

## Ear-IT WP1 Acoustic Test-bed Qualification

### D1.2: Minimum requirements for use of acoustic sensors

## Abstract

This document is the EAR-IT deliverable 1.2. It presents for some selected performance indicators the minimum requirements for use of acoustic sensors on the various EAR-IT test-beds based on WSN and IoT nodes with IEEE 802.15.4 radio technology. These performance indicators are categorized into (1) network performance indicators, (2) audio quality indicators and (3) energy indicators. We will specifically present minimum requirements for audio source, communication and buffering requirements for relay nodes and sensitivity of audio codecs regarding packet loss rates. These indicators will then serve for deliverable 1.3 "Methodology and tools for measurements and benchmarking on the use of acoustic sensors" with both lab and in-situ experiments to determine the performance level of the EAR-IT test-beds.

<b>Project Number:</b> 318381	<b>Project Acronym:</b> EAR-IT	<b>Project Title:</b> Experimenting Acoustics in Real environments using Innovative Test-beds
----------------------------------	-----------------------------------	--

<b>Instrument:</b> STREP	<b>Thematic Priority</b> Future Internet Research and Experiment
-----------------------------	---

<b>Title</b>  <b>Minimum requirements for use of acoustic sensors</b>
---

<b>Contractual Delivery Date:</b>  1 <sup>st</sup> December 2013	<b>Actual Delivery Date:</b>  1 <sup>st</sup> April 2014
--	--

<b>Start date of project:</b>  October, 1 <sup>st</sup> 2012	<b>Duration:</b>  24 months
--	-----------------------------------

<b>Organization name of lead contractor for this deliverable:</b>  EGM	<b>Document version:</b>  V0.9
--	--------------------------------------

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

**Authors (organizations) :**

Congduc Pham, EGM  
Philippe Cousin, EGM

**Abstract :**

This document is the EAR-IT deliverable 1.2. It presents for some selected performance indicators the minimum requirements for use of acoustic sensors on the various EAR-IT test-beds based on WSN and IoT nodes with IEEE 802.15.4 radio technology. These performance indicators are categorized into (1) network performance indicators, (2) audio quality indicators and (3) energy indicators. We will specifically present minimum requirements for audio source, communication and buffering requirements for relay nodes and sensitivity of audio codecs regarding packet loss rates. These indicators will then serve for deliverable 1.3 "Methodology and tools for measurements and benchmarking on the use of acoustic sensors" with both lab and in-situ experiments to determine the performance level of the EAR-IT test-beds.

**Keywords :**

Acoustic data, benchmark methodology, minimum requirement of test-bed, audio streaming

## Disclaimer

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Any liability, including liability for infringement of any proprietary rights, relating to use of information in this document is disclaimed. No license, express or implied, by estoppels or otherwise, to any intellectual property rights is granted herein. The members of the project EAR-IT do not accept any liability for actions or omissions of EAR-IT members or third parties and disclaims any obligation to enforce the use of this document. This document is subject to change without notice.

# Revision History

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

Revision	Date	Description	Author (Organisation)
0.1	1 <sup>st</sup> march 2014	Initial drafting	C. Pham (EGM)
0.5	17 March 2014	Adding comments and improvements	P. Cousin
0.9	April 2nd, 2014	Pre-final version for review	C. Pham (EGM)
1.0	April 9th, 2014	Final version	C. Pham & P. Cousin (EGM)



# Table of Content

<b>EAR-IT WP1 ACOUSTIC TEST-BED QUALIFICATION .....</b>	<b>1</b>
<b>D1.2: MINIMUM REQUIREMENTS FOR USE OF ACOUSTIC SENSORS .....</b>	<b>1</b>
<b>ABSTRACT .....</b>	<b>1</b>
1. INTRODUCTION .....	6
2. EAR-IT AND ACOUSTIC DATA.....	7
3. IOT NODE’S HARDWARE ON EAR-IT TEST-BEDS .....	10
<i>SmartSantander test-bed hardware .....</i>	<i>10</i>
<i>The HobNet test-bed hardware .....</i>	<i>10</i>
4. ADDING ACOUSTIC FEATURES TO IOT NODES .....	12
<i>Acoustic data sampling possibilities with IoT nodes .....</i>	<i>12</i>
<i>Current developments on target hardware platforms.....</i>	<i>12</i>
<i>Use of a generic sender to test other audio codecs.....</i>	<i>18</i>
<i>Summary of minimum requirements at the sender side .....</i>	<i>19</i>
5. MINIMUM REQUIREMENTS IN MULTI-HOP SCENARIO.....	20
6. NETWORK INDICATORS.....	21
<i>Review of maximum IoT relaying performance .....</i>	<i>21</i>
<i>Minimum buffer requirements at relay nodes .....</i>	<i>22</i>
7. AUDIO INDICATORS.....	26
<i>Benchmark methodology.....</i>	<i>26</i>
<i>Acoustic quality indicators.....</i>	<i>26</i>
<i>Acoustic quality with respect to packet loss rate (transmission quality) .....</i>	<i>27</i>
8. ENERGY INDICATORS .....	37
9. CONCLUSIONS AND SUMMARY OF MAIN RESULTS .....	38
<i>NETWORK: minimum sending/relaying rate .....</i>	<i>38</i>
<i>NETWORK: buffer size &amp; packet drop relationship at relay nodes .....</i>	<i>39</i>
<i>AUDIO: maximum supported packet loss rate .....</i>	<i>39</i>
ANNEX.A: REVIEW OF SOFTWARE ENVIRONMENT, TOOLS AND TEST HARDWARE .....	40
ANNEX.B: FUTURE DEVELOPMENTS ON TARGETED HARDWARE PLATFORMS .....	55
REFERENCES .....	56

# 1. Introduction

This document is the EAR-IT deliverable 1.2. It presents for some selected performance indicators the minimum requirements for use of acoustic sensors on the various EAR-IT test-beds based on WSN and IoT nodes with IEEE 802.15.4 radio technology. These performance indicators are categorized into:

1. Network performance indicators (NETWORK)
2. Audio quality indicators (AUDIO),
3. Energy indicators (ENERGY).

The document is organized as follows. Section 2 will present the EAR-IT context and will review basic acoustic requirements. Section 3 will review the Santander's SmartSantander and Geneva's HobNet test-beds used by the EAR-IT project. The IoT node's hardware and network components will be presented. In Section 4 we will present how audio features are added to SmartSantander and HobNet IoT nodes, depending on the hardware constraints. Various audio codecs will be used and we will present in more details their characteristics and minimum requirements. Section 5 will present the minimum requirements in a global, multi-hop scenario and the 3 categories of indicators that we will study. Section 6 will focus on the NETWORK indicators when it comes to support acoustic data: packet loss rate, relay latency and packet jitter to name a few. Section 7 will study the AUDIO requirements to determine, according to a given audio codec, the maximum acceptable packet loss rate. For the ENERGY indicator, Section 8 will discuss some energy considerations in order to provide both performance and usability indicators.

This deliverable 1.2 will be followed by deliverable 1.3 "Methodology and tools for measurement and benchmarking on the use of acoustic sensors" with both lab and in-site tests to determine the performance level of the EAR-IT test-beds.

## 2. EAR-IT and acoustic data

There is a growing interest in multimedia contents for surveillance applications in order to collect richer information from the physical environment. Capturing, processing and transmitting multimedia information with small and low-resource device infrastructures such as Wireless Sensor Networks (WSN) or so-called Internet-of-Things (IoT) is quite challenging but the outcome is worth the effort and the range of surveillance applications that can be addressed will significantly increase. The EAR-IT project is one of these original projects which focuses on large-scale "real-life" experimentations of intelligent acoustics for supporting high societal value applications and delivering new innovative range of services and applications mainly targeting to smart-buildings and smart-cities, see figure 1.



Figure 1: Santander's SmartSantander test-bed for EAR-IT

One scenario that can be demonstrated is an on-demand acoustic data-streaming feature for surveillance systems and management of emergencies. Other applications such as traffic density monitoring or ambulance tracking are also envisioned and are also requiring timely multi-hop communications between low-resource nodes. The EAR-IT project relies on 2 test-beds to demonstrate the use of acoustic data in smart environments: the smart city SmartSantander test-bed and the smart building HobNet test-bed. Figure 2 illustrates both test-beds and the multi-hop relaying issues of acoustic data in these environments.

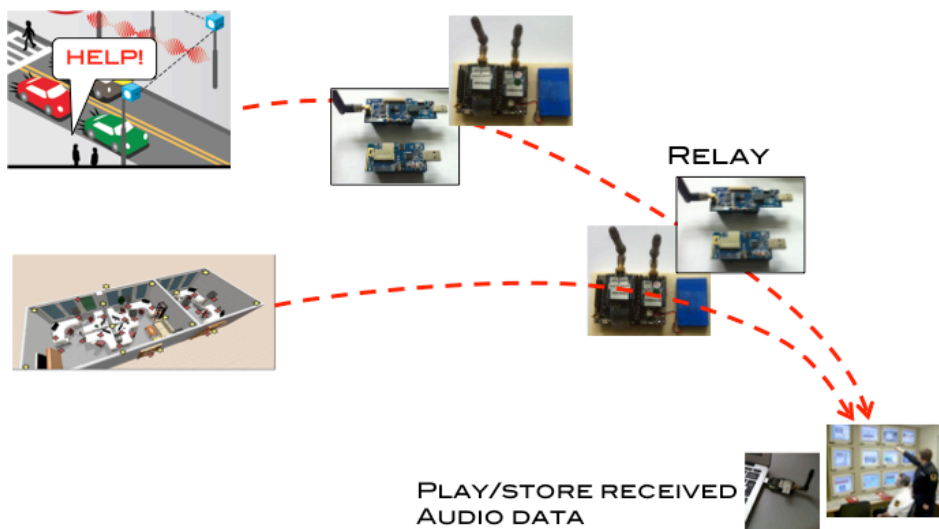


Figure 2: Acoustic data streaming on SmartSantander and HobNet

Acoustic data are usually obtained through a sampling process of an analog signal from a microphone. Narrow-band sampling processes use a sampling rate lower than 8KHz while wide-band sampling usually samples at a frequency greater than 16KHz. An A/D converter usually performs the sampling process providing the digital samples on a number of bits, e.g. a digital sample on 10 bits gives values between 0 and 1023 for instance. Sampling at 8KHz means that the A/D converter must provide 1 sample every 125us.

Most of audio processes used in communication networks are narrow-band audio with a sampling rate equal or lower than 8KHz. Also, samples are usually coded on 8 and 16 bits, meaning that the digital value provided by the A/D converter is usually mapped (quantization stage) on 8 or 16 bits. Therefore, in the so-called raw format, the continuous flow of audio data represents an 64kbit/s data flow if samples are 8 bits wide:  $8 \times 8000 = 64000$  bits.

The raw audio can be compressed in various manners and many compression algorithms have been proposed and used widely in communication networks and applications: traditional wired telephony systems, Voice over IP, GSM, ... Compression can provide a much smaller bit rate to adapt the required throughput to the available bandwidth of the transmission system. This is particularly important for near real-time audio in streaming applications. The term "audio codec" will then be used as a generic term to designate one audio compression scheme. There are hundreds of different audio codecs used in the telephony, music and video industry to name them all. Although not an authoritarian source, a quite exhaustive list of audio codecs and audio containers are presented on [http://en.wikipedia.org/wiki/List\\_of\\_codecs](http://en.wikipedia.org/wiki/List_of_codecs) and [http://en.wikipedia.org/wiki/Comparison\\_of\\_container\\_formats](http://en.wikipedia.org/wiki/Comparison_of_container_formats).

In the EAR-IT project, the hardware limitations of IoT nodes impose the use of narrow-band audio with sampling rates smaller or equal to 8KHz. Also, the limitations on the sending rate at the application level and on the radio bandwidth generally discard audio bit rates greater than 64kbps as pointed out in the EAR-IT deliverable 1.1 on the network qualification.

In addition to these constraints, we also wanted to use open-source codecs to insure largest dissemination, compatibility and interoperability. Another important criteria is the availability of libraries and tools that can be easily installed, used and integrated on any Linux-box on the market. We therefore selected 3 narrow-band and open-source audio codecs, `raw`, `codec2` and `speex`, which will be described later in the document. The minimum requirements therefore greatly depend on the audio codec that will be used.

Multi-hop transmissions as depicted by figure 3 below also increase the packet loss rates and introduce larger packet jitter. As audio traffic is isochronous, packet jitter can have a dramatic impact on the audio restitution quality.

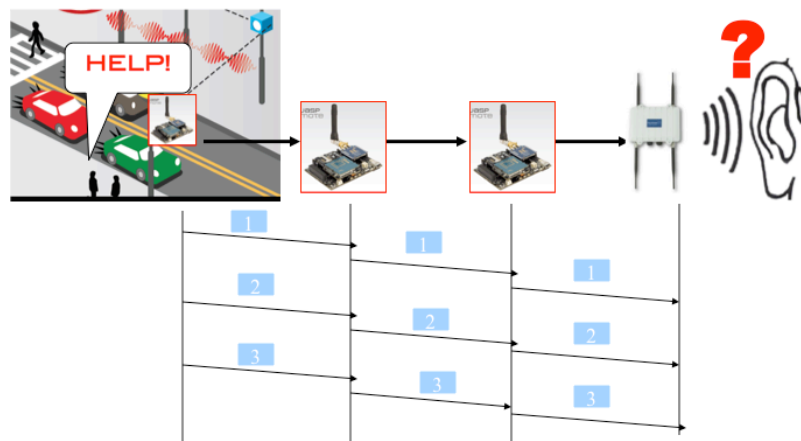


Figure 3: Multi-hop audio transmission issues

Near real-time audio streaming usually needs small packet jitter in order to avoid gaps in the audio play out. As bounded jitter is difficult to achieve because timing guarantees are difficult

to ensure in communication protocols at low cost, a best-effort approach is commonly used with an end-point play out buffer. Figure 4 below illustrates the basic principles of a play out buffer with the objective of shaping and regulating the packet output rate.

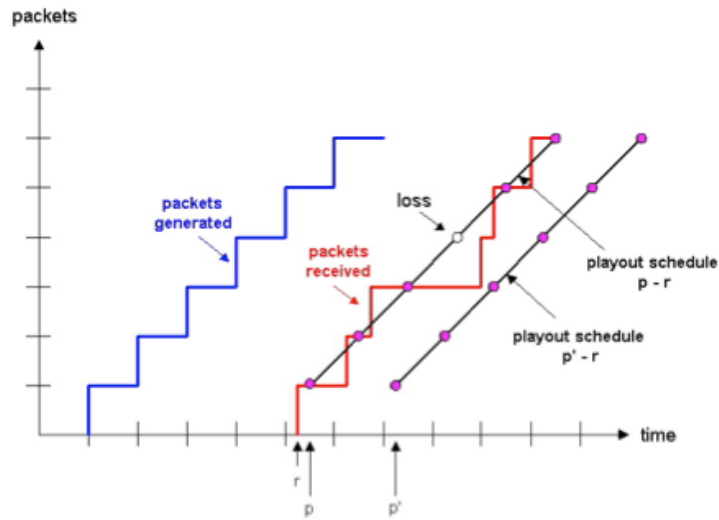


Figure 4: Principles of a play out buffer to handle packet jitter

Audio streaming is challenging on a multi-hop manner on low-resource IEEE 802.15.4 IoT nodes because relaying overheads and packet loss rates can be high. However, if the number of hops is small, a best-effort approach can be adopted for simplicity with a simple play out buffer at the end point (we assume that most of communication issues are between the IoT node and the gateway as once on the gateway, traditional Internet connection technologies such as Wi-Fi, 3G or wired Ethernet are sufficient enough).

The use of a play out buffer leverages the packet jitter issue at the cost of a higher play out latency. The minimum requirement regarding packet jitter is then to define at the application level the acceptable latency. For instance, for an on-demand audio streaming scenario, the maximum acceptable time between the audio request and the beginning of the audio play out must be defined. Low bit rate audio codecs have the advantage of not requiring large amount of buffers.

The next section will present the developed hardware for the EAR-IT's IoT nodes to support acoustic data, i.e. sampling and transmission.

### 3. IoT node's hardware on EAR-IT test-beds

The EAR-IT test-beds consist in (i) the SmartSantander test-bed and (ii) the HobNet test-bed. The SmartSantander test-bed is a FIRE test-bed with 3 locations. Being one location the Santander city in north of Spain with more than 5000 nodes deployed across the city. This is the site we will use when referring to the SmartSantander test-bed. HobNet is also a FIRE test-bed that focuses on Smart Buildings. Although the HobNet test-bed has several sites, within the EAR-IT project only test-bed located at MANDAT Intl and HEPIA are concerned. Many information can be found on corresponding project web site ([www.smartsantander.eu](http://www.smartsantander.eu) and [www.hobnet-project.eu](http://www.hobnet-project.eu)) but we will present in the following paragraphs some key information that briefly present the main characteristics of the deployed nodes.

#### SmartSantander test-bed hardware

##### *IoT nodes and gateways*

IoT nodes in the Santander test-bed are WaspMote sensor boards and gateways are Meshlium gateways, both from Libelium. Most of IoT nodes are also repeaters for multi-hops communication to the gateway. Figure 5 shows on the left part the WaspMote sensor node serving as IoT node and on the right part the gateway. The WaspMote is built around an Atmel ATmega1281 micro-controller running at 8MHz. There are 2 UARTs in the WaspMote that serve various purposes, one being to connect the micro-controller to the radio modules.

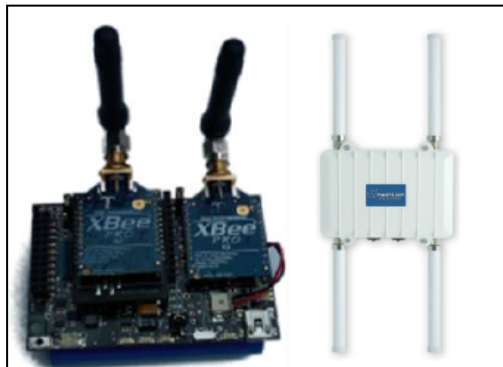


Figure 5: Santander's IoT node and gateway

##### *Radio module*

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 protocol along with more advanced coordination/node discovery functions. In this document, we only consider acoustic data transmission/relaying using the 802.15.4 radio module as the DigiMesh interface is reserved for management and service traffic. XBee 802.15.4 offers the basic 802.15.4 [802154] PHY and MAC layer service set in non-beacon mode. Santander's nodes have the "pro" version, set at 10mW transmit power, with an advertised transmission range in line-of-sight environment of 750m. Details on the XBee/XBee-PRO 802.15.4 modules can be found in [XBeeDigi] [DMDigi].

#### The HobNet test-bed hardware

##### *IoT nodes*

Sensor nodes in the HobNet test-bed consist in AdvanticsSys TelosB motes, mainly CM5000 and CM3000, see figure 6, that are themselves based on the TelosB architecture. These motes are



built around a TI MSP430 microcontroller with an embedded Texas Instrument CC2420 802.15.4 compatible radio module. The TelosB description and data-sheet can be found in [TELOSB]. Documentation on the AdvanticsSys motes can be found in [ADVAN]. AdvanticsSys motes run under the TinyOS system [TINYOS]. The last version of TinyOS is 2.1.2 and our tests use this version.

