CS262A Advanced Topics in Computer Systems
DOT3 Radio Stack

Jaein Jeong
Computer Science
UC Berkeley
jaein@cs.berkeley.edu

Sukun Kim
Computer Science
UC Berkeley
binetude@cs.berkeley.edu

ABSTRACT

Our network stack is implemented on a new platform of wireless sensor DOT3 which has better coverage and reliability than the current generation of wireless sensor, MICA. In outdoor tests, the packet receiving rate was close to 100% within 800ft and was reasonably good up to 1100 ft. This was made possible by using an error correction code and a reliable transport layer. Our implementation also allows us to choose a frequency among multiple channels. Using multiple frequency as well as reliable transport layer we could achieve high packet receiving rate by paying additional retransmission time when collision was increased with more number of sensor nodes. Being written in nesC programming language, our network stack is compatible with the latest generation of TinyOS code.

Keywords

TinyOS, nesC, radio stack and reliable communication.

1. INTRODUCTION

What is a wireless sensor? A wireless sensor is a tiny computer node that can sample analog signals and communicate with other nodes in wireless, especially in radio. Since the sampled data in the sensors is not useful in itself, it needs to be transferred to the host machine for analysis. NEST project in UC Berkeley is such an effort to make wireless sensors and apply them in a number of fields. The wireless sensors have evolved several generations and we are using MICA platform which can communicate with other nodes of the same platform with its on-board radio chip. With this multi-channel ADC and radio capability, MICA has served as a reasonable research tool for understanding wireless sensors and a number of practical applications on them. However, MICA was not enough for any large scale applications due to its short range and rather unreliable radio communication as shown in Figure 1.

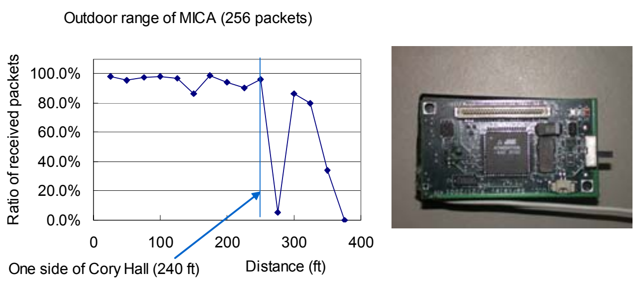


Figure 1. A MICA node and its outdoor range

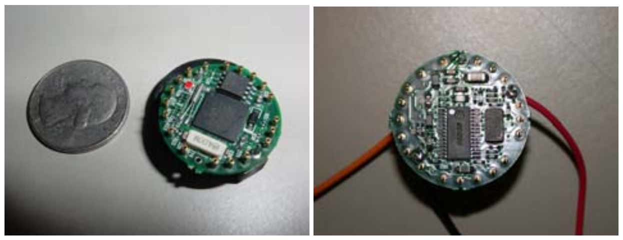


Figure 2(a). A DOT3 node and CC1000 radio chip (right)

Figure 2(b). A MICA2 node

As it is mentioned in the beginning, the data in each sensor needs to be forwarded to the host machine for persistent storage and further analysis. In an application that covers large area the reliable radio communication is more important because many of wireless sensors cannot send packets directly to the host machine and have to rely on intermediate nodes to pass the packets. We expect that wireless sensors of improved radio capability will allow us to build large area applications with small number of sensor nodes and host machines.

DOT3, shown in Figure 2(a), is a new hardware platform made with CC1000 radio chip. Since DOT3 is optimized for form factor (a quarter size compared to MICA of two AA battery size), some functions in MICA are missing. To name a few, accurate low frequency crystal for UART communication and power amplifier for longer battery lifetime. Thus, a variation of DOT3 having all the functions in MICA was made and this is called MICA2 platform (Figure 2(b)). However, MICA2 is a bit bigger than MICA platform. MICA2 platform is used to interface DOT3 to the host machine with its accurate UART clock and DOT3 is used for mobile applications with its small form factor. Since DOT3 and MICA2 has the same radio chip CC1000, they share the same network stack code. Our work is to implement a radio stack on the new hardware platform and evaluate its performance.

2. Background and Related Work

2.1 TinyOS and its programming model

The wireless sensors used in Berkeley are based on a brand of embedded processor ATMega 103L[1]. It comes with basic development tool called AVRGCC that supports plain C programming language and some library functions for hardware access. Tiny OS is the software that runs on top of AVRGCC providing modular programming interface and useful standard services. Application programmer can easily develop a wireless sensor application using these components.

In TinyOS[2], components are defined using modules. A module can have states using its member variable and communicate with other modules. The communication between two modules is bidirectional: module A can access methods of module B directly by calling the methods of B and module B can notify module A of any changes in its state by signaling events such as clock ticks and packet arrivals.

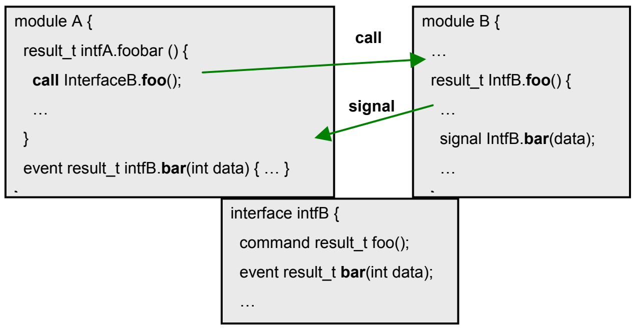
A mechanism is needed to figure out which methods are called and which methods are triggered by events. In TinyOS, this is defined in the interface which is separate from an implementation. With an interface, we can change the component just by changing the name of an implementation linked to the interface. Figure 3 shows the communication between two modules A and B. Module A calls a method foo( ) in module B by

```
call intB.foo( )
```

and module B notifies any events to the module A's event handler by

```
signal intfB.bar( )
```

Module A has event handler bar( ) and it is invoked whenever B signals an event.



```
module A {
  result_t intfA.foobar () {
    call InterfaceB.foo();
    …
  }
  event result_t intfB.bar(int data) { … }

module B {
  …
  result_t IntfB.foo() {
    …
    signal IntfB.bar(data);
    …

interface intfB {
  command result_t foo();
  event result_t bar(int data);
  …
```

Figure 3. Communication in TinyOS modules

[1] MICA platform uses ATMega 103L and DOT3 and MICA2 use ATMega 128L processor. Since ATMega 128L is backward compatible to ATMega 103L, we use ATMega 103L as a representative.

