A Flexible and Reliable Radio Communication Stack on Motes

Wei Ye, John Heidemann
Information Sciences Institute
University of Southern California

Deborah Estrin
Computer Science Department
University of California, Los Angeles

September, 2002

1 Introduction

This technical report describes the radio communication stack on the Mica Motes developed at USC/ISI and UCLA. It includes the architecture design, stack implementation, interfaces, and some testing and measurement results.

Our design and implementation of the communication stack are focused on the flexibility, reliability and efficiency. Some features are listed as follows, which are desirable for building different protocols and applications.

- Different layers/components are free to define their own packet formats. They can freely add their own headers to packets from their upper layers without causing interference.
- Packets with dramatically different lengths in fast consecutive transmissions can be reliably received. The current supported maximum packet length is 250 bytes.
- A full-featured MAC protocol for sensor networks, S-MAC [1], is implemented on the stack. It provides advanced features such as effective collision and overhearing avoidance, reliable and efficient transmission of long messages (can be much longer than 250 bytes), and low duty-cycle operations on radio.

2 Architecture Design

One of our design goals is to provide a flexible stack architecture that allows protocols at different levels can be easily built and compared with other similar protocols.

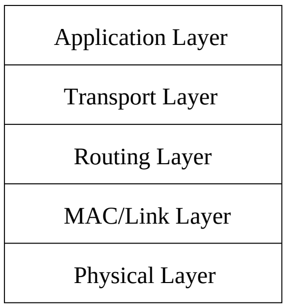

Figure 1: Generic layered model for sensor networks.

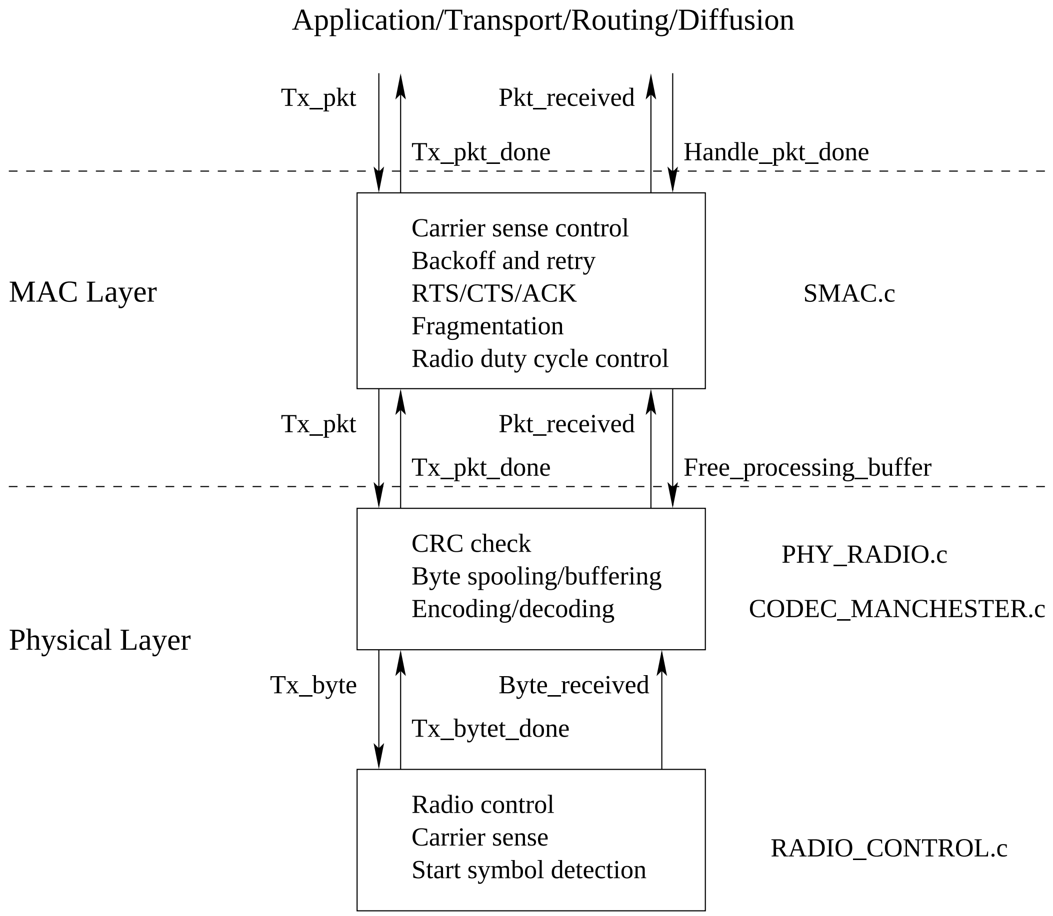

Figure 2: Stack structure and functions.

