The λMAC framework: redefining MAC protocols for Wireless Sensor Networks

Tom Parker Gertjan Halkes Maarten Bezemer Koen Langendoen
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
The Netherlands
{T.E.V.Parker, G.P.Halkes, K.G.Langendoen}@tudelft.nl mcb@robuust.nl

Abstract

Most current WSN MAC protocol implementations have multiple tasks to perform - deciding on correct timing, sending of packets, sending of acknowledgements, etc. However, as much of this is common to all MAC protocols, there is duplication of functionality, which leads to larger MAC protocol code size and therefore increasing numbers of bugs. Additionally, extensions to the basic functionality must be separately implemented in each MAC protocol.

In this paper, we look at a different way to design a MAC protocol, focusing on the providing of interfaces which can be used to implement the common functionality separately, and on the core MAC role of timing. We also look at some examples of MAC extensions that this approach enables. We demonstrate a working implementation of these principles as an implementation of B-MAC for TinyOS, and compare it with the standard TinyOS B-MAC implementation. We show a 35% smaller code size, with the same overall functionality but increased extensibility, and while maintaining similar performance. We also present results and experiences from using the same framework to implement T-MAC, LMAC, and Crankshaft. All are demonstrated with data from real-world experience using our 24 node testbed.

1 Introduction

Current Medium Access Control (MAC) protocol design for Wireless Sensor Networks (WSNs) covers a wide variety of different tasks. A MAC protocol is responsible not only for deciding when to send packets, but also what to send. For example, generating the standard Unicast sequence of RTS/CTS/DATA/ACK messages is usually the responsibility of the MAC protocol after the application has provided a data packet to be sent. The MAC must maintain an internal state machine monitoring which one of these packets it last sent or received, enabling it to determine what packet should be sent/received next.

Unfortunately, the decision about whether a MAC's implementation of Unicast uses RTS/CTS messages (which are seen by some designers as overhead, and by others as required for reliability) tends to be a somewhat haphazard affair. Often, whether they are required should be an application-level decision, and so some MAC protocols that implement RTS/CTS allow this functionality to be switched off and on at run or compile time. However, this is another example of a feature that may or may not be in a given MAC protocol depending on the whims of its designer.

Given that we have a set of functionality that should be common to all MAC protocols, but certain implementations do or do not have particular features implemented, we lose out on the advantage of common functionality: the idea that we can ideally use any given MAC protocol as a drop-in replacement. Additionally, because the duplication of effort results in both increased bug count due to multiple implementations of the same ideas (e.g. Unicast), and a system that is hard to extend, we conclude that the current design brief for MAC protocols has a number of significant problems, and so should be rethought.

In this paper we will set out an improved design brief for MAC protocols, and show how these principles can be implemented efficiently by demonstrating our example λB-MAC protocol. The same principles will then be shown to work for λ-layers implementations of T-MAC, LMAC, and Crankshaft we will show more data gathered from these protocols.

2 Rethinking MAC protocols

We wish to redesign the process for creating a MAC protocol such that the common functionality that does not necessarily need to be in a MAC protocol itself can be separated out. The first step to achieving this is to determine what is common functionality, and what are MAC-specific requirements.

2.1 Existing concepts

Before we can start rethinking the design process for MAC protocols, we need to look at the current state of the art. Current WSN MAC protocols are usually grouped in two different groups: TDMA protocols (LMAC [10], TRAMA [18], PEDAMACS [2], etc) and CSMA-based protocols (S-MAC [23], T-MAC [3], B-MAC [16], etc). These two approaches are usually regarded as being very different, and even within each approach we are shown many different protocols that all do things in drastically different ways. However, despite all the apparent differences, all of these protocols have one thing in common - they are designed to manage the available time in the radio medium in order to fulfil certain metrics while sending/receiving messages (latency, energy usage, etc).

Specifically, they all do this by managing when a particular node can send messages - TDMA protocols do this by separating the available time into slots and allowing nodes only to send in their slot; CSMA protocols do this by making nodes perform carrier sense before sending (and in the case of protocols like S-MAC, also by waiting until the beginning of the next "frame"). In total, a MAC protocol must do three things: given an application wishes to send a packet, determine what time this node will be able to send; perform a message exchange to send the packet at that point; and transmit appropriate control packets so that the application layer will be able to send packets in the future.

2.2 Role separation

We then looked at separating the large existing MAC protocols into 3 parts: below the MAC, above the MAC and a λMAC layer. This set of layers we refer to collectively as the MAC stack, and together they should do everything a traditional monolithic MAC layer would do on its own.

Our first task was looking at the modules required "below" the λMAC layer. Working from the conclusions of Section 2.1, we know that MAC protocols need to send/receive packets, and to decide when to send/receive. The former can be achieved with a "dumb" packet layer (no queueing, minimal latency, switches radio on/off only when told to); the latter requires medium activity detection (as part of the "dumb" packet layer) and/or a time synchronisation layer. Time synchronisation can also then be used to generate "frames" (periodic timers, as used by all TDMA protocols and S/T-MAC), but it would need to be designed such that it will not interfere with protocols that do not require time synchronisation (e.g. B-MAC [16]).

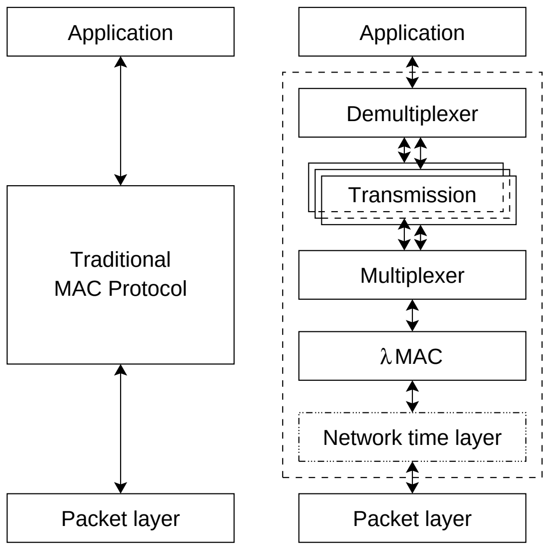


Figure 1: A traditional MAC protocol vs. the λMAC protocol stack.

The biggest question regarding how much we can pull out of a standard MAC layer was deciding what a λMAC layer actually really needs to do. Or in other words, knowing what a complete MAC stack needs to do, what makes one MAC protocol different from another? Our conclusion was simple: time management. One of the standard opinions about the role of WSN MACs is power management, and time management can be considered an extension of this - one of the time management roles is deciding when to switch the radio on/off, but the other is deciding when to start sending a packet sequence. However, once a node has started a packet sequence (e.g. all of Unicast after the RTS message), the code becomes remarkably generic and MAC-portable, yet is currently still embedded within the MAC. What if we could extract that - let the MAC decide when to initiate packet sequences, but then hand off to a generic module to perform the actual sequence itself? This new *transmission* layer module could then be reused in other MAC protocols.

Now that basic packet sending/receiving, time synchronisation, and the sending of particular packet sequences have all been separated out, the λMAC layer only needs to contain time management: that is, the maintenance of the knowledge about what time is a good time to send packets; allocating blocks of time as required by the *transmission* layer modules in order to allow them to both send and receive data; and switching the radio on/off as appropriate for the individual protocol. A block of time is simply an interval during which the radio is exclusively handed over to a particular transmission module which has previously requested that the λMAC layer give it *n* milliseconds in order to send a packet sequence; conversely time blocks are also allocated when a packet comes in informing the local node that another node will be performing a packet sequence for a short period from now and so the local node should not give the radio over to other transmission-layer requests for that time. Note that when we talk about a good time to send a packet, we imply that this is a time with a high probability that the destination node will be able to receive the packet, which is information that the λMAC layer needs to keep track of as part of its time management role.