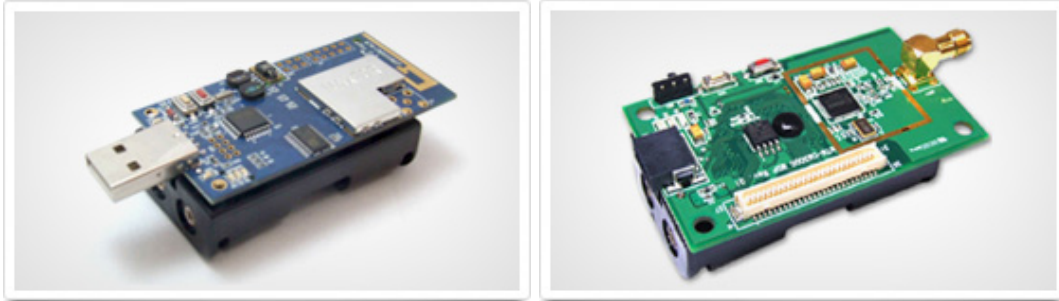


Figure 6: CM5000 (left) and CM3000 (right)

### **Radio module**

The CC2420 is less versatile than the XBee module but on the other hand more control on low-level operations can be achieved. 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 line which normally would allow for much faster data transfer rates. The CC2420 radio specification and documentation are described in [CC2420].

The TinyOS configuration by default uses a MAC protocol that is compatible with the 802.15.4 MAC (Low Power Listening features are disabled). It also uses ActiveMessage (AM) paradigm to communicate. As we are using heterogeneous platforms we will rather the TKN154 IEEE 802.15.4 compliant API. We verified the performances of TKN154 against the TinyOS default MAC and found them greater.

## 4. Adding acoustic features to IoT nodes

### Acoustic data sampling possibilities with IoT nodes

As stated previously, most of IoT nodes are based on low speed microcontroller (Atmel 1281 at 8MHz for the Libelium WaspMote and TI MSP430 at 16Mhz for the AdvanticSys) making simultaneous raw audio sampling and transmission nearly impossible when using only the mote microcontroller.

To leverage these performance issues, one common approach is to dedicate one of the 2 tasks to another microcontroller:

1. Use another microcontroller to perform all the transmission operations (memory copies and buffering, frame formatting, among others);
2. Use another microcontroller to perform the sampling operations (generates interruptions, reads analog input, performs A/D conversion and possibly encodes the raw audio data).

With the hardware platforms used in the EAR-IT project we can investigate these 2 solutions:

1. Libelium WaspMote uses an XBee radio module which has an embedded internal microcontroller that is capable of handling all the sending operations when running in so-called transparent mode (API mode 0 of XBee module);
2. Develop a daughter audio board for the AdvanticSys TelosB mote that will perform the periodic sampling, encode the raw audio data with a given audio codec and fill in a buffer that will be periodically read by the host microcontroller, i.e. the TelosB MSP430.

In the following sub-sections we will describe in more details these 2 solutions to demonstrate the audio capabilities of resource-constrained IoT nodes.

### Current developments on target hardware platforms

#### *Libelium WaspMote*

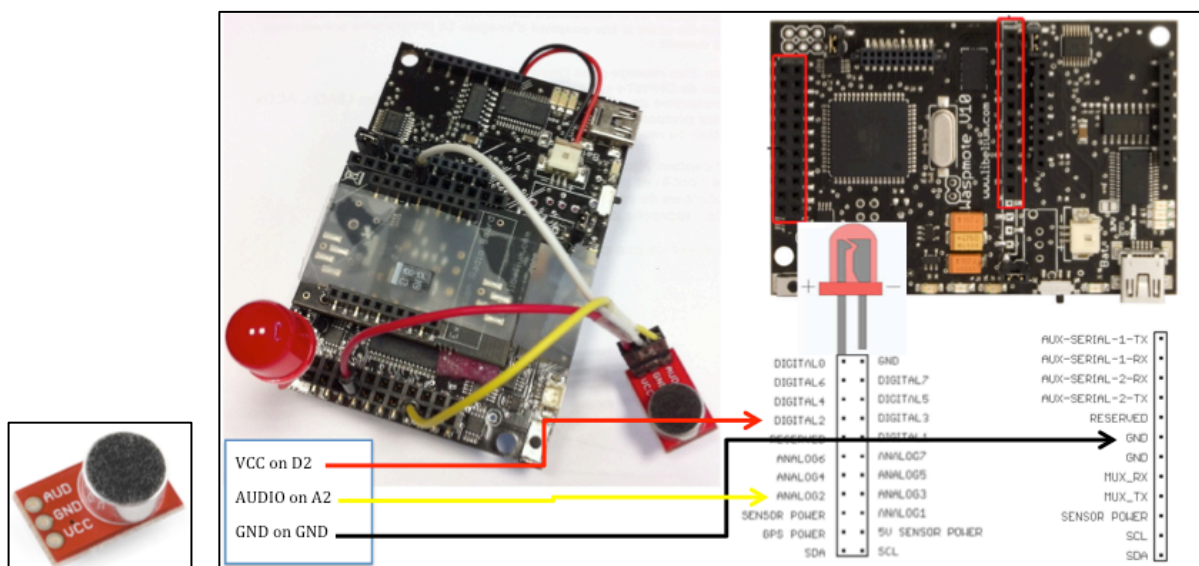


Figure 7: raw audio capture with Libelium WaspMote

## Synopsis

1. Use a pre-amplified MIC and connect it to an analog input of the Libelium WaspMote. We use the following MIC: <http://www.cooking-hacks.com/shop/sensors/sound/breakout-board-for-electret-microphone> (see figure 7, left) and connect it to the WaspMote (AUD to Analog2, VCC to Digital 2 to get 3.3V and GND to GND, see figure 7, right).
2. Configure an XBee radio module in transparent mode (API mode 0). Broadcast or unicast communications can be used but this has to be configured prior to sending any data because we let the XBee microcontroller do all the sending tasks. Here is a text taken from the XBee manual from Digi:

*« When operating in this mode, the modules act as a serial line replacement - all UART data received through the DI pin is queued up for RF transmission »*

*« Data is buffered in the DI buffer until one of the following causes the data to be packetized and transmitted:*

- a. No serial characters are received for the amount of time determined by the RO (Packetization Timeout) parameter. If RO = 0, packetization begins when a character is received.*
- b. The maximum number of characters that will fit in an RF packet (100) is received.*
- c. The Command Mode Sequence (GT + CC + GT) is received. Any character buffered in the DI buffer before the sequence is transmitted. »*

In our case, data will be sent by the XBee radio module internal microcontroller either on case (a) or (b).

3. Sample the analog input (Analog2) at 4KHz or 8KHz, i.e. read analog value once every 250us or 125us. A/D converter gives a 10-bit sample so it has to be converted into an 8-bit sample.
4. As the XBee radio module is connected to the host microcontroller, i.e. the Atmel 1281, with a serial UART line, we can just write in a dedicated register the 8-bit sampled value.
5. Receive on a PC or a gateway (Libelium Meshlium for instance) using an XBee radio module in APO mode that will send data to the serial interface.
6. Continuously read PC or gateway serial port and send data to standard output (usually `stdout` on a Unix machine). Use redirection to inject `stdout` into an audio player such as `play` (part of `sox` package on a Linux machine).

## Current development status

1. All the steps have been successfully demonstrated and validated.
2. 4KHz and 8KHz sampling version are available.

## **Review of sending performances**

We already reported in deliverable 1.1 the time spent in a generic `send()` function, noted  $t_{\text{send}}$ , and the minimum time between 2 packet generation, noted  $t_{\text{pkt}}$ .  $t_{\text{pkt}}$  will typically take into account various counter updates and data manipulation so depending on the amount of processing required to get and prepare the data,  $t_{\text{pkt}}$  can be quite greater than  $t_{\text{send}}$ . With  $t_{\text{send}}$ , we can easily derive the maximum sending throughput that can be achieved if packets could

be sent back-to-back, and with  $t_{pkt}$  we can have a more realistic sending throughput.

In order to measure these 2 values, we developed a traffic generator with advanced timing functionalities. Packets are sent back-to-back with a minimum of data manipulation needed to maintain some statistics (counters) and to fill-in data into packets, which is the case in a real application. On the WaspMote, we increased the default serial baud rate between the microcontroller and the radio module from 38400 to 125000. The Libelium API has also been optimized (for instance, we also remove the overhead of waiting for transmission status, which is not very relevant for real-time acoustic data) to finally cut down the sending overheads by almost 3 compared to the original Libelium API! Figure 8 shows  $t_{send}$  and  $t_{pkt}$  for the WaspMote.

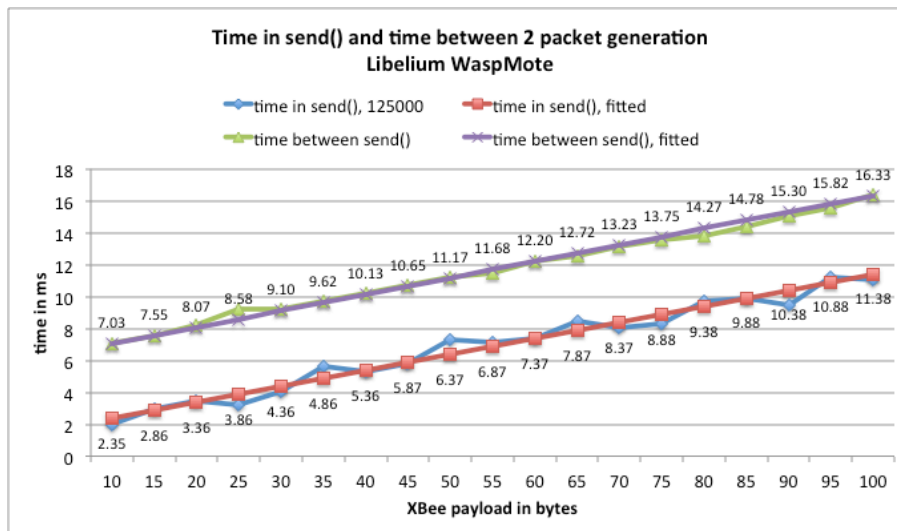


Figure 8:  $t_{send}$  and  $t_{pkt}$  for for WaspMote

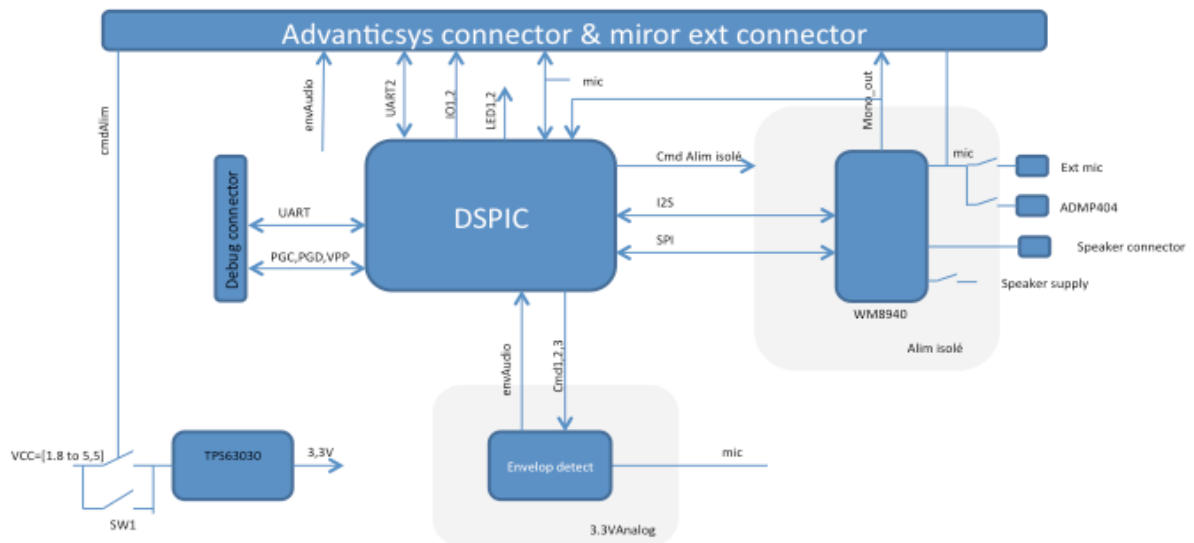
### Minimum requirements for raw audio

1. The default factory communication speed of an XBee module is 9600 bauds. Libelium ships the XBee module with the Libelium WaspMote configured at 38400 bauds. This baud rate can handle 4KHz sampling.
2. For 8KHz sampling, the baud rate must be increased to at least 64000 bauds. Due to clock constraints (that were explained in deliverable 1.1) standard baud rates (such as 115200) are not accurate enough and 125000 bauds should be used instead. Therefore the XBee module **MUST BE** configured at 125000 baud to handle 8KHz sampling.
3. It is difficult to use the *RO (Packetization Timeout)* for triggering the sending of buffered data. Therefore, acoustic data are sent once 100 8-bit samples have been buffered. This means that the communication stack **MUST BE** able to send a 100-byte radio packet every 25ms or 12.5ms depending on the sampling frequency, i.e. 4KHz or 8KHz.
4. According to deliverable 1.1 this is out of reach of the WaspMote with the radio module in API mode, even for 4KHz sampling rate, which requires a time window of 25ms, because the sending delays (shown in figure 8 above) are only valid with a microcontroller fully dedicated to the communication tasks. This is the reason why we set the XBee module in transparent mode, delegating the radio packet formatting overheads to the XBee embedded micro-controller while the main WaspMote micro-controller is dedicated to the sampling process.

## AdvanticSys TelosB

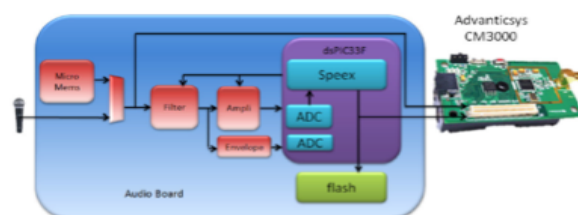
### Synopsis

1. Develop a daughter audio board with its own microcontroller that will be connected to the AdvanticSys expansion connector. The audio board will handle the sampling operations and encode in real-time the raw audio data into Speex codec ([www.speex.org](http://www.speex.org)). 8KHz sampling and 8-bit sample will be used to produce an optimized 8kbps encoded Speex stream (speex encoding library is provided by Microchip).
2. The audio board is designed and developed through collaboration with INRIA CAIRN research team. Here is a schematic of the audio board design:



The audio board has a built-in omnidirectional MEMs microphone (ADMP404 from Analog Devices) but an external microphone can also be connected. The microphone signal output is amplified, digitized and filtered with the WM8940 audio codec. The audio board is built around a 16-bit Microchip dsPIC33EP512 microcontroller clocked at 47.5 MHz that offers enough processing power to encode the audio data in real-time. From the system perspective, the audio board sends the audio encoded data stream to the host microcontroller through an UART component. The host mote will periodically read the encoded data to periodically get fixed size encoded data packets that will be transmitted wirelessly through the communication stack.

3. Connect the audio board to the AdvanticSys through the 51-pin expansion connector: from the system perspective, the audio board sends the audio encoded data stream through an UART connection to the host micro-controller.



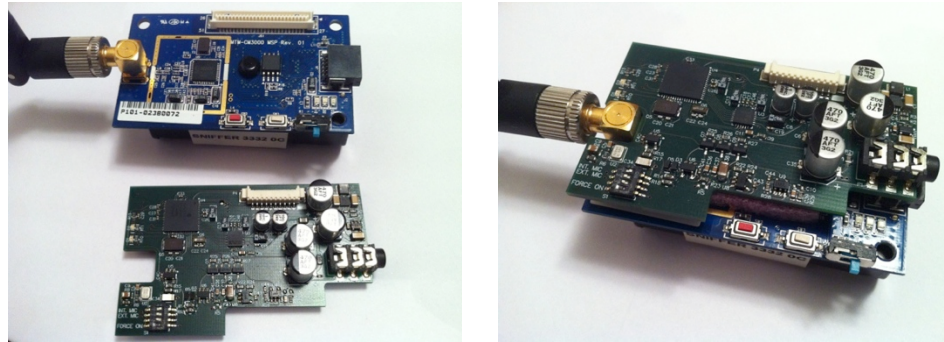
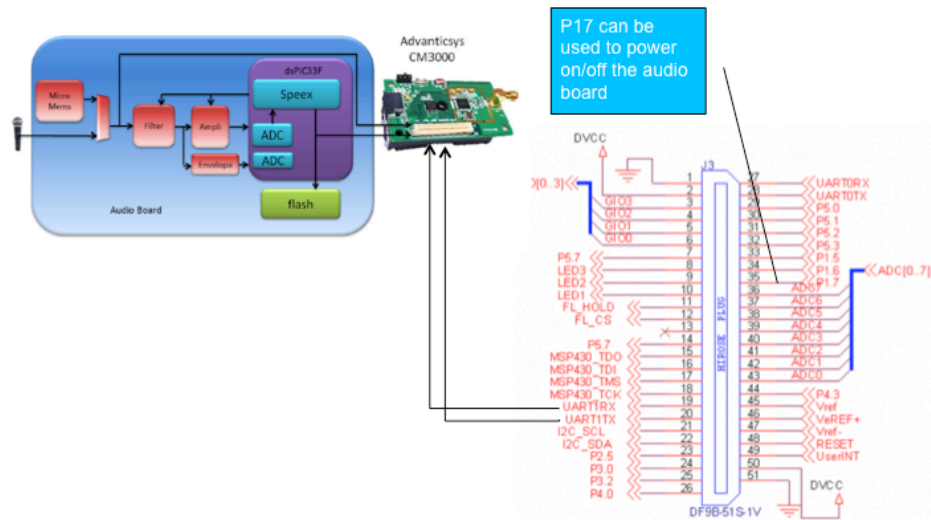
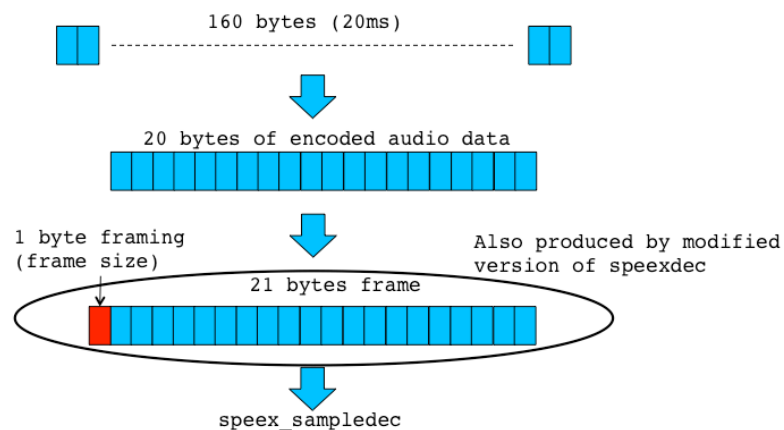


Figure 9: developed audio board and AdvanticsSys TelosB with the audio board



- 8KHz speex works with 20ms audio frames: every 20ms, 160 8-bit samples of raw audio data are sent to the speex encoder to produce a 20-byte audio packet.



