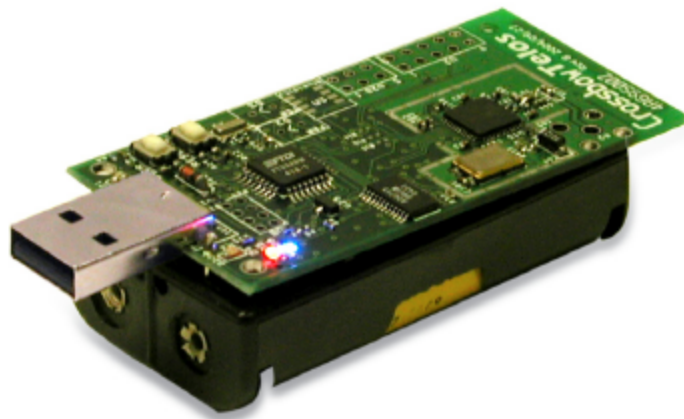


XOR packets

Miniproject - TIWSNE



June 19th, 2013

Group Delay

201300257 - Cecilie Crone Rasmussen

201300248 - Peter Jessen Vang

201300254 - Mathias Markussen

201201706 - Maiken Bjerg Møller

TIWSNE – Q4 2013

Mathias John Møjbæk Markussen

Peter Jessen Vang

Cecilie Kirstine Crone Rasmussen

Maiken Bjerg Møller



AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

Abstract



Content

[1. Introduction](#)

[2. Theory](#)

[2.1 Position in the OSI Model](#)

[2.2 Energy Problems](#)

[2.3 Retransmission](#)

[3. Implementation](#)

[3.1 Setup](#)

[3.2 Program flow](#)

[3.3 Packets](#)

[3.5 Test of the program](#)

[4. Results and Discussion](#)

[6. Conclusion](#)

1. Introduction

During this mini project we are investigating how a third mote in a network can relay transmissions between two motes, who want to exchange files but are out of reach. In the project two methods are considered.

One way is the naive way, where the middle node (Node C) acts like a repeater, and forwards the packets from one node to the other and vice versa. This is illustrated in figure 1.

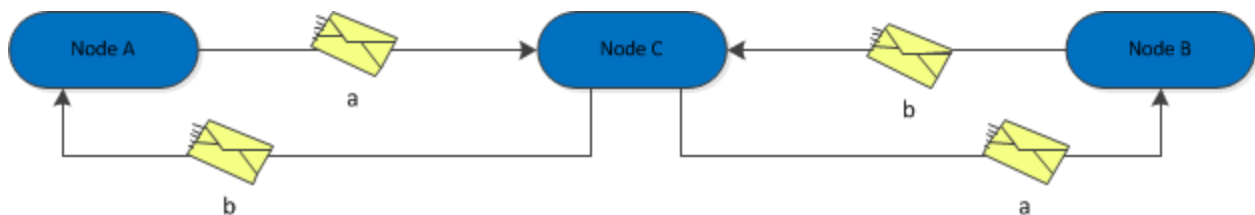


Fig. 1 - The naive way

The other way is the XOR way, where the middle node (Node C) receives packets from the end nodes (Node A and B) and XORs these packets together and broadcasts the result. The end nodes decode the received packets by XORing with the packets they originally sent out. This results in the exchange of files between the end nodes with only one broadcast, as opposed to two unicasts. This concept is illustrated in figure 2.

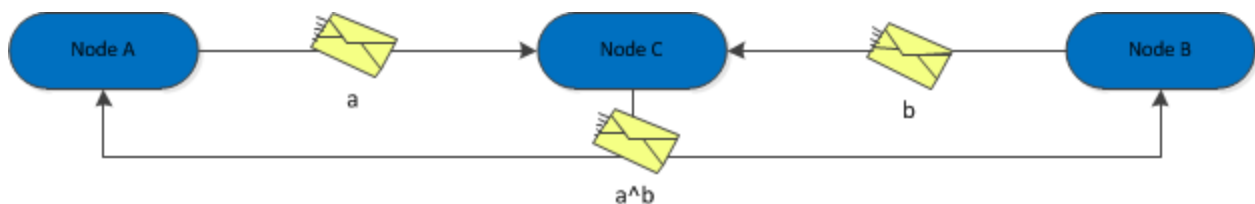


Fig. 2 - The XOR way. $a = (a^b)^b$. $b = (b^a)^a$.

