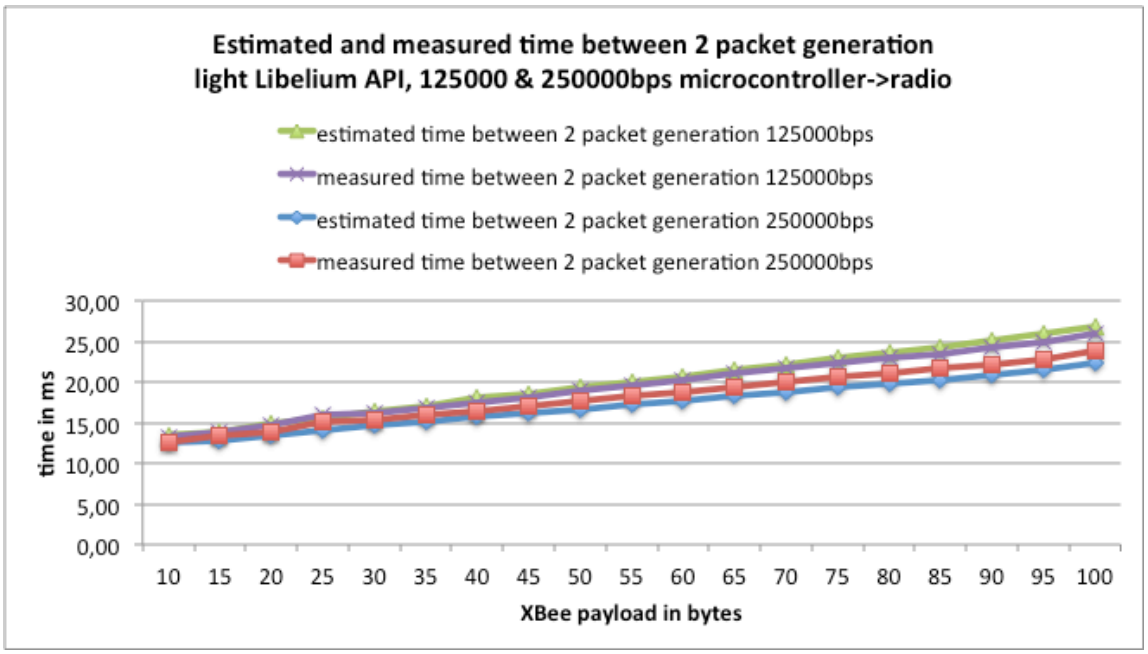


Figure 50: estimated and measured time between 2 packet generation for data transfer rates of 125000 and 250000

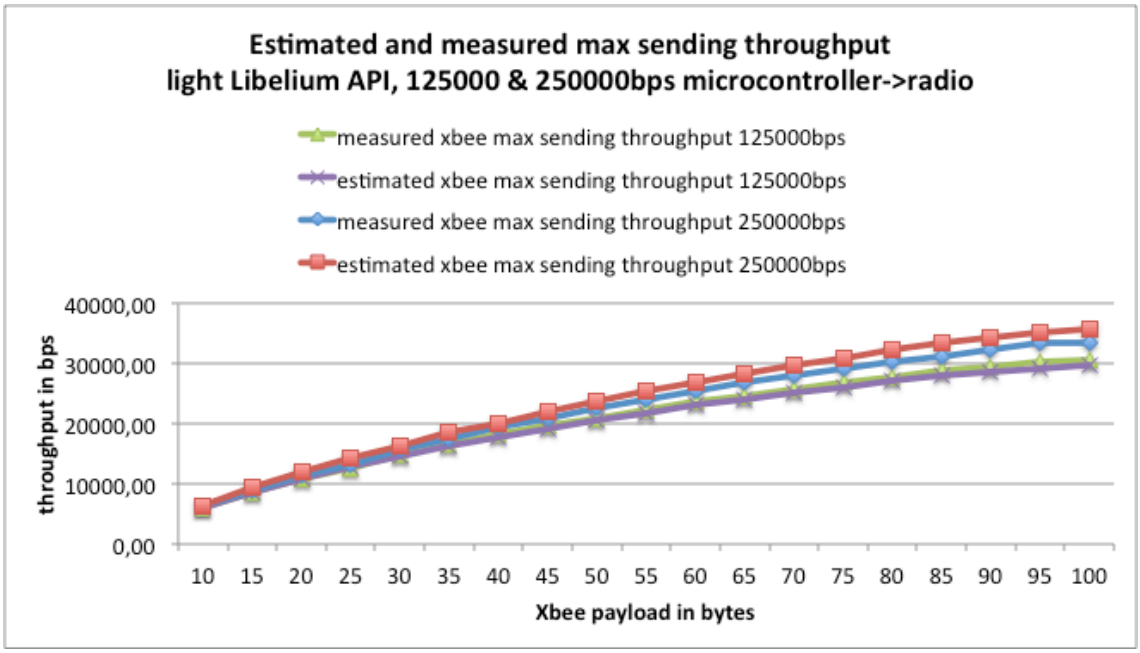


Figure 51: estimated and measured sending throughput for data transfer rates of 125000 and 250000

Comparison with Arduino platforms

Libelium WaspMote is very similar to Arduino board (actually, the IDE is based on the one of Arduino and many low level libraries come from the Arduino framework) as both use the same radio modules from Digi. Therefore it is interesting to compare quickly the throughput one can achieved with Libelium WaspMote to what could be obtained with an Arduino board. As a very light communication library for the Digi XBee is available on Arduino that basically write data to the radio XBee module without much overhead (http://code.google.com/p/xbee-arduino/), **we can consider that the Arduino board with the Arduino XBee library is among the lowest overhead communication stacks that could be produced on the same type of hardware. The throughput that can be obtained is therefore a practical upper bound that will be quite difficult to push further.**

Figure 52 shows the time in send() breakout with an Arduino MEGA 2560 and an Digi 802.15.4 module with the Arduino XBee communication library. We set the communication between the Arduino host and the XBee module to the default 38400 baud rate used by Libelium WaspMote.

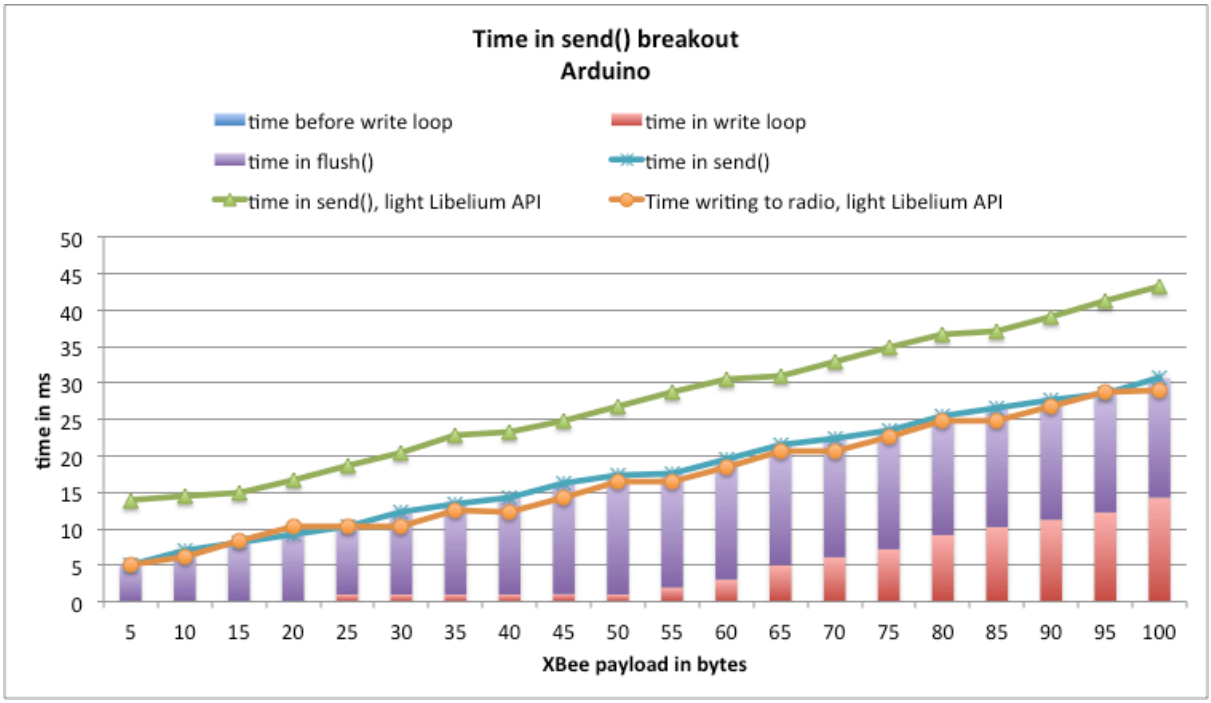


Figure 52: time in send() breakout, Arduino 802.15.4 module

We can see that the time to write to radio is quite similar but compared to the light Libelium API version, the time spent in send() is even smaller (on Arduino, the time spent in send() is the sum of the time before radio, the time in write loop and the time in flush()) leading to a higher maximum sending throughput. Figure 53 shows the time between 2 packet generation and the time in send() with the Arduino board and Arduino XBee communication stack.

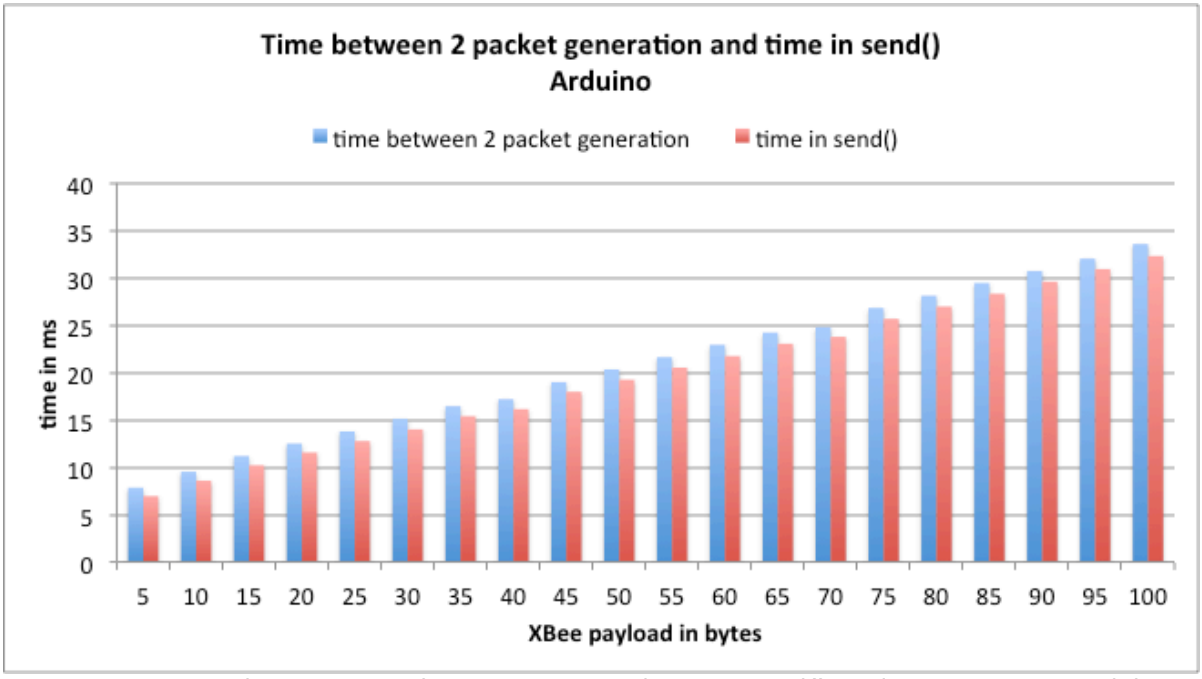


Figure 53: time between 2 packet generation and time in send(), Arduino 802.15.4 module

With the DigiMesh module, the performances are quite similar as depicted by figure 54 and 55.

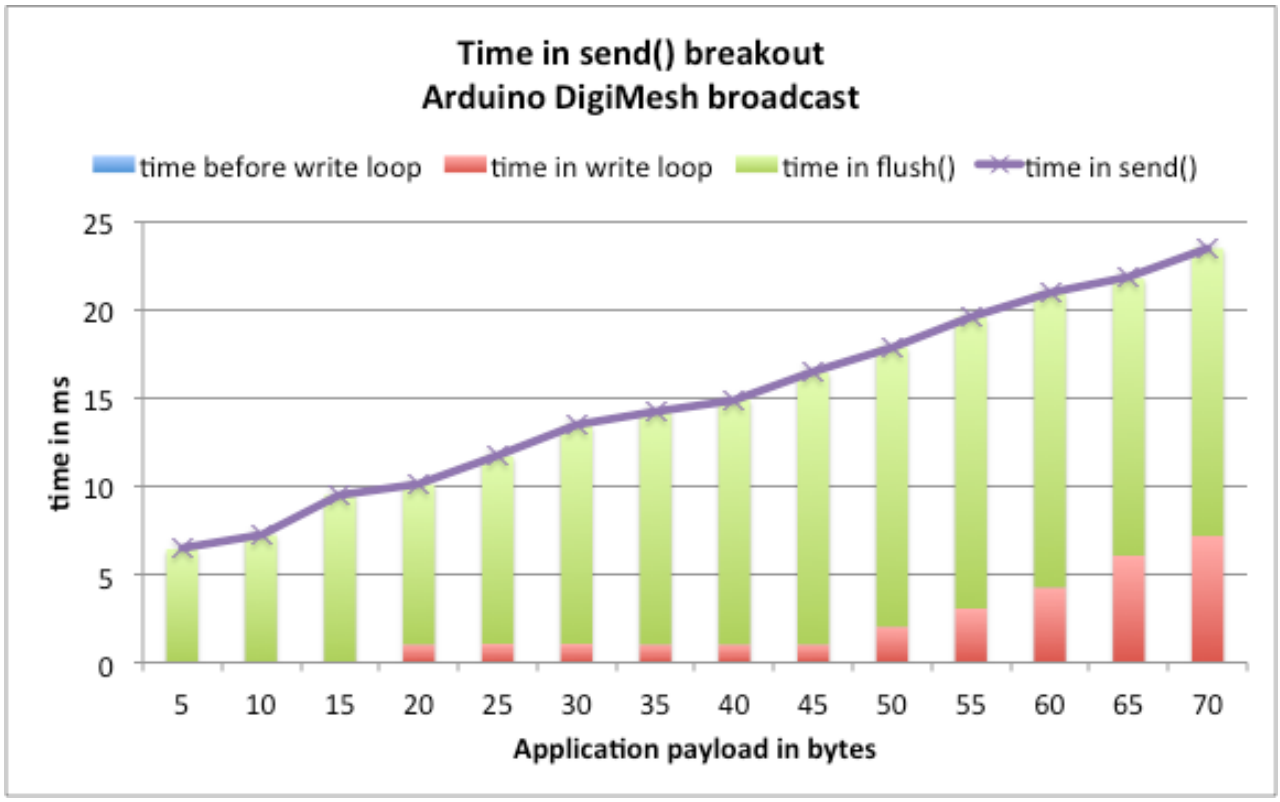


Figure 54: time in send() breakout, Arduino DigiMesh broadcast

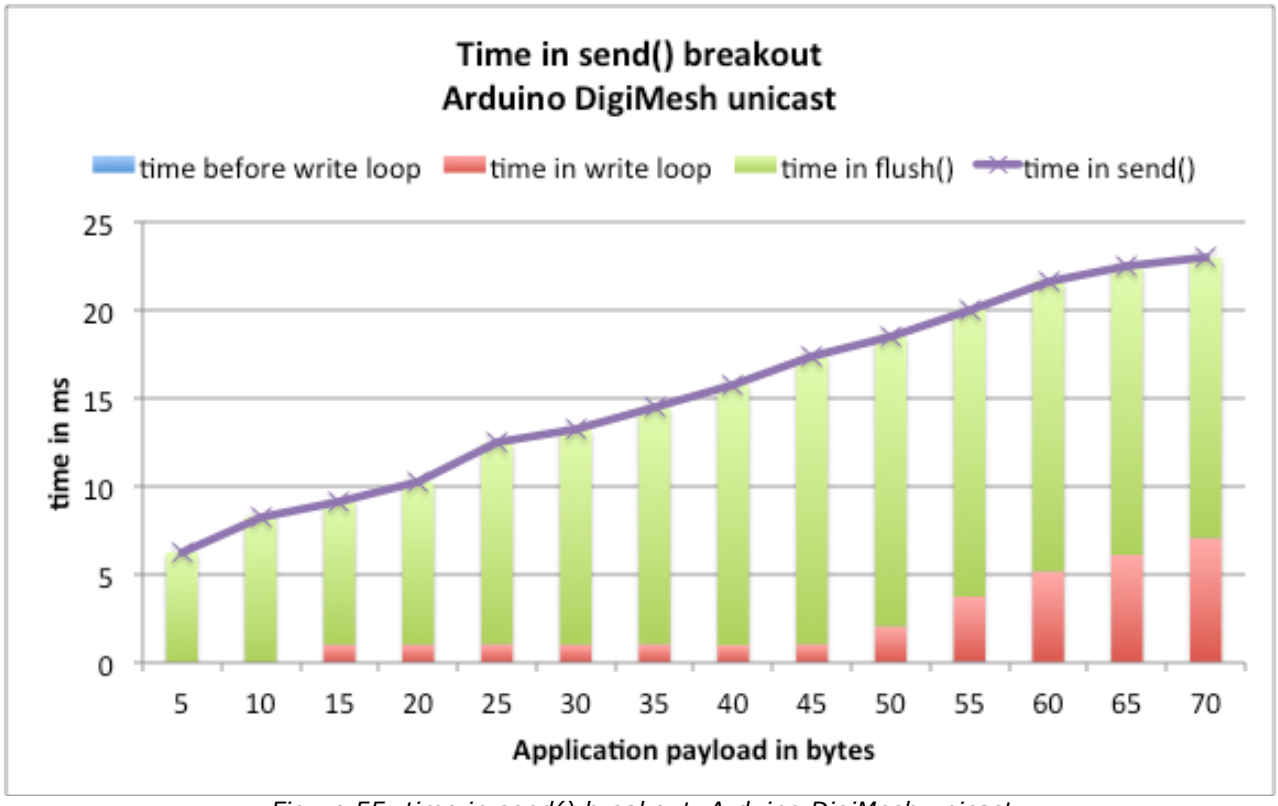


Figure 55: time in send() breakout, Arduino DigiMesh unicast

We can summarize all the 1-hop results in figure 56 that plots the application level maximum sending throughout for the various hardware and communication API.

