A Component-Based Architecture for Power-Efficient Media Access Control in Wireless Sensor Networks

Kevin Klues,* Gregory Hackmann, Octav Chipara, Chenyang Lu
Department of Computer Science and Engineering
Washington University in St. Louis
{klueska, gwh2, ochipara, lu}@cse.wustl.edu

Abstract

The diverse requirements of wireless sensor network applications necessitate the development of multiple media access control (MAC) protocols to meet their varying throughput, latency, and network lifetime needs. Building new MAC protocols has proven to be extremely difficult, however, given the monolithic nature of existing protocol implementations as well as their dependence on a particular radio or processor platform. To address these issues, we propose the MAC Layer Architecture (MLA), a component-based architecture for power-efficient MAC protocol development in wireless sensor networks. MLA consists of optimized, reusable components that implement a common set of features shared by existing MAC protocols, as well as abstractions that encapsulate the intricacies of the hardware platforms they run on. Through an instantiation of MLA in TinyOS 2.0.1, we have implemented five representative MAC protocols. Empirical results show that MLA results in significant code reuse among different protocols, while achieving comparative performance and memory footprints to monolithic implementations of the same protocols.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.13 [**Software Engineering**]: Reusable Software; D.4 [**Operating Systems**]: Organization and Design

General Terms

Design, Performance

Keywords

TinyOS, Power Management, Media Access Architecture

* Now with the Department of Computer Science, Stanford University – klueska@cs.stanford.edu

SenSys'07, November 6–9, 2007, Sydney, Australia.

1 Introduction

Sophisticated power management schemes are often used to meet the extremely long lifetime requirements typical of wireless sensor network deployments. At the heart of these schemes exist power-efficient media access control (MAC) protocols designed to meet the various throughput, latency, and lifetime requirements of different applications. With the diverse set of applications now being run on wireless sensor network hardware, it is becoming increasingly important to provide flexibility in the choice of MAC protocols that best meet the requirements of each particular application.

Experience has shown us, however, that providing such flexibility is an extremely difficult task. Existing MAC protocol implementations tend to be monolithic in nature, tying them to a particular radio platform or microprocessor. Development of a new protocol has meant redesigning the entire radio stack, forcing designers to have intimate knowledge of the hardware for which they are developing their protocols, as well as revisit many of the tricky system issues that plagued the developers of the original stack.

To address these issues, we propose the MAC Layer Architecture (MLA), a component-based architecture for MAC protocols in wireless sensor networks. We have distilled the various features common to existing protocols into a set of reusable components, optimized for the specific function they are intended to provide. Some of these components are low level, encapsulating the intricacies of a particular hardware platform. Others are high level, providing various functionality typical of MAC protocol design in a reusable fashion. Using these components, developers can quickly construct new MAC protocols that meet the demands of their specific applications.

It has been argued that power-efficient MAC protocols cannot be implemented without exploiting features specific to the target CPU and radio platform [1]. For example, TDMA protocols often have precise timing constraints that require fine grained access to hardware resources. Implementations of these protocols must handle timer interrupts immediately after they have been fired and gain access to the radio at the exact instant it is needed. CSMA/CA-based protocols are not as dependent on timing, but require special radio functionality such as *clear channel assessment* (CCA) and the ability to switch between different radio states quickly. These implementations typically sacrifice both modularity and portablity in favor of an increased

level of efficiency. MLA demonstrates that such a sacrifice is unnecessary in an optimized component-based architecture.

In [2], we propose a *Unified Radio Power Management Architecture (UPMA)* for developing flexible, cross-layer implementations of radio power management protocols in wireless sensor networks. UPMA defines link-layer interfaces that allow for a separation between "core" radio functionality and the logic required to perform radio power management [3]. MLA extends this work by identifying additional hardware-independent interfaces required by timing sensitive MAC protocols, and defining platform-independent reusable components that implement MAC layer logic on top of them. The resultant MLA architecture can be used to develop a large number of platform-independent MAC implementations, with little or no further effort required to adapt these implementations to new hardware platforms.

Our work is complementary to the Sensornet Protocol (SP) [1]. SP defines a narrow waist around which various networking and data link technologies can be implemented independently, while sharing a common set of data, and communicating through a unified set of interfaces. However, SP does not provide the power management support that MLA focuses on. In [4] a Network Layer Architecture (NLA) was proposed that allows different networking protocols to be built on top of SP. Our work focuses on providing a similar architecture for developing MAC protocols underneath.

Specifically, we make the following contributions with this paper. (1) We propose the design for a set of robust, reusable components that facilitate the development of a wide variety of MAC protocols. (2) We provide an instantiation of these components for the TinyOS operating system. (3) We evaluate the flexibility provided by these components through the development of five representative MAC protocols that span the existing protocol design space. (4) We provide empirical results that show that our architecture results in significant code reuse of existing components, while achieving comparative performance and memory footprint to monolithic implementations of the same MAC protocols.

The rest of this paper is organized as follows. Section 2 provides an overview of existing MAC protocols. Section 3 distills the common features of these MAC protocols into a component-based architecture. Section 4 describes our implementation of five component-based MAC layers on top of this architecture. Section 5 analyzes the code re-use, throughput, latency, and radio duty cycle of these MAC layer implementations. Finally, Section 6 concludes the paper.

2 Power-Efficient MAC Protocols

In this section, we provide an overview of existing power-efficient MAC protocols. We classify them into a small number of representative categories and identify the common set of features they all share. This classification motivates the component-based design of MLA for promoting code reuse. Typically, existing MAC protocols for wireless sensor networks fall into one of four categories: channel polling, scheduled contention, time division multiple access (TDMA), or hybrid [5].

Nodes equipped with channel polling protocols spend the majority of their lifetime in a sleep state, periodically polling the radio channel to check for activity. If radio activity is detected, the radio is turned on to receive a packet. Otherwise, the radio goes back to sleep until the next polling interval. Sender nodes prefix their data packets with extraneous bytes called a "preamble", to ensure that the destination node detects radio activity and wakes up before the payload is sent. B-MAC [6], one of the first channel polling protocols, accompanies each data packet with a preamble at least as long at the receiver's polling interval. This policy ensures that the receiver performs channel polling at least once while the preamble is being sent. X-MAC improves on B-MAC's energy efficiency and latency cost by embedding target address information into its preamble. When a node overhears a preamble packet, it checks to see if the preamble is addressed to itself. If so, it sends an acknowledgment (ACK) to the sender, which ends the preamble early and triggers the sending of the data packet. Otherwise, the node goes back to sleep until the next polling interval.

B-MAC and X-MAC have been incorporated into TinyOS 2.0.1's CC2420 radio stack. Because the CC2420 is a packet radio, these protocols have been implemented to construct preambles by repeating the data packet inside a tight loop. In general, the channel polling approach achieves good energy efficiency for applications with low data rates. For applications which send data more frequently, however, their energy efficiency may decrease due to the cost of sending long preambles.

Scheduled contention protocols such as S-MAC [7] and T-MAC [8] establish periodic intervals where all neighboring nodes wake up simultaneously to exchange packets. Once awake, nodes which want to transmit data contend for channel access through a process known as CSMA/CA. This approach eliminates the long preambles used by channel polling protocols, since the receiver is guaranteed to wake up in sync with the sender. The downside is that nodes must incur some amount of overhead in order to maintain these synchronized wakeup times.

TDMA protocols like the GTS portion of 802.15.4 [9] and DRAND [10] divide time into slots and allocate these slots to all nodes in a neighborhood. Nodes may transmit without contention in slots allocated to them, but cannot transmit at any other time. Because each node has exclusive access to the radio channel during the slots that have been assigned to them, collisions tend to be rare, and the overall number of packets lost is reduced. However, TDMA protocols often exhibit lower throughput than channel polling and scheduled contention protocols, since nodes may only send during their designated slots. Like scheduled contention, TDMA protocols require time synchronization between all nodes within communication range of one another. Unlike scheduled contention, TDMA protocols tend to be sensitive to changes in multi-hop network topologies, and must generate new slot allocations whenever these changes occur.

The various advantages and shortcomings of each of these three approaches has motivated the creation of hybrid protocols, such as SCP [5], Z-MAC [11], and Funneling MAC [12]. SCP combines scheduled contention and channel polling by synchronizing the time that nodes wake up to sample the radio channel. This synchronization allows sender

nodes to send data with very short preambles. Z-MAC employs a TDMA-style slot allocation for all nodes, but allows nodes to contend for access to other nodes' slots using channel polling. This approach combines TDMA's low channel contention with channel polling's high throughput. Finally, nodes equipped with Funneling MAC contend for channel access in the majority of the network via CSMA/CA, while using TDMA in regions close to sink nodes, where nodes experience high contention. Funneling MAC alleviates contention in the most active areas of the network, without requiring other nodes to create and maintain TDMA schedules.

