Tmote Implementation of BMAC and SMAC protocols

Kiran Kumar
Y5111047
vkirankr@iitk.ac.in

Phani Kumar
Y5104042
phani@iitk.ac.in

ABSTRACT

SMAC(sensor MAC) and BMAC (Berkeley MAC) are two most widely used protocols for sensor network applications. Currently their implementations are available for only Mica2 motes. Our intent is to implement the two protocols for tmotes. We modified the unicast preamble Send/Receive part of BMAC protocol, synchronization part in SMAC protocol. We added fragmentation of message and overhear avoidance features to BMAC, optional ACK, outlier based CCA (Clear Channel Assessment) features to SMAC. In this paper we explain the details of our implementation, modifications we have made, and the effects of the modifications. With our implementation of SMAC, we can disprove the statement "BMAC outperforms SMAC in all scenarios". We make a statement that "BMAC outperforming SMAC and SMAC outperforming BMAC is purely scenario based rather than a general statement."

1. INTRODUCTION

Due to sensor mote's small size and their ability to communicate through wireless, sensor networks are gaining popularity in recent years. Some applications of sensor network are Habitat monitoring, countersnipper detection, traffic monitoring etc. Limitations of Sensor motes are limited power, memory, CPU execution speed and range of communication. Many researchers focused on reduction of power usage by increasing the delay of communication. It is a known fact that power consumption for transmitting one bit is equivalent to processing 1000 or more instructions. Some of the known techniques to reduce power consumption are data aggregation, on-demand Adhoc routing etc. MAC protocol has greater responsibility to conserve power than any other protocols in higher layers. Many MAC protocols exist for sensor networks, out of which SMAC, TMAC, and BMAC have implementations in TinyOS.

SMAC [1] is CSMA based protocol, neighbor schedules are maintained at each node for synchronization, RTS/CTS is used to avoid hidden node problem. TMAC [2] is same as SMAC with a timeout added, for any node if none of transmit or receive event occurs within the timeout period then node goes to sleep, node will refresh the timeout period for each transmit or receive event. Under homogenous load SMAC and TMAC performs equally, under variable load TMAC out performs SMAC by factor of 5 [2]. BMAC [3] is also CSMA based protocol, avoids synchronization problem by sending a long preamble before the data packet. Unlike SMAC and TMAC, BMAC is a light weight protocol, with ACK as option. In order to make sure that whether channel is free or busy, BMAC uses outlier based CCA rather than threshold based CCA (SMAC/TMAC uses). We expect that the reader knows the features of SMAC and BMAC protocols, if not, we recommend to refer the references [1] and [3]. We mention the details of these protocols in abstract manner. We donot expect the reader to know the details of Tmote and CC2420. We mention the specific details whenever needed.

The problem with these efficient MAC protocols is that their availability is confined to only Mica2 CC1000 radio. These protocols are not implemented for other radios, e.g. Tmote's CC2420. Tmotes are gaining popularity because of their low cost. Known currently ongoing applications using Tmote are Bridge Monitoring [], and Vibration Analysis of bearings[]. These applications are under development. To our knowledge, SMAC and BMAC are not implemented for Tmotes. For Tmotes only CSMA MAC is available. These applications are using CSMA MAC unsatisfactorily. We intended to solve the problem, we implemented major part of BMAC and partly SMAC. We haven't solved the problem for the ongoing projects, but we may solve this problem for projects which start in near future.

Current implementation of BMAC (CC1000 specific) lacks proper documentation and merges the radio level details with the MAC features. It becomes hard to figure out which part of the code should be modified to port to CC2420 radio and this problem remains same if anyone wants to port for other type of radios. Documentation of SMAC is better than BMAC, but the code is complex to understand, every one of us believe that "BMAC outperforms SMAC" and is not worth to implement it for new radios. We feel these are the reasons why SMAC and BMAC are not ported for CC2420 radio.

Our problem definition is as follows "Implementing BMAC and SMAC code to CC2420, removing possible inefficiencies, easily portable to any other radio platforms". This problem statement made us to rewrite the code from the scratch. Currently CC2420 radio implements CSMA, Radio Send, Radio Receive modules. Our assumption is that CC2420 also implements Radio sleep, Radio wakeup, and Outlier based CCA. These assumptions aren't unrealistic; one of the members in our sensor network group implemented these features for CC2420. At the time we started implementation these were the assumptions made by us. Currently, they arenot considered as assumptions.