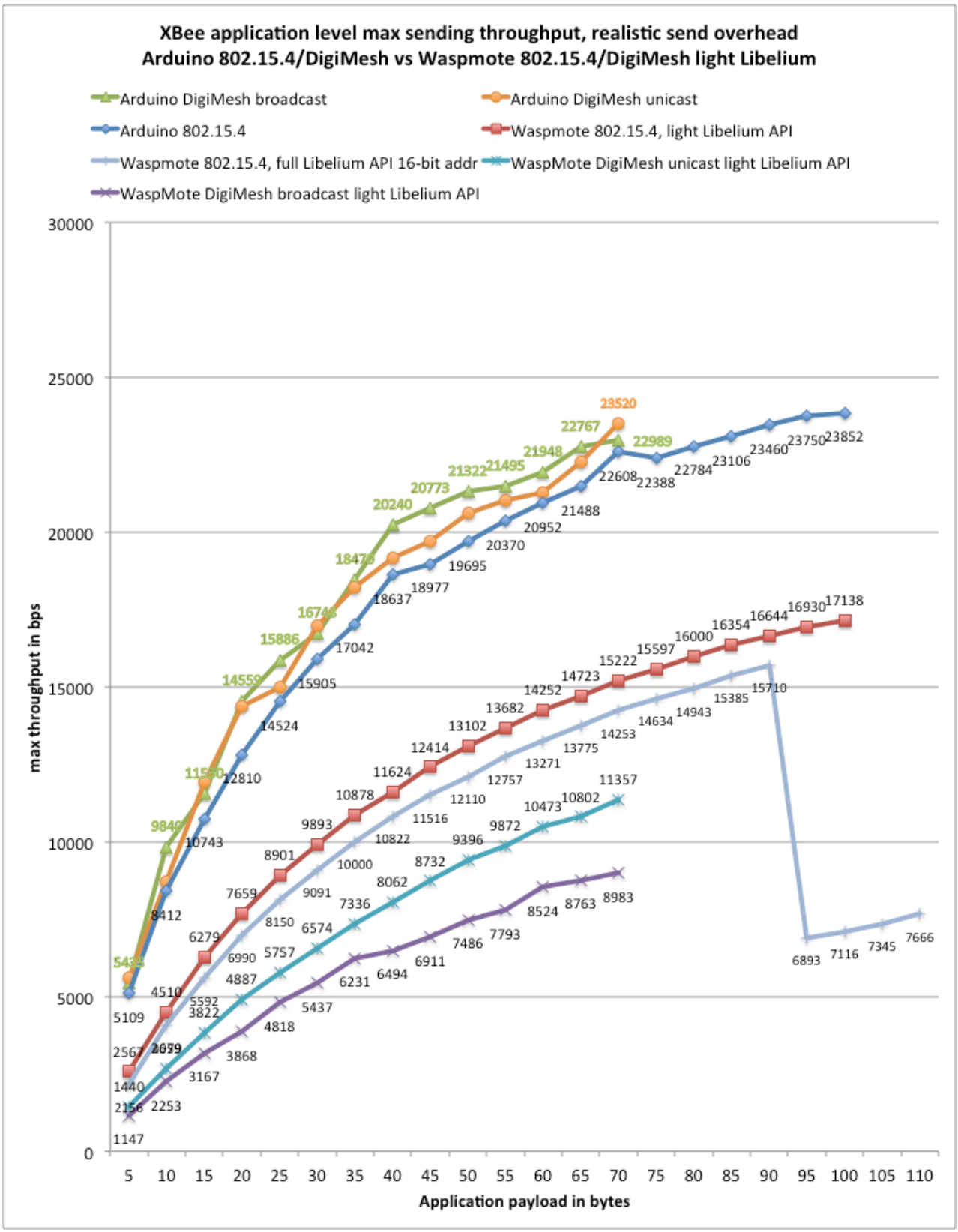


Figure 56: application level maximum sending throughput for various hardware and communication API

5. SmartSantander network qualification: 2-hops and beyonds

Theoretical study

The XBee 802.15.4 module that is used for application/user data traffic has no embedded routing features, unlike the Digimesh version that is used for control traffic on the SmartSantander network. Therefore, if multi-hop communication is required, this facility must be implemented at application level (the Libelium API is considered running at application level because the entire API code is compiled and executed as an application).

In this case, multi-hop communications with relay nodes can generically be represented by figure 57 below which depicts the case for 1 relay node.

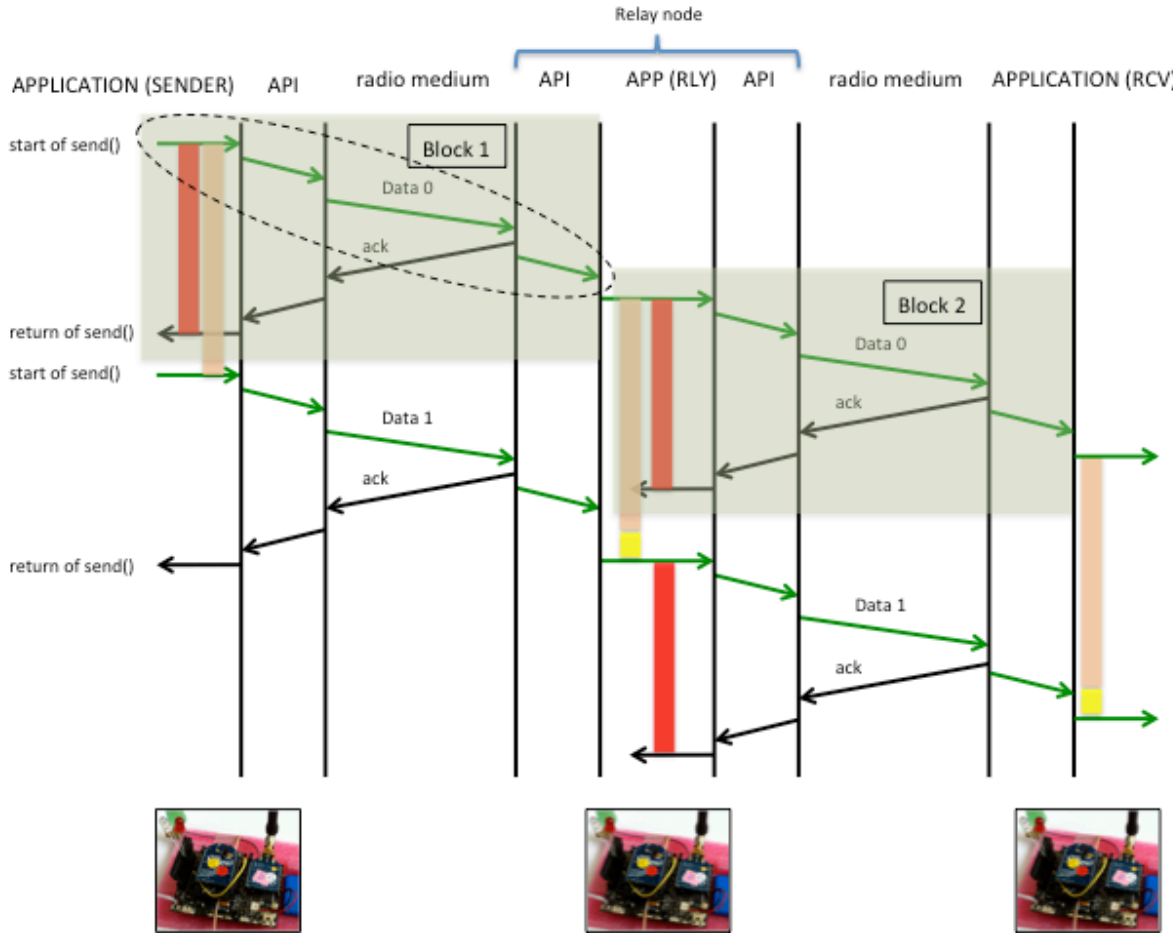


Figure 57: generic relay process, 2-hop with 1 relay node

We can actually see that the sending block (green blocks) is reproduced at each stage. For 2-hop with 1 relay node we can see that the 2-hop latency is at least the time between 2 packet generation at the relay node (block 2) plus the 1-hop latency between the sender and the relay node (dashed oval in block 1). From the previous experiments, we know the miminum time between 2 packet generation and we can determine the 1-hop latency as follows:

$$1\_hop\_latency = time\_before\_radio + time\_to\_write\_to\_radio + transmission\_time + propagation\ time + app\_rcv\_overhead + time\_to\_read\_from\_radio$$

transmission_time is derived from the 802.15.4 physical data rate and the study presented in section "802.15.4 PHY Maximum application throughput" can be used here. If we consider SHR+PHR+PSDU where PSDU = app_payload + addressing_field_size + minimumHeaderSize, we have for an application payload of 100 bytes and an addressing field size of 20 bytes (64-bit addresses) a total physical frame size of 131 bytes. Therefore the total overhead to add to the application payload is 31 bytes.

We have measured the time_before_radio and the time_to_write_to_radio (see for instance figure 16). The propagation time is neglectible at these time scales so we can just remove this time component. app_rcv_overhead is the overhead at the receiver to prepare the reception process and begin reading from the radio module incoming data. We can determine this parameter by comparing the time_between_packet_generation (see for instance figure 19) and the mean inter-arrival time at the receiver (see for instance figure 28). By taking the difference between these 2 values, we have an estimation of the app_rcv_overhead value. Table below gives the 1-hop latency (last column) when the payload is varied using the full Libelum API at the sender, under the assumption that the time_to_read_from_radio is similar to the time_to_write_to_radio.

XBee payload (bytes)	phy frame size (bytes)	time before radio (ms)	time writing to radio (ms)	app receive time (ms)	avg app receive time (ms)	time reading from radio (ms)	1-hop latency (ms)
20	51	119,70	8,65	8,48	6,89	8,65	145,52
25	56	119,95	10,35	7,95	6,89	10,35	149,33
30	61	119,85	11,15	6,52	6,89	11,15	150,99
35	66	120,10	12,50	8,33	6,89	12,50	154,10
40	71	119,90	14,55	7,95	6,89	14,55	158,16
45	76	121,10	15,80	7,11	6,89	15,80	162,02
50	81	121,15	17,00	6,53	6,89	17,00	164,63
55	86	120,15	19,00	6,22	6,89	19,00	167,79
60	91	119,05	19,25	7,92	6,89	19,25	167,35
65	96	118,80	20,30	7,21	6,89	20,30	169,36
70	101	119,70	22,60	6,16	6,89	22,60	175,02
75	106	119,60	24,00	6,16	6,89	24,00	177,88
80	111	120,56	23,77	6,37	6,89	23,77	178,53
85	116	121,25	25,30	5,50	6,89	25,30	182,45
90	121	121,35	26,90	4,42	6,89	26,90	185,91
95	126	121,35	26,80	6,26	6,89	26,80	185,87
100	131	120,90	28,95	7,98	6,89	28,95	189,88

Then the k-hop latency could be assumed to be k times the 1-hop latency in case of error-free environment. When the time between 2 sends is small, there will be many interferences between the sender and next relay node and also between 2 consecutive relay nodes themselves that may make the error-free environment assumption difficult to hold when k is greater than 2.

When the time between 2 sends is much larger than the transmission time, and under the error-free assumption, once the sending of packets at the sender has been initiated, the final receiver can receive after the k-hop latency a continuous flow of packets,. Therefore, if we look at the steady throughput (measured at the time of reception of the first packet), the receiving throughput could be quite close to the sending throughput. We have seen that the miminum inter-arrival time of packet at the receiver is a bit larger (by an app_rcv_overhead amount) than the minimum time between 2 packet generation. Therefore the receiver throughput will be limited by this miminum inter-arrival time of packet. Now, if we are interested in the effective throughput by considering that the sender starts sending packets at time TSS and stop sending packets at time TES then the final receiver at k hops will receive all the packets at time TER=TSS+TES+k*1_hop_latency in the best case. Again, if the number of packets is

large, the effective throughput will be close to the steady throughput.
More useful results should consider heavy traffic loads (either fast sends or concurrent sends from several active nodes), interferences between send and receive operations, relaying errors and contention on the radio channel. In this case, the packet transmission time may be increased, as well as the medium access time. There are some theoretical researches that determine the maximum throughput in a multi-hop environment [LI01][SUN06], propose multi-hop error and interference models [MOR10] or theoretical performance studies based on optimal link scheduling [MAO11] but these are beyond the scope of this report. However, we can quickly summarize the most important results which are:

1. when N active nodes, belonging to the same path, are within each other's transmission range, the maximum effective rate on that path is C=(N-1) since only one of the N nodes can transmit at any time.
2. the maximum throughput in a multi-hop wireless network is 1/3 when the radio interfering range and the transmission range is the same.
3. as the radio interference range is usually much larger than the transmission range, the effective end-to-end capacity of a chain configuration will further decrease.

Experimental measures of relaying performances

In order to get more accurate values of relaying performances, we experimentally measure the overhead of relay processing which consist in receiving the data and sending it to the next hop. In the previous theoretical study, we assumed that the time needed to read from the radio, noted t_{read} is similar to the time need to write to the radio. Actually, our measures show that this is generally not true: we found that the time to read the received data is quite independent from the communication baud rate between the micro-controller and the radio module. We tested with baud rates of 38400, 125000 and 250000, and t_{read} depends only on the data size. Figure 58 plots t_{read} (blue curve) for the WaspMote.

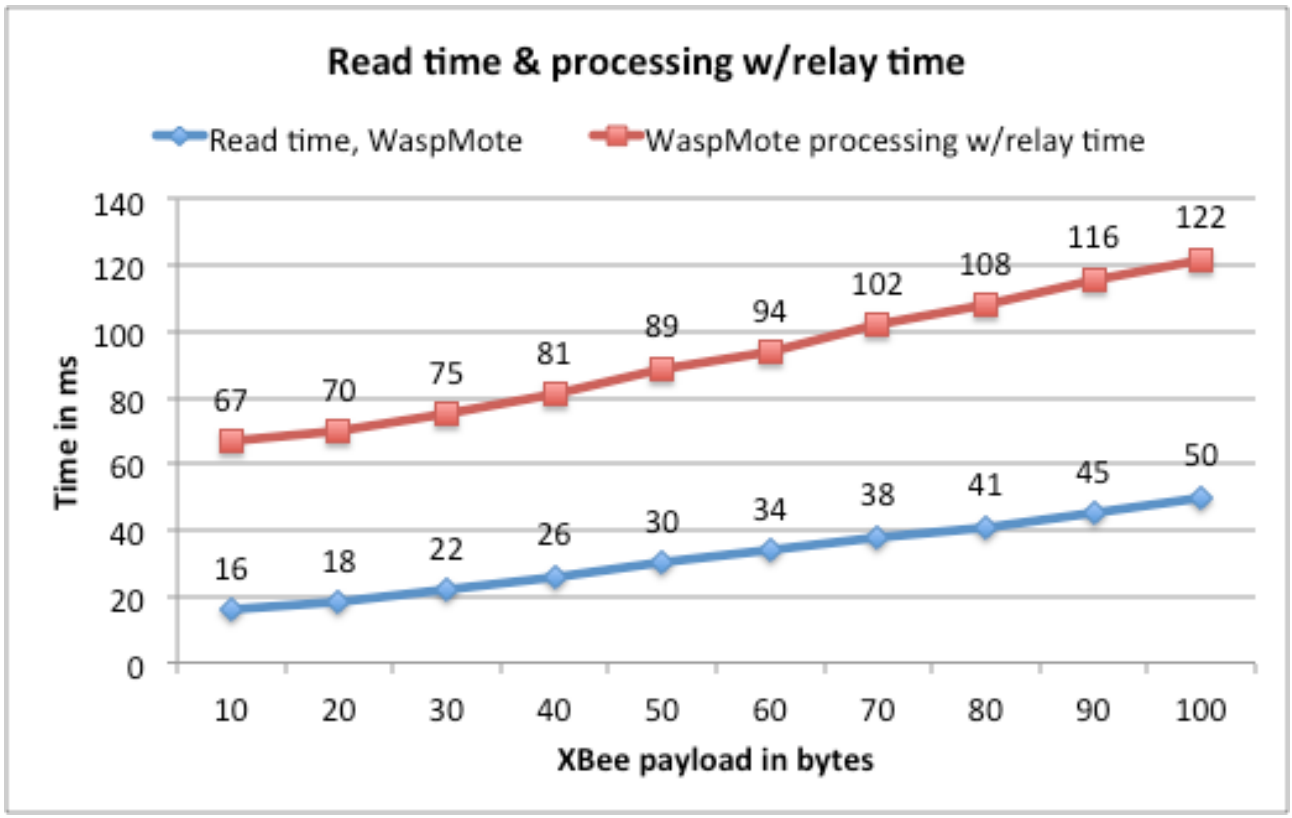


Figure 58: Time to read data from serial (blue) and total processing time w/relay

The reason why t_{read} does not depend on the communication baud rate between the micro-controller and the radio module, at least at the application level, is as follows: most of communication API used a system-level receive buffer and when a packet arrives at the radio, a hardware interrupt in raised and appropriate callback functions are used to fill in the receive buffer that will be read later on by the application. Therefore, the baud rate has only an impact on the time needed to transfer data from the radio module to the receive buffer. When in the receive buffer, the time needed to transfer the data from the receive buffer to the application depends on the speed of memory copy operations, therefore it depends mainly on the frequency used to operate the sensor board and the data bus speed. We measured this time on the WaspMote when the payload size is varied and Figure 58 showed that the time to read a packet of 100 bytes is about 50ms. We did experiments on an Arduino Mega2560 board which is very similar to the WaspMote hardware but running at 16Mhz instead on 8Mhz and we found that the read time is much smaller: 35ms for a 100-byte packet.