2.1 Layered Model

We adopt the layered model in traditional computer networks, and try to efficiently build it on Motes and TinyOS. The layers are intended to provide standard interfaces and services, so that various protocols can be developed in parallel. Figure 1 shows the generic layered model for sensor networks.

Our communication stack implemented the physical layer and the MAC/link layer. Figure 2 describes the detailed stack structure and functions of each component.

At the bottom, it is the component that controls the radio hardware. It provides clean interfaces for upper layers to put the radio into different states: idle, sleep, transmitting and receiving. This is essential for protocols that need to control the duty cycle of the radio. Physical carrier sense is implemented in this component. However, it gives full control to MAC layer about when to start carrier sense and its duration. The last important function in this component is detecting start symbol, synchronizing with the incoming packet and receiving each bit of it.

The component above the radio control provides physical layer interfaces to upper layers. For transmission, it accepts a packet from the MAC layer, calculates CRC, encodes each byte and spools it to the radio control component. The radio control component will then transmit the byte on the radio. For reception, the physical layer accepts each byte received from the radio, decodes it and checks CRC when the entire packet is received. It will pass the received packet to the MAC layer with the CRC error information.

On top of the physical layer, it is the MAC layer, whose basic function is to control the medium access for collision avoidance. We implemented S-MAC on the stack, which provides many advanced features compared with a basic CSMA MAC. First, for unicast packets, it uses RTS/CTS to solve the hidden terminal problem. They are also used for overhearing avoidance to prevent a node wasting its energy to receive data destined to other nodes. Second, its message passing function provides fragmentation support and enables efficient transmission of a long message. ACK and retransmission are used for each data packet/fragment for fast error recovery. Finally, it can put nodes into low-duty-cycle operations. In this mode, nodes perform periodic listen and sleep. The duty cycle is adjustable by user. The current default setting is 10%, *i.e.*, listening for 150ms, and sleeping for 1.35s. If S-MAC operates in fully active mode by disabling the periodic sleep cycles (work in progress), S-MAC becomes a protocol similar to the IEEE802.11 [2] in ad hoc mode.

MAC layer accepts a packet transmission request from its upper layer. If its carrier sense and backoff procedure successfully indicates that the medium is free it will pass the packet to the physical layer to start transmission immediately. Otherwise, it will go to sleep, wait until next available time and retry again. When the MAC layer receives a data packet from the physical layer without any errors, it delivers the packet to its upper layer.

2.2 Packet Format and Buffer Management

This section describes some techniques for efficiently building the layered architecture on the resource-constrained tiny nodes.

One of the advantages of the layered architecture is that each layer does not need to care about the implementation details in other layers. In traditional computer networks, each layer maintains its own buffers for packets to be transmitted and received. When a layer gets a packet to transmit from its upper layer, it copies the packet into its own buffer, fills in its header fields and passes the new packet to its lower layer. When it receives a packet from its lower layer, memory copy is also used and only the payload is passed to its upper layer.

On the highly resource-constrained tiny nodes like Motes, it is too costly for each layer to maintain its own buffers. Memory copy at each layer is also very inefficient. The TinyOS group at UC Berkeley proposed an efficient solution [3], in which a common packet format is defined in a centralized place. All components in the stack use the same fixed packet format, which includes header fields from all of them. This way, if different components maintain their own packet buffers, the buffer size will be exactly the same. It actually indicates that there is no need to maintain separate buffers. A common buffer can be shared by all layers. The downside of the solution is that all layers have to stick to the same packet format, which prevents them form defining their own packets and freely adding header fields in the common packet. But this flexibility is essential for various protocols to co-exist.

We propose a new solution that provides the desired flexibility. Meanwhile, it maintains the same efficiency with buffer sharing and without memory copy. The key idea is to use a *nested header* structure.

Each layer defines its own header structure, which is going to be added into packets coming from its upper layer. The first field of its header is the header of its immediate lower layer. This way, the header includes headers of all layers below it.