From the various approaches presented above, a set of common techniques can be seen to emerge. MLA identifies these techniques — such as periodic channel polling and time synchronization — and encapsulates them inside a set of reusable, optimized components. Through the use of these components, MLA is able to simplify the implementation of existing MAC protocols on new platforms as well as facilitate the development of completely new MAC protocols.

3 Design of the MLA

Creating a low-level yet hardware-independent component-based architecture poses three significant design challenges. First, the architecture must present a clean interface to upper layers, exposing as few hardware details as possible. Second, the radio stack must export needed low-level functionality using a set of platform-independent interfaces. Third, functionality common across MAC protocols must be identified and implemented inside a set of optimized, reusable components. In this section, we discuss how we have met these challenges and present the interfaces and components that we identify as necessary for an effective component-based MAC architecture.

Though MLA's architectural design is not inherently tied to the TinyOS operating system, we use TinyOS terminology and naming conventions throughout this section. TinyOS's component-based design offers a well-known vocabulary for discussing interactions among the components in MLA. We express the interfaces described in this section using nesC syntax [13] for analogous reasons.

For the purposes of discussion, we assume the use of packet radios (e.g., the CC2420 radio used on TelosB and MicaZ motes). We choose to focus on packet radios, since industry standards like 802.15.4 reflect a shift away from bit radios (e.g., the CC1000 radio used on Mica2 motes). This decision does not generally affect MLA's design, with the exception of how preamble packets are sent and received by the MAC layer. We defer a more detailed discussion of this artifact to Section 3.3.3.

3.1 Overview of the Architecture

We define two types of components for use in MLA. High-level, *hardware-independent* components are aimed at supporting flexibility by allowing different MAC protocol features to be composed together in a platform independent manner. Low-level, *hardware-dependent* components provide abstract, platform independent interfaces to features otherwise specific to a particular radio or microprocessor platform. Though the implementation of these hardware-dependent components is inherently platform specific, they export interfaces which support the development of fully platform independent high-level components. In this way, porting the set of protocols developed in MLA to a new platform is confined to providing new implementations of these low-level components alone.

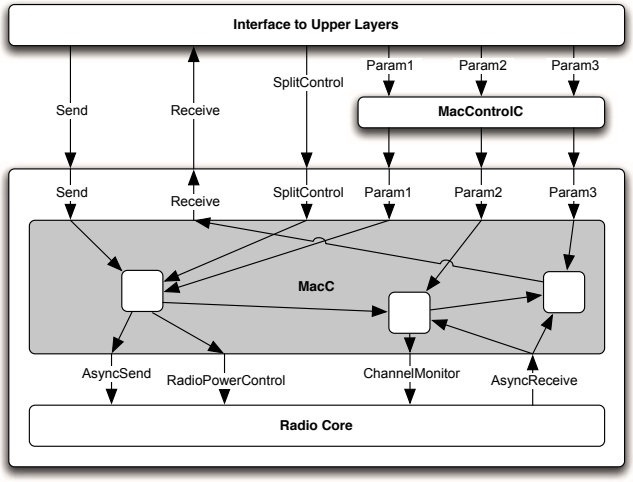

Figure 1. The application developer's view of MLA

Figure 1 provides an overview of how these components can be used to build sophisticated MAC protocols within MLA. Various components are composed together inside of a more general `MacC` configuration, using a set of unified interfaces provided by the radio, and exposing a set of (partially) unified interfaces to the upper layers. The following sections elaborate on these various interfaces, as well as provide detailed descriptions of the components that provide them.

3.2 Interfaces with Upper Layers

The interfaces which MLA provides to upper layers are driven by two specific design goals. First, the MAC protocols's runtime behavior should be as transparent as possible to the user. Application developer effort is best spent developing applications which treat packet I/O as a black box; the fact that packet transmission may be delayed for power-savings purposes or due to radio contention should not affect the application's core behavior. Second, the application developer should not need to be aware of the MAC protocol's internal composition. Developers should be able to treat the MAC protocol as a single coherent entity, and hence be able to insert, replace, or remove a MAC protocol with as little effort as possible.

We achieve both of these goals by exposing all MAC-level interfaces to the application through two distinct components, as shown in Figure 1. First, each MAC protocol defines a `MacC` configuration that composes any resuable MLA components and any protocol specific components together. In order to make its operation as transparent to the user as possible, the `MacC` component uses a fixed set of low-level interfaces (described later in Section 3.4) and produces corresponding application-level packet I/O (`Send/Receive`) and power control interfaces (`SplitControl`).

Upper layers call the `start()` and `stop()` commands of the `SplitControl` interface in order to enable/disable the

operation of the MAC protocol. Calling `stop()` shuts down the entire MAC protocol and puts the radio into an inactive power state. Calling `start()` turns the MAC protocol on, alternating between various radio power states according to the specification of the protocol.

The `Send` and `Receive` interfaces are used to relay packets between upper level components and the underlying radio stack. The MAC protocol's constituent components intercept commands called through these interfaces and apply MAC-specific transformations to them (e.g., adding a timestamp or buffering a packet until the appropriate time) without notifying any upper layer components.

Because `MacC` is exposed to upper layers as a single component with a fixed set of interfaces, developers can swap out different MAC protocols very easily. In the trivial case, MacC simply wires all interfaces directly through to the underlying radio stack, providing no true MAC functionality at all. More commonly, MacC integrates the power management features provided by reusable components in MLA with any protocol-specific components it requires.

Some MAC protocols need to export a small number of MAC-specific interfaces, e.g., to allow applications to control the length of B-MAC's sleep intervals or set the frame length of a TDMA based protocol. These interfaces are gathered into a `MacControlC` component, which provides a single place for developers to expose protocol-specific control interfaces to the application layer. Like `MacC`, each MAC protocol implementation offers different definitions of `MacControlC`. Though the set of interfaces exported by each `MacControlC` varies from protocol to protocol, its internal composition (i.e., which internal components the interfaces actually come from) is never exposed to the application.

3.3 Hardware-Independent Components

Though MLA presents a simple interface to the upper layers, different MAC protocols often exhibit a complex assortment of features. Implementing these features from scratch for each protocol can be difficult and time-consuming. MLA reduces this effort by identifying features that many MAC protocols share, and encapsulating them inside a set of optimized, reusable components.

Tables 1 and 2 list the various hardware-independent and -dependent components defined in MLA respectively. In this section we describe the hardware-independent components. In Section 3.4 we discuss the hardare-dependent ones.

	Channel Polling	Scheduled Contention	TDMA
Channel Poller	X	X	
LPL Listener	X	X	
Preamble Sender	X		
Time Synchronization		X	X
Slot Handlers			X
Low Level Dispatcher			X

Table 1. The various *hardware-independent* MLA components along with the types of protocols that use them

3.3.1 Channel Poller

Channel-polling MAC protocols achieve low duty cycles by performing *low-power listening* (LPL), which samples the radio channel for activity at fixed intervals. The `ChannelPollerC` component facilitates these checks by invoking the radio stack's CCA routines at a specified interval. Once the radio stack has completed the CCA check, the `ChannelPollerC` fires an `activityDetected` event. The `detected` parameter is set to `TRUE` or `FALSE` depending on whether the radio channel is occupied or free, respectively.

	Channel Polling	Scheduled Contention	TDMA
AsyncIOAdapter	X	X	X
Alarm	X	X	X
Local Time		X	X
Radio Core	X	X	X

Table 2. The various *hardware-dependent* MLA components along with the types of protocols that use them

3.3.2 LPL Listener

The `ChannelPollerC` does not provide a complete LPL scheme in and of itself, since it does not adjust the radio's power state based on channel activity. MLA provides two components that implement common LPL policies. `FixedSleepLplListenerC` sleeps for a fixed interval with the `ChannelPollerC` active. When radio activity is detected, `FixedSleepLplListenerC` activates the radio and disables the `ChannelPollerC`. After the radio becomes free, `FixedSleepLplListenerC` starts a timeout alarm. If no more radio activity is detected before the alarm fires, then the radio is powered down and the `ChannelPollerC` is re-enabled. This component encapsulates the LPL policy used by B-MAC and X-MAC.

The `PeriodicLplListenerC` component instead keeps the `ChannelPollerC` component active at all times, and immediately moves the radio into its sleep state when the channel is free. This policy guarantees that CCA checks occur at regular intervals, which scheduled contention and some hybrid protocols like SCP mandate. Consequently, the length of time that a radio will sleep between two CCA checks is reduced by the amount of time that the radio is active. In contrast, `FixedSleepLplListenerC` guarantees that the radio will sleep for a fixed interval, but may delay the next CCA check to do so.

