Evaluating the Performance of Synchronous and Asynchronous Media Access Control Protocols in the Contiki Operating System

A Major Qualifying Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

in Computer Science

by

Christopher Pinola

December 2012

Approved:

Professor Robert Kinicki, Advisor

Abstract

Contiki, a lightweight operating system for wireless sensor networks (WSNs), utilizes Radio Duty Cycling (RDC) to conserve sensor battery power. This project implemented and evaluated WPI-MAC, a reimplementation of the Crankshaft Media Access Control (MAC) protocol. Implemented as a new RDC driver within the Cooja simulator, WPI-MAC was compared via simulation against: ContikiMAC and X-MAC. Preliminary results indicate that under three common WSN traffic patterns ContikiMAC outperforms WPI-MAC in terms of energy efficiency and delay, but WPI-MAC yields lower average delays compared to X-MAC.

Acknowledgements

I would like to thank Professor Robert Kinicki for his insight and guidance throughout the course of this project. Furthermore, I would also like to thank fellow students, Kerry Lynn and Shary Llanos Antonio, for sharing their knowledge and experience.

Contents

List of Figures

List of Tables

1. Introduction

Wireless sensor networks (WSNs) are a relatively new field of research in computer science with many broad and exciting possible applications. WSNs monitor the structural integrity of bridges, observe tidal conditions, measure air quality, and even detect disasters like fires and landslides. These networks are made up of nodes, known as motes, which generally consist of a power source, wireless radio, microcontroller, and array of sensors. The mote networks are often deployed in harsh environments where it is not feasible to attach them to a source of constant power. This limitation makes it imperative to manage the limited supply of energy to these motes in a highly efficient manner. The mote wireless transceiver is often the most wasteful in terms of energy consumption and efficiency. The TelosB platform is a popular hardware standard for wireless sensors. While the TelosB unit is active, the processor draws a current of 1.8 milliamperes and the wireless transceiver draws 23 milliamperes [1].

Managing a wireless transceiver in an energy efficient manner in a mote is inherently challenging due to its very nature: it needs to be powered in order to transmit or receive packets. Furthermore, depending on the application, the mote may utilize its transceiver in an unpredictable fashion. A wireless transceiver consumes nearly the same amount of power while passively listening for messages over the air as when actively sending data [1]. Therefore, the only approach to limiting the power consumption of a transceiver is to power it off entirely [2]. Unfortunately, this introduces an entirely different set of problems related to reliable communication between multiple nodes. To combat these issues, Media Access Control (MAC) developers utilize a strategy known as duty cycling in WSNs to intelligently regulate when a mote's wireless transceiver is in the active power state.

Contiki is a recently developed open source operating system intended for use within wireless sensor networks [3]. Traditionally, duty cycling mechanisms are built into the MAC layer of the operating system's networking stack. Contiki, however, employs duty cycling tactics in a separate level of the networking stack known as the Radio Duty Cycling (RDC) layer. Currently, Contiki provides two RDC protocols that follow the asynchronous paradigm, which relies heavily on low-power probing and low-power listening (LPL). Conversely, synchronous protocols rely on tightly synchronized clocks and predictably repeating events to minimize wireless transmission collisions. This project introduces an existing synchronous protocol into the newer Contiki environment and analyzes its performance compared to X-MAC and ContikiMAC, the two asynchronous MAC protocols supported by the Contiki network stack.

This investigation is a continuation of a previous MQP (Major Qualifying Project) by Worcester Polytechnic Institute students Bates & Keating [4]. They implemented several MAC (Media Access Control) protocols for the TinyOS operating system. However, in recent years, TinyOS has begun losing popularity in the research community. Starting from the Bates & Keating implementation of Crankshaft in TinyOS on TelosB mote, this research updated this power-aware MAC protocol to work in Contiki. When Bates & Keating tested their protocols, they focused on minimizing WSN power consumption for three standard WSN communication traffic patterns. However, while minimizing energy usage is a key factor in the design of WSN MAC protocols, sensor message delay is also extremely important when assessing the overall performance of a MAC protocol. For the purposes of this report, delay is intended to be synonymous with latency.

The updated version of the Bates & Keating Crankshaft protocol developed for this investigation and, called WPI-MAC in this report, was debugged, and tested in the Cooja simulation environment for the Contiki operating system. Along with X-MAC and ContikiMAC, WPI-MAC was evaluated on three different types of network traffic: broadcast, local gossip, and convergecast. These tests were focused on measuring the power consumed by each protocol as well as the average packet delay over a simulated 11-node one hop wireless sensor network.

2. Background

This chapter provides definitions and explanation for several of the standard terms, concepts and conventions used in the literature to explore current wireless sensor network research. The chapter devotes sections to provide background information about Contiki, the Cooja simulator and a brief review of the power-aware WSN MAC protocols that are important in understanding where WPI-MAC fits within current WSN research.

2.1. Wireless Sensor Networks

Wireless sensor networks (WSNs) can be defined as a collection of wireless sensors, also known as motes intended to operate cooperatively via message transmissions to address one specific application task. While the breadth of applicable WSN applications has expanded considerably over the last decade, employing WSNs to monitor environmental conditions remains as one of the most common and viable methods to undertake this challenging task. Currently, motes are available for purchase from a few online electronics retailers for approximately $140 USD. While the prices of these will likely continue to diminish as demand increases, they are already cost effective enough for some businesses and researchers to begin implementing them. These motes contain several sensors capable of monitoring conditions from ambient light intensity to air temperature and moisture. Thanks to the inclusion of general purpose I/O (GPIO) pins on most motes, these devices can also play host to virtually any other variable resistor that a resourceful individual is willing to connect to them. In addition, they also include a small microprocessor, some RAM, and a wireless radio all on a small printed circuit board.

Unfortunately, the most appealing qualities of a sensor mote are also the ones which make them challenging to develop software for. From a computer science perspective, the two most challenging facets of coding software for a mote are their limited amounts of memory and constant need to conserve energy. Much research has gone into techniques for minimizing the amount of memory needed to operate a mote, but this project was significantly less concerned with that challenge. However, this project addresses the mote's need to converse as much power as it possibly can. These devices are often powered by a pair of AA batteries and are expected to sustain for several months at a time without replacement. Due to these strict power constraints, it is important that the operating systems used by these devices are optimized to maximize the device's lifetime.

Former Worcester Polytechnic Institute students who tested four scheduled MAC protocols (AS-MAC, SCP-MAC, Crankshaft, and BAS-MAC), found that certain protocols worked better for different categories of traffic [5]. They came to the conclusion that wireless sensor MAC protocols should be chosen based upon the application of the motes. In particular, they found AS-MAC to perform best when used with local gossip and convergecast traffic and SCP-MAC to dominate when it comes to broadcast traffic. As for the other two, Crankshaft and BAS-MAC were found to be a decent compromise between the different traffic patterns [4].

2.2. Contiki

Contiki is a relatively new open source operating system which "allows tiny, battery-operated low-power systems [to] communicate with the Internet" [6]. Contiki was built primarily by a researcher from the Swedish Institute of Computer Science, Adam Dunkels. While still heavily involved with the project, the open source community has embraced his work and continued to build upon his contributions. Contiki is special in that it is designed specifically to accommodate for the limiting factors of motes in wireless sensor networks. By present-day standards, the computational power of a mote is substantially less than that of a personal computer. For example, the average mote is outfitted with an 8-bit microcontroller and tens of kilobytes of RAM. It is with that in mind that Contiki was built to be incredibly lightweight [3].

While initially not very popular, it is now starting to gain some traction in the wireless sensor networking world due to the impressive list of features it has over its competitor, like TinyOS. The fully-compliant IPv6 stack found in Contiki was certified (and contributed to) by Cisco [6]. In addition to support for IPv6, Contiki also brings newer, low-power protocols like 6lowpan, RPL, and CoAP to its users. If a wireless sensor implementation does not need the full IPv6 stack, users can elect to use Contiki's lightweight Rime networking stack. This will break compatibility with the traditional Internet, but can reduce resource utilization on the mote in appropriate scenarios.

