Communication performance of low-resource sensor motes for data-intensive applications

Congduc Pham
University of Pau, LIUPPA Laboratory
Email: Congduc.Pham@univ-pau.fr

***Abstract*—We study the communication performance of low-resource sensor motes that are commonly found in smart cities test-beds or used by the research community. We focus here on 802.15.4 radio and we present a performance study of sending and receiving capabilities of Libelium WaspMote, Arduino-based motes, Telosb-based motes and MicaZ motes when a large number of packets needs to be streamed from sources to sink node. We provide measures for the minimum time spent in send procedure, minimum time needed to read data into application memory space and maximum sender/receiver throughput. We highlight the main sources of delays assuming no flow control nor congestion control to determine the best case performance level. Our contribution is therefore in determining the maximum realistic level of performance for commonly found mote platforms in order to predict end-to-end performances for data-intensive applications such as multi-hop multimedia streaming for surveillance.**

I. Introduction

This article considers data-intensive applications with Wireless Sensor Networks (WSN) such as those involving multimedia nodes where sensor nodes can provide images and/or acoustic data to a sink. The domain of multimedia WSN is becoming more and more important as it can be observed by the increasing number of scientific contributions in the last years [1], [2], [3], [4] to name a few. As images and acoustic data usually require a much larger amount of packets to be transferred, and most likely with some time constraints, they perfectly fit into the so-called data-intensive sensor network category. In this paper, we are focusing on the communication performance of low-resource sensor motes that are commonly found in smart cities test-beds or used by the research community. We therefore present a performance study of sending and receiving capabilities of Libelium WaspMote, Arduino-based motes, Telosb-based motes and MicaZ motes when a large number of packets needs to be streamed from sources to sink.

There have been previous works on image/multimedia sensors but few of them really specifically study the communication capabilities or limitations of realistic hardware and software API for sending/receiving packets. Recent work presented in [3], [4] are probably the closest work to ours with real experimentations on sensor motes such as iMote2 and TelosB sensors. However, their focuses were more on global performances than on a detailed study of the hardware and API limitations. There have also been some works on performance evaluation of radio technologies for multimedia traffic such as the works reported in [5], [6] but, once again, they did not investigate the communication overheads in a synthetic manner. Some studies on 802.15.4 radio performances are also available from radio module manufacturers [7], [8] but these studies remain at a very high-level and do not take into account software overheads. In this paper, we present experimentations with real sensor boards and real radio modules to transmit a large amount of packets in a streamed fashion. We use traffic generators and programming API timing to highlight the main sources of delays assuming no flow control nor congestion control to determine the best case performance level. One usage for this study could be to use these real performance measures in simulation models to provide more realistic performances for large-scale multimedia sensor networks deployment for instance. Although it is not possible to address the large variety of existing sensor boards (see [9] for a survey on image sensor platforms) we however provide measures for UART-based and SPI-based sensors that could be adapted to other type of sensors to determine the performance level that can be expected. Also, we address motes based on 2 well-known and well-used programming environments: Arduino-like IDE (cc.arduino.org) and TinyOS (www.tinyos.org). We will not address the capture process, i.e. how to capture image or acoustic data, nor the compression overheads, as these 2 components can be optimized in many various ways such as dedicated daughter boards with hardware optimizations. This paper focuses on the communication performances of motes using the IEEE 802.15.4 technology.

The paper is then organized as follows. Section II presents real measures on sensor hardware and radio modules of what could typically be expected with 802.15.4 communication stacks at the application level for WaspMote, Arduino, TelosB and MicaZ motes. API send time and minimum time between 2 packet generation, as well as maximum sender and receiver throughput will be presented. We will also show the performance of relaying mode for multi-hops communication scenarios in Section III. Conclusions will be given in Section IV.

II. Communication performances on real sensors

We consider Libelium WaspMote [10] that are used in a number of Smart Cities and environmental monitoring projects [11], [12], Arduino MEGA 2560 [13] (Libelium WaspMote IDE is largely based on Arduino), TelosB motes from AdvanticSys [14] and Crossbow MicaZ motes. The last 2 platforms being well-known to the WSN research community.

A. Hardware

1) Libelium WaspMote & Arduino: Libelium WaspMote use an IEEE 802.15.4 compliant radio module called XBee manufactured by Digi [15] which offers a maximum application level payload of 100 bytes. By default, the XBee module uses a $macMinBE$ value of 0 therefore the effective maximum throughput roughly corresponds to the 102-bytes payload case presented in the previous section. The WaspMote has an Atmega1281 running at 8MHz. The XBee module and the micro controller communicate through an UART, and for the WaspMote the default data rate is set to 38400bps by the Libelium API. In a first step we investigate the off-the-shelves performance of the WaspMote. However we use a modified version of the "light" Libelium API provided by Libelium. Compared to the "full" Libelium API that additionally handles long packets with fragmentation/reassembly support, the light Libelium API is much faster and our modified version further reduces the complexity of the send operation. As WaspMote is very similar to the well-known Arduino boards the results presented in this section also apply to the Arduino MEGA 2560 board which features an ATmega2560 running at 16MHz. This Arduino board is one of the fastest Arduino boards in the market and is quite representative of UART-based sensor boards. On the Arduino, we also use a very lightweight communication library[16] and our modified API for the WaspMote achieves the same level of performance than the Arduino's API. Both motes use the Arduino programming environment that offers a C++-like programming language. Various libraries are available for the Arduino and they can be used on the WaspMote with very little changes.