Current results are BMAC's MAC layer components (will be explained later) are working as expected if Radio sleep/wakeup and outlier CCA aren't used. SMAC components (will be explained later) are coded but not tested. The rest of the paper is organized as follows, Section 2 gives a brief overview of our approach and modifications done for the basic protocols, Section 3 discusses the added or modified features to SMAC and BMAC, Section 4 discusses the architecture of SMAC and

BMAC, Section 5 discusses how App layer and MAC communicate, Section 6 discusses the what are the parameters to be modified to improve the performance of BMAC protocol, Section 7 discusses the implementation details and issues of BMAC, Section 8 presents the power calculations of BMAC protocol, Section 9 presents the results and current progress of SMAC and BMAC implementation, and section 10 concludes the paper.

2. Our approach

As mentioned in previous section that our implementation not concentrated on just porting protocols to CC2420 radio, we also tried to add/modify features to the protocols to save power and increase flexibility. We differentiate our features with existing implementations by tables 1 and 2 given below. We explain each modification in detail in the later sections.

Feature	CC1000 impl.	CC2420 impl.	Reason to change
Broadcast Preamble Send/ Receive	Stream of preamble bytes	Stream of preamble packets	implementation issue with CC2420
Unicast Preamble Send/ Receive	stream of preamble bytes	sequence of preamble packets	to conserve power wastage.
Send/ Receive broadcast message	only one packet is transferred, no fragmentation	added fragmentation, virtually seems to App layer that its long message is transferred at a time.	to improve flexibility.
Send/ Receive unicast message	ACK is option, only one packet is transferred, no fragmentation	ACK is option, added fragmentation.	to improve flexibility.
Overhear avoidance	not implemented.	implemented.	to conserve power wastage.
Implementation details	hard to understand the code.	Layered architecture, hide radio details at MAC layer code.	to improve readability, and easy portability.

Table.1 Comparison of CC1000 and CC2420 implementations of BMAC

Feature	CC1000 impl.	CC2420 impl.	Reason to change
Synchronization	maintains neighbors' schedules, modifies node's local time with its synchronized node's time, maintains the time difference of other neighbors' time.	doesn't modify node's time, maintains the time difference of neighbor's time. SYNC frame is divided into SYNC frame1, SYNC ACK frame and SYNC frame2.	modification of node's local time doesn't solve the problem. To avoid network partitioning and to avoid logical asymmetric links.
Send/Receive broadcast message.	only one packet is transferred, no fragmentation.	added fragmentation.	to improve flexibility.
Send/Receive unicast message	ACK is compulsory, fragmentation available.	ACK is optional, fragmentation available.	to conserve power wastage, for unreliable data.
Clear Channel Assessment	Threshold based	Outlier based	to better assess the channel state.

Table.2 Comparison of CC1000 and CC2420 implementations of SMAC

3. Added/modified features of SMAC and BMAC

3.1 BMAC features

As the design features of BMAC protocol were efficient, we haven't made many modifications to the protocol.

One problem we identified with BMAC protocol is that application designer has to select the preamble period which determines number of preambles to be sent before sending a data packet. Finding an optimistic value of length of preamble period is difficult, because setting it to small value doesn't guarantee that the receiving node will wakeup in the assumed preamble length time, setting it to a high value solves the synchronization problem but increases power wastage. Choosing a value for preamble length based on theoretical calculations may not match in real deployments. We don't mean that finding an optimal value is too difficult, but it is not easy. So, we tried to make this problem a bit easy. We let the application designer to choose the preamble period to large value considering the worst case scenario that the receiving node is guaranteed to wake up within the chosen value. Our protocol doesn't send preamble packets for this whole period, but uses this value as a timeout value. We use the terms **preamble-block,** to denote the one preamble packet that sender sends. Our protocol works as follows:

1. Sender sends a preamble-block, waits for a preamble ACK from receiver for a small amount of time.
2. If receiver wakes up and receives preamble, sends preamble ACK to sender.
3. After receiving preamble ACK, sender stops sending preamble and starts sending data fragments.