- Read encoded data from the host mote to periodically get fixed size encoded data packets that will be transmitted wirelessly through the communication stack (provided by TinyOS environment).
- Receive on a PC or a gateway (Libelium Meshlium for instance) using another AdvanticsSys mote as a base station mote.
- Continuously read PC or gateway serial port and send data to standard output (usually `stdout` on a Unix machine). Use redirection to inject `stdout` into a Speex decoder that will also send on `stdout` the raw decoded audio data.



- Use redirection to inject `stdout` into an audio player such as `play` (part of `sox` package on a Linux machine).

#### Current development status

- All the steps have been successfully demonstrated and validated.
- Audio capture and data streaming can be triggered on an on-demand basis: the audio board can be controlled and configured remotely.

#### Review of sending performances

Similar to the WaspMote case, we reported in deliverable 1.1 the time spent in a generic `send()` function, noted  $t_{\text{send}}$ , and the minimum time between 2 packet generation, noted  $t_{\text{pkt}}$ , for the AdvanticsSys TelosB mote. Figure 10 shows  $t_{\text{send}}$  and  $t_{\text{pkt}}$  for the TelosB with the TKN154 802.15.4 protocol stack under TinyOS 2.1.2.

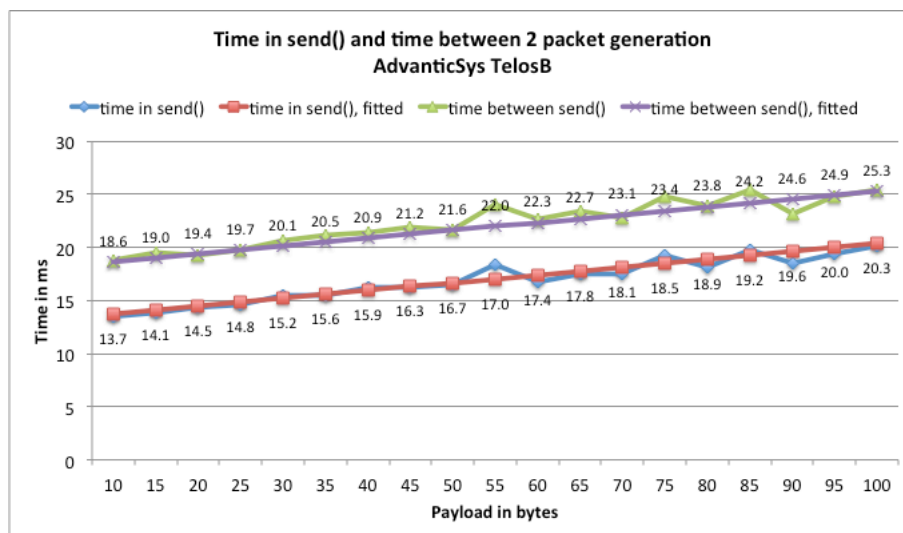


Figure 10:  $t_{\text{send}}$  and  $t_{\text{pkt}}$  AdvanticsSys TelosB

#### Minimum requirements of speex codec

- As the `speex` encoder produces a 20-byte audio packet every 20ms, the sender node **SHOULD BE** able to send a 20-byte radio packet every 20ms.
- However, as the payload of a 20ms audio sampling is smaller than the maximum radio payload (100 bytes), it is possible to aggregate several audio frames into 1 radio packet. Due to additional framing bytes (4 bytes) that are required for reliability and robustness issues (see ANNEX A, slide 16), the maximum number of audio frames that can be aggregated is 4 (noted A4 aggregation level), giving a total payload of 96 bytes. Therefore the sender **MUST BE** able to send a 96-byte packet every 80ms as the minimum requirement for loss-free aggregation mode.
- Without audio aggregation the time to send a 20-byte packet is very close to the time window of 20ms. Therefore audio data losses are likely to occur at the source. Starting from an aggregation level of 2 (2 audio frames in a radio packet, A2 level), the AdvanticsSys TelosB can easily sustain the required sending rate.

## Use of a generic sender to test other audio codecs

### Synopsis

1. We use an Arduino MEGA 2560 platform with an XBee radio module and an external SD card storage extension. We also have an LCD display to ease the interaction with the mote. Figure 11 below shows our Arduino-based generic sender node.

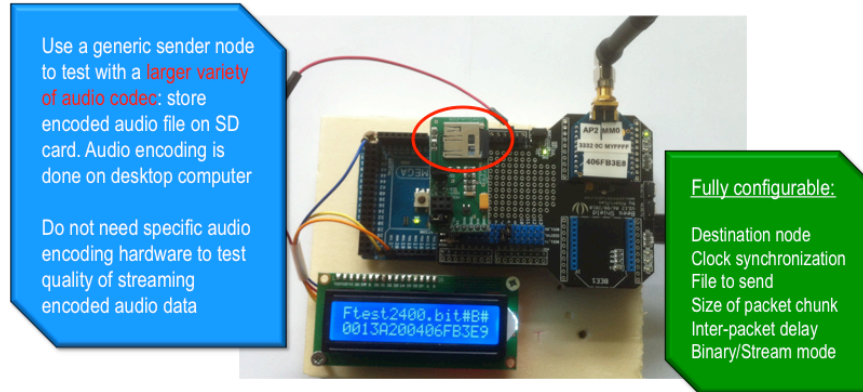
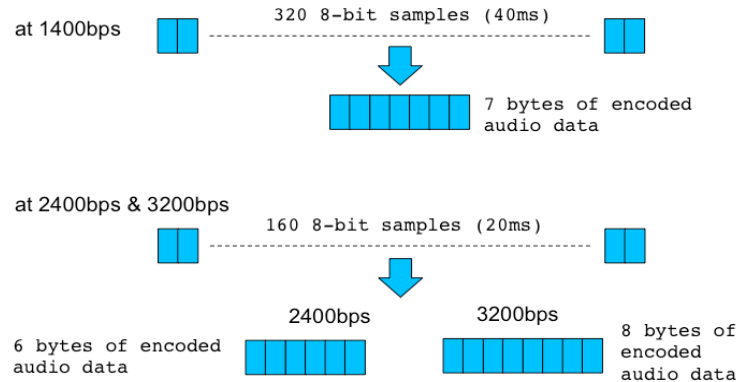


Figure 11: Arduino-based generic sender with an SD card extension

2. A desktop computer (e.g. Linux machine) is used to produce the desired audio codec. We use the open-source `codec2` codec, which is a very low bit rate codec. `codec2` proposes bit rates of 1400, 2400 and 3200bps. Figure below shows for the various bit rates the `codec2` operations.



3. The audio files are stored on an SD card and we can dynamically select which file is going to be sent. 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 the generation of two packets as well as the chunk size.
4. Receive on a PC or a gateway (Libelium Meshlium for instance) using any 802.15.4 gateway.
5. Continuously read PC or gateway serial port and send data to standard output (usually `stdout` on a Unix machine). Use redirection to inject `stdout` into a `codec2` decoder that will also send on `stdout` the raw decoded audio data.
6. Use redirection to inject `stdout` into an audio player such as `play` (part of `sox` package on a Linux machine).

## Current development status

1. All the steps have been successfully demonstrated and validated.

## **Review of sending performances**

The WaspMote board is very similar to the Arduino MEGA. However, the latter platform is more powerful as it runs at 16MHz instead of 8MHz or the WaspMote. In addition, the communication API is less complex. Therefore, on the Arduino, the communication API has better performances, especially for  $t_{pkt}$ : for a 100-byte packet  $t_{pkt}$  is about 13ms to be compared with the 16.3ms of the WaspMote.

## **Minimum requirements**

1. `codec2` encoder at 1400bps produces a 7-byte audio packet every 40ms. For 2400 and 3200 bit rates, the encoder works with 20ms time window and produces 6 and 8 bytes of encoded audio respectively. In this document, we will only consider 2400 and 3200 bit rates. Therefore, the sender node SHOULD BE able to send 6 or 8 bytes every 20ms.
2. However, as the payload of 20ms audio sampling is much smaller than the maximum radio payload (100 bytes), it is possible to aggregate several audio frames into 1 radio packet. Due to additional framing bytes (3 bytes) that are required for reliability and robustness issues (see ANNEX A, slide 25), the maximum number of audio frames that can be aggregated at 2400bps is 11 (A11). At 3200bps, only 9 frames can be aggregated (A9), giving a total payload of 99 bytes in both cases. Therefore the sender MUST BE able to send a 99-byte packet every 220ms or 180ms respectively as the minimum requirement for loss-free aggregation mode.

## Summary of minimum requirements at the sender side

Codec	Minimum sending rate
Raw 4KHz	100 bytes every 25ms
8KHz	100 bytes every 12.5ms
Speex 8000bps A1 A2 A3 A4	24 bytes every 20ms 48 bytes every 40ms 72 bytes every 60ms 96 bytes every 80ms
Codec2 2400bps A1 . . An ( $1 \leq n \leq 11$ ) 3200bps A1 . . An ( $1 \leq n \leq 9$ )	9 bytes every 20ms . . 9*n bytes every n*20ms 11 bytes every 20ms . . 11*n bytes every n*20ms

Table I: summary of the minimum requirements at the sender side

## 5. Minimum requirements in multi-hop scenario

Figure 12 illustrates from the source to the destination through relay nodes the various constraints and limitations that will impact the audio transmission in a multi-hop scenario.

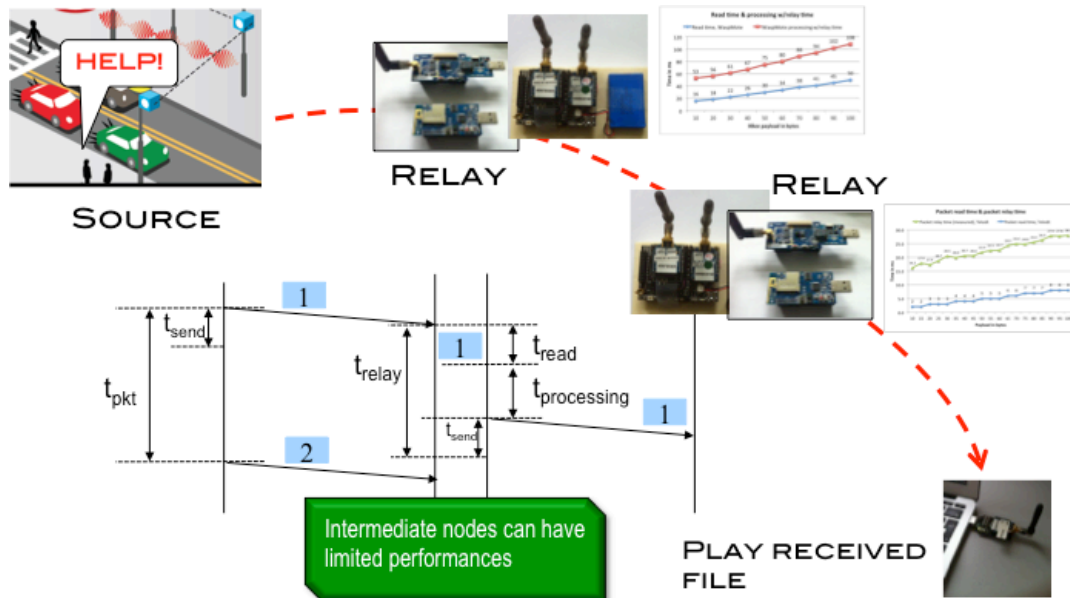


Figure 12: multi-hop constraints and limitations

Our focus in this document is to define performance indicators for acoustic data in a multi-hop environment. In the next deliverable we will measure experimentally these indicators both in lab test conditions and in-site test condition on the two EAR-IT test-beds. These performance indicators are categorized into:

4. Network performance indicators (NETWORK)
5. Audio quality indicators (AUDIO),
6. Energy indicators (ENERGY).

For network indicators, the minimum requirements will be determined for:

1. Packet relaying time and jitter at relay nodes
2. Buffering capability at relay nodes

For audio quality indicators, we will study:

1. Sensitivity of audio codecs
2. Impact of packet size on audio quality
3. Impact of packet losses on audio quality

to determine the minimum requirements for an acceptable audio quality at the receiver.

For energy indicators, we will discuss on:

1. Node lifetime for capturing and transmitting audio
2. Node lifetime for relaying audio data

## 6. NETWORK indicators

### Review of maximum IoT relaying performance

We also used the traffic generator to send packets to a receiver where we measured (i) the time needed by the mote to read the received data into user memory or application level, noted  $t_{read}$ , and (ii) the total time needed to relay a packet. Figure 13 shows the results.

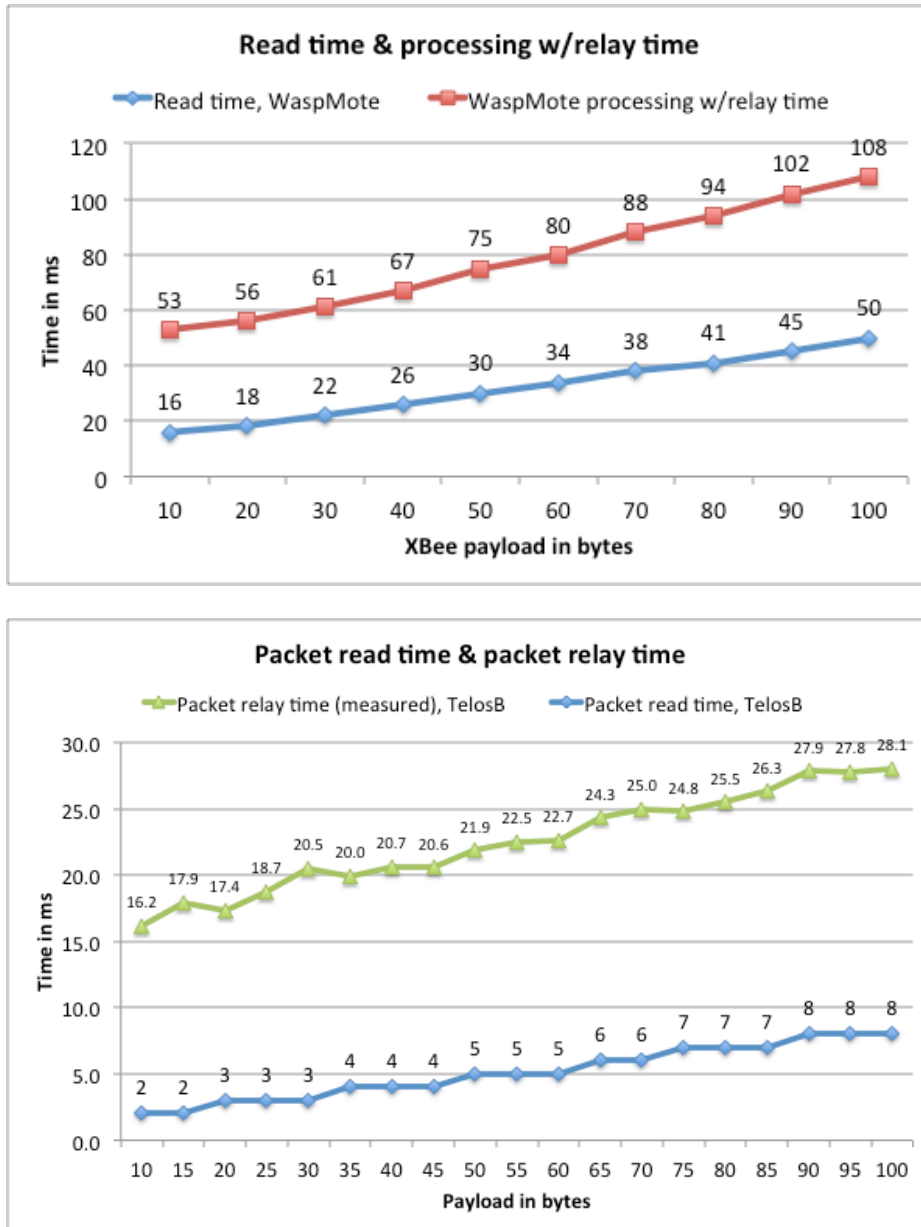


Figure 13:  $t_{read}$  and  $t_{relay}$  for for WaspMote (top) and AdvanticsSys TelosB (bottom)

On the WaspMote, we found that  $t_{read}$  is quite independent from the microcontroller to radio module communication baud rate as the main source of delays come from memory copies.

In figure 13(top), the relaying time is based on a radio to microcontroller communication speed of 38400bps. Unlike the previous case of audio source mote where we increased this speed to 125000, we chose to use the default speed as changing it needs a major change in the software and hardware configuration of sensor board which is not practically possible at large scale on the SmartSantander test-bed.

## Minimum buffer requirements at relay nodes

### *Libelium WaspMote audio source*

At the sender, raw audio at 4KHz and 8KHz with the WaspMote gives a 100-byte packet every 25ms and 12.5ms respectively. To sustain the relaying operation, a relay node **MUST BE** able to relay incoming packets as fast as it receives them. Buffers can be used to store incoming packets but the cost of memory copies must be taken into account. With buffering capabilities, we can have a more elaborated system.

**With WaspMote as relay nodes**, the relaying time (reception and transmission to next hop) for a 100-byte packet is about 108ms. During the relaying of a single packet, the WaspMote relay node will receive about 4 packets or 8 packets depending on the sampling rate of the source, i.e. 4KHz or 8KHz. The system is similar to a simple producer-consumer model with deterministic packet arrival time and service time. Given that the packet-incoming rate is more than 4 times greater than the output rate, the traffic intensity  $\rho$  is well above 1 therefore the system is not sustainable and packets will be dropped when the storage buffer is full.

If we take the relaying time as the system's cycle duration  $T$  and assuming a continuous behaviour (both for arrival and departure), the amount of bytes in excess every cycle  $T=108\text{ms}$  is:

- 4KHz:  $108/25 * 100 = 432$
- 8KHz:  $108/12.5 * 100 = 864$

Therefore, if  $\lambda$  is the byte arrival rate, we have  $\lambda_{4\text{KHz}}=432$  and  $\lambda_{8\text{KHz}}=864$ . If  $Q$  is the size of the buffer, we can write:

$$\text{(Eq. 1)} \quad Q(t) = \lambda * t - \mu * t$$

Where  $\mu$  is the departure rate, which is 100 bytes for a cycle.

If  $Q = 4000$  bytes which is the typical amount of dynamic memory available for the application on these low-resource motes, replacing  $Q$  in Eq. 1 gives:

$$\text{(Eq. 2)} \quad t = 4000/(\lambda - \mu)$$

Depending on the sampling rate, we have:

- $t_{4\text{KHz}} = 4000/(\lambda_{4\text{KHz}} - \mu) = 12.05$
- $t_{8\text{KHz}} = 4000/(\lambda_{8\text{KHz}} - \mu) = 5.24$

In summary, according to Eq. 2, a WaspMote relay node will start dropping incoming packets after  $12.05 * T = 12.05 * 108\text{ms} = 1.3\text{s}$  if the audio source is a 4KHz source, or after  $5.24 * T = 5.24 * 108\text{ms} = 0.57\text{s}$  if the audio source is an 8KHz source.