2) AdvanticSys TelosB-based mote and Crossbow MicaZ mote: AdvanticSys motes (we experimented on CM5000 and CM3000) are based on TelosB/TmoteSky motes and therefore are referred to as TelosB. Both TelosB and MicaZ motes are built around an TI MSP430 microcontroller at 8MHz with an embedded ChipCon CC2420 802.15.4 compatible radio module. The original TelosB description and datasheet can be found in [17]. Documentation on the AdvanticSys motes can be found in [14]. MicaZ description can be found in [18]. The CC2420 radio specification and documentation are described in [19]. The important difference compared to the previous Libelium WaspMote or Arduino is that the radio module is connected to the microcontroller through an SPI bus instead of a serial UART component. This normally would allow for much faster data transfer rates.

TelosB and MicaZ are usually programmed under the TinyOS system [20], although other OS can be used. In this paper, we only study the communication performance under TinyOS. The last version of TinyOS is 2.1.2 and our tests use this version. The default TinyOS configuration use a MAC protocol that is compatible with the 802.15.4 MAC (Low Power Listening features are disabled). The default TinyOS configuration also uses ActiveMessage (AM) paradigm to communicate and interoperable frames (IFRAME) are used to allow interoperability with non-TinyOS network. As we are using heterogeneous platforms we will not use the default MAC layer of TinyOS but rather the TKN154 IEEE 802.15.4 compliant API. However, we verified the performances of TKN154 against the TinyOS default MAC and found them similar (TelosB) or greater (MicaZ). Therefore, in all cases, the results that will be presented are the best case results that we could obtained with our tests.

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

B. Sending performances

One of the main objectives 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, the bottleneck is often the interconnection between the microcontroller 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 microcontroller 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 neglectible 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 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 image transmission that requires sending the image in 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 backoff exponent BE set to 0 (default is 3), we typically need a minimum of $5.44ms + 4.34ms = 9.78ms$ to send a single

100-byte packet if there is no error. Now, in more advanced hardware architecture the radio module can be connected to the microcontroller through a high-speed bus (SPI for instance) which allows for much higher data transfer rates, in which case a unicast transmission of a single 100-byte packet with the same MAC parameter would take $5.44ms + \epsilon$. However, as we will show later on, not only the sending side should be taken into account and sending fast is usually not reliable.

To highlight the importance of the time needed to write to the radio on some hardware, we measure on real sensor hardware and communication API the time spent in a generic send() function (most communication APIs have a function to send a packet), noted t_{send}, and the minimum time between 2 packet generation, noted t_{pkt}. t_{pkt} typically takes 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 use a traffic generator that sends packet 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. When possible, we also add non-intrusive accurate timing of the programming API.

1) Libelium WaspMote & Arduino, sending side: Figure 1 shows the time in send() breakout for the WaspMote (data transfer rate is 38400) where we can especially see the time required to write to the radio. Each value for a given payload size is an average value over 20 measures. The sum of all the timing represents what we called t_{send}. We can see that the bottleneck here is the time to write to the radio as the data transfer rate is only 38400bps.

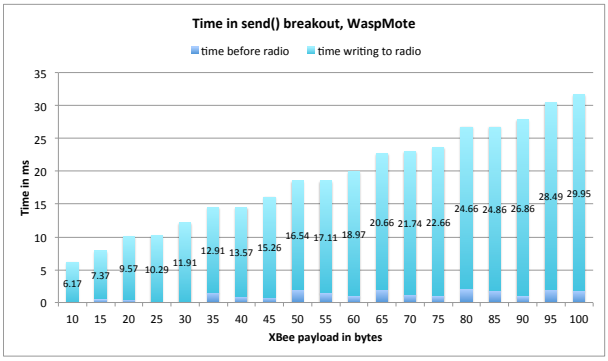

Fig. 1. Time in send() breakout, WaspMote

Figure 2 shows both t_{send} and t_{pkt} for the Waspmote. The maximum realistic throughput could be obtained from t_{pkt}. On the Arduino, the communication API has similar performances: for a 100-byte packet t_{send} is about 29.8ms and t_{pkt} is about 33.45ms.