[2] We are following the programming model and the convention in TinyOS version 1.0. nesC is the programming language used in TinyOS since version 1.0.

2.2 Network stack in TinyOS

Like many other network protocols, Tiny OS provides communication to applications using multiple layers. Figure 4 shows the network stack of MICA.

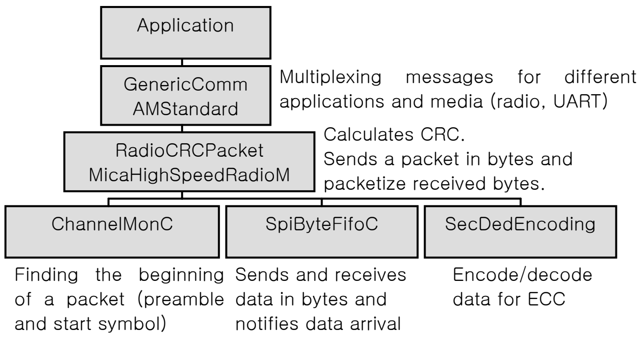

Application
GenericComm AMStandard
Multiplexing messages for different applications and media (radio, UART)
RadioCRCPacket MicaHighSpeedRadioM
Calculates CRC. Sends a packet in bytes and packetize received bytes.
ChannelMonC
SpiByteFifoC
SecDedEncoding
Finding the beginning of a packet (preamble and start symbol)
Sends and receives data in bytes and notifies data arrival
Encode/decode data for ECC

Figure 4. MICA Network Stack

An application sees the radio as a service through which it can send and receive data in fixed sized packet level. Right below the application layer, GenericComm and AMStandard components multiplex messages for different applications using Active Message number (AM number) , which is a kind of port number. The sender specifies the AM number when it sends a packet. On the receiver side, the event is notified to the processes which are registered to that AM number. In addition, AMStandard multiplex messages to different media (radio and UART) using the destination address and provides the common interface. We can think of GenericComm and AMStandard as a transport layer which supports best effort delivery and as a process to process communication.

Compared to a PC or a workstation which is connected to global network such as Internet, wireless sensors operate within a local domain with its locally unique address. Thus, TinyOS network stack doesn't have network layer which route packets across different local networks.

Most of TinyOS network stack is related to the link layer. Since the underlying radio chip has byte level interface to the microprocessor, a packet needs to be decomposed before they are sent on one end and a packet needs to be reconstructed from the received bytes on the other end. This is implemented in MicaHighSpeedRadioM. After that, each byte is sent or received through serial peripheral interface (SPI) which gives byte level abstraction over serial links. Since data is sent as bytes, the receiver side need to find the start of a packet. This is done by sending a specific sequence of bytes which is different from normal data byte patterns. This specific sequence of bytes are called preamble and start symbol. The receiver assumes the beginning of a packet when input bytes match preamble and start symbol. Optionally, the data bytes can be encoded with an error correction code before they are sent over the radio for integrity. In TinyOS, a single error correction and double error detection (SECDED) code is used as an error correction code.

Finally, the radio chip needs to be initialized with correct parameters and this is highly dependent on the underlying radio chip.

2.3 Earlier works in DOT3

Our project is based on some of the earlier works. Jason Hill wrote a preliminary version of network stack for DOT3 in TinyOS v0.6 style. This version of network stack supports AM number multiplexing and packet framing.

CC1000 can operate in three different bands of frequencies: 433MHz, 866MHz and 900MHz ranges. Each band requires different values for external components (capacitors and inductors used in filter and resonating circuit) and initialization parameters. Since the values for these external components are determined by physical components, a DOT3 node can operate in only one band once its external component values are fixed. The decision which band to use is determined by the coverage and the number of legally usable channels. 900MHz range is preferable for its relatively large selection of channels and 433MHz is better for its longer range it covers.

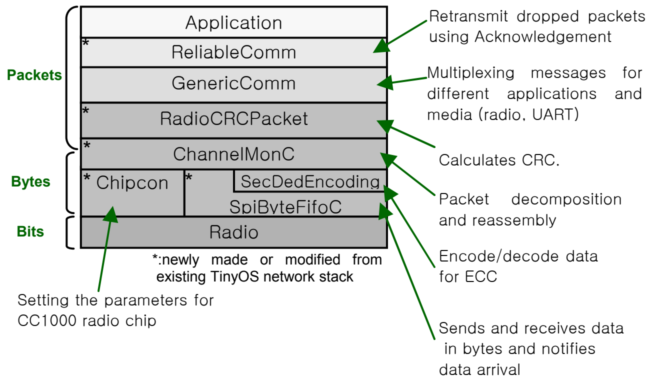
Jason Hill initially wrote his network stack in 900 MHz range and Crossbow technology modified it to support 433MHz range.

3. Design of Radio Stack

From the earlier works by Jason Hill and Crossbow technology, we found some chance of improvement:

- We need to write the network stack in nesC programming language, which is compatible with the current generation of TinyOS.
- We need to add error correction code like single error correction and double error detection (SECDED) to protect data from transient errors.
- We need to utilize multiple channels of CC1000 radio chip.
- We can support reliable communication.