**If the relay node is an AdvanticSys TelosB**, we have  $T=28\text{ms}$  instead of 108ms and,  $\lambda_{4\text{KHz}}=112$  and  $\lambda_{8\text{KHz}}=224$ . Again, depending on the source-sampling rate, we have:

- $t_{4\text{KHz}} = 4000/(\lambda_{4\text{KHz}} - \mu) = 333.33$
- $t_{8\text{KHz}} = 4000/(\lambda_{8\text{KHz}} - \mu) = 32.26$

In summary, an AdvanticSys TelosB relay node will start dropping incoming packets after  $333.33 * T = 333.33 * 28\text{ms} = 9.33\text{s}$  if the audio source is a 4KHz source, or after  $32.26 * T = 32.26 * 28\text{ms} = 0.9\text{s}$  if the audio source is a 8KHz source.

These results are summarized in Table II below



	Aggregation level	Relay time (ms)	pkt inter-arrival time (ms)	pkt size (byte)	$\lambda$	$\mu$	Q (byte)	t (#cycle)	t (second)
WaspMote audio 4KHz WaspMote relay	NA	108	25	100	432	100	4000	12.05	1.30
WaspMote audio 8KHz WaspMote relay	NA	108	12.5	100	864	100	4000	5.24	0.57
WaspMote audio 4KHz TelosB relay	NA	28	25	100	112	100	4000	333.33	9.33
WaspMote audio 8KHz TelosB relay	NA	28	12.5	100	224	100	4000	32.26	0.90

Table II: summary of the minimum requirements at relay node, WaspMote audio

Table III below shows the time before packet drop due to a full receive buffer when the amount of buffer Q is varied from 1000 bytes to 5000 bytes by a 500-byte increment. /W or /t denotes respectively a WaspMote and a TelosB relay node. Figure 14 illustrates these results.

WaspMote audio, WaspMote & TelosB relay nodes				
Q	4KHz/W	8KHz/W	4KHz/T	8KHz/T
1000	0.33	0.14	2.33	0.23
1500	0.49	0.21	3.50	0.34
2000	0.65	0.28	4.67	0.45
2500	0.81	0.35	5.83	0.56
3000	0.98	0.42	7.00	0.68
3500	1.14	0.49	8.17	0.79
4000	1.30	0.57	9.33	0.90
4500	1.46	0.64	10.50	1.02
5000	1.63	0.71	11.67	1.13

Table III: time before packet drop due to a full receive buffer, WaspMote audio

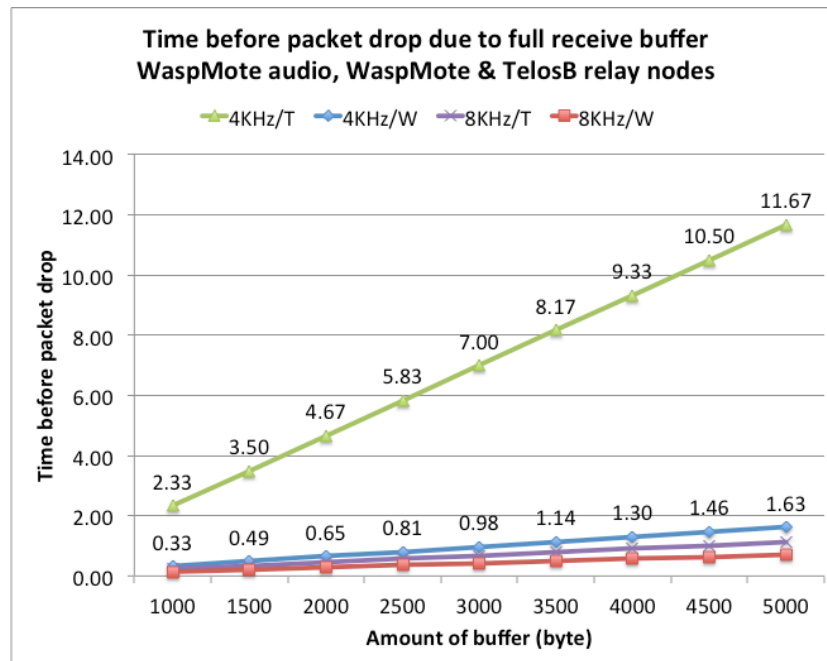


Figure 14: time before packet drop due to a full receive buffer, WaspMote audio

When building a minimal relay node with a WaspMote, we found that the amount of free memory for the application is about 2500 bytes. In this case a WaspMote node can relay less than 1s of 4KHz audio, without dropping any packet .

## AdvanticSys TelosB audio source

At the sender, `speex` encoded audio at 8kbps with AdvanticSys TelosB gives a 24-byte packet (with framing overhead) every 20ms without audio aggregation (A1 level). Again, to sustain the relaying operation, a relay node **MUST BE** able to relay incoming packets as fast as it receives them. Table I shown previously gives the minimum requirements in terms of relaying capabilities to avoid packet drops depending on the audio aggregation level.

**With WaspMote as relay nodes**, the relaying time (reception and transmission to next hop) according to the audio aggregation level is;

- A1, 24-byte packet: relay time is about 58ms
- A2, 48-byte packet: relay time is about 74ms
- A3, 72-byte packet: relay time is about 89ms
- A4, 96-byte packet: relay time is about 106ms

Table I shown previously also gave the packet inter-arrival time depending on the audio aggregation level. Therefore, if we take the same methodology than previously with the WaspMote audio node, we can determine the impact of buffering capability on the audio packet drop rate. The results are summarized in Table IV below for  $Q=4000$  bytes as previously.

	Aggregation level	Relay time (ms)	pkt inter-arrival time (ms)	pkt size (byte)	$\lambda$	$\mu$	Q (byte)	t (#cycle)	t (second)
TelosB audio/WaspMote relay	A1	58	20	24	69.60	24	4000	87.72	5.09
	A2	74	40	48	88.80	48	4000	98.04	7.25
	A3	89	60	72	106.80	72	4000	114.94	10.23
	A4	106	80	96	127.20	96	4000	128.21	13.59

Table IV: summary of the minimum requirements at WaspMote relay node, AdvanticSys TelosB audio

Again, Table V below shows the time before packet drop due to a full receive buffer when the amount of buffer  $Q$  is varied from 1000 bytes to 5000 bytes by a 500-byte increment, and for the various aggregation levels. Figure 15 illustrates these results.

TelosB audio board, WaspMote relay node					
Q	A1	A2	A3	A4	
1000	1.27	1.81	2.56	3.40	
1500	1.91	2.72	3.84	5.10	
2000	2.54	3.63	5.11	6.79	
2500	3.18	4.53	6.39	8.49	
3000	3.82	5.44	7.67	10.19	
3500	4.45	6.35	8.95	11.89	
4000	5.09	7.25	10.23	13.59	
4500	5.72	8.16	11.51	15.29	
5000	6.36	9.07	12.79	16.99	

Table V: time before packet drop due to a full receive buffer, TelosB audio board

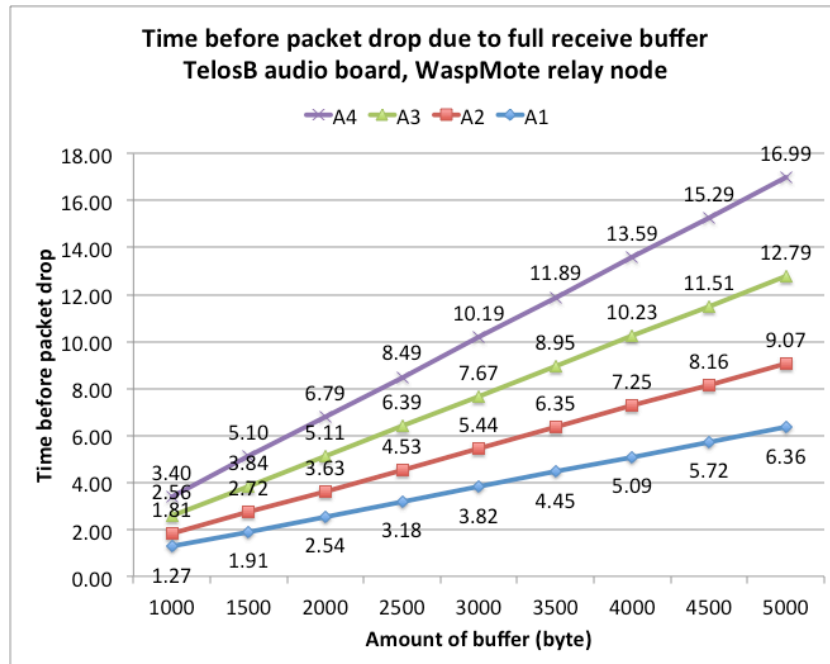


Figure 15: time before packet drop due to a full receive buffer, TelosB audio board

**With TelosB as relay nodes**, the relaying time (reception and transmission to next hop) according to the audio aggregation level is;

- A1, 24-byte packet: relay time is about 18ms
- A2, 48-byte packet: relay time is about 21ms
- A3, 72-byte packet: relay time is about 25ms
- A4, 96-byte packet: relay time is about 28ms

Therefore, theoretically, the TelosB can relay faster than the packet inter-arrival time. We then have a system where buffers are not needed.

## 7. AUDIO indicators

The WaspMote platform will be tested with raw audio data that will be referred to as `raw` codec. The AdvanticsSys TelosB with the audio board will be tested with `speex` codec. We also tested with `codec2` codec for comparison purposes. In this case, the generic sender node is used to send perform data segmentation and transmission.

### Benchmark methodology

Figure 16 shows the benchmark methodology for measuring audio codec sensitivity to packet losses for instance.

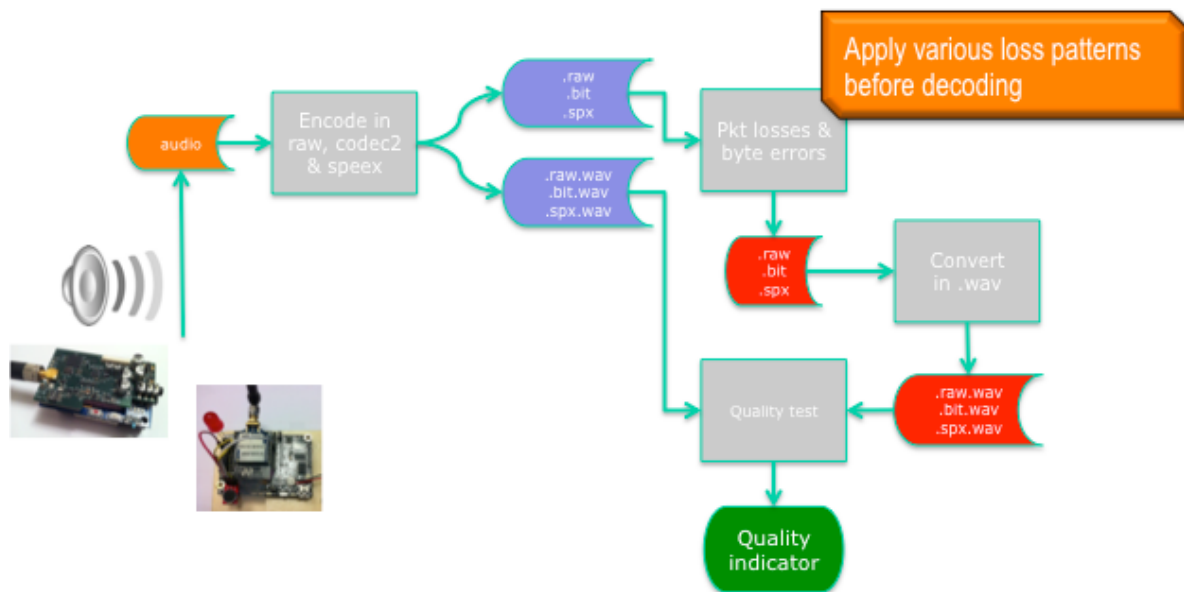


Figure 16: General process of audio quality tests

Going from left to right, audio data are converted into `raw`, `codec2` and `speex` codecs: `.raw`, `.bit` and `.spx` (purple boxes). We also create `.wav` format of these audio files: `.raw.wav`, `.bit.wav` and `.spx.wav` (purple boxes). Various packet sizes and packet loss rates are applied on the `.raw`, `.bit` and `.spx` files (red boxes), that are then converted into `.wav` format (red boxes). The audio quality between the original files and the files with packet losses will be determined and compared.

### Acoustic quality indicators

We can use ITU-T PESQ benchmark tool suite to determine the MOS (Mean Opinion Score) value for audio data (raw, codec2, speex). MOS value greater than 2.6 are usually considered very acceptable in telephone applications. Here are links to tools and documents related to the ITU-T PESQ benchmark

- ITU-T P.862 Perceptual evaluation of speech quality (PESQ): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs. Download software at: [https://www.itu.int/rec/dologin\\_pub.asp?lang=e&id=T-REC-P.862-200511-I!Amd2!SOFT-ZST-E&type=items](https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-P.862-200511-I!Amd2!SOFT-ZST-E&type=items)
- ITU-T PESQ in practice: <http://stackoverflow.com/questions/2329403/how-to-start-a-voice-quality-pesq-test>

- E-Model tutorial: <http://www.itu.int/ITU-T/studygroups/com12/emodelv1/tut.htm>
- E-model on-line tool: <http://www.itu.int/ITU-T/studygroups/com12/emodelv1/calcul.php>
- ITU-T P Series: Telephone transmission quality, telephone installations, local line networks : <http://www.itu.int/net/itu-t/sigdb/genaudio/Pseries.htm>
- ITU-T R-factor, E-model and MOS:  
[http://www.sageinst.com/downloads/960B/EModel\\_wp.pdf](http://www.sageinst.com/downloads/960B/EModel_wp.pdf)
- wav2rtp tool : <http://wav2rtp.sourceforge.net>

In all our tests, we use an original raw 8KHz audio test file of about 13s, referred to as `test.wav`. This file has then been converted into `codec2` (.bit) and `speex` (.spx) codec. Various bit rates can be used: usually the higher the better quality. `codec2` can use 1400, 1600, 2400 and 3200 bit rates while `speex` can propose 2150, 5950, 8000, 11000, 13000 and 15000 bit rates. Table VI below shows the MOS value (output of the ITU-T PESQ software) of comparison between the original `test.wav` file and the encoded one. We also have a 4KHz sampling file to compared with the 8KHz case. The MOSLQO column is the MOS value used for comparison purposes.

REFERENCE test.wav	DEGRADED test.wav	PESQMOS 4.500	MOSLQO 4.549	SAMPLE_FREQ 8000	MODE nb
test.wav	test4000Hz.raw.wav	0.769	1.115	4000	nb
test.wav	test2150.spx.wav	2.757	2.472	8000	nb
test.wav	test5950.spx.wav	3.428	3.454	8000	nb
test.wav	test8000.spx.wav	3.652	3.757	8000	nb
test.wav	test11000.spx.wav	3.941	4.093	8000	nb
test.wav	test13000.spx.wav	3.941	4.093	8000	nb
test.wav	test15000.spx.wav	4.085	4.235	8000	nb
test.wav	test1400.bit.raw.wav	2.625	2.293	8000	nb
test.wav	test1600.bit.raw.wav	2.648	2.323	8000	nb
test.wav	test2400.bit.raw.wav	2.768	2.487	8000	nb
test.wav	test3200.bit.raw.wav	2.801	2.533	8000	nb

Table VI : MOSLQO value of 4KHz raw, speex and codec2 codecs compared to original file

Most of `speex` bit rates have high MOS value. With the developed audio board, the `speex` bit rate is 8000bps. Compared to the original file, the MOS indicator shows a value of 3.757 which denotes a very good fidelity to the original file. With `codec2`, as the bit rate is very low, the MOS values are in all cases below 2.6. However, as the EAR-IT targeted applications are not telephone conversations but mainly short audio streaming with a human operator at the other end, we observed that even the 4000Hz sampling file with a MOS value of 1.115 still provides sufficient quality for a human operator to understand the speech. In the EAR-IT audio streaming scenario, the MOS value is therefore an useful indicator for comparison purposes but a low MOS value does not necessarily means that the audio data is un-exploitable by a human operator.

## Acoustic quality with respect to packet loss rate (transmission quality)

This section describes the benchmark tests illustrated in figure 9. We use `XBeeSendFile` (see Annex.B.8) to segment the encoded audio file and to apply packet losses. We then compared the original encoded file with the output of `XBeeSendFile` with the ITU-T PESQ software. We will present in the next paragraphs the results for `raw`, `codec2` and `speex` codecs.

## Raw

Raw format audio capture is realized with the Libelium WaspMote hardware as illustrated previously in figure 7. The sampling rate can be 4KHz and 8KHz. The XBee radio module runs in transparent mode and automatically sends the packet when the maximum radio packet size is reached, i.e. 100 bytes (see ANNEX.A slide 9).

It is however possible to trigger the sending of buffered data with adequate commands sent to the radio module. Figure 17 shows for the 4000Hz sampling case the MOSLQO value when the packet loss rate is varied from 5% to 70% and the radio packet size is set to 40 bytes. The first blue bar represents the loss-free case. Therefore, we have the maximum MOSLQO value when compared to the loss-free case. When a packet is dropped/lost, the receiver can simply either ignore it, or it can detect the packet loss (with gap in sequence number for instance) and fill-in the missing data with 0 values for instance.

Doing so can preserve the timing of the audio file and generally can improve the audio quality. The red bars represent the case where missing data is ignored, while the blue bars are for the case when the receiver detects the packet losses.