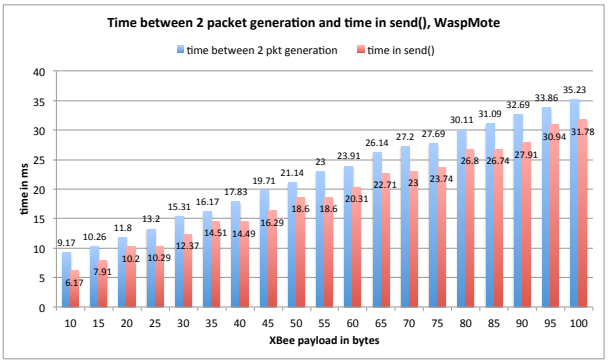

Fig. 2. Time between 2 packet generation and time in send(), WaspMote

We then increased the UART data transfer rate that is set by default to 38400bps. However, increasing the baud rate cannot be done without taking into account some timing constraints that may make the serial communication unreliable [21]. The WaspMote microcontroller runs at 8MHz while the XBee module has an 16MHz clock and requires that the frequency is 16 times the baud rate. It means that for a baud rate of 38400, the actual operating frequency need to be $16 \times 38000 = 614400$Hz. For reliable communication, the WaspMote clock should also produce a frequency close to 614000Hz. Since it runs at 8MHz, the dividing factor is $8000000/614000 = 13.020833$. Using the nearest integer dividing factor of 13, the actual baud rate is $8000000/16/13 = 38461,54$ which is 1.0016026 times greater than the target baud rate. The error is about 0.1602% which allows for reliable communication between the microcontroller and the XBee module. Actually, 38400, which is the value chosen by the Libelium API is the fastest standard baud rate that provides acceptable errors between the target baud rate and the actual baud rate. Using 57600 or 115200 baud rates would generate too many errors, making the communication very unreliable and therefore not functioning at all. Even on the XBee, 57600 and 115200 baud rates can not accurately be achieved with the 16MHz clock. Using these constraints, the perfect dividing factors for the WaspMote are 10, 5, 4, 2 and 1 which correspond to 50000, 100000, 125000, 250000 and 500000 baud rates respectively. As we showed that the maximum 802.15.4 effective throughput is roughly 166666bps in broadcast mode when there are no errors, there is no point to consider 500000 baud rate that would additionally overflow the transmission buffer. On the Arduino, as the clock runs at 16MHz, there is no problem in getting these baud rates with a dividing factor of 20, 10, 8, 4 and 2 respectively.

With a serial transmission of 8 data bits, 1 start bit and 1 stop bit that gives 10 bits per byte, Figure 3 shows the estimated time to write to radio if baud rates up to 250000 were applied. If we assume that reducing the time to write to radio does not change the other overheads, we can estimate the new t^B_{send} for a baud rate B higher than 38400 as follows:

$$t^B_{send} = t^{38400}_{send} - timeToWriteToRadio^{38400} + timeToWriteToRadio^B$$

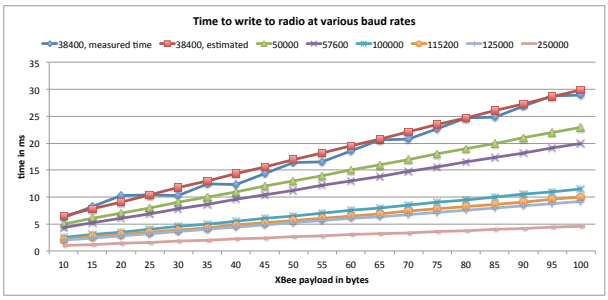

Fig. 3. Time to write to radio at various baud rates

Then, assuming that the overheads between 2 packet generation are also independent from the time to write to radio, we can estimate the new t_{pkt}^{B} for a baud rate B higher than 38400 as follows:

$$t_{pkt}^{B} = \quad t_{pkt}^{38400} - t_{send}^{38400} + t_{send}^{B}$$

To verify our assumption we set the baud rate of the XBee module to 125000 and 250000 and ran again the traffic generator on the WaspMote after having changed the default data transfer rate of the Libelium communication API from 38400 to 125000 and 250000. Figure 4 shows the estimated and measured time between 2 packet generation for data transfer rates of 125000 and 250000 with the WaspMote.