In total, when adding additional data handling overheads, a WaspMote relay node needs about 122ms[2] to process the incoming packet and to relay it to the next hop, once again for a 100-byte packet, see red curve in Figure 58. Figure 59 shows the maximum throughput with relay nodes (green curve) and compares it to the previous throughputs. We can see that multi-hop transmission on this type of platform adds a considerable overhead that put strong constraints on the achievable throughput.

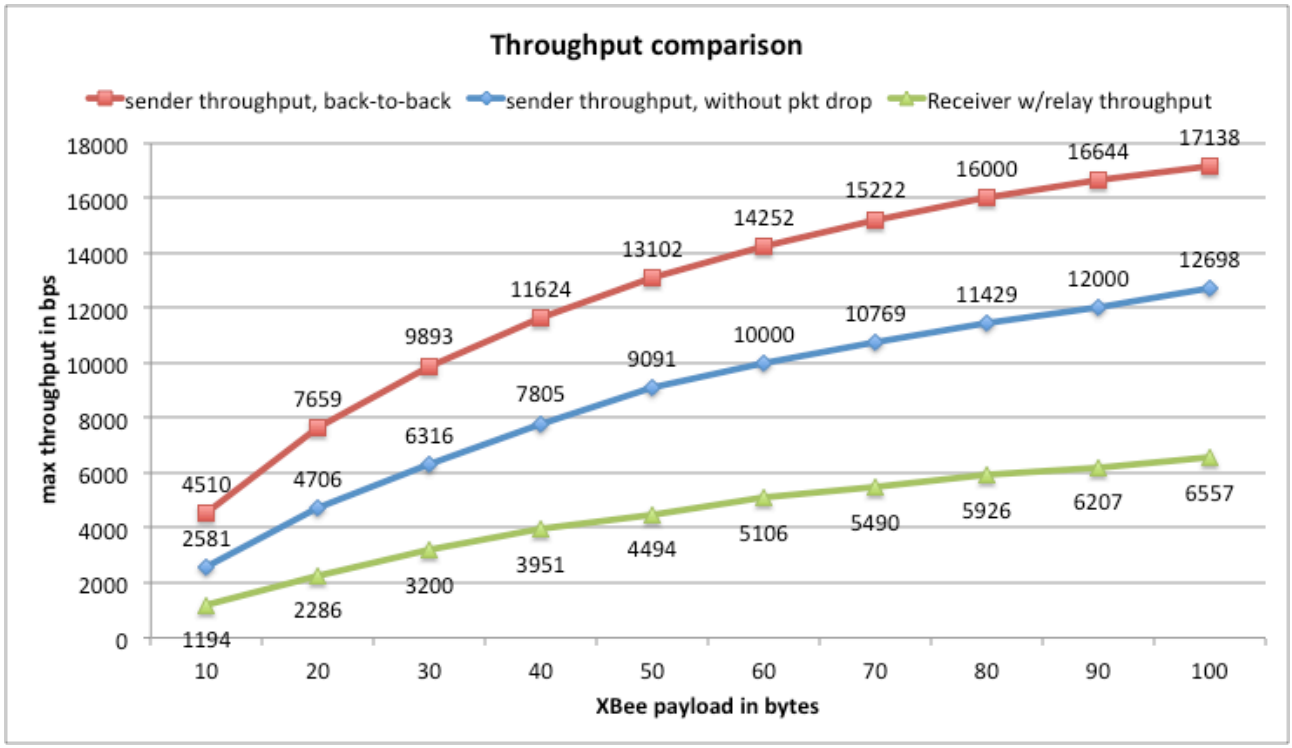


Figure 59: Throughput comparison

[2] *The total processing time with relaying consists in reading incoming data and sending it to the next hop. Here, the 122ms value is for a sending time when the communication between the radio module and the microcontroller is set to the default value of 38400bps. With higher transfer rates, such as 125000 or 250000bps, the sending time is reduced, see figures 47 and 49, therefore the total processing time w/relay can be reduced by about 19.7ms (28.9ms-4.6ms) and 24.3ms (28.9ms-4.6ms) for 125000bps and 250000bps transfer rates respectively, once again for a 100-byte packet.*

Preliminary test of multi-hop transmission

In the context of the SmartSantander network qualification, we will present in this document the experimental tests for a 2-hop communication as depicted by figure 57: the sender is a traffic generator, the relay node is a sniffer receiver that will relay incoming packets and the end receiver is a regular sniffer receiver without relaying facilities. The traffic generator is a SmartSantander IoT node that was kindly lend to us by the SmartSantander research group of University of Cantabria, see figure 60.

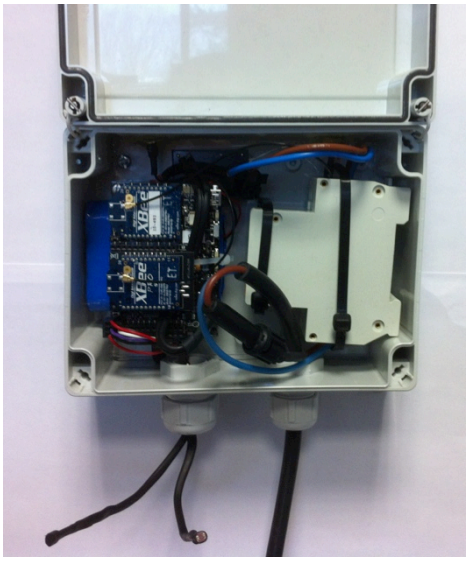


Figure 60: experimental test, SmartSantander IoT node as traffic generator

We deployed the experimental test-bed as depicted by figure 61. The maximum communication range of the traffic generator (802.15.4 PRO module) was found to be about 350m in open space, line of sight, with some trees configuration.

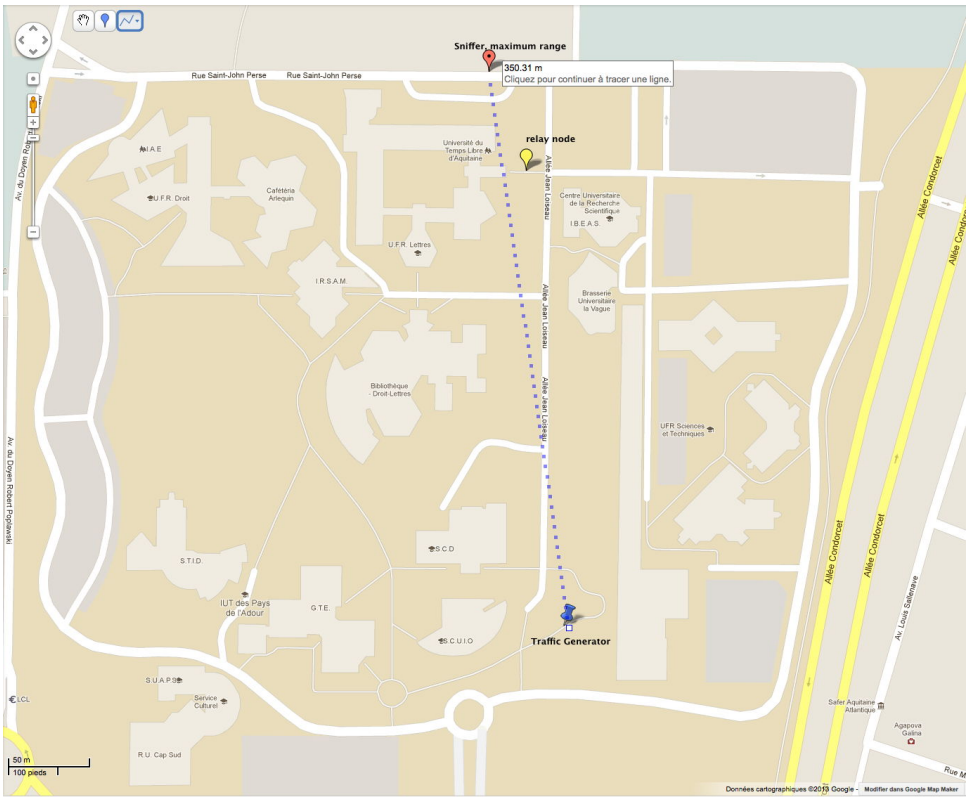


Figure 61: experimental test, traffic generator, final sniffer and relay node

The results are consistent to what could be predicted. When the traffic is light, i.e. the time between 2 packet generation is greater than the minimum time between 2 packet generation, relaying introduces an initial latency but the reception throughput at 2 hops is very close to the sending throughput. We set the traffic generator to send 1 packet with 35 bytes application payload every 800ms. With the full Libelium API, the minimum time between 2 packet generation is about 200ms. Therefore 800ms is 4 times this value, avoiding packet losses by interferences of uplink and downlink traffic. The sending throughput is set to 35*8/0.8=350kbps.

Packets are sent from the Traffic Generator (figure 60) with the following payload:

```
S194#L0macW:0013A20040781DDF*******
```

`S194` means that it is packet with sequence number 194 (the sequence number wraps back at 255) and `L0` means that the packet is sent by the Traffic Generator. `0013A20040781DDF` is the 64-bit address of the Traffic Generator.

The Relay node received these packets and relay them to the final sniffer receiver by changing `L0` into `L1`:

```
S194#L1macW:0013A20040781DDF*******
```

At 1 hop, in reception range, the sniffer receiver receives the packets without errors as can be seen in the following verbatim text:

```
...
tRCV(ms) 17358 tSL(ms) 811 RSSI(-dBm) 40
F(NO_L_API) 59B in 46ms. XAppPay 44B.  -b. 23
7E 00 37 80 00 13 A2 00 40 78 1D DF 28 02 || AC 01 23 02 57 41 53 50 23 53 31 37 32 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A DF
S172#L0macW:0013A20040781DDF*******
57

tRCV(ms) 18173 tSL(ms) 814 RSSI(-dBm) 36
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 23
7E 00 37 80 00 13 A2 00 40 78 1D DF 24 02 || AD 01 23 02 57 41 53 50 23 53 31 37 33 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
```

```
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A E1
S173#L0macW:0013A20040781DDF*******
58

tRCV(ms) 18979 tSL(ms) 807 RSSI(-dBm) 36
F(NO_L_API) 59B in 46ms. XAppPay 44B.  -b. 20
7E 00 37 80 00 13 A2 00 40 78 1D DF 24 02 || AE 01 23 02 57 41 53 50 23 53 31 37 34 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A DF
S174#L0macW:0013A20040781DDF*******
58

tRCV(ms) 19798 tSL(ms) 818 RSSI(-dBm) 36
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 21
7E 00 37 80 00 13 A2 00 40 78 1D DF 24 02 || AF 01 23 02 57 41 53 50 23 53 31 37 35 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A DD
S175#L0macW:0013A20040781DDF*******
58

tRCV(ms) 20608 tSL(ms) 809 RSSI(-dBm) 50
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 21
7E 00 37 80 00 13 A2 00 40 78 1D DF 32 02 || B0 01 23 02 57 41 53 50 23 53 31 37 36 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A CD
S176#L0macW:0013A20040781DDF*******
58

tRCV(ms) 21415 tSL(ms) 809 RSSI(-dBm) 48
F(NO_L_API) 59B in 46ms. XAppPay 44B.  -b. 20
7E 00 37 80 00 13 A2 00 40 78 1D DF 30 02 || B1 01 23 02 57 41 53 50 23 53 31 37 37 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A CD
S177#L0macW:0013A20040781DDF*******
58

tRCV(ms) 22230 tSL(ms) 814 RSSI(-dBm) 42
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 21
7E 00 37 80 00 13 A2 00 40 78 1D DF 2A 02 || B2 01 23 02 57 41 53 50 23 53 31 37 38 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A D1
S178#L0macW:0013A20040781DDF*******
58

tRCV(ms) 23040 tSL(ms) 809 RSSI(-dBm) 41
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 21
7E 00 37 80 00 13 A2 00 40 78 1D DF 29 02 || B3 01 23 02 57 41 53 50 23 53 31 37 39 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A D0
S179#L0macW:0013A20040781DDF*******
58

tRCV(ms) 23869 tSL(ms) 830 RSSI(-dBm) 41
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 21
7E 00 37 80 00 13 A2 00 40 78 1D DF 29 02 || B4 01 23 02 57 41 53 50 23 53 31 38 30 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A D7
S180#L0macW:0013A20040781DDF*******
58

tRCV(ms) 24675 tSL(ms) 807 RSSI(-dBm) 41
F(NO_L_API) 59B in 46ms. XAppPay 44B.  -b. 22
7E 00 37 80 00 13 A2 00 40 78 1D DF 29 02 || B5 01 23 02 57 41 53 50 23 53 31 38 31 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A D5
S181#L0macW:0013A20040781DDF*******
58

tRCV(ms) 25494 tSL(ms) 818 RSSI(-dBm) 48
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 21
7E 00 37 80 00 13 A2 00 40 78 1D DF 30 02 || B6 01 23 02 57 41 53 50 23 53 31 38 32 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A CC
S182#L0macW:0013A20040781DDF*******
58

tRCV(ms) 26304 tSL(ms) 809 RSSI(-dBm) 42
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 21
7E 00 37 80 00 13 A2 00 40 78 1D DF 2A 02 || B7 01 23 02 57 41 53 50 23 53 31 38 33 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A D0
S183#L0macW:0013A20040781DDF*******
58

tRCV(ms) 27107 tSL(ms) 807 RSSI(-dBm) 43
F(NO_L_API) 59B in 46ms. XAppPay 44B.  -b. 20
7E 00 37 80 00 13 A2 00 40 78 1D DF 2B 02 || B8 01 23 02 57 41 53 50 23 53 31 38 34 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A CD
S184#L0macW:0013A20040781DDF*******
58

tRCV(ms) 27926 tSL(ms) 816 RSSI(-dBm) 48
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 21
7E 00 37 80 00 13 A2 00 40 78 1D DF 30 02 || B9 01 23 02 57 41 53 50 23 53 31 38 35 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A C6
S185#L0macW:0013A20040781DDF*******
58

tRCV(ms) 28736 tSL(ms) 809 RSSI(-dBm) 63
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 21
7E 00 37 80 00 13 A2 00 40 78 1D DF 3F 02 || BA 01 23 02 57 41 53 50 23 53 31 38 36 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A B5
S186#L0macW:0013A20040781DDF*******
58

tRCV(ms) 29545 tSL(ms) 813 RSSI(-dBm) 53
F(NO_L_API) 59B in 46ms. XAppPay 44B.  -b. 20
7E 00 37 80 00 13 A2 00 40 78 1D DF 35 02 || BB 01 23 02 57 41 53 50 23 53 31 38 37 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A BD
S187#L0macW:0013A20040781DDF*******
57

tRCV(ms) 30362 tSL(ms) 814 RSSI(-dBm) 61
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 23
```

```
7E 00 37 80 00 13 A2 00 40 78 1D DF 3D 02 || BC 01 23 02 57 41 53 50 23 53 31 38 38 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A B3
S188#L0macW:0013A20040781DDF*******
58

tRCV(ms) 31170 tSL(ms) 807 RSSI(-dBm) 49
F(NO_L_API) 59B in 48ms. XAppPay 44B.  -b. 20
7E 00 37 80 00 13 A2 00 40 78 1D DF 31 02 || BD 01 23 02 57 41 53 50 23 53 31 38 39 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A BD
S189#L0macW:0013A20040781DDF*******
58

tRCV(ms) 31979 tSL(ms) 811 RSSI(-dBm) 50
F(NO_L_API) 59B in 46ms. XAppPay 44B.  -b. 23
7E 00 37 80 00 13 A2 00 40 78 1D DF 32 02 || BE 01 23 02 57 41 53 50 23 53 31 39 30 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A C3
S190#L0macW:0013A20040781DDF*******
57

tRCV(ms) 32794 tSL(ms) 814 RSSI(-dBm) 54
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -b. 21
7E 00 37 80 00 13 A2 00 40 78 1D DF 36 02 || BF 01 23 02 57 41 53 50 23 53 31 39 31 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A BD
S191#L0macW:0013A20040781DDF*******
58


20pkt 16229ms. XBee=433.84bps. App=345.10bps.
Rcv 40. Lost 1
ST 31754ms

tRCV(ms) 33602 tSL(ms) 807 RSSI(-dBm) 56
F(NO_L_API) 59B in 48ms. XAppPay 44B.  -b. 20
7E 00 37 80 00 13 A2 00 40 78 1D DF 38 02 || C0 01 23 02 57 41 53 50 23 53 31 39 32 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A B9
S192#L0macW:0013A20040781DDF*******
58

tRCV(ms) 34413 tSL(ms) 813 RSSI(-dBm) 48
F(NO_L_API) 59B in 46ms. XAppPay 44B.  -b. 23
7E 00 37 80 00 13 A2 00 40 78 1D DF 30 02 || C1 01 23 02 57 41 53 50 23 53 31 39 33 23 4C 30 6D 61 63 57 3A 30 30 31 33 41 32 30 30
34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A BF
S193#L0macW:0013A20040781DDF*******
57
...
```

At 2 hops, with a relay node between the Traffic Generator and the sniffer receiver put beyond the direct transmission range of the Traffic Generator, figure 62 shows that the reception throughput at the sniffer receiver node is close to the sending throughput when the packet loss rate is low. We can see with the payload that the packet received at the sniffer receiver is actually a relayed packet:

```
...
tRCV(ms) 1001366 tSL(ms) 805 RSSI(-dBm) 72
F(NO_L_API) 59B in 47ms. XAppPay 44B.  -u. 23
7E 00 37 80 00 13 A2 00 40 76 20 5B 48 00 || 5F 01 23 02 57 41 53 50 23 53 39 35 23 4C 31 6D 61 63
57 3A 30 30 31 33 41 32 30 30 34 30 37 38 31 44 44 46 2A 2A 2A 2A 2A 2A 2A 2A 92
S95#L1macW:0013A20040781DDF********
58

20pkt 16221ms. XBee=434.06bps. App=345.27bps.
Rcv 620. Lost 123
ST 1000328ms
...
```