These components' duty cycle can be controlled using the `LowPowerListening` interface listed below. This interface is derived from a similar, but radio-specific, interface used within the TinyOS 2.0.1 release of the `cc2420` radio stack. `LowPowerListening` allows the user to control the radio's sleep cycle by either explicitly setting the sleep interval (e.g., 100 ms), or by providing a target duty cycle (e.g., 5%).

```
interface LowPowerListening {
  async command void setLocalSleepInterval(t_ms);
  async command uint16_t getLocalSleepInterval();
  async command void setLocalDutyCycle(t_ms);
  async command uint16_t getLocalDutyCycle();
  async command uint16_t dutyCycleToSleepInterval(t_ms);
  async command uint16_t sleepIntervalToDutyCycle(t_ms);
}
```

3.3.3 Preamble Sender

For a sender node to ensure that the recipient nodes are awake when messages are sent, channel polling protocols prefix data with a stream of preamble packets. The `PreambleSenderC` component performs this procedure, providing the `PreambleSender` interface shown below. The `sendPreamble` command and `preambleDone` event form a split-phase operation for sending preambles. Unlike conventional data-packet sending functionality, `sendPreamble` requests a time interval (`t_ms`) in addition to a packet. Back-to-back copies of the packet are sent during this interval, and the `preambleDone` event is only fired at the end of this stream.

```
interface PreambleSender {
  async command error_t sendPreamble(msg, len, t_ms);
  async event void preambleDone(msg, err);
  async event resend_result_t resendPreamble(msg);
}
```

To save power and reduce latency, some MAC protocols (e.g, X-MAC) may interrupt the preamble stream before the end of the specified interval. To support this behavior, the `PreambleSenderC` fires the `resendPreamble` event after each copy of the packet is sent. This event queries the MAC protocol to decide if the next preamble packet is subject to backoffs (`RESEND_WITH_CCA`), not subject to backoffs (`RESEND_WITHOUT_CCA`), or simply should not be sent at all (`DO_NOT_RESEND`). `PreambleSenderC` will repeatedly send the specified packet until `resendPreamble` returns `DO_NOT_RESEND` or until the specified interval ends, whichever comes first. Power management schemes which do not need this behavior can implement a trivial event-handler that always returns either `RESEND_WITH_CCA` or `RESEND_WITHOUT_CCA`. For TinyOS based implementations of this component, this process is very efficient due to the function inlining features of the nesC compiler.

This component is designed with packet radios in mind, and operates by sending packets in a tight loop. This strategy is sub-optimal on byte radios, which can send streams of uninterrupted bytes. This limitation could be lifted by creating a second `PreambleSenderC` implementation optimized for byte radios. Such a change would be transparent to the other components, because all low-level preamble sending logic is encapsulated entirely within `PreambleSenderC`.

3.3.4 Time Synchronization

All TDMA protocols, as well as some scheduled contention protocols, require some form of time synchronization. Sometimes the method used by a particular MAC protocol is completely protocol specific, e.g., piggybacking protocol-specific timestamp information onto data packets. Other times, a more general approach can be used. We therefore provide two different mechanisms for defining time synchronization components within MLA.

First, we provide `BeaconedTimeSyncC`, which exchanges synchronization packets at regular intervals and adjusts the internal timers of both sender and receiver nodes accordingly. This time synchronization scheme is common among TDMA and scheduled contention protocols (e.g., 802.15.4 and S-MAC [7]).

For the definition of protocol-specific time synchronization components, we propose a general *design pattern* that their implementations should follow. A component should be created that contains pass-through filters for a radio's sending and receiving interfaces. This component adds timestamps to outgoing packets and adjusts the node's internal clock based on timestamps extracted from incoming packets. We follow this pattern in our implementation of SCP's time synchronization, as discussed in Section 4.

Note that the fine grained timing information required for proper development of time synchronization components is encapsulated within the hardware-dependent `LocalTime` and `Alarm` components discussed in Section 3.4. These components, while implemented in a platform specific manner, provide a standard set of timing related interfaces for use by platform independent components. Without them, it would be impossible to develop time-synchronization protocols in a platform independent manner.

Because MLA provides low-latency hooks to radio hardware, the time synchronization schemes for most MAC protocols can reasonably be implemented in software. For MAC protocols that require stricter time synchronization bounds, some radio hardware offer internal support for applying timestamps to packets immediately before they are sent. Supporting hardware timestamps in a platform-independent manner is non-trivial, since different radio hardware may use different timestamping formats. For this reason, MLA does not currently support the use of such hardware-specific timestamping. We will consider ways to wrap this functionality in a hardware-independent fashion in the future.

3.3.5 Slot Handlers

TDMA protocols divide time into periodic *frames*, and subdivide these frames evenly into *slots*. Each slot in the frame is assigned an operation. Though some of these operations are protocol-specific (e.g., sending control packets), many operations are common to various TDMA protocols: e.g., sending packets with or without contention, performing CCA checks, and going to sleep.

This commonality presents an opportunity for component reuse using the following scheme. Each MAC protocol implementation contains a set of *slot handlers*, each of which encapsulates the logic for one operation. Slot handlers for common operations, such as sending packets, are included in MLA, whereas protocol-specific ones must be provided by the respective protocol implementation. Each protocol also provides a *slot scheduler* component. This component hooks into a periodic `Alarm` (discussed in Section 3.4.6) which fires at the beginning of each new slot. The slot scheduler delegates these events to the appropriate slot handler, based on the operation assigned to the upcoming slot.

As an example, consider the GTS portion of 802.15.4. 802.15.4 assigns one of three actions to each slot: time synchronization, contention-based sending, and contention-free sending. Thus, 802.15.4 can be implemented in MLA by providing a slot handler for each of these three slot types, and a slot scheduler that invokes the appropriate handler at the beginning of each slot.

MLA currently provides two generic slot handlers. `TDMASlotHandlerC` implements contention-free packet

sending by disabling CCA and setting backoff intervals to 0 before sending a packet. It also adds a guard time before transmitting packets in order to accommodate for clock drift and processing jitter. `CSMASlotHandlerC` provides contention-based packet sending, by using the `ChannelMonitor` interface described in Section 3.4.2 to verify that the channel is free before sending the packet. These generic components encapsulate operations which are used by many MAC protocols.

3.4 Hardware-Dependent Components

Any non-trivial MAC protocol requires special radio layer support not required by ordinary application code: timely access to hardware resources, the ability to change the radio's power level, etc. Monolithic implementations of these protocols achieve the necessary level of support by adding protocol-specific hooks into low-level radio code, or by interleaving the MAC code directly into the radio stack. These approaches generally result in poor portability, and create "forks" of the low-level radio code which become out-of-sync with the original implementations. For example, prior to the release of TinyOS 2.0.1, two versions of each of the `cc1000` and `cc2420` radio stacks were included in the TinyOS distribution: one with support for B-MAC-style low power listening, and one without [14]. Enhancements and bugfixes to the baseline `cc2420` stack were not consistently applied to the LPL-enabled stack, resulting in a confusing and increasingly-diverging codebase.

MLA avoids these pitfalls by identifying the key low-level capabilities required by MAC protocols, and providing rich hardware-independent interfaces that expose them. Since the implementation of these interfaces is highly platform specific, different implementations must exist for each radio and microprocessor platform that supports them. We expect that the effort to adapt a new radio stack to support these low-level interfaces should be low, since these interfaces generally wrap existing functionality in the radio stack.

Following the components introduced in Table 2, we first present the set of hardware-independent interfaces that MLA defines for exposure by the core radio stack: `RadioPowerControl`, `ChannelMonitor`, `CcaControl`, `Resend`. We then provide details of the interfaces provided by the AsyncIOAdaptor, Alarm, and LocalTime components respectively.

3.4.1 Radio Power Control

The ability to control the power state of the radio is the most fundamental need of any power managing MAC protocol. In [3], we proposed the `RadioPowerControl` interface to allow a MAC protocol to switch the radio between two distinct power states: active and sleep. MLA adopts the same interface for controlling the power state of the radio.

```
interface RadioPowerControl {
  async command void start();
  async event void startDone(error);
  async command void stop();
  async event void stopDone(error);
}
```

The `RadioPowerControl` interface is simple by design. Many radios provide multiple low-power states with varying degrees of latency and power consumption when switching between them. We note, however, that most existing MAC protocols only utilize one of these low-power states. The implementation of the `RadioPowerControl` interface must therefore choose the most appropriate one. This interface can be extended to expose multiple power states if future MAC protocols require them.

Many applications expect some degree of explicit control over the radio's power state. Rather than providing the application with direct control through the `RadioPowerControl` interface, MLA exposes a standard interface for enabling or disabling the entire MAC layer. When the MAC layer is disabled, all of its internal components are also disabled, and the radio is powered down. Conversely, whenever the MAC layer is enabled, its internal components are also enabled, and the MAC protocol is responsible for managing the radio's power state. This policy, which we use throughout the MAC layer implementations described in Section 4, greatly simplifies management of the radio's power state.