Two obvious questions comes into mind "How long the sender waits for preamble ACK?" and "Will the sender sends preamble continuously, if receiver fails?", we solved these two problems using two timeouts. Sender starts a small timeout called **preamble-block ACK timeout** when sending the preamble (this value is not the user sets as preamble length, this timeout is very less than the preamble length), if it doesn't receive preamble ACK from receiver within preamble-block ACK timeout, sender again sends another preamble-block. Coming to the second question, now the user specified preamble length period is used as timeout, sender send preamble only for preamble length period, if receiver doesn't respond within this time an exception is raised to App layer indicating that receiver is not responding. We haven't made the sending preamble part automated. Still selection of optimum preamble length remains same for broadcast messages, because broadcast preamble there is no ACK. We solved the problem only for unicast case only.

We added the fragmentation feature to BMAC, this doesn't add any power to the protocol, but increases the flexibility to use. We have hidden the radio level details to application layer and defined a message structure to application layer, which allows the application layer to Send/Receive 100 bytes at a time. The value 100 is variable; user can set a value to suit the need. We explain the fragmentation in the detail in implementation section. Fragmentation feature is borrowed from SMAC protocol.

BMAC doesn't say any thing about hidden node problem and overhear avoidance at the initial stages of implementation we were unable to decide whether to include these features at MAC layer or to leave the implementation to upper layers. We felt that overhear avoidance is essential feature for every application, rather than making every application to implement it, we implemented for application layer. We didn't attempt solve the hidden node problem (using RTS/CTS), we found that using even using RTS/CTS, the hidden node problem is unsolved and is not worth to implement.

3.2 SMAC features

Some design features of SMAC also were not as efficient as BMAC in terms of power savings, so added the best fit features of BMAC to SMAC, so that the basic protocol is not violated. We made ACK as option for unicast messages, replaced threshold based CCA with outlier based CCA. SMAC doesn't provide fragmentation to broadcast messages, but we added fragmentation feature to broadcast messages too. The heart of SMAC protocol is synchronization, but we identified that synchronization cannot be achieved by maintaining schedules; we explain this with an example.

Consider the topology shown in Figure.1, Let us suppose that node 1 selects the smallest random wait value. Node 1 broadcasts its schedule, node 2 and 3 receive schedule packet, adjusts their local time, back offs for random time to broadcast node 1 schedule, when node 4 and 5 receive the packet corrects their time and back offs for random amount time to broadcast node 1 schedule. Node 6 and 7 corrects their time with node 1 time. It can be noted that how far times of nodes 6 and 7 are in synchronization with node 1 time.

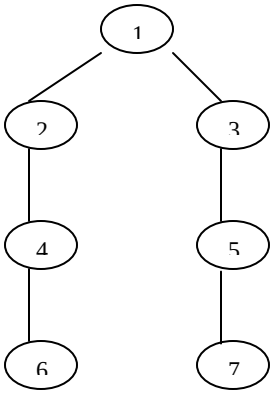

Figure.1 Topology considered

‡ We thought that changing node's local time with neighbor's time doesn't help to solve the synchronization problem, so we made our protocol implementation not to change the node's local time, but maintains only neighbor's time difference, rest is similar to SMAC protocol to send data to its neighbor. One other modification made is that SYNC frame is divided into three sub frames, SYNC Frame1, SYNC ACK frame, SYNC Frame2. In SYNC Frame1, each node sends its local time even though it receives SYNC frame from its neighbors (this is not the case with original SMAC, if a node receives SYNC packet from its neighbor it stops transmitting its SYNC packet, but transmits received schedule packet). If a node receives SYNC packet from other node, includes in neighbor table then the other node is considered as neighbor. The above point means that not all nodes from which a node receives SYNC packets are considered as neighbors. Only from nodes it receives first m packets are considered as neighbors. This is simplistic, but inefficient way of choosing neighbors. We leave RSSI based neighborhood table maintenance as future work. The value m is the size of neighbor table, which is a constant, selected by application designer.

In SYNC_ACK frame every node broadcasts its neighborhood list. The necessity of this frame will be clear shortly. Each node after receiving others' neighborhood list, checks whether it is a neighbor of any other node. In SYNC_Frame2, only nodes which are not neighbor of any other node sends SYNC packet again. In SYNC_Frame1 collision probability will be high. So, SYNC packet sent by a node may be lost; and this node will not be neighbor for any other node. No node will sent unicast packets to this node. If a node does not receive any SYNC packet and if all other nodes doesn't receive its SYNC packet, this node will be isolated from the network. Though it may seem that probability of this problem occurrence is low, but effect is high if it occurs and difficult to debug the problem. In order to avoid this problem, we introduced SYNC_ACK frame. With this frame each nodes knows it's receive neighbors. If a node doesn't have at least one receive neighbors, sends SYNC frame again in SYNC_Frame2.