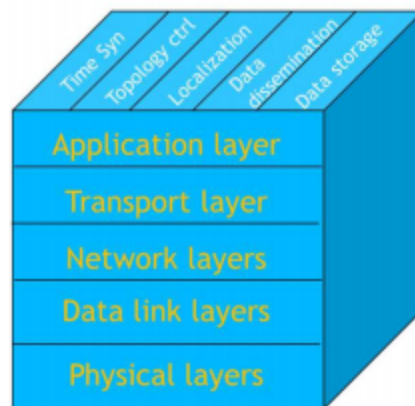
The benefit of the XOR way is that the middle node only needs to transmit half as many times as in the naive way. On the other hand, all three nodes have to do the XOR computations. It is assumed, however, that the XOR way will save power, compared to the naive way, because computation consumes much less power than broadcasting. The performance of the described ways is investigated in this project.

2. Theory

This chapter briefly describes the part of the theory about wireless sensor network, that is important for this project. This includes the relevant layers in the OSI model, and how this project is related to the different layers. In addition, the theory about energy challenges and the need for retransmission is described.

2.1 Position in the OSI Model

Figure (XXX) show the different layers of the OSI model, and the hierarchy of the layers. This mini project's implementation works mainly on one layer of the model, the transport layer.



The transport layer protocol is provided with the knowledge of incoming data, e.g. how many packets are to be received in this transmission and the order of the packets. To ensure a correct transmission, the transport layer has to request a retransmission in case of lost packets.

The underlying layer is the network layer, which is responsible for the routing and the allocation of network unique addresses and roles in the network. In this mini project the network layer is not implemented directly, because the roles and unique addresses for the nodes are decided when programming the individual motes (see section 3.1).

2.2 Energy Problems

In wireless sensor networks it is important to consume as little energy as possible. This is to avoid the need for changing or recharging the batteries in each network node. Transmission costs a lot of energy compared to sleep or even computation, so a good way to minimize the energy consumption in the network is to reduce the transmission ratio. By doing some local computation or data aggregation, it is possible to reduce the transmission ratio. This is exactly

what this project is about. Doing the XOR operation reduces the number of packets that the middle node has to send by a factor two.

Another thing that can save energy is to avoid or reduce collision, overhearing and idle listening in the wireless sensor network. When collision happens, data is lost. This means that energy is wasted by way of useless transmitting and useless listening for the packet. It also means that retransmission is required. As mentioned transmission is expensive, so collision avoidance is very important if energy is to be saved. Overhearing means that a node in the network receives a packet that is not meant for it, and therefore discards it. In other words, the overhearing node could as well be sleeping, and thus save some energy. Waste in energy could also happen if a node is awake (radio is on), and do not receive anything. This concept is called idle listening. The overall solution to avoid collision, overhearing and idle listening is to strictly control when each node transmits, as well as when each node should be awake for receiving data. These problems should be handled by the MAC protocol in the data link layer.

2.3 Retransmission

Because of packets loss and collision retransmission is necessary. But it is an issue how to find out this packets loss and/or collision has taken place. By knowing the sequence numbers the node is able to detect a gap. When that happens the retransmission can be done in different ways; Proactive fetch reacts immediately after a packet loss is detected and stops transmission until this packet is retransmitted. Otherwise the transmission will continue and then report missing packets after the transmission. This demands more memory because lost packets have to be cached in the transmitting node.

Another thing to consider is the order of the packets. For a transmission where the order of the packets is important, the retransmission has to be done immediately.

3. Implementation

For this mini project where files are to be exchanged, test files are chosen to be text files with lyrics from two different songs. The end nodes are provided with the two different text strings from two h-files which are included from the code. The h-files are like other program files saved in the internal memory of the mote. This is of course not a realistic scenario in a real-world application. It's much more likely that the data would be fed to the mote, for example through RS232 or I2C, or sampled from a sensor.

When the data is received, it is printed directly to a terminal window using Printf along with debugging data and error messages. This is not a very realistic scenario either. In a real implementation the data would either be saved to local storage to be retrieved later, or it would be printed without debugging information after it had been verified.

These decisions were made in order to focus on making the data exchange and network parts of the code as good as possible and not spend time researching how to move data to and from the

motes.

3.1 Setup

The software used for setup and tests:

- VMware player 4.0.3
- Ubuntu 10.04 (running as virtual machine in VMware)
- TinyOS version 2.1.1
- nesC 1.3.1 (nesC compiler)

The hardware used for setup and tests:

3 x Mote TelosB (Crossbow Telos Rev B)

The software for this project is written in nesC, a variant of the programming language C. Linux is used as environment for source code editing, compiling and Mote communication. VMware is used to run Linux, and TinyOS 2.1.1 is used as operating system for the Mote.