2.2.1. Differences from TinyOS

Up until a few years ago, TinyOS was the only wireless sensor operating system available. Originally developed by a team of researchers from the University of California at Berkeley, it is credited as being the first operating system specifically for wireless sensor networks. Above all else, it strives to achieve the tiniest memory footprint possible. One of the ways in which the developers achieved this was by electing to code the operating system in nesC, a dialect of the C programming language. nesC is intended for use in small, embedded networking devices and thus seems like a logical choice for a wireless sensor network operating system. Among the modifications to C that can be found in nesC is the concept of split-phase operations. Split-phase operations aim to eliminate blocking-wait calls which can seize control of the processor and waste valuable CPU time, and thus power. Unlike traditional C, nesC divides methods at the point where they invoke a lengthy system call. Due to this segmentation, the developer now becomes responsible for writing callback methods which are executed upon the completion of these system calls which are often unpredictable in duration. For example, if the developer wishes to turn the wireless radio on, they invoke the *on* method for that radio and then write the next part of their program logic in the radio's *onComplete* callback method. This callback method is automatically invoked once the hardware has reported that the wireless radio has either been powered on or an error occurred while attempting to do so.

Contiki abandons nesC in favor of the standard C programming language and GNU development tools. nesC, and thus TinyOS, require a modified set of tools in order to build the operating system and user programs. This is just an added nuisance to the developer who now must maintain a separate development environment when dealing with TinyOS. It is for that reason that TinyOS is, by default, packaged and distributed as a virtual machine appliance. Of course, a developer may elect to install the necessary nesC components themselves but this is actually strongly discouraged by members of the TinyOS community. For the sake of convenience Contiki is also released as a virtual machine variant, dubbed Instant Contiki.

2.2.2. The Radio Duty Cycling Layer

Strategies for minimizing the duty cycle of a radio are typically employed at the Media Access Control (MAC) layer of the networking stack in an operating system. However, the designers of Contiki recognized the importance of duty cycling and actually split the MAC layer of their networking stack into two components: MAC drivers and RDC (Radio Duty Cycling) drivers. A MAC driver in Contiki has the highly specific goal of detecting radio collisions (as reported by lower layers) and utilizing the RDC driver to retransmit a packet that is known to be lost. Currently, there is only one MAC driver in Contiki, CSMA (carrier sense multiple access). The CSMA driver in Contiki maintains a buffer of packets bound for a fixed number of hosts. For the sake of conserving memory, this is usually set to four. When the user-level process wishes to send a packet, it gets sent to the CSMA layer first. This MAC driver places the packet into a queue with other packets that are destined for the same recipient. The MAC layer then submits the first packet in the queue to the lower RDC driver. It is the responsibility of the RDC driver to ensure that the wireless transceiver remains powered off for as long as possible. Therefore, it is also the RDC driver's responsibility to make sure that the mote wakes up when it needs to in order to send and listen for radio activity [7]. After attempting to send the packet, the RDC driver invokes a callback method to the MAC driver indicating what happened to that packet. If the packet was successfully sent, it gets removed from the MAC queue, otherwise it stays and is scheduled for retransmission.

In order to develop a new RDC driver, one must implement several methods as defined by the Contiki RDC driver interface, the most important being the initialization method, the send method, and the incoming packet method. When the RDC send method is invoked by the MAC driver, the MAC driver first ensures that the packet it intends to have sent is correctly loaded into Contiki's *packetbuf* mechanism. This mechanism is used to align the packet in a contiguous region of memory and transfer that memory address to the appropriate radio register. Fortunately, Contiki also provides the *queuebuf* mechanism which is capable of capturing all the state about an outgoing packet if its transmission needs to be deferred in any way. This proved to be extremely useful during the development of WPI-MAC.

2.3. Media Access Control Protocols

The protocols used in wireless sensor networking media access control can be classified as either synchronous or asynchronous protocols. The term is in reference to when a mote transmits a packet over the radio medium relative to when the program presents this packet for transmission. These protocols attempt to balance power consumption with latency, often sacrificing delay to achieve lower power consumption.

Asynchronous protocols are predominately used in Contiki due to the fact that the developers have been able to demonstrate the diminished usefulness of synchronous MAC protocols in wireless sensor networks [2]. The argument against this class of protocols is that they are very susceptible to ide listening and overhearing. Idle listening is the condition in which a wireless transceiver is powered on and listening to a radio medium in which no activity is occurring. Overhearing refers to the reception of packets bound for another mote. For example, while idly listening to a channel, *Mote B* overhears *Mote A* send a packet to *Mote C*. This is a significant waste of energy because *Mote B* will immediately discard the packet it just spent energy receiving.

2.3.1. Synchronous Protocols

Synchronous protocols typically divide time into subunits in which certain motes are allowed to conduct radio operations. A program requesting to send a packet has no bearing on when the RDC driver will actually transmit it over the air. Synchronous protocols schedule actions in a predictable or cyclical fashion, rather than try to dynamically adjust to varying conditions. A major benefit of a synchronous protocol is that it provides a guaranteed maximum amount of time that a RDC driver will have to wait before transmitting its data. This factor helps keep delay under control. Furthermore, allocating periods of time to specific actions can reduce potential transmit and receive collisions. However it can also be problematic, as waking up a mote when it has nothing to send or receive, known as idle listening, can be wasteful from an energy perspective.

Being that synchronous protocols are time sensitive, they rely heavily on having their clocks synchronized across nodes. This is accomplished through the use of control packets introduced by the MAC protocols to ensure that clocks don't drift away from a common standard. Some protocols also utilize the control packet as a way of sharing scheduling data among neighboring sensors. This is used in scenarios when a node wishes to advertise details about its wakeup cycle to its neighbors. Its neighbors can then utilize this information to intelligently decide when to attempt to contact that node. Crankshaft, and thus WPI-MAC, are synchronous protocols which will be described in more detail later. While synchronous protocols work well and are relatively easier to implement, they are not the only solution to this problem.

2.3.2. Asynchronous Protocols

Asynchronous protocols don't concern themselves with clock drift and rigid timeslots. Rather, they focus on being able to wake up on-demand and send whenever they please. There are a few ways to go about accomplishing this, but a common tactic is to employ low-power listening. Low-power listening (LPL) is a technique which a wireless transceiver can use to get a sense of what is happening in the radio medium without having to continuously turn the radio on. It can be thought of as a more energy efficient, but less reliable, clear channel assessment (CCA). By measuring the received signal strength (RSSI) of the radio channel, the mote can infer whether or not there is currently a radio transmission occurring. This is due to the fact that the RSSI will spike when the radio hears another node transmitting.

Asynchronous protocols will employ low-power listening in either a sender-initiated or receiver-initiated fashion. In the sender-initiated model, when a node wishes to communicate with others, it begins transmitting a series of strobe packets when it detects that the air is available. These strobe packets are tiny and simply indicate to others that the sending node has a larger packet to send. The other nodes periodically wakeup to listen if they detect a packet transmission. If they detect a strobe, they analyze it to determine who the sender wishes to communicate with. If the sender intends to send a packet to the node, the intended receiver will remain on and send an acknowledgement (ACK) back to the sender, otherwise it goes to sleep. When the sender receives the acknowledgement packet, it knows that it is now okay to transmit the entire packet to its target.

One such protocol is ContikiMAC, included with Contiki as an RDC driver. ContikiMAC makes a few alterations to the traditional asynchronous protocol. When a node wishes to send in ContikiMAC, it simply begins transmitting the entire data packet until it receives an acknowledgement [2]. This assures the sender that the recipient was awake and received its transmission at the time it sent the data packet. After this exchange is completed, the nodes quickly return to sleep. This protocol, however, behaves differently when broadcast traffic is involved. In the event that a ContikiMAC node needs to transmit a broadcast packet, it will repeatedly transmit the entire data packet for one entire period [2]. This ensures that no matter when the other nodes' wake up to check for activity, they will have all received the entire packet at least once. Most notable about this approach is that ContikiMAC will continue to send the packet even if all of the intended recipients have received it.

Receiver-initiated variants behave similarly, but the roles are somewhat reversed. Instead of sending strobes when a node is ready to send, potential receivers will instead advertise that they are eligible to receive packets. If a potential sender hears that its target is awake, it will respond with an acknowledgment instructing the target to stay awake for its incoming transmission, or simply send the packet.

X-MAC is a protocol from this category. Before transmitting a packet, X-MAC will send out short preambles ("strobes") naming the intended recipient of its pending transmission. Other nodes in the network listen for this preamble and then quickly determine if they are the intended target or not. If they are, they remain powered on and await the packet, if not they return to their sleep state. At this point, a target node will send an ACK to the sender to signal the start of the transmission. The sender dispatches the data packet as soon as it hears the acknowledgement as it now knows there is a recipient awake and waiting for the packet [8]. In the Contiki implementation of X-MAC broadcast messages are handled in a fashion similar to ContikiMAC. Instead of sending strobes, a node wishing to send a broadcast transmission will simply send the data packet repetitively for the duration of the period [9].

