1. UNCERTAINTY-DRIVEN BMAC (UBMAC)

1.1. Introduction

One of the most widely used MAC protocols for wireless sensor networks is BMAC. BMAC is a Carrier Sense Media Access (CSMA) protocol that provides options for ultra low power operation, effective collision avoidance, and high channel utilization. It supports various employs low-power listening modes, which reduce the energy cost of idle network listening by increasing the energy cost of the transmitter. More specifically, before every packet transmission the transmitter sends a preamble that is guaranteed to handle the worst case time uncertainty between itself and the receiver. The problem with this approach is the huge energy overhead that is added to each packet transmission. This overhead increases, when the duty cycle decreases, as it can be seen in Table 1.1.

Duty cycle	Preamble size (bytes)	Packet loss rate (measured)
100%	20	1%
35.5%	94	1%
11.5%	250	1%
7.53%	371	1%
5.61%	490	5%
2.22%	1212	1%
1.00%	2654	10%

Table 1.1 BMAC preamble sizes and packet loss rate in SOS

Of these bytes, only four bytes would be required if the sender and receiver were perfectly synchronized. Two of these bytes correspond to the minimum preamble size, and two bytes are normally added to take care of miscellaneous factors such as radio on/off time, software variations etc. Extra bytes over the minimum of 4 bytes are added by BMAC to take care of time uncertainty between the nodes. Specifically, addition of 1

byte allows a leverage of around 416 microseconds. This approach is followed by a MAC protocol that uses time synchronization, in order to reduce the overhead of the big preamble.

Uncertainty-driven BMAC (UBMAC) is a MAC protocol that combines BMAC with time synchronization modules. Currently, UBMAC integrates only with TPSN and RATS, however its architecture is easily expandable, therefore, more time synchronization modules can be added. One of UBMAC's main advantages is that it can use different time synchronization modules, in order to synchronize with different nodes at the same time. For example, one application may specify TPSN to synchronize with one node and RATS to synchronize with another one, whereas at the same time a second application might use a third time synchronization module, in order to synchronize with a third node.

1.2. Implementation

UBMAC is split into 2 parts, which are closely correlated. The first part (Synchronization module) takes care of the exchange of synchronization information between the nodes and the calculation of the wakeup time of the destination nodes. The second part (Radio module) uses the above information, in order to transmit the packets successfully. UBMAC's transmission and reception procedures can be seen in Figure 1.1, whereas the complete architecture can be seen in Figure 1.2