2.3 Design conclusions

Given our new formulation of a MAC protocol stack, we redefine the required modules and connections as follows (see Figure 1 for an overview of how these interact):

- Packet layer - responsible for the actual sending/receiving of a packet, radio state changes (Rx/Tx/sleep) and for providing carrier sense functions (for CSMA-based λMAC protocols). The sending/receiving radio state here is "dumb" - it does things right now, with no options for delay or smart decisions considered. In the case of byte-based radios, we also provide a platform-specific byte interface layer (which can only be talked to via the Packet layer), and for packet-based radios the Packet layer is a slim layer on top of the existing hardware capabilities. This allows us to abstract away from the differences of these two paradigms, as only packet-level information is required for the λMAC implementation.

- Network Time layer - responsible for storage and generation of time-synchronisation information to provide event synchronisation, e.g. frame timers. This is not required by all λMAC layers and therefore optional. However, there are several reasons to include the Network Time layer here. First, time-synchronisation information is useful to a large quantity of WSN MAC layers, due to the energy savings that can be made if nodes are able to agree when transmit/receive periods should be. Second, the information is potentially useful to other layers. Finally, doing accurate timing information above the MAC layer, given the uncertainty of timing in at least the 10-msec range above most WSN MAC protocols is very difficult. For these reasons, we designed the Network Time layer as a general service to the entire application stack.

  The Network Time layer can, through its placement in the stack, override the λMAC layer's decisions on when to keep the radio in receive mode in order to do neighbour discovery. The overrides will make the radio be in receive mode more than it would be normally off, but will not switch the radio off when the MAC wishes it to be on, or switch the radio from transmit to receive mode (or vice versa).

  The Network Time layer here provides the same interfaces as the Packet layer in addition to the Network Time interface in order to allow insertion of time-synchronisation headers in packets on their way to/from the Packet layer itself. For more information, see Section 2.5.

- λMAC - responsible for time management. Allocates time blocks in response to requests from the Transmission layer, at times that are considered to be "good". Talks to the Packet/Network Time layer in order to send its own control packets, as well as for carrier-sense checking in order to determine if the radio medium is free for sending (for CSMA-based λMAC layers), and decides when to switch the radio on and off. Passes packet send requests/receive events from/to the Transmission layer to/from the Packet/Network Time layer, possibly adding and removing headers on said packets along the way. Given the roles now allocated to other layers, the λMAC layer will be considerably smaller than a traditional MAC layer.

- Multiplexer - (de-)multiplexer to allow for the λMAC to only provide a singular interface to the upper modules, yet talk to many Transmission layer modules.

- Transmission layer - contains the Unicast, Broadcast and other application-level primitives of this nature. Requests time blocks from the λMAC layer as required, and then sends packets during the allocated time. The transmission layer is fully explored in Section 3.

- Demultiplexer - provides the standard MAC interface to the application and hands off the packets from the application to the appropriate Transmission layer module.

There is one limitation on the choice of MAC protocol for the λMAC layer: packet exchanges should performed in a contiguous block of time. To allow optimal flexibility, this block of time should be usable for both sending and receiving by a node. This is possible for all contention-based MACs, and for some TDMA-based MACs, but this may require some alterations to the protocols.

2.4 λ Interfaces

As we wish to define common connections between the λMAC and Transmission layers to enable reuse of the Transmission modules, we need to define some standard interfaces for these connections. We use here the terminology of nesC [6] to provide common semantics, and also because our reference implementation is implemented on top of TinyOS [9]. There should however be no obstructions to implementing this with any other WSN software platform.

All modules in the λMAC stack use the standard TinyOS Send and Receive interfaces for passing messages up and down the stack. Furthermore, we define the AllocateTime interface, which defines the necessary functionality for a Transmission module to allocate time from the λMAC layer. In general, a Transmission level module requires a single instance of the AllocateTime interface, plus one instance of the Send and Receive interfaces per message type (e.g. the Broadcast module requires a single Send/Receive interface set, and a standard Unicast requires 4 Send/Receive interface sets (RTS, CTS, DATA and ACK)). The λMAC layer, however, only needs to provide a single instance of each of AllocateTime and Send/Receive to the Multiplexer module. The Multiplexer module provides generic multiplexing services to create a parametrised interface to both AllocateTime and Send/Receive, thus enabling the capability for multiple Transmission layer modules to be enabled in a single stack, without having to deal with the multiplexing complexity in each λMAC layer.

Individual Transmission layer modules could be implemented using a single Send/Receive interface set per module. However for modules that require multiple message types (e.g. Unicast), the implementers of the Transmission modules would have to both add their own type field to the sent messages, and do de-multiplexing of the different types at the receiver side. As the Multiplexer module allows for multiple instances of Send/Receive already (in order to allow multiple Transmission modules in a single application), the Transmission layer protocol design can be simplified by using multiple Send/Receive interface sets, and this also removes the necessity for the overhead of an additional type field.

The interface between the packet layer and the λMAC layer is much simpler, and as this is more in keeping with traditional WSN MAC design, we will not cover it in detail here. The Packet layer must provide interfaces to change the radio state (Tx/Rx/sleep), and also to send/receive packets using the commands and events of the Send and Receive interfaces. For a CSMA-based λMAC layer, the Packet layer will also require an interface to carrier-sense operations. As we stated before, the Packet layer is "dumb" in the sense that all of the smart decisions regarding when to send, to listen and to sleep are decided by the particular λMAC layer in use.

2.5 Network Time

In order for many MAC protocols to operate correctly, they require a mechanism to synchronise nodes in order so that differing nodes can agree on events happening at the same time e.g. synchronised awake times. Additionally, placing this between the packet layer and the λMAC layer also allows us to integrate time synchronisation information into each outgoing packet, thus reducing the need for additional control packets whenever we are sending other data packets. However, as we wish the Network Time layer to not override λMAC-layer decisions about when to send packets, in the case where we do not have a sufficient rate of outgoing packets to guarantee time synchronisation the Network Time layer will send a packetRequired event (Table 2) to the λMAC layer requesting that it send a packet "soon" in order to maintain time synchronisation.

In keeping with the idea of the Network Time layer as a generic layer, and also because we wish to provide information to modules other than the λMAC layer, we need to define the timing information appropriately. We started with the work of Li et al [13] on the *global schedule algorithm* (GSA), but then expanded it one step further. In GSA, nodes keep track of how much time has passed since they were switched on, and add this information to their outgoing packets. If a node sees an incoming packet with a greater age than the local age, the local age is updated to be the same as the incoming packet, thus allowing the network to converge towards a shared timing value based on the oldest (first switched-on) node's age. Note that although we chose to use a synchronisation algorithm which results in a global network synchronisation, this is an implementation decision, not a decision resulting from the requirements of the design.

Name	Type	Arguments	Return	Function
requestBlock	command	uint16_t msec, am_addr_t to	error_t	Request a period of *msec* milliseconds to send a message to *to*, which can be the broadcast address. A return value of FAIL indicates a persistent failure i.e. the requested period is too long.
requestSafeBlock	command	uint16_t msec, am_addr_t to	error_t	As requestBlock, but asks the λMAC layer to trade off increased latency for a better chance of success. Should only be called after a previous block has run to completion, but has completely failed i.e. no response has been received from any other nodes at all.
startBlock	event		void	Called on the successful start of a period. Always corresponds to the last call to requestBlock or requestSafeBlock.
cancelBlock	command		void	Cancel a previous requestBlock or requestSafeBlock request. Should only be called before the requested block has started.
endBlock	event	bool myStart	void	Called at the end of a period, where *myStart* indicates whether this was a block started at this node or a block initiated by another node's packet.
sleepRemaining	command		void	Switch the radio off for the remaining length of the AllocateTime period. This is intended for periods when there will be packets in the air, but none of them are destined for this node.
sendTime	command	uint16_t length, bool firstPacket	uint16_t	Query how long a packet of *length* bytes should take to be transmitted with the relevant headers. *firstPacket* indicates if the packet will be the first packet in the packet sequence, which can be used by protocols using low-power listening techniques to determine whether a long preamble will be used.

