Delft University of Technology
Parallel and Distributed Systems Report Series

The λMAC framework: redefining MAC protocols

Tom Parker and Koen Langendoen
{T.E.V.Parker,K.G.Langendoen}@tudelft.nl

Maarten Bezemer
M.C.Bezemer@ewi.tudelft.nl

report number PDS-2007-004

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.ewi.tudelft.nl

Information about Parallel and Distributed Systems Section:
http://pds.ewi.tudelft.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 a modified implementation of T-MAC for TinyOS, and compare it with unmodified T-MAC. We show a 14.8% 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 LMAC (a TDMA-based protocol). Both are demonstrated with data from real-world experience using our 16 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 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 λT-MAC protocol. The same principles will then be shown to work for a λ-layers implementation of LMAC, and we will show more data gathered from this protocol.

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 [17], TRAMA [13], PEDAMACS [2], etc) and CSMA-based protocols (S-MAC [20], T-MAC [16], B-MAC [11], 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 fulfill 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 two things: given an application wishes to send a packet, determine what time this node will be able to send and 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 first can be achieved with a "dumb" packet layer (no queueing, minimal latency, switches radio on/off only when told to); the second 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 [11]).

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

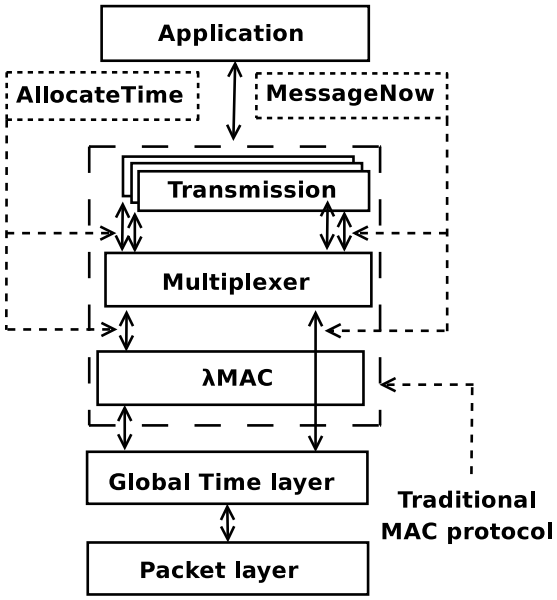

Figure 1: λMAC protocol stack

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 time. Note that when we talk about the 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.

- Global Time layer - responsible for storage and generation of global time information to provide cross-network event synchronisation, e.g. frame timers. This is not required by all λMAC layers, but given that global time 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), that the information is potentially useful to other layers, and 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, we implemented the Global Time layer here as a general service to the entire application stack.
  Responsibility for when to send packets is still the province of the λMAC layer, but the Global Time layer will add its own information on sending.
  The Global Time layer will also override the λMAC layer's decisions on when to stay awake on a periodic basis 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 Global Time layer here provides the same interfaces as the Packet layer in addition to the Global Time interface in order to allow altering of packets (for the purposes of timing information) 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 Global 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 Global Time layer, possibly altering 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 single AllocateTime/MessageNow 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.

There is one limitation on the choice of MAC protocol for the λMAC layer - that it is possible to allocate contiguous blocks of time that can be used 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 [4] to provide common semantics, and also because our reference implementation is implemented on top of TinyOS [6]. There should however be no obstructions to implementing this with any other WSN software platform. The following interfaces are based on the extensions described by the Guesswork routing protocol [10].

We define two separate interfaces, AllocateTime (Table 1) and MessageNow (Table 2). AllocateTime defines the necessary functionality for a Transmission module to allocate time from the λMAC layer, and MessageNow allows the sending and receiving of messages during the allocated time. In general, a Transmission level module requires a single instance of the AllocateTime interface, plus one instance of the MessageNow interface per message type (e.g. the Broadcast module requires a single MessageNow, and a standard Unicast requires 4 MessageNow interfaces (RTS, CTS, DATA and ACK)). The λMAC layer, however, only needs to provide a single instance of each of AllocateTime and MessageNow to the Multiplexer module. The Multiplexer module provides generic multiplexing services to create a parametrised interface to both AllocateTime and MessageNow, thus enabling the capability for multiple Transmission layer modules to be enabled in a single application, without having to deal with the multiplexing complexity in each λMAC layer.

