Binary TDMA schedule by means of Egyptian fractions for real-time WSNs on TMotes

Binary TDMA scheduler by means of Egyptian Fractions for Real-Time WSNs on TMotes

Wim Torfs, Chris Blondia
University of Antwerp - IBBT,
Middelheimlaan 1, B-2020 Antwerp, Belgium
Email: {wim.torfs, chris.blondia}@ua.ac.be

***Abstract*—In Wireless Sensor Networks (WSNs), the user should not be bothered with configuring the network, depending on the kind of sensors that he/she is using. It should only be a matter of deploying the sensors and they should do the necessary effort to construct a network. Since most of these networks use low power devices, the energy consumption is vital. The most suited medium access method in order to waste as few energy as possible, is Time Division Multiple Access (TDMA). TDMA has however the reputation of being a rigid access method. In this paper, we present a TDMA access method for passive sensors, i.e. sensors are constant bitrate sources. The presented protocol does not only take into account the energy, but it also ensures that every sensor gets its data on time at the sink and this in a fair fashion. Even more, the latency of the transmission is deterministic. We could say that real-time communication is possible. The protocol is developed keeping in mind the practical limitations of actual hardware by limiting the memory usage and the communication overhead. The schedule that determines which sensor can send when, which can be encoded in a very small footprint, and needs to be sent only once. To prove our statement, we implemented our scheduling protocol on TMotes.**

I. Introduction

A Wireless Sensor Network (WSN) should be able to autonomously set up the connections between different devices, without any predefined topology. Devices in such network are considered to be small, low cost devices with limited resources, such as a low amount of working and program memory, low processing power and also a low battery capacity. Since we are dealing with devices with limited processing power, we make the assumption that no preprocessing of the sampled data is performed. As a consequence, every sensor can be considered as a constant bitrate source, of which the bitrate depends on the type of sampled data. This results in a heterogeneous WSN that needs to be able to cope with different rates in a flexible manner.

Algorithms running within a WSN try to utilize the resources as efficiently as possible. TDMA is one of the possibilities, not only to avoid collisions, but also to provide a sleep schedule. However, there are a few issues concerning TDMA. First of all, every sensor needs to follow the same schedule. Even if the schedule is adjusted to comply with changes in the network, all sensors need to adopt the new schedule at the same time, otherwise collisions occur. Secondly, a variable slot size or a variable number of slots in a frame are not desirable because of this strict schedule that needs to be followed by every sensor. Changing the slot size or number of slots every frame amounts to to passing a new schedule to all nodes every frame. Keeping in mind that the wireless medium is lossy, there is no guarantee that all sensors adopt the same schedule, since they might have missed its announcement.

The characteristics of TDMA and a WSN contradict in certain areas. A WSN needs to be flexible, while TDMA has a strict schedule of fixed size slots. We propose a TDMA scheduling algorithm that complies to the characteristics of both, this means that it is flexible, but also stable. By means of Egyptian Fractions and binary trees, we can compose a TDMA schedule that allows sensors to send in specified slots during certain frames, just enough to guarantee their required bandwidth. This schedule is periodic, which leads to a low protocol overhead, the TDMA schedule needs to be sent only once. Due to the specific construction of the schedule, additional bandwidth allocations do not require other sensors to adjust their schedule, rather the change is locally contained. A side effect of this schedule is that the latency is perfectly predictable, which means that the protocol is suited for real-time applications.

We claim that the protocol is perfectly suited for devices with limited resources. To prove this statement, we implement the protocol on TMotes[1], by means of TinyOS [2]. Afterwards, we compare the performance of this implementation with the results we got from simulating the scheduling protocol.

In the next section, we discuss some related research. Section III explains in detail how our protocol handles the scheduling of slots. Afterwards, we are discussing the implementation on the hardware in Section IV. Section V discusses the results of both simulations and hardware tests. And finally we give our conclusions.

II. Related work

A TDMA scheduling algorithm allows the sharing of a common resource, the wireless medium, among different sensors. Our algorithm allows every sensor to use the wireless medium for a time, relative to its requested bandwidth. The Weighted Fair Queuing (WFQ) [3][4][5], also provides the capability to share a common resource, and gives guarantees regarding the bandwidth usage. WFQ is an established protocol in the scheduling theory in order to achieve a fair schedule. It is also known as packet-by-packet generalized processor sharing (PGPS) [6]. Our algorithm uses a fractional representation of the requested bandwidth in order to determine the number of

resources. WFQ is comparable, it is a packet approximation of Generalized Processor Sharing (GPS) [7], where every session obtains access to the resource, but only for $1/N^{th}$ of the bandwidth, where N represents the available bandwidth divided by the requested bandwidth.
The most interesting related work that we found is [8], which deals with most regular sequences. The authors use a most regular binary sequence (MRBS) to express the requested rates that form a rational fraction of the total available bandwidth. The result is a cyclic and deterministic sequence, which determines for each session in which slot data should be sent in order to achieve the requested rate. The most regular sequences of different sessions can try to allocate the same slot, which needs to be solved by means of a conflict resolution algorithm. By means of a most regular code sequence (MRCS), the authors have found a way to share a single slot. The authors do not go into detail how a slot is shared, but it is assumed that it is possible. The MRCS creates exactly the same sequence as the MRBS, with the exception that the *1*'s and *0*'s are replaced by codes, which are a power of two, respectively higher and lower than the requested fraction of the capacity. The result is a rate that is the one time too fast, the other time too slow, which results in an average rate equal to the requested rate.