Starting from the existing network stack, we implemented a network stack for DOT3 as shown in Figure 5:



Application
* ReliableComm
GenericComm
* RadioCRCPacket
* ChannelMonC
* Chipcon
* SecDedEncoding
SpiByteFifoC
Radio
Packets
Bytes
Bits
Retransmit dropped packets using Acknowledgement
Multiplexing messages for different applications and media (radio, UART)
Calculates CRC.
Packet decomposition and reassembly
Encode/decode data for ECC
Sends and receives data in bytes and notifies data arrival
Setting the parameters for CC1000 radio chip
*:newly made or modified from existing TinyOS network stack

Figure 5. DOT3 Network Stack

First, we need to look at the data flow of our network stack. While data is transferred in packets in application layer, underlying radio only provides byte level and serial interface. Section 3.1 explains how we can send and receive data in bytes to the radio chip and how we can configure the chip.

Then, a packet needs to be decomposed into bytes before being sent and a new packet needs to be reconstructed from incoming bytes. Section 3.2 explains the steps for packet framing.

Once we have data in packets, we can use CRC to find any errors in packet level. Before sending a packet, RadioCRCPacket calculates CRC for the data bytes except the last two bytes used for CRC. Then, it appends CRC bytes after the data bytes. When it receives a packet, it calculates CRC for the data bytes. It relays the packet to the upper layer only when the calculated CRC matches the CRC value appended at the end of the packet.

In the existing network stack implementations, a packet was sent to each application after GenericComm module multiplexes a packet according to the application specific identifier called AM ID (Active Message ID). We implemented another layer ReliableComm on top of GenericComm for reliable communication and this is explained in section 3.3.

3.1 Interface to the radio chip.

3.1.1 ATMega 103L external device interface

ATMega 103L microprocessor interfaces to external devices using data ports, and the radio chip is considered as one of them. Each data port of the microprocessor is 8-bit wide and can be read or written in bytes (can be written only when the device is output only). AVRGCC supports a group of library functions that access data ports. Since the notations of these functions are not easy to understand, TinyOS provides macros to assign mnemonics to the function calls. And this is listed in Table1.

Table 1. List of functions that access external devices

AVRGCC function	Description
sbi(port, bit)	Sets the bit for the pin as '1' and `TOSH_SET_***_PIN()` is the TinyOS macro.
cbi(port,bit)	Clears the bit for the pin as '0' and `TOSH_CLR_***_PIN()` is the TinyOS macro.
outp(byte,reg)	Writes a byte to the register.
inp(reg)	Reads a byte from the register.
outp(byte,data direction reg)	Determines the direction of data port pins. For each pin, '1' sets the direction as output and '0' sets it as input. `TOSH_MAKE_***_OUTPUT` and `TOSH_MAKE_***_INPUT` are the TinyOS macros.

3.1.2 Data interface

Even though data can be transferred using bits, its programming interface becomes rather complicated because the data needs to converted between a byte and bits using bitwise operation.

ATMega 103L processor gives a byte level interface, which is called serial peripheral interface (SPI), to transfer data to external devices and this is shown in Figure 6.

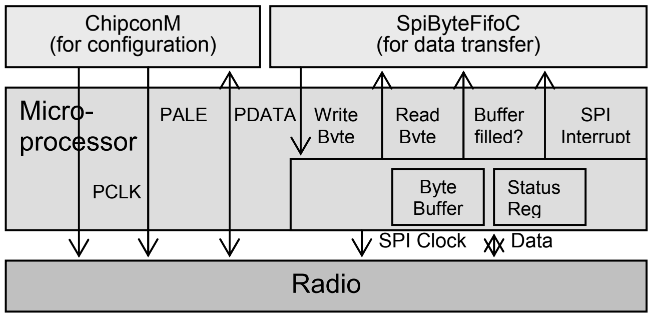


Figure 6. Interfacing microprocessor to the radio

SPI has a byte buffer (SPDR: SPI data register) and an outgoing data byte waits here until all the bits are sent. And SPI assembles incoming bits into a byte in the buffer. Since there is only a single buffer, either a send or a receive can be done at a time. SPI can be checked whether it has an incoming byte or not with its status register (SPSR): the most significant bit of SPI becomes high when there is an incoming byte in the buffer and it becomes low otherwise. SPI can be switched between send and receive mode by changing data direction of data pin (Bit-3 of port B). Finally, data should be read or written at the same rate with that of external device. CC1000 radio chip is synchronized to the microprocessor with the SPI clock and the clock triggers an interrupt at regular interrupt. This is done by connecting SPI clock output pin of the microprocessor (Bit-1 of port B) to the data clock pin of the radio chip. The SPI clock interrupt is triggered while the interrupt is enabled and stops when the interrupt is disabled. HPLSpiC is a module that gives easy to understand programming interface. Table 2 summarizes this subsection.

Table 2. CC1000 data interface to ATMega 103L processor

ATMega 103L	Description	CC 1000
SPSR	Most significant bit (bit-7) is '1' if there is incoming byte, '0' otherwise **HPLSpi methods**: `is_empty()`	
SPDR	Write: send a byte to CC1000 Read: read a byte from CC1000 **HPLSpi methods**: `write_byte()` and `read_byte()`	
SPCR	Enables SPI clock interrupt (write 0xC0) or disables it (write 0x40). **HPLSpi methods**: `enable_intr()` and `disable_intr()`	
Port B bit-3	Switches transmit/receive mode **HPLSpi methods**: `txmode()` and `rxmode()`	**DIO**
Port B bit-1	Connects SPI clock to the data clock of CC1000	**DCLK**

3.1.3 Serial configuration interface

The microprocessor still needs to communicate with the radio chip to configure or monitor the status of it. The radio chip needs to be configured when it starts to operate or it needs to change its property, such as changing frequency (explained in the next section) and power consumption level. CC1000 exposes three pins (PALE, PCLK and PDATA) for this purposes.