3.4.2 Channel Monitor

The `ChannelPollerC` component described in Section 3.3.1 assists the MAC layer with periodic CCA checks. However, it is reliant on radio stack support to actually perform these checks. The mechanism for doing so varies significantly from radio to radio. For example, the CC2420 radio features a simple hardware CCA pin, whereas CCA checks must be emulated on the CC1000 radio by taking RSSI samples [6]. Likewise, the type of filtering applied to these raw CCA checks (e.g., how many times to sample the CC2420's CCA pin, and what percentage of these samples must be positive) is radio-specific.

MLA adapts the `ChannelMonitor` interface described in [3] as seen below:

```
interface ChannelMonitor {
  async command void check();
  async event void free();
  async event void busy();
  async command void setCheckLength(t_ms);
  async command uint16_t getCheckLength();
}
```

This interface provides a simple, platform-independent wrapper around complex radio-specific CCA procedures. After the MAC layer signals the `check` command, the radio performs a CCA check in a split-phase fashion. It then fires a `free` or `busy` event according to the results of its CCA check. We have augmented this interface to support `setCheckLength` and `getCheckLength` commands. These commands allow the MAC layer to customize the maximum duration of time that a CCA check can last. This degree of control is important in many TDMA protocols, which require deterministic packet processing time in order to maintain tight clock synchronization.

Implementing an effective CCA check routine inside the radio stack can be challenging. While many of these routines are generally already present in some form in many existing radio stacks, they tend to use radio-specific interfaces and are not portable accross different CPU platforms.

In Section 4.6 we discuss how this challenge can be overcome so that augmenting an existing radio stack to provide platform-independent `ChannelMonitor` support simply involves wrapping the radio specific CCA check routine with this interface.

3.4.3 CCA Control

MAC protocols with tight timing constraints, such as TDMA protocols, require strict bounds on packet latencies. However, packet backoff intervals can contribute an unpredictable amount of latency if backoff decisions are left entirely to the discretion of the radio stack. Specifically, the radio may apply an initial backoff; perform a CCA check to see if the shared medium is free; and apply an additional backoff if contention is detected. To achieve the timing requirements that some MAC protocols require, they must be able to set these backoff intervals appropriately, or even disable them altogether.

The `CcaControl` interface (similar to the `MacControl` interface provided in [6]) allows a MAC protocol to control the radio layer's CCA checks and subsequent backoff behavior. Before each packet is sent, the radio layer fires a `getInitialBackoff` event to allow the upper layer to specify the packet's initial backoff time. Likewise, the `getCongestionBackoff` event is fired when a transmission must back off due to channel contention. This event not only allows upper layers to change the default congestion back off time, but also indicates to the MAC protocol that a packet has experienced contention, allowing it to cancel the pending transmission if the protocol requires. Finally, the `getCca` command allows contention-free protocols to turn off CCA checks entirely, keeping packet latency to a minimum. All three of these events pass along the corresponding default values defined by the radio, so that the MAC layer can selectively leave them unchanged.

```
interface CcaControl {
  async event bool getCca(msg, default);
  async event uint16_t getInitialBackoff(msg, default);
  async event uint16_t getCongestionBackoff(msg, default);
}
```

3.4.4 Low-Cost Packet Resending

MAC protocols often send multiple copies of packets in tight loops, e.g., to create a long preamble out of shorter packets. In many of these situations, the cost of resending a packet should be as small as possible; e.g., the packets in a preamble should have minimal gaps between them. Some radio hardware, such as the CC2420, specifically offer support for this activity: they can quickly resend the last packet, skipping time-consuming activities like loading the hardware packet buffer. Our `Resend` interface exposes this capability in a hardware-independent fashion. The `resend` command resends the last packet sent over the radio using the fastest retransmission path supported by the radio hardware. In the absence of native radio support, packet resending can be emulated using a simple one-packet buffer.

3.4.5 Low-Latency I/O

In order to achieve maximum power efficiency, many MAC protocols expect to send packets with as short a delay as possible. In particular, TDMA-based protocols and some scheduled contention-based protocols stamp outgoing packets with time synchronization information. To preserve the accuracy of these timestamps, the MAC layer must be able to send and process clock synchronization information in a very time-sensitive fashion.

This low latency can be achieved by exposing radio hardware events to the MAC layer with as thin a layer as possible. In general, existing radio stacks do not follow this approach. Instead, *asynchronous* hardware interrupts are converted into *synchronous* tasks or threads, whose execution can be deferred for an indeterminate length of time. This policy is sensible at the application layer: executing I/O-handling code within the context of a hardware interrupt is potentially dangerous, and can prevent the radio from responding in a timely fashion. However, this trade-off is not acceptable at the MAC layer: when hardware interrupts fire in the middle of a long-running computation, the corresponding tasks or threads produced to handle them often face arbitrary delays.

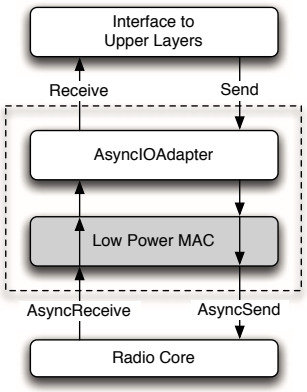

Figure 2. Asynchronous to synchronous I/O adaptation

MLA therefore exposes the radio's send and receive operations through `AsyncReceive` and `AsyncSend` interfaces, as shown in Figure 2. These interfaces are similar to the corresponding synchronous `Receive` and `Send` interfaces used in TinyOS, but are invoked or reachable directly from interrupt handlers (i.e., asynchronous context) rather than tasks or threads (i.e., synchronous context).

For the reasons listed above, MLA does not expose these radio events directly to the upper layers. All I/O events that the MAC layer does not consume are passed through an `AsyncIOAdapter`, a component which converts asynchronous I/O events into their synchronous counterparts by posting tasks or scheduling a thread. Monolithic radio stacks generally use a similar adapter internally; we simply move this logic outside of the radio stack so that both our hardware-independent and hardware-dependent components can interface with the radio in a timely fashion. This subtle, but important, change allows components developed above the core radio stack to perform operations on incoming and

outgoing packets directly within asynchronous code signaled from radio hardware events.

In a threaded operating system, the components defined by MLA would all be invoked within the asynchronous event handlers of the radio device driver. MLA's operation is therefore independent of any higher-level threading functionality offered by the OS. For example, any threads that send a packet would make blocking I/O calls to the radio driver using the OS's application-layer networking APIs. Only once the I/O operation had completed would the driver schedule the thread to run again. Receiving a packet would trigger a similar series of operations in the reverse direction.

3.4.6 Alarms

Many MAC protocols are timing-sensitive, requiring certain events to be triggered at precise intervals. This can be achieved by hooking into the periodic interrupts generated by low-level hardware timers. However, existing radio stacks do not generally allow direct access to this hardware in a platform-independent way. Like radio I/O events, timer events are converted to tasks or threads. Again, while this policy discourages application code from running within the context of a hardware interrupt, it often imposes a long delay before timer events can be handled. This delay can be critical at the MAC layer, which may require sub-millisecond precision to perform activities like time synchronization.

TinyOS 2.0 includes an `Alarm` interface which provides the needed hardware-independent abstraction that we seek. This interface is reproduced below for the reader's convenience. The `start` command instructs the alarm to trigger an interrupt after a specified time interval elapses. This interrupt is exposed to the higher layers as a `fired` event. The `stop` command cancels the previous `start` command without firing the `fired` event, and `isRunning` checks to see if some alarm is pending. Finally, `getNow` provides some notion of absolute time (e.g., the number of milliseconds since the sensor was booted).

```
interface Alarm {
  async command uint16_t getNow();
  async command void start(time);
  async command void stop();
  async command bool isRunning();
  async event void fired();
}
```

Apart from a hardware-independent alarm interface, we also require a hardware-independent mechanism for wiring hardware alarms into our MAC layer. We propose that each platform implement generic `AlarmMilliC`, `Alarm32khzC`, and `AlarmMicroC` components that provide Alarm interfaces with millisecond-, 32 KHz-, and microsecond-resolution, respectively. Because not all platforms provide alarms with exactly these resolutions, these components may need to internally transform the available hardware alarms into the required form.