Individual Transmission layer modules could be implemented using a single MessageNow interface 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 MessageNow already (in order to allow multiple Transmission modules in a single

Name	Type	Args	Return	Function
requestBlock	command	uint16_t msec	result_t	Request an AllocateTime period of *msec* milliseconds. A return value of FAIL indicates a persistent failure i.e. the requested period is too long.
requestSafeBlock	command	uint16_t msec	result_t	Request an AllocateTime period of *msec* milliseconds, trading off increased latency for a better chance of success. Should only be called after a previous AllocateTime block has run to completion, but has completely failed i.e. no response has been received from any other nodes at all. A return value of FAIL indicates a persistent failure i.e. the requested period is too long.
startBlock	event	result_t success	void	Called on the successful start of an AllocateTime period, or when a requestBlock is currently impossible. Always corresponds to the last call to requestBlock.
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	uint8_t length	uint8_t	Query how long a packet of *length* bytes should take to be transmitted with the relevant headers
endBlock	event		void	Called at the end of an AllocateTime period
availableBlock	event		void	After a startBlock(FAIL), this will be called next time a good opportunity to call requestBlock occurs e.g. next time the radio returns from a sleep period.
notifyEndBlock	command		void	Notify module on end of block. endBlock events happen by default for locally initiated blocks (i.e. blocks starting with a startBlock() event), but are switched off by default for more non-locally initiated blocks. notifyEndBlock() switches on endBlock events for the currently active AllocateTime block.

Table 1: AllocateTime interface

application), the Transmission layer protocol design can be simplified by using multiple MessageNow interfaces, 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 - similar to the send/sendDone/receive commands and events of MessageNow. 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” - 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 Global 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 within the packet 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 Global 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 Global Time layer will send a phyRequired event (Table 4) to the λMAC layer requesting that it send a packet “soon” in order to maintain time synchronisation.

In keeping with the idea of the Global Time layer as a generic layer, and also because we wish to provide infor-

Name	Type	Args	Return	Function
send	command	TOS_MsgPtr msg, uint8_t length	result_t	Sends a packet right now. Fails if we are already sending something. Should only be called during an AllocateTime period.
sendDone	event	TOS_MsgPtr msg	void	Called on completion of a send()
setAddressFiltering	command	bool enable	void	Enables/Disables automatic destination address filtering for this interface i.e. dropping all incoming packets not destined either for this node or for the broadcast address. Default is not to filter. If filtering is switched on, packets not destined for this node will cause sleepRemaining() to be called in order to avoid overhearing the packet sequence.
receive	event	TOS_MsgPtr msg, uint16_t fromAddr	bool	Called when a message comes in that is not filtered (see setAddressFiltering). Implementations should return TRUE if they wish to stay awake for the rest of the AllocateTime period, and FALSE otherwise.
reservedBytes	command		uint8_t	Number of bytes reserved at the beginning of the data section of the TOS_Msg by lower layers
setPreambleLength	command	uint8_t length	void	Set length of packet preamble to *length* bytes. Defaults to 1 if not called.

Table 2: MessageNow interface

Name	Type	Args	Return	Function
setFrameTime	command	uint32_t msec, uint32_t fuzz	void	Set time between frame timers (*msec* milliseconds) as well as allowable fuzz time (*fuzz* milliseconds)
frameIndex	command		uint32_t	Determine location within the current frame i.e. milliseconds since last frame timer.
globalTime	async command	globaltime_t *temp	void	Get a copy of the current local value of the global timer. May or may not be currently synchronised with other nodes.
frame	async event	SanityState synchronised	void	Frame Timer event. *synchronised* variable indicates current synchronisation level with other nodes: (NOT_)SANE indicates (not) synchronised with other nodes, TX_SANE indicates that this node will be “sane” once a packet has been sent, and that PhyRequired.PhyRequired() has already been sent.
frameSkipped	async event		void	Indicates that one or more frame() events have been skipped due to Global Timer alterations.

Table 3: TimeSync interface

mation to modules other than the λMAC layer, we need to define the timing information appropriately. We started with the work of Li et al [8] 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

Name	Type	Args	Return	Function
phyRequired	event	bool slowneighbour	void	Indicates that a packet (any packet) should be sent "soon". In the case where *slowneighbour* is true, this should be immediately in order to be able to communicate with the neighbour with a significantly younger Global Time value that has just been noticed.