III. The Algorithm

The goal of our protocol is to create a periodic TDMA schedule at runtime. The schedule should allocate bandwidth to the sensors, so that it approximates the requested bandwidth. The periodicity of the schedule ensures that the scheduling information only needs to be given once. The goal is to allow a regular data flow from all sensors, both from high as low bandwidth sensors. Any subsequent change in the network and thus the schedule, should have no influence on already existing schedule rules. All of these goals need to be fulfilled while restricting the protocol overhead.
Our solution to meet these requirements is twofold. First of all, every request is represented by the quotient and remainder of the fraction of the requested bandwidth and the available bandwidth of a slot. The quotient, the number of full slots, can be seen as a fraction of the total number of slots. The remainder can be considered as the fraction of a single slot. To approximate the requested bandwidth as close as possible, both fractions are represented by means of an Egyptian Fraction (the sum of distinct unit fractions)[9][10]. The use of Egyptian Fractions ensures the periodicity and approximation. Secondly, in order to guarantee the robustness, every unit fraction is scheduled by means of a binary tree. The scheduling of the fractions that represent the number of slots, leads to the allocation of slots to the sensor. The scheduling of the fractions that represent the partial slot, leads to the allocation of a slot during a certain frame to the sensor.
A periodic schedule signifies that the slot allocations per frame change frame after frame according to a sequence, which restarts after a certain number of frames. The scheduling of full slots results in a static slot allocation per frame, hence periodicity is guaranteed. The scheduling of fractional slots results in a slot which is allocated to a sensor only for a certain number of frames. The slot can be used by other sensors during the remaining frames. A schedule that is repetitive can only be found if there is a common factor between the fractions that represent the requests of the different sensors. The fractions need to be approximated in order to find a periodic sequence between all requested fractions. As Fig. 1 shows, one of the

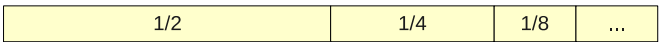

Fig. 1. Binary split up

possibilities is to use unit fractions with a denominator equal to a power of two. Unit fractions that have denominators equal to a power of two, can be easily combined without resulting into conflicts. Additional unit fractions can be fitted in the remaining space, without disturbing the already allocated fractions. However, the approximation towards such unit fractions has a large quantization error. Hence the approximation of the requested fraction by an Egyptian Fraction, where all unit fractions have a denominator equal to a power of two. The lowest possible denominator is bounded in order to prevent infinite or very long sequences. An interesting property of this action is that the requested bandwidth is split up into higher and lower frequency parts. A sensor gets access to the medium at least in periodic intervals equal to the highest frequency. The lowest frequency determines the period of the schedule.

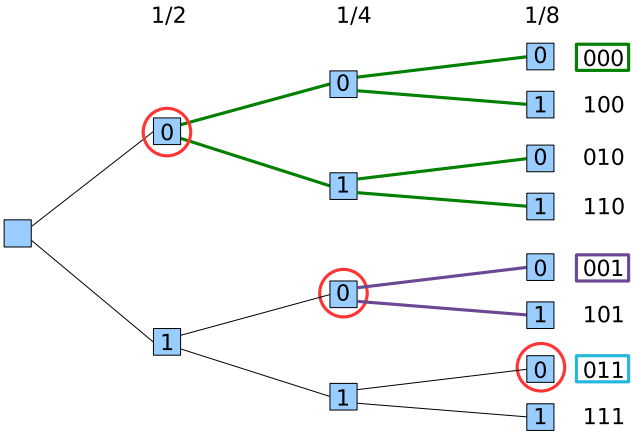

Fig. 2. Binary tree

By considering the requested bandwidth as a frequency, it is possible to allocate the number of required slots once during the period of that frequency. A bandwidth request of half the bandwidth, would then have a frequency of $\frac{1}{2}$. Applying the proposed method would allocate a slot every two slots for this request, resulting in an evenly distributed allocation. This prevents a sensor from seizing the wireless medium, while obstructing other sensors. Another advantage of using frequencies in a periodic system, is that there is only need for the frequency and the start position. By means of the frequency, the start position can be deducted by means of a binary tree, depicted in Fig. 2. We explain by means of

an example. The positions for each of the following fractions $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32}$ can be found by following the tree until its level has been reached. The most restrictive fraction, $\frac{1}{2}$, uses the resource half of the time. Thus, it can have 0 or 1 as start position. Both positions in the binary tree at the level of $\frac{1}{2}$ are still free. As a rule, first the path with the 0 is followed, hence position 0 is preserved for the fraction $\frac{1}{2}$. The next unit fraction that needs to be scheduled, is the fraction $\frac{1}{4}$. Fraction $\frac{1}{2}$ already occupies position 00000 and 00010, the only remaining positions at the level $\frac{1}{4}$ are 00001 and 00011. The rule to follow first the path with a 0 leads to the reservation of position 00001 for the fraction $\frac{1}{4}$. By repeating the procedure for all unit fractions, a start position can be found for each unit fractions, such that no fraction interferes with another. The resulting allocation of the positions can be found in Fig. 3.

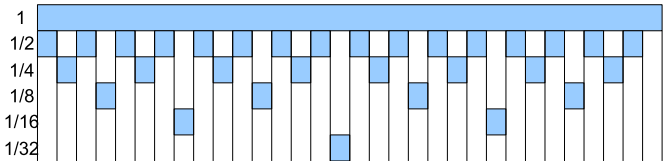

Fig. 3. Binary split allocation of $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32}$

The use of a binary tree ensures us that any additional fraction does not interfere with the already scheduled fractions, but it also ensures that the fractions are equally spread out over the available slots. The scheduling problem is thus reduced to merely following the path in a binary and checking whether the path is still free.