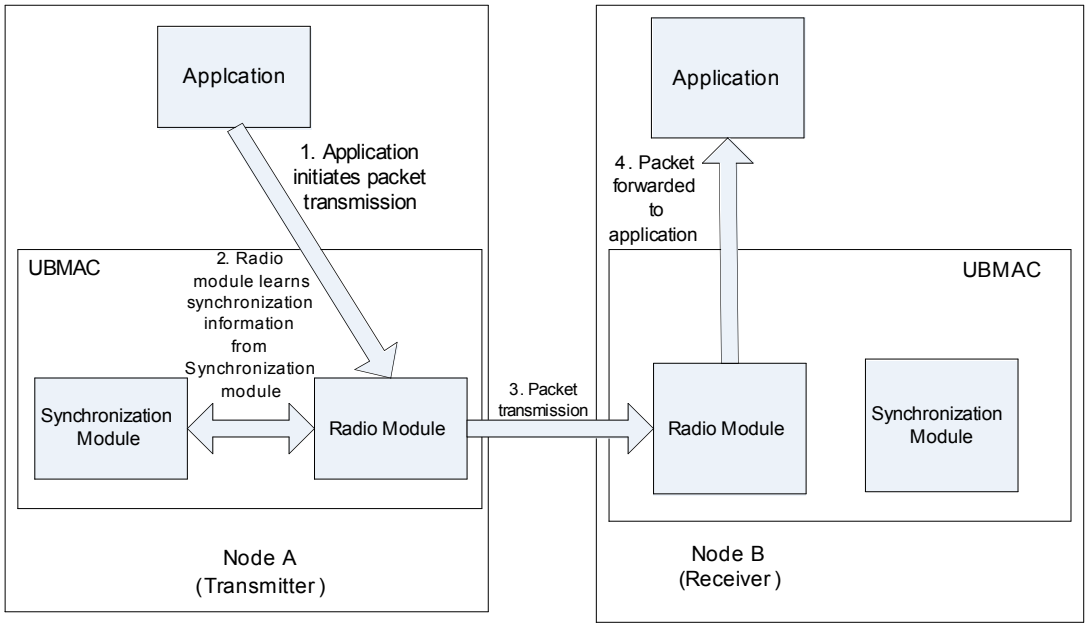

Figure 1.1 UBMAC's transmission and reception procedures

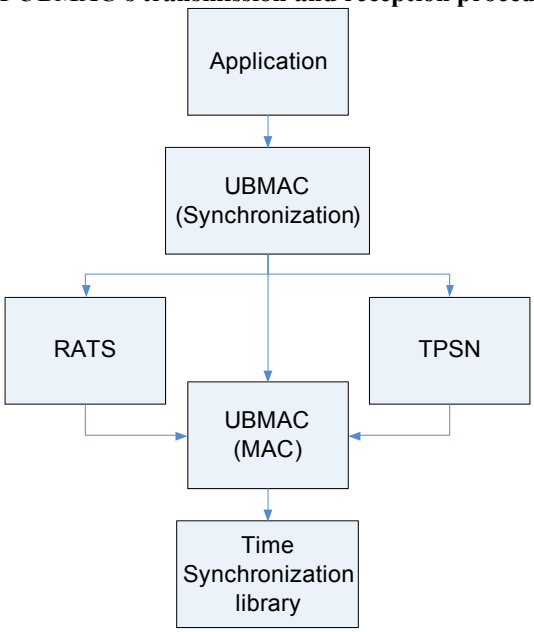

Figure 1.2 UBMAC structure

1.2.1 UBMAC- Synchronization module

The responsibilities for this module are two-fold:

- Transmit the node's information, so that it can be used by other nodes, in order to calculate the local wakeup time

- Use the time synchronization modules, in order to calculate the wakeup time of the other nodes that are needed by the applications

In order to implement the first functionality, UBMAC transmits periodically the period of the MAC duty cycling, as well as the latest wakeup time. After the node boots, UBMAC enters a learning phase, during which the packets are transmitted with a small period (the default duration for the learning phase is 15 minutes, during which the node's information is transmitted every 1 minute). After the learning phase is over, the information is transmitted at a lower rate, which by default is once every 15 minutes. In order to store the latest wakeup time, the Synchronization module exports a function called notifyUbmac(), which takes the current time as an argument and stores it. This function is also used to calculate the period of the duty cycling, which is actually the difference between the current and the previous wakeup times. However, in order to take care of cases, in which a few consecutive wakeup interrupts were lost and the corresponding period is increased, UBMAC allows the value of the period to be updated, only if the new value is within a window, whose values and range depend upon MAC's "lplpower" setting.

UBMAC's second responsibility starts, when an application sends it a MSG_START_TIMESYNC message and passes a ubmac_init_t struct. This struct has the following fields:

Data type	Field name	Description
unsigned short	node_id	Id of the node, to whom the application wants to send packets
unsigned char	timesync_mod_id	Id of the module, which should be responsible for the time synchronization (e.g. RATS, TPSN, etc)
		Precision of the time synchronization in milliseconds. This determines RATS'

unsigned int	sync_precision	packet transmission period, as well as the preamble that is used by UBMAC.