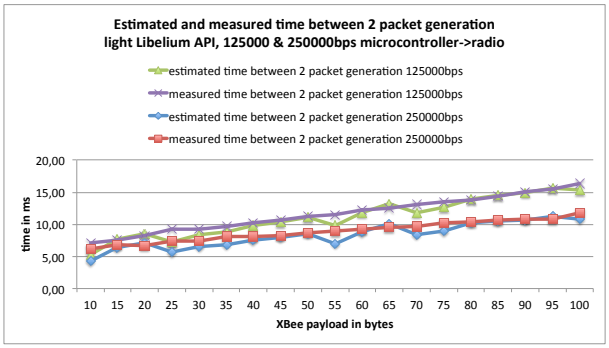

Fig. 4. Estimated and measured for t_{pkt}^{125000} and t_{pkt}^{250000}

We can see that the estimated and the measured curves are very close each other, thus validating our estimation method of the time to write to radio and the constant overheads of the communication API. In summary, when using a 100-byte payload, we can have $t_{pkt}^{125000} \approx 16ms$ and $t_{pkt}^{250000} \approx 12ms$ for both the WaspMote and the Arduino. These results can be seen as the minimum time between 2 packet generation that could be achieved for sensor nodes with a similar architecture using UART lines for communications between microcontroller and radio module. Figure 5 shows maximum sending throughput that have been measured at various baud rates. However, we observed many transmission errors between the microcontroller and the radio module at 250000 baud rate that make the whole transmission very unreliable. Therefore, in practice, WaspMote and the Arduino can not really be used with a serial data transfer rate higher than 125000bps.

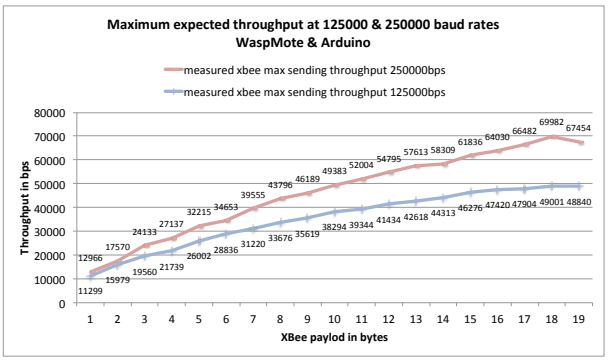

Fig. 5. Maximum sending throughput at various baud rates

2) TelosB and MicaZ, sending side: Again, we use a traffic generator to send packet back-to-back at the sender side. We then measure the time in send() (t_{send}) and the minimum the time between 2 packet generation (t_{pkt}) under TinyOS. The so-called "fitted" curve is a linear approximation and the label values on the curves correspond to the fitted curve. Figure 6 and Figure 7 show the measures for the TelosB and MicaZ respectively.

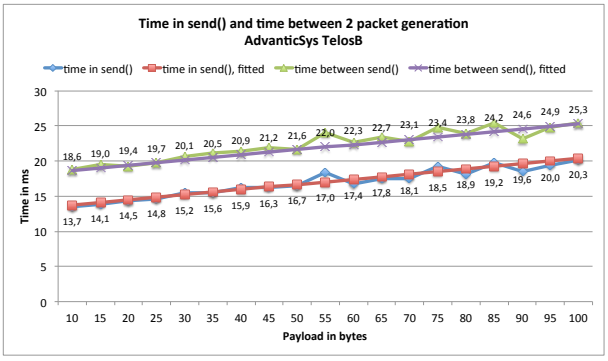

Fig. 6. Time in send() and time between 2 packet generation, TelosB

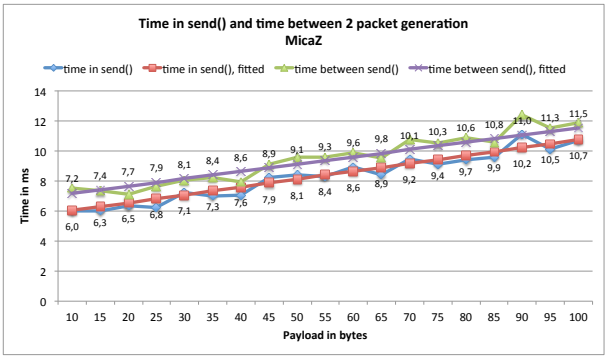

Fig. 7. Time in send() and time between 2 packet generation, MicaZ

Figure 8 shows the corresponding maximum sending throughput and the maximum realistic throughput at the sender side. Once again, we differenciated the maximum sending throughput case when only the time in send() is considered

from the case the time between 2 packet generation is used, which is a more realistic scenario.

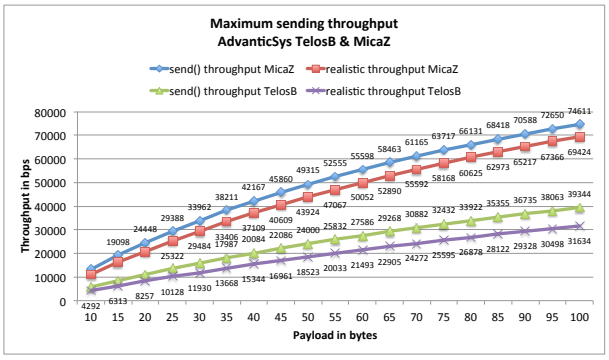

Fig. 8. Maximum sending throughput for TelosB and MicaZ

We can see that the MicaZ platform has the best performance level, both in terms of time in send() and time between 2 packet generation: it can realistically send one 100-byte packet in about 11ms. The resulting maximum realistic sending throughput is close to 70kbps. Compared to the previous WaspMote and Arduino platforms pushed at 125000baud/s, MicaZ motes are still much more performant while TelosB has the lowest maximum realistic sending throughput at about 32kbps.