Three Motes are used in this setup.

- Mote 1 (A) and 2 (B) with the EndNode program implemented
- Mote 3 (C) with the MidNode program implemented

Terminal commands used for setup and tests:

To watch the traffic through the radio, the java Printf Client application provided from tinyOS is used. Java Print Client:

```
~/ $ java net.tinyos.tools.PrintfClient -comm serial@ttyUSB0:telosb
```

To program the mote with the desired application the following command is used. The parameter after install, (comma) is the node ID and the parameter after bsl, (comma) is the path to the mote.

```
~/<application-path> $ make telosb install,1 bsl,/dev/ttyUSB0
```

3.2 Program flow

The purpose of this program is to make the two end nodes able to exchange data through the middle node. This section explains the overall idea behind the implementation which makes this possible.

The data exchange starts with each end node sending a request to the middle node, telling that it has data that it wants to exchange with the other end node. When both nodes has made their request, the middle node answers with an acknowledgment, which causes the end nodes to start the data transmission. The two end nodes each broadcast one packet at a time, and will

not broadcast a new packet until it has received a new packet from the middle node. When the middle node has received a packet from each of the end nodes, it will xor each byte in the packets into one new packet and then broadcast this to the end nodes. When the end nodes receive an xor'ed packet, they will decode it to recover the packet sent from the other end node by performing another xor operation, this time using the received packet as well as the one that was sent out previously. This process repeats until one end node has sent out all the data it has in the send buffer, after which it will start to only listen to the data that comes in from the other node. This means that the data that is being exchanged can have different sizes in the two nodes. The process of starting the data transmission and how the transmission works when no errors occur is shown in figure.

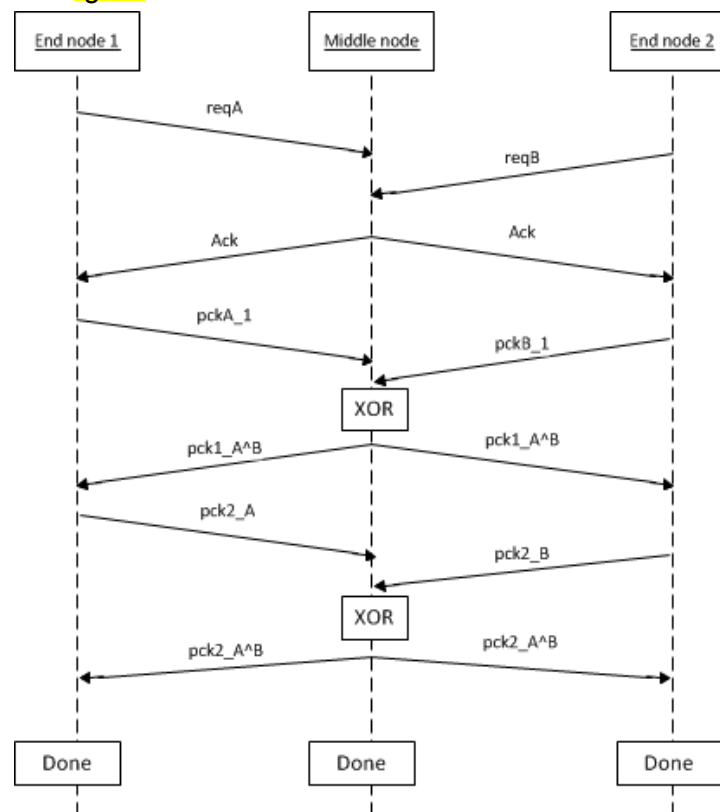


Figure 1 - Shows the data flow when an exchange is started and completes without error.

When transmitting errors can happen, and a scheme handling retransmission of lost packets is therefore necessary. The retransmission is implemented so it is done during the exchange. This means that if an end node does not receive the expected packet from the middle node, it will request a retransmission and not transmit the next packet until the retransmission has occurred. A counter in the packets is used to keep track of which package is expected from the middle node. This ensures that all packets are received in the right order at the end nodes. Packets can be lost both from end node to middle node, and the other way around. How retransmission is

implemented in these two different scenarios can be seen in [figure](#).