3.4.7 Local Time

Many time-synchronization protocols exchange timing information in some form (e.g., how long has elapsed since the last time a synchronization message was received). This information is generally derived from a hardware counter, which automatically increments at small intervals (e.g., once a millisecond). Unfortunately, these counters are not suitable for long-lived applications, since they will quickly overflow. In the common case of a 16-bit counter which increments every 1 ms, the counter will overflow in just over 2 days. Although these overflow events are often exposed to the application layer, this forces the application to count and accommodate for these overflows each time they occur.

```
interface LocalTime<local_time_t> {
  async command local_time_t getNow();
  async command local_time_t add(t1, t2);
  async command local_time_t sub(t1, t2);
  async command bool lessThan(t1, t2);
  async command bool greaterThan(t1, t2);
  async command bool equal(t1, t2);
}
```

To handle these overflow cases, we introduce a series of components (`LocalTime32khz32C`, `LocalTimeMilli16C`, etc.) which wrap hardware counters of the corresponding width and frequency, using the `LocalTime` interface shown above. This component also tracks and counts hardware counter overflows. When the `getNow` command is invoked, the component returns a timestamp in a `local_time_t` structure, which is twice the counter's native width and is adjusted according to the number of times the counter has overflowed. We also provide convenience commands for adding, subtracting, and comparing these timestamps.

4 Implementation

We have implemented MLA in the TinyOS 2.0 operating system [14]. This implementation is available in the `tinyos-2.x-contrib` section of TinyOS's CVS repository [15]. Wherever feasible, components were implemented by porting portions of the existing TinyOS 2.0.1 code base, including the monolithic implementations of B-MAC and X-MAC in the `cc2420` radio stack. We removed the monolithic power-management code from this stack and augmented it to export the low-level interfaces described in Section 3.4.

Additionally, we have built five MAC protocols using MLA: two channel polling, one TDMA, and two hybrid. Specifically, we implemented B-MAC, X-MAC, SCP, a pure TDMA protocol, and a hybrid protocol called SS-TDMA ("SS" stands for "slot-stealing") that combines TDMA with CSMA/CA. These protocols have been thoroughly tested for our augmented CC2420 radio stack on MSP30-based TelosB motes, but should be deployable "as-is" on other radio stacks that provide the low-level interfaces specified by MLA.

In this section, we discuss the composition and implementation of the five MAC protocols within MLA, focusing specifically on component re-use throughout the system. We also highlight some of the system-related challenges that we faced during the implementation process.

All of our implementations contain a protocol-specific component that is used to enable and disable the MAC protocol as a whole upon request. Due to space limitations we omit these components, and focus on the details of implementing the key logic of each protocol. For clarity, we also omit discussion of the `AsyncIOAdapter` that sits directly on

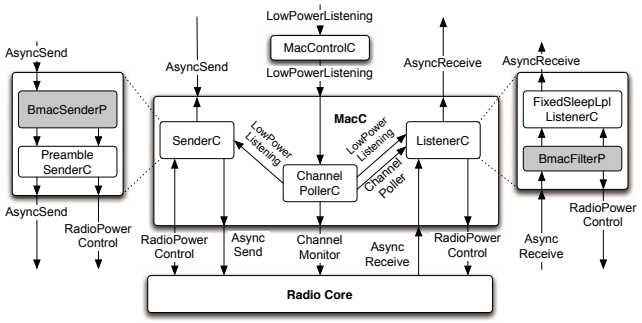

Figure 3. Composition of the B-MAC protocol; unshaded boxes represent reusable components, and shaded boxes represent protocol-specific components

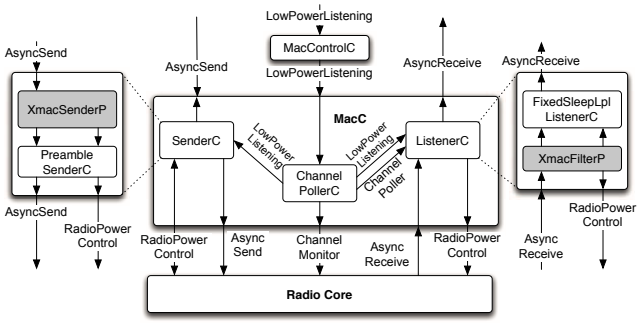

Figure 4. Composition of the X-MAC protocol

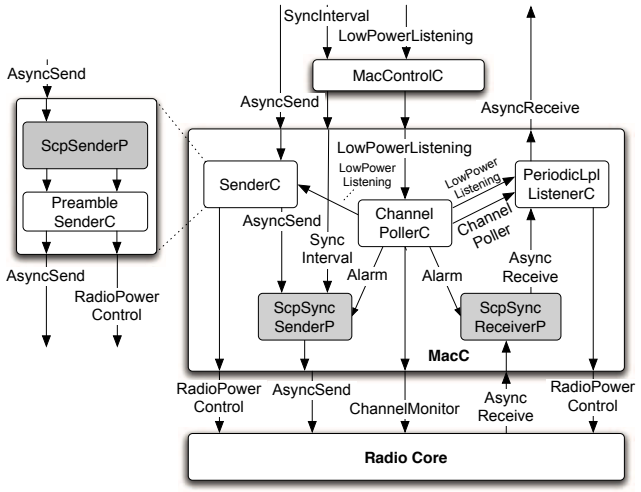

Figure 5. Composition of the SCP protocol

top of all five MAC layer implementations. We refer the reader back to Section 3.4.5 for details on this component.

4.1 B-MAC

B-MAC, shown in Figure 3, is the simplest MAC protocol implemented in MLA. It employs a B-MAC specific `BmacSenderP` component for intercepting outgoing packets from upper layers. It buffers these packets, invokes `PreambleSenderC`'s `sendPreamble` command, sends the buffered packet, and turns off the radio using the `RadioPowerControl` interface. To simplify implementation, `BmacSenderP` uses a copy of the buffered data packet to serve as the preamble packet. The number of preamble packets to send is obtained from `ChannelPollerC`'s `LowPowerListening` interface, since B-MAC sends preambles for the entire duration of LPL sleep interval.

B-MAC includes a small `BmacFilterP` component which emulates two aspects of the monolithic `cc2420` stack's behavior. First, it counts the number of consecutive overheard packets which are destined for other nodes. When this count reaches a certain threshold, `BmacFilterP` starts `FixedSleepLplListenerC`'s timeout alarm. This optimization allows nodes that overhear a transmission to go back to sleep early, without waiting for the preamble to end. Second, it intercepts incoming packets and holds them in a queue until no radio activity is detected (i.e., until it has received the final packet containing data at the end of a senders preamble stream). These behaviors are not part of the original B-MAC specification [6]; the first is taken from X-MAC, and the second is unique to the monolithic `cc2420` stack. Nevertheless, we implement both behaviors for consistency with the existing monolithic stack.

4.2 X-MAC

X-MAC extends B-MAC by adding optimizations to end the preamble stream as soon as possible. As a result, X-MAC's composition closely mirrors that of B-MAC, with two distinct differences. As seen in Figure 4, we first replace `BmacSenderP` with `XmacSenderP`, which sends preambles in accordance with X-MAC's early-ACK optimization. `XmacSenderP` is identical in implementation to `BmacSenderP`, except that it responds to `resendPreamble` events with `RESEND_WITH_CCA` before the recipient ACKs a preamble packet, and `DO_NOT_RESEND` thereafter. With this optimization, the sender node will terminate the preamble as soon as the recipient ACKs one packet, allowing both the sender and receiver to go back to sleep early.

Second, X-MAC does not queue incoming packets, since the sender will end the preamble as soon as one ACK is received. X-MAC's introduces an `XmacFilterP` component, which is analogous to the `BmacFilterP` component except that it excludes this queue.

4.3 SCP

Like X-MAC, SCP reuses several MLA components used by B-MAC. SCP also introduces new components to embed and extract timestamps for incoming and outgoing packets. As shown in Figure 5, the `ScpSyncSenderP` component appends each outgoing packet with a 2-byte counter representing the time remaining on `ChannelPollerC`'s internal alarm (i.e., how long until the node performs its next CCA check). `ScpSyncReceiverP` reads these timestamps from incoming packets and adjusts the local `ChannelPollerC`'s alarm accordingly. This adjustment ensures that all nodes wake for CCA checks simultaneously. The `ScpSyncSenderP` periodically sends explicit time synchronization packets if no other data is sent during some interval; the application may customize this interval using the `SyncInterval` interface exported through the `MacControlC` component.

SCP also introduces a `ScpSenderP` component to buffer outgoing packets. In contrast to `BmacSenderP` and `XmacSenderP`, which begin sending preambles as soon as they are prompted to send a packet, `ScpSenderP` waits until just before its next CCA check (i.e., just before all the nodes are scheduled to wake up). This change allows the sending and receiving nodes to synchronize their radio activity, and allows SCP to use a short preamble (9 ms on CC2420 radios). `ScpSenderP` also implements the packet pipelining optimization described in [5], which allows the application to send multiple packets within one LPL interval.

Another fundamental difference is that `ScpSenderP` constructs its preamble from explicit preamble packets containing alternating 1s and 0s, rather than reusing the buffered data packet directly. Preamble packets arrive so rapidly that `ScpSyncReceiverP` cannot afford to do any processing on them, or else it will overrun its interrupt handler. This change allows `ScpSyncReceiverP` to immediately identify and discard incoming preamble packets with minimal processing. Note that preamble packets contain no data useful to the upper layers: they exist simply to occupy the channel while the recipient node is listening.