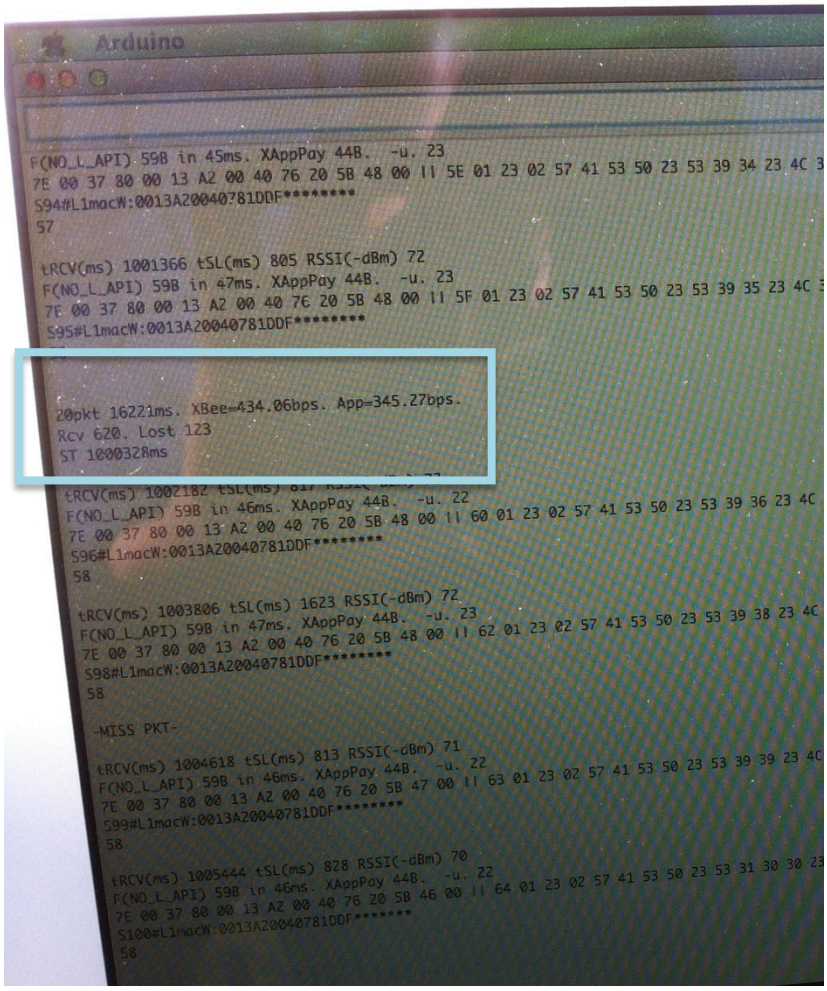


Figure 60: experimental test, traffic generator, final sniffer and relay node

When reducing the time between 2 packet generation and when this time is close to the minimum time between 2 packet generation, there are many transmission errors at the Traffic Generator and at the relay node, with a direct consequence on the receiver throughput.

Future tests will be realized on the SmarSantander test bed to quantify the impact of having many sources, complex interferences patterns,...

6. Preliminary tests of audio streaming on the SmartSantander test-bed

Experimental test-bed

The experiment uses 1 source node consisting of an Arduino Mega2560 with an XBee 802.15.4 module. The audio files are stored on an SD card and we can dynamically select which file is going to be sent, see Figure 61(left).

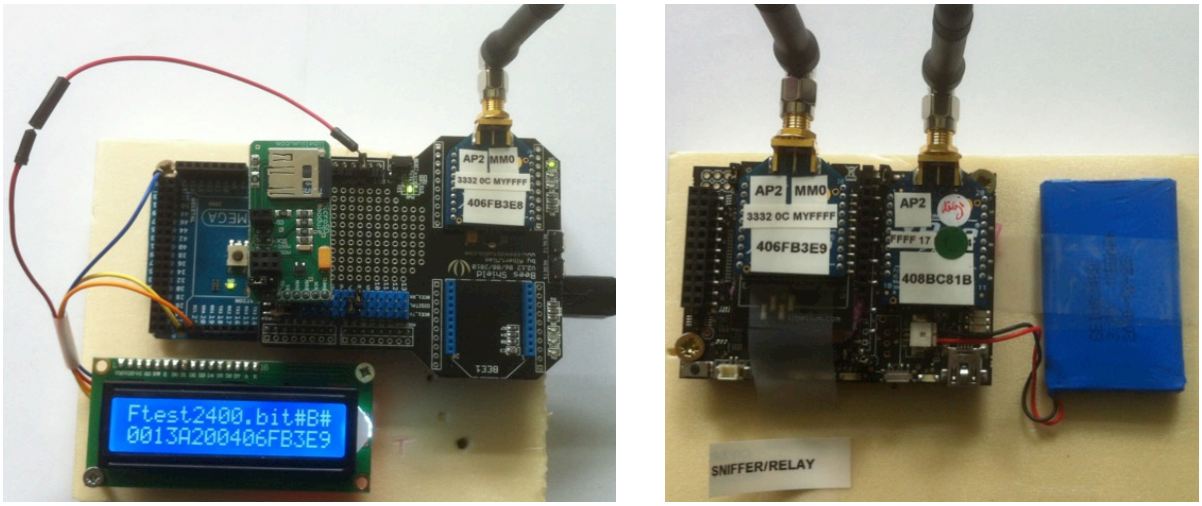


Figure 61: Left: Arduino Mega2560 for sending acoustic data stored in the SD card. Right: A WaspMote relay node

The Arduino board was used rather than a WaspMote because of its much higher flexibility regarding the hardware that could be connected to the board (LCD display, SD card,\ldots). The audio file will be transmitted in a number of packets according to the defined chunk size. When the sending is triggered, we can choose the time between 2 packet generation as well as the chunk size. We then have a number of relay nodes that are programmed to relay incoming packets to the sink which is, in our case, an XBee module connected to a Linux computer running the reception program to receive audio packets. Figure 61(right) shows the relay node based on WaspMote hardware that reproduces an IoT node of the Santander test-bed. Our test nodes have been deployed in the Santander test-bed at the location depicted in figure 62.

Figure 62: Test of acoustic data streaming, topology

We placed our nodes on the street lamps indicated in figure 62 at locations 392, 11, 12 and

29. The sender node is always on location 392 and location 11 always act as a relay. With 1 relay node, the receiver is at location 12 while with 2 relay nodes, location 12 will serve as a relay and the receiver is at location 29. The original IoT nodes of the Santander test-bed are placed on street lamp as shown in figure 63(left) We strapped our nodes as depicted by figure 63(right).

Figure 63: Test of acoustic data streaming, placement of test nodes

Tools

We developed a number of tools for the test-bed. First, the program that runs on the sender node can be dynamically configured to define the file to send, the destination address (64-bit broadcast or unicast address), the chunk size that will be used for fragmenting the file and the time between 2 packet generation. Second, the program that runs on a relay node can be dynamically configured to define the destination relay address and an additional relay delay, that will not be used in our tests here. Third, we developed a receiver program, called `XBeeReceive` that runs on a Linux machine and that will receive the incoming packets from a connected XBee gateway to either save them to a file or to redirect the binary flow to the standard output for streaming purposes. And fourth, a simple program, called `XBeeSendCmd` has been developed to send ASCII command strings to the various nodes for configuration purposes. It supports both 802.15.4 and DigiMesh firmware as well as provides the possibility to send remote AT command to configure the XBee radio module itself. A shell script can make successive calls to `XBeeSendCmd` to configure various test scenarios parameters as well as configuring each relay node with the right next-hop information. For instance, we have the `2relay-node.sh` script that takes 5 parameters, 4 MAC addresses (sender node, relay 1, relay 2 and receiver) and a file name, to configure a 2-hop scenario. We choose this solution rather than having a simple routing protocol because we wanted to have full control on the routing paths, allowing us to define multiple distinct paths if needed.

Audio codecs

Given the low receiver throughput shown in Figure 59, the choice of an audio codec is of prime importance. Codecs that are designed for audio music are not suitable and our choice clearly goes towards codecs used for digitized voice (telephony or VoIP). In this case, GSM codec that is used in mobile telephony system can be tractable (for the low rate version at about 6kbps) but we use instead an efficient open-source voice codec called `codec2` (http://codec2.org) that offers very low rates (1400, 1600, 2400 and 3200bps rates are available) while keeping a high voice quality and, most importantly, fully documented and implemented coding and decoding tools that can be used in streaming scenarios. The `codec2` package comes with the `c2enc` program that encodes an audio raw file into the codec2 format and the `c2dec` program that will decode a file into a raw format. We then use `play` and `sox` to play and to

convert the raw file into other format, if necessary, for play out in well-know players. Playing a codec2 file, `test2400.bit` in a streaming fashion can be realized as follows, assuming that the encoding rate is 2400bps:

(1) `cat test2400.bit | c2dec 2400 - - | play -r 8000 -s -2 -`

We use these tools with our `XBeeReceive` tool in the following way:

(2) `XBeeReceive -B -stdout test2400.bit | bfr -b1k -m10% - | c2dec 2400 - - | play --buffer 50 -t raw -r 8000 -s -2 -`

The command uses an intermediate playout buffer (`bfr` tool) to add more control on the data injection into the `c2dec` program. The -B and -stdout options of `XBeeReceive` are for indicating the binary mode and the redirection to standard output respectively. At the sending side, each packet carries the offset in the file (or flow for streaming mode) and missed data at the receiver are filled by a "neutral value" to enhance the play out quality. For the moment, the neutral value was empirically found to be 0x55 for 1400 bit rate, 0x77 for 2400 bit rate and 0x01 for 3200 bit rate. There are probably better values or better ways to enhance the play out quality with missed data but we leave this issue for future works.

We recorded an audio test file of about 13.2s (using a smart-phone for instance). An 8-bit PCM encoding scheme would give a bit more than 104000 bytes. We used `sox` to convert the recorded file into a 8-bit sample raw file at 8000Hz. Then with `c2enc` we produced codec2 files at 1400, 2400 and 3200bps. The file size are 2338, 4014 and 5352 bytes respectively. All these files can be downloaded in .wav format for immediate playout in most players at http://web.univ-pau.fr/~cpham/SmartSantanderSample/. These files are placed on the SD card of the Arduino sender node.

Results

We performed multi-hop transmissions with 1-relay node and 2-relay node configuration, see figure 62. Previous tests on the Santander test-bed showed that most of the IoT nodes deployed can reach their corresponding Meshlium gateway in a maximum of 2 intermediate hops. We then start the `XBeeReceive` command and issue send commands to the sender node by specifying the inter-packet time and the chunk size. After complete reception, we verified the audio quality by playing the received file with command (1) described above. We also tested the streaming version with command (2) described above.

Instead of using the maximum packet size that maximizes the throughput but makes the impact of any packet loss very harmful, we use smaller packet size that however provides at least the required throughput according to the encoding bit rate. For instance, if the packet size is 30 bytes and we need a throughput of 2400bps, then the maximum inter-packet time would be 30*8/2400=100ms. Figure 64 shows the maximum inter-packet time for various packet size and encoding rate. We also plot the total processing time depicted previously in Figure 58 to show which packet size in not compatible with a given inter-packet time. For instance, we can see that if the packet size in 20 bytes, the maximum inter-packet time for an 3200bps encoding is 50ms while the total processing w/relay at a relay node is 70. Therefore, it is expected that either the bit rate will not be met, or packets will build up in relay node buffer with high risk of packet drops.

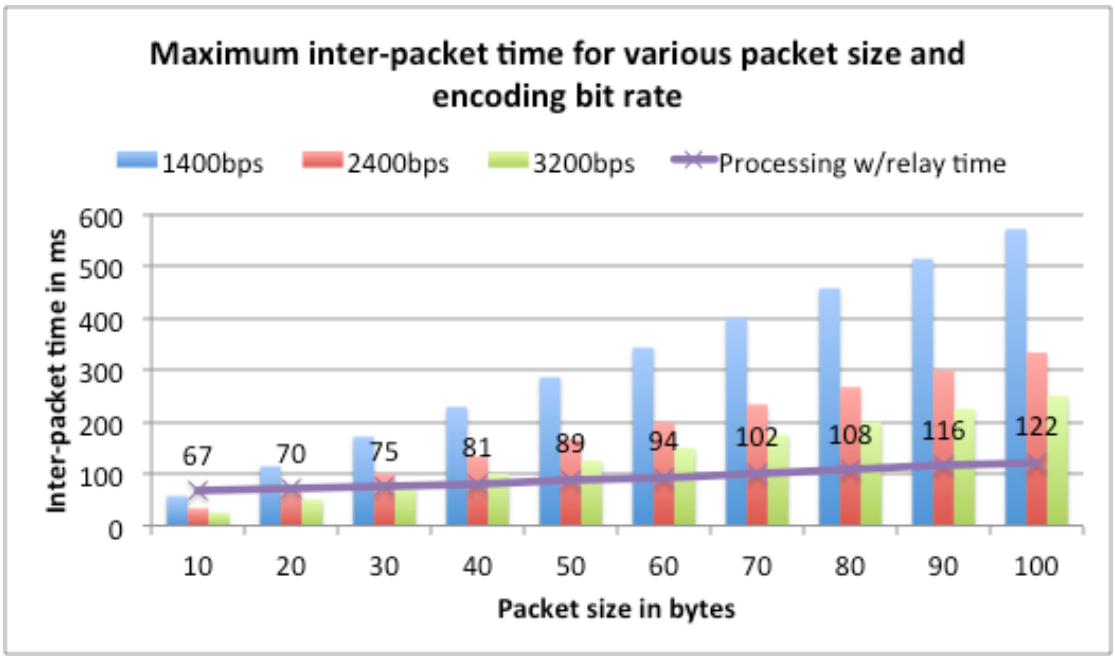


Figure 64: Maximum inter-packet time at various packet size and encoding rate

However, Figure 64 also shows that for all the considered bit rates, using a packet size greater or equal to 40 bytes is compatible with the maximum inter-packet time. For the tests we present in this paper we propose to use packet size of 40, 50 and 60 bytes. However, for 3200 encoding bit rate, it is not safe to use 40-byte packets in streaming mode since the maximum inter-packet time is 100ms to provide at least a throughput of 3200bps. We performed several tests to determine the inter-packet time for sending packet at the sender node that gives a correct delivery of the audio file. We found these inter-packet time to be 110ms, 120ms and 125ms for packet size of 40, 50 and 60 bytes respectively.

Table I below summarizes the 1-relay scenario results and indicates for each encoding bit rate and packet size the number of packets that are sent (n_{pkt}). We show the number of packet losses for inter-packet 110ms, 120ms and 125ms (t_{pkt}), but also reported the number of observed packet losses when using a smaller inter-packet time (i.e. 105ms, 110ms and 120ms). Reducing further the inter-packet time generates an overwhelming number of packet drops during our tests. We indicate the time needed for sending all the packets (t_s), the time for receiving the packets (t_{rcv}) and the time at which the play out begins in streaming mode (t_{play}). Once again, the received audio files can be downloaded in .wav format for immediate playout in most players at http://web.univ-pau.fr/~cpham/SmartSantanderSample. For the 2-relay node scenario, the results are summarized in Table II.

1-relay scenario									
bit rate	1400bps			2400bps			3200bps		
pkt size	40	50	60	40	50	60	40	50	60
n_{pkt}	59	47	39	101	81	67	134	108	90
t_{pkt}	105	110	120	105	110	120	105	110	120
n_{lost}	8	6	7	6	5	5	8	9	8
t_{pkt}	110	120	125	110	120	125	110	120	125
n_{lost}	1	0	0	0	2	2	3	1	3
t_s, s	6.5	5.6	4.8	11.1	9.7	8.3	14.7	14.4	11.2
t_{rcv}	6.9	6.4	5.2	11.6	10.1	8.8	15.4	15	11.7
t_{play}	4.7	4.5	3.7	8.4	8.2	6.1	13.1	12.8	9.8

TABLE I
1 RELAY NODE SCENARIO

2-relay scenario									
bit rate	1400bps			2400bps			3200bps		
pkt size	40	50	60	40	50	60	40	50	60
n_{pkt}	59	47	39	101	81	67	134	108	90
t_{pkt}	105	110	120	105	110	120	105	110	120
n_{lost}	9	7	7	7	7	7	8	8	10
t_{pkt}	110	120	125	110	120	125	110	120	125
n_{lost}	2	1	1	0	1	2	2	1	2
t_s, s	6.4	5.6	4.9	11.2	9.8	8.3	14.6	14.4	11.3
t_{rcv}	7.1	6.6	5.3	11.8	10.2	9	15.7	15.2	12
t_{play}	4.9	4.8	3.9	8.7	8.5	6.4	13.3	13	10.1

TABLE II
2 RELAY NODE SCENARIO

Some pictures of the test campaign

7. SmartSantander important MAC parameters

Reliability issue with XBee 802.15.4 and DigiMesh module

Reliability can be realized at MAC layer with MAC layer acknowledgement (ACK) mechanism. The XBee 802.15.4 module has 4 MAC mode (controlled with the AT+MM command) and transmitted packets can be acknowledged in mode 0 (Digi Mode) and mode 2 (pure 802.15.4 mode with ACKs). Note that ACKs are possible only in unicast mode where the destination address is a specific MAC address (different from 0x000000000000FFFF). The MAC mode is taken into account at the transmitting device : if the sender XBee module uses MAC mode 2 and the receiver XBee module uses MAC mode 1, then the receiver XBee module MAC layer will still issue the ACK. The DigiMesh module has no option for alternative MAC mode and unicast traffic always has the acknowledgment requirement.

When ACK is required (unicast communication) the IEEE 802.15.4 standard stipulates that the transmission of an acknowledgment frame (in a non-beacon enabled network) commences *aTurnaroundTime* symbols after the reception of the data frame, where *aTurnaroundTime* is equal to 192 µs. Regarding the transmitting node, it has to wait *macAckWaitDuration* symbol periods for an acknowledgment before it attempts a retry, where *macAckWaitDuration* is equal to 54 symbol periods (0.864 ms).