Let's look at an example about how the nested header works. We assume that there are three layers in the example: application, MAC and physical layer. The header of the physical layer only has one field, packet length, which is the length of the entire packet. It uses a trailer field, the CRC, at the end of each packet. The physical layer defines it header as

```
typedef struct{
   unsigned char pktLength;
} PhyHeader;
```

Suppose the MAC layer has two fields in its header: the receiver's address and transmitter's address. It will define its header as

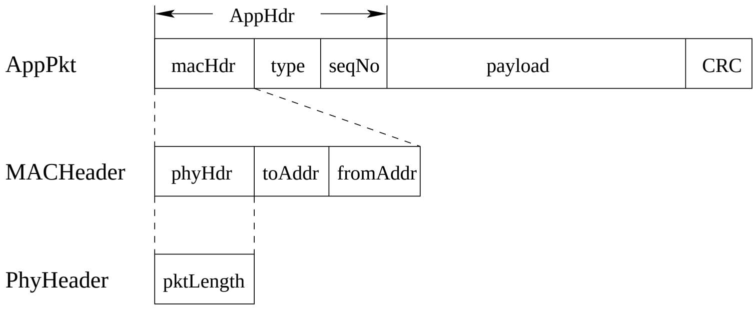

Figure 3: Application layer allocates a buffer for a packet using nested header structure.

```
typedef struct{
   PhyHeader phyHdr;
   short toAddr;
   short fromAddr;
} MACHeader;
```

Suppose the application layer has two fields in its header: packet type and sequence number. It defines its header as

```
typedef struct{
   MACHeader macHdr;
   char type;
   char seqNo;
} AppHeader;
```

When the application layer defines a packet format, it includes its header first, and then the payload, and then CRC as the last field.

```
typedef struct{
   AppHeader appHdr;
   char payload[length];
   short crc;
} AppPkt;
```

For an outgoing packet, the application layer allocates a buffer for it. The buffer has reserved space for headers of all lower layers. Figure 3 shows the allocated buffer in the above example. When the buffer is passed down to each lower layer, the lower layer only needs to process its own header fields.

For an incoming packet, the physical layer provides a buffer. When an entire packet is received, it passes the buffer to its upper layers. Similarly, each of them only needs to process the fields that belong to itself.

Using the nested header structure, each layer is free to define its own packet types. For example, if a protocol needs two types of control packets that have different header fields, it simply defines two packet types for each of them. The only thing it needs to conform with is to put the header of its immediate lower layer as the first field of its packets and put CRC as the last field.

3 Stack Implementation and Interfaces

This section describes the APIs that each layer provides to its upper layers and some features of implementation behind these APIs.

Code	Code Rate	Error Detection	Error Correction
4B/6B	2/3	1 bit of 6 bits	None
Manchester	1/2	1 bit of 2 bits	None
SEC/DED	1/3	2 bits	1 bit

Table 1: Comparison of different coding schemes that have been implemented in TinyOS. They are all DC balanced.

3.1 Physical Layer

The physical layer provides the following three groups of functions and APIs to its upper layer:

- Packet transmission and reception.
- Radio state control.
- Carrier sense and start symbol detection.

The APIs for packet transmission and reception are as follows.

```
char PHY_TX_PACKET(void* packet, unsigned char length);
void* PHY_RX_PACKET_DONE(void* packet, char error);
```

The first function is called by the upper layer when it wants to start a packet transmission. The physical layer accepts any types of packets with arbitrary packet length up to the limit defined by `MAX_PKT_LEN`. The default value of `MAX_PKT_LEN` is specified in `phy_radio_msg.h`, which can be overridden by each application in its `Makefile`. The maximum allowed packet length by the physical layer is 250 bytes. Our measurement results show that the physical layer can very reliably transmit and receive packets at this length.

When a packet is received, it signals its upper layer and passes the packet by calling the second function listed above. CRC check is perform by default for each received packet. The packet and its error information is passed to the upper layer. Passing an erroneous packet is necessary if the upper layer tries to recover the error using techniques like packet-level forward error correction (FEC).

The channel coding scheme used in our stack is the Manchester code [4, 5, 6]. The radio transceiver requires DC-balanced code for proper operation [7]. Manchester code is DC balanced — 0 is encoded to 01, and 1 is encoded to 10. It has the capability to detect 1 bit error in every 2 bits, since 00 and 11 are invalid codes. We use this feature in the physical layer to detection errors in the "length" field to avoid receiving incorrect number of bytes in a packet.

Other codes that have been used in TinyOS are the SEC/DED (single error correction/double error detection) code [8] and the 4B/6B code [7]. Table 1 compares these different codes. The code rate is defined as the ratio of data bits to total bits after encoding. A high code rate indicates that information content is high and coding overhead is low. The SEC/DED code is widely used in high-speed computer memory systems. Its implementation in TinyOS makes sure that the output is DC-balanced. The 4B/6B code encodes every 4 bits into 6 bits. It guarantees that the output is DC-balanced. It has the capability to detect 1 bit error in every 6 bits.

