Streaming the Sound of Smart Cities: Experimentations on the SmartSantander test-bed

Congduc Pham
University of Pau, LIUPPA Laboratory
congduc.pham@univ-pau.fr

Philippe Cousin
Easy Global Market
phlippe.cousin@eglobalmark.com

***Abstract*—Smart Cities have emerged as an efficient infrastructure to contribute to so-called global sensing or situation-awareness applications. One example of large scale deployment of sensors in the city is the SmartSantander test-bed. Most of the deployment so far propose traditional scalar physical measures such as temperature or luminosity for a number of environment-related applications. The EAR-IT project moves a step further and proposes large-scale "real-life" experimentations of intelligent acoustics for supporting high societal value applications. One scenario that will be demonstrated is an on-demand acoustic data streaming feature for surveillance systems and management of emergencies. In this paper, we will present experimentations on streaming encoded acoustic data on low-resources devices. We will highlight the main sources of delays assuming no flow control nor congestion control to determine the best case performance level and will demonstrate that streaming acoustic data can be realized in a multi-hop manner on the SmartSantander infrastructure.**

***Index Terms*—Smart Cities, Sensor networks, Audio streaming, surveillance**

I. Introduction

In the last few years, the research efforts in the field of Wireless Sensor Networks (WSN) have shown high potentials for surveillance applications and have paved the way to nowadays so-called ubiquitous/global sensing and smart cities paradigm that extends WSN to a more generic Internet-of-Thing (IoT) concepts. A number of leading projects on global sensing and smart cities have been launched recently and the SmartSantander infrastructure [1] is probably one of the most important one in term of deployment scale and in number of hosted applications test-beds and project. One of the hosted project is the EAR-IT project [2] 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. One scenario that will be demonstrated is an on-demand acoustic data streaming feature for surveillance systems and management of emergencies. Figure 1 depicts the EAR-IT context with a 2-tier architecture of sensing nodes. The first tier consists of a limited number of powerful Acoustic Processing Units (APU) with advanced analysis capabilities to accurately detect events of interest. The second tier is composed of a large number of low-cost, low-power sensing devices, noted IoT nodes in the figure, that can be used in a complementary way to capture, on an on-demand basis, acoustic data that will be streamed to the central control system using other IoT nodes as relay nodes. Delay can be an important factor as the on-demand scenario is typically intended for a human operator requesting acoustic data on well-identified parts of the city.

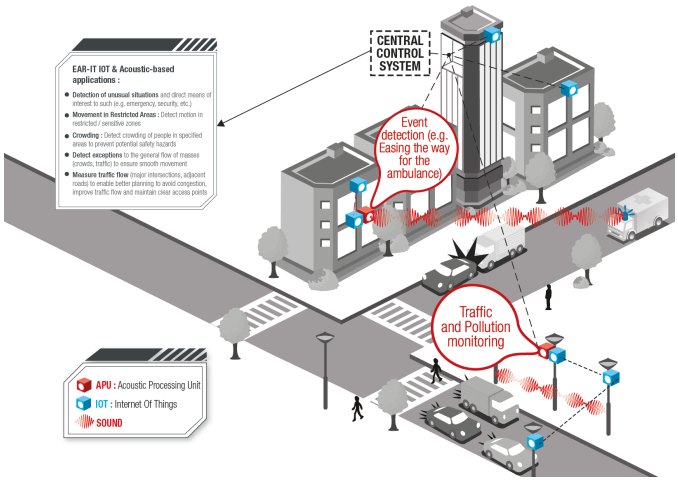

Fig. 1. EAR-IT context on-demand audio data streaming

Although the acoustic capture system on the numerous IoT nodes are not as efficient and powerful than the one on the APU, the advantage of IoT nodes is their density that provides a large-scale coverage of the city. Therefore a human operator could request acoustic data from a set of IoT nodes to improve its understanding of the emergency. Note that the central control system depicted in figure 1 is actually a gateway node that manages a number of APU and IoT nodes. Many gateways are deployed across the test-bed and a gateway is connected to the Internet with a large bandwidth network technology: WiFi, wired Ethernet or 3G depending on what is available. We will then consider that the difficult part is to stream acoustic data from an IoT to its corresponding gateway, and once the data has reached the gateway, powerful and traditional streaming tool/software/protocol could be used to transfer the acoustic data to the final destination.