C. Receiver performances

In the next set of experiments, we use the traffic generator to send 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 rather send packets as fast as possible, measure the number of received bytes at the receiver and compute the receiver throughput each time the payload size is increased. Prior to this test we also wanted to measure the time needed by the mote to read the received data into user memory or application level, noted t_{read}. Normally, the receiver throughput is linked to t_{read} following a simple producer-consumer model: if packets arrive faster than they can be made available to the application they are dropped by the limited receiver buffer.

1) Libelium WaspMote & Arduino, receiver side: Figure 9 shows for both the WaspMote and the Arduino the time needed to read a packet into the application memory space. Actually, we found that t_{read} is quite independent from the communication baud rate between the microcontroller and the radio module. In all our experimentations, for baud rates of 38400, 125000 and 250000, t_{read} remains constant and depends only on the data size. The reason why t_{read} only depends on the data size, at least at the application level, is as follows: most of communication APIs use 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 depending mainly on the frequency used to operate the sensor board and the data bus speed. As we can see in figure 9, t_{read} on the WaspMote is about 50ms and t_{read} on the Arduino is about 35ms, for a 100-byte packet. Figure 10 shows the maximum receiver throughput for both the WaspMote and Arduino motes.

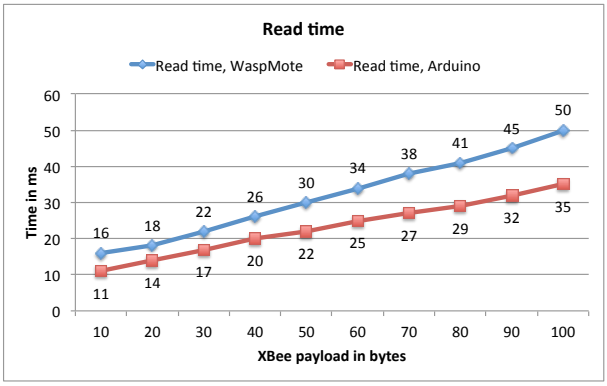

Fig. 9. Minimum read time on WaspMote and Arduino

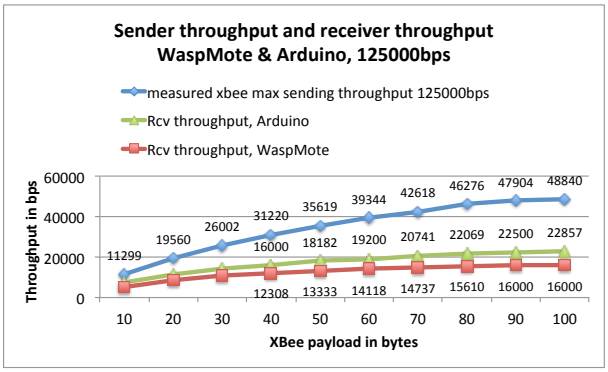

Fig. 10. Maximum sending throughput and maximum receiver throughput on WaspMote and Arduino

2) TelosB and MicaZ, receiver side: Following the same methodology applied for WaspMote and Arduino, Figure 11 shows for both the TelosB and the MicaZ the time needed to read a packet into the application memory space. Compared to values for WaspMote and Arduino shown in figure 9, we can see that the more advanced architecture using in part an SPI bus for the communications between the microcontroller and the radio module offers very small data read time. We therefore plot in Figure 12 the maximum receiver throughput on TelosB and MicaZ based on these values. Then Figure 13 shows the maximum measured receiver throughput for these platforms using the MicaZ traffic generator.