Table 4: PhyRequired interface

on the oldest (first switched-on) node's age.

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. However, TimeSync 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 to skip the event entirely and wait for the next one (e.g. not doing a λT-MAC awake period that is drastically out of sync with other nodes). This allows us to cope with small changes in the global 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 8.

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 MessageNow and AllocateTime interfaces 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.
- 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()).
- Setting setAddressFiltering() is recommended for all protocols that set the destination address to non-broadcast addresses, as this will enable the λMAC layer to reduce the level of calls to the Transmission layer, and will also simplify Transmission layer design. The λMAC layer will also be able to use the transmitted AllocateTime value to avoid overhearing the rest of this packet sequence.
- A receive() event's return value says whether to stay awake for the rest of this AllocateTime period or

not. This is automatically handled using sleepRemaining(), and the Transmission layer will not generally need to call sleepRemaining except in certain special situations (for example, if you wish to receive packets for a short period after receive(), then go to sleep).

3.2 Broadcast

Broadcast is simply implemented on top of a single MessageNow and AllocateTime pair. Sending is implemented as follows

1. Call requestBlock() for sendTime(packet length) milliseconds
2. On startBlock(), call send().
3. On sendDone(), call sleepRemaining()

Receiving is also very simple, as all instances of receive() will return FALSE, as we will no longer be receiving additional packets during this period.

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 should call setAddressFiltering() with TRUE, and set *control_length* to the return 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. We return FALSE from the ACK receive in order to sleep for any left over processing time.

At the destination receiver node, we first see a receive() with an RTS packet. As this is destined for us, we return TRUE from receive(), after first posting a task to send a CTS with send(). Then, the receiver waits for DATA, sends an ACK with send() 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 automatically filter out these messages and go to sleep (due to the use of setAddressFiltering()).

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.

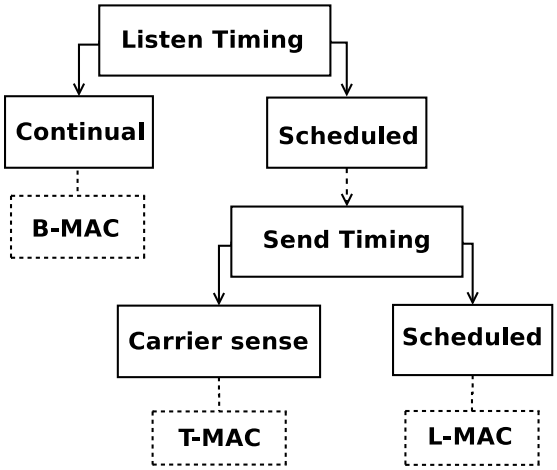

Figure 2: WSN MAC protocol division

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). In the next two sections we intend to describe our implementations of a λMAC implementation of T-MAC (Section 5) and LMAC (Section 6); the latter being an example of a TDMA protocol. We will go into further details of the protocol implementations in the relevant sections, but given the built-in concept of time allocation due to the scheduler mechanisms in each MAC protocol, conversion to the λMAC framework was relatively simple. λLMAC caused more difficulties due to the single-sender semantics of TDMA time allocation, but as we will show, was still feasible.

The only protocol class that we did not have sufficient time to implement yet was the group of which B-MAC [11] is a prominent example - no consistent scheduling, continual sampling of the radio medium (using LPL in B-MAC's case), and a complete lack of built-in time

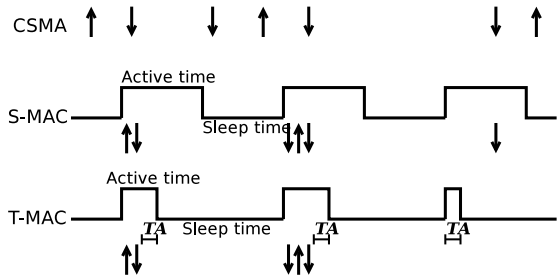

Figure 3: T-MAC

management. One of the challenges for the λMAC framework was to be sufficiently flexible to be capable of implementing such a protocol, while still providing the same level of functionality as with other MAC protocols. However, despite the differences to other protocols, most of the issues that we expect to encounter during the implementation of a λB-MAC have already been dealt with during our creation of λT-MAC. Implementing B-MAC given our work on T-MAC requires two significant blocks of new code - the LPL channel sampling can be implemented like the active/sleep periods of T-MAC, except much shorter and with a fixed awake time rather than T-MAC's dynamic one; and use of the setPreambleLength() function of the MessageNow interface (see Table 2) is needed to allow for the longer preambles required by LPL.