In the beginning we mentioned that the requested bandwidth can be split up in a number of slots and a fraction of a slot. The fractional slot is approximated by means of Egyptian Fractions and scheduled over the different frames, as mentioned earlier. The lowest possible denominator also determines the number of frames that a periodic cycle consists of. The full slots are allocated to the sensor for each frame. The allocation of the full slots forms no issue in determining in which frame the slot can be used. However, the allocation of the slots in a single frame also needs to support the predefined goals, such as a spread out and fair allocation. One way of allocating the full slots is to represent the fraction of the number of required slots over the total number of slots per frame by means of an Egytian Fraction. By putting a constraint on the total number of slots per frame, such that the total number always needs to be a power of two, the fraction of the requested slots over the total number of slots can be represented perfectly by means of an Egyptian Fraction with a denominator equal to the power of two. The unit fractions of the Egyptian Fraction can also be scheduled by means of the binary tree, with the only difference that the start positions now indicate a certain slot in the frame, instead of a certain frame within the periodic cycle.

The accuracy of the Egyptian Fractions is perfect concerning the number of required slots, since the total number of slots per frame is limited to a power of two. The accuracy regarding the fractional slots depends on the smallest unit fraction possible. The lower this unit fraction, the more accurate the approximations are, but also the more frames a cycle consists of. Later on in the analysis part will be explained what kind of influence this has.

As an example, imagine that we have a frame that is split into four slots. The total available bandwidth is 4 bytes per second, which is the capacity of a frame, which is split into four slots. A sensor needs to be scheduled that requires 2.25 bytes per second, $9/16^{th}$ of the total bandwidth. First, the full slots are scheduled by dividing the requested rate by the slot size, which is 2.25, and schedule the fraction of the integer part over the number of slots, $\frac{2}{4}$. The representation of the fraction in an Egyptian Fraction format is $\frac{1}{2}$ and according to the binary allocation tree, slot 0 can be used as the first slot. The next slots are calculated based upon a frequency of $\frac{1}{2}$, thus every two slots. Up till now, $\frac{8}{16}$ is scheduled, which is not enough. The

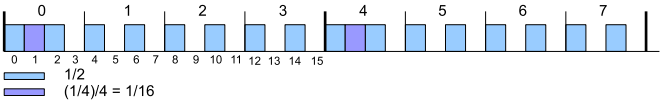

Fig. 4. Allocation of 9/16 in a 4 slot period

requested amount is $\frac{9}{16}$, so an extra $\frac{1}{16}$ needs to be scheduled. First, a single slot that can be shared by multiple sensors needs to be scheduled. The first slot that can be found in the allocation tree is slot 1. Next, the fraction of the remainder (1) and the slot size (4) needs to be approximated by means of an Egyptian Fraction. The fraction $\frac{1}{4}$ is a perfect Egyptian Fraction and does not need to be simplified. According to the binary fraction allocation tree, position 0 is the first allocation that can be used for the fraction $\frac{1}{4}$. The result is shown in Fig. 4, where slot 1 is scheduled in frame $0, 4, 8, \ldots$ which results in a perfect fit of the requested rate of $9/16^{th}$ of the total available bandwidth.

Note that the approximation is made up of fractions of which the nominator is a power of two. If this information is put in a binary representation by representing each fraction as the number of smallest fractions that are needed to come to this fraction, a lot of information can be given with only a few bytes of data. Only a single byte is needed in order to represent of which fractions the approximation consists. For each fraction, there is then the need to specify the slot id and the frame in which it is scheduled. The size of the slot id is determined by the number of slots within a frame, while the size of the information, that holds the frame in which the slot is scheduled, is the precision of the fractions.

IV. Implementation on TMotes

A. TDMA MAC

The proposed protocol is a TDMA scheduling protocol that calculates and assigns free slots according to the requested bandwidth. Not only the implementation of this protocol is required to test it, but also the provision of a TDMA framework. This TDMA framework should not only provide the framing

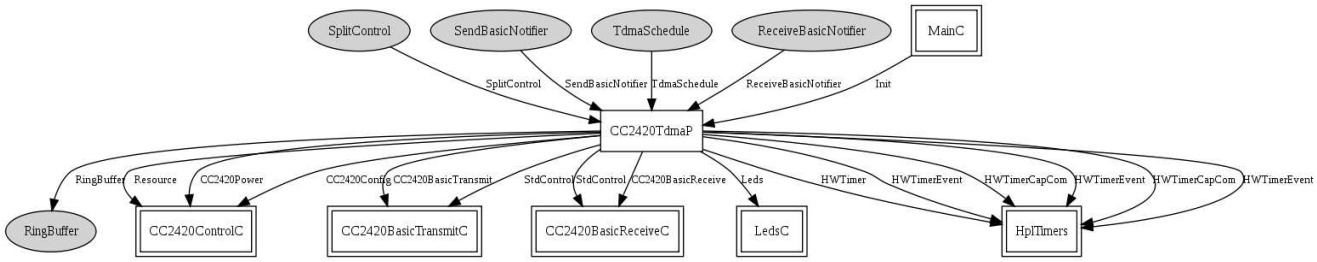

Fig. 5. TinyOS module CC2420Tdma

and slots, but also a means to synchronize and autonomously construct a network. During the network construction, the required bandwidth from the requesting sensor needs to be passed to the scheduling protocol. Due to the scope of this research, the network is limited to a star topology, where sensors try to synchronize with the sink sensor and send it their data.