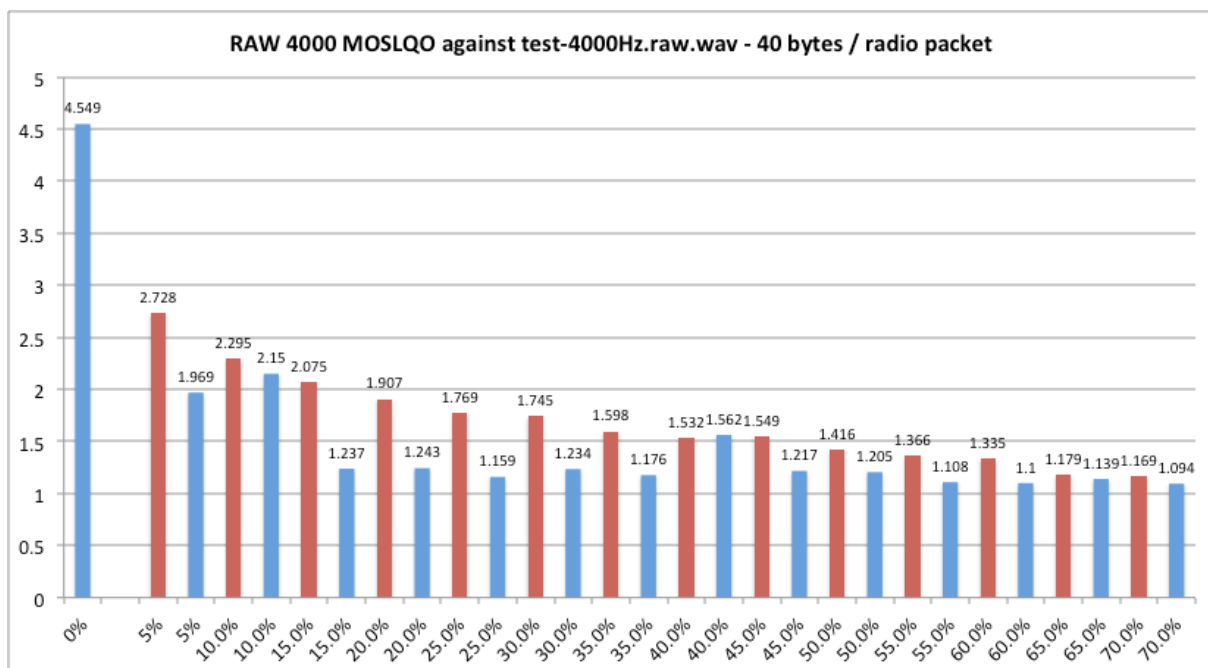


Figure 17: MOSLQO value for 4KHz raw format as packet loss rate is varied, 40 bytes payload

Figure 18 shows the MOS when the radio packet size is set to 80 bytes



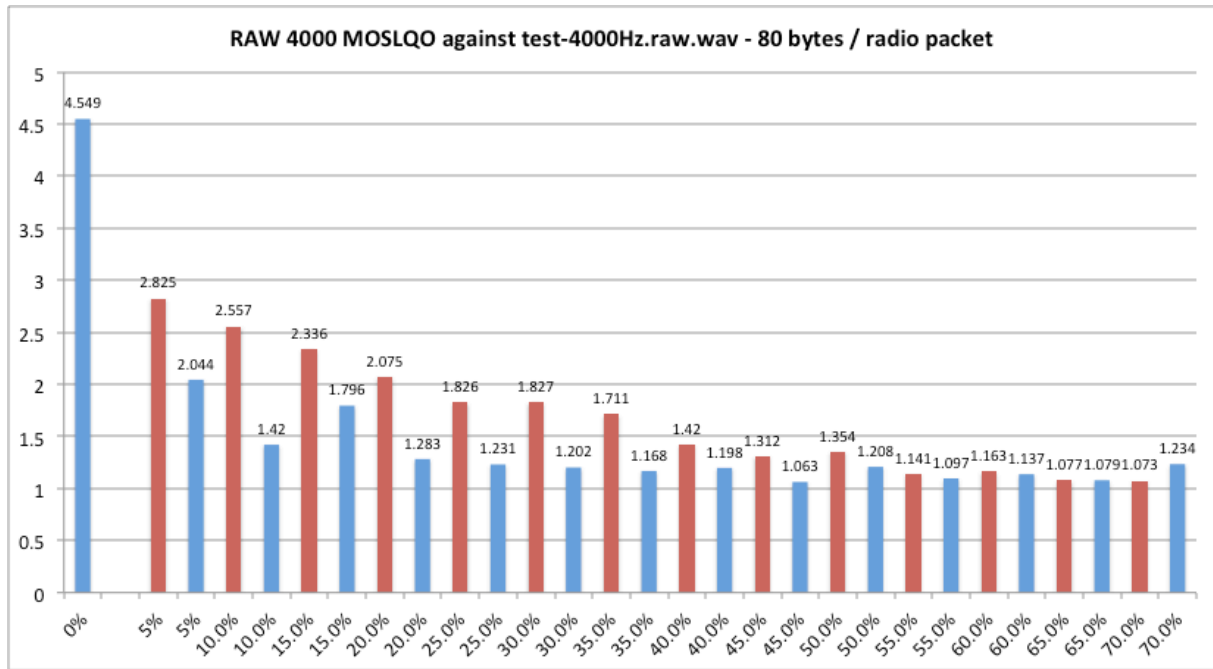


Figure 18: MOSLQO value for 4KHz raw format as packet loss rate is varied, 80 bytes payload

Even though the MOS value is well below 2.0 for packet loss rates greater than 25%, we observed that the audio quality is still sufficient for an easy understanding of the speech up to 50% packet losses.

Figure 19 shows for the 8000Hz sampling case the MOSLQO value when the packet loss rate is varied from 5% to 70% and the radio packet size is set to 100 bytes.

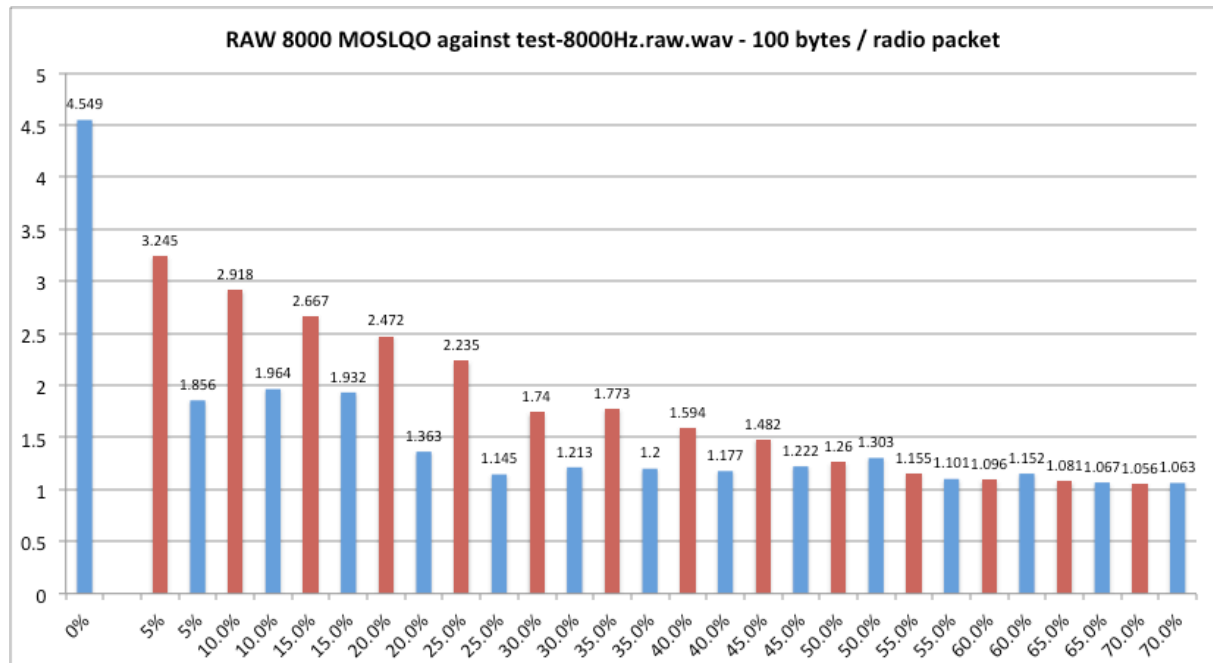


Figure 19: MOSLQO value for 8KHz raw format as packet loss rate is varied, 100 bytes payload

Again, a packet loss rate of 50% still provides a sufficient quality for an easy understanding of the speech.

## Speex

As mentioned previously, the developed audio board uses a `speex` codec with a bit rate of 8000bps. The `speex` codec at 8kbps works with 20ms audio frames: every 20ms, 160 8-bit samples of raw audio data are sent to the `speex` encoder to produce a 20-bytes audio packet. 4 framing bytes are then added to the audio data for transmission as illustrated by figure 20.

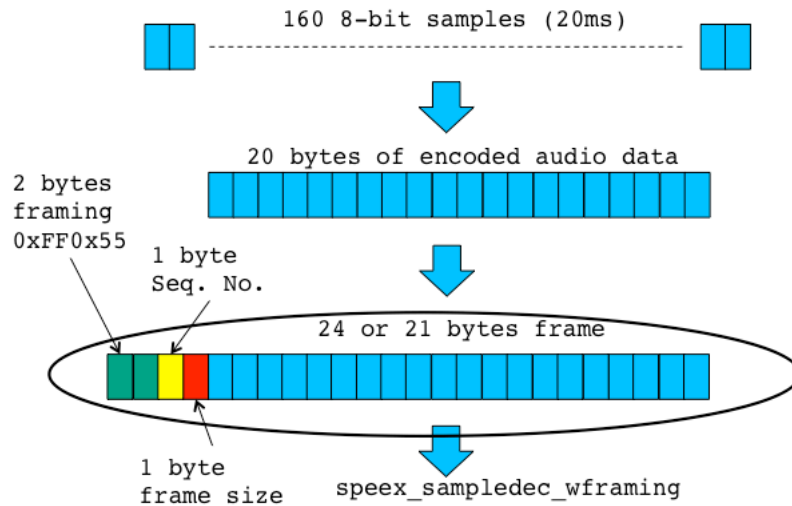


Figure 20: *speex* audio data at 8000bps

The receiver recognizes an audio packet by the usage of the first two framing bytes (0xFF0x55). Then a sequence number can be used to detect packet losses. The last framing byte stores the audio payload size (in our case it is always 20 bytes).

As the basic audio frame is only 20 bytes long, it is possible to aggregate several audio frames into one radio packet. Doing so could serve to increase the time window for network-relaying operations as this is will be shown and discussed later on. Figure 21 shows the case when 1 audio frame is sent in 1 radio packet and the MOSLQO value is determined according to the packet loss rates. This will be referred to as A1 aggregation case and the real payload is 20 audio bytes with 4 framing bytes, giving a total of 24 bytes.

Once again, the first blue bar represents the loss-free case. Therefore, we have the maximum MOSLQO value when compared to the original file. When a packet is dropped/lost, the `speex` receiver can simply either ignore it, or it can detect the packet loss (with gap in sequence number for instance) and use a dedicated decoding procedure. Once again, doing so can preserve the timing of the audio file and generally can improve the audio quality. The red bars represent the case where missing data are ignored, while the blue bars are for the case when the receiver detects the packet losses.

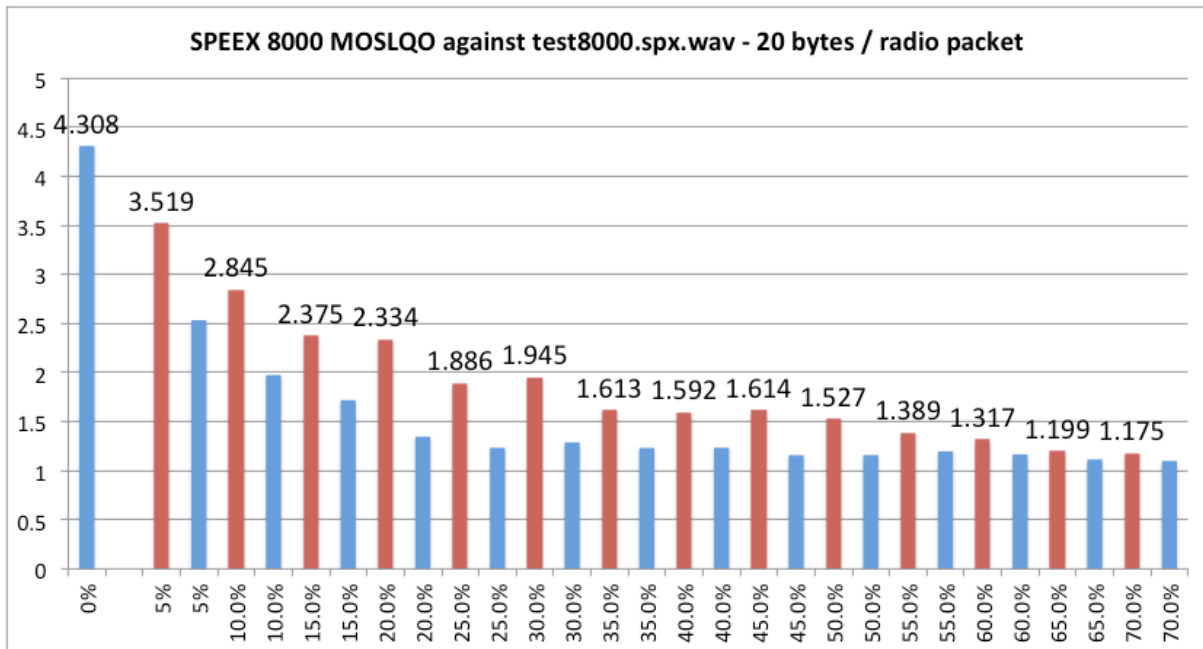


Figure 21: MOSLQO value for 8kbps speex codec as packet loss rate is varied, A1 level

For packet loss rates up to 20% the MOSLQO value is well over 2.0 and the audio quality is very good. However, we also observed that a packet loss rate up to 35% still provide a sufficient quality for an easy understanding of the speech.

Aggregating 2 audio frames into 1 radio packet gives a real payload of 48 bytes with 40 bytes being the audio data. Figure 22 shows this case, noted A2 level. Figure 23 and 24 respectively show the A3 and A4 aggregation level. A4 level is the maximum aggregation level as the real payload is 4 times 24 bytes, giving a total payload of 96 bytes (102 bytes being the maximum payload size of IEEE 802.15.4).

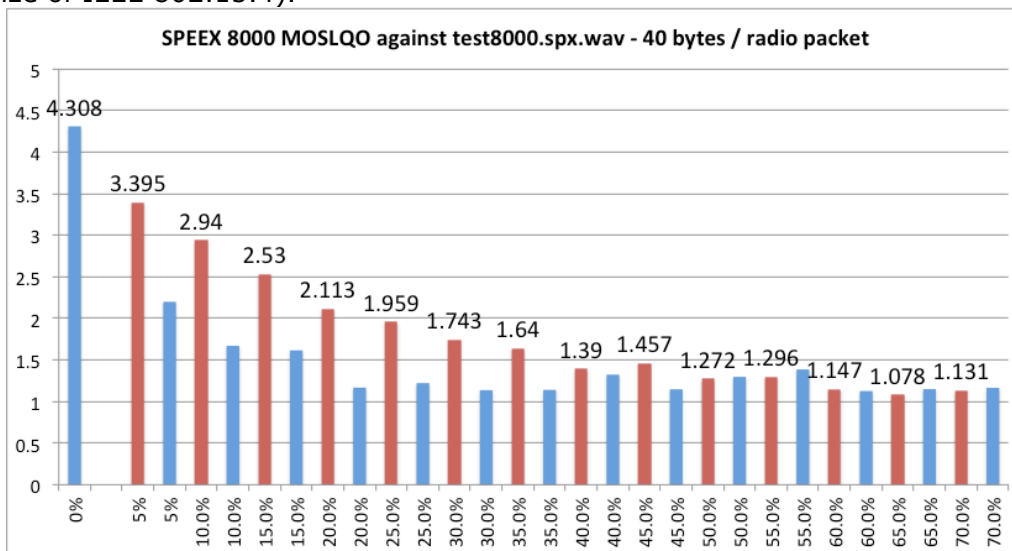


Figure 22: MOSLQO value for 8kbps speex codec as packet loss rate is varied, A2 level

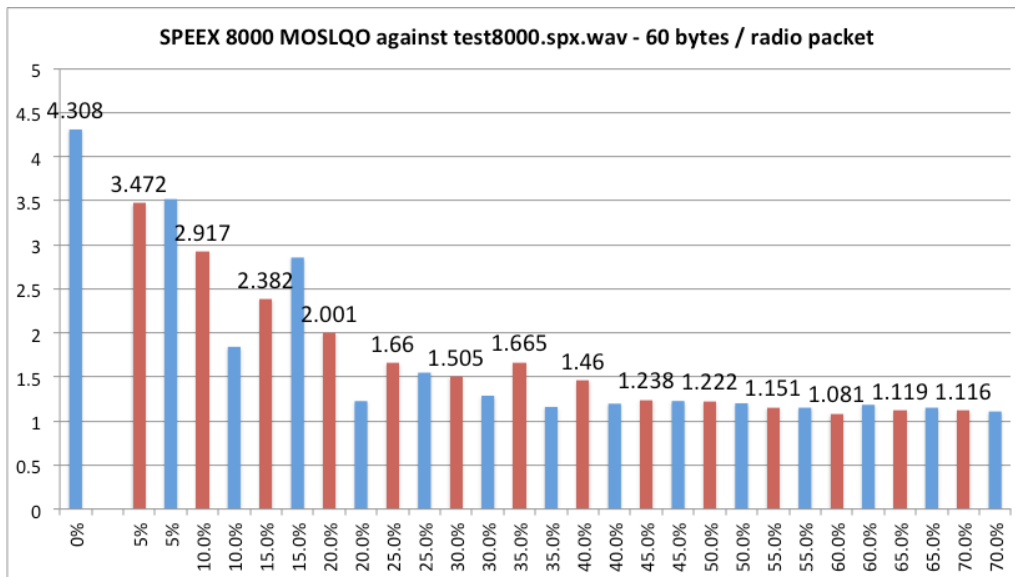


Figure 23: MOSLQO value for 8kbps speex codec as packet loss rate is varied, A3 level

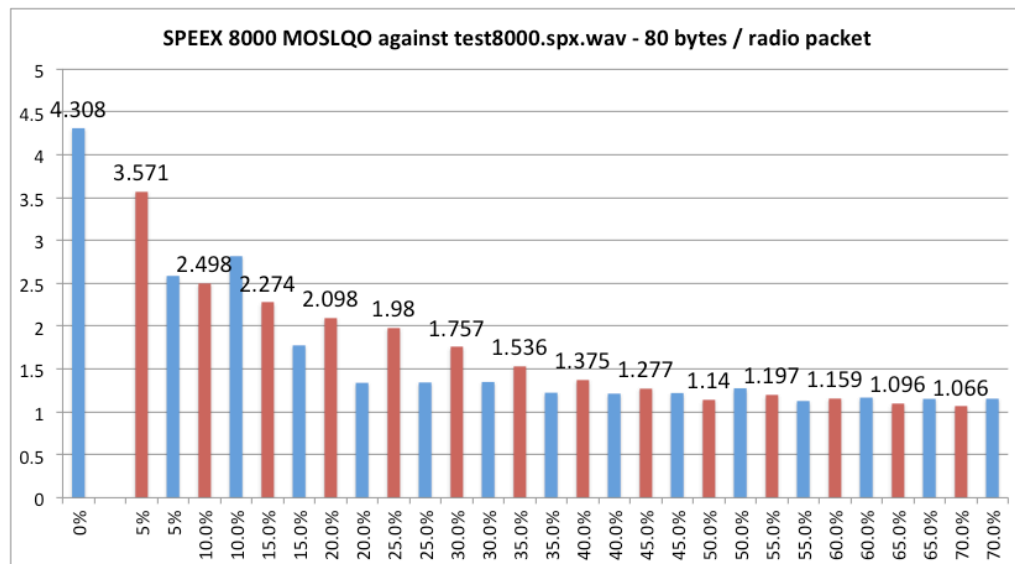


Figure 24: MOSLQO value for 8kbps speex codec as packet loss rate is varied, A4 level

## Codec2

Given the low bit rate of `codec2`, we will perform the tests with 2400 and 3200 bit rate as they provide higher audio quality. The `codec2` encoder works as follows: (i) the raw audio data is segmented into 160 8-bit samples, representing 20ms of audio; (ii) these 160 bytes are encoded into 6 bytes or 8 bytes according to the final bit rate, i.e. 2400bps or 3200bps. 3 framing bytes are then added to the audio data for transmission as illustrated by figure 25. The receiver uses the first two framing bytes (0xFF0x55) to recognize an audio packet. Then a sequence number can be used to detect packet losses.