We believe that by showing that T-MAC, LMAC and B-MAC can be implemented with the λMAC framework, and by providing data from our experiments running the first two on our testbed, we adequately demonstrate that the λMAC framework is suitably generic to be able to be a base for implementing a large proportion of currently proposed WSN MAC protocols.

5 λT-MAC

So far we have mostly looked at generic concepts of a λMAC layer. In this section, we describe our implementation of the λT-MAC layer, based on T-MAC [16] for TinyOS [6].

5.1 Scheduling

T-MAC is a CSMA-based MAC protocol, derived from S-MAC [20], but with adaptive duty cycling. The adaptive duty cycling is based on the idea of going to sleep shortly (*TA* milliseconds, defined by 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).

	ROM Size	RAM Size	Max Packet rate
T-MAC	22518	2123	9.9 packets/s
λT-MAC	21678	2192	9.3 packets/s

Table 5: Continual sending test

We took the implementation of T-MAC for TinyOS, and adapted it to provide a λMAC layer, including the removal of its integrated Broadcast and Unicast functionality. Adapting the existing T-MAC protocol to provide the λMAC functionality was relatively simple. We used the frame timers from the Global Time layer to remove a lot of the complexity from T-MAC. A significant part of the existing code was dedicated to schedule synchronisation (including discovery of new schedules); a role now subsumed by the Global Time layer. On a requestBlock() call, λT-MAC places the requested amount of AllocateTime into a nextAllocateTime variable. When T-MAC would normally check if it has a packet to send, λT-MAC instead checks if nextAllocateTime is not 0, 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 with SUCCESS and λT-MAC waits until the end of the AllocateTime period before doing anything else. MessageNow send() and receive()'s 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 we get a phyRequired event (a request from the Global 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.

5.2 Testbed data

In order to test whether the λMAC concept was viable, we compared λT-MAC to the existing T-MAC implementation. Our testing was done on the TNOde platform, a WSN node derived from the mica2dot design [5]. We wished to check whether the switching from a monolithic MAC protocol to the separated λMAC design had affected the code size, generated program size, maximum packet transmission rate and awake/sleep ratios.

To check how large the implementations of the core modules were in each case, we measured the nesC code with SLOCCount [19]. For λT-MAC, this was not only the TMACM module, but also the RadioMessageM module that was used to interface between GenericComm (part of the standard TinyOS network stack) and λT-

MAC. This added together to a total 1247 SLOC (Source Lines Of Code) vs. 1730 lines for T-MAC, a reduction of 27.9%. Adding in the Unicast and Broadcast modules to λT-MAC added a further 226 SLOC, reducing the savings to 14.8%, but the reduced code size for the same functionality is still quite impressive.

For our application testing, one of the example T-MAC applications was used - a simple radio testing application. All packets in the test application had 10 bytes of dummy data in them, and all experiments were run for 60 seconds. We compiled the application with nesC 1.2.4 and gcc 4.0.2 for the AVR.

To test the the maximum output packet rate, we used a version of the application that sends broadcast packets continually. Notably, T-MAC was not designed as a high data rate MAC, but we felt this was still a useful reference test. The result of this test are in Table 5, but the reduction of the packet rate of only 6%, which for a unoptimised reference λMAC was we felt an acceptable loss.

We also tested the active duty cycle of the protocols while sending 1 test packet every second. The result of this test are in Table 6, which shows that the change to the λT-MAC implementation resulted in a <1% increase in the amount of time that the node needed to stay awake in order to send the requested packets. The ~14% duty cycle is quite high for T-MAC, but this is due to a combination of a 610ms frame timer and a 69ms *TA*, giving a minimum duty cycle of ~11% even without any packets being sent, and optimisation of the core protocol implementation could improve this significantly.