Table 1.2 Format of ubmac_init_t struct

After the reception of the ubmac_init_t struct, UBMAC adds an entry to its internal database. Each entry has the following information:

Data type	Field name	Description
unsigned short	node_id	Id of the node, to whom the application wants to send packets
unsigned int	wakeup_time	Latest known wakeup time of the node (translated to the local time by the time synchronization module)
unsigned int	untranslated_wakeup_time	Latest known wakeup time of the node (before the time translation)
unsigned char	is_wakeup_time_valid	It is set to true, after the current node has received the first wakeup time of the destination node
unsigned char	period	Period of the destination node's duty cycling
unsigned char	ref_count	Reference counter. Counts the number of the applications that have registered the particular destination node id.
unsigned char	timesync_mod_id	Id of the module that is responsible for the time translations
unsigned int	sync_precision	Precision of the time synchronization in milliseconds. This determines RATS' packet transmission period, as well as the preamble that is used by UBMAC

Table 1.3 Format of UBMAC's internal database

After the synchronization module finishes with its learning phase, it is able to perform a valid time translation of the wakeup times that are being sent by the other nodes. The Synchronization module exports 2 functions that are being used by the Radio module, in order to transmit packets.

- ubmacGetTime(uint16_t node_id) : It returns the interval, after which the destination node will wake up. This interval is expressed in clock ticks, so that they can be used to set an SOS timer.
- ubmacGetPreamble(uint16_t node_id) : It returns the preamble that needs to be used, in order to send a packet to the destination node. The value of the preamble depends only on the precision of the time synchronization between the current node and the destination, as it was passed to UBMAC by the application.

Until the learning phase of the time synchronization module is terminated, the untranslated wakeup time is stored and the corresponding packets are transmitted using BMAC. In addition, BMAC is used as a failsafe mechanism anytime that UBMAC is not able to calculate either the preamble (e.g. because no application has requested time synchronization with the corresponding node id) or the wakeup time of the destination node (e.g. because the time synchronization module wasn't able to do the latest conversion).

Finally, in order for an application to remove a particular node id from UBMAC's database, it needs to send a MSG_STOP_TIMESYNC message to UBMAC using post_short. The id of the node is passed as the unsigned short value that is sent through post_short.

1.2.1.1 UBMAC with RATS

The implementation of the UBMAC synchronization module is expandable, therefore many different time synchronization modules can be used at the same time, in order to achieve time synchronization with different nodes. There are 3 places, in which the code flow depends on the time synchronization module:

- After the reception of MSG_START_TIMESYNC
- After the reception of the (untranslated) wakeup time from a node
- After the reception of the translated wakeup time from the time synchronization module

This section explains the cooperation between UBMAC and RATS, whereas the next section explains the cooperation between UBMAC and TPSN.

After UBMAC receives MSG_START_TIMESYNC, it sends a MSG_RATS_CLIENT_START to RATS. Both the values for the time synchronization precision, as well as the value for the node id are the same ones that were used, when MSG_START_TIMESYNC was received. This allows RATS to start the time synchronization procedure with the destination node.

Whenever a wakeup time is received from another node, RATS stores it and sets the variable is_wakeup_time_valid equal to true. There is no need for time translation at this point.

1.2.1.2 UBMAC with TPSN

TPSN's integration to UBMAC is simpler than RATS'. First of all, since TPSN is a stateless protocol, there is nothing that needs to be done, whenever UBMAC receives a

MSG_START_TIMESYNC message. This is due to the fact that TPSN calculates only the instant clock offset and doesn't perform long-term synchronization.

Whenever, a wakeup time is received from another node, UBMAC sends a tpsn_t struct to RATS. Like with RATS, the return message type is again set equal to MSG_TIMESYNC_REPLY.

Afterwards, TPSN calculates the offset between the clock of the local node and the one of the destination and sends the reply back to UBMAC. Again, UBMAC checks, if the reply is valid and sets the translated time to be equal to the untranslated plus the offset that was designated by TPSN.

It needs to be noted here that the buffer that is passed from UBMAC to TPSN is still owned by UBMAC, after the time translation, therefore it needs to be freed, in order to avoid a memory leak. Also, it is worth reminding here that since TPSN overwrites duplicate requests in its internal buffer, it is possible that some of the time translation requests might not be returned back to UBMAC.