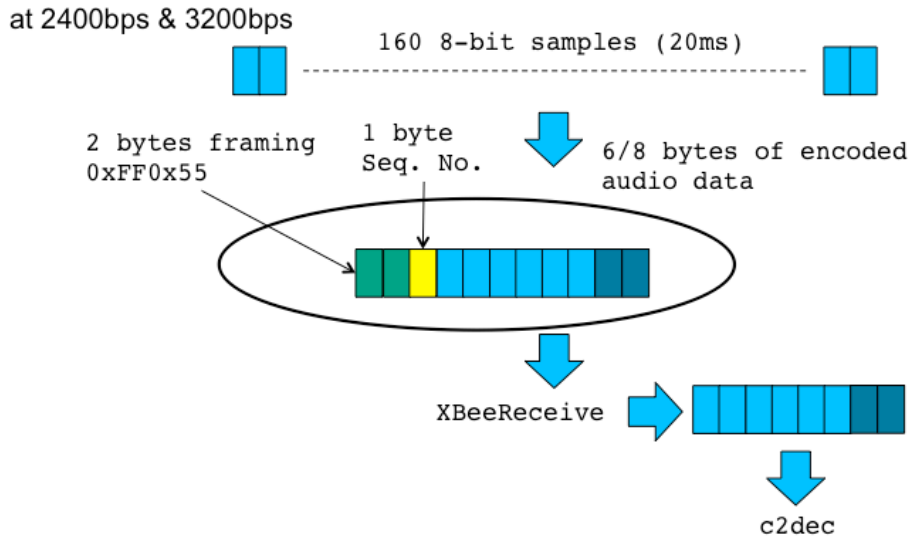


Figure 25: `codec2` audio data at 2400bps and 3200bps

Figure 26 shows for `codec2` at 2400bps the case when 1 audio frame is sent in 1 radio packet and the MOSLQO value is determined according to the packet loss rates. Again, this will be referred to as A1 aggregation case and the real payload is 6 audio bytes with 3 framing bytes, giving a total payload of 9 bytes.

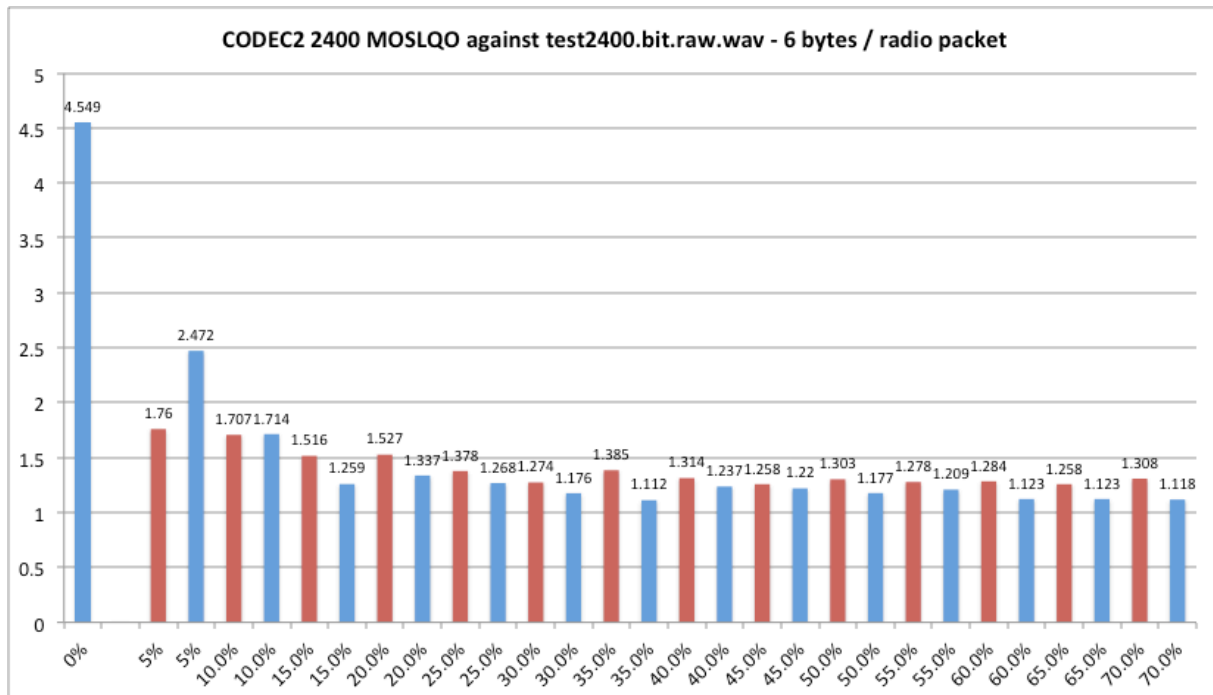


Figure 26: MOSLQO value for 2400bps `codec2` as packet loss rate is varied, A1 level

Once again, the first blue bar represents the loss-free case. Therefore, we have the maximum MOSLQO value when compared to the original file. When a packet is dropped/lost, the receiver can simply either ignore it, or it can detect the packet loss (with gap in sequence number for instance) and fill-in missing data with a pre-defined value prior to injection into the `codec2` decoder. Again the red bars represent the case where missing data is ignored, while the blue bars are for the case when the receiver detects the packet losses. We found empirically that for 2400-bit rate, a filling value of 0x77 does give good results. Better values may be possible but this is out of the scope of this study.

Figure 27, 28 and 29 respectively show the MOSLQO value for A2, A4 and A6 aggregation.

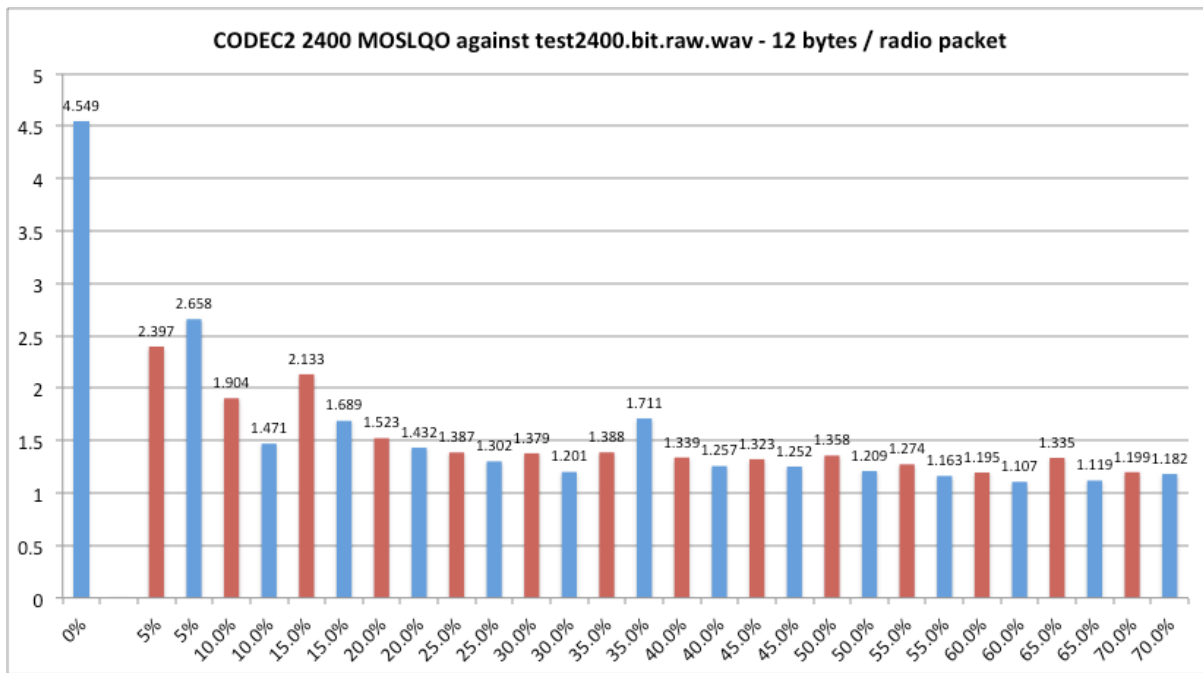


Figure 27: MOSLQO value for 2400bps codec2 as packet loss rate is varied, A2 level

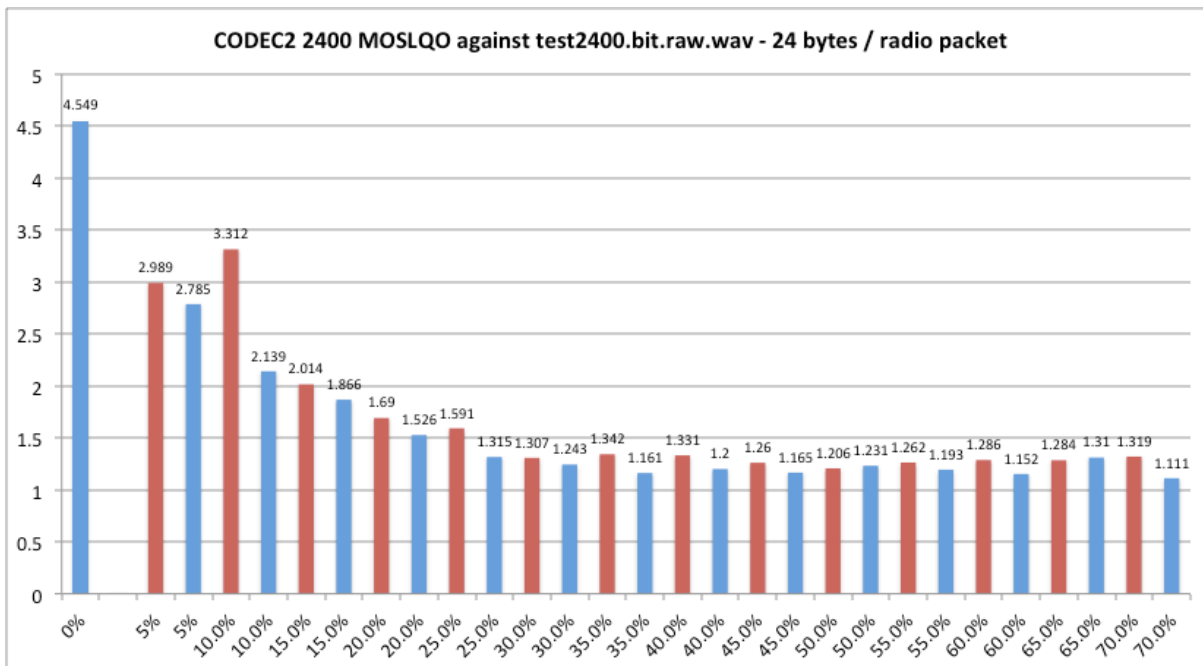


Figure 28: MOSLQO value for 2400bps codec2 as packet loss rate is varied, A4 level

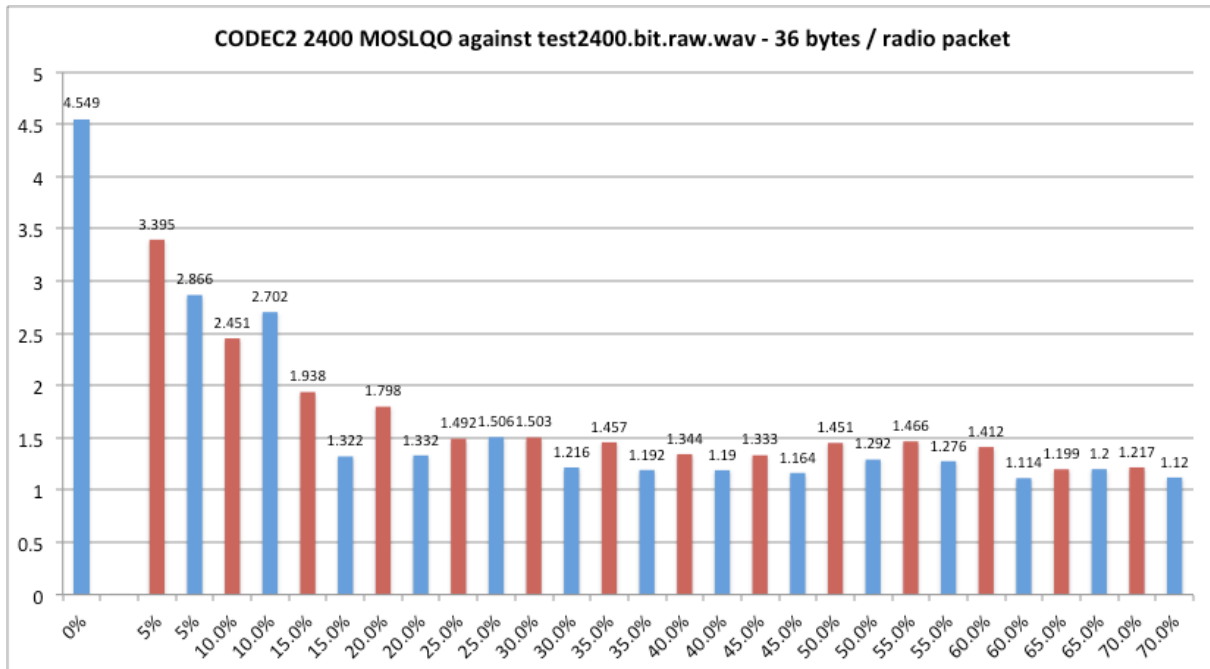


Figure 29: MOSLQO value for 2400bps codec2 as packet loss rate is varied, A6 level

Again, we observed that a packet loss rate up to 20% still provide a sufficient quality for an easy understanding of the speech. Over 20% packet loss rates, the audio quality is very degraded due to the very low bit rate of `codec2`.

Figure 30 and 31 respectively show the MOSLQO value for `codec2` 3200bps with A6 and A7 aggregation. Recall that `codec2` 3200bps uses 8-byte audio frames. Here, we observed that a packet loss rate up to 30% still provide a sufficient quality for an easy understanding of the speech.

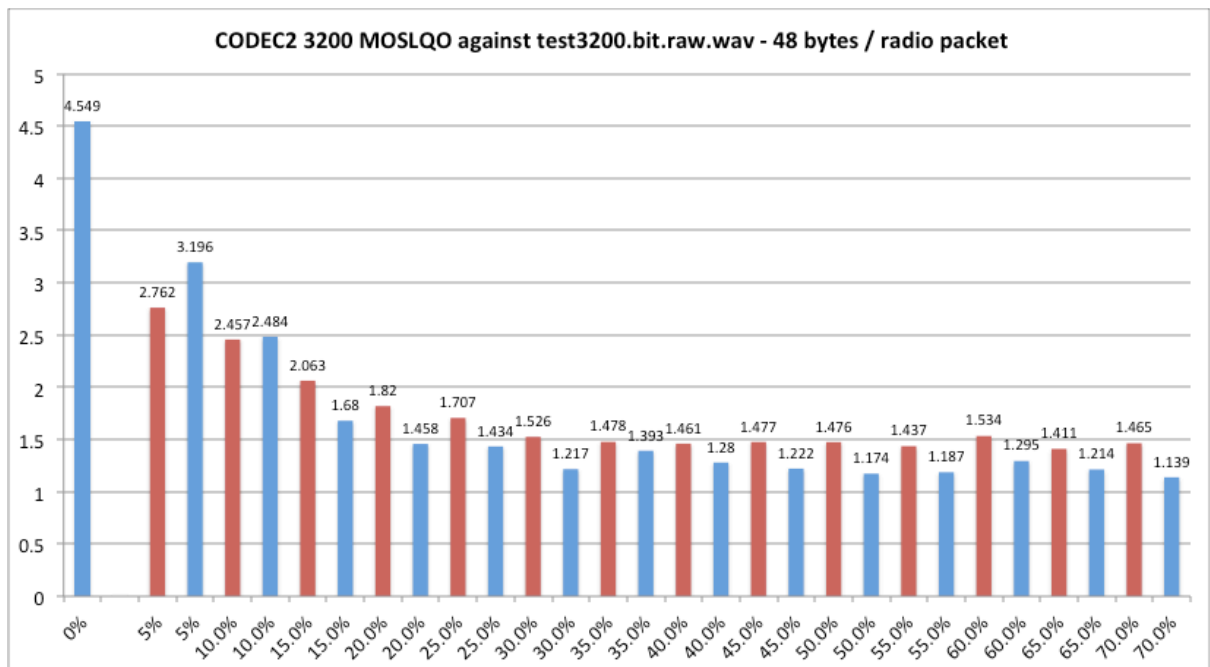


Figure 30: MOSLQO value for 3200bps codec2 as packet loss rate is varied, A6 level



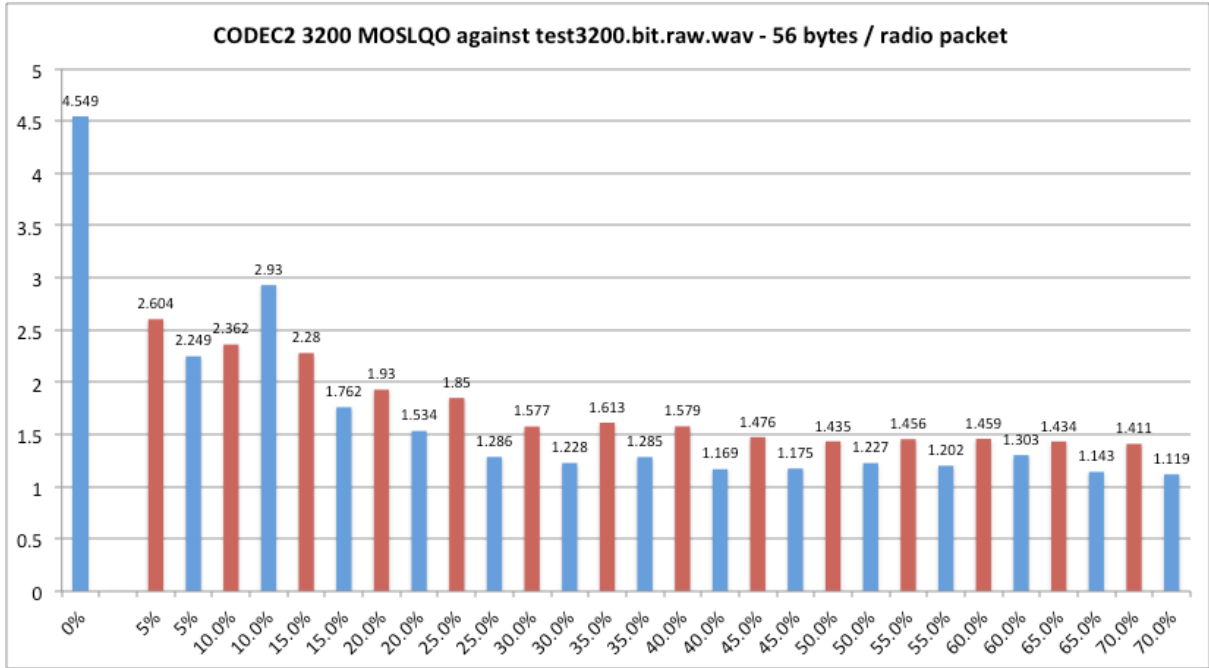


Figure 31: MOSLQO value for 3200bps codec2 as packet loss rate is varied, A7 level

### Summary

Table VII below summarizes the main results for the audio quality under packet losses.