The XBee module is compliant with this behavior as described in [XBeeDigi]

> *If the transmission is not a broadcast message, the module will expect to receive an acknowledgement from the destination node. If an acknowledgement is not received, the packet will be resent up to 3 more times. If the acknowledgement is not received after all transmissions, an ACK failure is recorded.*

The XBee module therefore has the 3 retries as stipulated by the 802.15.4 standard. Additional retries can be controlled by the "XBee Retries" parameter (or "Unicast MAC Retries" for the DigiMesh firmware) that can be controlled with the AT+RR command. By default, RR is 0 so only the 3 802.15.4 retries are used. An RR value of 1 would add 3 more retries at the MAC level. This will certainly add latency but can reduce transmit error at the application level in high loaded environment.

With the XBee module, the response from the radio module completly takes into account the possible retransmissions. We have not notice significant additional overheads due to enabling ACK or possible retransmissions given the time scale of the communication stack with the default value of RR=0.

For the DigiMesh firmware, broadcast traffic has some kind of reliability mechanism referred to as "Broadcast multi-transmit" controlled by the AT+MT parameter. MT is 3 by default and this is one reason why sending broadcast traffic is more costly than unicast because [All broadcast packets are transmitted MT+1 times to ensure it is received]. Note that [Broadcast transmissions will be received and repeated by all routers in the network] so one has to use broadcast carefully with DigiMesh firmware. The "Broadcast Radius" parameter can however be controlled by the AT+BH command but the default value is 0 which indicate the maximum radius value. Additionally, [In order to avoid RF packet collisions, a random delay is inserted before each router relays the broadcast message]. This is determined by the "Network Delay Slot" parameter controlled by the AT+NN command. NN has the default value of 3 and one network delay slot is approximately 13ms.

Again, with DigiMesh firmware, as there is a native on-demand routing mechanism, unicast transmissions must use a so-called maximum "Network Hops" parameter, controlled by the AT+NH command, to calculate the timeout value for acknowledgement failure. If the network

maximum number of hops is greater than 7, which is the default value of NH, then one has to increase this value at the cost of delaying automatic retransmissions of lost packets.

Channel access time

As seen in the 802.15.4 review section, each time a device needs to transmit, it waits for a random number of unit back-off periods in the range {0, 2^{BE}-1} before performing the Clear Channel Assessment (CCA). The BE exponent is set to *macMinBE* in the 802.15.4 standard and its default value is 3.

With the XBee module, this value is referred to as the "Random Delay Slot" parameter and can be controlled with the AT+RN command. By default, the value is 0 which reduce the initial backoff time to the LIFS as seen previously. However, in high loaded network, a value of 0 can have a dramatic impact of performances as channel contentions are more likely to happen.

8. HobNet network qualification

TinyOS and 802.15.4 radio support

TelosB motes (AdvanticSys motes are mainly TelosB) run under the TinyOS system [TINYOS]. The last version of TinyOS is 2.1.2 and our tests use this version.

The default TinyOS configuration use a MAC protocol that is compatible with the 802.15.4 MAC (Low Power Listening features are disabled). The default TinyOS configuration also uses ActiveMessage (AM) paradigm to communicate and interoperable frames (IFRAME) are used to allow interoperability with non-TinyOS network.

Therefore 2 bytes in the payload are reserved for a network identifier (1 byte) and for an ActiveMessage identifier (1 byte). By default, TinyOS proposes the TINYOS-6LOWPAN network identifier which has value 0x3F (see TEP125 and TEP126 of TinyOS). Then an ActiveMessage identifier can be used to identify the source application, this value being set by the user. As the standard 802.15.4 frame can allow more than 102 bytes, it is still possible to have a user payload of 100 bytes for comparison purposes with XBee radio module (remember that TelosB nodes have a CC2420 radio module).

TinyOS a component-based and event-driven operating system that has more elaborated control than Arduino or Libelium systems. One main difference resides in the sending process where a packet send may be posted by an application (send request) and the system will issue a `sendDone` event when the send is completed. A busy flag should be used to indicate that a sending is undergoing. This flag should then be released when the `sendDone` event is process. Nevertheless, we can use the same methodology than for the qualification of WaspMote and Arduino boards: we will measure the time between the send post and the `sendDone` event as the "time in send()". The "time between 2 packet generation" will also measure the minimum time between 2 send.

Synthetic workload with Traffic Generator, sending side – 1 hop

Figure 65 shows at the sender side the "time in send()" when the payload size in varied.

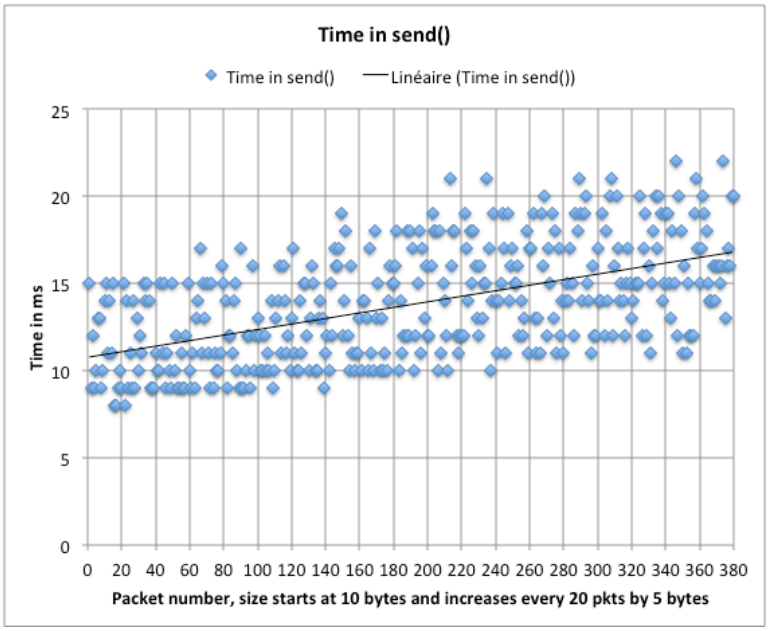


Figure 65: time in send() for various payload

In these tests, 380 packets are sent. The initial packet size is 10 bytes and increases by 5 bytes every 20 packets. Therefore, the x-axis on Figure 65 is the packet sequence number. We can see that as opposed to the WaspMote architecture, the time in send() is not constant but can vary from one call to another. However, it is possible to see a general tendency as the payload increases. In addition, as we are mainly interested in streaming capability of sensor motes, it is possible and reasonable to consider the mean "time in send()" as the standard deviation is small. Figure 66 shows the "time between 2 packet generation" and the same remarks can be applied. For TinyOS, the "time between 2 packet generation" takes into account the various overheads of task scheduling.

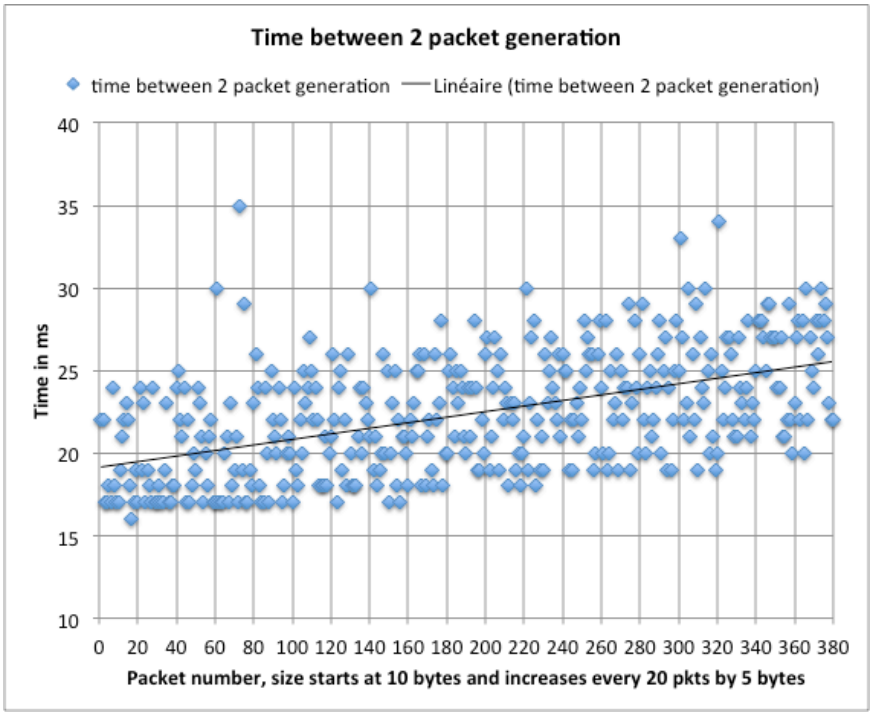


Figure 66: time between 2 packet generation for various payload

Figure 67 therefore shows the mean "time in send()" and the mean "time between 2 packet generation" as the payload is varied. Linear interpolation curves (fitted curves) are also shown.

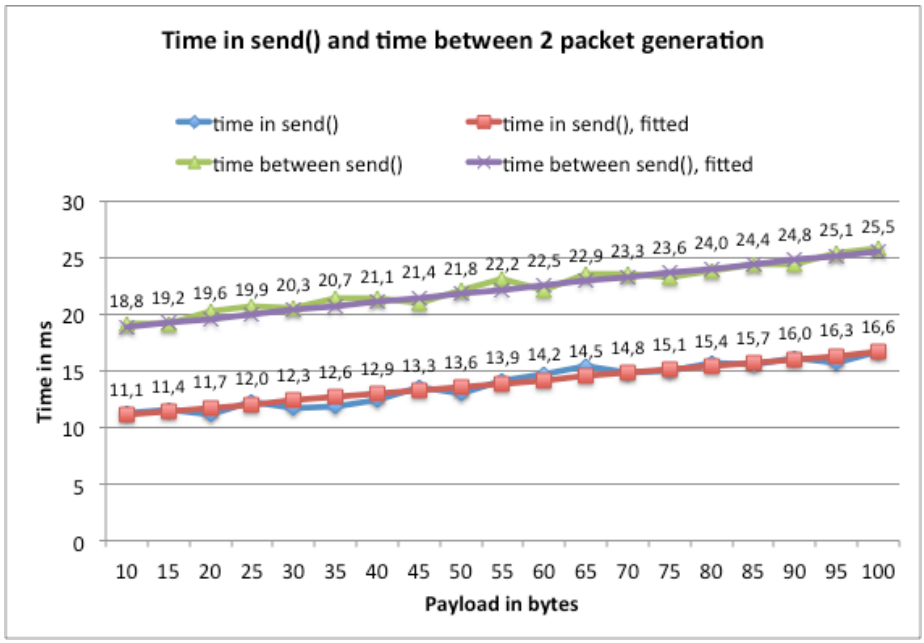


Figure 67: mean time in send() and mean time between 2 packet generation for various payload

Then, Figure 68 shows the corresponding maximum throughput at the sender side. Once again, we differenciate the maximum sending throughput case when only the time in send() is considered from the case the time between 2 packet generation is used, which is a more realistic scenario. If we compare with Figure 56, we can see that the maximum realistic sending throughput is much greater than for the Arduino (which had the best realistic sending throughout for UART-based platforms).

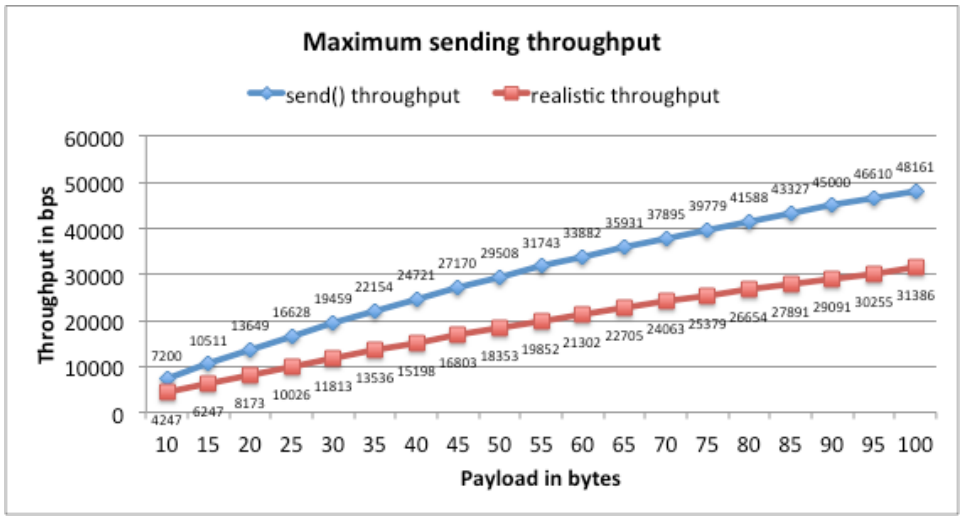


Figure 68: maximum sending throughput for various payload

Synthetic workload with Traffic Generator, receiver side – 1 hop

If there is no bottleneck, the receiver throughput should be close to the sender throughput. We saw previously that WaspMote (and Arduino) boards do have a read time overhead (see Figure 58) that greatly limits the reception throughout. Here, we verified that at the receiver side does not encounter to many packet drops under the minimum "time between 2 packet generation" overhead at the sending side. We measured the receiver throughput by sending 100 packets of a given size (from 10 bytes to 100 bytes with 5 bytes increment) and by computing the real receiver throughput at each size change. Figure 69 compares the maximum realistic sending throughput and the measured receiver throughput.

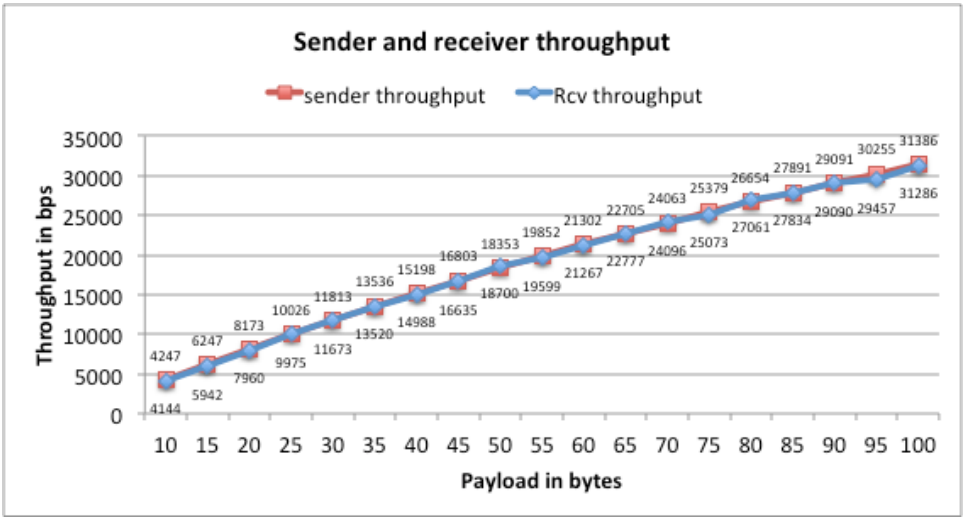


Figure 69: maximum sending throughput for various payload

We can see that the measured receiver throughput is very close to the maximum realistic sending throughput. From these results we can say that (i) our methodology which consisted in computing the mean "time between 2 packet generation" to compute a maximum sending

throughput is reasonable, and (ii) the receiver throughput is limited here by the sender throughput: if the sender can send faster, it is most likely that the receiver could increase its throughput.

Multi-hop issues

In a multi-hop transmission, a relay node will need to first read to incoming packet and then forward it to the next node by sending the newly received packet. As previously with WaspMote board, we will not consider the cost of finding routes. Under TinyOS, a `receive` event will be sent to the application when a packet has been received and ready for the application. The time at which the packet was received by the communication module can be known and the time at which the `receive` event is sent can also be known. Therefore, the time difference can be interpreted as the time needed for the operating system to receive physically the packets and to performed the required I/O bus and memory operation before making the packet available. This time is referred to as why t_{read} like previously. Figure 70 shows t_{read} as the payload is varied (blue curve) and also plots the time needed to forward a packet by considering that relaying 1 packet means read the packet (t_{read}) + send it (the previously measured time between 2 packet generation). This is shown with the red curve as "Packet relay time (th)". We also experimentally measured the forwarding/relaying time by taking the time difference between the time at which the `sendDone` event is notified (forwarding has been done) and the time at which the packet was received by the communication module. This is shown with the green curve. We can see that the red and green curves are very close each other. Labels above the red curve correspond to the red curve.

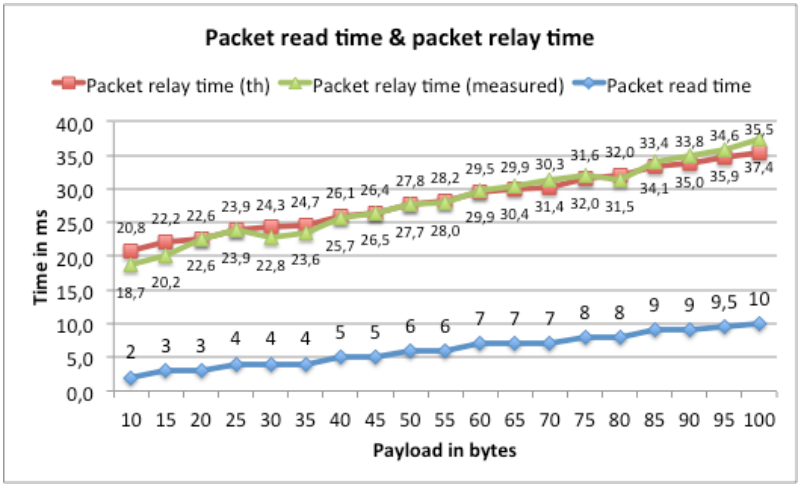


Figure 70: packet read time and packet relay time

Figure 71 shows the distribution of the forwarding time as measured by our experiments and used to compute a mean forwarding time for figure 70 above.