Table 3(a). CC1000 configuration interface to ATMega 103L

ATMega 103L	TinyOS macros	CC 1000

Port D bit-5	TOSH_SET_POT_SELECT_PIN( )
	TOSH_CLR_POT_SELECT_PIN( )
		PALE

Port D bit-6	TOSH_SET_RFM_CTL1_PIN( )
	TOSH_CLR_RFM_CTL1_PIN( )
		PCLK

Port D bit-7	TOSH_SET_RFM_CTL0_PIN( )
	TOSH_CLR_RFM_CTL0_PIN( )
	TOSH_READ_RFM_CTL0_PIN( )
		PDATA

These pins are mapped to data port pins (port D bit 5,6 and 7). By setting or clearing these pins, the microprocessor can send a sequence of bits (control register address, a byte data and a bit representing whether the operation is write or read) as it is shown in the following figure:

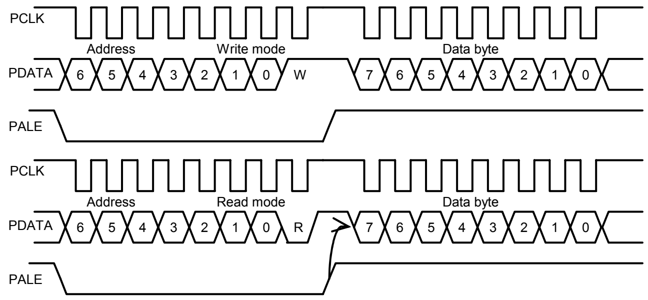


Figure 7. Steps to write or read CC1000 control registers

This is implemented as `init()`, `write()` and `read()` in HPLChipconC module.

3.1.4 Using multiple channels

Whereas TR1000 radio chip in MICA operates at a fixed frequency of 916.5 MHz, CC1000 can be set up to operate at one of the three different bands (433 MHz, 868 MHz and 900 MHz). The band is determined by selecting capacitor and inductor values for the resonator and the filter in the wireless sensor board.

Within each band, we can select one channel at run time among a number of frequencies. The purpose of selecting channels is to reduce any interference from neighboring sensor nodes or other wireless devices. The 900 MHz band is preferable for its wider range frequencies that can be selected where as 433 MHz has longer range due to its longer wavelength. We chose 433 MHz band in favor of longer range.

For the CC1000 radio chip to operate at a specific frequency, it needs to be configured with the correct frequency words and clock divisor byte. CC1000 transmits and receives at different frequency and these frequencies are represented by two 24-bit frequency words. These frequencies are generated by dividing the frequency synthesizing clock (we are using 14.7456 MHz) with the clock divisor byte. These values are set up in ChipconC module. A recommended values are listed in [3], but none of them worked for 433MHz band. We found 4 working channels by measuring signal strength for different values within 433 MHz and 435 MHz. Here are the channels we found:

Table 4. Channels available for DOT3 in 433MHz band

		CH 1	CH 2	CH 3	CH 4	MICA
TX	Frequency (MHz)	433.02	433.64	434.20	434.71	916.50
	CC1000 reg 4-6	57f785	581785	583785	585785	-
RX	Frequency (MHz)	433.09	433.71	434.27	434.78	916.50
	CC1000 reg 1-3	580000	582000	584000	586000	-
CC1000 reg 12 Divisor (PLL)		60	60	60	60	-
Output Power (dBm)		-45	-45	-47	-47	-52

Figures 8 (a) and (b) show the waveforms in Spectrum analyzer when the configuration is correct and wrong. In Figure 8(a), all the external components such as resonator and filter are set to 433MHz band and the control register of CC1000 is correctly configured. The waveform has the peak around 433MHz and its peak output power had around -45dBm using inducting antenna in Spectrum analyzer input. In Figure 8(b), control registers are set to 433MHz, but the inductor in the resonator is set to the value used in 900MHz band. Since this resonator value doesn't match the other external components and the configuration value, it results in the peak somewhere middle between 433MHz and 900MHz and its output power is much weaker that it should be. Actually, initial build of 433MHz DOT3 nodes had this bug and it was one of difficulties in early stage of our project.

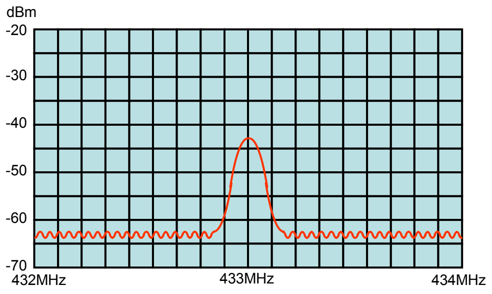


Figure 8(a). Waveform when configuration is correct.

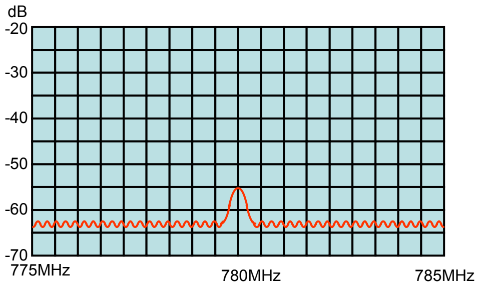


Figure 8(b). Waveform when resonator value is misconfigured.

3.2 Packet decomposition and reassembly

Packet decomposition and reassembly is done in ChannelMonC with the help of SpiByteFifoC module. SPI is synchronized to the microprocessor with the clock and generates an interrupt at regular interval.

At each interrupt invocation, the interrupt handler SIG_SPI in SpiByteFifoC module is called. Then, we can determine we can send or receive a byte by looking at the control register (SPSR) as it is shown in Figure 9:

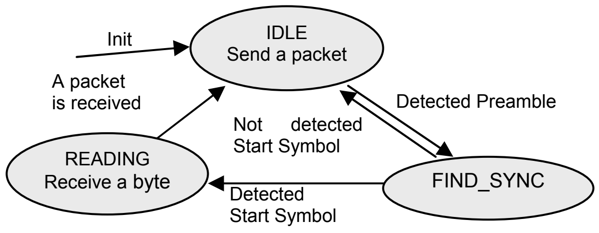


Figure 9. State Transition diagram for packet decomposition and reassembly

Initially, the state is in IDLE state. If no incoming bytes are available, ChannelMonC sends a packet. Since radio chip transfers data in bytes, we need to tell the beginning and the end of a packet. This is done by having a special sequence of bytes (preamble and start symbol) in the beginning and the fixed number of bytes after that. After sending preamble and start symbol, ChannelMonC sends data bytes. Data bytes can be sent as they are or can be sent after being encoded with error correction code for integrity. We used SecDedEncoding TinyOS module which implements a single-error-correction-and-double-error-detection (SECDEC) code. The version of ChannelMonC with error correction code is ChannelMonEccC.