There have been studies on multimedia sensors but few of them really consider timing on realistic hardware constraints for sending/receiving flows of packets [3], [4], [5], [6]. In this paper, we will present experimentations on streaming encoded acoustic data on low-resources devices. We will highlight

the main sources of delays assuming no flow control nor congestion control to determine the best case performance level. The motivation of this article is to present original experiments of acoustic surveillance systems in a real large-scale test-bed of deployed IoT nodes with 802.15.4 multi-hop connectivity.

The paper is then organized as follows: Section II reviews the SmartSantander test-bed architecture and especially the sensor node hardware. Section III presents real measures on sensor hardware and radio modules to qualify the 802.15.4 communication stacks at the application level. The motivation is to know exactly the performance level that could be obtained for streaming applications. Experimental results of multi-hop acoustic data transmissions with 802.15.4 radio modules will be presented in Section IV. We will present in this section the experimental test-bed, the developed tools for the tests and the audio codec. Conclusions will be given in Section V.

II. The SmartSantander test-bed hardware

The SmartSantander test-bed is a 3-location infrastructure project. One main location being the Santander city in north of Spain with more than 2000 nodes deployed across the city. Many information can be found on the project web site [1] but we will present in this section some key information that briefly present the main characteristics of the deployed nodes.

A. IoT nodes and gateways

IoT nodes in the Santander test-bed are WaspMote sensor boards and gateways are Meshlium gateways, both from the Libelium company [7]. Most of IoT nodes are also repeaters for multi-hops communication to the gateway. Figure 2 shows on the left part the WaspMote sensor node serving as IoT node and on the right part the gateway [1].

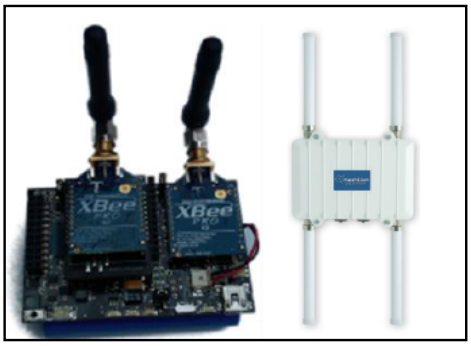

Fig. 2. Deployed IoT node (left) and gateway (right)

The WaspMote is built around an Atmel1281 micro-controller running at 8MHz with 128KB of flash memory available for the user application code. RAM memory is limited to 4KB but an SD card of 2GB can be put on the board. 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 micro-controller to the radio modules: UART0 and UART1 for the default XBee Socket and an Expansion Radio Board Socket respectively. The XBee socket can directly receive an XBee radio module from Digi International [8] that offers various connectivity technologies: 802.15.4, Digimesh/ZigBee, WiFi, 900 & 860MHz. The Radio Expansion Socket can receive a dedicated GSM/GPRS module or a radio expansion board that offers a second XBee connectivity board which is the case for nodes in the test-bed. Various connectivity combinations can therefore be realized. The WaspMote depicted in Figure 2 has 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. The gateway is built on the Linux operating system and has many connectivity possibilities: XBee 802.15.4, XBee Digimesh, WiFi, Ethernet, GSM/GPRS, Bluetooth to name the few that are readily available.

B. 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 protocols along with more advanced coordination/node discovery functions. XBee 802.15.4 offers the basic 802.15.4 [9] 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. Santander's nodes have the "pro" version set at 10mW transmit power which is the maximum allowed transmit power in Europe. With this transmit power, the module has 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 [10], [11].

The 802.15.4 module is available for experimentations (mesh 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 logs 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.

In this paper, we only consider acoustic data transmission using the 802.15.4 radio module connected to the UART1 of the WaspMote.