In order to achieve all this, we splitted the TDMA frame into three distinct periods: the *'Synchronization period'*, the *'Identification period'* and the *'Data period'*. The *'Synchronization period'* contains four *'Sync slots'*, the *'Identification period'* contains four *'Id slots'* and the *'Data period'* contains *n 'Data slots'*. The number of *'Data slots'*, *n*, needs to be a power of two in order to comply with the requirements of the scheduling protocol. The TDMA framework has already synchronization provisions for a multihop network, hence the four *'Sync slots'* and *'Id slots'*. For a star topology, we have only a single *'Sync slot'* and a single *'Id slot'*. The primary function of the *'Sync slots'* is to send information to the other sensors that allows them to synchronize to the sender. The *'Id slots'* are used to send information regarding the required amount of bandwidth the sender needs. The actual data is sent in the *'Data slots'*, assigned to this sensor by the scheduling protocol.

A sensor needs to go through a couple of stages before it can send its data. First, the sensor needs to capture all radio transmissions and filter out the synchronization messages. The synchronization messages contain the information required to synchronize. Out of these synchronization messages, the sensor will pick the one whose sender has the best link quality. By using the information that is embedded in the synchronization message, the sensor synchronizes its frame to that of the sender of the synchronization message. After the verification that the two sensors are in sync, by checking whether a new synchronization message is received at the expected slot from the expected sender, the sensor enters the next stage.

The next step is the sensor sending an identification message in the *'Id slot'* with the same index as the *'Sync slot'* where the sensor receives the synchronization messages. The identification message contains, amongst others, information regarding the required bandwidth of the sensor. The sink sensor receives the identification message and extracts the necessary information. The amount of requested bandwidth is forwarded to the scheduling protocol. Our scheduling protocol represents the requested bandwidth as an Egyptian Fraction and calculates the slots and the frames to which they need to be assigned. The resulting schedule is placed in a synchronization message with as destination the *id* of the sensor that requested the bandwidth. As soon as the requesting sensor receives the schedule, it can start sending its data to the sink in the *'Data slots'* that are assigned to it. If the sensor does not receive any assignment information within a certain time frame, it resends the identification message by means of a simple backoff algorithm.

B. TinyOS implementation

In order to implement the TDMA framework, we created a TDMA scheduler, CC2420TdmaP (Fig. 5), that offers the capability to schedule tasks at a specified time and with a certain maximum duration. The task can be either a receive order or transmit order. The TDMA scheduler makes use of a counter which upper boundary determines the TDMA frame size. This results that the time when a task needs to be executed, is expressed as the number of times the counter incremented since the beginning of the frame, or simply said, the counter value. As counter, the TimerB counter is used in *Up mode*, with as clock source a 32KHz crystal (through a proprietary interface in order to speed up the interrupt signaling). Despite the low frequency of the clock, it is the only one that is stable enough to perform the task. The internal oscillator drifts too much, resulting that we would need to resynchronize the sensors too often. By means of the stable crystal, a small correction of a single count needs to be performed every four seconds.

Synchronizing two sensors, means they need to set their TDMA frame counter to the same value as the other sensor.The synchronization messages contain the information necessary to perform the synchronization, i.e. the *'Sync slot'* in which the message has been sent. By using the radio characteristic that it gives an interrupt when it detects a *'Start of Frame Delimiter'* (SFD), it is possible to know at what time the message was received. The SFD interrupt triggers the capture register CCR1 to take a snapshot of the counter value. By comparing the expected and the actually received counter values, we can increase or decrease the current frame size for a single frame. The next frame, the TimerB boundary value is reset to the original frame size.

Since it is most unlikely that two unsynchronized sensors have

the same counter value, we have no idea when to look for a synchronization message. Therefore we need to let the sensor scan for messages, meaning that the sensor needs to listen for messages for an undetermined time. Upon reception of a message, after checking that the message is a synchronization messsage, we can apply the above method to synchronize the sensors and to start receiving and sending in specific slots. After that, we still can adjust the frame size by one count, without any loss of packets, if a small correction seems to be required. This prevents the two sensor counters to drift away, forcing to perform a larger adjustment, which most likely would result in the loss of data.

The adjustment of the TDMA frame counter is performed at the CC2420TdmaP module, but the intelligence concerning the different stages is contained in the CC2420ActiveMessageTdmaP module (Fig. 6). The module is responsible for feeding the CC2420TdmaP module with new tasks. This happens in a controlled manner, the tasks for a certain frame are scheduled during the previous frame, or at least at the beginning of the current frame in case there is not enough space at the TDMA scheduler.