When there is an incoming byte, ChannelMonC reads the byte and see whether the sequence of bytes received up to now matches the preamble. Then, it goes to FIND_SYNC state. If the next incoming bytes match the start symbol, it goes to READING state. After reading the fixed length of data (36 bytes is default), ChannelMonC notifies the arrival of a packet to RFCommM module.

3.2.1 Error correction using a SECDED code

SecDedEncoding module takes 1 byte data and generates 3 byte output. Since preamble and start symbol are 7 bytes long and the data in a packet is 36 bytes long, a packet sent using SecDedEncoding has 31% of utilization (= $\frac{7+36}{7+3\times36}$). 3 bytes is not the optimal size when we encode 1 byte.

According to even-odd code explained in [1], when we have d-data bits and r-parity bits, r should meet the following inequality.

$$\sum_{1\le 2i-1\le r}^{r} \binom{r}{2i-1} \ge d + r$$

For d = 8, r = 5 is the smallest number that meats the inequality.

$$\binom{5}{1}+\binom{5}{3}+\binom{5}{5}=16\ge 8+5=13$$

Thus, we outputs 13-bits for 1-byte input. The reason why existing SECDED implementation used more bits (24-bits) is they tried to balance the number of 0's and 1's in a bit sequence (bit stuffing).

According to [7], 0 or 1 is detected by comparing received bytes with the average voltage level up to now. A long sequence of 0's or 1's changes the average voltage level from the center and detection of 0 and 1 may not be correct. This bit stuffing is needed in NRZ encoding which sends high for 1 and low for 0. CC1000 provides not just NRZ encoding but also Manchester encoding. Manchester encoding avoids long sequences of 0's or 1's by sending low-to-high or high-to-low. However, we haven't yet found the economical error correction code that can be used in Manchester encoding.

3.3 Reliable Transport Layer

We wanted reliable communication. But we also wanted compatibility and ease of use. So we designed to give the same interface as that of existing best-effort transport layer. As shown in Figure 10, we designed reliable transport layer so that it can be inserted between best-effort transport layer and application layer without any significant modification of application.

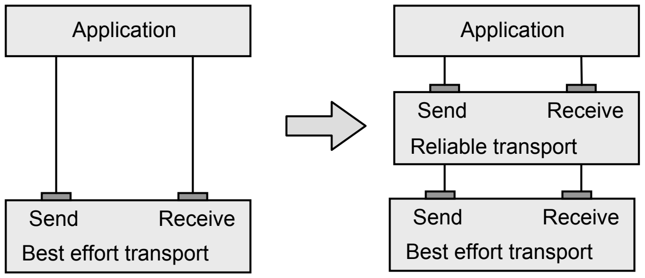


Figure 10. Compatible interface of reliable transport layer

We also wanted light-weight layer. Compatible and light-weight approach made us to implement connection-less communication. Interface of existing best-effort transport layer supports only connection-less communication. In the reliable transport layer, Connection information is managed globally.

To guarantee reliable communication, we mainly used acknowledgement and retransmission. Packet structure came to incorporate more information. Sender and receiver use this information and react properly according to it.

3.3.1 Reliable Message

For reliable communication, additional meta-data (source address, acknowledgement number) needs to be included in each packet. The packet size is 36 bytes. 7 bytes are already used by lower layers for meta-data. And 29 bytes are used as date field. 4 more bytes (2 for source address, 2 for acknowledgement) are taken from data field for meta-data in the reliable transport layer. Now the length of data field decreased from 29 bytes to 25 bytes (13.8 percent of loss). Sender gets a usual packet from upper application layer. And it realigns data, adds source address and acknowledgement number, and sends it to lower best-effort transport layer. Receiver also does the same conversion. Packet structure is shown in Figure 11. The use of additional meta-data is transparent to applications except the decreased size of data field.

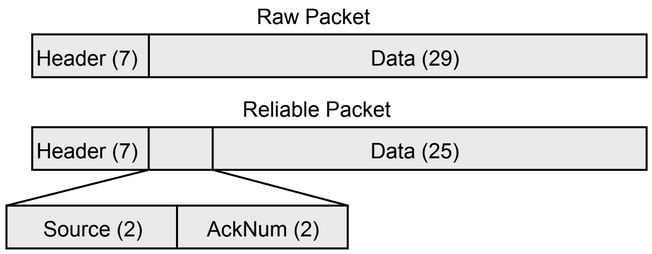


Figure 11. Packet structure for reliable transport layer

3.3.2 Sender

Sender is a finite state machine. When sender gets a packet from upper application layer, it adds source address and acknowledgement number as shown in the Figure 11, realigns data, and passes the packet to the lower best-effort transport layer.

If the sender receives acknowledgement from receiver, it reports success to the upper application layer. If it does not receive acknowledgement until time out, it retransmits the unacknowledged packet. The amount of waiting time is a random number between T and 2T. After N successive time-outs, sender reports failure to the upper application layer.

For simplicity, sender uses block-and-wait strategy. Sender only needs to remember current receiver's information. Figure 12 shows main part of state diagram of the sender.

Since there is no queue in the lower layer, if sender tries to send a packet while the receiver of the same node is also replying by sending acknowledgement, the packet of sender can be lost. So buffer of size 1 is used for sender. When acknowledgement is under process, sender saves the packet in buffer and transmits after the receiver of that sensor completes acknowledgement.