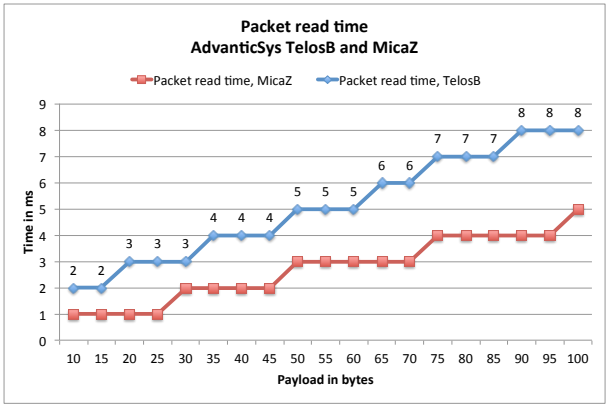

Fig. 11. Minimum read time on TelosB and MicaZ

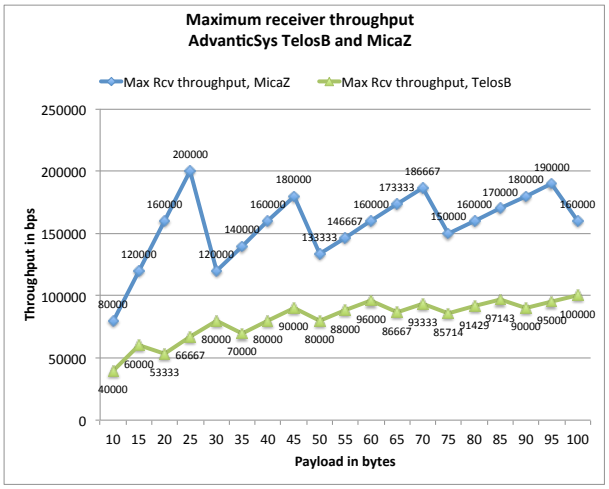

Fig. 12. Maximum receiver throughput on TelosB and MicaZ

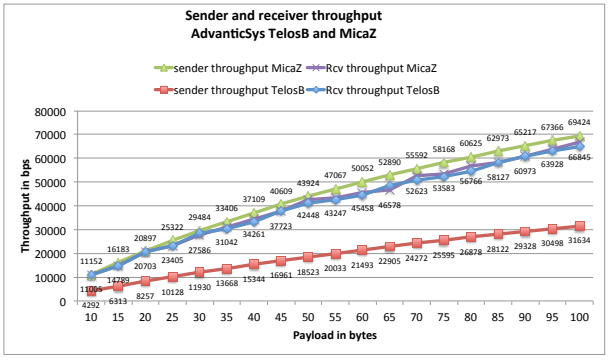

Fig. 13. Maximum sending throughput and maximum receiver throughput on TelosB and MicaZ

Our first experiments on the TelosB did use a TelosB sender which sending throughput was found around 31.6kbps for a 100-byte packet. Then, we found the receiver throughput very close to the sender throughput which showed that the limiting factor was at the sender side. We then use a MicaZ sender which is the fastest sending mote available to us and ran the experiments again. The measured receiver throughput on the TelosB in figure 13 is for this last case where we can see that TelosB and MicaZ have similar receiving performances (the labels for the receiver throughput are for the MicaZ motes). Figure 13 also shows the maximum realistic sending throughput on the TelosB and MicaZ motes previously shown in Figure 8 for comparison purposes.

As we do not have the possibility to send packets faster than 1 packet every 11ms (maximum performance of the MicaZ platform), even with a 802.15.4 bridge plugged into a desktop workstation, at this point of our experimentation, we can not state for sure whether the receiving performance is much smaller than the theoretical maximum receiver throughput as depicted in Figure 12, or could be close to theoretical values based on t_{read}. However, we can see in Figure 13 that the receiver throughput starts to diverge from the sender throughput for large packet sizes. Therefore our opinion is that the maximum realistic receiver throughput will be well below the theoretical maximum receiver throughput because of various overheads of the reception operations: hardware interrupts, tasks handling, memory copies, bus transfers,...

III. Multi-hop issues

In data-intensive WSN such as multimedia motes, a large amount of 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.

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. However, as we found that the minimum time between 2 packet generation is much greater than the radio transmission time (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. Upon reception of the packet from the source node, a relay node needs an additional delay to get data from the radio before being able to send it to the next hop. This delay was referred to as the time to read data, t_{read}, and was found to be quite large in simple mote architecture such as the WaspMote and Arduino (see figure 9).

In total, when adding additional data handling overheads, we measured that a relay node based on a WaspMote needs about 108ms to process the incoming packet and to relay it to the next hop, once again for a 100-byte packet. The Arduino can do it in about 94ms, see Figure 14, red (WaspMote) and purple (Arduino) curves. In case the next packet from the source node arrives before the previous packet has been read, the reception buffer may overflow quite quickly.