Table 1: AllocateTime interface

Name	Type	Arguments	Return	Function
packetRequired	event		void	Notify the λMAC layer that it should send a packet soon to maintain synchronisation.
sendDummySyncPacket	command		void	Request the time synchronisation module to send an empty packet for the purpose of time synchronisation, when the λMAC layer has no useful data to send.
isSyncPacket	event	message_t msg	bool	Ask the λMAC layer whether the message *msg* was useful to maintain time synchronisation. Only packets that are likely to be received by all neighbours should be indicated as such.

Table 2: TimeSync interface

In the original implementation of GSA, schedule information (time since last frame timer) was also distributed with the *age* value in order to calculate the correct current frame timer for the MAC protocol. In the λMAC framework, we have a separate TimeSync module, which is used by the λMAC framework as a storage location for the current local value of the *age* value, and a separate FrameTimer module which derives frame-timer events from this age value. The FrameTimer module provides periodic frame timers (of variable length up to $(2^{32} - 1)$ms) to all application modules that require this capability (not just λMAC layers that need it) - e.g. for experiments that require an entire field of nodes to make a measurement at the same time (a commonly wanted requirement for many biological experiments being proposed for sensor networks). We do this by taking the *age* value modulus the frame length to provide a frame timer every time *(localAge mod FrameTime) = 0*. This allows the creation of multiple frame timers for different application modules, while only requiring synchronisation on the single *age* value.

All of the periodic frame timers also have an allowable "fuzz" value: if because of updating the local clock, we jump over the time when we should have fired a frame timer, but we jump over by less than the "fuzz" value, then we fire the timer anyways. This bounds the acceptable jitter in the frame timer event. In the event we jump too far over the event point, the safest approach is usually just

Name	Type	Arguments	Return	Function
setFrameTime	command	uint32_t ms, uint32_t fuzz	void	Set time between frame timers (*msec* milliseconds) as well as allowable fuzz time (*fuzz* milliseconds)
clearFrame	command		void	Stop this FrameTimer
frameStart	command		uint32_t	Retrieve the local time at which the current frame started.
frameIndex	command		uint32_t	Retrieve the time in milliseconds since the start of the current frame.
globalTime	command		globaltime_t	Retrieve the current value of the network time.
frame	event	sanitystate_t sanity	void	Indicates that the new frame has started. *sanity* indicates whether the network time layer has fully synchronised or is still in one of the start-up phases.
frameGuaranteed	event	sanitystate_t sanity	void	Indicates that *fuzz* milliseconds have passed since the start of the frame.
frameSkipped	event		void	Indicates that one or more frame events have been skipped due to network time re-synchronisation.

Table 3: FrameTimer interface

to skip the event entirely and wait for the next one (e.g. not doing a TDMA frame that is out of sync with the other nodes). This allows us to cope with small changes in the network clock due to varying speeds of clocks on different nodes.

3 Transmission layer modules

In this section we will look at how to implement Transmission layer modules, with a focus towards the standard set of WSN Transmission modules on top of the λMAC layers i.e. the set of functions that would be expected from a standard MAC protocol. An exploration of what can be done with non-standard modules is in Section 6.

3.1 Notes on Transmission module design

Before we go into a more detailed look at how to build basic Transmission modules, a number of features of the AllocateTime interface should be noted:

- The point of an AllocateTime period is to grab time in order to send packets, with a reasonable guarantee about our neighbours being in a state where they are able to receive our packets. A node does not need to be in an AllocateTime period for any other purpose.
- The AllocateTime period (as marked by a startBlock() event) is only started when a certain level of guarantee can be given that the radio medium will be at least relatively quiet. In CSMA-based protocols this will be done via a carrier sense mechanism of random length (to resolve contention issues between multiple nodes wishing to start AllocateTime), and in TDMA-based protocols this is guaranteed by the time slot mechanism.
- The λMAC layer will piggyback information about the remaining AllocateTime period on outgoing packets, in order to place other nodes into the AllocateTime state as well. If the MAC protocol implemented uses a slot structure, this may be avoided as the allocation will always be for an entire slot.
- Once an AllocateTime period is started, it cannot be stopped. This is because of the difficulty of telling other (possibly asleep) nodes of this change of plans. A node can be told to go to sleep for the rest of the time period however (via sleepRemaining()).

3.2 Broadcast

Broadcast is simply implemented on top of a single set of Send/Receive and AllocateTime interfaces. Sending is implemented as follows:

1. Call requestBlock() for sendTime(packet length) milliseconds.
2. On startBlock(), call send().
3. On sendDone(), call sleepRemaining().
4. On reception, call sleepRemaining() as no other packets will be forthcoming in this period. The receiving node can determine the remaining length of this period from the message it received.

3.3 Unicast

Unicast is somewhat more complicated than Broadcast, partly because it can have variants both with and without RTS/CTS. For the case with RTS/CTS, an example implementation runs as follows. During the initialisation of this module, we set *control_length* to the value of sendTime(0), as this is the length of a control (RTS, CTS or ACK) packet, because they contain no data, only MAC headers.

To send a packet, we first calculate *packet_time* as sendTime(packet length) + 3**control_length* plus some platform-dependant allowance for processing and radio state transition delays. We need 3 *control_length* intervals for the RTS, CTS and ACK packets. We then call requestBlock() with *packet_time*. On startBlock() (as we have a reasonable guarantee about the time slot, so we can start immediately), we start to cascade through the RTS-CTS-DATA-ACK sequence i.e. we send an RTS packet using send(), wait to receive a CTS, then send a DATA packet with send(), then wait to receive the ACK. After receiving the ACK packet, we tell the λMAC layer to sleep for any remaining time in the block.

At the receiver node, we first see a receive() with an RTS packet. If the RTS packet is destined for the receiving node it sends a CTS packet. Then, the receiver waits for a DATA packet, sends an ACK packet and calls sleepRemaining() (in order to go to sleep for any remaining left over processing time). Other nodes that are not the destination for this Unicast sequence will go to sleep by calling sleepRemaining() after receiving the initial RTS packet.

This is a simplified description for an example Unicast module, and our complete implementation includes retries for lost/missed packets. However, it gives a flavour of how Unicast can be implemented on top of the λMAC layer.

4 Integrating existing MAC types

Now that we have shown how we intend to split up existing monolithic MAC protocols into a more generic and reusable stack (Section 2.3), and described how that stack works (Sections 2.4, 2.5 and 3), we need to go back and show that all of this can work with existing MAC protocols.

We divide WSN MAC protocols into 3 groups; dividing first into continual listening vs. scheduled, and then further divide scheduled listening protocols into how they decide when to send - carrier sense vs. scheduled (see Figure 2 for a diagrammatic view of this). We have implemented a protocol for each of the groups to demonstrate the flexibility of the λMAC framework. For the continual listening group we implemented B-MAC [16], the standard TinyOS MAC protocol. For the scheduled listening with carrier-sense based send timing group we implemented both T-MAC [3] and Crankshaft [7]. In the scheduled listening and sending group we implemented LMAC [10], a TDMA protocol. We believe that by showing that these protocols can be implemented with the λMAC framework, and by providing data from experiments on our testbed using these protocols, we adequately demonstrate that the λMAC framework is suitably generic to serve as a base for implementing a large portion of currently proposed WSN MAC protocols.