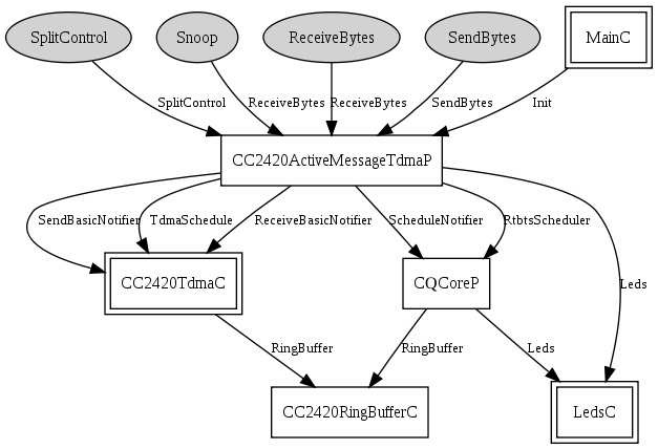

Fig. 6. TinyOS module CQCore

The passing of data is obviously crucial when trying to send or receive something. We decided not to use the TinyOS methods for a couple of reasons. First of all, we did not want to perform memory copies all the time. Secondly, we are working with messages of different sizes. When storing them in the TinyOS manner, the message has the size of the largest possible message. This is not a good policy when working with devices that have limited memory resources. This explains why we have designed the RingBuffer module. This module can contain multiple ringbuffers of different sizes, indicated by a type identifier. At startup, the module needs to be configured, for each buffer type, it needs to know the id of the buffer, its size and the number of buffers. When requesting an empty buffer of a certain type, a pointer to the first buffer of that type which is free, is given, while indicating in the RingBuffer module that the given buffer is busy. This gives the new owner of the buffer the opportunity to fill the buffer with data, after which the buffer can be marked as full. Analogous, when requesting a buffer of a certain type which is full, the first full buffer is given, while indicating that the given buffer is busy.

Knowing that the CC2420ActiveMessageTdmaP module has exclusive access to the RingBuffer module regarding the marking of buffers, guarantees that race conditions cannot happen. The ring buffer approach allows to fill buffers and to mark them as full while performing actions. When a message of that type is scheduled for transmission, the buffer is picked out and a pointer to it is given to the TDMA scheduler, together with the corresponding timing information. The same counts for the reception of data, a pointer to a free buffer is handed to the TDMA scheduler, which at a certain point starts the reception of the data. After completion, the TDMA scheduler notifies the CC2420ActiveMessageTdmaP module that the reception has been completed. Even while the reception of that data is going on, a new task of the same type can be scheduled for reception, since the RingBuffer module has multiple buffers of a single type. There is no need to worry about processing the data in time that we might risk to loose data.

The processing of the received data can easily happen in a background process. Depending on the type of data, the CC2420ActiveMessageTdmaP module posts a new task, every time it is notified about a new data arrival. In case the data is an identification message and the requesting sensor has not been granted yet the bandwidth it requests, the requested bandwidth is extracted from the message and forwarded to the CQCore module, which is our scheduling protocol. The protocol checks whether enough bandwidth is available and allocates sufficient bandwidth for the requesting sensor. This information is returned to the CC2420ActiveMessageTdmaP module, which puts the new scheduling information for the sensor in the next synchronization message. The information regarding the scheduling is only destined for the requesting sensor. The other sensors might use the synchronization message for synchronization purposes, but they may not use the scheduling information. The allocated slots and the fractions during which the sensor may use them is stored in memory by the CQCore module. The RingBuffer module is also used for this purpose. One of the buffers that are created during configuration of the RingBuffer module, is a buffer where the scheduling information is kept. There is only one instance of this buffer, so it is hardly to call a ring buffer. The reason for using the RingBuffer module, is that there is an inverse relation between the sizes of the data buffers and the scheduling buffer. The size of the TDMA frame is limited by the 16 bit precision of the hardware timer. Hence, the maximal datasize can only be used as long as the number of slots is low enough. When it is required to have more data slots, the slots need to be smaller than the maximum number of bytes that can be transmitted. Hence, the size of the data buffers decreases. On the other hand, the number of bytes required to hold the scheduling information increases with the number of slots. This results in a sort of equilibrum because of the shared memory of the RingBuffer module. The RingBuffer module

led to the need to change the interfaces towards the radio. Also due to the TDMA scheduling, changes were required towards the radio interfaces and their control. One of the advantages of TDMA is the elimination of idle listening. The radio is only active when it really needs to perform some action. The CC2420TdmaP module (Fig. 5) supports this characteristic by controlling two radio interface modules: CC2420BasicTransmitC and CC2420BasicReceiveC. When a task, either an RX or TX task, needs to be scheduled, the respective radio interface is activated and the task is executed. Either after completion of the task, or after the specified duration of the task, the radio interface is deactivated again. Deactivation of the interface means that the radio interface is put in idle state or in sleep state if there is enough time.

V. Results

A. Simulation

Before implementing the protocol on actual hardware, we performed simulations to analyze in detail the performance of our protocol. We simulated the byte-wise arrival of data, that would be sent in bursts at the instants in time, specified by our scheduling protocol. To keep the focus completely on the analysis of the scheduling protocol, we assumed an ideal network (without any preambles, switch times,$\cdots$) that is always synchronized, hence without overhead. In other words, the available bandwidth divided by the number of slots is the bandwidth of a single slot. For our simulations, we took a radio with a bandwidth of 2400 bytes per second, a frame size of one second and we simulated different rates with 8, 16 and 32 number of slots per frame. We picked a couple of those results to show them.