Unlike the previous two protocols, SCP contains a small portion of radio-dependent code. This code enumerates a handful of radio-specific constants, such as the amount of time it takes to wake up the radio. It is impossible to completely remove this radio-dependent portion, since SCP achieves its high energy efficiency by adjusting the length of its preamble according to these radio-specific properties. However, the CC2420-specific portion of SCP contains only 8 lines which declare constants derived from experimental data. The effort required to add support for additional radios should be minimal.

While our implementation in MLA implements most features of the SCP protocol, we have not yet implemented its overhearing avoidance feature described in [5]. Our architecture could support this optimization in the future by adding a component analogous to `XmacFilterP`, with additional logic to prevent interference with SCP's separate packet pipelining optimization.

4.4 Pure TDMA

Pure TDMA, shown in Figure 6, is a TDMA protocol designed for a single-hop network in which nodes communicate to a single base-station. This protocol is similar to the GTS portion of 802.15.4. The `PureTDMASchedulerC` component implements a slot scheduler for a TDMA frame that contains both active and sleep periods. The length of each period can be configured by the application using the `FrameConfiguration` interface.

The first slot in the active period is reserved for the `BeaconedTimeSyncC` component to exchange time synchronization information. The remainder of the slots in the active period are assigned to nodes for transmitting packets without contention using the `TDMASlotHandlerC` slot handler. Access to the radio is arbitrated between these handlers using the `LowLevelDispatcherC` component. The scheduler also turns off the radio in the first slot of the sleep period and wakes it up in the last slot of the active period.

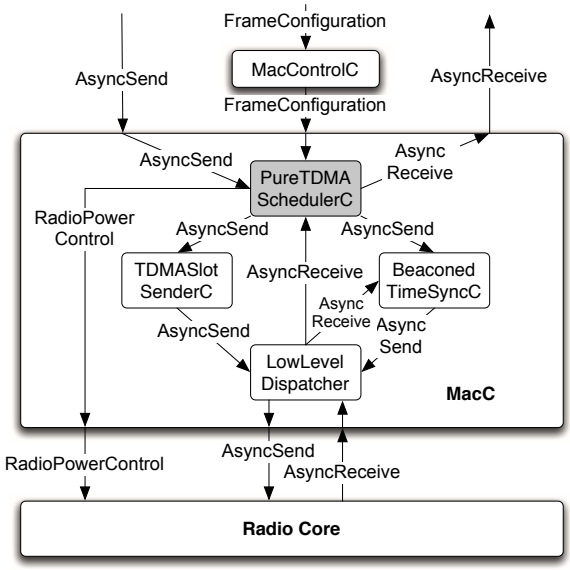

Figure 6. Composition of the pure TDMA protocol

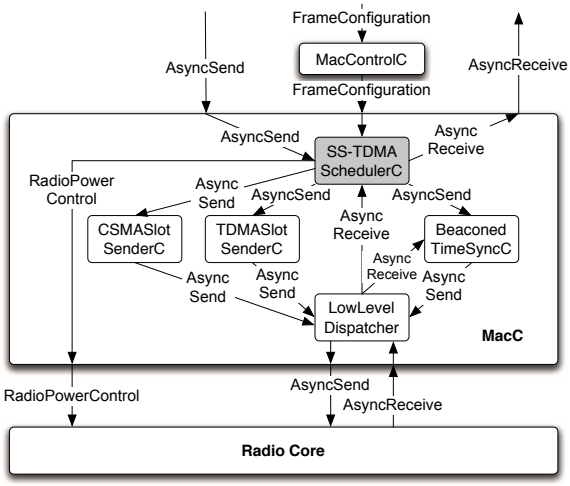

Figure 7. Composition of the SS-TDMA protocol

4.5 SS-TDMA

SS-TDMA, shown in Figure 7, is a hybrid protocol which combines TDMA and CSMA/CA. SS-TDMA is an extension of the Pure TDMA protocol which includes an optimization originally introduced by Z-MAC [11] to allow the MAC layer to reuse idle slots. Like Pure TDMA, SS-TDMA divides a frame into periods of activity and sleep. A node scheduled to transmit during one of the slots in the active period is said to "own" that slot. Slot owners are given preference to transmit, and send data using a short initial backoff. If a slot owner does not have any data to transmit, other nodes may contend for its use after some additional delay.

SS-TDMA is implemented as follows. Like in Pure TDMA, the first slot in each frame's active period is reserved for the `BeaconedTimeSyncC` component to exchange time synchronization information. For all other slots in the active period, the `PureTDMASchedulerC` is used to deter-

mine which node is currently scheduled to transmit. Whenever a node has a packet to send, the PureTDMASchedulerC first buffers the packet, and then invokes one of two slot handler components in order to send it. During slots that the node owns, the TDMASlotSenderC component sends the packet immediately without a CCA check or backoff. For all other slots, the CSMASlotSenderC component uses the ChannelMonitor interface to perform a CCA check and determine if the current slot's owner is transmitting. If not, then the node performs a short backoff followed by another CCA check. This second CCA check reduces collisions among multiple nodes that try to steal the same slot simultaneously. If this second CCA check determines that the radio is still free, then CSMASlotSenderC transmits the buffered packet.

4.6 Lessons Learned

Because the MAC layer operates in an asynchronous context close to the radio layer, implementing the components described in this process is an inherently complex process, rife with potential concurrency and timing errors. We discuss here several of the most notable challenges we faced while implementing MLA and the five MAC protocols described in this section.

Implementing the ChannelMonitor interface in the CC2420 radio layer required some subtle adjustments. Because hardware CCA checks on the CC2420 radio are instantaneous and only consider the last 8 samples from the radio, the CCA pin must be queried in a loop that executes at least as long as the gap between packets (up to 8 ms, according to our measurements). To perform this check, we initially extracted the getCca task from the CC2420DutyCycleP component found in the cc2420 stack. [1] The results were poor; according to our oscilloscope measurements, our CCA check loop would last under 5 ms, resulting in a number of false negatives and an alarming number of dropped packets, particularly in conjunction with X-MAC.

After inspecting the code, we discovered that the loop in this task executed for a fixed number of iterations determined by compile-time constants: e.g., the monolithic X-MAC implementation used 500 iterations on MSP430-based platforms and 400 on others. This policy is inherently nonportable, since the amount of time required to execute each iteration will vary from CPU to CPU. Even the small changes we made to the loop's contents would drastically affect the length of the CCA check interval. We eventually developed a cross-platform solution using our LocalTime interface based around a 32 KHz hardware counter. For example, to achieve a check time of 8 ms, we terminate the loop after the counter increases by 256 (8 ms x 32 KHz). This approach guarantees that the loop will execute for the same MAC-protocol specified length on any platform, regardless of the CPU used.

Additionally, our implementations of ScpSyncSenderC and ScpSyncReceiverC did not initially treat preamble packets as special cases. This had three unanticipated effects. The sender required extra time to apply timestamps to the outgoing preamble packets, increasing the gap between packets and hence creating the possibility of packet gaps larger than the receiver's CCA check. When the receiver did receive a packet, it would always attempt to adjust its timer using the embedded timestamp. Because the preamble packets are sent almost immediately before the sender's CCA check — as short as 1 ms prior — their timestamps are correspondingly short. The recipient would adjust its timer to this very short interval, often causing it to fire almost immediately after each preamble packet was processed. Most seriously, since the receiver node would process the timestamps on all incoming packets directly within the radio hardware's interrupt handler, its incoming packet buffer would often overflow, and the data packet would be lost. As noted above, we avoided these side effects by short-circuiting preamble packets through ScpSyncSenderP without applying timestamps, and dropping them as soon as possible in ScpSyncReceiverP.

These implementation details underscore the need for a platform-independent MAC layer architecture. The errors we made highlight the inherent complexity of low-level, timing critical code, and were exacerbated by the relative ineffectiveness of traditional debugging tools like GDB in such critical sections of code. It is paramount that capable developers be able to produce definitive, well-tested implementations of components required by MLA, which can be reused many times by less-experienced developers on different platforms. Though the potential for unexpected platform/MAC layer interactions still exists in our architecture, we expect that they will be far less common than the bugs produced by grafting a monolithic MAC protocol into a new radio stack, since this involves essentially reimplementing the MAC layer from scratch. Moreover, fixes to radio layer bugs are automatically "inherited" by all protocols which use MLA, whereas they must be merged manually into each monolithic MAC layer implementation.