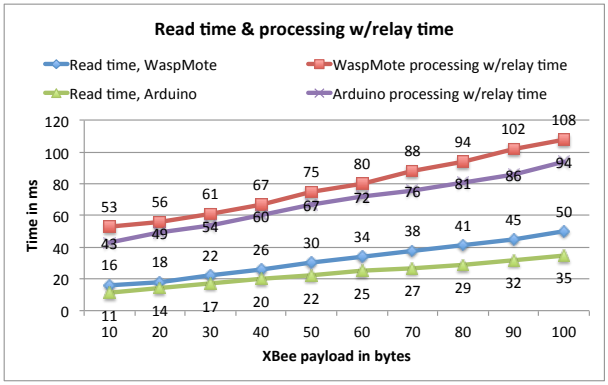

Fig. 14. Measured time to read and relay data on WaspMote and Arduino

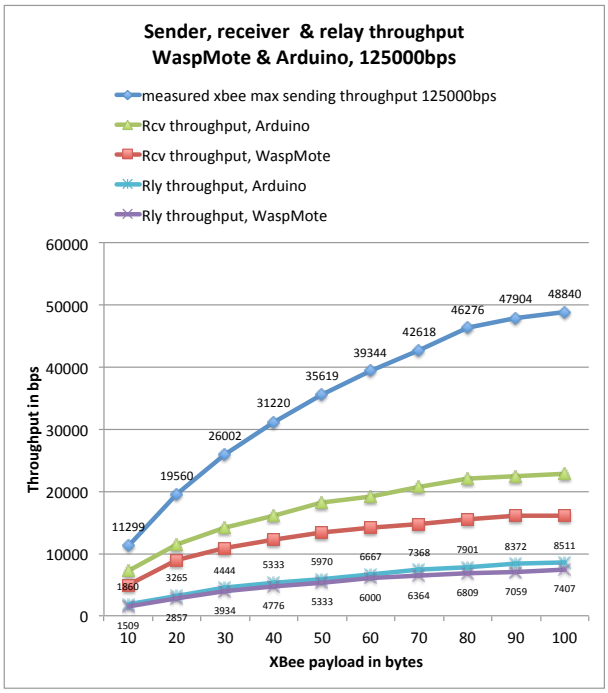

Fig. 15. Maximum sending throughput, receiver throughput and relay throughput on WaspMote and Arduino

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. We can see that multi-hop transmission on this type of platform adds a considerable overhead that put strong constraints on data-intensive sending applications. Figure 15 shows the measured relay throughput for both WaspMote and Arduino. We also plot the sending throughput (WaspMote and Arduino have same level of performance) and the receiver throughput previously shown in Figure 10 for comparison purposes.

On the TelosB and the MicaZ we also measured the time needed to receive a packet from the time the frame delimiter was received by the radio and to the time when relay has been performed (when the packet is notified to be sent). For each packet size, we receive and relay 20 packets, then the packet size increases by 5 bytes, starting with an initial size of 10 bytes. The relay time for each packet is then shown in Figure 16.

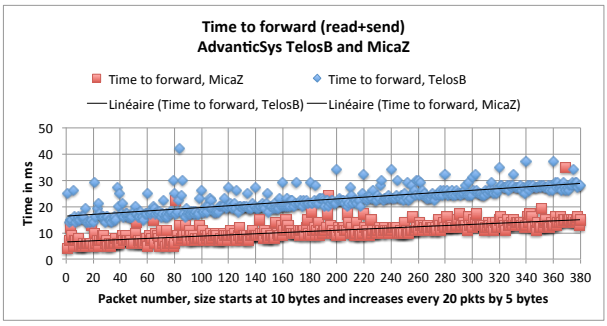

Fig. 16. Measured time to relay data on TelosB and MicaZ

Figure 17 compares the theoretical relay time computed by adding t_{read} and t_{send} to the measured relay time obtained by averaging the 20 values for each packet size, for both the TelosB and the MicaZ. We also show t_{read} for comparison purposes.

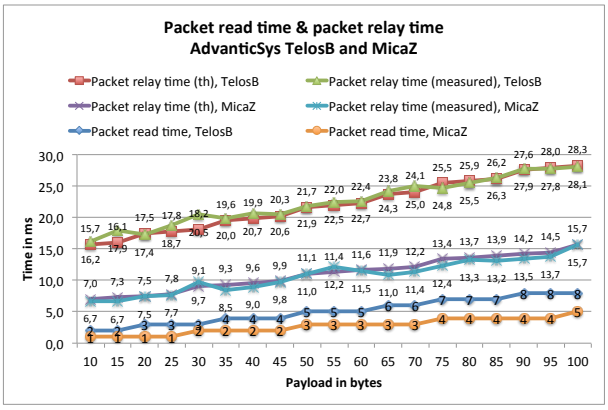

Fig. 17. Measured time to read and relay data on TelosB and MicaZ

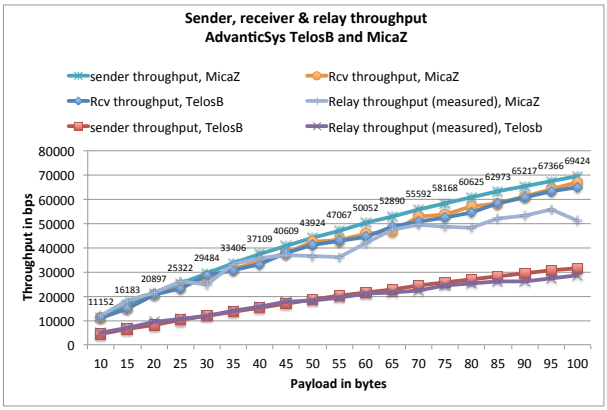

Fig. 18. Maximum sending throughput, receiver throughput and relay throughput on TelosB and MicaZ

Finally, Figure 18 shows the various throughput for the TelosB and MicaZ: sending, receiver and relay throughput. For the TelosB, as expected, the relay throughput is limited by the sending throughput and we can see that both curves are very close. On the MicaZ, the relay throughput stay close to the receiver throughput for packet size smaller than 45 bytes. In all our experiments, the MicaZ motes appeared to be the most performant mote for sending, receiving, and thus relaying, operations.