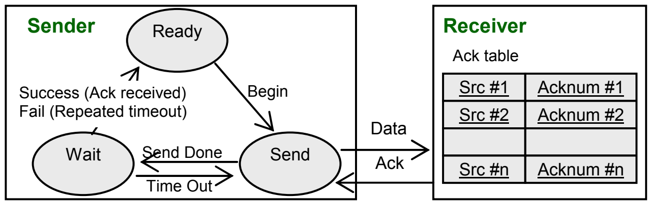


Figure 12. Schematic of reliable transport

3.3.3 *Receiver*

Receiver is also a finite state machine. When receiver gets packet from the lower best-effort transport layer, it looks at source address and acknowledgement number. If it is a new packet, it sends acknowledgement and passes the packet to the upper application layer. If it is a packet already received, it only sends acknowledgement to the sender.

To decide whether the received packet is a new packet or an already received one, it keeps connection information in the 'Ack table'. The table has a pair of sour address and acknowledgement number as an entry. In case of lost acknowledgement, as in Figure 13, duplicate data packets can come. Then we should not report the second packet, and we need table for this purpose.

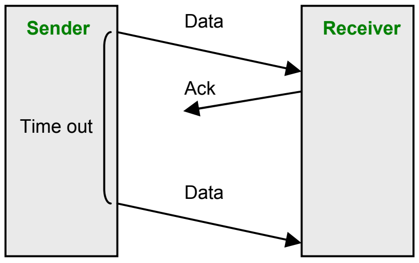


Figure 13. Lost acknowledgement

Since the size of table is limited, it can not handle arbitrary number of connection all the time. So FIFO algorithm is used to replace entry in the table. To reduce table lookup time, reverse chronological search is used. It looks up most recent connection first, and then next recent connection, and so on.

3.4 Installation

TinyOS keeps program files dependent on a specific platform under `$TOSROOT/tos/platform` directory. We placed the files related to DOT3 platform under `$TOSROOT/tos/platform/mica2dot` directory. In Current 1.0 version TinyOS, the path to the platform specific directories are scanned only when the nesC compiler is built unlike 0.6 version. Thus just copying platform specific files and modifying make files doesn't work. These are the steps to install DOT3 specific files including network stack:

- Modify `$TOSROOT/nesc/tools/ncc`, `$TOSROOT/nesc/tools/ncc.in` and `$TOSROOT/apps/Makerules` to handle DOT3 platform. We simply copied the statements for mica platform as mica2dot and changed the name `mica` to `mica2dot`.
- Copy files specific to DOT3 platform including our radio implementation under `$TOSROOT/tos/platform/mica2dot`
- Delete existing installation of nesC compiler and make install nesC compiler. This is done by

```
rm /usr/local/bin/ncc
cd $TOSROOT/nesc
make
su
make install
```

4. Evaluation

We set up two kinds of experiments to see the effectiveness of our network stack implementation. In outdoor experiments, the sender sends a number of packets and the receiver counts how many packets it received from the sender as we moves the sender far from the receiver. We did outdoor test in the middle of Berkeley campus as shown in Figure 14:

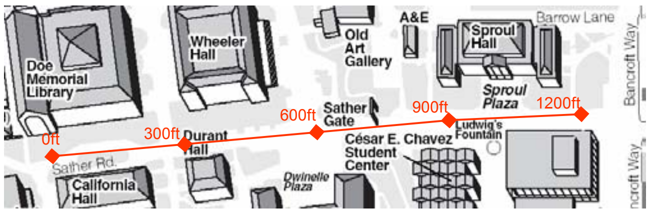


Figure 14. Locations of outdoor experiments

In the indoor experiments, we had senders and receivers send and receive packets in a short distance as we vary number of senders or number of channels used. We used the ratio of successfully received packets as an indicator of effectiveness of each transmission method.

4.1 Comparison with MICA

Since the immediate goal is to make DOT3 work as well as MICA, we set up an experiment to compare our DOT3 network stack implementation with that of MICA. We measured the packet receiving rate and the transmission time for MICA and DOT3 as we vary the distance between the sender and the receiver. Both implementation used SECDED non-retransmission scheme.

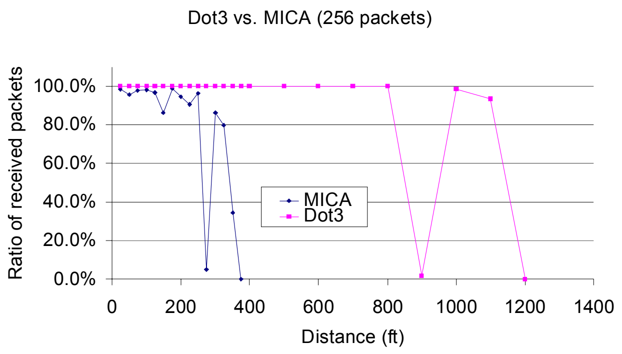


Figure 15. Ratio of received packets (DOT3 vs. MICA)

Due to the underlying radio chip, DOT3 network implementation produced better properties: close to 100% receiving rate up to 800 ft and more than three times the coverage (1200 ft vs. 350 ft). However, the transmission time of DOT3 was slower than that of MICA. Even though we consider the twice slower transfer rate of DOT3 (19Kbps vs. 40Kbps), our DOT3 radio implementation was more than twice slower (60sec vs. 9sec for 512 packets). This is because we used slower interrupt handler than that in MICA (SPI instead of timer interrupt).

4.2 Effects of error correction code

To see the effectiveness of using error correction code, we measured the packet receiving rate for the two implementations: the one with error correction code (SECDED) and without it.

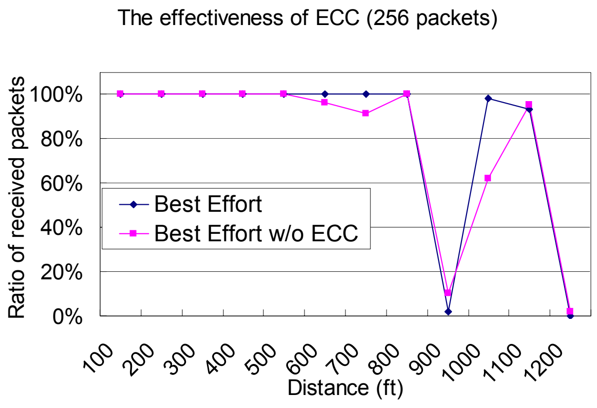


Figure 16. Ratio of received packets (ECC vs. non-ECC)

As it is shown in Figure 16, the packet receiving rate decreased as the sender moves away from the receiver, especially after 500 ft. Whereas the ECC implementation was more resilient to errors and had better packet receiving ratio up to its limit (1200 ft).