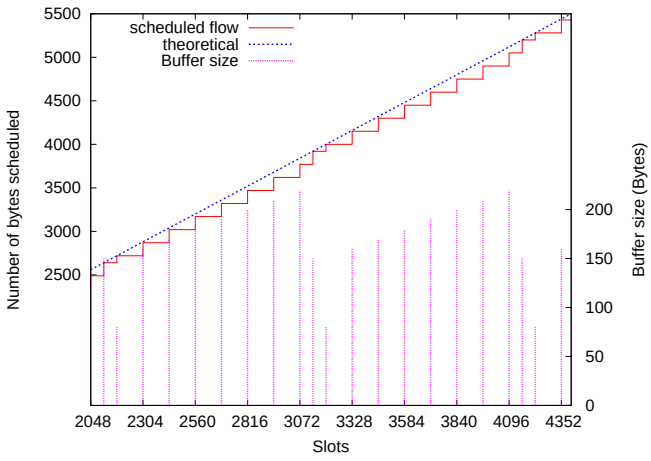

Fig. 7. data arrival and buffer size, simulated requested rate = 20 bytes per frame, approximated as $\frac{1}{8} + \frac{1}{64}$

Fig. 7 depicts a rate of 20 bytes per second with 16 slots per frame. It shows the resulting accumulated data, compared to the bit per bit arrival of the data, which can be seen as the red line in steps compared to the *'theoretical'* gradient of the bit by bit arrival (left axis). The buffer size in the figure (right axis) depicts the difference in bytes between the *'theoretical'* gradient and the stepwise data transmission, which is the result of the scheduling protocol. The periodicity of the buffer size is a direct result of the scheduling protocol. The rate is approximated as one fraction, which is too small, and a second fraction that compensates the difference between the requested rate and the first fraction. This leads to a situation where there is not enough capacity to send all the received data at once. The data in the buffers increases slowly until the slot, which is allocated according to the second fraction, is scheduled for transmission. The maximum buffer size that we get with this rate is 220 bytes.

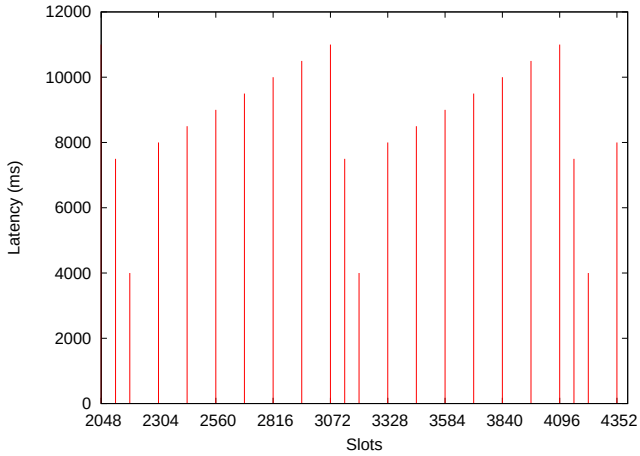

Fig. 8. Latency, simulated requested rate = 20 bytes per frame, approximated as $\frac{1}{8} + \frac{1}{64}$

Not sending the available data at once, means of course that not only the buffer size increases, but also the latency, as can be seen in Fig. 8. The figure depicts the latency for the same requested rate, 20 bytes per second, with 16 slots per frame. The latency is defined as the time between the arrival and the scheduling of the data. The maximum latency that we get with this rate is 11 seconds, or 11 frames. The maximum latency can be calculated from the maximum buffer size, since the maximum latency can be regarded as the time needed to fill a buffer with the maximum buffer size, at the given rate. If we do the math, 220 bytes / 20 bytes per second gives us 11 seconds for the maximum latency, which matches with the figure.

Fig. 9 depicts the buffer size (right axis) and the cumulative amount of scheduled data (left axis) versus the *'theoretical'* gradient (left axis) of the rate 58 bytes per second with 16 slots per frame. The rate is approximated as $\frac{1}{4} + \frac{1}{8} + \frac{1}{64}$. Following the same reasoning we did last time, the first two fractions do not provide sufficient slots within a certain amount of time to send all data that has arrived during that interval. The slots scheduled according to the second fraction compensates the difference a bit between the requested rate and the first fraction, but not sufficient. A third fraction is necessary.

Thanks to the periodicity of the scheduling protocol, we get some very interesting properties. As can be seen from the previous figures, there is some kind of relation between

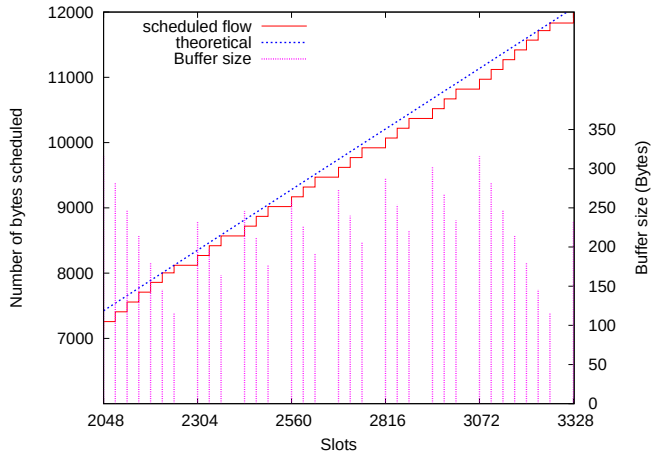

Fig. 9. Data arrival and buffer size, simulated requested rate = 58 bytes per frame, approximated as $\frac{1}{4} + \frac{1}{8} + \frac{1}{64}$

the maximum buffer size and the fractions that approximate the requested rate. After analyzing thousands of rates and slot sizes, we came up with the following formula for the maximum buffer size, of which the deduction can be found in technical document [11]:

$$
\begin{aligned}
&if\ n = 0: \\
&Max\_buffer = bw * f_0 \\
\\
&if\ n > 0: \\
&Max\_buffer = S \\
&+ \left(\tfrac{f_0}{f_1} - 1\right) \tfrac{1}{f_0} (bw - f_0 S) \\
&+ \sum_{2 \le i \le n} \left(\frac{f_{i-1}}{f_i} - 2\right) \frac{1}{f_{i-1}} \left(bw - \sum_{0 \le j \le i-1} f_j S\right)
\end{aligned}
\tag{1}
$$

with S as the slot size, bw as the requested bandwidth and $f_0 + \cdots + f_n$ as the Egyptian Fraction that approximates the requested bandwidth. Since there is a direct relation between the maximum buffer size and the maximum latency, we can calculate the maximum latency if we know the slot size, the requested bandwidth and the fractions that approximate the requested bandwidth.