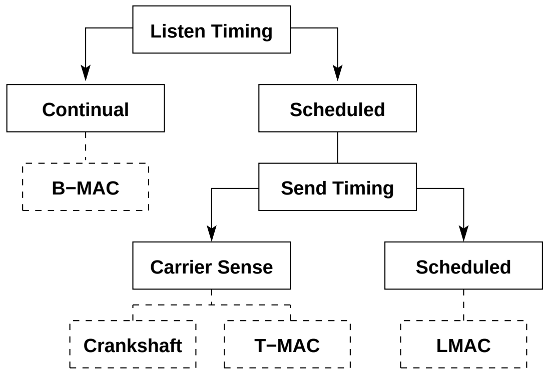


Figure 2: WSN MAC protocol division.

In the remainder of this section we describe how we implemented the distinguishing features of the four implemented protocols. We will also show how the concept of time allocation maps to the scheduling mechanisms in the selected MAC protocols.

4.1 λB-MAC

B-MAC is a prominent example of the continual listening group, which uses the Low Power Listening (LPL) technique to save energy. The LPL technique requires periodic sampling of the medium, and long preambles on the first message of a message sequence. The periodic sampling of the medium is not directly supported by the λMAC framework, but can be easily implemented using timers and the carrier-sense primitives provided by the packet layer. The packet layer also provides an interface to change the preamble length at run-time, such that the long preambles required by B-MAC can be generated.

λB-MAC should also provide the same contention resolution mechanism as B-MAC. This means it can start sending at any time, provided it performs carrier-sense and random back-off first. However, in the case of λB-MAC, once a node determines that it has won the con-

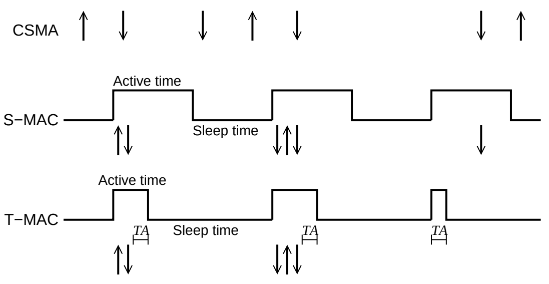


Figure 3: T-MAC.

tention it does not start sending itself, but instead signals the requesting transmission module that its block has started through the startBlock() call. λB-MAC will then wait for the block to end. The transmission module will in the mean while ask the λB-MAC protocol to send a message for it. λB-MAC appends its own header, after which it will pass the message on to the packet layer immediately. When transmission modules are used that mimic the packet sequences used by the B-MAC protocol for broadcast and unicast, the behaviour of the MAC stack with the λB-MAC protocol will be as described for the B-MAC protocol.

A final feature of the B-MAC protocol is that it allows on-line tuning of the sampling interval by higher layers in the protocol stack. Although we have not implemented this feature it could easily be added as an additional interface implemented by the λB-MAC protocol itself. This is much like the monolithic implementation which also has an additional interface, next to the standard MAC protocol interface, which allows such tuning.

4.2 λT-MAC

T-MAC is a CSMA-based MAC protocol, derived from S-MAC [23], but with adaptive duty cycling. The adaptive duty cycling is based on the idea of going to sleep shortly (*TA* milliseconds, defined by the length of the contention window, the time needed to receive a minimal packet, process it, and send another minimal packet) after the last "interesting" event - which can be a message going out, another message coming in or the periodic firing of a frame timer every so often (see Figure 3). The frame timer length is a trade off between energy efficiency (with longer sleep times between awake periods) and latency (due to the length of sleep before the next time we can send a packet).

We used the frame timers from the Network Time layer to assume a lot of the complexity from T-MAC. A significant part of the code of an existing T-MAC implementation was dedicated to schedule synchronisation (including discovery of new schedules); a role now subsumed by the Network Time layer. On a requestBlock() call, λT-MAC places the requested amount of time into a variable. When T-MAC would normally check if it has a packet to send, λT-MAC instead checks if a request was made, and if so requests that the packet layer do a carrier sense check. If the carrier sense returns an idle radio medium, then startBlock() is called and λT-MAC waits until the end of the AllocateTime period before doing anything else. Send and receive calls pass almost uninhibited through the λT-MAC layer. Notably, the send is not delayed waiting for anything else to complete, but is passed through to the packet layer as rapidly as possible.

If λT-MAC gets a packetRequired event (a request from the Network Time layer for a packet to be sent), λT-MAC sends out a Sync packet - a packet with no actual data payload, and only containing timing information in order to maintain the inter-node time synchronisation.

4.3 λLMAC

LMAC [10] is a TDMA-based MAC protocol, aimed at giving WSN nodes the opportunity to communicate collision-free, and at minimising the overhead of the physical layer by reducing the number of transceiver state changes. The LMAC protocol is self-organising in terms of time slot assignment and synchronisation, starting from a sink node (specified by the application). Upon start-up, the sink node sets a frame schedule and chooses the first slot in the frame as its sending slot. Next, one-hop neighbours receiving the sink's transmissions, choose their sending slots based on the frame schedule of the sink node. This is then repeated for all next-hop neighbours. When an application wants to send a message, LMAC delays the transmission until the start of the node's next sending slot.

We created a TinyOS implementation of λLMAC based on the protocol description and the OMNeT++ code available from the LMAC authors. For time synchronisation between the nodes, we used the Network Time layer, and so were able to use a frame timer to determine the start of each slot. This way, all nodes agree on the exact start time of all slots. When using a frame timer to determine only the start of each LMAC frame, intermediate clock updates during the frame may lead to inaccurate start times of slots near the end of an LMAC frame.

Although λMAC supports sending multiple packets in a single slot, in LMAC it is only possible for a node to transmit a single message per frame. The authors suggest gluing together multiple messages to the same destination to prevent high latency, but this suggestion is not implemented in the available OMNeT++ program code. To make our results comparable to the OMNeT++ implementation we had available, we did not implement this feature.

On a requestBlock() call, λLMAC sets a flag indicating that there is a packet waiting to be sent at the node's next time slot. During its time slot, a node will always transmit a packet. If a node has no data to send, an empty Sync packet is sent to keep the network synchronised and to keep a claim on the slot. Otherwise λLMAC signals startBlock() and waits until the end of the time slot to call endBlock().

Since a TDMA-based MAC-protocol does not need the full Unicast RTS/CTS/DATA/ACK sequence to keep other nodes from transmitting at the same time, we created a Unicast module that only sends the DATA packet. As the TinyOS message header already contains information about destination node and packet length, this information was removed from the LMAC-specific header.

4.4 λCrankshaft

The Crankshaft protocol [7] uses the frame and slot structure of TDMA protocols like LMAC, but instead of scheduling the senders to eliminate contention all together, it schedules receivers to limit contention. The advantage of the receiver scheduling over the sender scheduling is that it does not suffer from the over-provisioning that is associated with TDMA protocols for WSNs.

To achieve maximum energy efficiency, Crankshaft combines techniques from several different protocols. For example the synchronised channel polling mechanism from SCP-MAC [24] is used to allow receiving nodes to sleep most of their slots when no message is sent to them. This requires preamble sampling and long preambles, just like λB-MAC, but also the network synchronisation. The synchronisation is provided by the Network Time layer.

As far as implementation using the λMAC framework is concerned the most distinguishing feature is that the correct moment to contend and send depends on the destination (owing to the receiver scheduling). This feature requires that when a block is allocated, the destination is also made available to the λCrankshaft implementation. Crankshaft uses a simple DATA/ACK message sequence which is readily implemented using our stock Unicast Transmission module.