‡: This idea was suggested by our instructor Dr. Bhaskaran Raman, Asst. professor , Dept of CS&E , IIT Kanpur, India.

4. Architecture

In this section we explain our implementation style of two protocols. First we explain the BMAC architecture then we show SMAC architecture and point out the similarities between BMAC and SMAC features.

4.1 BMAC Architecture

To make the code more readable, we used a layered architecture, most common architecture for network protocol implementation. The architecture is illustrated in Figure 2. We defined four layers App layer, MAC layer, PhyCommlayer, and CC2420 radio layer. We define App layer as union of routing layer, TCP layer and Application layer. MAC layer components are Send/Receive Broadcast Preamble for specified preamble period, Send/Receive Unicast Preamble with Preamble ACK enabled, Send/Receive Broadcast messages with fragmentation added, Send/Receive Unicast messages with ACK as option and fragmentation added, and Overhearing avoidance. CC2420 layer components are CSMA, Radio Send/Receive, Outlier based CCA, Radio Sleep/Wakeup. It can be noted there are no components in PhyComm layer, the reason the functionality included in PhyComm layer is tunneling BMAC packet into TOS_Msg. In otherwords, PhyComm layer is an adapter between MAC layer and CC2420 layer. The necessity of this layer is to hide the TOS_Msg details (radio details) at MAC layer code, we expect that this improves code readability and also help in porting our protocol to any other radio type.

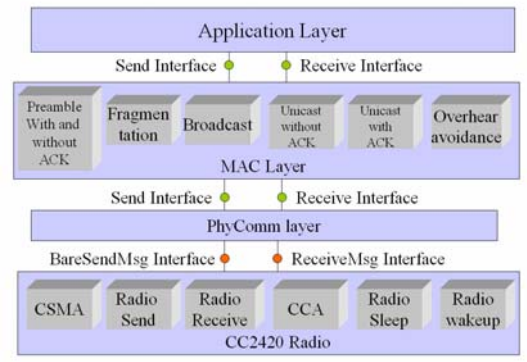

Figure.2. BMAC Layered Arch.

Radio Send/Receive and CSMA components are readily available in CC2420 radio package. As mentioned earlier one of the members in our sensor network group implemented Outlier based CCA, Radio Sleep/Wakeup. Considering CC2420 layer as black box, we started implementing MAC layer components.

4.2 SMAC Architecture

For SMAC also, we used a layered architecture. SMAC CC1000 implementation also used Layered Architecture. The architecture is illustrated in Figure 3. It can be noticed that most of the components are same as BMAC components, componets reused from BMAC implementation are Outlier CCA, Radio Sleep/Wakeup, Overhearing avoidance, Receive Broadcast messages, Recive Unicast messages with optional ACK and RTS/CTS.

We were unable to reuse Send Broadcast and Unicast messages, because in BMAC Send operation is preamble based, but in SMAC it is synchronization based. Implementation of Sending Broadcast message with fragmentation was a difficult part in SMAC, because next fragment should be transmitted if previous fragment is transmitted to all neighbors, which is not possible by calling send operation once, but should be called multiple times for a fragment. But most of functionality of Unicast Send message with fragmentation are reused from BMAC.

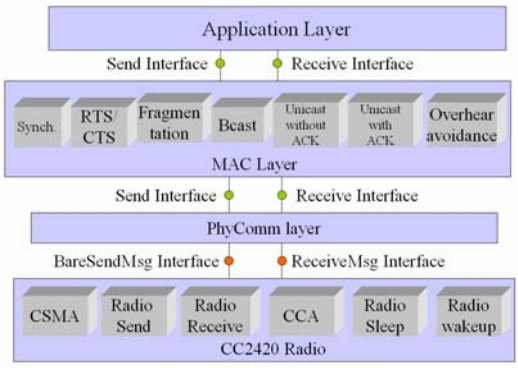

Figure.3. SMAC layered Arch.

5. Interaction between App layer and MAC layer

In order App layer to communicate with the MAC it has to use the SendInterface, ReceiveInterface, MsgToMAC structure, MsgFromMAC structure. App layer should also understand the type of exception codes sent by MAC layer. Whether the MAC protocol is BMAC or SMAC, the interaction between App layers doesn't change. The reason for this is SMAC provides all features of BMAC to App layer.