Note that for both variations of the test application that the compiled ROM size for λT-MAC was reduced by somewhere at least 840 bytes vs. T-MAC (the exact reduction varies, depending on the level of optimisation that the compiler was able to do for the particular application). This is not as significant as the SLOC reduction (~3%), but this was because λT-MAC is only one part of the larger application, and so existing code provides most of the used ROM.

	ROM Size	RAM Size	Duty cycle
T-MAC	22726	2133	14.26%
λT-MAC	21798	2202	14.4%

Table 6: Broadcast cycle time test

6 λLMAC

LMAC [17] 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 MAC protocol is self-organising in terms of time slot assignment and synchronisation, starting from a sink node (specified by the application). Upon startup, 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.

6.1 Implementation

We created a TinyOS implementation of λLMAC based on the protocol description and the OMNeT++ [18] code available from the LMAC authors. For time synchronisation between the nodes, we used the Global 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. Otherwise λLMAC calls startBlock() with SUCCESS 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.

7 Testing

We performed a series of tests comparing the λMAC versions of LMAC and T-MAC to earlier 'monolithic' implementations. In the case of T-MAC, we had the existing implementation for TinyOS to compare against. As there was no existing TinyOS code for LMAC, we had to work from simulation data. Our simulation work is based upon

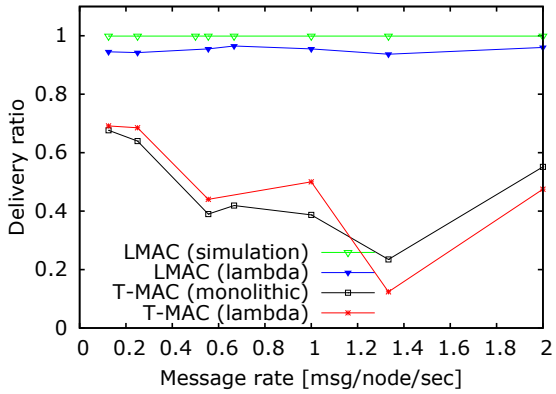

Figure 4: 'Cloud' test

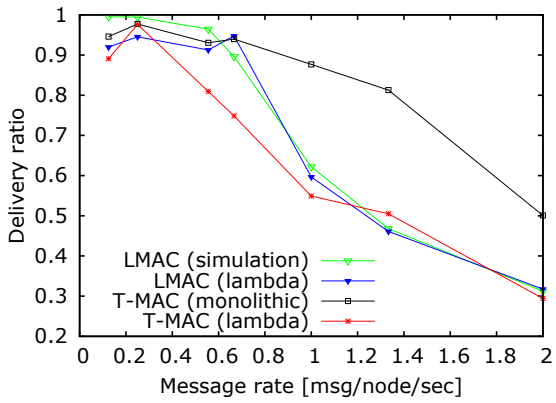

Figure 5: Unicast test

the simulation framework from [7], with various parameters (byte times, frame times, etc) altered in line with the parameters used by the λLMAC implementation. We used two tests: a Unicast test (Figure 5), with all nodes sending to a single 'sink' node; and a 'Cloud' test (Figure 4), with two nodes designated as A and B trying to send packets to each other, while the other nodes send broadcast data around them. In the case of the Cloud test, we measure the packet success rate as the success rate for packets between A and B, ignoring all other packets. The testbed data is from our deployed network of 16 nodes, with power levels set to create a single-cell network with all 16 nodes within one hop of each other; the simulation data is also from a single-cell network with 16 nodes. LMAC was set to a slot time of 50ms, with 32 slots, giving a 1.6s frame. T-MAC was set to the standard frame time of 610ms in all cases.

As can be expected from this form of multi-environment experiment, we encountered a number of interesting results; however, the end data does prove a number of useful things. The Unicast test showed remarkably similar numbers for both of the LMAC implementations - the drop-off curve illustrated on the graph was as we expected as we start to exceed the 1 packet/frame limits

Component	Lines of Code	% of MAC Stack
MAC Framework	3961	variable
λT-MAC	1426	26%
λLMAC	814	17%

Table 7: λMAC sizes

of LMAC. Both versions of T-MAC illustrate the characteristic curve of an overloaded network, but λT-MAC appears to be suffering from additional factors reducing its capability to transmit and receive packets successfully. As the packet sizes are relatively unchanged between implementations, and they both require the same amount of sync packets in order to maintain time consistency, we are currently unsure as to the cause of this drop, but as λLMAC is performing similarly to simulation, we believe that this is an issue with λT-MAC, rather than the the framework.