These types of protocols are significantly more challenging to implement but will drastically reduce the problem of overhearing. However, unlike their synchronous counterparts, they offer no guaranteed maximum delay. While they can be less wasteful in terms of energy, it usually comes at the expense of increased latency.

2.4. Cooja: The Contiki Simulator

One of the major advantages to the Contiki operating system for sensor software developers is the inclusion of the Cooja simulator environment. Cooja is a feature-rich simulator which allows developers to virtually deploy an environment of motes and gather a host of metrics about each one. Additionally, Cooja is capable of emulating several hardware platforms that Contiki could be compiled for. Better yet, it can also simulate a wireless radio medium, enabling developers to debug their applications and behave how they would function in a real-world scenario. Perhaps most importantly for a project such as this, it is bundled with Instant Contiki and is completely free of charge. For these reasons, all debugging and experimentation for this project occurred in the Cooja simulator. Theoretically all results could be duplicated on physical hardware, but due to time constraints could not be performed during the course of this project

Cooja enables the user to have control over a few parameters for the simulation environment. To account for interference and packet loss, one can modify the quality of the radio medium and even adjust, on a mote-by-mote basis, the probability with which that mote will successfully send or receive a packet. Cooja also allows for a maximum mote startup delay to be set along with a random seed. These parameters are used to calculate a random amount of time that it will take each mote to complete its boot sequence and begin executing its program. This accounts for the time that would be taken to connect each mote to a source of power. If all the motes are being powered by AA batteries, the odds of them all being inserted by a human being and booting at the same exact instant are slim.

To ease the development of some operating system components, like WPI-MAC, this can be disabled so that all motes begin executing at the same exact moment in time. The settings used for the development and testing of WPI-MAC are explained in greater detail later in this report.

3. Implementation

The primary accomplishment of this investigation was the development of WPI-MAC, an RDC driver for Contiki. This process presented an interesting set of challenges but also yielded a deep understanding of the Contiki operating system. Perhaps the greatest hurdle in developing a new RDC driver was gaining familiarity with the organization of the various networking modules present in Contiki. The published documentation for Contiki is still in its infancy and often fails to reflect the most current release of the operating system. The networking modules in particular have changed drastically in the past few minor releases of Contiki, yet fail to be described in the documentation. Fortunately, because Contiki is an open source project, one can search through the source tree in order to answer questions that the documentation cannot.

To aid in development, Contiki includes a *nullrdc* driver, which is an RDC driver containing everything required to compile the networking stack and nothing more. This RDC driver actually does not perform any duty cycling and simply leaves the radio powered on at all times. It provides the scaffolding necessary to build a new driver by providing templates for the absolutely crucial methods that an RDC driver is expected to provide. These methods are utilized by the operating system and executed in response to specific events. The most significant of these events are triggered by system initialization, requests to power the radio down or turn it back on, requests to send a packet, and the radio receiving a packet.

WPI-MAC began as a copy of the *nullrdc* driver and was developed from there. The most important aspect of this driver was the rigid and precise time events that distinguish Crankshaft. In Crankshaft, no node is ever permitted to transmit outside the bounds of its respective time slot. Therefore it is imperative that each node in the network transitions to a new slot at the exact same time. Contiki provides a handful of timer mechanisms, such as *etimer* and *ctimer*, which can be preempted by other system events. Fortunately, the *rtimer* construct was recently added. This new mechanism behaves like *ctimer* but is given precedence above all other components in the system. Namely, when *rtimer* expires, its resulting callback method will preempt any other running process [10].

The strategy with which *rtimer* is employed in WPI-MAC is straightforward: the timer is used to signal the start of each transmission slot. When the timer expires, it begins executing the *advance_slot* method. This method is responsible for scheduling the timer to begin again, indicating the start of the next successive slot. Additionally, this method decides what the node needs to accomplish during the time slot based on its hardware address and the identification number of the current slot. This method is responsible for switching the wireless transceiver on or off depending on the context. For example, the current slot may be the broadcast slot and the node has no packets to send, but because it is the broadcast slot, the transceiver still needs to be powered. Therefore this method will invoke the radio's *on* method if it is not already active. If it is the case that the node must use this slot to transmit a packet, then in this scenario, the method also serves as a dispatch mechanism. This is achieved by checking WPI-MAC's built-in queuing system for any pending outbound packets. The queuing system is implemented as a first-in-first-out linked list which is used to retain entire packets until they can be transmitted at an appropriate later time. The majority of the queue manipulation occurs in the *send_packet* and *real_send* methods.

Admittedly, the selection of the transmission window duration for this project was arbitrary. It was intentionally set to be longer than necessary to help ensure that the protocol was performing in accordance with the Crankshaft specification. While the protocol works as intended, as shown later, it does suffer from high latencies and unnecessary idle listening that could likely be reduced by adjusting this parameter.

The *send_packet* method is one that all Contiki RDC drivers must include. It is invoked by higher layers of the Contiki networking stack wishing to send a packet to the radio, and eventually, transmit to another node. The WPI-MAC variant provides scheduling for the protocol. When *send_packet* is called, the method is provided with a pointer to the outbound packet header and a callback method to be executed after attempting to send the packet. The initial phase of *send_packet* allocates a *queuebuf* structure to house the header and payload of the outbound packet. The contents of the packet are then encapsulated inside a WPI-MAC-specific queue structure, and placed at the end of the appropriate queue. WPI-MAC maintains a global array of these queues, each representing the collection of packets destined to be transmitted during a specific slot. For example, the queue located at position *zero* in the array of queues contains a linked list of the packets that need to be transmitted during the broadcast slot.

The *real_send* method is where the actual transmission of packets occurs. This is not registered with the operating system like some of the other methods, but rather the method is used internally by WPI-MAC's scheduling system. When a new slot begins and *advance_slot* determines that there is a pending outbound packet for the current slot, it invokes *real_send*. The primary function of *real_send* is to copy the outbound packet from the WPI-MAC queue back to the radio and attempt to transmit it over the radio medium. Before this can occur, the node must win the contention period. Each time slot is divided into four relatively short contention windows, followed by a longer message exchange window. As described previously, the node will randomly choose one of these four contention windows beginning the transmission of the filler packet. This differs from the original Crankshaft which employs a weighted probability protocol called *Sift* to choose a contention window.

Four contention windows were chosen in order to provide a reasonably low rate of collision without negatively impacting latency too profoundly, given the number of transmission slots. The goal is to be the node that begins transmitting the soonest. If a node is attempting to

Parameter	Value
Transmission Slots	12
Broadcast Slots	1
Unicast Slots	11
Transmission Slot Duration	15 ms
Total Period Duration	180 ms
Contention Windows	4
Contention Window Duration	2 ms
Message Exchange Window	7 ms

Table 3.1 WPI-MAC Parameter Specification

send during the current slot and hears another node transmitting its filler packet, it knows it has lost the contention window and forfeits the message exchange window. In this scenario, the packet's callback method is invoked including its status, indicating that the packet could not be transmitted due to congestion. If multiple nodes attempt to seize the same contention window, they will interfere with each other and no nodes will have access to the message exchange window; this is the worst case scenario. When this occurs, each node will report a failure when calling its packet's callback method. Optimally, one node will transmit its filler packet uninterrupted and have exclusive access to the message exchange window. During that time, it will proceed to transmit the original packet over the radio medium. Theoretically this attempt should succeed due to the privilege that the node has earned to occupy the message exchange window. Regardless of what transpires, the packet's callback method is invoked along with its resultant status.

Once the packet has been transmitted, the sender does not wait to receive a positive acknowledgement (ACK) from the recipient. Due to the inclusion of the sender contention mechanism and lack of receiver collisions offered by the Crankshaft model, it was decided that ACKs would be superfluous and only serve to increase latency overall. Ultimately, the *real_send* method is responsible for removing the packet from the internal queue collection in order to avoid accidental retransmission.

For quick reference, Table 3.1 has been included to detail the most important parameters related to WPI-MAC. It is worth noting that these parameters are not hard-coded into the driver source code, but rather symbolically defined as preprocessor directives. Modifying one of these symbols and recompiling Contiki will cause the updated value to take effect.

4. Methodology

Using the TinyOS version developed by Bates & Keating as inspiration, this project re-implemented the Crankshaft MAC protocol without ACKs for the Contiki operating system. Due to the more logical segmentation of media access control logic in Contiki, this implementation was realized as a radio duty cycling driver. This RDC driver was compiled into Contiki as a Sky mote firmware, which could then be flashed onto a virtual device inside the Cooja simulator.