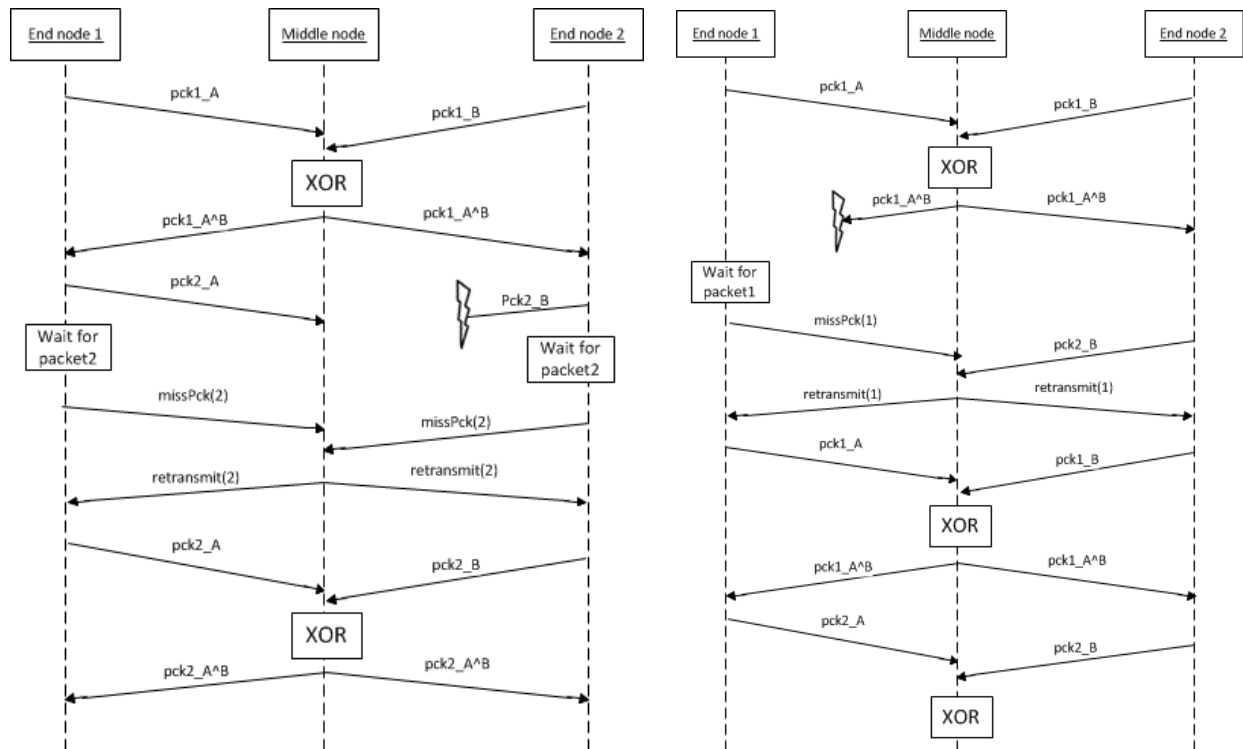


Figure 2 - shows two different scenarios where retransmission is necessary. The diagram on the left hand side shows a situation where a packet is lost on its way to the middle node, and the other one shows a situation where the xor'ed packet does not reach an end node.

The figure on the left hand side shows a scenario where a packet is lost from end node to middle node, in this case packet number 2 from end node 2. This means that an xor-packet will not be generated. This means that both end nodes send a packet to the middle node, telling that they missed packet number 2. The middle node will then request both end nodes to transmit packet number 2 again. If the packet is lost from the middle node to end node, as in the figure to the right, where end node 1 does not receive the xor'ed packet, it is only end node 1 which requests a retransmission from the middle node, and the middle node still requests a retransmission from both end nodes. End node 2 has received the xor'ed packet correct, and may therefore transmit the next packet before it gets the request to resend. This will not have any effect on the system, the packet is just lost.

The complete implementation described above is covered in more detail in the next sections.