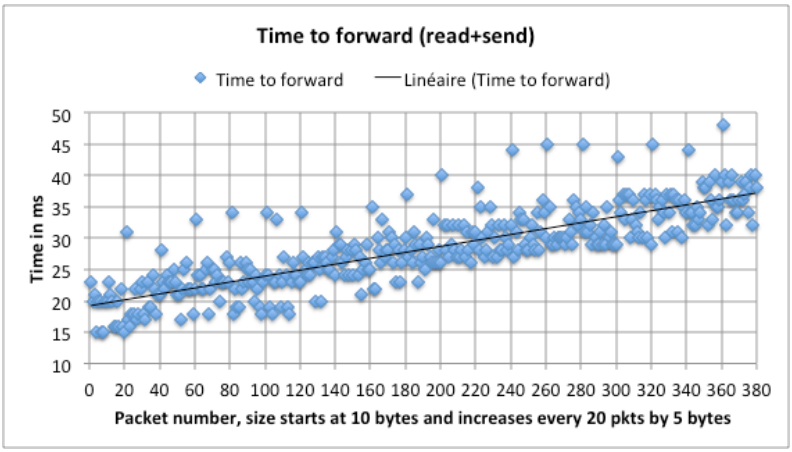


Figure 71: packet read time and packet relay time

Like previously for figure 66, the experiment consisted in sending 380 packets by a Traffic Generator. These packets are received at a relay node that forwards it. The initial packet size is 10 bytes and increases by 5 bytes every 20 packets at the sender side. Therefore, the x-axis on Figure 71 is the packet sequence number.

Figure 72 shows the relay throughput at an intermediate node derived from the forward/relay time. Comparison with sender and receiver throughput is provided. We can see that the AdvanticSys platform has much better communication performances than Libelium WaspMote.

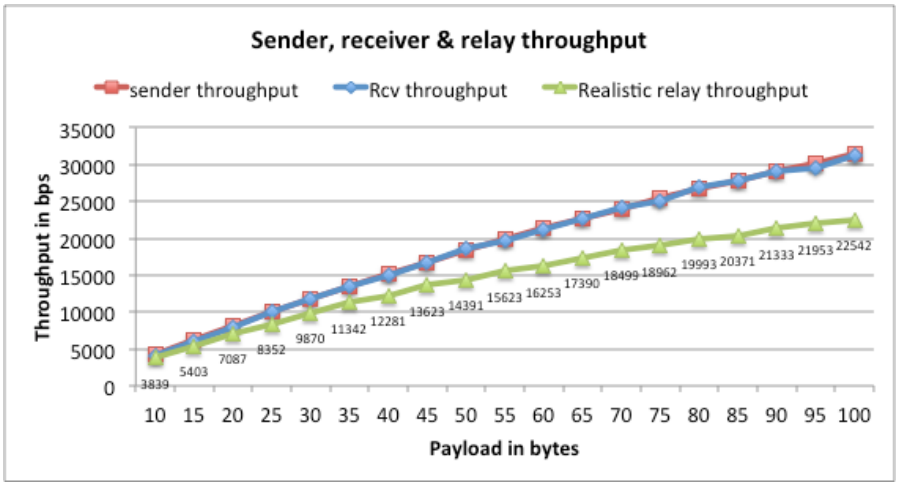


Figure 72: Realistic relay throughput

Preliminary tests of audio streaming with Advanticsys motes

Using the same sender node than in Section 6, we tested with Advanticsys node as relay nodes and found that the inter-packet time can be reduced to about 60ms (compared to about 110-120ms) which greatly reduce the time needed for transferring the acoustic data in a multi-hop manner. Further detailed experimentations are planned.

IP protocol stack on Advanticsys

BLIP/6LowPan

TinyOS proposes a support for IPv6 and emerging protocols for the Internet of Things. The IPv6 support is provided through the BLIP communication stack (Berkeley Low-power IP stack). The last version of TinyOS uses BLIP v2.0. The IPv6 support is realized by implementing the 6LowPan header compression techniques and mainly using UDP on top of the 6LowPan compression. UDP header can also be compressed. Figure 73 from [T6LOW] shows the various overheads for using IPv6 technologies that most of the time needs 7 bytes to be taken from the user payload of the 802.15.4 frame. In addition to the 7 bytes overhead, the IPv6 support provides by BLIP adds a processing overhead that has been studied in [KO11]. Further investigation is needed to determine whether UDP usage for streaming acoustic traffic is feasible or not on these type of platform.

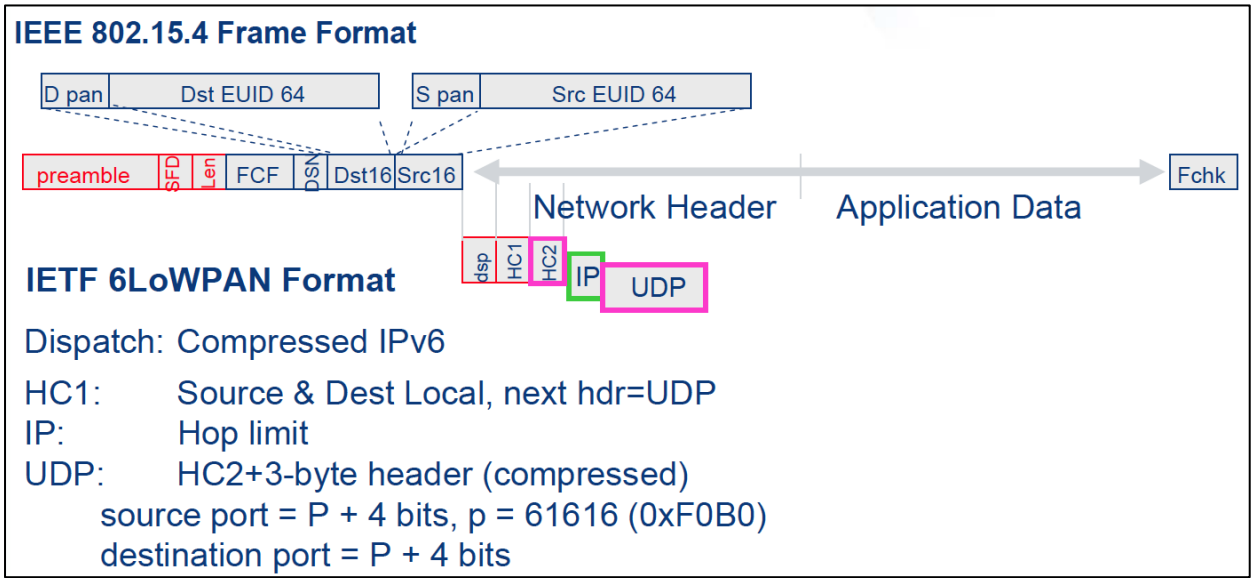


Figure 73: IPv6 (6LowPan)+ UDP overheads

CoAP/BLIP/6LowPan

TinyOS also support the CoAP protocol (Constrained Application Protocol) with the CoapBlip implementation that allows queries to be simply made between heterogeneous devices. CoAP can be been as an HHTP-equivalent protocol for resource-constrained devices. As it is intended for machines, it is often referred to as the HTTP protocol for the Internet of Things (or Internet of Machines or Machine-2-Machine). Adding CoAP feature is highly relevant for increasing portability and interoperability, but this is done at the cost of a much higher overhead, see [KUL11], that is hardly compatible with streaming applications. Note that recently a new implementation of CoAP for TinyOS, referred to as TinyCoAP [LUD13], has been proposed that offers slightly improved performances compared to CoapBlip. However, GET or PUT operations are still in the order of 200ms leaving these CoAP features for query-oriented applications.

9. Conclusions

This document decribed the first phase of the SmartSantander and HobNet network qualification process which consists in the following tasks:

1. Qualification of the Libelium WaspMote and Advanticsys platforms: throughput, latency, reliability level and loss rate,
    a. Radio module to radio module
    b. Microcontroller to radio module
    c. Impact of software APIs
2. Identification & tests of important radio module parameters (radio and Medium Access Control level) and their impact on performances in a networked environment
    a. 1-hop and 2-hops
    b. prediction for k>2

The main objective of this study is to determine the upper bounds on the performances that one could get from the SmartSantander and HobNet infrastructure (both hardware and software considerations) as a premiminary step towards support of audio traffic for the EAR-IT project.

Regarding the SmartSantander test-bed at Santander, we have seen that using the light Libelium API (or the full Libelium API with 16-bit address) can provide a maximum realistic sending throughput of about 17100 bps with 802.15.4 modules. DigiMesh firmware could not reach this performance level as the DigiMesh maximum payload is greatly reduced. When taking the reception side, a maximum of about 12000bps could be achieved without error.

We can foreseen for audio traffic that the fragmentation and reassembly support is not really necessary compared to the need of higher throughput. Therefore, audio traffic will most likely be handled in frames of maximum size of 100 bytes. In this case, although it is beyond the scope of this document, is it possible to greatly improve the performances of the Libelium API to have the same level of performance than Arduino boards (almost 24000bps) with the lightweight communication API for XBee modules.

One major bottleneck of the SmartSantander network is the usage of the default 34800 baud rate defined by the Libelium API. We showed that due to clock constraints, using higher standard baud rate is not tractable but it is possible to use custom baud rates using well-chosen custom divisor that can offer up to 250000bps for data transfer between the microcontroller and the XBee radio module. This possibility greatly increases the performance of the network as shown by our predictions and experimental measures using 250000 baud rate: almost 38000bps for the maximum sending throughput.

These preliminary results are quite promising regarding the possibilities of sending audio traffic on the SmartSantander network. With the appropriate improvements (API and higher baud rate) streaming audio traffic is not out of reach.

Now for the HobNet test-bed, we found that AdvanticSys node based on a TelosB mote architecture have a much higher level of performance, resulting in faster sending rate and forwarding capabilities, than the WaspMote board. If AdvanticsSys nodes were used, the constraints on the audio encoding techniques are a bit smaller as one can easily achieve a throughput of about 14000bps with 50-bytes packets.

10.References

[802154] IEEE Std 802.15.4™-2006.

[ADVAN] http://www.advanticsys.com/shop/wireless-sensor-networks-802154-mote-modules-c-7_3.html

[CC2420] ChipCon CC2420, 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. www.ti.com/lit/ds/symlink/cc2420.pdf

[DMDigi] XBee®/XBee-PRO® DigiMesh RF Modules product manual (90000991_E), Digi International Inc. January 6, 2012

[FOS11] John Foster. XBee Cookbook. 2011

[JENNIC] Application Note: JN-AN-1035. Calculating 802.15.4 Data Rates. Jennic.

[KO11] JeongGil Ko et al. Evaluating the Performance of RPL and 6LoWPAN in TinyOS. ISPN'2011.

[KUL11] Koojana Kuladinithi et al. Implementation of CoAP and its Application in Transport Logistics. ISPN'2011.

[LAT06] Benoît Latré, Pieter De Mil, Ingrid Moerman, Bart Dhoedt and Piet Demeester. Throughput and Delay Analysis of Unslotted IEEE 802.15.4. Journal of Networks, vol. 1(1), may 2006.

[LI01] Jinyang Li, Charles Blake, Douglas S. J. De Couto, Hu Imm Lee, Robert Morris. Capacity of Ad Hoc Wireless Networks. ACM MobiCom 2001.

[LUD13] Ludovici, A.; Moreno, P.; Calveras, A. TinyCoAP: A Novel Constrained Application Protocol (CoAP) Implementation for Embedding RESTful Web Services in Wireless Sensor Networks Based on TinyOS. J. Sens. Actuator Netw. 2013, 2, 288-315.

[MAO11] Guoqiang Mao. The Maximum Throughput of A Wireless Multi-Hop Path. Journal Mobile Networks and Applications, Volume 16 Issue 1, February 2011, Pages 46-57.

[MOR10] Eduardo Morgado, Inmaculada Mora-Jiménez, Juan J. Vinagre, Javier Ramos, and Antonio J. Caamaño. End-to-End Average BER in Multihop Wireless Networks over Fading Channels. IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, VOL. 9, NO. 8, AUGUST 2010.

[TELOSB] www.willow.co.uk/html/telosb_mote_platform.html and/or http://bullseye.xbow.com:81/Products/productdetails.aspx?sid=252

[T6LOW] D. Culler and J. Hui. 6LowPan tutorial. ArchRock.

[TINYOS] The TinyOS operating system. http://www.tinyos.net/

[SUN06] Tony Sun, Ling-Jyh Chen,Chih-Chieh Han, Guang Yang, and Mario Gerla. Measuring Effective Capacity of IEEE 802.15.4 Beaconless Mode. IEEE WCNC 2006.

[WASP] WaspMote technical guide, Libelium. Document version: v3.4 - 11/2012

[WASP802] Waspmote 802.15.4 Networking Guide, Libelium. Document Version: v0.4 - 07/2010

[XBeeDigi] XBee®/XBee-PRO® RF Modules product manual (90000982_G), Digi International Inc. August 1, 2012.

[XCTU] X-CTU Configuration & Test Utility Software User Guide, Digi International Inc.

11.Annex: Status of 6lowpan and CoAP protocols

EAR-IT projects aims to conduct large-scale 'real-life' experimentation of intelligent acoustics for supporting high social value applications fostering innovation and sustainability.

All the defined application scenarios and use cases rely on the capacity to fully benefit from the services provided by the wireless sensor network (WSN). Applications may require combined inputs from multiple sensors, or advanced processing capabilities that cannot be provided by standard IoT motes[3]. As a result, important amount of data need to be exchanged between nodes and transported by the underlying network:

- Raw audio data from IoT sensors to Acoustic Processing Units (APU);
- Aggregation information from APUs to the management center;
- Aggregation information and commands to be sent to actuators;
- …

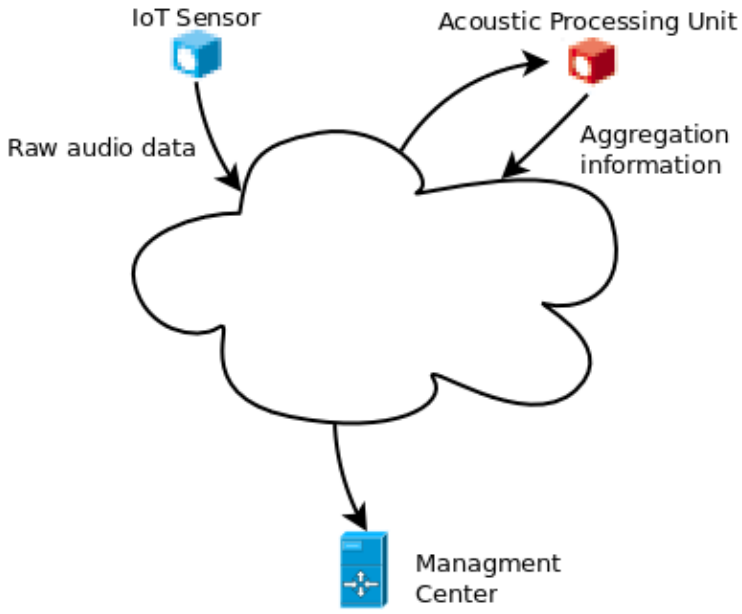


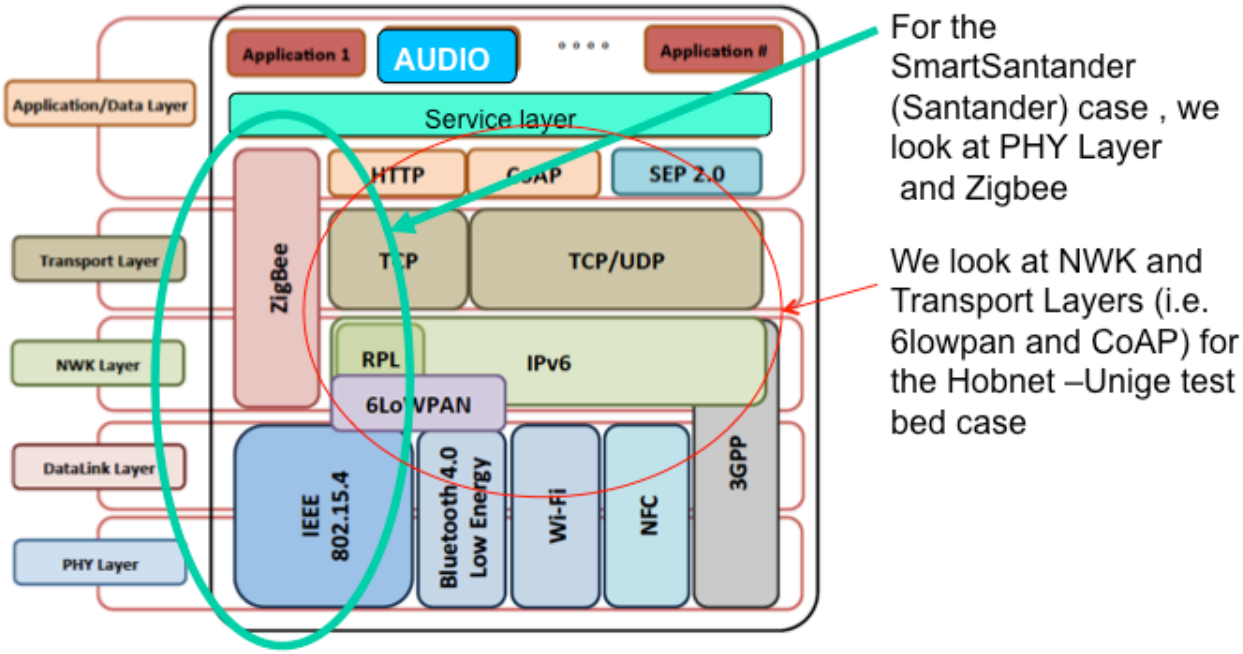
Figure 1: Data flow inside WSN

Besides the physical capacity of the network to transport such data, it is important to ensure that the information is correctly and efficiently routed across the network. Due to the nature of the WSN itself (low-power environment, lossy context, topology changes caused by sleeping nodes …), traditional Internet-related protocol (HTTP, IPv4/IPv6 …) cannot be used directly: most of them require important resources (power, memory, CPU, bandwidth) that are not available in such constrained environments.

Alternative protocols or adaptation layers have been developed or are still under development with aim to target wireless sensor networks and address all these issues. However, these protocols are still "young" and not fully validated. They have not yet reached the maturity level of traditional Internet protocol, and both implementation devices and procedures need to be evaluated and tested for conformance and interoperability.