The Cloud test was designed as an example of a test that LMAC should succeed at, as illustrated by the near-perfect line of the simulation LMAC. One current issue with the simulation environment is its lack of detail regarding the quality of radio links, and this is is probably why the λLMAC is unable to sustain data rates at this level. λT-MAC on the other hand, outperforms monolithic T-MAC on this test.

7.1 Code Size

To check how large the implementations of the core modules were in each case, we measured the nesC code with SLOCCount [19] (Source Lines Of Code). λT-MAC and λLMAC's proportion of the total stack is in Table 7. For λT-MAC, we had an existing TinyOS implementation, and so we could compare λT-MAC to the older implementation. The original "monolithic" T-MAC had a total of 4367 lines of code vs. the 1426 lines of λT-MAC, making λT-MAC only 32% of the original size. Notably, we do not count the lines of code in the MAC framework that are required by λT-MAC, as we only count the code that would have to be written by someone building a new implementation of the MAC protocol in each case, which is the point of the code reuse due to the MAC framework.

7.2 Power Tests

To further check the performance of λT-MAC, we wanted to check its power usage. Unfortunately, the existing TinyOS T-MAC implementation turned out to have a number of bugs regarding power usage (specifically, it used a lot more than it should), hindering direct comparisons, so instead we decided to stick to scenarios where existing research (i.e. the original T-MAC paper [16]) provided us with examples of how a T-MAC implementation

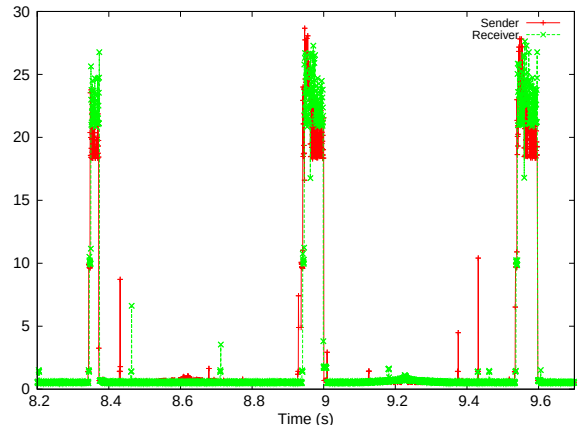

Figure 6: Basic λT-MAC power trace

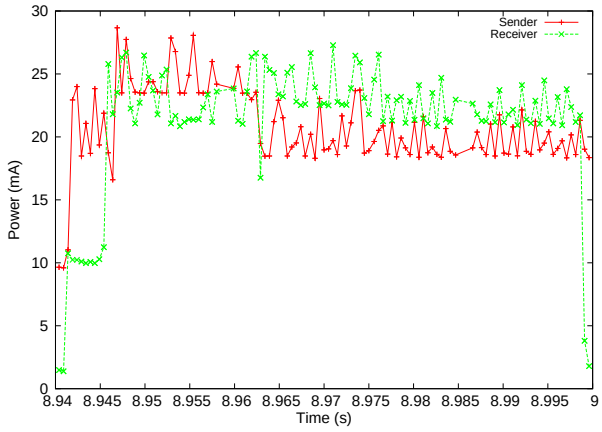

Figure 7: Detail of send/recieve sequence

should behave in terms of power used. We stuck to a simple two-node, unicast sender-reciever pair, with the sender node transmitting 1 packet/second.

Figure 6 shows several seconds of the power readings from this application, with λT-MAC demonstrating the classic T-MAC "awake for short time, sleep for long period" graph, clearly demonstrating good synchronisation between the two nodes.

Figure 7 shows a detail from part of the send/receive sequence from the nodes. The CC1000 radio used by our nodes has similar TX and RX power levels, so the details are difficult to make out, but between 8.945s and 8.962s the DATA packet is being transmitted, and the ACK is being sent between 8.962s and 8.977s. The amount of power used, and the time spent in transmit and receive mode is consistent with our expectations for a T-MAC implementation, giving us additional confidence in the ability of the λMAC framework to correctly implement this protocol.

8 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.