4.1. Simulation Environment

As previously mentioned, all experimentation was conducted in the Cooja simulator. The version used was the one bundled with Instant Contiki 2.6. The Instant Contiki image was run on a virtual machine provided by Oracle's VirtualBox 4.2 hypervisor. The host machine was running the 64-bit edition of Windows 8 Pro powered by an Intel Core i7-3770K. The virtual machine was allocated 3 GBs of RAM and two hyper-threaded CPU cores clocked at 3.5 GHz, for a total of four logical processor cores. No modifications were made to the Instant Contiki image and no operating system updates were applied to the bundled installation of Ubuntu.

The Cooja settings were left at their default values. For every simulation, the Unit Disk Graph Medium was selected as the radio medium with a mote startup delay of 0ms. This was done to ensure that all motes would start with a synchronized clock, which is important because at present, WPI-MAC does not include a mechanism to prevent clock drift. Each simulation consisted of 11 Sky motes all located within radio range of each other. As motes are placed about the simulation environment, Cooja provides feedback about the probability that each node has of successfully transmitting and receiving. All motes were placed within an area no greater than one square meter and were indicated by Cooja to have a 100% change of successfully sending and receiving data to all other motes in the simulation. The only aspect of the Sky platform that was modified was the active RDC driver depending on the experiment. All simulations were allowed to run for 10 minutes of simulated time, which was often less than the wall clock duration. To facilitate easier replication, the firmware used on the virtual motes were only slightly modified versions of the sample programs that come bundled with Contiki. The only change made to the sample code was the inclusion of timestamps on the sending and receiving of packets, which was done simply to make delay easier to calculate. In order to measure power consumption, the project leveraged the tools included in Cooja to produce statistics regarding how much time the mote radios spent in each state.

4.2. Tested MAC Protocols

In order to gauge how the reimplementation of Crankshaft (WPI-MAC) performed, it was benchmarked against the two most popular RDC drivers in Contiki: ContikiMAC and X-MAC.

4.2.1. WPI-MAC

This project resulted in the development of a new synchronous MAC protocol based on Crankshaft, tentatively named WPI-MAC, which was implemented as an RDC driver in the Contiki operating system. The code left behind by Bates & Keating (which includes an implementation of Crankshaft) was used as inspiration for this version. However, their drivers were written in nesC for the TinyOS environment and needed to be adapted to work with Contiki. The Crankshaft protocol gets its name from an internal combustion engine, comparing the fixed wireless transmission slots that it gives to its receivers to the fixed offset between the start of the rotation of a crankshaft and the moment that a piston fires. In Crankshaft, time is divided into frames, which each can be thought of as a fixed period of time. Within every frame, there are a fixed number of slots which are intended to be used as listening windows. Each slot is either a broadcast slot or a unicast slot. For broadcast slots, all nodes in the neighborhood will wake up to receive a broadcast message, regardless of whether or not there is one to be transmitted. Conversely, a unicast slot is a period of time assigned to one specific node (by hardware address) in which that node stays on to receive packets from neighboring sensors. Within each slot, there is a brief contention window, followed by a much longer message exchange window. When a sender wishes to send to another node, it must first “win” the contention window in that specific node’s slot, and then send its message during the message exchange window [11]. In an effort to reduce power consumption, nodes that are scheduled to wake up in order to listen in WPI-MAC do not do so until after the contention window has passed. As one would imagine, this protocol depends heavily on each node in the neighborhood knowing the others’ schedule and having a synchronized clock mechanism. For the purposes of this project, schedule synchronization and node initialization were assumed to have successfully taken place ahead of time. Additionally, nodes will not enter or leave the neighborhood once the simulation has begun.

While Crankshaft served as the basis for the new protocol, some minor modifications were made. The new protocol does not adopt the Sift weighted-probability contention window mechanism found in the original Crankshaft. In WPI-MAC, instead there are four small contention windows which motes select with a random number generator. To increase randomness, no truncation or ceiling functions were used. Instead the domain of the random number generator was split into regions that corresponded to the contention slots. In WPI-MAC, there are twelve transmission slots per period: one broadcast slot, followed by eleven unicast receiving slots. This is to accommodate the simulation environment, in which there will be eleven virtual motes: a base station and ten clients. Lastly in terms of assumptions and modifications, WPI-MAC will only be concerned with single-hop environments. Requiring nodes to keep track of scheduling for multiple neighborhoods could become rather memory intensive. Furthermore, the complexity involved with implementing a multi-hop protocol is beyond the scope of this project.

4.2.2. ContikiMAC and X-MAC

Currently, ContikiMAC and X-MAC are the two most popular RDC drivers in Contiki. Both drivers perform relatively well and accomplish the goal of keeping their radios off most of the time, but perhaps they are too similar to one another. Both of them are asynchronous protocols built around the idea of low power listening. Low power listening is a mechanism that can be used to detect if there is radio traffic within range of a transceiver without drawing too much power. This essentially relies upon the radio's ability to provide a Received Signal Strength Indicator (RSSI), which should be significantly higher during times of radio activity.

The fundamental difference between ContikiMAC and X-MAC is that ContikiMAC is sender-initiated while X-MAC is receiver-initiated. In ContikiMAC, when a node wants to send a packet, it begins transmitting that packet repeatedly until it receives an acknowledgement from another node (or some timeout condition is met). This acknowledgement indicates that the packet has been successful received by the other node and allows the sender to go back to sleep [2]. Conversely, nodes wishing to send a packet using X-MAC are not permitted to send a packet until knowing that there is another node awake and ready to receive the packet. This is accomplished through the use of strobes which are essentially just short preambles. A sender will start sending strobes repeatedly until it receives an acknowledgement from another node [8]. This allows the sender to transmit its packet over the radio medium knowing that there is an extremely good chance of another node being awake to receive it. Upon the completion of sending, the sender goes back to sleep.

These protocols are regarded as the two best available for Contiki by the Contiki community of developers. Therefore they make excellent choices for evaluating the performance of a new protocol.

4.3. Benchmarking Traffic Patterns

This investigation uses three different traffic patterns that well represent standard applications of wireless sensor networks. It is important to note that across all of these tests, the sample packets used were very small and not large enough to cause the packet to be separated into multiple frames.

4.3.1. Broadcast

Broadcast traffic is defined as one node in the network transmitting a packet intended to be received by all nodes in the network. This project used a modified version of the broadcast example program found in the IPv6 samples directory in the Contiki source tree. The sample was modified so that broadcasts were initiated once per second and only sent by the base station. The payload of these broadcast packets simply contain the number 4, represented by a single unsigned 8-bit integer. Thus the payload size for the broadcast packet is a single byte. By default, the sample includes timestamps for when a node sends a packet and when it is received by the other nodes. The difference between these two timestamps is calculated for each packet and then averaged over the receiving nodes to determine the average latency for a given protocol.

4.3.2. Local Gossip/Unicast Conversations

Local gossip is defined here as multiple nodes in a network simultaneously having unicast conversations with one another. For the experimentation portion of this project, I was again able to leverage some of the included sample code to facilitate this test. The sample code has each node randomly choose a conversation receiving partner (other than itself) for a unicast message, sent at a random time over a ten second interval. The payload of this message is a 10-byte ASCII string. Being that each node is simultaneously executing this test, it is possible that, on occasion, multiple nodes will attempt to send to the same node. If anything, this is a more rigorous test of a protocol as it is a scenario that is likely to occur in a real-world application.

4.3.3. Convergecast

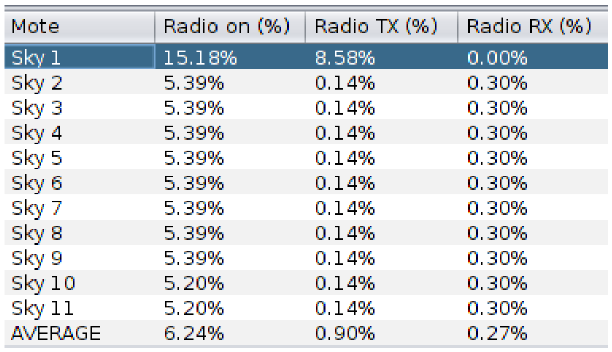
Convergecast is defined as having all nodes in a network send unicast transmissions to a single node. This node is often known as a base station or a sink. The test code used in this project mimics the way wireless sensors are typically employed. The base station periodically broadcasts a message to all the leaf nodes in the system once over a ten second period. At this point, all of the leaf nodes then attempt to transmit a unicast message back to the sink simultaneously, which is likely to be a source of contention amongst the leaf nodes. This message contains a 32-byte snapshot of the current values of all the motes sensors as well as some other data, such as the amount of time the node has been running. In an actual implementation, the base station might be connected to an Internet web server. When a user issues a request to the web server, the web server has the base station mote request the current status of all the other sensors in the network and report their findings back. The reported values are then served to the web server and back out over the Internet.