3.3 Packets

To handle the exchange of files, the data is divided into different packets. All packets used are of the type Active Message (AM) and three different packet structures are defined in the mini project's h-files.

radio_init_msg: is a packet for initiating the size of a given file to be transferred, by stating the number of packets in the file. The packet is constructed like this:

```
typedef nx_struct radio_init_msg {  
    nx_uint16_t numPkt1;  
    nx_uint16_t numPkt2;  
} radio_init_msg_t;
```

numPkt1 og numPkt2 is the number of packets that end node 1 and end node 2 wants to send. They are assigned and broadcasted when data is to be exchanged. This is done by pressing the user button.

radio_data_msg: is the packet containing the data that i exchanged. It is constructed like this:

```
typedef nx_struct radio_data_msg {  
    nx_uint8_t senderID;  
    nx_uint16_t pktNum;  
    nx_uint8_t msgString[STRLEN];  
} radio_data_msg_t;
```

senderID tells which node the packet comes from, and is assigned the TOS_NODE_ID. pktNum is the packet number, and msgString is the data that is transmitted. STRLEN is set to 25, which means that msgString can contain 25 bytes.

radio_goto_msg: this packet is used if retransmission is necessary, and is constructed like this:

```
typedef nx_struct radio_goto_msg {  
    nx_uint8_t senderID;  
    nx_uint16_t gotoNum;  
} radio_goto_msg_t;
```

It consist of a senderID like the data message as well as a goto number (gotoNum). The goto number is the number of the packet which the senderID node wants retransmitted. This packet

is used both when a packet is lost from end node to middle node, and vice versa.

3.4 Program structure

The implemented software is two event-based applications that run on top of TinyOS. The `Boot.booted()` event is fired every time the motes has been powered up, and is therefore used to initiate the platforms. The radio controller is started, along with the needed timer and buttons in the end nodes. Furthermore the end nodes load the appropriate data files based on the individual IDs that they were assigned during compiling. After this the program goes into a waiting state, until the data transfer is initiated.

3.4.1 Hand shaking

The data transfers is initiated by the end nodes, by sending an initialization message to the middle node. In this message they each specify the number of packets they want to send by writing it to one of the fields in the message. In the developed prototype programs the data is hardcoded into the motes and the initialization is sent when the user button is pressed. When the mid node has received an initiation message from each end node, it broadcasts an initialization message, where both fields are assigned with the respective length from the two nodes. When the end nodes receive this package they store the number telling how many data packets they are supposed to receive, and set some flags telling that the transfer is in progress.

3.4.2 Transmission

During the transfer of the data packets the timing is controlled by a timer in each of the end nodes. These timers fires is initiated at boot time and an event every 100 ms. Since the timers are initiated at different times and the transmission time is very short, this implements a very crude form of collision avoidance. The mid nodes doesn't use a seperate timer, but simply broadcasts the XORed message after it has received a data packet from both end nodes. The data reception is handled in a separate interrupt. In here the type of the packet is determined, and the appropriate flags are set and the content stored. Every time the timer of an end node is fired, it sends out a new message based on what has been received since last interrupt. This can result in three different scenarios:

3.4.2.1 Recieved step back command

First the `gotoFlag` is evaluated. If this has been set, it means that a packet has been lost somewhere in the system, and the mid node has broadcasted a command to go back to a specified packet number. If this is the case, the packet counter is set to the requested value, and the associated packet is resent.

3.4.2.2 Recieved correct packet

In case the gotoFlag isn't high, it evaluates the data packet that has been received most recently. This is done by comparing the packet id number to the current counter value. If these two numbers match, it means that the last step of the transfer was successful. In this case the data of the received packet is decoded and output to the serial interface. Afterwards the packet counter is incremented, and the next packet is sent.

3.4.2.2 Missing packet

If the number of the most recently received packet doesn't match the value of the internal counter, it indicates that a packet has been lost somewhere in the communication system. In this case the end node sends a goto packet to the mid note, requesting to get a retransmission of the missing packet. When the mid note receives this goto packet, it will immediately broadcast it to both nodes, to make sure that the next message they each send out will be the missing one.