III. IoT node qualification

A. Sending side

One important part of our work in this paper is to take into account the real overheads and limitations of realistic sensor hardware. Most of simulation models or analytical studies only consider the frame transmission time as a source of delay. However, before being able to transmit a frame, the radio module needs to receive the frame in its transmission buffer. In

many low cost sensor platforms, which is the case for Libelium WaspMote, the bottleneck is often the interconnection between the micro-controller and the radio module. Many sensor boards use UARTs (serial line) for data transfer which data transfer rate lies somewhere between 38400bps and 230400bps for standard bit rates. Non-standard baud rates are usually possible, depending on the micro-controller master clock, and also, depending on UARTs, higher speed can be achieved. Nevertheless, in addition to the radio transmission time, one has to take into account the time needed to write data into the radio module's buffer. This time is far from being negligible as most of serial communications also adds 2 bits of overhead (1 start bit and 1 stop bit) to each 8-bit data. Therefore, with a serial data transfer rate of 230400bps, which is already fast for a sensor board UART, writing 100 bytes of application payload needs at least $100 \times 10/230400 = 4.34ms$ if the 100 bytes can be passed to the radio without any additional framing bytes. In many cases, one has to add extra framing bytes, making the 4.34ms a sort of minimum overhead to add to each packet transmission in most of UART-based sensor boards. If we consider an audio flow that requires sending a multitude of packets, we clearly see that the minimum time before 2 packet generation is the sum of the time to write frame data to the radio and the time for the radio to transmit the frame. According to the 802.15.4 standard, if we consider a unicast transmission with the initial back-off exponent BE set to 0 (default is 3), we still typically need a minimum of $5.44ms + 4.34ms = 9.78ms$ to send a single 100-byte packet if there is no error.

To highlight the importance of the time needed to write to the radio, we propose to measure the time spent in the send() function, noted t_{send}, and the minimum time between 2 packet generation, noted t_{pkt}. t_{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_{pkt} can be quite greater than t_{send}. With t_{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 will use a Libelium WaspMote as a traffic generator to send packet back-to-back with a minimum of data manipulation needed to maintain some statistics (counters) and to fill-in data into packets, and we added accurate timing of the programming API. We want to investigate the off-the-shelves performance of the WaspMote that are deployed in Santander so the UART transfer rate is kept to the default 38400 baud rate. We also use light version of the Libelium API that provides much higher performance level compared to the "full" Libelium API that additionally handles long packets with fragmentation/reassembly support. This is done at a much higher cost while being mostly unnecessary for streaming acoustic data.

Figure 3 shows the time in send() breakout for the WaspMote where we can especially see the time required to write to the radio. The time in parse_message() is the time to wait for the TX status from the XBee radio module. This is a source of improvement as it is possible to remove this overhead from the send() function in order to return faster from the function call as the XBee Arduino library does on Arduino boards [12]. In Figure 3 the sum of all the timing represents what we called t_{send}. We can see that the time needed to write to the radio does represent the main overhead of the send procedure.

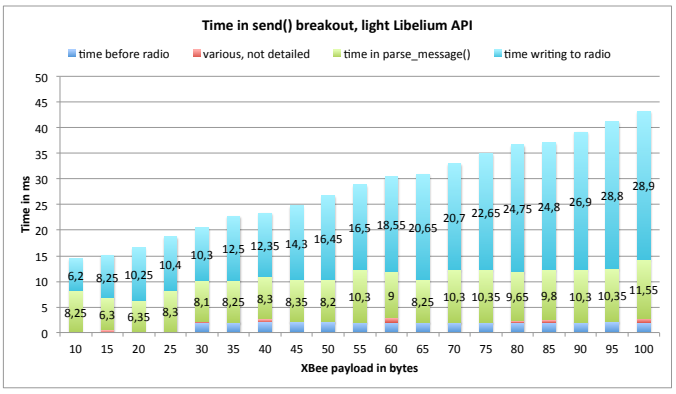

Fig. 3. Time in send() breakout, WaspMote

Figure 4 shows both t_{send} and t_{pkt} for the WaspMote. The maximum realistic sending throughput can be derived from t_{pkt} and this is depicted in figure 5.

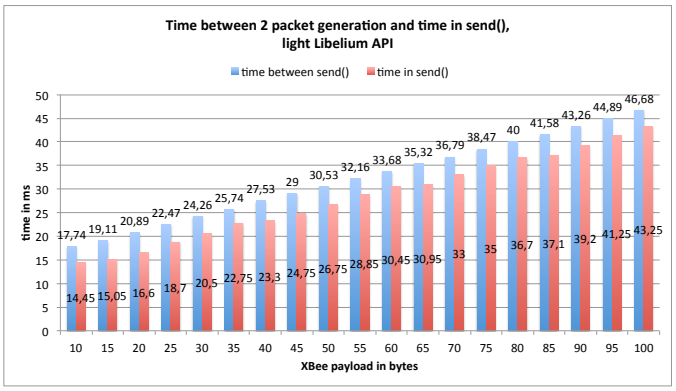

Fig. 4. Time between 2 packet generation and time in send()