B. Tmote tests

The TDMA frame is configured to have 16 slots of the maximal slot size, 127 bytes. By also taking into account the synchronization overhead, the information exchange overhead and the preamble and radio switch time of each packet, the total frame size is 4242 timer counts large, or 132.6 ms. Note that the 127 bytes slotsize also includes the two byte CRC and data header. The data header we use is 9 bytes large, since we send information that is needed for the performance analysis, such as the buffer size, the frame number and the value of the frame counter at the time of arrival of the sent data. In order to make a well-founded comparison with the simulations, we implemented a routine at the sensor side that generates data, byte per byte, at the requested rate. To achieve the highest possible accuracy regarding the results, we do not represent the requested rate as number of bytes per second, but as number of bytes per frame. This allows us not to perform any mathematical operations on the requested rate that could result in the loss of precision on an embedded device.

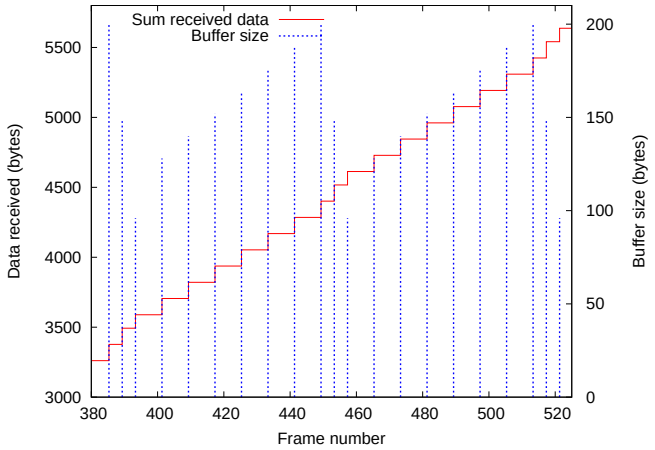

Fig. 10. Data arrival and buffer size, requested rate = 16 bytes per frame, approximated as $\frac{1}{8} + \frac{1}{64}$

A few results are depicted here. Fig. 10 represents the cumulative amount of data received at the sink (left axis) and the buffer size at the receiver side (right axis) for a rate of 16 bytes per frame. The rate is approximated as $\frac{1}{8} + \frac{1}{64}$. Notice that the figure looks similar as Fig. 7, which depicts the simulation result for the same approximation, but with a different slot size. We know that the results of the simulations can be quantified by Formula (1). In order to compare both of them, we calculate what the expected maximum buffer size is, according to the formula. For the given parameters, $S = 116, bw = 16, f_0 = \frac{1}{8}\ and\ f_1 = \frac{1}{64}$, we get that we should have a maximum buffer size of 200 bytes. Fig. 10 shows that the maximum buffer size is indeed 200 bytes.

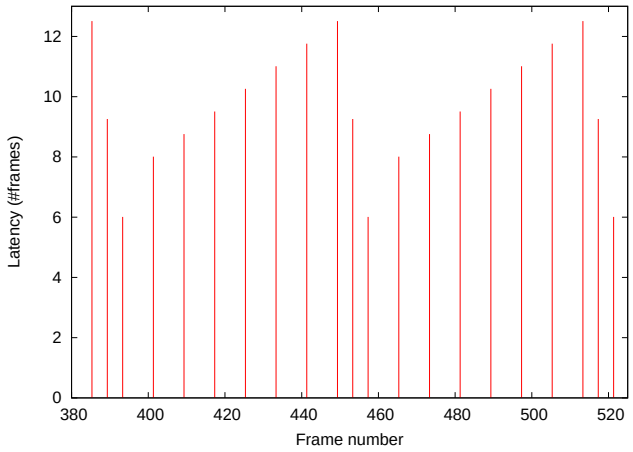

Fig. 11. Latency, requested rate = 16 bytes per frame, approximated as $\frac{1}{8} + \frac{1}{64}$

Fig. 11 shows the latency of the data received at the sink, for a requested rate of 16 bytes per frame. The resulting latency

is achieved by calculating the difference between the time the data arrived at the sink and the time the data *'arrived'* at the sensor, information that is sent together with the data. We notice that the latency has a similar flow as the maximum buffer size. By using the same methodology as with the simulation results, we calculate the expected maximum latency from the maximum buffer size. This means that the maximum latency is the time needed to fill a buffer of the maximum buffer size (200 bytes) at the specified rate (16 bytes per frame). The calculated maximum latency is 12,5 frames, which is precisely the maximum latency that can be found in the figure.