The physical layer maintains two buffers. When a packet is received and passed to upper layers for processing, it immediately switches to use the second buffer for receiving the next packet. It reduces the chance of packet loss in the case that the second packet arrives immediately after the first packet. If only one buffer is used, The physical layer cannot receive the next packet before upper layers finish handling the current packet.

The radio has four different states: idle, sleep, transmitting and receiving. The APIs for radio state control are as follows.

```
char PHY_RADIO_IDLE(void);
char PHY_RADIO_SLEEP(void);
```

The first function sets the radio into idle state, in which the radio tries to detect a start symbol. Carrier sense can be started in this state. The second function sets the radio into sleep state. The radio is turned off in this state and it cannot receive anything.

The function PHY_TX_PACKET that we described previously sets the radio into transmitting state. The radio will automatically go to receiving state from the idle state when a start symbol is detected.

The APIs for carrier sense and start symbol detection are as follows.

```
char PHY_START_CARR_SEN(unsigned short numBits);
char PHY_CHANNEL_BUSY(void);
char PHY_CHANNEL_IDLE(void);
char PHY_START_SYM_DETECTED(void);
```

The upper layer can start carrier sense by calling the first function, in which it specifies how long (in terms of number of sampled bits at a fixed sampling rate of 20Kbps) to monitor the channel. If the radio detects a busy channel during the carrier sense, it signals the upper layer about it immediately by calling the second function. The third function is called if the channel is sensed as idle for the entire carrier sense time. The last function is called to signal the upper layer about the detection of a start symbol. This group of functions helps the MAC layer to properly perform backoff procedures for collision avoidance. They provide necessary support for any contention based MAC protocols.

3.2 MAC Layer

The MAC layer that we have implemented on the stack is S-MAC [1]. Most of the APIs and implementations described here are only applicable to S-MAC. However, our stack architecture is flexible enough for people to build their own MAC protocol on top of the physical layer if they want.

The major APIs that S-MAC provides to its upper layer for sending and receiving a message are listed as follows.

```
char MAC_BCAST_MSG(void* msg, unsigned char length);
char MAC_UCAST_MSG(void* msg, unsigned char length, short toAddr, unsigned char numFrags);
char MAC_TX_NEXT_FRAG(void* fragment);
void* MAC_RX_MSG_DONE(void* packet);
```

S-MAC uses different mechanisms to send broadcast and unicast messages. For broadcasting, only the CSMA protocol is used. The first function listed above is called by the upper layer to send a broadcast message. Before sending the message, the MAC performs carrier sense with randomized duration. If carrier sense indicates that the channel is idle, the MAC will pass the message to the physical layer to send it out immediately.

The CSMA protocol does not address the hidden terminal problem. Even if the channel is idle at the time when the first node starts sending, a hidden node cannot detect the transmission by just doing carrier sense, and thus may start sending in the middle of the first transmission, which results in collision.

The exchange of RTS/CTS is an effective way to solve hidden terminal problem, and is adopted in S-MAC for unicasting. (Broadcasting cannot directly use this mechanism because of multiple CTS replies.) Collisions can also happen on RTS packets. But since they are very short (10 bytes in current implementation), the cost is much lower than the collision on a long data packet. However, if the data packet is indeed very short, it is not worth to use the RTS/CTS. For this reason, S-MAC defines a variable RTSThreshold that is used to turn off RTS/CTS if the length of a data packet is smaller than it (work in progress).

Message passing is another feature for unicasting in S-MAC. It allows the upper layer to efficiently transmit a long message, which can be much longer than the MAX_PKT_LEN set by the physical layer. To use message passing, the upper layer needs to divided a long message into multiple fragments. The length of each fragment should be smaller than MAX_PKT_LEN.

Functionality and features	ISI stack	Berkeley stack
Effective throughput	10kbps	13.3kbps
Radio bandwidth	20kbps	40kbps
Code rate of channel coding	1/2 (Manchester)	1/3 (SEC/DED)
Packet format	Nested headers	Fixed
- Layer-specific packets	Supported	Not Supported
- Layer-specific headers	Supported	Not Supported
- Variable length packets	Supported	Supported
- Memory copy across layers	No	No
Clear separation of MAC and PHY	Yes	No
Code size	8KB	4KB
Old & nesC formats	Old only	Old and nesC

Table 2: Functionality comparison of ISI stack with Berkeley stack: overall features.