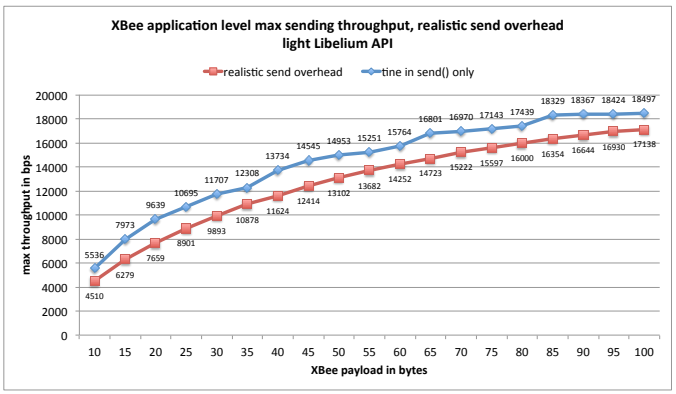

Fig. 5. Maximum realistic sending throughput

We can see that Libelium WaspMote with an unmodified communication library from Libelium can reach a maximum sending throughput of about 17100bps.

B. Taking into account the receiver side

In the next set of experiments, we use the traffic generator to generate at the sending side packets to a receiver. In general, flow control and congestion control can be implemented but any control would slow down the transmission anyway. Therefore, we are not using flow control nor congestion control but experimentally determine the minimum time between 2 packet generation at the sending side that would not overflow the receiver. Figure 6 shows the minimum time between 2 packet generation to avoid frame drops or incomplete frames at the receiver. We can see that with a receiver and the concern that packets are not arriving too fast at the receiver side, the minimum time between 2 packet generation increases from $\approx 47ms$ to $\approx 63ms$ for the maximum payload size.

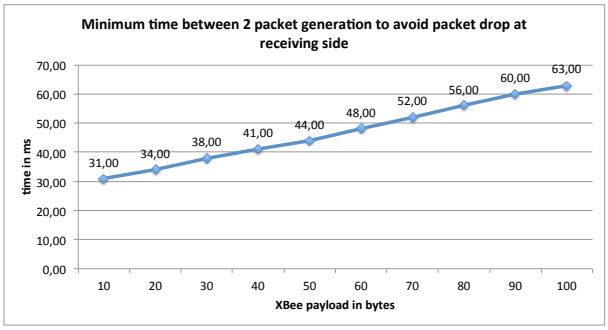

Fig. 6. Minimum time between 2 packet generation

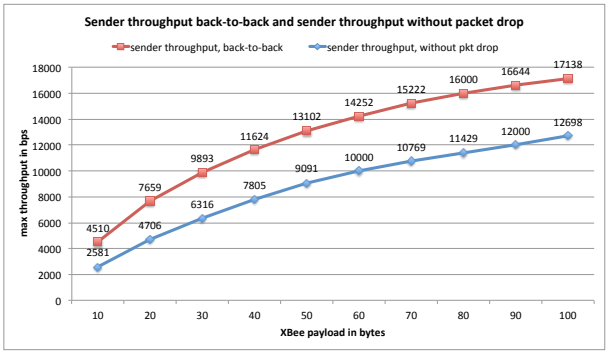

Fig. 7. Sender throughput (back-to-back) and maximum receiver throughput

Figure 7 shows the difference between the maximum sender throughput and the maximum receiver throughput which is about 12700bps. Note that we are using the standard low-level serial communication library of the WaspMote (which is based on Arduino library) except that we increased the buffer size of the UART1 to the same value than for UART0 (512 bytes). We also did not modify the queue management policy which is a drop tail behavior on receiving characters.

C. Multi-hop issues

These results are for a single hop transmission and therefore probably represent the best case. It means that these values of minimum time between 2 packet generation are really a minimum that avoids having high packet loss rate. If there are many sensors sending at the same time, congestion control with advanced QoS control and prioritization between packets should probably be needed at the cost of higher latencies which is often not tractable for streaming or near real-time applications. This is the reason we are not considering flow control nor congestion control in this paper since we want to see to which extend streaming acoustic data is feasible and at what maximum performance level.