4.3 Effects of retransmission

To see the effectiveness of retransmission, we set up the two experiments. In the first experiment, we vary the distance between the sender and the receiver for different implementations: the implementation with no retransmission and the ones with 2,3 or 5 retransmissions. All implementations used SECDED for integrity. In the second experiment, we located multiple senders (1, 2 or 4) and a receiver closely to see the effects of retransmission when the collision rate is different. We measured the packet receiving rates and the transmission time for two extreme cases: no retransmission and 5 retransmissions.

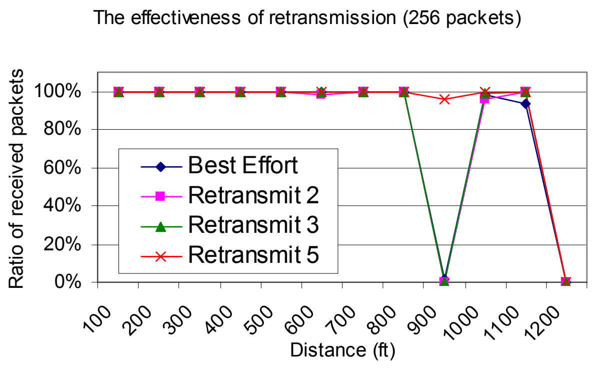


Figure 17. Ratio of received packets (Non-retransmission vs. Retransmission implementations)

In the first experiment, retransmission was slightly better than the best effort transmission. The difference between three retransmission schemes was not that noticeable except that retransmission 5 could receive the message all other methods failed at 900 ft. We find this was possible because radio waves from the sender took different paths somehow while the sender tries to retransmit the packets.

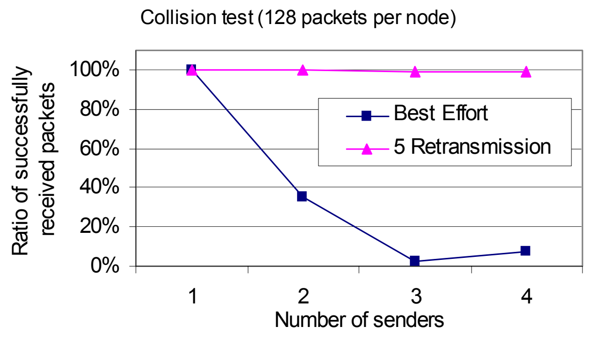


Figure 18(a). Ratio of received packets with multiple senders

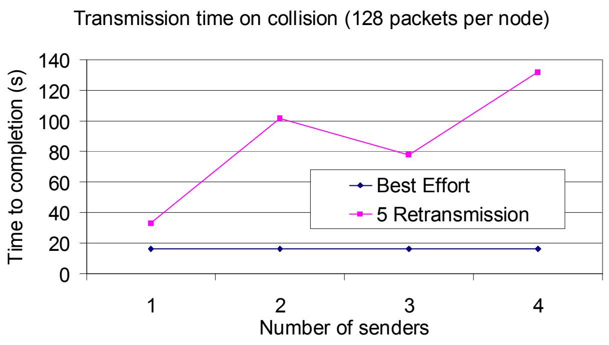


Figure 18(b). Time to transmit packets with multiple senders

The effects of retransmission in a closely populated area was very noticeable. It reduced most of the packet drops due to collision with increased transmission time. This shows that packets are very likely to be dropped when multiple nodes are sending packets in bursts and the packet drops can be avoided with retransmission.

As the rate of collision gets higher, retransmission takes more time, because current version of retransmission does not consider collision condition.

4.4 Effects of using multiple channels

In this experiment, we prepare 8 nodes, and divided them into 4 groups, each of which is composed of one sender and one receiver.

We measured the packet receiving rates and the transmission time for non retransmission implementation and 5 retransmission implementation with four senders. We varied the number channels used(1, 2 and 4) to see the effects of multiple channels for the two implementations.

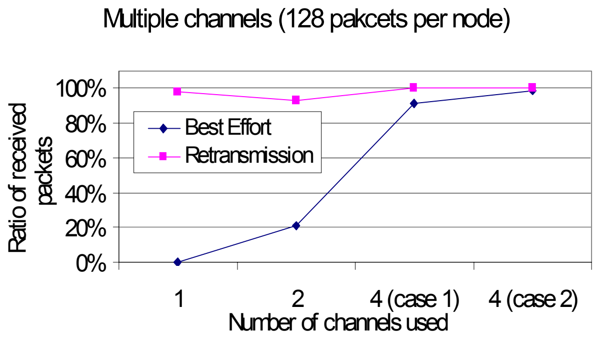


Figure 19(a). Ratio of received packets with multiple channels

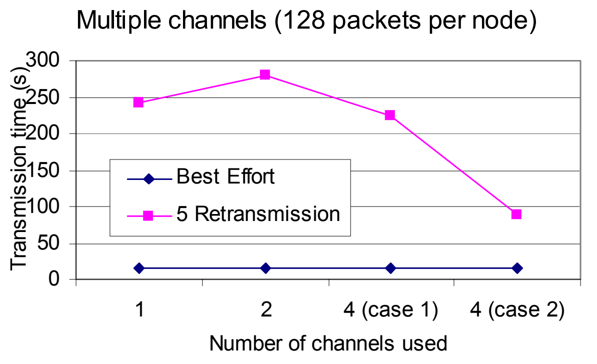


Figure 19(b). Time to transmit packets with multiple channels

The results show that using multiple channels was more effective to non-retransmission implementation than retransmission implementation. This is expected from the results of section 4.2 in that retransmission received most of the packets whereas non-retransmission got only 10% of the packets.