Functionality and features	ISI stack	Berkeley stack
File names of components	PHY_RADIO	CRC_PACKET
		STACK_MANAGER
	CODEC_MANCHESTER	SEC_DED_ENCODING
	RADIO_CONTROL	NETWORK_LISTENER
		SPI_BYTE_FIFO
		RADIO_TIMING
	SLAVE_PIN	SLAVE_PIN
Buffers provided	1 for Rx/1 for processing	1 for Rx & processing
CRC check on Rx	Progressive	Check after Rx
Packet processing delay	Fixed	Varies w/ packet length
Coding scheme	Manchester	SEC/DED
Simple commands to turn radio on/off	Yes	No
Time stamping	work w/ other components	Integrated
Carrier sense interface for different MACs	Yes	No

Table 3: Functionality comparison of ISI stack with Berkeley stack: the physical layer.

To start a unicast transmission, the function `MAC_UCAST_MSG` is called. The arguments include the receiver's address, the length of each fragment and the number of fragments in this message. If there is no fragmentation in the message, simply set the number of fragments to 1.

For each transmitted fragment, the transmitter expects an ACK packet from the receiver. If it receives the ACK, S-MAC signals its upper layer. If there are more fragments, the upper layer calls the function `MAC_TX_NEXT_FRAG` to transmit the next fragment. If the ACK packet is not received by the transmitter, S-MAC will retransmit the current fragment for fast error recovery.

S-MAC calls the function `MAC_RX_MSG_DONE` to pass a received packet to its upper layer. Only packets that are received without any errors will be passed for further processing.

4 Functionality Comparison with Berkeley's Stack

This section gives a detailed functionality comparison of our stack with Berkeley's stack in the standard TinyOS release. The comparison is divided into three parts: overall features, the physical layer and the MAC layer.

Table 2 compares the overall features currently provided by the two stacks. The main advantages of ISI stack over Berkeley stack are the flexible packet format and the clear separation of MAC and the physical layer (PHY). The nested header structure allows each component to freely define its own packet formats and add its header fields in each packet from its upper layers. In Berkeley's stack, there is only

Functionality and features	ISI stack	Berkeley stack
File names of components	SMAC RANDOM_LFSR CLOCK	NETWORK_LISTENER RANDOM_LFSR
Broadcast	CSMA/backoff	CSMA/backoff
Unicast	RTS/CTS/Data/ACK	Data/ACK
- Address hidden terminal problem	RTS/CTS	No
- Retry on RTS lost	Yes	N/A
- Re-Tx data/fragment on ACK timeout	Yes	???
- Fragmentation support	Yes	No
- Sleep while neighbors are talking	Yes	No
Low-duty-cycle operation	Configurable	No

Table 4: Functionality comparison of ISI stack with Berkeley stack: the MAC layer.

one packet format, which is used by all components. They are restricted on freely adding layer-specific headers in the packet, since that may break other components. Clear separate of the MAC and physical layers allows different MAC protocols can be easily built on top of the same PHY.

Table 3 compares the physical layer of the two stacks. Our stack is designed to reliably and efficiently handle packets with dramatically different lengths. More importantly, it provides a clean interface to support different MAC protocols. In Berkeley's stack, the physical layer is tightly integrated with the MAC layer. It becomes very difficult to build a different MAC on the stack.

Table 4 is the comparison at the MAC layer. One of the most important features that S-MAC provides is the low-duty-cycle operation on the radio. It is able to make tradeoffs on latency for energy savings.

For broadcast, the two stacks use the same mechanism, *i.e.*, CSMA with random backoff. For unicast, S-MAC provides much more features (as listed in the table) than Berkeley's MAC. These features make the unicast much more reliable than broadcast. The fragmentation support provides an efficient way (message passing) to transmit a message that is longer than the physical-layer limit.

5 Performance on Testbed Experiments

In order to characterize the basic performance of our communication stack, we have done testing and measurement on the Mica motes. The primary version of Mica motes that are used at USC/ISI has the radio operating at 433MHz. It comes with a matched external whip antenna. We have also done some measurement on the version of Mica motes with 916MHz radio and an on-board antenna. As a comparison, we have done some similar measurement on the communication stack in the TinyOS release.

5.1 Physical Layer Performance

We focus on two aspects of the performance of the physical layer. The first one is that if it is able to efficiently and reliably handle packets with dramatically different lengths. The other one is the maximum transmission range. The experiments are performed in a clean environment without strong noise and interference.

In the first test, we use only one transmitter and one receiver with a distance of 1 meter between them. The transmitter sends 3 groups of packets with a fixed length of 40, 100 and 250 bytes. Each group has 100 packets being sent back-to-back, *i.e.*, start sending the second packet as soon as the sending of the first packet is done. The test is repeated for 10 times.