5 Evaluation

In this section, we provide an empirical performance evaluation on TelosB motes. We measure the amount of code reuse across all five MAC protocols implemented on MLA, and compare the code size, throughput, latency, and radio duty cycle of our new implementations of B-MAC and X-MAC against the same protocols implemented in the TinyOS 2.0.1 cc2420 radio stack. We include benchmark results for a version of the TinyOS 2.0.1 cc2420 stack augmented to use our portable CCA check, thereby demonstrating the effects of this change independently from our architecture. We also present the performance results of SCP, Pure TDMA, and SS-TDMA, to validate MLA's ability to support TDMA and hybrid protocols[2].

5.1 Code Reuse and Footprint

A primary goal of MLA is to significantly reduce the effort required to implement a new MAC protocol. To quantify the reduction in developer effort, we measure the amount of component reuse across different MAC protocols implemented in MLA. Table 5.1 lists the MLA components de-

[1] We did not wish to use the entire component, since parts were fairly specific to the built-in B-MAC and X-MAC implementations.

[2] It is not possible to make a fair comparison of B-MAC, X-MAC, and SCP with their original monolithic implementations presented in [6], [16], and [5], respectively, since their implementations were based on TinyOS 1.0 and different hardware platforms.

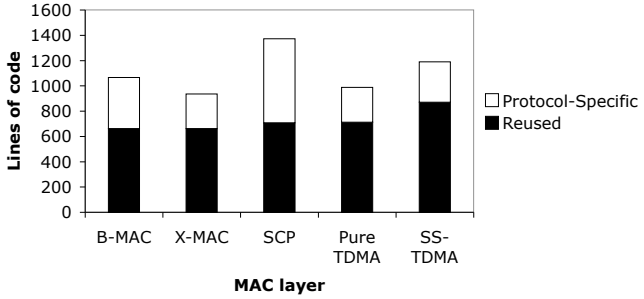

Figure 8. The proportion of reusable code in each MAC protocol

scribed in Section 3.3 and indicates their use in our implementations of B-MAC, X-MAC, SCP, Pure TDMA, and SS-TDMA. We also note the number of components specific to each protocol as discussed in Section 4. In all five MAC layers, the generic components outnumber the newly-implemented components.

	B-MAC	X-MAC	SCP	Pure TDMA	SS-TDMA
Channel Poller	X	X	X		
Fixed Sleep LPL Listener	X	X			
Periodic LPL Listener			X		
Preamble Sender	X	X	X		
Time Synchronization			X	X	X
TDMA Slot Handler				X	X
CSMA Slot Handler					X
Low Level Dispatcher				X	X
AsyncIOAdapter	X	X	X	X	X
Alarm	X	X	X	X	X
Local Time			X	X	X
Radio Core	X	X	X	X	X
Reused Components	**6**	**6**	**8**	**7**	**8**
Other Components	**3**	**3**	**4**	**2**	**2**

Figure 8 summarizes the number of lines of code in the MLA components and protocol-specific components in each protocol[3]. The contribution of these reusable components to the overall codebase ranged from 51% to 73%, indicating a significant savings in developer effort.

Table 3 examines the impact of using MLA on binary code footprint. We list the ROM and RAM size of our throughput benchmark application (discussed in Section 5.2) when compiled for the TelosB motes, as reported by the TinyOS toolchain. MLA has comparable RAM usage to the monolithic stack, but consumes about 2 KB more ROM for both B-MAC and X-MAC. Our radio stack's new CCA check routine, which uses the `LocalTime` interface to obtain accurate hardware counter information, accounts for these two kilobytes. Rolling back our new CCA check to the original code makes the code size in ROM close to the monolithic stacks. However, as noted in Section 4.6, the original CCA routines are inherently non-portable across CPU platforms. We therefore opted to continue using our enhanced CCA routine despite the code size increase. Additionally, ROM is generally not a scarce resource: applications are more likely to be constrained by RAM, where we perform comparably, and developer effort, which MLA reduces significantly over the monolithic approach through code reuse.

[3]This figure does not include the augmentations to the `cc2420` radio stack described in Section 3.4, since these lines of code are woven into the stack and hence difficult to count accurately.

	ROM	RAM
B-MAC (UPMA)	20136	927
B-MAC (UPMA w/orig CCA)	18338	921
B-MAC (monolithic)	17586	922
X-MAC (UPMA)	19854	876
X-MAC (UPMA w/orig CCA)	18000	864
X-MAC (monolithic)	17408	866
SCP (UPMA)	21372	1056
SCP (UPMA w/orig CCA)	19534	1050
Pure TDMA (UPMA)	20304	918
Pure TDMA (UPMA w/orig CCA)	18494	910
SS-TDMA (UPMA)	20912	932
SS-TDMA (UPMA w/orig CCA)	18974	926

Table 3. The ROM and RAM footprint of various MAC implementations in bytes

5.2 Throughput

To measure the throughput of the MAC protocols, we deployed each protocol on one recipient node, and on 1–10 sender nodes arranged in a 1 m-diameter circle around the recipient. The sender nodes sent packets with 28-byte payloads addressed to the recipient in a tight loop, as fast as permitted by the MAC layer. The recipient counted the number of packets received successfully over the course of 60 seconds. This experimental setup emulates the throughput experiment presented in [6]. In the case of SCP, each node waited 15 additional seconds before the beginning of each run, to allow enough time for the nodes' clocks to synchronize. All nodes were configured with a sleep interval of 100 ms. Both Pure TDMA and SS-TDMA were configured with a slot size of 10 ms, and 16 slots per frame: including 1 synchronization slot, 10 active slots, and 5 sleep slots.

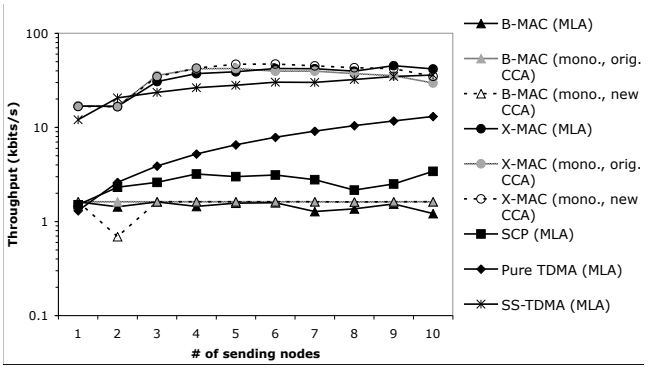

Figure 9. The throughput of various MAC implementations

The results are plotted in Figure 9. All three B-MAC implementations exhibit the lowest throughput of all protocols. For both B-MAC and X-MAC, the average throughput of the monolithic implementations is not significantly affected by the CCA routine used. The throughput achieved by the MLA-based B-MAC and X-MAC implementations are also comparable to their monolithic counterparts. These results demonstrate that the component-based architecture of MLA has negligible impact on MAC throughput.

Since no monolithic implementations currently exist for SCP in TinyOS 2.0, it is impossible to provide a fair comparison with its MLA implementation. Instead, we compare its MLA implementation to the rest of the protocols and verify that the results obtained are consistent with the expected operation of SCP. SCP's throughput is an average of 81% higher than MLA's B-MAC implementation. SCP is able to achieve higher throughput than B-MAC for two reasons. First, SCP performs an extra back-off between preamble and payload which B-MAC does not perform. This additional back-off interval reduces the number of collisions under high contention. Moreover, SCP includes optimizations for handling bursty traffic that allow nodes to transmit more than one packet per LPL interval.

Both TDMA protocols perform better than B-MAC and worse than X-MAC. Pure TDMA has a linear increase in throughput as the number of nodes increases, since exactly one send slot is allocated to each sender node. SS-TDMA achieved higher throughput than Pure TDMA since it allows unallocated slots to be stolen for sending. SS-TDMA's throughput increases as more nodes are added, since multiple nodes may be able to share the same stolen slots. Adding more nodes also shrinks the performance gap with Pure TDMA, since there are fewer unallocated slots. SS-TDMA's throughput levels off after more than 4 senders, due to increased contention for stolen slots.

5.3 Latency

We evaluated the latencies of B-MAC, X-MAC, and SCP on a multi-hop network. We placed 6 nodes in a line, spaced 1 m apart. The first mote injected a packet into the network every 3 seconds. Each subsequent mote forwarded the packet to its next neighbor. When a packet reached the end of the line, the last node reversed the packet's direction, and the nodes forwarded the packet back to the first node. We recorded each packet's round-trip time at each of the first five nodes. We computed the average latency and its standard deviation over 50 round trips.