In most WSN, data are sent from sensor nodes to a sink or base station. This sink is not always the final destination because it can also transmit the data to a remote control center, but it is generally assumed that the sink has high bandwidth transmission capabilities (such as the gateways in the Santander test-bed). Figure 8 shows a more detailed time diagram of a multi-hop transmission. We can see that all the sensor nodes along the path from the source node to the sink do have the same constraints regarding the minimum time between 2 packet generation.

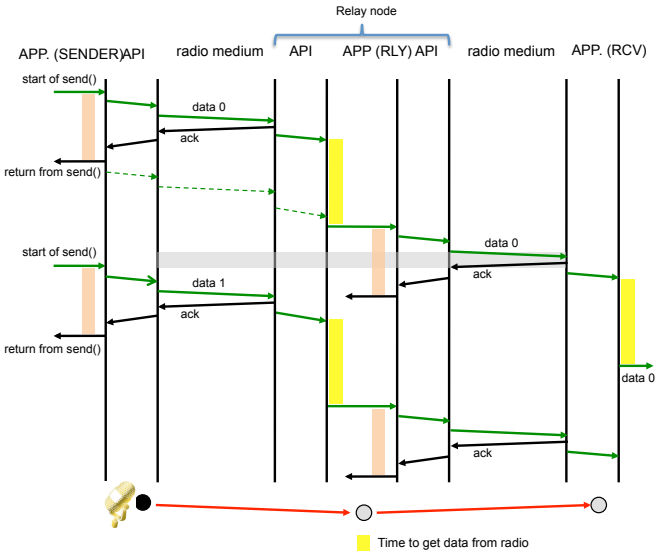

Fig. 8. Multi-hop transmission

Actually, it is well-known that multi-hop transmissions generate higher level of packet losses because of interference and contention on the radio channel (uplink, from the source; and downlink, to the sink). In this case, when the minimum time between 2 packet generation is too small, there are contention issues between receiving from the source and relaying to the sink. This is depicted in figure 8 by the gray block. However, as we found that the minimum time between 2 packet generation is much greater than the radio transmission time (which is about 5ms for a 100-byte packet), multi-hop transmissions in this case will most likely rather suffer from

high processing latencies than from contention problem. On the figure, we can see that the relay node, upon reception of the packet from the source node, needs an additional delay to get data from the radio (yellow block), before being able to send it to the next hop. This delay is far from being negligible as in the best case it is similar to the time to write to the radio.

We also found that the time to read the received data, noted t_{read}, 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 9 plots t_{read} (blue curve) for the WaspMote.

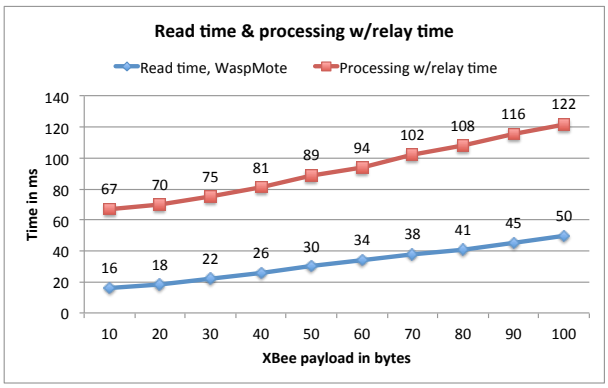

Fig. 9. Time to read data and total processing w/relay time

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 9 shows 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 almost divided by 2.

In total, when adding additional data handling overheads, a relay node needs about 122ms 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 9. Figure 10 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 audio encoding scheme.

In case the next packet from the source node arrives before the previous packet has been read, the reception buffer may overflow quite quickly. This is depicted by the dashed green arrow from the source to the first relay node. On more elaborated OS and processors, it is possible to have a multi-threaded behavior to processed the received packet earlier but in this case contention on serial or data buses need to be taken into account. In all cases, we clearly see that in the best case the next packet will not be sent before the return of the last send. In the next section, we will show real experimental results of sending acoustic data from a source sensor node to a sink with relay nodes in between.