Table 5 shows the results using Berkeley stack and ISI stack. Berkeley stack has some systematic loss of long packets being sent back-to-back. Then we add a fixed delay of 20ms between two consecutive packets, and repeat the same tests. There is no systematic loss this time. Table 6 shows the new results.

Physical layer	Packet length		
	40 bytes	100 bytes	250 bytes
Berkeley stack	100%	50%	50%
ISI stack	100%	100%	100%

Table 5: Reception rates of fixed-length packets being sent back-to-back.

Physical layer	Packet length		
	40 bytes	100 bytes	250 bytes
Berkeley stack	100%	99.5%	99.3%
ISI stack	100%	100%	100%

Table 6: Reception rates of fixed-length packets being sent with 20ms packet interval.

In the second experiment, we measure the reception rate as the distance between the transmitter and receiver increases. There are one transmitter and multiple receivers that are put along a line with an equi-space of 1 meter. In this test, the transmitter sends 100 packets in a group with packet length randomly changes between [10–250] bytes. Since Berkeley stack needs some packet interval to handle long packets, we still put 20ms delay between two consecutive packets when testing Berkeley stack. When testing ISI stack we send the variable-length packets back-to-back. The test is repeated 10 times for each stack.

This test is performed in the hallway within the ISI's building during a weekend. We put marks on the location of each node to ensure that the same node is put at exactly the same location after we change the communication stack on it. During the period of test, there are no people walking around, and most doors are closed. (Some doors were open, but had kept the same positions during the entire test.) This arrangement tries to keep the radio propagation conditions the same for all tests.

Figure 4 shows the reception rate at different distances using Berkeley stack and ISI stack. A close look reveals that ISI stack obtained almost all 100% reception rate on the receivers whose distance are between 1m to 16m. In fact, except the two receivers at 13m and 16m, who obtained 99.9%, all other receivers within 16m achieved 100% reception rates. Now we look at the same communication range when using Berkeley stack. Except the receiver at 7m, whose reception rate is about 40%, other receivers obtained from 98.5% to 99.8%. The reason of the low reception rate on the receiver at 7m is not clear. It could be caused by undesirable multipath propagations. However, the same node with ISI stack on the same location obtained 100% reception rate.

On the other hand, Berkeley stack obtained longer transmission range. It acchieved 95.8% reception rate at 20m. The transmission range of ISI stack is about 18m. After that, the reception rate decreases very quickly. The longer transmission range of Berkeley stack is due to the coding scheme it adopts. This is verified by the next experiment which uses ISI stack with different channel coding schemes.

The third experiment is performed along with the previous experiment at the same place and during the same period of time. It compares different channel coding schemes with only ISI stack. The locations of the sender and the receivers are exactly the same as those in the second experiment.

Figure 5 shows the measured reception rate at different distances. Each line is based on 10 tests. The 4B/6B code has the minimum coding overhead, but it is not as robust as the other two. Using SEC/DED code, ISI stack obtains similar results on transmission range as Berkeley stack. Moreover, all receivers at the distances between 1m to 16m got 100% reception rate.

The SEC/DED code can correct single-bit errors in each byte after encoding. As distance increases the received signal power decreases, so does the signal-to-noise ratio. As a result, the bit error rate (BER) increases. The error correction capability of SEC/DED code increases the transmission range by reducing the BER.