1.2.2 UBMAC – Radio module

The implementation of UBMAC's Radio module resides in cc1k_radio.c, which also includes the implementation of BMAC. Both MAC protocols can coexist in the same kernel, since they don't interfere with each other's functionality. The protocol stack uses one single FIFO queue, in which all the packets, which are about to be transmitted, are enqueued. When a packet is dequeued, the network stack checks, if the flag SOS_USE_UBMAC is enabled and if the packet needs to be unicast. In that case the packet is transmitted using UBMAC. If the flag is not enabled or the packet needs to be broadcast, then the packet is transmitted using BMAC. Also, if UBMAC cannot handle

the packet (i.e. there is no information that allows UBMAC to calculate the destination node's wakeup time or the needed preamble), then the packet is sent using BMAC. This check is conducted both whenever a new packet is passed to the Radio module (function radio_msg_alloc()), if the packet queue is empty, and after the termination of a packet transmission, if the packet queue is not empty.

In order to transmit a packet using UBMAC, the Radio module needs to know the destination node's wakeup time, therefore it calls function ubmacGetTime(). This function is provided by the Synchronization module and returns the offset between the current time and the time, when the destination module will wakeup. The Radio module uses this offset, in order to start the UBMAC timer. When the UBMAC timer expires, the node waits for a random CSMA delay and then starts transmitting the packet. In order to find out the preamble that needs to be used, it executes ubmacGetPreamble(), which is also provided by the Synchronization module. According to the preamble size should be (4+⌈synchronization_precision/416⌉. The variable synchronization_precision is expressed in microseconds and corresponds to the value that was passed from the application to the Synchronization module. Therefore, if the synchronization_precision is set to 1000 microseconds, then the preamble length should be 8 bytes. However, in order to remove certain time uncertainties, we have added a leverage of 24 bytes. Apart from that, we have set the transmitter to wakeup for a predefined interval (set to 50 milliseconds) before the receiver and wait for the random backoff delay to expire. Since the backoff interval is random, this means that the period between its expiration and the wakeup of the receiver is random (but limited to no more than 50 milliseconds). Therefore, the

preamble of the packet varies accordingly, in order to make sure that the receiver will receive it.

1.3. Compilation - Usage

In order for somebody to use UBMAC, he needs to specify the flag RADIOSTACK=ubmac during compilation. Currently UBMAC is implemented only for the mica2 platform, so he needs to execute:

make mica2 install RADIOSTACK=ubmac

If this flag is specified, then RATS will also be included in the compiled target. If somebody needs to use TPSN, then he can either load it dynamically as a module or change the application's makefile to include it.

Finally, in order for somebody to use UBMAC, he needs to do 2 things:

1)Use the struct ubmac_init_t, in order to register a node id with a time synchronization module. The form of the struct is:

Data type	Field name	Description
unsigned short	node_id	Id of the node, with whom packets will be transmitted using UBMAC (all broadcast packets are transmitted using BMAC)
unsigned char	timesync_mod_id	Id of the time synchronization module (currently it can be either TPSN or RATS)
unsigned short	sync_precision	Required synchronization precision. The granularity of the clock (which will directly impact the accuracy of the time synchronization) is defined in the "systime.h" file.

2)In order to send a packet to a node using UBMAC, the flag SOS_MSG_USE_UBMAC needs to be set in post_ net, e.g.

post_net(s->pid, s->pid, MSG_TEST, 0, NULL, SOS_MSG_USE_UBMAC, RECEIVER_ID);

If the flag is not specified, then the packet will be transmitted by BMAC. This means that by defining RADIOSTACK=ubmac somebody can use either BMAC or UBMAC to transmitt packets. Also, if the packet needs to be broadcasted, then it'll be sent using BMAC regardless of whether the SOS_MSG_USE_UBMAC flag has been specified. Finally, if the node id of the desination node hasn't been registered with UBMAC, then the packet will be transmitted using BMAC.

Currently, UBMAC uses the systime timer for duty cycling and since the period of the systime timer is approximately 530 microsends, this means that the duty cycling periods will be different if RADIOSTACK=mica2 is specified than if it’s not. It's in the future plans to use a different hardware timer (preferably Timer0, so that we'll be able to duty cycle not only the radio, but also the processor), so this will change in the future.