Send message. In order to send a message the App layer should specify the following details to MAC layer: data of the message, length of the message, receiver node address/id, type of message (unicast or broadcast) and reliability level (ACK needed or not). To transfer mentioned information from App layer to MAC layer, we defined a message structure MsgToMAC.

```
struct MsgToMAC    {
      int8_t *data;     // pointer to data to be sent
      uint8_t len;      // length of the message
      uint8_t toAddr;  // address of destination
      bool isReliable; // ACK as option
      bool isUnicast;  // unicast or broadcast
  };
```

For broadcast messages MAC doesn't use the toAddr and isReliable fields. For broadcast the receiver address is set to 0xff.

Receive message. When a message is received, BMAC should specify the following details to App layer: data of the received message, length of the received message, sender node address/id. To transfer mentioned information from MAC layer to App layer, we defined a message structure MsgFromMAC.

```
struct MsgFromMAC   {
     int8_t data[MAX_MAC_MSG_SIZE];
     uint8_t len;      // length of the message
     uint16_t fromAddr;    // source address
  };
```

MAX_MAC_MSG_SIZE is the size of receive buffer.

Send Interface. SendIntrerface is defined as follows

```
interface SendInterface    {
     command result_t send(void* msg);
     event result_t sendDone(void* msg,
                                  uint8_t errorcode );
  }
```

App layer calls the *send* command to send MsgToMAC object to BMAC layer, MAC layer fragments the message, sends the message, depending on its type, to the specified receiver's address and signals the sendDone event to App layer.

Receive Interface. ReceiveIntrerface is defined as follows

```
interface ReceiveInterface  {
     event void* receiveDone(void* msg,
                                  uint8_t errorcode);
}
```

When all the specified number of fragments are received from sender the receiver signals an event to App layer that a message has received.

Type of exceptions: Following are the list of exceptions raised by MAC layer to the App layer:

- **NO_EXCEPTION.** Indication of success.
- **NOT_READY_ERR.** Rises if App layer tries to send a message to MAC layer when it is not in idle mode.
- **NULL_DATA_ERR.** Rises if App layer sets the data field of message to null.
- **ZERO_LEN_ERR.** Rises if the length of the message to be sent is zero.
- **LEN_OVERFLOW_ERR.** Rises if the length of the message to be sent is greater than the receive buffer size. This exception will be raised at sender and message will not be sent.
- **SOURCE_NOT_RESPONDING.** Rises if the receiver times out waiting for a fragment from sender.
- **PHY_FAILED_ERR.** Rises if the CC2420 doesn't signal SendDone event with the specified time.
- **PREAMBLE_TX_ERR.** Rises if the receiver doesn't send Preamble ACK within the Preamble period.
- **DATA_PKT_TX_ERR.** Rises if the receiver doesn't send Data ACK even after maximum number of retransmissions.

6. Performance Tuning

In addition to best design features, performance (power savings) of a protocol to a specific application depends on the values of the tunable parameters. For BMAC, these parameters are Preamble period, Maximum number of retransmissions, Receive buffer length, and Payload size of TOS_Msg (TOSH_DATA_LENGTH).

With our implementation preamble length may not be a an issue for unicast transmissions, but will be a tradeoff for power savings for broadcast messages, because preamble packets packets are sent continuously in the specified period. We used the preamble period as 20 preamble packets as default.

More the number of retransmissions, more the consumed. But it is the tradeoff between level of reliability and power wastage. We limit the maximum number of retransmissions to 16, can be tuned from 0 to 15 by application designer. Default value is set to 8.

As we know that for sensor mote, size of main memory is minimal. So, memory savings is also important issue. The length of data sent from App layer to MAC layer is limited by receive buffer size. The size of receive buffer size is purely application's choice. The minimum value is 21 bytes (payload of one BMAC packet) and maximum size is application's choice. We set the default buffer size to 100.

Changing the payload length of TOS_Msg (TOSH_DATA_LENGTH) is not feature of our protocol, but is a feature of CC2420 radio module. The upper and lower limits of the TOSH_DATA_LENGTH can be referred from CC2420 data sheet. Setting TOSH_DATA_LENGTH greater than receive buffer size doesn't cause any harm to the functionality of protocol, but our protocol uses only receive buffer size number of bytes, rest of the bytes are transmitted with garbage value. Default value is set to 28, we found for another version of tinyos it was 29.