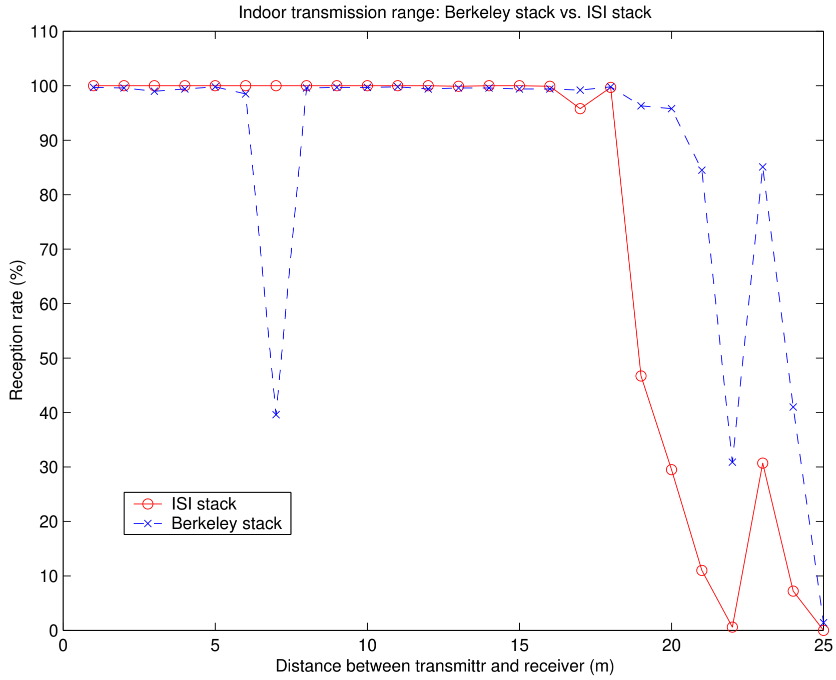

Figure 4: Comparison of Berkeley stack and ISI stack on packet reception rates at different distances between the transmitter and the receiver. Each line is based 10 repeated tests of 100 packets with packet length randomly changes between [10–250] bytes.

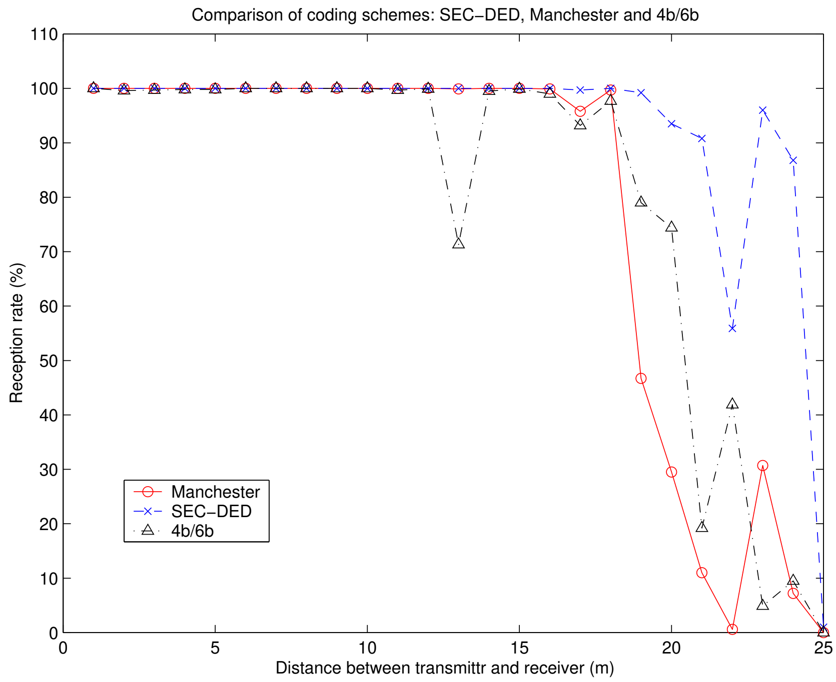

Figure 5: Comparison of different channel coding schemes on ISI stack measured by packet reception rates at different distances between the transmitter and the receiver. Each line is based 10 repeated tests of 100 packets with packet length randomly changes between [10–250] bytes.

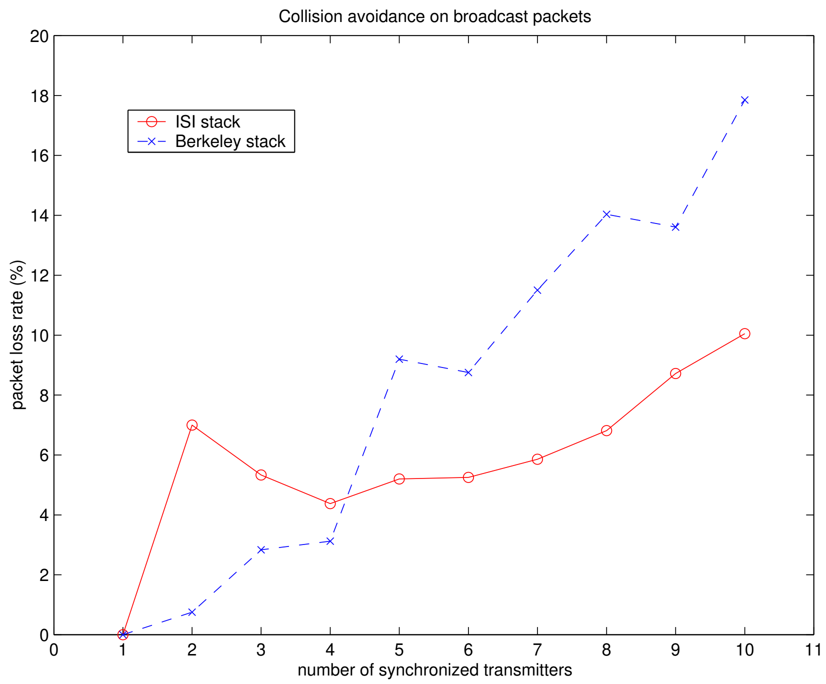

Figure 6: Packet loss rate due to collisions from multiple synchronized transmitters. Each transmitter sends 20 packets in each test. Each point in the figure is based on 10 repeated tests.