Using multiple channels also helped the retransmission time. It paid smaller amount of transmission time when more channels are available. When there are less channels available, it spent more for transmission and achieved still high packet receiving rate. This implies that retransmission and use of multiple channels can be beneficial for reliable packet delivery.

We can also infer that there is some interference among channels. Otherwise, for the case of 4 channels, ratio of received packets should be very close to 100% for best effort transport.

4.5 Overhead of reliable transport layer

To measure overhead of sender, we eliminated wait for acknowledgement in sender side. And for 512 packets, we measured completion time. The result is shown in Table 5.

Table 5. Time to send/receive 512 packets

Best Effort	Retransmission (5retransmission)	Retransmission (0 retransmission)
31 sec	64 sec	32 sec

The overhead is negligible for the sender.

To measure overhead of receiver, we made receiver send data to another sensor node. However, the two sensor node except the receiver also interfered each other. Unfortunately we could not get correct result. We surely expect some overhead for receiver side, because it should send a packet for each incoming packet while this is not needed in best effort transport.

In retransmission, every packet involves two transmissions. This explains the reason why retransmission takes about twice longer than best effort does.

4.6 Rayleigh fading and theoretical limit of range

If we look at the range test results in previous Figures, the graphs consistently had dips at 900 ft. Once the sender moves farther from that distance, the receiver received the packets from the sender again. This happened because radio signal is propagated through waves. Radio waves from the sender take paths while they travel and their phase can change when they reflect on some obstacles. Waves of opposite phase cancel each other and the resulting signal becomes weaker than the sensitivity of the receiving node, thus packets cannot be heard. This phenomenon is called Rayleigh fading and illustrated in Figure 20.

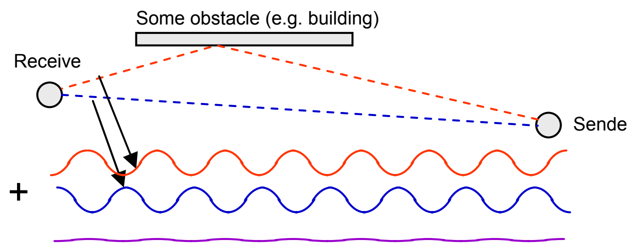


Figure 20. Rayleigh fading

More complicated devices like CDMA cellular phone use multiple antenna of different phase to avoid problem, but we cannot depend on this method because CC1000 has only single antenna. However, we can around this by having intermediate nodes between the two nodes and by having the intermediate nodes relay the packets.

5. Discussion and Conclusion

As it is shown in section 4.1, our network implementation had slower transmission time than MICA. We expect this will be cured by using timer interrupt which is faster than the one used.

Our reliable transmission scheme was effective in reducing packet losses, but the overhead was a bit high when there was much collision. This is because senders still try to resend unacknowledged packets after randomly chosen time within the timing window of fixed size. Even though waiting time varies within the timing window, it was not helpful when collision is high. We expect increasing timing window size like 'exponential back-off' will reduce the rate of bytes sent so that the overall system can make progress.

In reliable transport layer, sender's window size is 1 and this causes the sender block and wait. Increasing window size will reduce the waiting time, and improve transfer rate. Sender will need 'Ack table' and buffers for unacknowledged packets. The 'Ack table' in the sender is similar to the one in the receiver. And the receiver needs buffer to support in-order delivery.

The results in section 4.4 showed that using multiple channels was very effective for reducing collision when multiple senders burst packets. Currently, the channel is tuned with the identifier (group ID) which is given at compile time. Since channels are statically determined, performance can degenerate into that of single channel when they are misconfigured. We expect that dynamic frequency allocation like the frequency hopping in Bluetooth is needed for our implementation.

We used existing SecDedEncoding module in TinyOS for error correction code. For 1 byte data, SecDedEncoding generates 3 byte output, which is larger than the optimal value 13-bits. The use of Manchester encoding in CC1000 gives us to chance to transfer data with less bytes because 0-1 balancing is not needed.

We consider an application that utilizes the long range coverage of DOT3 radio in monitoring facilities in Microlab in UC Berkeley. Since the Microlab is heavily dependent on liquid nitrogen in many of silicon manufacturing processes, they monitor the status of nitrogen tanks such as nitrogen pressure and flow. The nitrogen tanks are outside Cory Hall and connected to the Microlab via wires (around 100 ft). These wires are not easy to maintain and make it hard to relocate Microlab facilities. We expect that some number of DOT3 wireless sensors can substitute these wires that run through the building.

6. ACKNOWLEDGMENTS

Crossbow Technology helped us in fixing the hardware bugs and configure the radio chip correctly. We also thank Jason Hill and Phil Levis for explanation on their reference implementation, Kris Pister, Eric Brewer, David Culler and Turgut Aytur for their comments on our works.

7. REFERENCES

[1] M. Y. Hsiao, A Class of Optimal Minimum Odd-weight-column SC-DED Codes, IBM J. Res Develop, vol 14, no 4, July 1970

[2] Nelson Lee, Philip Levis and Jason Hill MICA High Speed Radio Stack http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/tinyos/tinyos-1.x/doc/stack.pdf

[3] CC1000 Data Sheet http://www.chipcon.com/files/CC1000_Data_Sheet_2_1.pdf

[4] Programming the CC1000 frequency for best sensitivity http://www.chipcon.com/files/AN_011_CC1000_optimized_for_best_sensitivity_1_1.pdf

[5] ATMega 103L Microprocessor Data Sheet (http://today.cs.berkeley.edu/tos/hardware/design/data_sheets/ATMEGA103.pdf)

[6] A Software Architecture Supporting Networked Sensors, Jason Hill. . Masters thesis, December 2000.(http://today.cs.berkeley.edu/tos/papers/TinyOS_Masters.pdf)

[7] Computer Networks - A Systems Approach, 2nd Edition by Larry L. Peterson and Bruce S. Davie, Morgan Kaufmann, 2000