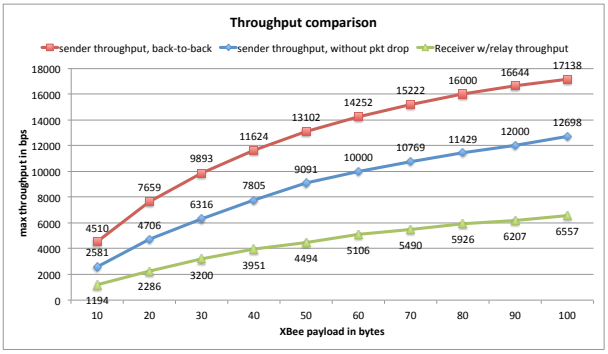

Fig. 10. Throughput comparison

IV. Streaming acoustic data

A. 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 11.

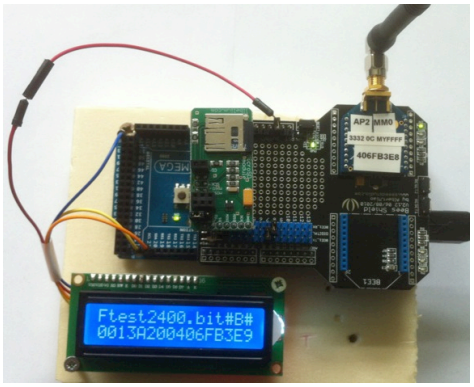

Fig. 11. Arduino Mega2560 for acoustic data stored in the SD card

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,...). 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 12 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 13.

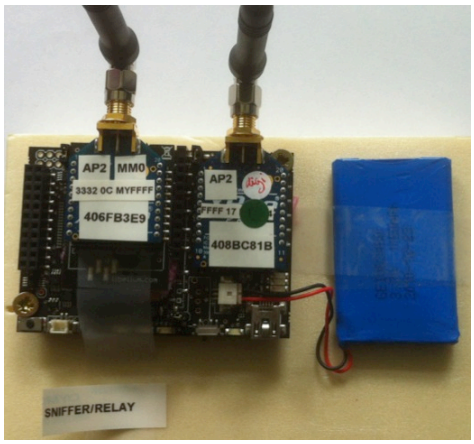

Fig. 12. A WaspMote relay node

Fig. 13. Test of acoustic data streaming: topology

Fig. 14. Test of acoustic data streaming: placement of nodes

We placed our nodes on the street lamps indicated in figure 13, 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 14(left). We strapped our nodes as depicted by figure 14(right).

B. 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.

C. Audio codecs

Given the low receiver throughput shown in Figure 10, 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` [13] 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: (1) `cat test2400.bit | c2dec 2400 - -`

`| play -r 8000 -s -2 -`, assuming that the encoding rate is 2400bps.

We use these tools with our `XBeeReceive` tool in the following way: (2) `XBeeReceive -B -stdout test2400.bit | bfr -b1k -m10% - | c2dec 2400 - - | play -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` indicate 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 smartphone 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 an 8-bit sample raw file at 8000Hz. Then with `c2enc` we produced codec2 files at 1400, 2400 and 3200bps. The file sizes are 2338, 4014 and 5352 bytes respectively. All these files can be downloaded in .wav format for immediate playout in most players at [14]. These files are placed on the SD card of the Arduino sender node.

D. Results

We performed multi-hop transmissions with 1-relay node and 2-relay node configuration, see figure 13. 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 15 shows the maximum inter-packet time for various packet size and encoding rate. We also plot the total processing time depicted previously in Figure 9 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.

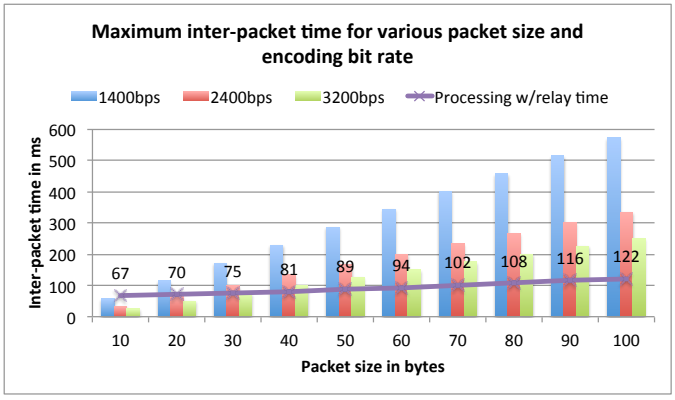

Fig. 15. Maximum inter-packet time at various packet size and encoding rate

However, Figure 15 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.