5.2 Collision Avoidance in MAC

After measuring the physical layer performance, we have done a simple test to measure the collision avoidance capability of the MAC layer of both Berkeley stack and ISI stack.

The experiment is designed to measure the packet loss rate at a single receiver, which is only caused by collisions because of the synchronized transmissions from multiple transmitters. We use a control node to send a packet to all transmitters. Upon receiving the control packet, each transmitter starts a timer which fires every half second. A packet will be generated on each transmitter when its timer fires if the previous packet is already sent out. Each transmitter sends 20 packets in each test. The nodes take slightly different time to finish sending all their packets because of the backoff delay.

So far we have only done the measurement on broadcast packets. In this case, both Berkeley stack and ISI stack are using CSMA, with only some difference in implementation details.

Figure 6 shows the packet loss rate due to collisions as the number of synchronized transmitters increases. With a small number of transmitters, *i.e.*, $2-4$, Berkeley stack obtains better results. When the number of transmitters are more than 4, ISI stack achieves better performance.

It would be interested to see the performance of unicast packets where the ACK and retransmission can be utilized. We expect that the loss rate can be significantly reduced.

Another interesting experiment is to measure collisions over multi-hop networks, where hidden terminal problem becomes an important issue. We expect the RTS/CTS mechanism can effectively resolve it.

6 Summary

This report describes the design and implementation details about the communication stack on Mica Motes developed at USC/ISI and UCLA. Experimental results are presented to characterize some performance of the stack. Some comparison studies are also made with the communication stack developed by the TinyOS group at UC Berkeley.

As mentioned earlier, we plan to do more testing and measurements on our stack, including throughput and collision avoidance over multi-hop networks.

The following features will be available in S-MAC soon.

- A user-configurable threshold to control the use of RTS/CTS. If there is only one fragment and its length is smaller than the threshold, RTS/CTS will not be used.
- A user-configurable option to completely turn off the periodic sleep. It is for some applications that cannot tolerate the latency introduced by the sleep.

Acknowledgments

This work is in part supported by NSF under grant ANI-0220026 as the MACSS project, and by DARPA under grant DABT63-99-1-0011 as the SCADDS project. The work is also support by the Center for Embedded Networked Sensing (`http://cens.ucla.edu/`) and a grant from the Intel Corporation.

The authors would like to acknowledge the discussions from members of the SCADDS projects, the TinyOS group (`http://tinyos.millennium.berkeley.edu/`) at UC Berkeley, and researchers at Intel Labs. Specifically, we would like to thank Mark Yarvis and David Culler for their detailed feedback; Athanasios Stathopoulos, Jerry Zhao and Naim Busek for testing the stack and in-depth discussions.

References

[1] Wei Ye, John Heidemann, and Deborah Estrin, "An energy-efficient mac protocol for wireless sensor networks," in *Proceedings of the IEEE INFOCOM*, New York, USA, June 2002, pp. 1567–1576.

[2] LAN MAN Standards Committee of the IEEE Computer Society, *Wireless LAN medium access control (MAC) and physical layer (PHY) specification*, IEEE, New York, NY, USA, IEEE Std 802.11-1999 edition, 1999.

[3] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister, "System architecture directions for networked sensors," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, USA, Nov. 2000, pp. 93–104, ACM.

[4] Andrew S. Tanenbaum, *Computer Networks*, Prentice-Hall, Inc., New Jersey, USA, 3 edition, 1996.

[5] Wiliam Stallings, *Data and Computer Communications*, Prentice-Hall, Inc., New Jersey, USA, 5 edition, 1997.

[6] Roger Forster, *Manchester encoding: opposing definitions resolved*, `http://www.engj.ulst.ac.uk/sidk/quintessential/chapters/manchester.htm`, 2000.

[7] RF Monolithics Inc., "Ash transceiver designer's guide," `http://www.rfm.com/`, 2002.

[8] Daniel J. Costello Jr., Joachim Hagenauer, Hideki Imai, and Stephen B. Wicker, "Applications of error-control coding," *IEEE Transactions on Information Theory*, vol. 44, no. 6, pp. 2531–2560, Oct. 1998.