5. Results

5.1. Broadcast

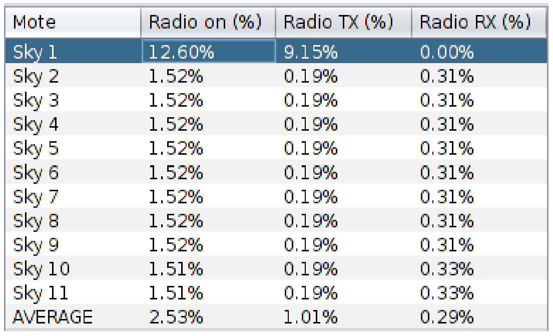
The broadcast test exposes a significant flaw in WPI-MAC: when all the traffic occurs during a single transmission window, the nodes encounter substantial idle listening. Idle listening is the wasteful condition that occurs when a wireless transceiver is fully powered and listening to a radio channel on which no activity is occurring. Figures 5.1-5.3 represent the output of Cooja's PowerTracker tool. This tool provides a per-mote summary of the amount of time that the wireless transceivers were in a particular state. The 'Radio on' statistic expresses the percentage of the total simulation time that the mote spent with its wireless transceiver fully powered; this is inclusive of time spent sending transmitting or receiving data as well as idle listening. Similarly, the 'Radio TX' and 'Radio RX' represent the percentage of total simulation duration spent transmitting or receiving packets, respectively. The 'Radio RX' percentage includes any overhearing that may occur as it is simply a measure of the amount of time the transceiver spent receiving a packet, regardless of the packet's intended recipient. For these tests, 'Mote 1' acted as the base station that was transmitting the broadcast messages; all other motes were receiving.

As illustrated in 5.1 and 5.2, the asynchronous protocols are able to achieve low power-on times for the receivers, but at the cost of a disproportionately high usage on the part of the sender. The results for X-MAC and ContikiMAC are extremely similar except X-MAC requires roughly four times the amount of power to accomplish the same task. The WPI-MAC results do not vary as drastically from sender to receiver.



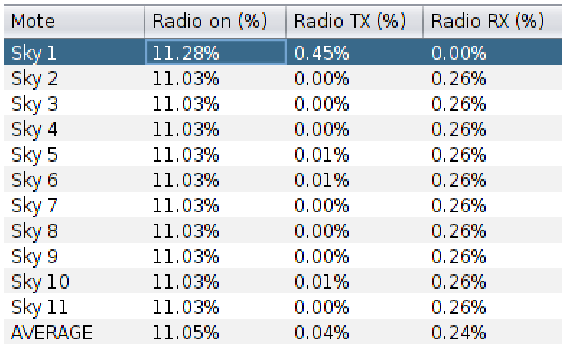
Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	15.18%	8.58%	0.00%
Sky 2	5.39%	0.14%	0.30%
Sky 3	5.39%	0.14%	0.30%
Sky 4	5.39%	0.14%	0.30%
Sky 5	5.39%	0.14%	0.30%
Sky 6	5.39%	0.14%	0.30%
Sky 7	5.39%	0.14%	0.30%
Sky 8	5.39%	0.14%	0.30%
Sky 9	5.39%	0.14%	0.30%
Sky 10	5.20%	0.14%	0.30%
Sky 11	5.20%	0.14%	0.30%
AVERAGE	6.24%	0.90%	0.27%

Figure 5.1: X-MAC Radio Duty Cycling Data – Broadcast Test



Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	12.60%	9.15%	0.00%
Sky 2	1.52%	0.19%	0.31%
Sky 3	1.52%	0.19%	0.31%
Sky 4	1.52%	0.19%	0.31%
Sky 5	1.52%	0.19%	0.31%
Sky 6	1.52%	0.19%	0.31%
Sky 7	1.52%	0.19%	0.31%
Sky 8	1.52%	0.19%	0.31%
Sky 9	1.52%	0.19%	0.31%
Sky 10	1.51%	0.19%	0.33%
Sky 11	1.51%	0.19%	0.33%
AVERAGE	2.53%	1.01%	0.29%

Figure 5.2: ContikiMAC Duty Cycling Data – Broadcast Test



Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	11.28%	0.45%	0.00%
Sky 2	11.03%	0.00%	0.26%
Sky 3	11.03%	0.00%	0.26%
Sky 4	11.03%	0.00%	0.26%
Sky 5	11.03%	0.01%	0.26%
Sky 6	11.03%	0.01%	0.26%
Sky 7	11.03%	0.00%	0.26%
Sky 8	11.03%	0.00%	0.26%
Sky 9	11.03%	0.00%	0.26%
Sky 10	11.03%	0.01%	0.26%
Sky 11	11.03%	0.00%	0.26%
AVERAGE	11.05%	0.04%	0.24%

Figure 5.3: WPI-MAC Duty Cycling Data – Broadcast Test

Figures 5.4-5.6 are screenshots from Cooja's timeline tool which are intended to provide a visual representation of the behavior of the wireless transceiver. Each row is numbered and represents a different mote. The grey lines indicate that the radio was powered on and idly listening to the medium, while blue and green marks respectively indicate the sending and receiving of packets. These screenshots are samples of what the timeline displayed during the transmission of a single broadcast packet.

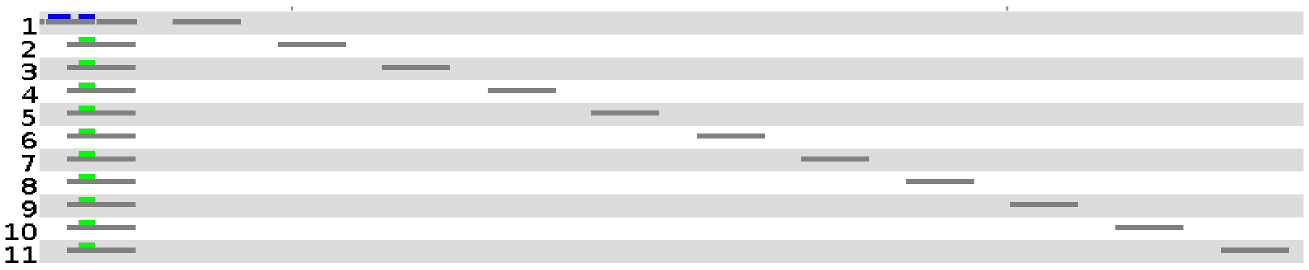


Figure 5.4: WPI-MAC Cooja Timeline – Broadcast Test

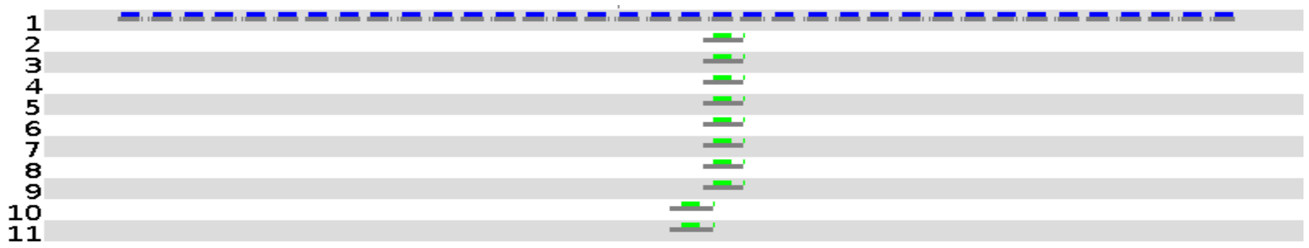


Figure 5.5: X-MAC Cooja Timeline – Broadcast Test

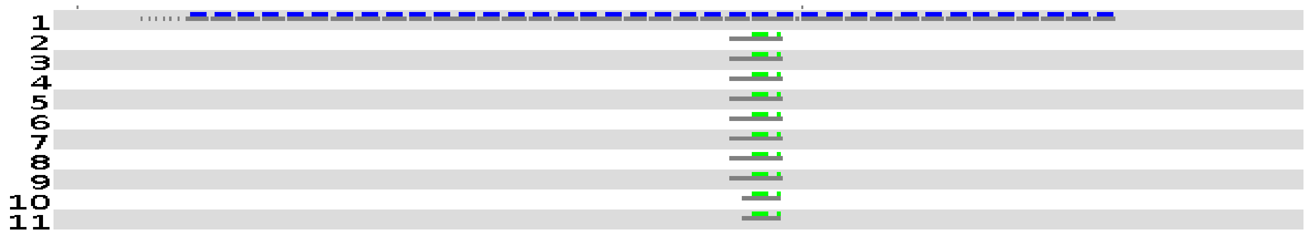


Figure 5.6: ContikiMAC Cooja Timeline – Broadcast Test

In order to more accurately evaluate the performance of these protocols, one must also consider the latency experienced by packets in the system. Table 5.1 shows the average latency for broadcast packets transmitted using each of the protocols. The latencies shown here do not account for delays due to retransmissions. Furthermore, they do not include the exchange of ACKs that may take place after the message has been successfully received. Shown below is simply an average measure between the time when the program submitted its packet to the MAC layer for transmission and the time in which that same packet was received by its intended target.