Codec	Maximum packet loss rate for speech understanding
Raw 4KHz & 8KHz	50%
Speex 8000bps	35%
Codec2	
2400bps	20%
3200bps	30%

Table VII : summary of the maximum packet loss rate for understanding the speech

## 8. ENERGY indicators

Energy consumption is an important criterion to take into account as continuous audio sampling and transmission is very demanding. However, in the context of the EAR-IT test-beds where most of IoT nodes have recharging capabilities, the important issue is to define how long and how often audio data must be provided to the decision makers.

Therefore, the discussion that will be presented in this deliverable is very limited and deliverable 1.3 will present in more detail the energy consumption study of the various audio and network hardware elements to determine how long they can provide the acoustic services without tempering the other deployed services.

Minimum requirements in terms of acoustic data needs largely depend on the targeted applications. In an on-demand audio streaming scenario, when audio samples are requested by a human operator on emergency, it seems reasonable to have several minutes of streaming capability for an audio node. 10 to 15 minutes per audio node can then be considered as a minimum requirement.

For relay nodes, as they can be shared for multiple audio streaming sessions, they should be able to sustain about 1 hour of relaying features.

In deliverable 1.3, we will perform energy measures on the developed hardware to verify whether these requirements could be met.

## 9. Conclusions and summary of main results

In this deliverable the minimum requirements for use of acoustic sensors on the various EAR-IT test-beds based on WSN and IoT nodes with IEEE 802.15.4 radio technology are investigated. We presented our development on the targeted hardware to provide acoustic features and we defined performance indicators that are categorized into 3 categories:

1. Network performance indicators (NETWORK)
2. Audio quality indicators (AUDIO),
3. Energy indicators (ENERGY).

In the NETWORK category, we presented the minimum requirements at the audio source node and at the relay nodes in terms of **minimum sending rate**, **minimum relaying rate** and the **impact of buffer capacity** on the packet drop rate.

Regarding the AUDIO category, according to the 3 selected audio codecs (*raw*, *speex* and *codec2*) we determined the impact of packet size and packet losses on the audio quality and using the ITU-T PESQ benchmark tool suite to determine the MOS (Mean Opinion Score) value, we quantified the resulting audio quality. We then were able to give some indications on the **maximum supported packet loss rate** for still providing an understandable audio stream.

For the ENERGY category, the discussion we will have in this deliverable is very limited and deliverable 1.3 will present in more details the energy consumption study of the various audio and network hardware elements to determine how long they can provide the acoustic services without tempering the other deployed services.

The various results on minimum requirements are shown again below.

### NETWORK: minimum sending/relaying rate

Codec	Minimum sending/relay rate
Raw 4KHz 8KHz	100 bytes every 25ms 100 bytes every 12.5ms
Speex 8000bps A1 A2 A3 A4	24 bytes every 20ms 48 bytes every 40ms 72 bytes every 60ms 96 bytes every 80ms
Codec2 2400bps A1 . . An ( $1 \leq n \leq 11$ ) 3200bps A1 . . An ( $1 \leq n \leq 9$ )	9 bytes every 20ms . . 9*n bytes every n*20ms 11 bytes every 20ms . . 11*n bytes every n*20ms

## NETWORK: buffer size & packet drop relationship at relay nodes

Time before packet drop due to a full receive buffer when using the WaspMote audio with both WaspMote and TelosB relay nodes. Q is the amount of available buffer in bytes.

WaspMote audio, WaspMote & TelosB relay nodes				
Q	4KHz/W	8KHz/W	4KHz/T	8KHz/T
1000	0.33	0.14	2.33	0.23
1500	0.49	0.21	3.50	0.34
2000	0.65	0.28	4.67	0.45
2500	0.81	0.35	5.83	0.56
3000	0.98	0.42	7.00	0.68
3500	1.14	0.49	8.17	0.79
4000	1.30	0.57	9.33	0.90
4500	1.46	0.64	10.50	1.02
5000	1.63	0.71	11.67	1.13

Time before packet drop due to a full receive buffer when using the AdvanticSys TelosB audio board with WaspMote relay node. Q is the amount of available buffer in bytes.

TelosB audio board, WaspMote relay node				
Q	A1	A2	A3	A4
1000	1.27	1.81	2.56	3.40
1500	1.91	2.72	3.84	5.10
2000	2.54	3.63	5.11	6.79
2500	3.18	4.53	6.39	8.49
3000	3.82	5.44	7.67	10.19
3500	4.45	6.35	8.95	11.89
4000	5.09	7.25	10.23	13.59
4500	5.72	8.16	11.51	15.29
5000	6.36	9.07	12.79	16.99



When using TelosB relay nodes, theoretically, it can relay faster than the packet inter-arrival time of the TelosB audio board. Therefore we have a system where buffers are not needed.

## AUDIO: maximum supported packet loss rate

Codec	Maximum packet loss rate for speech understanding
Raw 4KHz & 8KHz	50%
Speex 8000bps	35%
Codec2	
2400bps	20%
3200bps	30%

# ANNEX.A: Review of software environment, tools and test hardware

1





## Development environments

- Linux-based systems for higher flexibility and better interoperability
  - most of software tools are targeted for Unix
  - most of gateways devices are Linux-based (Meshlium, Beagle, Rasperry,...)
- When possible, avoid Java development and privilege C, C++ and scripts (shell, python)

*the sounds of smart environments*

2





## Standard IDE & software tools

- Libelium WaspMote
  - Libelium IDE (Arduino-based) & API development environment
- AdvanticSys TelosB
  - TinyOS 2.1.2 development environment
- Audio
  - Codec2 software ([www.codec2.org](http://www.codec2.org)): c2enc, c2dec
  - Speex software ([www.speex.org](http://www.speex.org)): speexenc, speexdec
  - sox and play package (Linux)
- Serial & frame analysis
  - minicom, cutecom
  - wireshark

*the sounds of smart environments*

3

3





## Customized speex audio tools

- Simple « pure » speex audio decoder without any header
  - Modified version of speex's `sampledec.c`
  - `speex_sampledec_wframing` : expects framing bytes
  - `speex_sampledec_nframing` : no framing bytes
- To get a « pure » speex audio encoded file without any header
  - Modified version of `speexdec.c` (yes `speexdec.c` and not `speexenc.c`) compatible with speex's `sampledec.c`

*the sounds of smart environments*

4



## Development of dedicated tools

- Serial tools to read host computer serial port
  - `XBeeReceive` (C language)
  - `SerialToStdout` (python script)
    - 115200 baud version
    - 38400 baud version
- Communication tool to send control command packets
  - `XBeeSendCmd` (C language)
- Communication tool to send binary files
  - `XBeeSendFile` (C language)

*the sounds of smart environments*

# XBeeReceive

- XBeeReceive
  - Main target is 802.15.4 XBee-based gateway
  - Translates XBee API frame
  - Reads from the serial port : /dev/ttyUSB0, /dev/ttyS0, ...
  - Reconstructs file in binary mode (handles packet losses)
    - Assumes each packet with 4 bytes header: 2 bytes for file size & 2 bytes for offset
  - Can write to Unix stdout & can act as a transparent serial replacement
  - Can act in a data stream fashion: no header for packets



```

USAGE: ./XBeeReceive -baud b -p dev -B -ap0 -v val -stdout -stream file_name
USAGE: -baud, set baud rate, default is 38400
USAGE: -p /dev/ttyUSB1
USAGE: -B indicates binary mode. Assumes 4-bytes header for each pkt (that will be removed)
USAGE: -framing expect for framing bytes 0xFF0x55 for binary data
USAGE: -ap0, indicates an XBee in AP mode 0 (transparent mode) so do not decode frame structure
USAGE: -v 77, use 0x77 to fill in missing value in binary mode
USAGE: -stdout, write to stdout for pipe mode in binary mode
USAGE: -stream, assumes no header & write to stdout for pipe mode in binary mode
USAGE: file_name, name for saving binary file

```

5

*the sounds of smart environments*

# SerialToStdout.py

- Simple python script to read serial port when no translation is needed
- Change baud rate and port as needed

```

import serial
import sys

ser = serial.Serial('/dev/ttyUSB0', 38400, timeout=0)

# flush everything that may have been received on the port to make sure
# that we start with a clean serial input
ser.flushInput()

while True:
    out = ''
    sys.stdout.write(ser.read(1024))
    sys.stdout.flush()

```

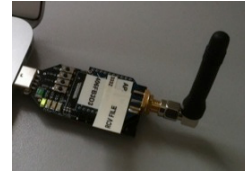
- SerialToStdout.py can be use instead of XBeeReceive with an XBee in transparent mode

6

*the sounds of smart environments*



- XBeeSendCmd
  - Main target is 802.15.4 XBee-based gateway
  - Send ASCII command with Xbee
  - Can be used to sent remote AT command to other Xbee module
  - Support DigiMesh firmware
  - Example
    - XBeeSendCmd -addr 0013a2004069165d "/@D0100#"



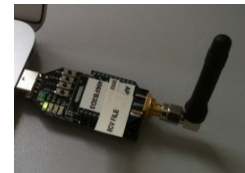
```

USAGE: ./XBeeSendCmd -p dev [-L][-DM][-at] -tinynos -tinynos_amid id_hex -mac|-net|-addr|-b message
USAGE: -p /dev/ttyUSB1
USAGE: -mac 0013a2004069165d HELLO
USAGE: -net 5678 HELLO
USAGE: -addr 64_or_16_bit_addr HELLO
USAGE: -b HELLO
USAGE: -at to send remote AT command: -at -mac 0013a2004069165d ATMM
USAGE: -L insert Libelium API header
USAGE: -DM to specify DigiMesh firmware
USAGE: -tinynos to forge a TinyOS ActiveMessage compatible packet (0x3F0x05 are inserted)
USAGE: -tinynos_amid 6F, to set the ActiveMessage identifier to 0x6F (0x05 is the default)
  
```

the sounds of smart environments

7

- XBeeSendFile
  - Main target is 802.15.4 XBee-based gateway
  - Send binary files with Xbee with controlled timing
  - Can use any packet size between 1 and 100 bytes
  - Can insert framing bytes, can introduce packet losses



```

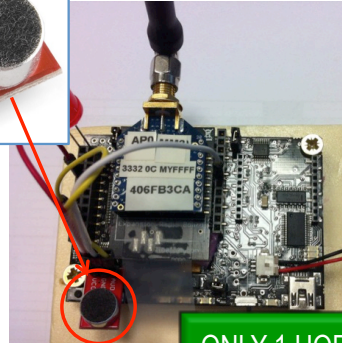
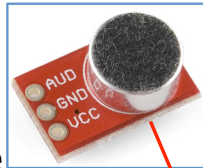
USAGE: ./XBeeSendFile -baud baudrate -p dev -sensor -timing tpkt_us tserialbyte_us tafterradio_us -nw -fake
-drop rate -v val -fill -pktf -pktd -pktf -size s -stdout -mac|-net|-addr|-b file
USAGE: -baud 125000, 38400 by default
USAGE: -sensor, will send image pkt to a sensor sniffer
USAGE: -framing, will use framing bytes 0xFF0x55+SN for binary packets (e.g. audio)
USAGE: -timing 50000 20 25000 by default
USAGE: -nw, do not wait for TX status response
USAGE: -fake, emulate sending. Will write in fakeSend.dat
USAGE: -drop 50, will introduce 50 of packet drop. Useful with -fake
USAGE: -v 77, use 0x77 to fill in missing bytes in lost packet
USAGE: -fill, will fill missing bytes
USAGE: -pktd, display generated XBee frames
USAGE: -pktf, generate a pkt list file
USAGE: -size 50, set packet size to 50 bytes
USAGE: -stdout, write to stdout for pipe mode
USAGE: -mac 0013a2004069165d
USAGE: -net 5678
USAGE: -addr 64_or_16_bit_addr, set either 64-bit or 16-bit dest. address
USAGE: -b
  
```

the sounds of smart environments

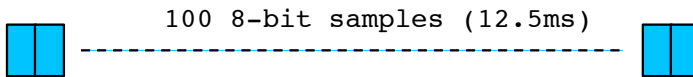
8

# WaspMote+XBee in raw mode

- Electret mic with amplifier
- XBee in AP0 mode (transparent mode)
- 8-bit 4Khz sampling gives 32000bps
- 8Khz sampling gives 64000bps, requires custom API



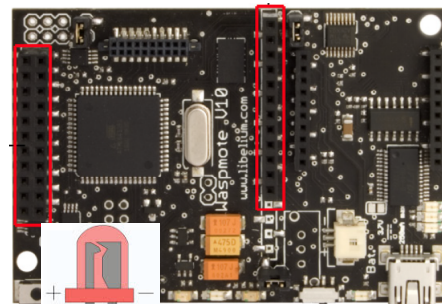
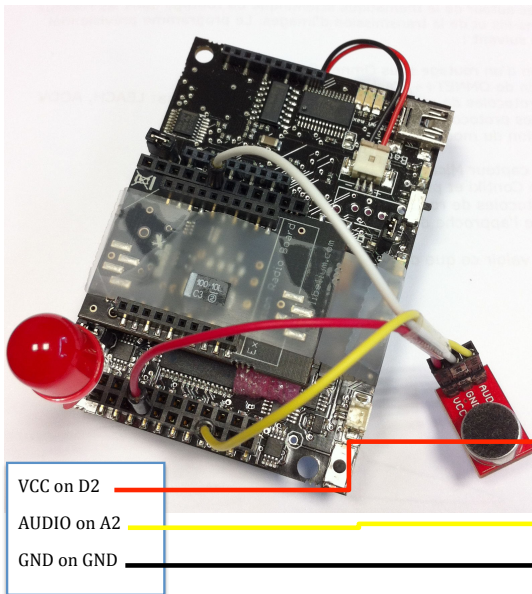
ONLY 1 HOP!



9

*the sounds of smart environments*

# Details of pin connection

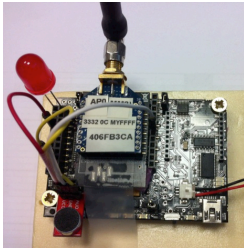


DIGITAL8	•	GND	AUX-SERIAL-1-TX
DIGITAL6	•	DIGITAL7	AUX-SERIAL-1-RX
DIGITAL4	•	DIGITAL5	AUX-SERIAL-2-RX
DIGITAL2	•	DIGITAL3	AUX-SERIAL-2-TX
RESERVED	•	DIGITAL1	RESERVED
ANALOG6	•	ANALOG7	GND
ANALOG4	•	ANALOG5	MUX_RX
ANALOG2	•	ANALOG3	MUX_TX
SENSOR POWER	•	ANALOG1	SENSOR POWER
GPS POWER	•	5V SENSOR POWER	SCL
SDA	•	SCL	SDA

VCC on D2  
 AUDIO on A2  
 GND on GND

10

*the sounds of smart environments*



```
void loop() {
  val = analogRead(ANALOG2) ; // read analog value
  val8bit = ((val >> 2) ) ; // convert into 8 bit

  // write on UART1, need an XBee module
  // with AP mode 0

  serialWrite(val8bit,1);
}
```



Xbee GW

With XBee GW also in AP0 mode

```
4KHz sampling
> XBeeReceive -baud 38400 -ap0 -stdout dumb.dat | play --buffer 50 -t raw -r 4000 -u -1 -

8KHz sampling
> XBeeReceive -baud 125000 -ap0 -stdout dumb.dat | play --buffer 50 -t raw -r 8000 -u -1 -

Save raw data for off-line playing
> XBeeReceive -baud 38400 -ap0 -stdout dumb.dat > test.raw
> play -t raw -r 4000 -u -1 test.raw
```

Alternatively using SerialToStdout python script, at 38400 baud only

```
> python SerialToStdout | play --buffer 50 -t raw -r 4000 -u -1 -
```

11

*the sounds of smart environments*

- The receiving XBee module may need to be in packet mode (AP2) due to deployment constraints
- Adds overhead of XBee API frame decoding: 8KHz sampling may be not supported

```
4KHz sampling
> XBeeReceive -baud 38400 -stream dumb.dat | play --buffer 50 -t raw -r 4000 -u -1 -
```

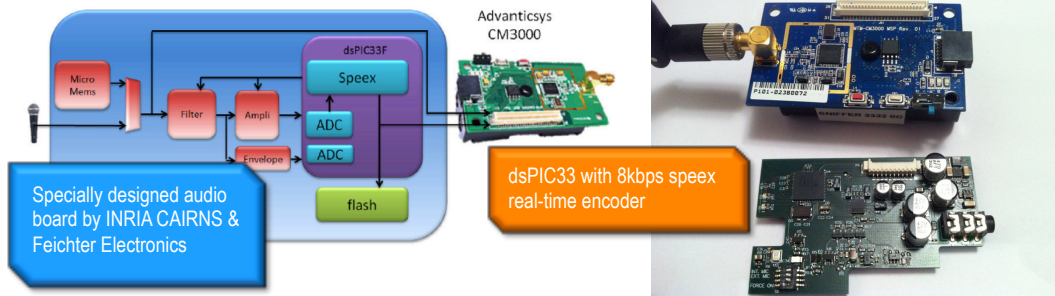
```
Save raw data for off-line playing
> XBeeReceive -baud 38400 -stream dumb.dat > test.raw
> play -t raw -r 4000 -u -1 test.raw
```