7. Implementation

In this section we present reasons behind the following implementation issues:

- Separation of data from code.
- Send and Receive interfaces.
- Tunneling.
- Fragmentation.

7.1. Separation of data from code.

We separated the data which any two layers share, from code of lower layer. The data files are BmacConst.h, BmacMsg.h, PhyCommMsg.h and source code files are BMacM.nc, BmacC.nc, PhyCommM.nc, and PhyCommC.nc.

BmacConst.h contains the values defined for tunable parameters and constant values defined for exceptions. BmacMs.h contains the message structures of send message and receive message to/from MAC layer. PhyMsg.h contains the message structures through which MAC layer and PhyComm layer communicate with each other.

BmacM.nc and BmacC.nc are module of configuration files of BMAC layer, PhyCommM.nc and PhyCommC.nc are module and configuration files of physical layer.

7.2. Send and Receive interfaces.

It can be noted from Figure.2 and Figure.3 that App layer and MAC layer, MAC and PhyComm layer communicate through SendInterface and ReceiveInterface, but PhyComm and CC2420 layer communicate through BareSendMsg and ReceiveMsg interfaces. We defined SendInterface and ReceiveInterface.

The reason for this involves C language pointer details. The parameters to BareSendMsg and ReceiveMsg is TOSMsgPtr, this doesn't work for our implementation. The SendInterface from App layer to MAC layer should be of type pointer to MsgToMAC, and MAC layer to physical layer should be of type pointer to PktToPhy, similarly for receive interface from MAC layer to App layer should be of type pointer to MsgFromMAC, and PhyComm layer to MAC layer should be of type pointer to PktFromPhy.

To serve our requirements we defined SendInterface and ReceiveInterface with parameter of type void pointer. Void pointer allows to pass pointer parameters of any data type.

7.3. Tunneling. Tunneling is the most frequently used technique in network protocol designs. We defined the BMAC packet structure as follows:

```
struct PktToPhy    {
      uint8_t length;   // length of packet
      uint8_t toAddr;  // receiver address
      uint8_t fromAddr;  // sender address
      uint8_t timeRemaining;
      struct type_retx typeinfo;
      struct msgid_seqno msginfo;
      int8_t data[PHYCOMM_LENGTH];
       uint16_t crc;    // checksum
  }
```

In order to send BMAC packet over wireless medium, we fit BMAC packet into payload field of TOS_Msg. Figure.4 illustrates tunneling. BMAC packet is defined such that its length matches exactly with payload size of TOS_Msg. As we discussed earlier that length of TOS_Msg can be varied. The payload length of BMAC packet adjusts itself with the varying length of TOS_Msg.

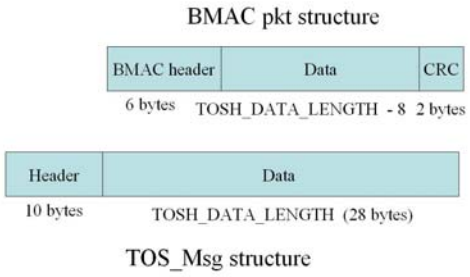

Figure.4. Tunneling of BMAC packet into TOS_Msg

7.4. Fragmentation.

Our implementation of BMAC protocol supports fragmentation, which is not supported by CC1000 implementation of BMAC. Fragmentation is illustrated in Figure.5. With the default values of receive buffer size equal to 100 and TOSH_DATA_LENGTH equal to 28, a message of length 100 bytes from App layer is fragmented into 5 packets (20 bytes in each packet) and sent to receiver. The receiver appends the data from each packet and sends an 100 bytes message to App layer. Fragmentation is invisible to App layer. Number of fragments for a message is informed to receiver in the preamble packet.

Providing fragmentation doesn't mean that App layer should send 100 bytes message each time, it can send message of any length between 1 and 100. With a message of length less than or equal to 20 bytes, no fragmentation is needed. This means that fragmentation is an option but not compulsory to App layer.