3.4.3 End of transmission

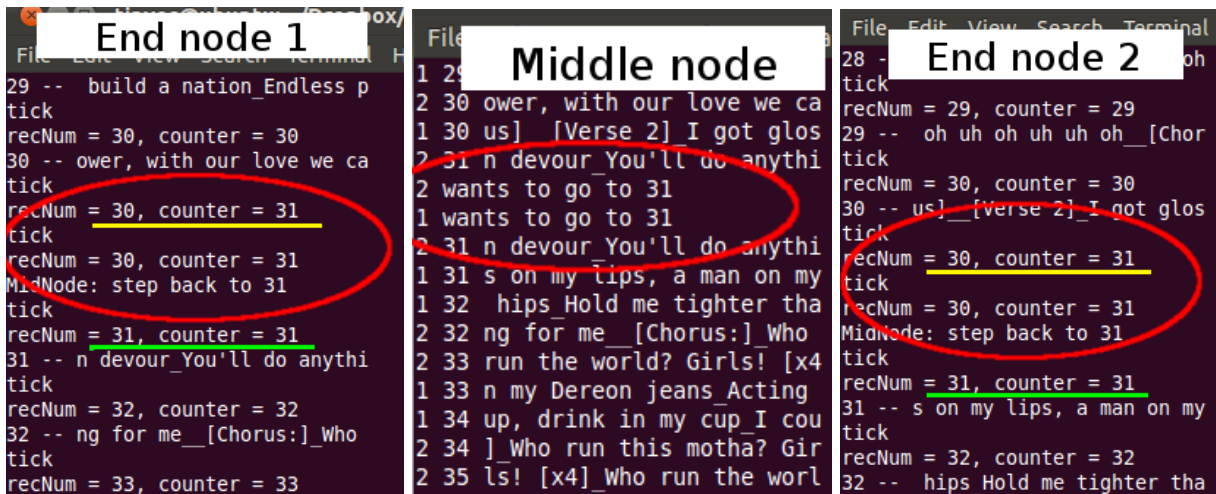
The programs are implemented to support transfers where the two files have different lengths. Since both file lengths are broadcast during the handshaking, all units know the individual file lengths. This is used in the end nodes to check whether they are supposed to send and receive at each step of the transfer. This is done by clearing their local send and receive flags when the associated limits is reached. When both the flags has being has been reached, the nodes goes out of transfer mode, and waits for a new handshake.

The mid node also needs to keep track of how many packets each node is sending. When one of the end nodes stops transmitting, it simply repeats the packets from the one that is still sending, without waiting for something to XOR it with. When it has repeated the last packet from both nodes it reinitializes and waits for a new handshake.

3.5 Test of the program

During development functionality is testing through blinking LEDs and debugging messages in the terminal. The motes shall be connected via USB to output to the terminal. With all three motes connected to terminals it is possible to follow the transmission and verify that retransmission occurs when needed and is performed as implemented.

The retransmission is tested by, provoking some errors. This is done by reducing the transmission power to -25 dBm and covering the antenna. **Figure** shows an example of a retransmission.



End node 1

```

29 -- build a nation_Endless p
tick
recNum = 30, counter = 30
30 -- ower, with our love we ca
tick
recNum = 30, counter = 31
tick
recNum = 30, counter = 31
MidNode: step back to 31
tick
recNum = 31, counter = 31
31 -- n devour_You'll do anythi
tick
recNum = 32, counter = 32
32 -- ng for me_[Chorus:]_who
tick
recNum = 33, counter = 33

```

Middle node

```

1 29 ower, with our love we ca
2 30 us] [Verse 2] I got glos
1 30 us] [Verse 2] I got glos
2 31 n devour_You'll do anythi
2 wants to go to 31
1 wants to go to 31
2 31 n devour_You'll do anythi
1 31 s on my lips, a man on my
1 32 hips_Hold me tighter tha
2 32 ng for me_[Chorus:]_who
2 33 run the world? Girls! [x4
1 33 n my Dereon jeans_Acting
1 34 up, drink in my cup_I cou
2 34 ]_Who run this motha? Gir
2 35 ls! [x4]_Who run the worl

```

End node 2

```

28 -- oh uh oh uh oh uh oh_[Chor
tick
recNum = 29, counter = 29
29 -- oh uh oh uh oh uh oh_[Chor
tick
recNum = 30, counter = 30
30 -- us] [Verse 2] I got glos
tick
recNum = 30, counter = 31
tick
recNum = 30, counter = 31
MidNode: step back to 31
tick
recNum = 31, counter = 31
31 -- s on my lips, a man on my
tick
recNum = 32, counter = 32
32 -- hips_Hold me tighter tha

```

As it is seen in the figure, end node 1 and end node 2 receives number 30 (recNum = 30) and its own counter is also 30 (counter = 30), then packet 30 is shown in the terminal window. Packet number 31 is expected, but the received number is still 30, as shown in the red ring. The middle node is requested to transmit packet 31 (see figure XXX - middle node terminal) again, whereupon the transmission is on track again (see figure end node 1 and 2). This shows that the retransmission works as expected.

4. Results and discussion

As a result of the implementation of the XOR packet solution the application is able to exchange two files from two different nodes, who are out of reach from each other, via a third node. The solution is able to retransmit when packet loss occurs (see figure XXXa, b and c). Even though the size of the files differs the transmission continues until the largest file is transmitted, see figure XXXb where the transmission continues after node 1 has received the whole file.