Protocol	Parameter	Value
B-MAC	Sleep time	85 msec
LMAC	Number of slots	32
	Slot length	50 msec
Crankshaft	Number of unicast slots	8
	Number of broadcast slots	2
	Slot length	48 msec

Table 4: Protocol parameters as used during the experiments. λMAC and non-λMAC versions use the same parameter values. Time values are in binary milliseconds.

5 Testing

We performed a series of tests comparing the λMAC versions of B-MAC, T-MAC, LMAC and Crankshaft to other implementations. In the case of λB-MAC we compare against the standard TinyOS B-MAC implementation. Although we have a TinyOS 1 implementation of T-MAC, it does not perform to an acceptable level to allow proper comparison. Therefore, for T-MAC, as well as for LMAC and Crankshaft we had to resort to comparing against simulation results. We use an enhanced version of our OMNeT++ based MAC protocol simulator used in earlier work [12]. We do not compare against TOSSIM, as our goal is to check against a different implementation of the same protocol, so as to verify the correct operation of our λMAC implementations.

All real-world experiments are performed on our 24 node testbed [8]. The nodes in our testbed are mica2 class nodes, with Atmel ATMega128 processors and CC1000 radios. Table 4 show the parameters used for the different protocols for all our tests. The simulator has been set to match both the hardware and software characteristics as closely as possible. Both simulations and real-world experiments are performed five times.

5.1 Testbed Results

We tested the implementations of λB-MAC, λT-MAC, λLMAC and λCrankshaft on our full testbed. At the default transmit power setting, our testbed is a single cell network of 24 nodes. First of all we perform a benchmark test, in which two nodes A and B communicate with each other while the other nodes are sending broadcast packets. Nodes A and B send at a fixed rate of one message per two seconds, while the message rate of the broadcasting nodes is varied. We measure the packet success rate as the success rate for packets between A and B, ignoring all other packets. Figure 4 shows the result for the λB-MAC and λT-MAC, as well as the results for the standard

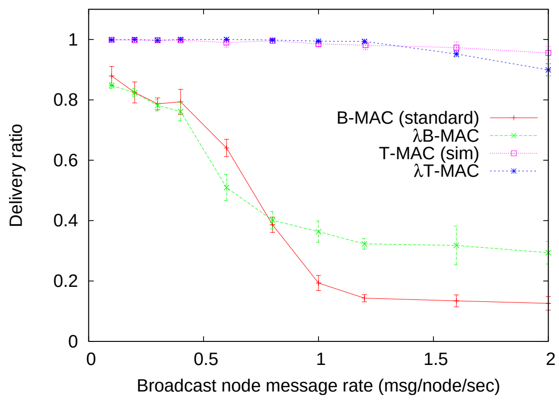


Figure 4: Unicast communication test between two nodes surrounded by broadcasting nodes.

TinyOS B-MAC protocol on our testbed and simulation results for T-MAC. We have omitted the graph with the results for λLMAC and λCrankshaft because, as expected, they show (almost) 100% reception for all broadcast-node message-rates.

As can be seen in the figure, the standard B-MAC implementation outperforms the λB-MAC implementation by a small margin for low contention. However, as the contention increases, λB-MAC starts to outperform the standard B-MAC approximately 17% percentage points. In our original test runs the delivery ratio of the standard B-MAC implementation dropped to almost zero due to a bug in the dynamic noise-floor algorithm, which caused B-MAC to consider all transmissions as background noise when channel utilisation was high. The results shown in this paper are the results with the bug fixed. The remaining performance gap is likely due to a failing clear channel assessment that causes collisions that λB-MAC avoids.

The results for the T-MAC protocol show that the λT-MAC implementation shows a similar curve as the simulation-based T-MAC implementation. As the simulator uses a simple SNR based radio model with free-space propagation, it is expected that the simulation results are better than the real-world results. However, we do expect that both curves show similar decay in delivery ratio, as we see in the figure.

As a second test, we let all the nodes in the testbed send to a single node. The results are plotted in Figure 5. The top graph shows the same protocols as before. Again the simulated T-MAC implementation outperforms the λT-MAC implementation, but shows similar drop in delivery ratio.

The λB-MAC implementation and the standard B-MAC implementation show almost identical delivery ratios. Even though the clear channel assessment in the standard B-MAC implementation is not optimal, in this scenario it is not so much of a problem. If two messages partially overlap, the second message will still be received (cf. Figure 6). For the unicast test this counts as one successfully received message. In the previous test such collisions cause problems if the first message is a unicast message and the second message is a broadcast message, as only the unicast messages count for the delivery ratio. This explains why for the unicast test the standard B-MAC implementation does not suffer a large performance hit at high contention.

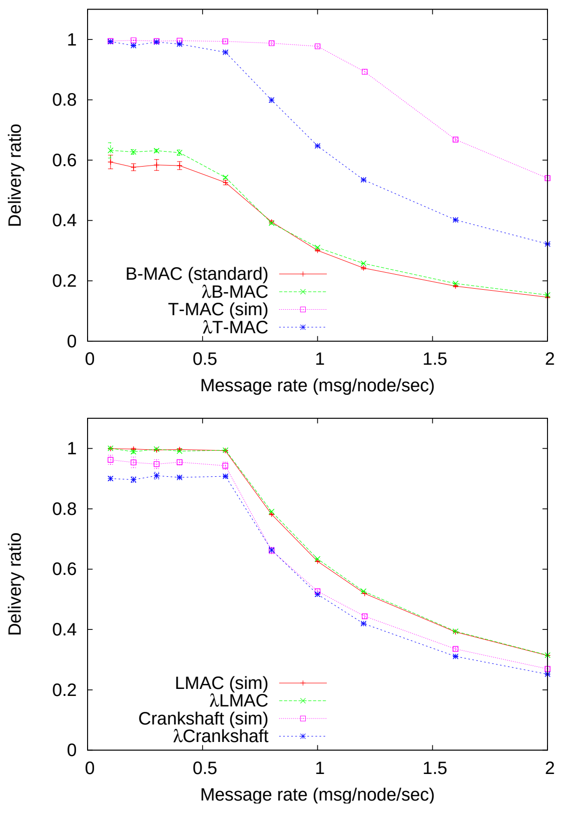


Figure 5: Unicast test.

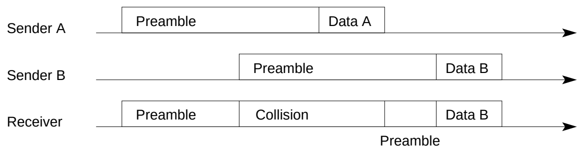


Figure 6: Two partially overlapping B-MAC packets cause a single packet to be received.

The bottom graph in Figure 5 shows the results for

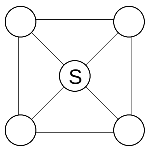
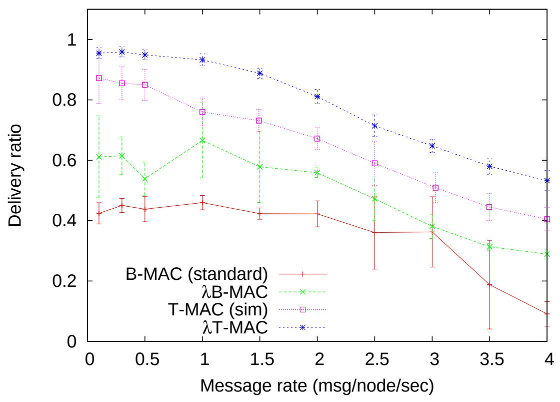


Figure 7: Hidden-terminal setup.