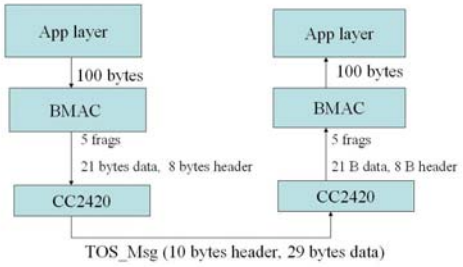

Figure.5. Fragmentation of message

8. Power Calculation.

The current characteristics of CC2420 is given Table.3. The Tmote data sheet of doesn't specify scaling unit of these values, we expect the scaling unit is time in milliseconds. Although CC2420 data sheet says that data rate is 250kbps, practical observed data rate was only 40kbps.

mode	Normal (mA)	Max (mA)
Standby	5.1	21.0
Transmit	19.5	21
Receive	21.8	23

Table.3. current specifications of CC2420

Since our implementation of BMAC doesn't implement sleep/wakeup, if only BMAC running on motes the radio will be in idle listening mode, because BMAC itself cannot generate any packets by itself. With 3.3Ah battery, the lifetime of a mote is 3.3*1000/5.1 hours (or) 647.05 hours. The lifetime of a node is purely dependent on application (rate of transmitting messages) running on top of BMAC layer. We calculate the power in the following scenarios:

- Transmitting a broadcast message.
- Transmitting a unicast message with ACK disabled.
- Transmitting a unicast message with ACK enabled.
- Receiving a broadcast message.

- Receiving a unicast message with ACK disabled.
- Receiving a unicast message with ACK enabled.

In our implementation, maximum of 20 preamble packets are sent before sending data message. With fragmentation, 100 byte message is fragmented into five BMAC packets (with TOSH_DATA_LENGTH = 28). To avoid timing errors temporarily, we set inter packet delay to 250ms (large value, to be optimized).

Whether it is preamble or data packet, the packet size sent/received on radio is 38 bytes (CC2420 header (10 bytes) + payload (28 bytes)). With 40 kbps data rate, radio will be in transmit/receive mode for (38* 8)/(40 * 1000) seconds (or) 7.6ms to transmit/receive a packet.

We explain current calculations for an application, we ran to test BMAC. The message size used in that application is 100 bytes.

8.1. Transmitting a broadcast message

For broadcast messages all 20 preamble packets are sent.

$I_{TxBcast} = I_{TxPreamble} + I_{TxData}$

$I_{TxPreamble} = I_{Tx20pkt} + I_{19InterPktDelay}$

$= 20 * 7.6 * 19.5$ mA + 19 * 250 * 5.1 mA

$=$ 27189 mA.

$I_{TxData} = I_{Tx5DataPkt} + I_{4InterPktDelay}$

$=$ 5 * 7.6 * 19.5 mA + 4 * 250 * 5.1 mA

$=$ 5481mA.

$I_{TxBcast} =$ 27189 + 5481

= 32670mA.

8.2. Transmitting a unicast message with ACK disabled.

For unicast messages number of preamble packets sent depends on receiver response (preamble ACK) for preamble. Number of preamble packets sent is probabilistic rather deterministic. We assume that receiver didn't fail, implies receiver sends preamble ACK for one of preamble packet.

Expected number of preamble packets sent =

1 * pr {receiver responds for first packet}

+ 2 * pr {receiver responds for second Packet} +..

+ 20 * pr { receiver responds for 20th packet}.

= 1 * 1/20 + 2 * 1/20 + .. + 20 * 1/20

= 10.5 packets

$I_{TxUnicastNoACK} = I_{TxPreamble} + I_{TxData}$

$I_{TxPreamble} = I_{Tx10.5pkt} + I_{RxPreambleACK} + I_{10InterPktDelay}$

$=10.5*7.6*9.5 + 7.6*21.8 + 10*250*5.1$

$=$ 14471.78 mA.

$I_{TxData} =$ 5481mA (same as broadcast).

$I_{TxUnicastNoACK} = 14471.78 + 5481$

= 19952.78 mA.

8.3. Transmitting a unicast message with ACK enabled.

$I_{TxUnicastWithACK} = I_{TxPreamble} + I_{TxData}$

$I_{TxPreamble} =$ 14471.78 mA.(same as unicast with ACK disabled)

$I_{TxData} = I_{Tx5DataPkt} + I_{Rx5DataACK} + I_{4InterPktDelay}$

$= 5*7.6 *19.5 + 5*7.6* 21.8 + 4*250*5.1$ mA

$=$ 6309.4 mA.

$I_{TxUnicastWithACK} = 14471.78 + 6309.4$

$=$ 20781.18 mA.