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

Table I 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 [14]. For the 2-relay node scenario, the results are summarized in Table II.

V. Conclusions

Multi-hop multimedia streaming on low-resource devices (WSN, IoT) is a promising techniques for surveillance applications. In this paper, we presented experimentations on the SmartSantander test-bed for acoustic data streaming in the EAR-IT project. Prior to the streaming experimentation itself, we first qualify the SmartSantander hardware and highlight the main sources of delays assuming no flow control nor congestion control. The purpose of the study is to determine the best case performance level that could be expected when considering IEEE 802.15.4 multi-hop connectivity. We showed that there are incompressible delays due to hardware constraints and software API that limit the time between 2 successive packet send.

The experiment we performed with the audio codec2 encoding scheme demonstrated that streaming acoustic data is feasible on Smart Cities infrastructures with low-resource devices. The codec2 encoding scheme is a very low bit rate audio codec therefore leaving room for higher bit-rates if higher quality is required. However, the WaspMote hardware/software capabilities are quite limited and 6kbps is probably the maximum encoding bit-rate that is compatible with low latency and streaming feature on this type of platform. If one wants to go beyond this performance limit, relaying must be done at the lowest level of the communication API in order to reduce the data handling time at relay nodes. We plan to investigate the usage of DigiMesh radio module that have a MAC-level AODV-like routing feature in future works.

We did not use packet size larger than 60 bytes to reduce the impact of packet losses. Obviously, it is possible to reduce the transmission latency by increasing the packet size. In this case, the inter-packet time may be increased to limit the number of packet drops. Another solution, which is beyond the scope of this paper but probably very promising, is to perform prediction on incoming acoustic data in order to fill missing data with more appropriate values.

Finally, we chose to not address the overhead for sampling and encoding the acoustic data. In practice, in addition to the communication latency, encoding the acoustic data may add a large processing delay depending on the complexity of the encoding scheme. Our rationale for not having addressed these issues is because there is a very large range of hardware possibilities and some specific hardware can incredibly speedup the encoding scheme. Moreover it is quite possible that dedicated audio boards will be able to perform most of processing tasks independently from the main micro-controller. Collaboration is on-going for building dedicated audio boards.

Acknowledgment

This work is partially supported by the EU FP7 EAR-IT project. The authors would like to thank the SmartSantander research team lead by Pr. Luis Muñoz of University of Cantabria for all the information on the SmartSantander test-bed they provided. Special thanks to PhD students M. Diop and E. Muhammad for their invaluable help during the test campaign.

References

[1] SmartSantander, "http://www.smartsantander.eu," accessed 4/12/2013.
[2] EAR-IT, "http://ear-it.eu/," accessed 4/12/2013.
[3] M. Rahimi, R. Baer, O. I. Iroezi, J. C. Garcia, J. Warrior, D. Estrin, and M. Srivastava, "Cyclops: In situ image sensing and interpretation in wireless sensor networks," in *ACM SenSys*, 2005.
[4] I. F. Akyildiz, T. Melodia, and K. R. Chowdhury, "A survey on wireless multimedia sensor networks," *Computer Networks*, vol. 51, pp. 921–960, 2007.
[5] M. R. Misra and G. Xue, "A survey of multimedia streaming in wireless sensor networks," *IEEE Communications Surveys & Tutorials*, 2008.
[6] S. Soro and W. Heinzelman, "A survey of visual sensor networks," *Advances in Multimedia*, 2009.
[7] Libelium, "http://www.libelium.com/," accessed 4/12/2013.
[8] Digi, "http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/," accessed 4/12/2013.
[9] IEEE, "Ieee std 802.15.4-2006." 2006.
[10] Digi, "Xbee/xbee-pro rf modules product manual (90000982_g), digi international inc. august 1, 2012." 2012.
[11] ——, "Xbee/xbee-pro digimesh rf modules product manual (90000991_e), digi international inc. january 6, 2012," 2012.
[12] A. Rapp, "http://code.google.com/p/xbee-arduino/," accessed 4/12/2013.
[13] D. Rowe, "http://codec2.org," accessed 2/10/2013.
[14] C. Pham, "http://web.univ-pau.fr/˜cpham/SmartSantanderSample/," accessed 5/13/2013.