λLMAC and λCrankshaft. For the LMAC protocol the graphs for the simulated LMAC and λLMAC are virtually the same. As the LMAC protocol does not use a contention resolution algorithm, the only impact the channel model has is in message detection and correct transmission. Because our testbed is a single cell network in the default configuration, the chance of incorrect transmission is minimal. The only factor determining the delivery ratio of the LMAC protocol is therefore the maximum throughput. As the simulator was set to match the the real nodes as closely as possible, it is no surprise that there is hardly any difference in delivery ratio between simulation and real life.

In the Crankshaft protocol results we see a similar sudden drop in performance as we see in the LMAC protocol result. Again this is caused by bandwidth limitations inherent in the protocol. The Crankshaft protocol does use a contention resolution algorithm which relies on a correct clear channel assessment and low contention to get a 100% delivery ratio. Because the test is set up in such a way that all nodes generate their messages at the same time, there is always a limited contention. The simulated version has a perfect clear channel assessment, which explains the slightly better performance compared to the λCrankshaft protocol.

Finally we performed a small hidden-terminal test with five nodes. The setup is shown in Figure 7. Nodes on opposite ends of the sink node (S) cannot directly communicate. For carrier-sense based protocols, this situation presents problems because the "hidden nodes" are not aware of each others transmissions and thereby cause collisions. We chose this simple setup because it can be replicated in our simulator. Figure 8 shows the results for the different protocols. The figure shows that the carrier-sense based protocols λT-MAC and λCrankshaft perform better than their simulated counterparts. This is most likely due to the fact that even though the "hidden nodes" in the testbed cannot successfully receive messages, they can in some cases detect the ongoing transmission through signal strength measurements. As in the first experiment, λB-MAC outperforms the standard B-MAC due to better clear-channel assessment. Because the LMAC protocol does not depend on carrier sensing, the λLMAC protocol



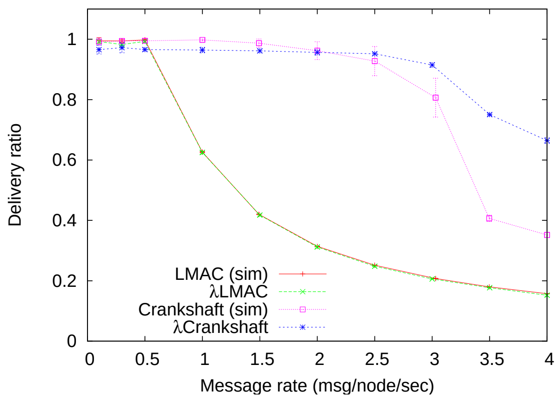


Figure 8: Hidden-terminal test results.

and the simulated LMAC protocol again show very similar performance.

From these experiments we conclude that the λMAC implementations of the tested protocols perform as expected.

5.2 Power Test

To further demonstrate the correct working of the protocols, we used the power tracing capability of our testbed. As an example, we include a trace of a λT-MAC packet exchange in Figure 9. The only difference with power traces from previous implementations like for example in [3] is that nodes briefly switch off the radio after the exchange is complete. This is an artifact of our unicast transmission module, which tells the MAC layer it may go to sleep once it has sent/received an ACK message. As other nodes should not be sending until the block is complete this has no impact on further message exchanges. After the message exchange we can see the wait for activity before the radio is turned off, which is characteristic of the T-MAC protocol. We do not show the traces for

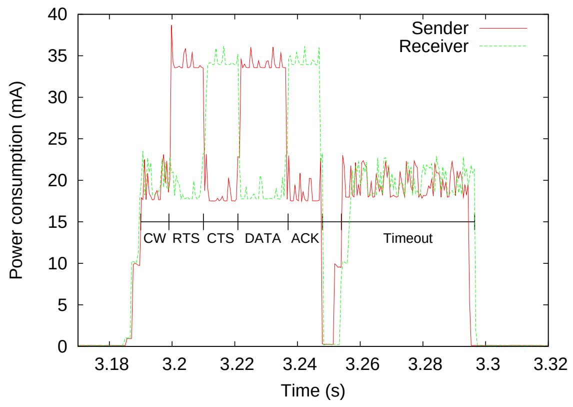


Figure 9: Power trace from a run using the λT-MAC protocol.

Protocol	SLOCCount
B-MAC	499
λB-MAC	324
T-MAC (TinyOS 1)	1090
λT-MAC	330
λCrankshaft	523
λLMAC	426
λMAC transport & multiplexing	775
λMAC time synchronisation	431

Table 5: Lines of code in the implementation of the tested protocols.

the other protocols as they provide limited value over the λT-MAC trace.

5.3 Code size

Next, we examine the lines of the code for the different MAC layer implementations. In this case we can use the original TinyOS 1 implementation for T-MAC as a comparison as bug-free operation of the protocol is not required for this comparison. First we look at the lines of code required to implement the protocol. We use this measure, as it is a measure of effort required to implement the protocol.

Table 5 shows the number of lines of code as counted by the SLOCCount utility [22] for the tested protocols. For the standard B-MAC implementation we counted the lines for all the modules above the packet layer interface (i.e. CC1000CsmaP.nc and CC1000ActiveMessageP.nc). The same holds for the TinyOS 1 T-MAC implementation. For the λMAC protocols we only count the parts specifically implementing the functionality of the protocol, and

Protocol	RAM	ROM
B-MAC	164	9114
λB-MAC	249	11664
T-MAC (TinyOS 1)	538 *	17514
λT-MAC	381	16320
λCrankshaft	409	17158
λLMAC	611	16120

* After correction for serial stack and packet-layer buffers

Table 6: RAM and ROM sizes in bytes of the same empty application for different MAC protocol implementations.

not the shared parts of the λMAC framework. Again, this is because we are interested in the implementation effort required for the protocol, not the MAC stack as a whole. The lines of code of the λMAC framework (excluding the packet layer) are listed in the table as well. The time synchronisation is listed separately because it is an optional component. The λB-MAC implementation for example does not use the time synchronisation module.

The λMAC implementations are significantly smaller than there monolithic counter parts. The λB-MAC implementation is 35% smaller and the λT-MAC implementation is 70% smaller. Because the B-MAC protocol does not include any form of time synchronisation, the code size gain obtained by using the λMAC framework is smaller than it is for the T-MAC protocol which does include time synchronisation.

For the λCrankshaft and λLMAC implementations we do not have monolithic implementations. However, what the the table does show is that even a complex protocol like Crankshaft can be implemented within the λMAC framework with relatively little effort.

From the lines of code comparison in we can also see that the flexibility of the λMAC framework and the reduced per-protocol complexity comes at the expense of a larger total lines of code for the MAC stack as a whole. However, lines of code only indirectly translate to RAM and ROM size. Therefore we also compare the RAM and ROM size of a minimal program that includes the MAC layer (see Table 6). Note that contrary to the lines-of-code comparison, the numbers in Table 6 include the size of the λMAC stack.

What is immediately clear is that λB-MAC uses 52% more RAM than the standard B-MAC implementation. The actual λB-MAC module only uses 9 bytes of memory for state variables. The overhead is therefore mainly due to the λMAC framework modules. The most important source of overhead is in the unicast module. The unicast module incorporates a message buffer which is used to send control messages. In the case of λB-MAC it is used as an ACK message. The standard B-MAC implementation has a hard coded ACK sequence which also does not include any information about the message that the ACK was sent in response to. Therefore it can store this sequence (5 bytes) in ROM rather than keeping a 44 byte message buffer in RAM. Other significant overheads introduced by the λMAC framework are bitmaps and tables in the multiplexer (16 bytes), and state variables in the unicast module (12 bytes). Analysis of the causes of the difference in ROM size is thwarted by the inlining employed by the compiler.