End node 1	End node 2	Middle node
<pre> 4 -- _motha? Girls! [x4] who r 5 -- un the world? Girls! [x4] 6 -- [Verse 1:] Some of them 7 -- men think they freak thi 8 -- s like we do But no they 9 -- don't Make your check com 10 -- e at they neck, Disrespec 11 -- t us no they won't Boy d 12 -- on't even try to touch th 13 -- is Boy this beat is crazy 14 -- This is how they made me 15 -- Houston Texas baby This 16 -- goes out to all my girls_ 17 -- That's in the club rockin 18 -- g the latest Who will buy 19 -- it for themselves and ge 20 -- t more money later I thin 21 -- k I need a barber None of 22 -- these niggas can fade me 23 -- I'm so good with this, I MidNode: step to 24 24 -- remind you I'm so hood w 25 -- ith this Boy I'm just pla 26 -- ying Come here baby Hope 27 -- you still like me F- you 28 -- pay me My persuasion can 29 -- build a nation Endless p 30 -- ower, with our love we ca 31 -- n devour You'll do anythi 32 -- ng for me [Chorus:] Who 33 -- run the world? Girls! [x4] 34 --] Who run this motha? Gir 35 -- ls! [x4] Who run the worl 36 -- d? Girls! [x4] [Verse 2:] 37 --] It's hot up in here DJ 38 -- don't be scared to run th 39 -- is, run this back I'm _S Reception Completed Transmission Completed </pre>	<pre> 6 -- nus up [verse 1] up in th 7 -- e club, we just broke up_ 8 -- I'm doing my own little t 9 -- hing You decided to dip b 10 -- ut now you wanna trip Cau 11 -- se another brother notice 12 -- d me I'm up on him, he up 13 -- on me don't pay him any 14 -- attention Cause I cried m 15 -- y tears, for three good y 16 -- ears Ya can't be mad at m 17 -- e [Chorus:] Cause if you 18 -- liked it then you should 19 -- have put a ring on it If 20 -- you liked it then you sh 21 -- ould've put a ring on it_ 22 -- Don't be mad once you see MidNode: step to 24 24 -- that he want it If you l 25 -- iked it then you should'v 26 -- e put a ring on it Wuh u 27 -- h oh uh uh oh oh uh oh uh 28 -- uh oh Wuh uh oh uh oh 29 -- oh uh oh uh uh oh [Chor 30 -- us] [Verse 2] I got glos 31 -- s on my lips, a man on my 32 -- hips Hold me tighter tha 33 -- n my Dereon jeans Acting 34 -- up, drink in my cup I cou 35 -- ld care less what you thi 36 -- nk I need no permission, 37 -- did I mention Don't pay h 38 -- im any attention Cause yo 39 -- u had your turn And now y Transmission Completed 40 -- ou gonna learn What it re 41 -- ally feels like to miss m 42 -- e [Chorus] Wuh uh oh uh </pre>	<pre> 1 8 2 8 1 9 I'm doing my own little t 2 9 don't Make your check com 1 10 hing You decided to dip b 2 10 e at they neck, Disrespec 1 11 ut now you wanna trip Cau 2 11 t us no they won't Boy d 1 12 se another brother notice 2 12 on't even try to touch th 1 13 d me I'm up on him, he up 2 13 is Boy this beat is crazy 1 14 on me don't pay him any 2 14 This is how they made me 1 15 attention Cause I cried m 2 15 Houston Texas baby This 1 16 y tears, for three good y 2 16 goes out to all my girls_ 1 17 ears Ya can't be mad at m 2 17 That's in the club rockin 1 18 e [Chorus:] Cause if you 2 18 g the latest Who will buy 1 19 liked it then you should 2 19 it for themselves and ge 1 20 have put a ring on it If 2 20 t more money later I thin 1 21 you liked it then you sh 2 21 k I need a barber None of 1 22 ould've put a ring on it_ 2 22 these niggas can fade me 1 23 Don't be mad once you see 2 23 I'm so good with this, I 1 24 remind you I'm so hood w 2 24 wants to go to 24 1 24 that he want it If you l 2 24 remind you I'm so hood w 1 25 iked it then you should'v 2 25 ith this Boy I'm just pla </pre>

The XOR packet solution consists of three transmissions system wide, while the naive way needs four transmissions, since the middle node must broadcast packets from end node 1 and 2 after each other. The execution time for a radio transmission of one packet is longer than to do the XOR operation on one packet. This means the overall execution time for the XOR implementation is shorter than the naive implementation. Furthermore, the reduced number of transmissions could result in a higher effective bandwidth, which can also potentially bring energy savings, because the motes will be able to finish the transmissions faster, and then get back to sleep.