IV. Conclusions

We presented experimentations with various sensor boards, radio modules and software API to highlight the main sources of delays assuming no flow control nor congestion control. The motivation of our work is to determine the best case performance level that could be expected when considering IEEE 802.15.4 multi-hop connectivity for data-intensive applications such as those involving multimedia sensor nodes for image and acoustic surveillance systems. We showed that there are incompressible delays due to hardware constraints and software API that limit (i) the time between 2 successive packet send and (ii) the receiver throughput. We found that typical sending throughput is about 50kbps for the WaspMote and Arduino motes, and about 32kbps and 70kbps for TelosB and Micaz motes respectively. Even if WaspMote and Arduino motes can send data faster than a TelosB, both are much more limited for data reception. TelosB and MicaZ have reception throughput of about 67kbps while WaspMote and Arduino merely reach 16kbps and 23kbps respectively. In all our experiments, the MicaZ motes appeared to be the most performant mote for sending, receiving, and thus relaying, operations.

Our contributions can also be used for building more realistic simulation models by taking into account the real communication overheads and not only the radio transmission time. Also, by identifying the limitations and the bottleneck, more suitable control mechanisms could be studied and proposed. For instance, while flow control and congestion control are of prime importance we believe that traditional approaches based on buffer management or rate control are not efficient because they will add to much latency that is not compatible with data-intensive surveillance applications. Our further works, based on the results presented in this paper, will rather propose to have scheduling mechanisms to explicitly prevent nodes from sending flows of packets at the same time in the same area.

Acknowledgment

This work was partially supported by the Aquitaine-Aragon OMNIDATA project.

References

[1] V. Berisha, H. Kwon, and A. Spanias, "Real-time acoustic monitoring using wireless sensor motes," in *International Symposium on Circuits and Systems (ISCAS 2006)*, 2006.

[2] R. Mangharam, A. Rowe, R. Rajkumar, and R. Suzuki, "Voice over sensor networks," in *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, 2006.

[3] S. Paniga, L. Borsani, A. Redondi, M. Tagliasacchi, and M. Cesana, "Experimental evaluation of a video streaming system for wireless multimedia sensor networks," in *Proceedings of the 10th IEEE/IFIP Med-Hoc-Net*, 2011.

[4] P. Chen et al., "A low-bandwidth camera sensor platform with applications in smart camera networks," *ACM Trans. Sen. Netw.*, vol. 9, no. 2, pp. 21:1–21:23, Apr. 2013.

[5] T. F. Abdelzaher, S. Prabh, and R. Kiran, "On real-time capacity limits of multihop wireless sensor networks," in *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004)*, 2004.

[6] D. Brunelli, M. Maggiorotti, L. Benini, and F. L. Bellifemine, "Analysis of audio streaming capability of zigbee networks," in *Proceedings of EWSN 2008*, 2008.

[7] Jennic, "Application note: Jn-an-1035. calculating 802.15.4 data rates. http://www.jennic.com/files/support_files/jn-an-1035 calculating 802-15-4 data rates-1v0.pdf," accessed 4/12/2013.

[8] Digi, "Sending data through an 802.15.4 network latency timing. http://www.digi.com/support/kbase/kbaseresultdetl?id=3065," accessed 8/2/2013.

[9] B. Tavli, K. Bicakci, R. Zilan, and J. M. Barcelo-Ordinas, "A survey of visual sensor network platforms," *Multimedia Tools Appl.*, vol. 60, no. 3, pp. 689–726, Oct. 2012.

[10] Libelium, "www.libelium.com/," accessed 4/12/2013.

[11] ——, "www.libelium.com/top_50_iot_sensor_applications_ranking/," accessed 4/12/2013.

[12] SmartSantander, "http://www.smartsantander.eu," accessed 4/12/2013.

[13] Arduino, "http://arduino.cc/en/main/arduinoboardmega2560," accessed 4/12/2013.

[14] AdvanticSys, "http://www.advanticsys.com/," accessed 4/12/2013.

[15] Digi, "http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/," accessed 4/12/2013.

[16] A. Rapp, "http://code.google.com/p/xbee-arduino/," accessed 4/12/2013.

[17] CrossBow, "http://bullseye.xbow.com:81/products/product_pdf_files/wireless_pdf/telosb_datasheet.pdf," accessed 4/12/2013.

[18] ——, "http://bullseye.xbow.com:81/products/product_pdf_files/ wireless_pdf/micaz_datasheet.pdf," accessed 4/12/2013.

[19] T. Instrument, "http://www.ti.com/lit/gpn/cc2420," accessed 4/12/2013.

[20] TinyOS, "http://www.tinyos.net/," accessed 4/12/2013.

[21] J. Foster, "Xbee cookbook issue 1.4 for series 1 (freescale) with 802.15.4 firmware, www.jsjf.demon.co.uk/xbee/xbee.pdf," April 26th, 2011. Accessed 4/12/2013.