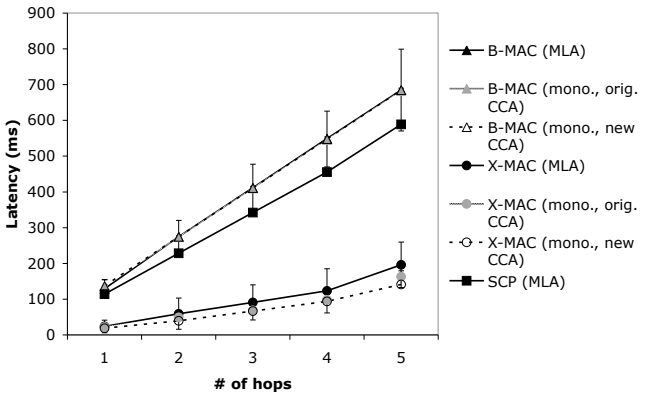

Figure 10. The average packet latency of contention-based MAC implementations

The results are presented in Figure 10. SCP serves as a useful baseline, since we expect the average latency to increase by 100 ms at each hop[4]. Our experimental results fit this prediction well. On rare occasion, a node would falsely detect activity on the radio, and hold onto the packet for an extra 100 ms before forwarding it. This adds a small average delay above the expected 100 ms at each hop.

Also as expected, B-MAC and X-MAC have linearly-increasing average latencies, with X-MAC having a much smaller slope. Our implementations of B-MAC and X-MAC perform comparably with the monolithic implementations. Combined with the throughput measurements presented in the previous section, these results demonstrate that the performance impact of MLA's component-based architecture on MAC protocols is negligible.

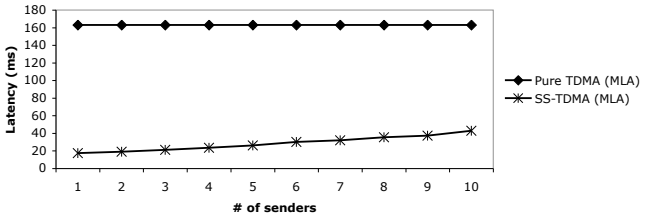

Figure 11. The average packet latency of TDMA-based MAC implementations

Since Pure TDMA and SS-TDMA are designed for single-hop networks, we evaluated their latency in a separate single-hop network. For these protocols, we repeat the throughput experiments performed for B-MAC, X-MAC, and SCP, and record the average latency of each packet transmitted. Again, we use a single sink node and vary the number of sender nodes from one to ten.

Pure TDMA has an average packet latency of 163.0 ms, regardless of the number of senders. This is because each node attempts to send a new packet immediately after the previous packet is sent. Because each node is only allowed to send one packet per slot, it must wait 160 ms for the next frame before it can transmit again. In contrast, SS-TDMA has an average latency of 28.6 ms. Unlike Pure TDMA, SS-TDMA's latency increases as the number of nodes increases. This is because nodes try to steal unallocated slots in order to transmit the packet more quickly. As the number of nodes increases, so does the contention for those slots.

5.4 Duty Cycle

To measure the duty cycle of the various MAC layer implementations, we augmented the radio stacks to count the time that various hardware components were active. Specifically, we used a 32 KHz hardware counter to record the time spent executing each of the three stages involved in turning on the CC2420 radio: enabling the voltage regulator, turning on the oscillator, and finally powering the radio into receive mode. We collected this data during a separate run of the latency benchmark application, which emulates a typical low-to-medium load application. We do not consider Pure TDMA or SS-TDMA in this experiment, as they have fixed duty cycles by design.

Figure 12 shows the average duty cycle of the six nodes. Our B-MAC and X-MAC implementations had 17% and

[4]Though SCP includes optimizations for reducing multi-hop latency, they only take effect when a single node produces multiple packets within one LPL interval.

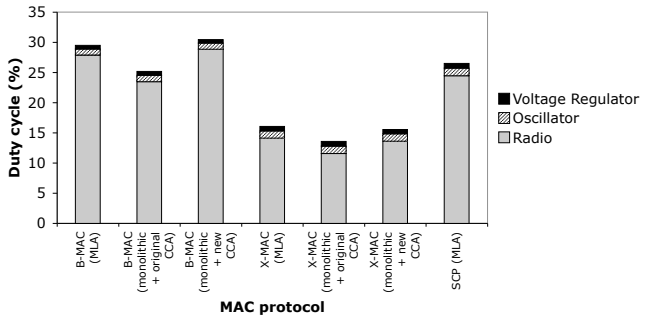

Figure 12. The average duty cycle of various MAC implementations

18% higher duty cycles than their respective monolithic counterparts. When the monolithic stack is augmented with the portable CCA routine, the difference in duty cycle between the two stacks becomes insignificant. Specifically, when using the same CCA routine, there is a 4% difference between the B-MAC implementations and a 3% difference between the X-MAC implementations. In the case of B-MAC, the MLA implementation actually outperformed that of the monolithic stack. These results indicate that the differences in the duty cycles are caused by our particular implementation of the CCA routine instead of the overall MLA architecture. We expect that the difference between the two CCA routines could be reduced by further tuning the portable CCA check's length and sensitivity.

The duty cycle is highly dependent on the senstivity of the CCA check routine. Tuning the duration of this CCA check allows the developer to achieve a desired tradeoff between performance and the duty cycle. As described in Section 3.4.2, MLA's `ChannelMonitor` interface allows developers to tune the duration (in ms) of the CCA check in a *platform-independent* fashion, using the underlying `LocalTime` interface. In contrast, platform-dependent hand-tuning of the CCA check, as is done inside of the monolithic `cc2420` stack, can potentially offer better energy efficiency at the cost of extensive re-tuning for each new sensor platform. In the future, we will consider ways for developers to optionally associate a MAC protocol implementation with a platform-dependent CCA routine within MLA. This capability would allow developers to achieve optimal energy efficiency on specific sensor platforms, while still permitting the MAC protocol to fall back on (potentially less-efficient) platform-independent tuning on other platforms.

Our implementation of SCP exhibits a 11% lower duty cycle than our B-MAC implementation. This difference is due to the fact that SCP can turn the sender's radio off after sending a short preamble and a single payload packet. However, because we have not yet implemented SCP's overhearing avoidance optimization, SCP still pays the penalty of keeping the radio on for the entire polling interval if it overhears a preamble. This prevents our implementation of SCP from achieving a duty cycle comparable to X-MAC.

6 Conclusion

We have developed MLA, a component-based architecture for the MAC layer. MLA consists of high-level, hardware independent components as well as low-level, hardware-dependent components. We have implemented MLA on the TinyOS 2.0.1 operating system, and evaluated its flexibility through the implementation of five representative MAC protocols that span the protocol design space. Empirical results show that our architecture can achieve up to 73% code reuse, while achieving comparative performance and memory footprint to monolithic implementations of the same protocols. These results demonstrate the reusability, flexibility, and efficiency of our component-based architecture in supporting a diverse MAC protocols for use in wireless sensor networks.

Acknowledgement

This work is supported by NSF NeTS-NOSS Grant CNS-0627126. We would also like to thank Lama Nachman and the reviewers for their valuable feedback.

7 References

[1] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica, "A unifying link abstraction for wireless sensor networks," in *SenSys*, 2005.

[2] K. Klues, G. Xing, and C. Lu, "Towards a unified radio power management architecture for wireless sensor networks," in *WWSNA*, 2007.

[3] ——, "Link layer support for flexible radio power management in wireless sensor networks," in *IPSN*, 2007.

[4] C. T. Ee, et. al., R. Fonseca, S. Kim, D. Moon, and A. Tavakoli, "A modular network layer for sensornets," in *OSDI*, 2006.

[5] W. Ye, F. Silva, and J. Heidemann, "Ultra-low duty cycle MAC with scheduled channel polling," in *SenSys*, 2006.

[6] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *SenSys*, 2004.

[7] W. Ye, J. Heidemann, and D. Estrin, "Medium access control with coordinated adaptive sleeping for wireless sensor networks," *IEEE/ACM Trans. Netw.*, vol. 12, no. 3, 2004.

[8] T. van Dam and K. Langendoen, "An adaptive energy-efficient MAC protocol for wireless sensor networks," in *SenSys*, 2003.

[9] IEEE Computer Society, "Part 15.4: wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (LR-WPANs)," 2003.

[10] I. Rhee, A. Warrier, J. Min, and L. Xu, "DRAND: distributed randomized TDMA scheduling for wireless ad-hoc networks," in *MobiHoc*, 2006.

[11] I. Rhee, A. Warrier, M. Aia, and J. Min, "Z-MAC: a hybrid MAC for wireless sensor networks," in *SenSys*, 2005.

[12] G.-S. Ahn, E. Miluzzo, A. T. Campbell, S. G. Hong, and F. Cuomo, "Funneling-MAC: A localized, sink oriented MAC for boosting fidelity in sensor networks," in *SenSys*, 2006.

[13] 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 *PLDI*, 2003.

[14] P. Levis, "TinyOS 2.0 overview." [Online]. Available: http://www.tinyos.net/dist-2.0.0/tinyos-2.0.0/doc/html/overview.html

[15] [Online]. Available: http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/wustl/upma/

[16] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks," in *SenSys*, 2006.