The RAM size for the TinyOS 1 T-MAC implementation shown here is after correction. In TinyOS 1, the radio stack and the serial stack cannot be enabled and disabled separately. The serial stack uses approximately 130 bytes of RAM. Furthermore, the packet layer used in the TinyOS 1 T-MAC implementation uses an extra 94 bytes in packet buffers that the TinyOS 2 packet layers do not. Including these extra overheads would distort the comparison. Of course the different TinyOS versions already distort the comparison somewhat, but that cannot be avoided. The ROM size cannot be easily corrected for the inclusion of the serial stack due to the inlining performed by the compiler.

Closer inspection of why the TinyOS 1 T-MAC implementation uses more memory than the λT-MAC implementation revealed that the monolithic implementation includes three packet buffers more than λT-MAC. These packet buffers account for 134 bytes, which leaves a difference of only 23 bytes between the two implementations. The remaining difference can be mostly attributed to the different TinyOS versions used, but the exact variables are hard pinpoint exactly.

The overall picture that arises from these RAM and ROM size numbers is that the less complicated the protocol, the larger the overhead incurred by the λMAC implementation. However, the overhead is not very large. Even in the case of the relatively simple B-MAC protocol, the overhead in RAM size is limited to only 85 bytes. For the more complex T-MAC protocol, the overhead is negligible. As ROM size is generally not a resource bottleneck, the small absolute increase in ROM size is not a problem.

5.4 CPU-cycle Overhead

Generally, the flexibility offered by using a framework rather than creating a monolithic implementation comes at a price. As we have seen in the previous section, using the λMAC framework incurs an overhead in RAM and ROM use. In the context of sensor networks another important factor is energy consumption. The two main energy consuming parts of a sensor node are the CPU and the radio. In this section and the next we therefore quantify the overhead of the λMAC framework with respect to CPU and radio use.

Experiment	**Overhead**
Idle	275.2%
Idle (adjusted)	-5.0%
Send	60.7%
Send (adjusted)	-0.2%

Table 7: CPU cycle overhead for λB-MAC, with and without correction for radio switching optimisation.

Because we have two TinyOS implementations of B-MAC, we can perform detailed comparisons. First of all, we compare the CPU overhead for the λB-MAC protocol compared to the original B-MAC protocol implementation. We use the Avrora emulator [21] to get accurate CPU cycle counts. We have tested two situations: first, we compare the cycles used when there is no communication taking place. Second we look at the situation where a node is sending. Both tests are the result of taking the cycle count over a 20 second period. In the send test, the node was sending one message per second. Unfortunately, Avrora does not simulate the RSSI output of the CC1000 radio, which means that the channel polling will always detect an idle channel. As a result, we cannot provide CPU overhead numbers for a receiving node. Table 7 summarises our results.

The CPU cycle test results show a significant overhead for λB-MAC in the idle test, and to a lesser extent in the send test. Closer examination of the results showed that the original B-MAC employed an optimisation in the radio switching. Instead of switching the radio on completely for channel polling, it only switches the radio on to the state where RSSI measurements can be taken. Because the timing during the radio switches is done through busy waiting, only performing part of the radio state switch uses fewer cycles. Although it is currently not possible to tell the packet layer used in the λMAC stack to do this partial radio state switch, we can calculate the extra time used in busy waiting to do a complete switch. If we subtract the difference between the busy waiting time required for a complete state switch versus a partial state switch, we can see what would be the CPU cycle overhead if the packet layer used could perform the optimised radio switching. In the table these numbers are shown as “adjusted”. The results show that λB-MAC could in principle be as CPU cycle efficient as the original B-MAC, indicating that the λMAC framework does

Protocol	Average Rate
B-MAC	9.0 (msg/sec)
λB-MAC	8.8 (msg/sec)

Table 8: Average maximum broadcast rate for B-MAC and λB-MAC.

Protocol	Sender	Receiver	Average
B-MAC	15.4%	7.4%	11.4%
λB-MAC	13.6%	8.0%	10.8%

Table 9: Average radio duty-cycle for one node sending unicast messages to another node, using B-MAC and λB-MAC.

not introduce any significant CPU-cycle overhead.

5.5 λB-MAC Micro Benchmarks

Next we performed two micro benchmarks with the two TinyOS B-MAC implementations. First of all we setup two nodes, one receiver and one sender node. The sender simply tried to send as many broadcast packets as it can in 50 seconds. Both the original B-MAC and the λB-MAC implementation approach the theoretical maximum of approximately 10 messages per second, and there is only a small difference between the two implementations (cf. Table 8).

Second, we measured the average radio duty-cycle for the same two nodes, sending unicast messages from one node to the other at a rate of one message per second. Table 9 shows the average radio duty-cycle over 590 seconds for the sender, the receiver, and the average of both nodes. Again, both the standard B-MAC and λB-MAC implementations perform very much the same.

6 Further Transmission modules

In this section we look at some Transmission modules that can be implemented on top of the λMAC layer that would not be considered part of a standard MAC protocol, but would provide useful additional primitives for other applications. Notably, these would be non-trivial to add to most normal MAC protocols, as we would either have to try and build them out of Broadcast and Unicast operations, which would be significantly sub-optimal; or we would need to rebuild the MAC entirely. Our modular approach makes these additions not only possible, but relatively easy.

6.1 ExOR

ExOR (Extremely Optimistic Routing) is a "one send, many replies" approach to reliable multicast for routing protocols. It was first explored by Biswas and Morris [1], and an extended version was proposed in the Guesswork routing protocol [15]. Both variants can be implemented on top of the AllocateTime interface, but would require significant effort to implement inside existing MAC protocols.

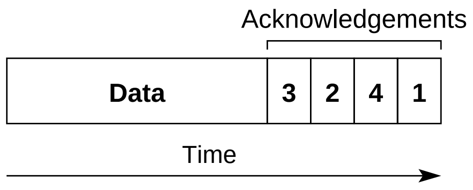


Figure 10: Example ExOR packet time-line

An ExOR sending node sends a packet that not only contains the data for the packet, but also a list of other nodes that should respond (in the order that they are meant to respond in). Every node that is in the list that receives the packet waits sufficient time for all of the earlier nodes in the list to respond, and then sends an ACK to the sender node (see Figure 10). This can be used for a number of things - for example, implementing Reliable Broadcast, as the sending node knows that all nodes that it receives an ACK from have received the packet; or making a best-effort next-hop transfer in a routing algorithm (by using the ACKs to implement an election mechanism to pick the "best" possible next-hop node that has correctly received the original packet).

From the point of view of implementing ExOR as a Transmission layer, it can be considered as a variant of Unicast, with no RTS/CTS and a series of receiver nodes, all of which need to pause a variable amount of time before sending their ACK packets, and then call sleepRemaining() to avoid overhearing the remaining ACKs.

6.2 Priority Queueing

Another possibility that arises once the λMAC layer has been implemented is an option that has been requested by various applications, namely priority queueing [14, 20] - allowing for messages to be sent out in an order different from that which they were received (either from other nodes in routing scenarios, or events from local sensors). In standard MAC protocols, the "send" method is a fire-and-forget concept i.e. once the "send" has been called, cancelling the message (or even being aware of whether the message is queued or actually being sent right now) is impossible.

Using the λMAC layer, a priority queue can be implemented. By using the cancelBlock() call, a previous request to the λMAC layer can be revoked, after which a different block can be requested. Priority queueing would change the default MAC interface semantics in the sense that currently a MAC protocol would not accept a new packet before the last packet was completely handled. However, when using priority queueing, the MAC stack would have to accept more than one packet. Also, an interface should be provided for the priority queue to determine which packet to send first.