8.4. Receiving a broadcast message.

Number of preambles received by receiver is probabilistic, the mean number of received preamble packets is 10.5

$I_{RxBcast} = I_{RxPreamble} + I_{RxData}$

$I_{RxPreamble} = I_{Rx10.5pkt} + I_{10InterPktDelay}$

$= 10 .5 * 7.6 * 21.8 + 10 * 250 * 5.1$ mA

$=$ 14489.64 mA

$I_{RxData} = I_{Rx5DataPkt} + I_{4InterPktDelay}$

$=$ 5 * 7.6 * 21.8 + 4 * 250 * 5.1 mA

$=$ 5924 mA

$I_{RxBcast} = 14489.64 + 5924$

= 20413.64mA.

8.5. Receiving a unicast message with ACK disabled.

$I_{RxUnicastNoACK} = I_{RxPreamble} + I_{RxData}$

$I_{RxPreamble} = I_{Rx10.5pkt} + I_{TxPreambleACK} + I_{10InterPktDelay}$

$= 10.5*7.6*21.8 + 7.6*19.5 + 10*250*5.1$

$=$ 14637.64 mA.

$I_{RxData} =$ 5924mA (same as broadcast).

$I_{RxUnicastNoACK} = 14637.64 + 5934$

= 20571.64 mA.

8.6. Receiving a unicast message with ACK enabled.

$I_{RxUnicastWithACK} = I_{RxPreamble} + I_{RxData}$

$I_{RxPreamble} =$ 14637.64 mA. (same as unicast with ACK disabled)

$I_{RxData} = I_{Rx5DataPkt} + I_{Tx5DataACK} + I_{4InterPktDelay}$

$$= 5*7.6*21.8 + 5*7.6*19.6 + 4 * 250 * 5.1$$

$$= 6309.4 \text{ mA}.$$

$$I_{RxUnicastWithACK} = 14637.64 + 6309.4$$

$$= 20947.04 \text{ mA}.$$

With above calculations, the lifetime of a node depends on number and type of messages transmitted and received. The calculations are done for a specific application (bottom up manner), one can easily generalize the calculations for other applications.

9. Results.

BMAC executes as expected for Send/Receive broadcast preamble, Send/Receive unicast preamble, Send/Receive broadcast message (with fragmentation), Send/Receive unicast message without ACK (with fragmentation), Send/Receive unicast message with ACK(with fragmentation). Though BMAC is functionally working properly, we don't expect/suggest to use our current implementation of protocol in applications, because it works correctly in the following preconditions:

- No radio sleep (100% radio wakeup)
- Inter packet delay is 250ms

We know that, sensor MAC protocol without integrating radio sleep/wakeup don't have much value. Till now we concentrated only on functional attributes of BMAC, but not its quality attributes. Our future work for BMAC includes

- integrating radio sleep/wakeup and CCA.
- reducing inter packet delay.
- thorough testing of actions taken for each timeout conditions.

SMAC is coded, we are out of compilation errors. We don't claim this as completed work, but we feel that our first step for its implementation is finished. The implementation details mentioned in sections 6 and 7 are specific only to BMAC, SMAC is not mentioned in those sections.

10. Conclusion.

In this paper, we discussed the architecture and implementation details of SMAC and BMAC to Tmote. We mentioned some implementation features of SMAC and BMAC implementations for Mica2, and discussed how we improved them. Some modifications are done at design level (Division of SMAC SYNC frame into SYNC frame1, SYNC ACK frame, and SYNC frame2, optional ACK for SMAC unicast messages, replacing threshold based CCA with outlier based CCA, and reduction of preamble period in BMAC) and some modifications at implementation level (added fragmentation feature for BMAC broadcast/unicast and SMAC broadcast, changing the code structure of BMAC). In the theoretical view, we feel that our implementation of SMAC can outperform BMAC in many scenarios, thus we contradict the statement "BMAC outperforms SMAC in all scenarios" and we support for "performance of SMAC and BMAC are application dependent, each can be considered as alternatives for any application".

11. Acknowledgements

We thank our instructor Bhaskar Raman, who gave an opportunity to handle this interesting problem, encouraged and guided us in all respects to make this project successful. We also thank Jagan, who implemented Radio wakeup/sleep and outlier based CCA and co-operated with us in many cases. We also thank Nilesh and Hemanth, helped us in debugging problems.

12. References

[1] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002), New York, NY, USA, June 2002.

[2] T. van Dam and K. Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the FirstACM Conference on Embedded Networked Sensor Systems*, Nov. 2003.

[3] Joseph Polastre, Jason Hill, and David Culler, "Versatile low power media access for wireless sensor networks," in *SenSys'04*, 2004.