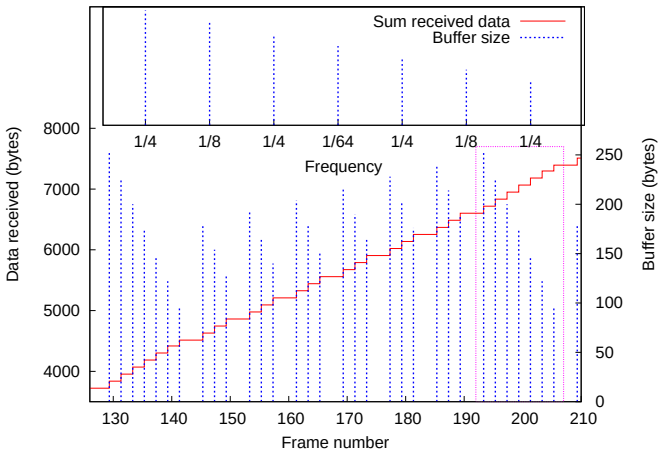

Fig. 12. Data arrival and buffer size, requested rate = 45 bytes per frame, approximated as $\frac{1}{4}+\frac{1}{8}+\frac{1}{64}$

A second example (Fig. 12), shows the cumulated amount of data that arrived at the sink (left axis) and the buffer size (right axis) for a requested rate of 45 bytes per frame. We notice again the similar flow as the Fig. 9, that represents the simulation results for the same Egyptian Fraction: $\frac{1}{4}+\frac{1}{8}+\frac{1}{64}$. The resulting value from Formula (1) for the maximum buffer size is 252 bytes. When comparing this value again with the results from the test with the Tmotes, we notice that we get the same value for the maximum buffer size.

By calculating the maximum latency from the maximum buffer size for this rate, we see that we should achieve a maximum latency of 5,6 frames. Fig. 13 shows that the perceived maximum latency at this rate is indeed 5,6 frames.

VI. Conclusion

We proposed here a slotted scheduling protocol. The protocol is designed for fairness, even when the network is crowded with high bandwidth sensors, the low bandwidth sensors should still be able to send their data. The second goal is energy preservation. It is better to use the network for a small unit of time and then utilizing the full bandwidth, than sending each time just a bit of data. This is also more efficient concerning overhead. The more data that is being sent at a time, the smaller the header is in comparison to the amount of data.

The protocol takes several limitations into account in order to

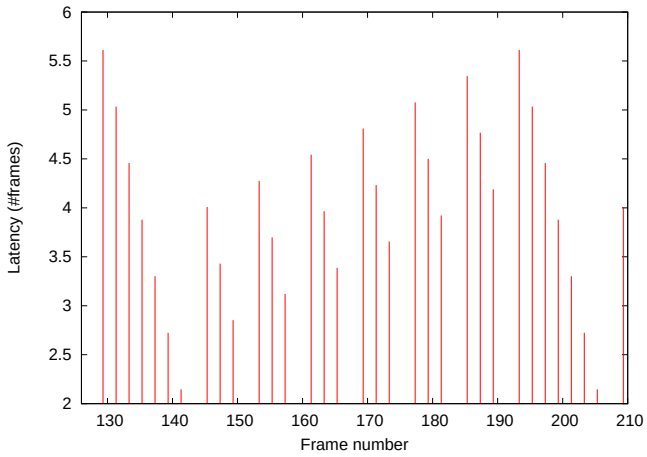

Fig. 13. Latency, requested rate = 45 bytes per frame, approximated as $\frac{1}{4}+\frac{1}{8}+\frac{1}{64}$

preserve the feasibility to implement it on hardware. First of all, it is designed in such a way that the slot allocation needs to be sent only once during the whole lifetime of the network. The design is based on a periodical cycle, which allows to send the slot allocation, which is afterwards repeated over and over again. Any subsequent changes in the topology also do not influence the already scheduled slots. The design is so efficient concerning the information containing the slot allocations, that it can be sent in a very small number of bytes.

By means of simulations and extensive analysis of the results, we found out that the scheduling protocol is predictable, because of its determined schedule. We can calculate the maximum required buffer size a sensor needs for a given bandwidth. By means of this maximum buffer size, the latency can be calculated. We have created a scheduling protocol that is capable to schedule real-time tasks. Although the simulation network assumptions discarded synchronization overhead, preambles and switching times, the results match perfect with the results obtained by testing the protocol on actual hardware for rates lower than the slot size. A more extensive study needs to performed on the rates larger than the slot size.

References

[1] http://www.snm.ethz.ch/Projects/TmoteSky
[2] http://www.tinyos.net/
[3] http://en.wikipedia.org/wiki/Weighted_fair_queuing
[4] http://www.sics.se/~ianm/WFQ/wfq_descrip/node21.html
[5] A. Demers, "Analysis and Simulation of a Fair Queuing Algorithm", *Internetworking Research and Experience, 1990*
[6] A. Parekh, R. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single node case", *IEEE INFOCOM 1992*
[7] http://en.wikipedia.org/wiki/Generalized_processor_sharing
[8] Chung Shue Chen, Wing Shing Wong, "Bandwidth allocation for wireless multimedia systems with most regular sequences", *IEEE Transactions on Wireless Communications, Volume 4, Issue 2, March 2005 Page(s): 635 - 645*
[9] Kevin Gong, "Egyptian Fractions", *Math 196 Spring 1992*. UC Berkeley.
[10] Algorithms for Egyptian Fractions: http://www.ics.uci.edu/~eppstein/numth/egypt/intro.html
[11] Deduction of Egyptian Fraction scheduling algorithm Formula: http://www.pats.ua.ac.be/wim.torfs