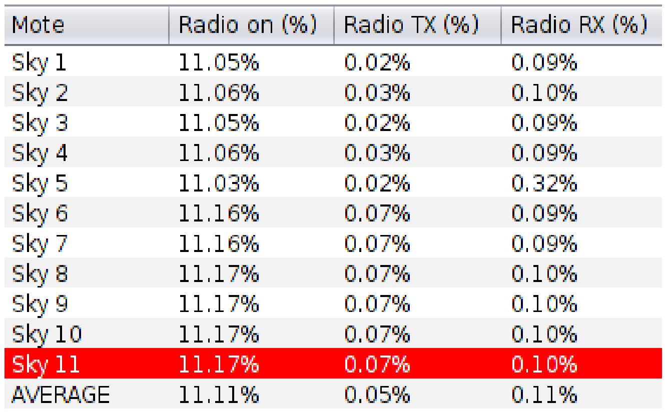
	ContikiMAC	X-MAC	WPI-MAC
Latency (ms)	91	104	111

Table 5.1: Average Latency by Protocol – Broadcast Test

While WPI-MAC did not perform as well as the asynchronous protocols in this test, the figures above help to illustrate how the protocol could be optimized for this scenario. The unutilized unicast transmission slots are the cause of the higher latency experienced by WPI-MAC, but because they are so rigidly defined, they allow WPI-MAC to offer the most consistent latency measurements. Reducing the number of transmission slots in WPI-MAC would help reduce delay, but would also cause the motes to wake up more frequently, thus consuming more energy. This exemplifies the stereotypical tradeoff between energy and latency that exists in the world of MAC protocols

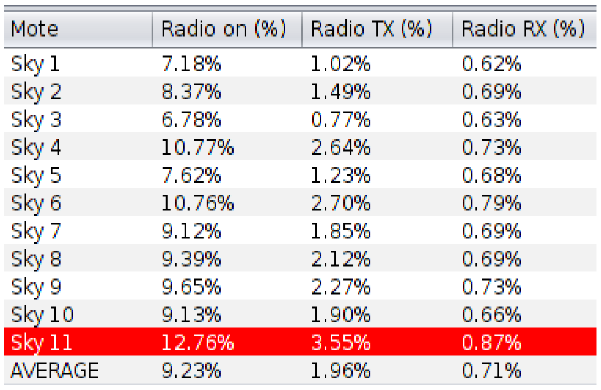
5.2. Local Gossip

The results of the local gossip test were fairly similar when it came to power consumption. Figures 5.7-5.9 below also illustrate the amount of time each protocol kept the transceiver in a particular power state during the test period. The Cooja PowerTracker highlights the mote with the highest level of power consumption in red, though due to internal rounding this may not be obvious.



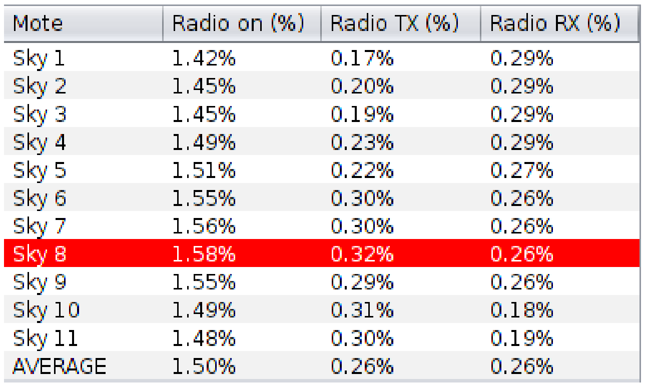
Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	11.05%	0.02%	0.09%
Sky 2	11.06%	0.03%	0.10%
Sky 3	11.05%	0.02%	0.09%
Sky 4	11.06%	0.03%	0.09%
Sky 5	11.03%	0.02%	0.32%
Sky 6	11.16%	0.07%	0.09%
Sky 7	11.16%	0.07%	0.09%
Sky 8	11.17%	0.07%	0.10%
Sky 9	11.17%	0.07%	0.10%
Sky 10	11.17%	0.07%	0.10%
Sky 11	11.17%	0.07%	0.10%
AVERAGE	11.11%	0.05%	0.11%

Figure 5.7: WPI-MAC Duty Cycling Data – Local Gossip Test



Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	7.18%	1.02%	0.62%
Sky 2	8.37%	1.49%	0.69%
Sky 3	6.78%	0.77%	0.63%
Sky 4	10.77%	2.64%	0.73%
Sky 5	7.62%	1.23%	0.68%
Sky 6	10.76%	2.70%	0.79%
Sky 7	9.12%	1.85%	0.69%
Sky 8	9.39%	2.12%	0.69%
Sky 9	9.65%	2.27%	0.73%
Sky 10	9.13%	1.90%	0.66%
Sky 11	12.76%	3.55%	0.87%
AVERAGE	9.23%	1.96%	0.71%

Figure 5.8: X-MAC Duty Cycling Data – Local Gossip Test



Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	1.42%	0.17%	0.29%
Sky 2	1.45%	0.20%	0.29%
Sky 3	1.45%	0.19%	0.29%
Sky 4	1.49%	0.23%	0.29%
Sky 5	1.51%	0.22%	0.27%
Sky 6	1.55%	0.30%	0.26%
Sky 7	1.56%	0.30%	0.26%
Sky 8	1.58%	0.32%	0.26%
Sky 9	1.55%	0.29%	0.26%
Sky 10	1.49%	0.31%	0.18%
Sky 11	1.48%	0.30%	0.19%
AVERAGE	1.50%	0.26%	0.26%

Figure 5.9: ContikiMAC Duty Cycling Data – Local Gossip Test

Most notably, the duty cycling figures demonstrate that ContikiMAC is exceptional at keeping power consumption low for this type of traffic. Meanwhile, X-MAC yielded rather sporadic results and was only marginally more conservative than WPI-MAC on the whole. WPI-MAC again offers more consistent and uniform performance due to the rigid timing of the Crankshaft model. Below, figures 5.10-5.12 depict interesting moments from the testing of each protocol.