Figure 2: Overview of WSN-related protocols

[3] *IoT objects typically have energy constraints that impact their processing capabilities*



The picture above shows some simplify view of layers and stack of an Internet of things. The previous chapters described the network qualification in the Santander test bed with 802.15.4 and Zigbee. Only Hobnet at the Unige test bed uses (new) protocols such as 6lowpan and CoAP. Chapter 8 described the impact on transport network and some aspect of 6lowpan and CoAP only from the audio transport point of view, which is the main interest of EAR-IT and the WP1.

However as experts are also in contact, through other side activities and project (e.g. FP7 Probe-IT project), this annex gives an overview of the status of maturity of 6lowpan and CoAP through interaction with IETF WG and interoperability activities organised.

This annex focus on two of protocols which can be qualified " new":

- 6LoWPAN, providing mechanisms to adapt IPv6 to WSNs;
- CoAP, representing a good alternative to HTTP in constrained environments.

Evaluation of "transport" protocols

6LoWPAN

Overview

IPv6 over Low-power Wireless Personal Area Networks (6LoWPAN) is the name of an IETF working group formed in 2004 with an aim of enabling IPv6 communication over low power radio such as IEEE 802.15.4. The main output of this group is RFC 4944, defining encapsulation and header compression rules aiming to allow transmission of IPv6 packets over IEEE 802.15.4 based networks. It is important to note that, being an IETF standard, 6LoWPAN is open, accessible and reliable.

6LoWPAN combine two powerful features: the powerful communication that IPv6 contribute to Internet and the low power requirements of IEEE 802.15.4 that gives the possibilities to connect wireless sensor networks to the Internet.

Adaptation of IPv6 to IEEE 802.15.4 networks is indeed a challenging task and requires following problems to be addressed carefully:

- Fragmentation and reassembly.

By design, IPv6 requires the maximum transmission unit (MTU) to be 1280 octets. On the other side, IEEE 802.15.4 offers a standard packet size of 127 octets. These constraints would lead to excessive fragmentation and reassembly issues.

- IPv6 header compression.
  IPv6 standard header is 40 octets long, and can optionally be increased by extension headers. This represents a huge overhead compared to the packet size of IEEE 802.15.4, and requires IPv6 headers to be compressed in order to optimise data transfer on a 6LoWPAN network.
- Routing.
  Wireless sensor networks are typically composed of multiple nodes organised in star topology or in mesh topology. Specific routing mechanisms taking into account these topologies and the nature of the nodes (power consumption, memory and processing limitations ...) need to be developed to improve network capabilities.

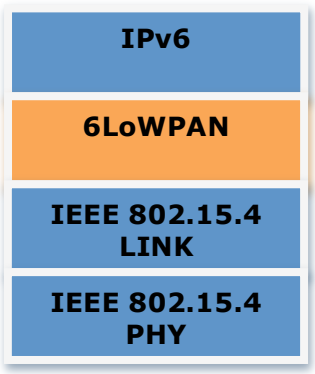


Figure 3: 6LoWPAN protocol stack

6LoWPAN Network Architecture

The 6LoWPAN architecture is made up of low-power wireless area networks (LoWPANs), which uses a set of IPv6 independent networks to connect each other. The typical 6LoWPAN architecture is presented in Figure 4.
Three different kinds of LoWPANs have been defined:

- Simple LoWPANs;
- Extended LoWPANs;
- Ad hoc LoWPANs.

A LoWPAN has a set of 6LoWPAN nodes, which share a common IPv6 address prefix (the first 64 bits of an IPv6 address), meaning that regardless of where a node is in a LoWPAN its IPv6 address remains the same.
An Ad hoc LoWPAN is not connected to the Internet, but instead operates without an infrastructure.
A Simple LoWPAN is connected through one LoWPAN Edge Router to another IP network.
An Extended LoWPAN has set of 6LowPAn nodes with more than one edge along with a backbone link (e.g. Ethernet) interconnecting them. Edge routers plays a vital role in 6LowPAN architecture by providing IPv4 interconnectivity, Neighbor discovery etc., and 6LowPAN nodes plays a major role of router or host with one or more edge router.

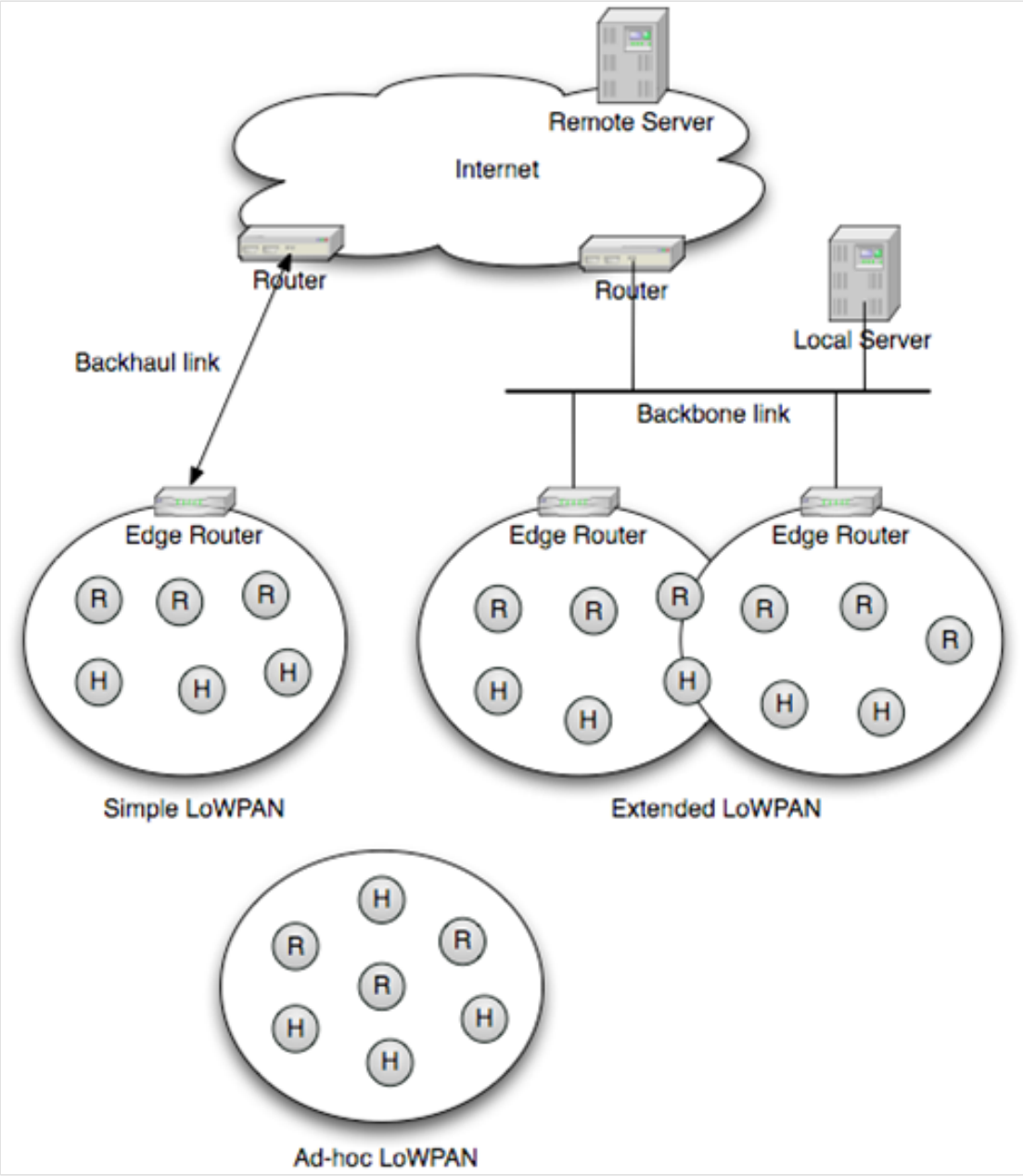


Figure 4: 6LoWPAN network architecture

CoAP

Overview

Constrained Application Protocol (CoAP) is an application layer protocol developed by the CoRE[4] working group of IETF. It is aiming to be used as a substitute to HTTP by simple, resource constrained, electronic devices typically found in wireless sensor networks such as low power sensors, switches, valves...

Due to its design, including very low overhead and simplicity, it is particularly popular in machine to machine (M2M) applications such as smart energy, building automation, smart cities, etc. CoAP can be used on most devices supporting UDP and its typical protocol stack is

[4] *Constrained RESTful Environments*

described in Figure 5.

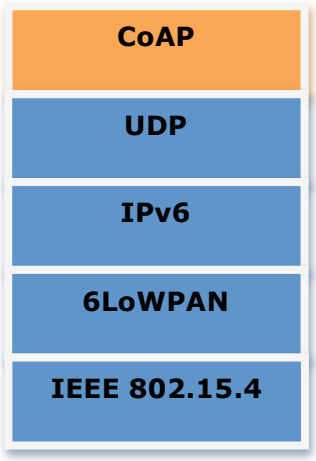


Figure 5: Typical CoAP protocol stack

- RESTful design guarantying easy mapping to HTTP and allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way.
- Low header overhead.
  CoAP base header length ranges from 4 to 12 bytes.
- Low parsing complexity.
  CoAP headers and options are all binary encoded and do not require any heavy encoder/decoder.
- Support for URIs and Content-Type.
- Resource discovery.
  Using CoRE Link format, it is possible to easily discover resources offered by CoAP servers.
- Support for resource subscription.
  The Observe feature allows CoAP clients to register to resources on CoAP servers and to be informed when these resources are modified
- Caching mechanism.
  In order to optimize power consumption, CoAP provides mechanisms to allow data caching among nodes
- Support for fragmentation.
  CoAP provides a block-wise transfer mode, allowing transmission of large data over several CoAP messages

It is important to note that at the time the present document is written, CoAP is still under development, although most of its features are considered as stable.

Protocol details

CoAP architecture is built around the web service paradigm. A client requests an action to be performed on a resource identified by a URI on a server, using a method code defined below:

- GET
  This method is used to retrieve the content of the resource identified by the URI contained in the request.
- POST
  This method requests the server to process the resource representation transmitted in the request. This processing is server specific and may result in resource to be created, modified or deleted.

- PUT
  This method requires the resource designated in the request to be created or updated on the server.
- DELETE
  This method is used to request the deletion of a server resource.

In the context of M2M interactions, CoAP implementations would typically act as both server and client. Communication between client and server is based on the request/response model, using the four message types offered by the protocol:

- Confirmable messages (CON), requiring an acknowledgment message to be sent by the recipient;
- Non-Confirmable messages (NON), which are not to be acknowledged by the recipient;
- Acknowledgment (ACK), used to inform the correspondent of the good reception of a message;
- Reset (RST), indicating that a received message cannot be processed properly.

Importance of Testing

The objective of interoperability testing is that independent implementations of the same standard interoperate. It is a well-known fact that, even following the same standard, two different implementations might not be interoperable. The heterogonous nature of IoT technologies requires interoperability issues to be solved before the deployment of the product. Having a simple view on interoperability for different technologies may not be possible at this point of time. Since it is a complex topic and needs more research activities to face the challenges raised. To efficiently address this problem, it is necessary to see the interoperability addressing all components within the complete development chain (standards, products, tests, tests tool, etc) with different tools

Interoperability testing events are important and pragmatic tools to validate standards improve implementation and finally improve interoperability.

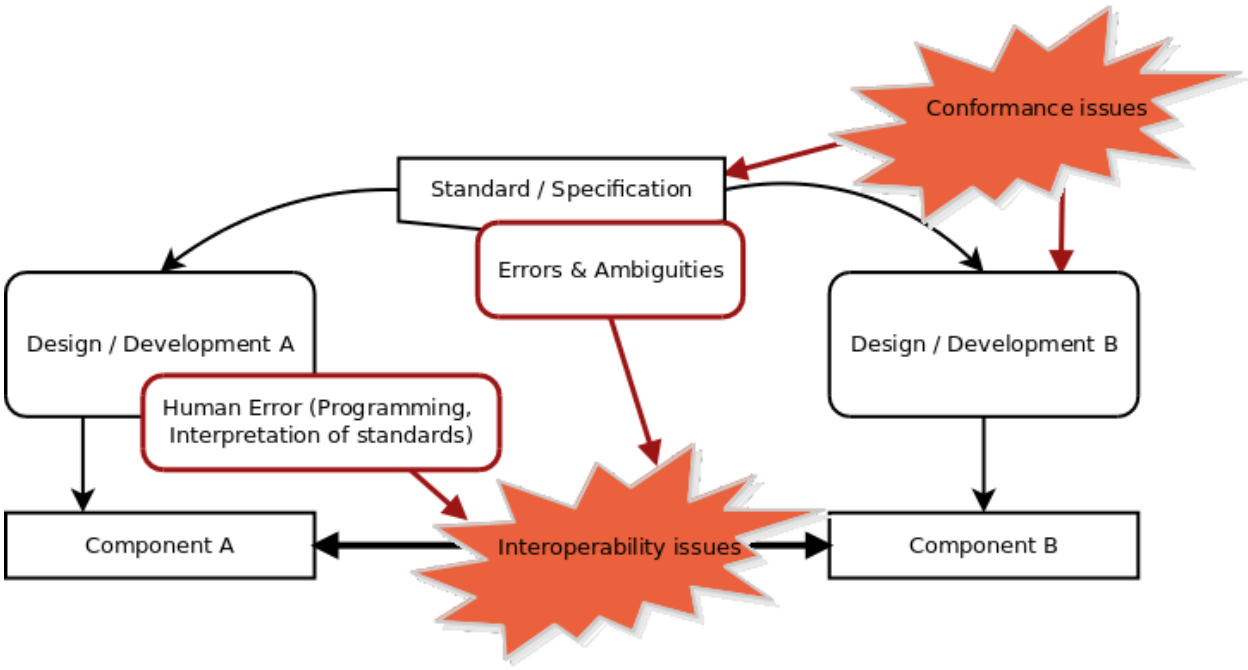


Figure 6: Need for testing

Main stakeholders 6LoWPAN testing

The Internet Engineering Task Force (IETF) is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. It is open to any interested individual. The IETF Mission Statement is documented in RFC 3935. The 6lowpan Working Group has completed two RFCs: "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals" (RFC4919) that documents and discusses the problem space and "Transmission of IPv6 Packets over IEEE 802.15.4 Networks" (RFC4944) which defines the format for the adaptation between IPv6 and 802.15.4.
The Working Group will generate the necessary documents to ensure interoperable implementations of 6LoWPAN networks and will define the necessary security and management protocols and constructs for building 6LoWPAN networks, paying particular attention to protocols already available. 6LoWPAN will work closely with the Routing Over Low power and Lossy networks (roll) working group which is developing IPv6 routing solutions for low power and lossy networks (LLNs).

The IPSO Alliance is a global non-profit organization serving the various communities seeking to establish the Internet Protocol as the network for the connection of Smart Objects by providing coordinated marketing efforts available to the general public. Our purpose is to provide a foundation for industry growth through building stronger relationships, fostering awareness, providing education, promoting the industry, generating research, and creating a better understanding of IP and its role in connecting Smart Objects.

Main stakeholders in CoAP testing

ETSI is an independent, non-for-profit organization whose mission is to produce Information and Communication Technologies standards. ETSI unites nearly 700 Members from five continents, and brings together manufacturers, network operators, service providers, administrations, regulators, research bodies and users – providing a forum in which all the key players can contribute. The ETSI TC M2M is currently examining how CoAP and the RESTful interfaces can be used to define horizontal Service Capabilities that can support multiple M2M applications over multiple core and access networks

The IPSO Alliance is the primary advocate for IP for smart objects for use in energy, consumer, healthcare and industrial applications. The Alliance, a non-profit organization whose members include leading technology, communications and energy companies, is providing the foundation for a network that will allow any sensor-enabled physical object to communicate to another as individuals do over the Internet. The IPSO Alliance membership is open to any organization supporting an IP-based approach to connecting smart objects.

Conformance Testing

Conformance testing (also referred to as compliance testing) is defined as the process of assessing the extent to which an implementation of a given protocol entity follows the requirement stated in the associated specification documents (e.g. standard specifications). While conformance testing mostly uses the black-box testing technique, whereby the implementation under test (IUT) would be considered only from its externally observable behaviour, other techniques such as white-box or grey-box testing may also be applied, depending on the requirements stated by the base specifications.
Conformance testing is generally viewed as a facilitator for interoperability, based on the idea that if the base specification is correct and all implementations thereof strictly align to it, then

the mechanisms it defines to ensure interoperability among different vendor implementations will work as expected and interoperability will be the logical consequence.

ISO/IEC Conformance Testing Methodology Framework (CTMF)
ISO/IEC 9646 (ITU-T X.290) defines the CTMF for the implementation of OSI and ITU protocols and is probably the mostly used conformance testing methodology standard available. ISO 9646 consists of 7 parts, one of which defines the Tree and Tabular Combined Notation (TTCN).
The CTMF is a generic conformance testing methodology in that it was designed to address the conformance testing concerns for all telecommunication protocols following the ISO/OSI layer model.

ETSI's Conformance Testing Methodology
After several years, the TTCN notation defined by ISO 9646 was found to ignore or neglect other aspects of testing that play a continuously growing role in computing systems nowadays. In fact, it was acknowledged that besides the telecommunications domain, the need for a solid conformance testing methodology was also required for other domains in which different communication paradigms apply. Therefore, after ISO stopped working on CTMF in the early nineties, the European Telecommunication Standards Institute (ETSI) took over to further develop the initial concepts to address the massive changes in the IT and telecommunications industries of that time. The results of those efforts were the TTCN-3 notation, which is now the only standardized notation specifically dedicated to conformance testing and is widely used in several domains including, IT, banking, automotive and its traditional (tele-)communications domain.

Two TTCN-3 demonstration testers exist for CoAP (developed by BUPT) and 6LoWPAN (developed by ETSI). These prototype test platforms can be completed to reach a full coverage of base specification, and **can be used in the context of the EAR-IT projec**t.