7 Related work

At some levels, the core concepts of λMACs vs. traditional MAC protocols can be viewed as similar to the micro vs. macro-kernel debate in more conventional operating systems. In common with micro kernel design [5, 19], the λMAC layer is able to separate out parts of a WSN application that would normally be considered a very complex part of the system (as both MAC layers and operating system kernels in general tend to be regarded by many programmers as "here be dragons" areas of code), and these separated parts are then able to be altered with a significantly lower chance of affecting the rest of the code base.

Polastre et. al [17] proposed the Sensornet Protocol (SP) that provided a greater level of control to applications wishing to influence the choices made by lower level protocols. Their system created a much more horizontal design for differing levels of an application stack, as opposed to the more traditional vertical design in normal MAC protocols. This design allowed a lot of control at application-level, with the trade-off that an application was able to tweak core parts of the MAC layer that could potentially introduce significant instabilities in the MAC, unless the application was fully aware of how the particular MAC would react to those changes. In the λMAC design, applications have large quantities of control - they can allocate arbitrary blocks of time and do pretty much whatever they like during this time - but in a way that preserves the integrity of the λMAC layer, as it is able to delay AllocateTime requests until it is a "good" (for values of "good" defined by the individual λMAC layer) time for the application to have control. The λMAC separation of control, with most timing control out of the hands of the application designer, allows for cleaner, safer, and simpler design.

The MAC Layer Architecture (MLA) proposed by Klues et. al. [11] also provides a component-based architecture for WSN MAC protocols. MLA provides a set of modules implementing common MAC building blocks like channel polling and TDMA slot handling. The common building blocks identified by MLA are at the level of mechanisms and orthogonal to the role separation proposed in our λMAC framework. For example, different from our work MLA still requires the MAC implementer to manually code the packet exchanges into the MAC specific code, sacrificing flexibility.

Ee et. al [4] attempted similar goals, but for routing protocols. Their approach looked at providing a generic toolkit for building routing protocols, and for creating modules that could be used to piece together protocols, including the possibility of new hybrid protocols built from parts of earlier protocols. Their wish to do provide a toolkit as opposed to a framework design such as we proposed is possibly indicative of a wider variety of options in routing protocol design, as opposed to the relatively small set (time management) that we have identified here for MAC protocols.

8 Conclusions

We set out to redesign and rethink how MAC protocols are designed for WSNs, to create a new and improved design concept, and to modularise common functionality. We have managed to do this, and along the way also provide new capabilities and a refocused take on the role of a MAC in the WSN network stack. The reduction in the roles of a MAC protocol to its core feature of time management, by separating out the Network Time layer to provide node-wide time synchronisation, as well as the Transmission layer modules to allow for clean separation of the logic required for features like Unicast, has given a new look at an old topic.

Through our testing we have managed to show that our initial attempt at a reference λMAC layer (λB-MAC) was able to achieve similar performance, both in terms of data rates and power usage, to a traditionally designed MAC protocol, but with a significant decrease in complexity. Lines of code is not always a good indicator of system complexity, but the reduction of duties required of λB-MAC vs. monolithic B-MAC is. Implementations of LMAC, T-MAC and Crankshaft within the λMAC framework show the λMAC framework's flexibility. As it turns out, the TDMA-based LMAC protocol that we expected to be the most difficult case, was not so hard to implement.

By implementing several significantly different MAC protocols, we have shown that our framework is sufficiently generic to be used by the wider community as a general-purpose MAC creation framework. Especially for experimental platforms, the importance of allowing people to extend existing work without having to reinvent the wheel cannot be overemphasised.

We hope that one of the side effects of our creation of the λMAC framework will be the creation of more MAC protocol implementations for real hardware, as many new MAC protocols are currently only implemented in simulation. We feel that this is important because simulation is a poor guide to how something as low-level and radio hardware dependant as a MAC protocol will behave on real hardware.

References

[1] S. Biswas and R. Morris. Opportunistic routing in multi-hop wireless networks. *SIGCOMM Comput. Commun. Rev.*, 34(1):69–74, 2004.

[2] S. Coleri-Ergen and P. Varaiya. PEDAMACS: Power efficient and delay aware medium access protocol for sensor networks. *IEEE Trans. on Mobile Computing*, 5(7):920–930, July 2006.

[3] T. van Dam and K. Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *1st ACM Conf. on Embedded Networked Sensor Systems (SenSys 2003)*, pages 171–180, Los Angeles, CA, USA, November 2003.

[4] C. Ee, R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica. A modular network layer for sensornets. *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

[7] G. Halkes and K. Langendoen. Crankshaft: An energy-efficient MAC-protocol for dense wireless sensor networks. In *4th European conference on Wireless Sensor Networks (EWSN'07)*, pages 228–244, Delft, The Netherlands, January 2007.

[8] I. Haratcherev, G. Halkes, T. Parker, O. Visser, and K. Langendoen. PowerBench: A scalable testbed infrastructure for benchmarking power consumption. In *Int. Workshop on Sensor Network Engineering (IWSNE)*, pages 37–44, Santorini Island, Greece, June 2008.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGARCH Comput. Archit. News*, 28(5):93–104, 2000.

[10] L. van Hoesel and P. Havinga. A lightweight medium access protocol (LMAC) for wireless sensor networks. In *1st Int. Workshop on Networked Sensing Systems (INSS 2004)*, Tokyo, Japan, June 2004.

[11] K. Klues, G. Hackmann, O. Chipara, and C. Lu. A component based architecture for power-efficient media access control in wireless sensor networks. In *5th ACM Conf. on Embedded Networked Sensor Systems (SenSys 2007)*, pages 59–72, Sydney, Australia, November 2007.

[12] K. Langendoen and G. Halkes. Energy-efficient medium access control. In R. Zurawski, editor, *Embedded Systems Handbook*, pages 34.1 – 34.29. CRC press, 2005.

[13] Y. Li, W. Ye, and J. Heidemann. Energy and latency control in low duty cycle MAC protocols. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, New Orleans, LA, USA, March 2005.

[14] K. Lorincz, D. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, S. Moulton, and M. Welsh. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing, Special Issue on Pervasive Computing for First Response*, Oct-Dec 2004.

[15] T. Parker and K. Langendoen. Guesswork: Robust routing in an uncertain world. In *2nd IEEE Conf. on Mobile Ad-hoc and Sensor Systems (MASS 2005)*, Washington, DC, November 2005.

[16] J. Polastre and D. Culler. B-MAC: An adaptive CSMA layer for low-power operation. Technical Report cs294-f03/bmac, UC Berkeley, December 2003.

[17] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica. Unifying link abstraction for wireless sensor networks. In *3rd ACM Conf. on Embedded Networked Sensor Systems (SenSys 2005)*, San Diego, CA, November 2005.

[18] V. Rajendran, K. Obraczka, and J. Garcia-Luna-Aceves. Energy-efficient, collision-free medium access control for wireless sensor networks. In *1st ACM Conf. on Embedded Networked Sensor Systems (SenSys 2003)*, pages 181–192, Los Angeles, CA, November 2003.

[19] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

[20] M. A. Taleghan, A. Taherkordi, and M. Sharifi. Quality of service support in distributed sink-based wireless sensor networks. In *2nd IEEE International Conference on Information and Communication Technologies: from Theory to Applications (ICTTA '06)*, April 2006.

[21] B. Titzer, D. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *4th Int. Symp. on Information Processing in Sensor Networks (IPSN05)*, April 2005.

[22] D. A. Wheeler. SLOCCount. http://www.dwheeler.com/sloc/, 2001.

[23] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *21st Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1567–1576, New York, NY, June 2002.

[24] W. Ye, F. Silva, and J. Heidemann. Ultra-low duty cycle mac with scheduled channel polling. In *4th ACM Conf. on Embedded Networked Sensor Systems (SenSys 2006)*, pages 321–334, Boulder, CO, November 2006.