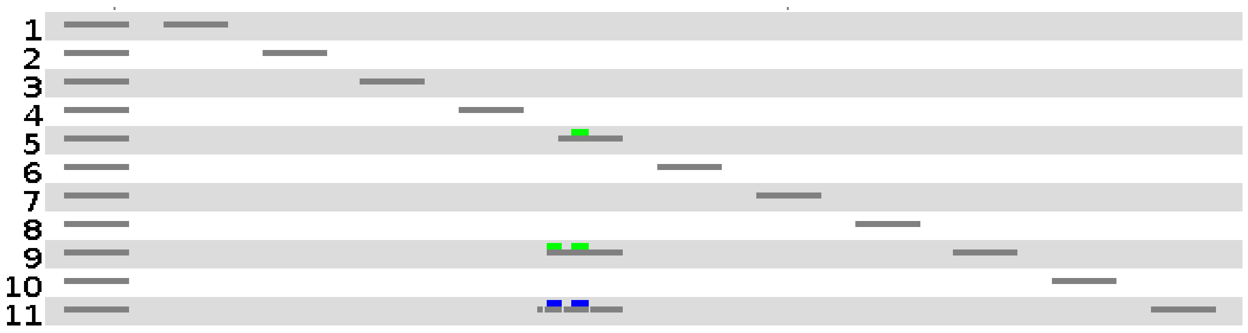


Figure 5.10: WPI-MAC Cooja Timeline – Local Gossip Test

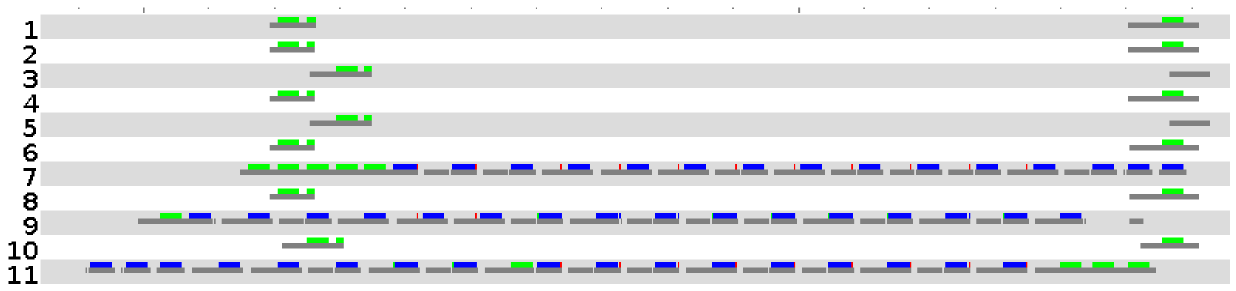


Figure 5.11: X-MAC Cooja Timeline – Local Gossip Test

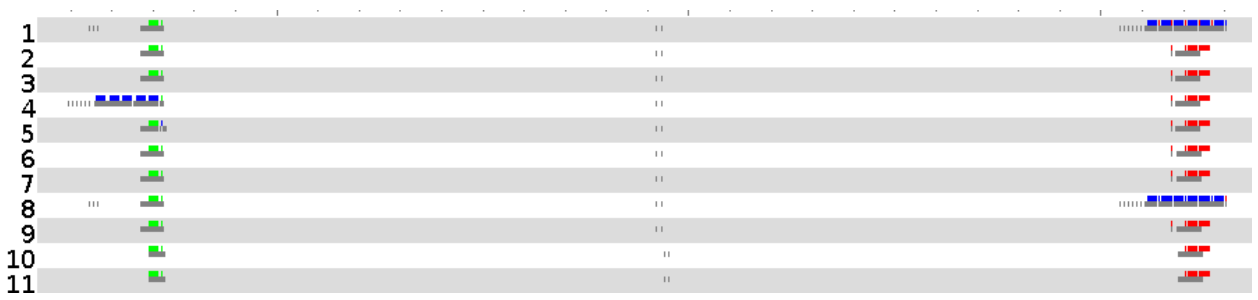


Figure 5.12: ContikiMAC Cooja Timeline – Local Gossip Test

Figure 5.10 depicts nodes 11 and 9 vying for contention over node 5's unicast receiving window. Node 11 starts transmitting sooner than 9 does, causing 9 to back off and not attempt to send. As a result, 9 is awake to overhear the packet bound for 5, but this packet is discarded by 9 as it is not the intended recipient. Figure 5.11 illustrates how X-MAC comes away with such high radio utilization. The radio medium becomes extremely congested by X-MAC's excessive transmission of strobe packets and dependence on ACKs. ContikiMAC's early ACK and fast-sleep optimizations are cause for a much calmer state of affairs in Figure 5.12. However, ContikiMAC does experience some packet collisions (represented by the red blocks) when two nodes attempt to transmit concurrently. Once again, WPI-MAC consumes more energy overall than the other two protocols. But how does it compare in terms of latency? The table below shows the average latency of the three protocols during this test.

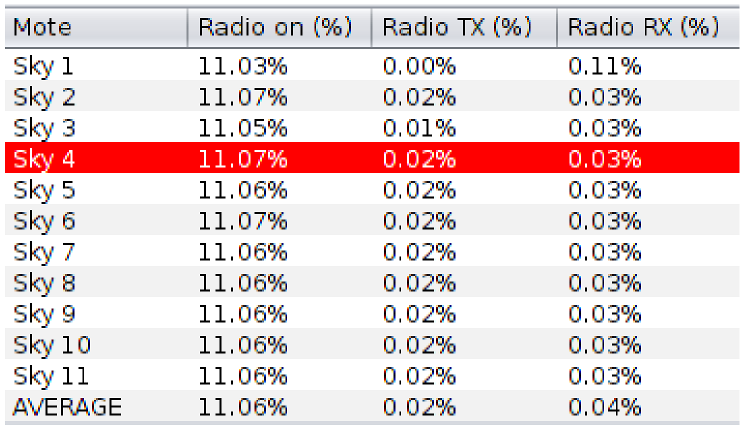
	ContikiMAC	X-MAC	WPI-MAC
Latency (ms)	111	423	134

Table 5.2: Average Latency by Protocol – Local Gossip Test

Unlike the previous test, ContikiMAC is actually the clear winner for this category of traffic. Not only does it use significantly less energy than the other two, it also experiences the lowest latencies of the group. While WPI-MAC did not fare too well in terms of power consumption, it could be a decent alternative to X-MAC if the application was highly latency dependent. In fact, X-MAC does not perform well at all under these conditions. This is likely attributed to collisions that occur when multiple unicast senders are transmitting their entire packets over the radio medium while multiple receivers are simultaneously attempting to announce their availability to receive. Such an occurrence would cause X-MAC to attempt the transmission again later, when it would likely encounters the same problem at least a few times before successfully sending a packet.

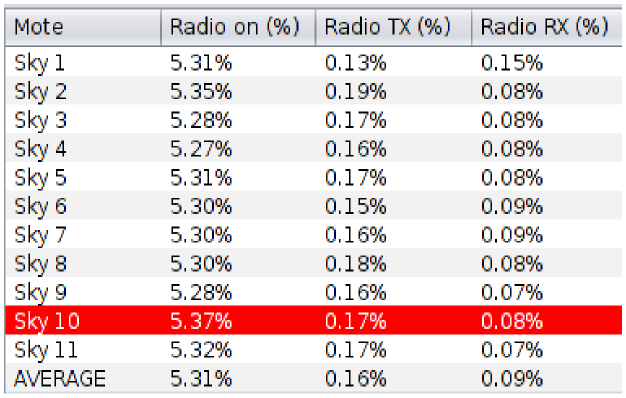
5.3. Convergecast

The results for the convergecast tests bear slight resemblance to the local gossip results. They both share the common trait of attempting to, at times, transmit multiple unicast messages to a single node. Once more, Figures 5.13-5.15 below depict the amount of time that each protocol kept its motes' transceivers in a powered state. The red highlights are simply intended to draw attention to the mote with the highest radio usage.



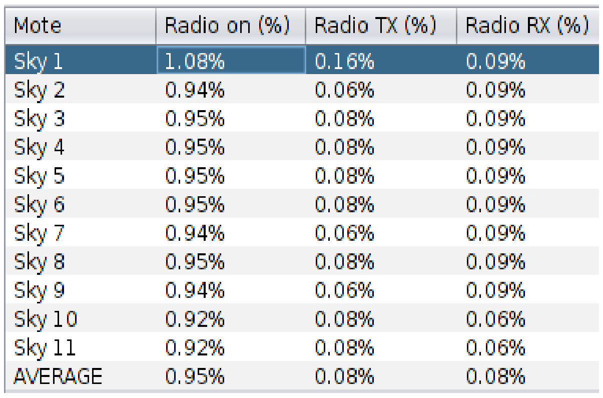
Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	11.03%	0.00%	0.11%
Sky 2	11.07%	0.02%	0.03%
Sky 3	11.05%	0.01%	0.03%
Sky 4	11.07%	0.02%	0.03%
Sky 5	11.06%	0.02%	0.03%
Sky 6	11.07%	0.02%	0.03%
Sky 7	11.06%	0.02%	0.03%
Sky 8	11.06%	0.02%	0.03%
Sky 9	11.06%	0.02%	0.03%
Sky 10	11.06%	0.02%	0.03%
Sky 11	11.06%	0.02%	0.03%
AVERAGE	11.06%	0.02%	0.04%