8.1 ExOR

ExOR (Extremely Optimistic Routing) is a "one send, many replies" approach to reliable multicast for routing protocols, first explored by Biswas and Morris [1], and an extended version was proposed in the Guesswork routing protocol [10]. Both variants can be implemented on top of the MessageNow and AllocateTime interfaces, but would require significant effort to implement inside existing MAC protocols.

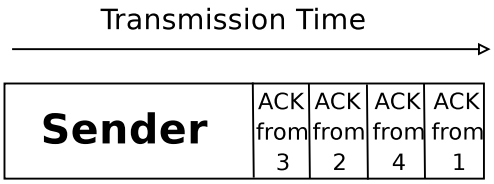

Figure 8: Example ExOR packet timeline

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 **??**). 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. As the destination address field is invalid in this case

(as there is a list of destination nodes later on in the packet), we need to switch off address filtering (using setAddressFiltering()) and do the separation between destination and non-destination receiver nodes in the Transmission layer.

8.2 Priority Queueing and other options

Another possibility that arises once the λMAC layer has been implemented is an option that has been requested by various applications, namely priority queueing [9, 15] - 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.

However, using the λMAC layer, a priority queue can be implemented. Specifically, that requestBlock() corresponds to the normal "send" call, and that although the corresponding startBlock() would normally be the time to send the original packet, any other packet can be sent. To implement a good priority queue, requestBlock() should be called when there is a packet to send out, but with a length appropriate to the maximum size packet that we may wish to send. On startBlock(), we then send the highest priority packet that we have on hand (which may well have arrived since the requestBlock() call), and call sleepRemaining() on sendDone() to trim the listening time appropriately to the length of the packet we actually sent.

9 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 microkernel design [3, 14], 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 codebase.

Polastre et. al [12] 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.

Ee et. al [**?**] 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 this 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.

10 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 Global Time layer to provide application-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.

From our testing here, we have managed to show that our initial attempt at a reference λMAC layer (λT-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 λT-MAC vs. monolithic T-MAC is. We were also able to show that LMAC, a TDMA-based protocol that we expected to be a difficult case, turned out to be not so hard to implement. Some modifications to our existing work were required, and more work with λLMAC is required, but it

has already managed to show good performance vs. existing work with traditionally designed implementations.

By implementing two 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.

10.1 Further work

Certain adaptions of the AllocateTime interface would allow further integration with other MAC protocols, and enable more efficient implementations of TDMA-based protocols. Extending the AllocateTime interface to provide more information about what nodes are the destinations of the packets to be sent during the interval would allow better allocation by TDMA schemes, and possibly noting that certain time slots are more reliably allocated than others, as most TDMA protocols have more reliable guarantees about the lack of other nodes transmitting vs. CSMA protocols with carrier sense. In general, finding better ways to specify more information about the usage patterns for a given AllocateTime slot in a generic way to the λMAC layer will help smarter λMAC protocols allocate time more effectively. We would also like to explore possibilities for more types of Transmission modules.

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 TinyOS, as many new MAC protocols are currently only implemented in simulation, and 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] Sanjit Biswas and Robert 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] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[4] 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.

[5] J. Hill and D. Culler. Mica: a wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November 2002.

[6] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGARCH Comput. Archit. News*, 28(5):93–104, 2000.

[7] 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.

[8] Yuan Li, Wei Ye, and John 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.

[9] K. Lorincz, D.J. Malan, T.R.F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton. Sensor networks for emergency response: challenges and opportunities. *Pervasive Computing, IEEE*, 3(4):16 – 23, 2004.

[10] Tom Parker and Koen 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.

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

[12] 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.

[13] 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.

[14] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alesandro Forin, David Golub, and Michael 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.

[15] Madjid Alkaee Taleghan, Amirhosein Taherkordi, and Mohsen 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.

[16] Tijs van Dam and Koen 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.

[17] 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.

[18] A. Varga. The OMNeT++ discrete event simulation system. In *European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, June 2001.

[19] David A. Wheeler. SLOCCount. http://www.dwheeler.com/sloc/, 2001.

[20] Wei Ye, John Heidemann, and Deborah Estrin. Medium access control with coordinated, adaptive sleeping for wireless sensor networks. *ACM/IEEE Transactions on Networking*, 12(3):493–506, June 2004. A preprint of this paper was available as ISI-TR-2003-567.