Overview of conformance Test Platform

Typically a TTCN-3 test platform is composed of four different components:

- The TTCN-3 test tool providing necessary software to execute the abstract test suites;
- The hardware equipment supporting TTCN-3 test execution and adaptation to SUTs;
- The codecs which convert protocol messages into their abstract TTCN-3 representation;
- The Test Adapter (TA) implementing interfaces with the device under test.

The interaction of these components is described in the figures below.

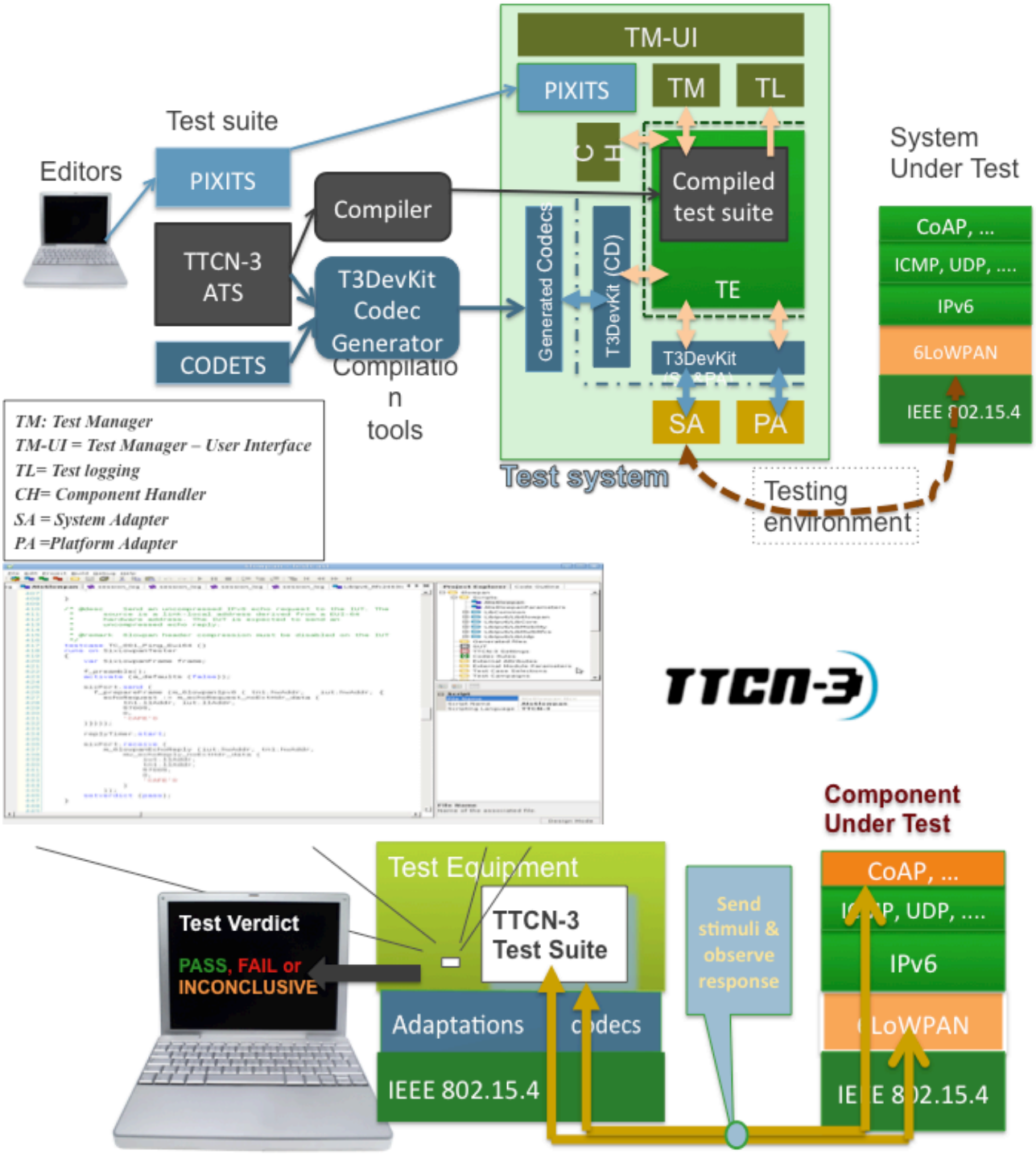


Figure 7: General Architecture of a TTCN-3 test platform

6lowpan-CoAP test evaluation platform

The CoAP/6LoWPAN test platform is composed of two hardware equipments, a standard PC and an IEEE 802.15.4 adapter box.

The main hardware component of the CoAP/6LoWPAN test platform is a standard PC. Its role is to host the execution of the test suites using a commercial TTCN-3 test tool.
Whatever operating system is installed on the computer, it is necessary to ensure that the following points are taken into account:

- No firewall interference with traffic generated by the Test System and/or SUT
- Excellent time synchronisation between the SUT and the test system

- Test system processes (especially the test adapter) have to be granted unrestricted control to telecommunication hardware

Time synchronisation is maybe the most critical point to be checked before starting any test session, as it can be the source of strange SUT behaviour and generate incoherent results.
This PC is equipped with two network cards, one being used for CoAP communication with SUT (lower layers link), the other one being used for exchanging upper tester messages (upper tester transport link). Separating these two communications on different hardware interfaces is not an absolute necessity, but it is a good practice and it ensures that there will be no interaction between the flows.
The communication between the SUT and the test system is achieved through UDP/IP if the SUT supports it or using IEEE802.15.4 adaptation hardware.

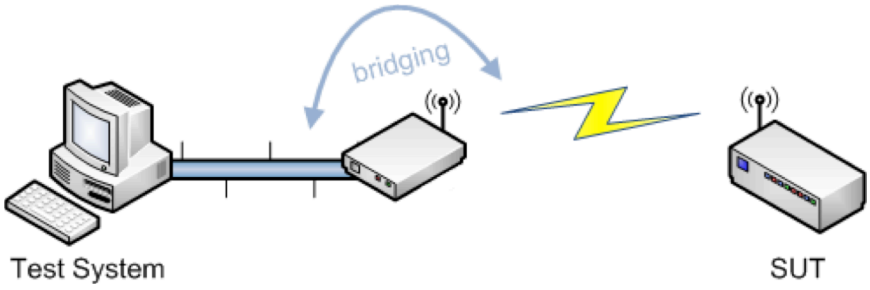


Figure 8: Communication via IEEE802.15.4 adaptation hardware

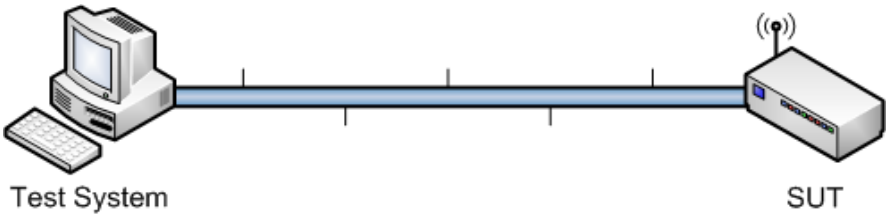


Figure 9: Communication via UDP/IP

To achieve IEEE802.15.4 connectivity, dedicated hardware equipment needs to be added to the test platform. The role of this adaptation equipment is to handle all radio-related tasks transparently and to act as a bridge for the test system.
In the field of EAR-IT project, TelosB has been chosen to fulfil this task. This device is fully IEEE 802.15.4 compliant and provides as well a USB interface so that it can be used as a transparent bridge between the test system and the SUT, as depicted in Figure 2.
To transfer frames received on the USB interface to the radio interface and vice versa, it is necessary to install and execute a small bridge application on the test system.

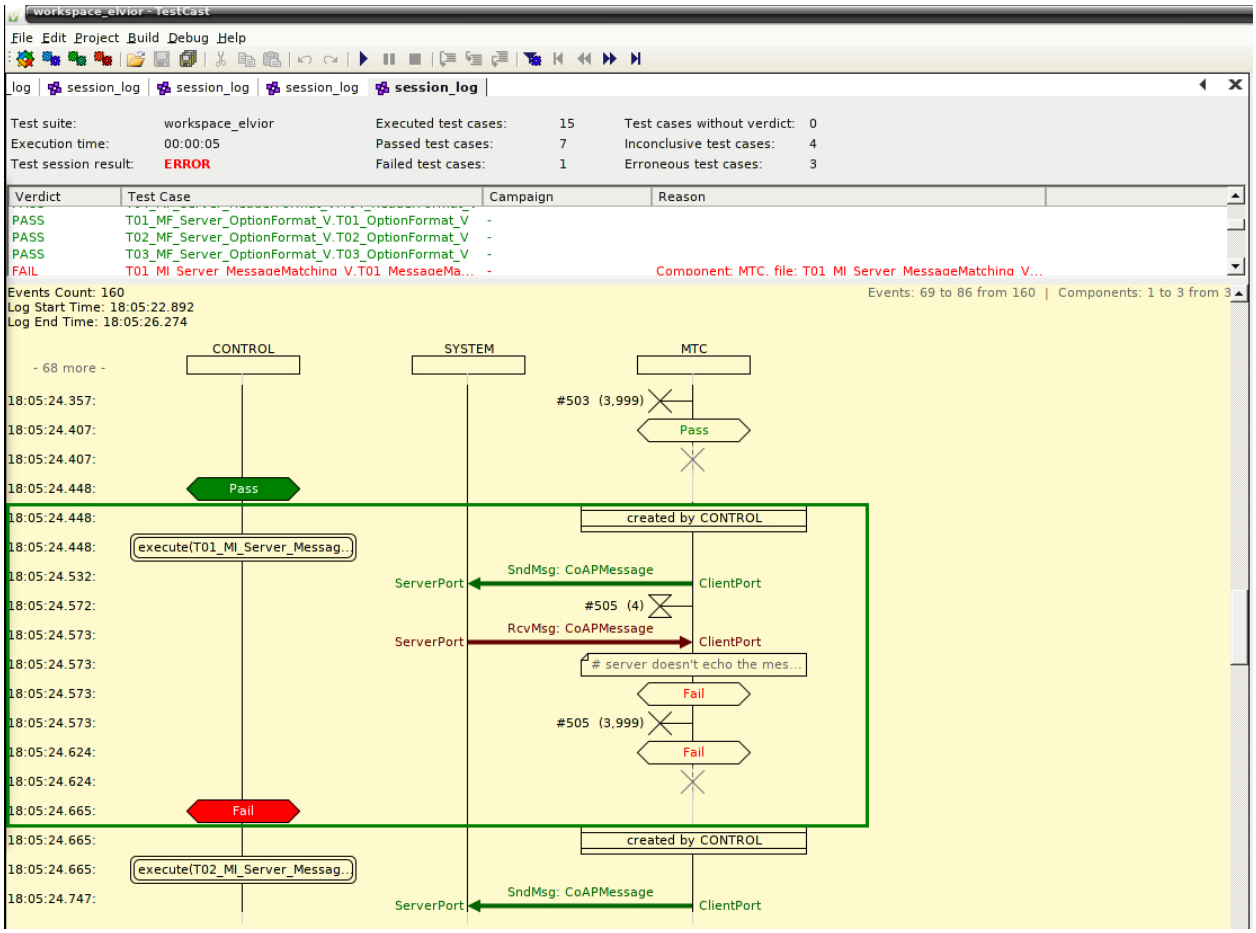


Figure 10: Example of conformance testcase execution

Interoperability Testing and worldwide events

The objective of interoperability testing is that independent implementations of the same standard interoperate. It is a well-known fact that, even following the same standard, two different implementations might not be interoperable. The heterogonous nature of IoT technologies requires interoperability issues to be solved before the deployment of the product. Interoperability testing events are important and pragmatic tools to validate standards improve implementation and finally improve interoperability.

Interoperability events

Two CoAP interoperability events have been organised in 2012, and two 6LoWPAN interoperability events are planned for 2013.

1st CoAP event (Paris, France)

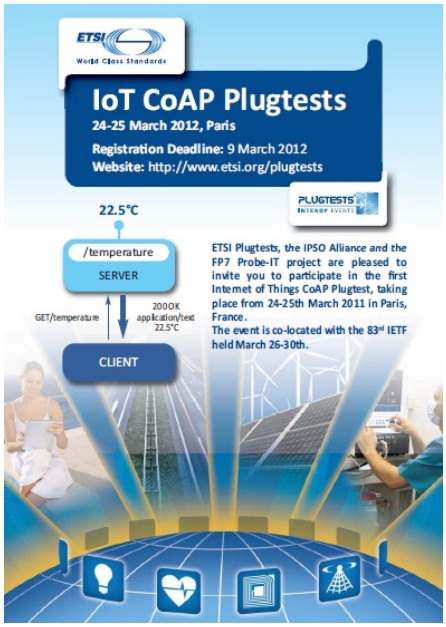
The first CoAP interoperability event, organised in joint cooperation by ETSI (European Telecommunication Standards Institute), FP7 Probe-IT project and the IPSO Alliance, took place in Paris (France) on 24-25th March 2012 and was collocated with IETF#83.

This CoAP plugtest was a two days long event carried out during IETF#83 meeting in Paris to motivate vendors to verify the interoperability of their equipment with other's. Equipment is considered interoperable once they have successfully showed that their implementation is able to communicate with implementation of other vendors without any issues. Before the testing session, participants must agree on a set of configurations to test their equipment/implementations. In all tests, at least two different vendor products must be available to conduct a suite of selected interoperability testing scenarios.
The main objectives of this event are

- First opportunity to test their CoAP implementations/equipment in one place with test suites provided by world's best test labs
- Verify the interoperability of your product with other major actors in the market
- Identify the issues and improve your CoAP implementation effectively with test suites provided
- Share experience and improve interoperability of your product

Scope:

- CoAP base specification
- CoAP Block Transfer
- CoAP Observation
- CoRE Link Format.



Coming from China, EU, Japan and Korea, 18 companies participated to this event, bringing CoAP clients and/or servers:

Company	Client	Server
Actility/Watteco	X	
ETH Zurich	X	X
Hitachi	X	X
Huawei	X	X
Intecs	X	X
KoanLogic Srl	X	X
Patavina Technologies	X	X
Rosand Technologies		X
Sensinode Ltd	X	X
Toyota ITC USA	X	
TZI Uni Bremen	X	X
Vitaverna	X	
Watteco		X
Uni Rostock	X	X
RTX	X	X
IBBT	X	X
Consorzio Ferrar Ricerche	X	X

Results:

A total of 3141 tests were executed during the test event and 94% of executed tests gave pass verdict. According to ETSI and PROBE-IT test experts, getting 90% and above of pass verdict in a first Plugtests event for a new technology is a success showing that the tested components are almost near to be fully compliant and interoperable.
There were a total number of 234 test sessions during this two days event. In each session 27

tests were executed and 388 tests were not executed due to time constraints and/or because some CoAP features (mostly from BLOCK and OBSERVE) were not yet implemented in tested components.

The result obtained for the CORE group of tests, in other words it is the overall results for mandatory tests for a total of 2843 tests executed, shows that 2679 (94.2%) of the tests passed. This confirms that most of tested components support necessary features of CoAP base specification.

The overall results of optional tests that are also near 91% of pass verdict, showing that most of implementations successfully support optional features. Even though there is not much difference in the optional test results, BLOCK group (86.3% of pass verdict) seems to have a little bit more remaining implementation issues.

To conclude, this first CoAP Plugtests event was a success as 94% of tests executed were pass, which reveals that almost all implementations from the participants are almost mature enough to be interoperable with each other. These results also show the need for more such events in the future to improve both CoAP protocol specification and corresponding implementations paving by this way the road for successful deployment of this technology.

2nd CoAP event (Sophia-Antipolis, France)

The first CoAP interoperability event, organised in joint cooperation by ETSI (European Telecommunication Standards Institute), FP7 Probe-IT project and the IPSO Alliance, took place in Sophia-Antipolis (France) on 28-30th November 2012. This event was collocated with a workshop, giving participants and other interested parties to discuss an disseminate topics such as Embedded Web Services, Introduction to CORE, New CORE features, Architecture, Integration with ETSI M2M…

Scope:

- The base CoAP specification
- CoAP Block Transfer
- CoAP Observation
- The CoRE Link Format
- Proxy
- Security DTLS
- IPSO Application Template
- Full set of options
- Resource Directory
- Basic ETSI M2M CoAP binding tests

This second CoAP interoperability events gave the participants the opportunity to run the same tests as in previous event[5] and also more advanced additional tests, involving CoAP proxies (caching mechanisms), resource discovery, ETSI M2M bindings.

Participants

Even if the test sessions were long (4 hours), most of the companies had several devices (client and servers) that ofcourse increased the number of possible pairing combinations. Globally, the feedback that the participants gave is that the testing was very dense and they concentrated themselves on the mandatory tests.

[5] *Repeating interoperability tests can be very useful, especially if new versions of implementations are available. Newest versions may not be interoperable.*

97.8% of the test verdicts were PASS which shows a very high level of maturity of the implementations.

1st 6LoWPAN event (Berlin, Germany)

The first 6LoWPAN interoperability event, organised by ETSI with the support of IPSO Alliance, FP7 PROBE-IT project and IPv6 Forum, is planned to take place in Berlin, Germany, on 27-28th July 2013. It will be collocated with IETF#87.

Scope:

- Header Compression (RFC 6282)
- Neighbor Discovery (RFC 6775)
- Frame Format

Testing will be run on the following setups:

- IEEE 802.15.4-2006 2.4 GHz
- IEEE 802.15.4g 868 MHz
- Other PHY/MAC combinations possible depending on participants' wishes

2nd 6LoWPAN event (Beijing, China)

The second 6LoWPAN interoperability event, organised by FP7 PROBE-IT/BUPT with the support of IPSO Alliance, IPv6 Forum and ETSI is planned to take place in Beijing, China, on 20-23rd August 2013. Being located in China, it is foreseen that this event will permit to involve more participants from Asian companies.

Scope:

- Header Compression (RFC 6282)
- Neighbor Discovery (RFC 6775)
- Frame Format

Testing will be run on the following setups:

- IEEE 802.15.4-2006 2.4 GHz
- IEEE 802.15.4g 868 MHz
- Other PHY/MAC combinations possible depending on participants' wishes