To confirm that the theory holds in practice, an experiment has been carried out using a 10Ω shunt resistor in series with the batteries used to power the mote. By looking at the voltage drop across this resistor, the instantaneous power consumption can be calculated. Figure shows a

screen shot from the oscilloscope with arrows superimposed to show what happens at different times in the sequence shown. The sequence is generated with the test program TestApp, which was developed to test this.

[INDSÆT BILLEDET scope_6_arrows.png]

figurtekst: A screen shot from the oscilloscope showing the energy consumption of the mote waking from sleep and sending a packet. The timebase is 10 ms/division, and the scale is 50mV/division.

In detail, the sequence shown is:

- At the baseline, the CPU is idling. The only code running is TinyOS itself, as the program is waiting for a timer interrupt.
- Then the interrupt comes, and the MCU does a few calculations to determine the status of some flags and so on.
- After some time the radio comes on. Note that before this time, the radio was completely off. If the radio was just asleep, the idle consumption might be slightly higher, and the radio wake-up time would probably be shorter. The radio coming on leads to an increase in voltage drop over the shunt of 181,25mV.
- Then a package is sent out. The package is of the type radio_data_msg, so it is consistent with the type used in the implementation. This leads to a further increase in voltage drop of 17,5mV. The transmission power is at maximum in the test program.
- At the end it seems like the radio changes from sending state to receiver state. This leads to a very small, decrease in power consumption. This seems to be insignificant.

The power consumption calculated from the voltage measurements can be seen below:

$$R_{\text{shunt}} := 10\Omega$$

$$V_{\text{batt}} := 2.77\text{V}$$

$$V_{\text{idle}} := 18.125\text{mV} \quad I_{\text{idle}} := \frac{V_{\text{idle}}}{R_{\text{shunt}}} = 1.812\text{mA} \quad P_{\text{idle}} := I_{\text{idle}} \cdot V_{\text{batt}} = 5.021\text{mW}$$

$$V_{\text{receive}} := 199.375\text{mV} \quad I_{\text{receive}} := \frac{V_{\text{receive}}}{R_{\text{shunt}}} = 19.938\text{mA} \quad P_{\text{receive}} := I_{\text{receive}} \cdot V_{\text{batt}} = 55.227\text{mW}$$

$$V_{\text{send}} := 216.875\text{mV} \quad I_{\text{send}} := \frac{V_{\text{send}}}{R_{\text{shunt}}} = 21.688\text{mA} \quad P_{\text{send}} := I_{\text{send}} \cdot V_{\text{batt}} = 60.074\text{mW}$$

The data shows that there is a ten-fold increase in power once the radio is powered up. This

clearly means that the longer the radio can be kept asleep, the better. This clearly indicates that the radio is the biggest power consumer in the system. Since the xor-method halves the number of transmissions from the mid node, this is at the same time a good indicator that the idea behind the project is sound.

On the other hand, it is also clear that there is not a big difference between sending and listening on the radio. If the mid node has to listen very often, the energy consumption gains will not be as great as it seems, because the xor method does not decrease the number of packages that should be received. There should still be some savings, though, since computation is both faster and cheaper than transmitting.

Of course, sleeping the radio should be implemented, but as noted earlier the focus for this prototype has been reliable network communication.

Another thing that should be considered is how to deal with a node dropping out during an exchange. If an end node drops out in the current program, the other end node will enter a loop where it requests to step back to a certain package, and this will continue until the node reappears or the system is reset. If the middle node drops out, both end nodes will enter a similar loop. In a real system some protection should be added. This could be a maximum time without a new package from one node in the middle node, or a maximum number of retries to the same package in a row.

6. Conclusion

From the mini project we are able to conclude that the implementation of the XOR packets was successful and the preliminary assumptions about energy consumptions, to save energy through minimizing the number of transmissions, was right.

The difference between the naive and XOR implementation was simple and since the XOR operation is cheaper than transmission the overall energy consumption is lowered in spite of the implementation of the XOR operation on all three nodes.

Compared to the change in the implementation from the naive way to the XOR way, the XOR solution is highly recommendable for saving energy when a relay in a wireless sensor network is necessary.