12

*the sounds of smart environments*

# Multi-hop audio solution

- Use dedicated audio board for sampling/storing/encoding at 8kbps

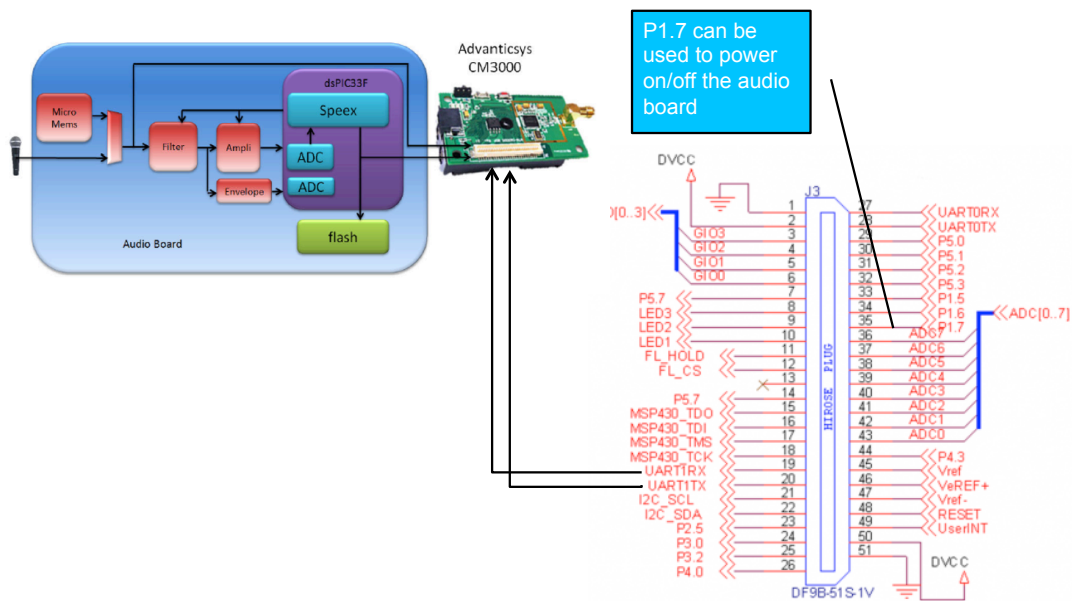


- Allows for multi-hop, encoded audio streaming scenarios

13

*the sounds of smart environments*

# Details of pin connection



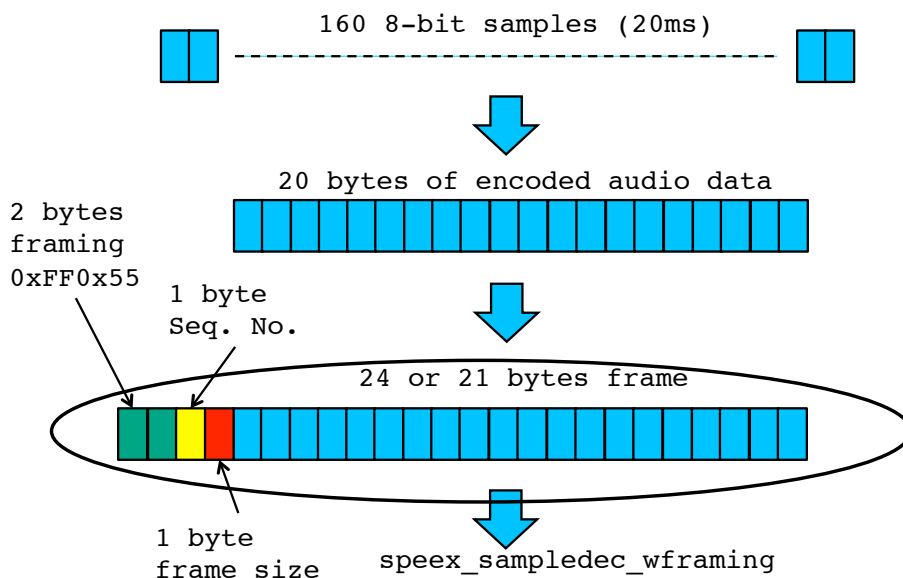
14

*the sounds of smart environments*

- The audio board captures 160 bytes (20ms) of raw audio and uses speex codec at 8kbps to produce 20 bytes to encoded audio data
- It sends the encoded audio data through an UART line to the host micro-controller
- The host micro-controller receives the encoded data and sends them wirelessly to the next hop
- The last hop is a base station that will forward the encoded audio into a speex audio decoder
- Output of the speex audio decoder is in raw format that can be feed into a player (play)

15

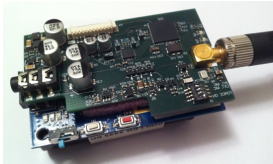
*the sounds of smart environments*



16

*the sounds of smart environments*

# AdvanticSys+audio board



```

async event void UartStream.receiveDone(uint8_t* buf,
uint16_t len, error_t error){
    post sendMsg();
}
    
```



With AdvanticSys base station (115200 baud)

```
> python SerialToStdout | speex_sampledec_wframing | play --buffer 100 -t raw -r 8000 -s -2 -
```

With XBee GW in AP0 mode



```
> XBeeReceive -baud 38400 -B -ap0 -stdout dumb.dat | speex_sampledec_nframing |
play --buffer 100 -t raw -r 8000 -s -2 -
```

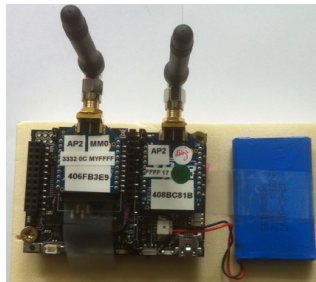
With XBee GW in AP2 mode (pkt mode)

```
> XBeeReceive -baud 38400 -B -stream dumb.dat | speex_sampledec_nframing |
play --buffer 100 -t raw -r 8000 -s -2 -
```

17

*the sounds of smart environments*

# Relay nodes



**LIBELIUM  
WASPMOTE**



**ADVANTICSYS  
CM5000, CM3000**

Fully configurable:

- Destination node
- Additional relay delay
- Clock synchronization

```

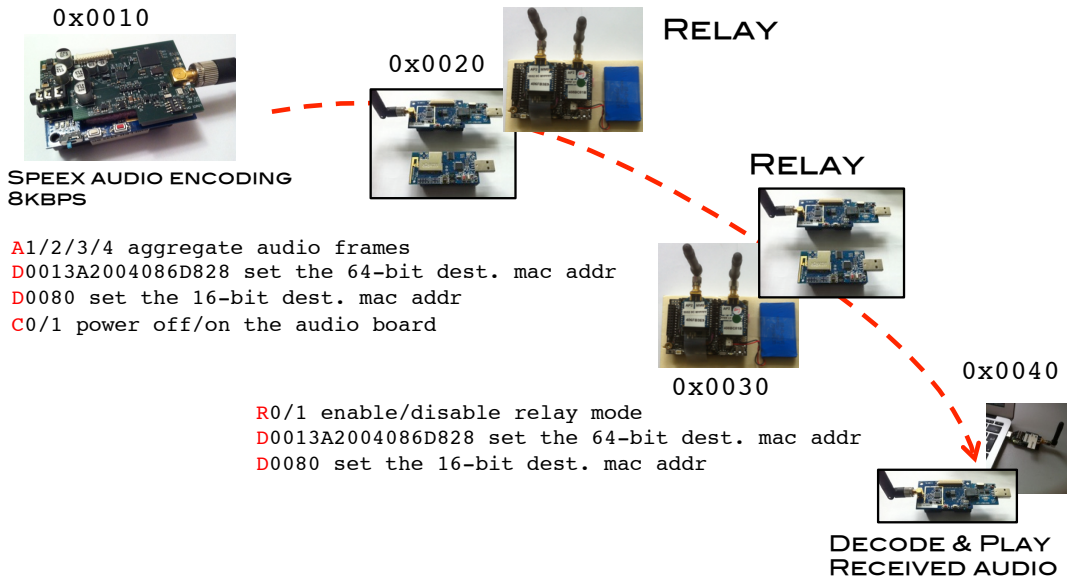
R0/1 enable/disable relay mode
D0013A2004086D828 set the 64-bit dest. mac addr
D0080 set the 16-bit dest. mac addr
    
```

18

*the sounds of smart environments*



# Multi-hop test-bed w/audio board



19

*the sounds of smart environments*

# Generic & controlled sender

Use a generic sender node to test with a larger variety of audio codec: store encoded audio file on SD card

Do not need specific audio encoding hardware to test quality of streaming encoded audio data

Fully configurable:

- Destination node
- Clock synchronization
- File to send
- Size of packet chunk
- Inter-packet delay
- Binary/Stream mode

Ftest2400.bit#B#  
0013A200406FB3E9

20

*the sounds of smart environments*



0x0010

0x0020

RELAY

RELAY

0x0030

0x0040

DECODE & PLAY RECEIVED AUDIO

T130 transmit with inter pkt time of 130ms  
 Z50 set the pkt size for binary mode  
 Ftest2400.bit set the file name to test2400.bit  
 D0013A2004086D828 set the 64-bit dest. mac addr  
 D0080 set the 16-bit dest. mac addr  
 B or S set to binary mode/set to stream mode

All commands must be prefixed by « /@ »  
 and ended/separated by « # »

/@T130#, /@Ftest2400.bit#B#

21

*the sounds of smart environments*

- Use codec2/speex encoding software to produce encoded audio file
- Store encoded audio file (.bit/.spx) on SD card
- Configure the generic sender for sending the encoded audio file
  - Define packet size
  - Determine inter-packet send time
- Receive the encoded audio stream, decode the data and determine audio quality

22

*the sounds of smart environments*



## Produce encoded audio file: codec2

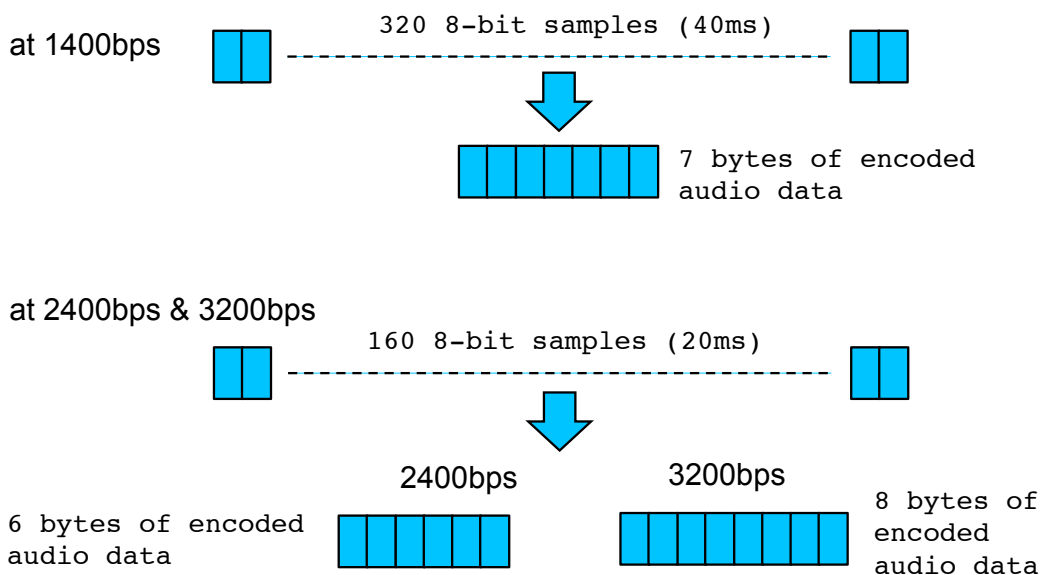
- Initial file: `test.raw` in 16-bit, signed
- Use `sox` to get 16-bit, signed if your raw file is not in this format
- Encode at 2400bps with
  - `c2enc 2400 test.raw test2400.bit`
- Store `test2400.bit` on SD card

23

*the sounds of smart environments*



## Codec2 encoding

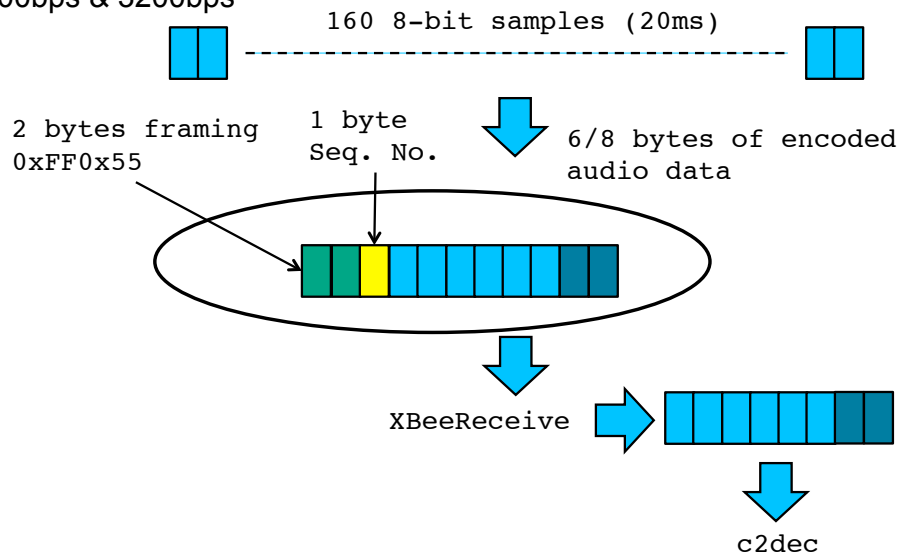


24

*the sounds of smart environments*

# Codec2 at 2400bps & 3200

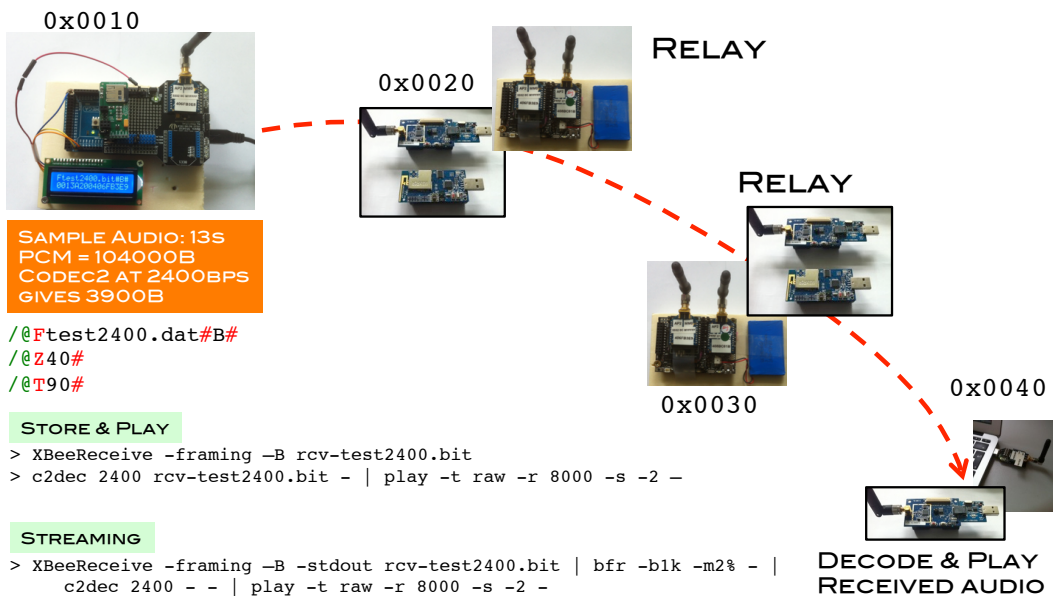
at 2400bps & 3200bps



25

*the sounds of smart environments*

# Multi-hop tests with codec2



26

*the sounds of smart environments*

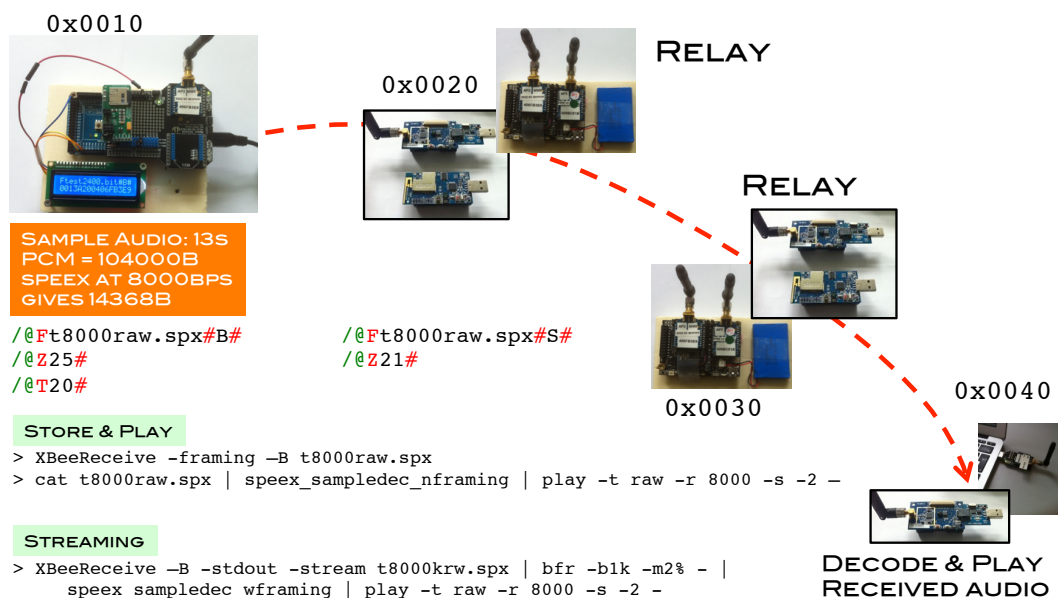
# Produce encoded audio file: speex

- Initial file: `test.raw` in 8-bit unsigned or 16-bit signed
- Encode at 8000bps with
  - `speexenc --8bit --bitrate 8000 test.raw test8000.spx`
- Produce a raw speex byte stream with modified version of `speexdec`
  - `speexdec test8000.spx > t8000raw.spx`
- Store `t8000raw.spx` on SD card

27

*the sounds of smart environments*

# Multi-hop tests with speex



28

*the sounds of smart environments*

**speex at 8kbps on slow relay nodes**

160 bytes (20ms)

20 bytes of encoded audio data

Add framing bytes

1 2 3 4 5 6 7 8

2 3 4 6 8

A6 aggregate audio frames

Capture 6 audio frames (120ms) but only send 4

Need to be able to relay 96-byte pkt every 120ms

29

*the sounds of smart environments*

**Apply packet loss rate**

- Use XBeeSendFile to control
  - Timing between packet sending
  - Packet loss probability

Codec2 2400bps, series of 6-byte encoded audio packets

1 2 3 4

> XBeeSendFile -fake -drop 25 -stdout test2400.bit > test2400-25loss.bit

1 3 4

> XBeeSendFile -fake -v 77 -fill -drop 25 -stdout test2400.bit > test2400-25loss-fill.bit

1 2 3 4

77 77 77 77 77 77

30

*the sounds of smart environments*

## ANNEX.B: Future developments on targeted hardware platforms

### ***Libelium WaspMote + audio board***

1. Use the developed audio board that has been developed initially for the AdvanticSys on the Libelium WaspMote

### ***AdvanticSys TelosB + MTS SE1000***

1. Use the MTS\_SE1000 sensor board that includes a small-amplified electret MIC to test whether 4KHz sampling can be realized with simultaneous transmission of raw audio data as the AdvanticSys mote can normally sustain a sending throughput of 48kbps. See figure below of the MTS\_SE1000 sensor board.



## References

- [802154] IEEE Std 802.15.4™-2006.
- [ADVAN] [http://www.advanticsys.com/shop/wireless-sensor-networks-802154-mote-modules-c-7\\_3.html](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](http://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.
- [TELOSB] [www.willow.co.uk/html/telosb\\_mote\\_platform.html](http://www.willow.co.uk/html/telosb_mote_platform.html) and/or <http://bullseye.xbow.com:81/Products/productdetails.aspx?sid=252>
- [TINYOS] The TinyOS operating system. <http://www.tinyos.net/>
- [XBeeDigi] XBee®/XBee-PRO® RF Modules product manual (90000982\_G), Digi International Inc. August 1, 2012.