Figure 5.13: WPI-MAC Duty Cycling Data – Convergecast Test



Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	5.31%	0.13%	0.15%
Sky 2	5.35%	0.19%	0.08%
Sky 3	5.28%	0.17%	0.08%
Sky 4	5.27%	0.16%	0.08%
Sky 5	5.31%	0.17%	0.08%
Sky 6	5.30%	0.15%	0.09%
Sky 7	5.30%	0.16%	0.09%
Sky 8	5.30%	0.18%	0.08%
Sky 9	5.28%	0.16%	0.07%
Sky 10	5.37%	0.17%	0.08%
Sky 11	5.32%	0.17%	0.07%
AVERAGE	5.31%	0.16%	0.09%

Figure 5.14: X-MAC Duty Cycling Data – Convergecast Test



Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	1.08%	0.16%	0.09%
Sky 2	0.94%	0.06%	0.09%
Sky 3	0.95%	0.08%	0.09%
Sky 4	0.95%	0.08%	0.09%
Sky 5	0.95%	0.08%	0.09%
Sky 6	0.95%	0.08%	0.09%
Sky 7	0.94%	0.06%	0.09%
Sky 8	0.95%	0.08%	0.09%
Sky 9	0.94%	0.06%	0.09%
Sky 10	0.92%	0.08%	0.06%
Sky 11	0.92%	0.08%	0.06%
AVERAGE	0.95%	0.08%	0.08%

Figure 5.15: ContikiMAC Duty Cycling Data – Convergecast Test

The nodes were instructed to send a small packet back to the base station once every 10 seconds. As in prior tests, node 1 served as the base station. As per usual, ContikiMAC managed to achieve the lowest energy consumption of the group, while WPI-MAC's utilization was higher but more consistent. However, X-MAC faired significantly better in this trial than it did during the local gossip test. This behavior is to be expected, as only one unicast receiver is attempting to initiate a transmission, unlike the local gossip trial.

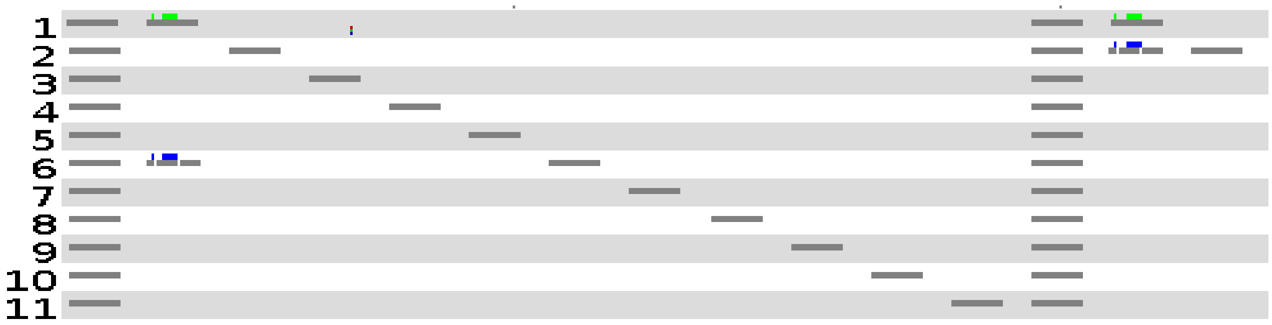


Figure 5.16: WPI-MAC Cooja Timeline – Convergecast Test

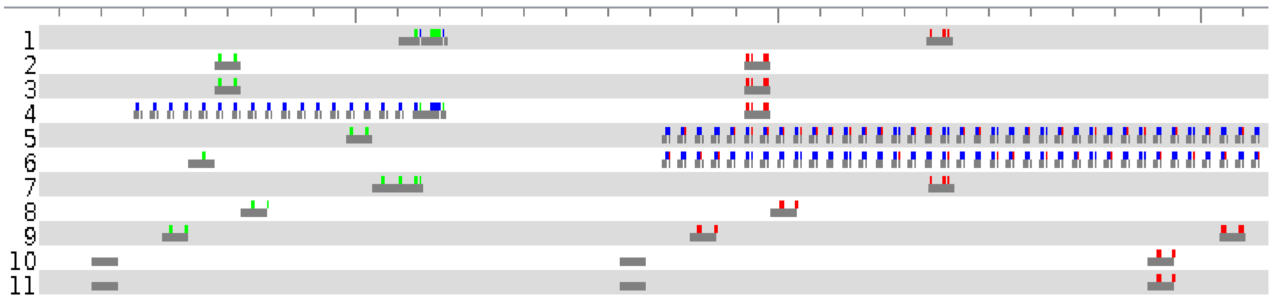


Figure 5.17: X-MAC Cooja Timeline – Convergecast Test

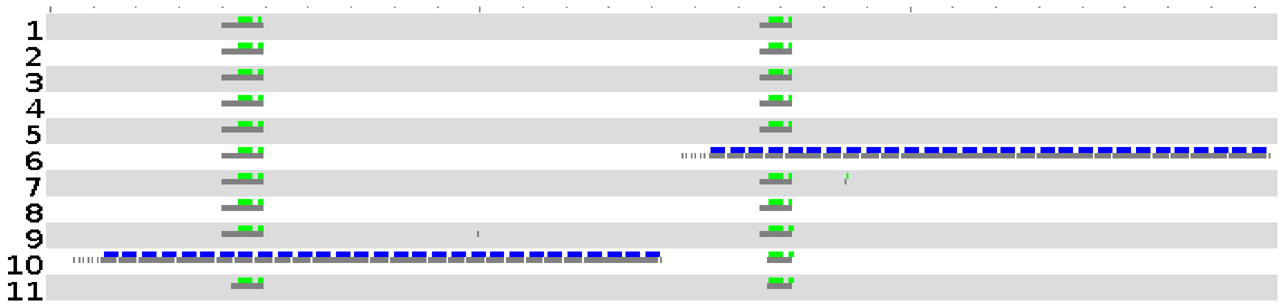


Figure 5.18: ContikiMAC Cooja Timeline – Convergecast Test

Figures 5.16 - 5.18 reflect interesting moments from the trials. The WPI-MAC base station is unfortunately limited to receiving one unicast message per period. Figure 5.16 illustrates this gap and helps to explain the increased latency exhibited in Table 5.3. X-MAC again proves it can clutter the radio medium for a long time when faced with multiple unicast transmissions. The frequent need for retransmissions attributes to the poor delay experienced by this protocol. Lastly, ContikiMAC demonstrates that even by waking its motes up frequently, it can conserve more energy by being able to swiftly power them off again.

	ContikiMAC	X-MAC	WPI-MAC
Latency (ms)	131	274	173

Table 5.3: Average Latency by Protocol – Convergecast Test

Not surprisingly, ContikiMAC is again the clear victor in this test. However, WPI-MAC continues to offer lower latencies than X-MAC but at twice the energy cost. At this point it is clear that X-MAC does not perform very well under these conditions and should be avoided if latency is a factor.

6. Conclusions

Ultimately, there is still plenty of room for growth and additional research in the field of wireless sensor networks. While WPI-MAC does not perform better than ContikiMAC in any scenario tested during this investigation, it does, at times, fare better than X-MAC in regard to latency. However, the tests performed in this experiment do not adequately exercise scenarios in which WPI-MAC would be advantageous. It would be interesting to see how WPI-MAC performed with the inclusion of the proposed changes in the follow chapter.

In the case of wireless sensor networks, choosing a MAC protocol will almost always be dependent on the application of that particular system. Of course, protocols can always be adapted and tuned to perform better in certain situations. Though perhaps the community as a whole can benefit from the existence of several varied and specialized protocols.

7. Future Work

7.1.1. Support Multi-hop Traffic

Although not exercised by the preliminary tests conducted in this investigation, one of the major benefits to choosing ContikiMAC is its ability to relay packets across multiple sensor neighborhoods. In order to be viewed as a more serious contender to ContikiMAC, WPI-MAC would also have to support this same multi-hop functionality. In ContikiMAC, nodes function as sleepy routers, which are capable of waking up to relay a packet to another neighborhood and then quickly return to sleep. Duplicating this behavior in WPI-MAC would be challenging due to the need for multi-clock synchronization.

7.1.2. Fast Sleep Optimization

Presently, the transmission slot length in this implementation of WPI-MAC is too long. Reducing the duration of the slot could help combat excessive idle listening. As soon as a recipient node has successfully received, there is no need for it to leave its transceiver powered. In the current version of the driver, the node remains awake until the next transmission slot begins (which subsequently triggers the transceiver's shutdown method).

Additionally, the current behavior for a unicast or broadcast receiver is to wake up *after* the contention windows. If the receiver(s) woke up at the start of the contention windows, they could immediately go back to sleep if they detect no filler packet being transmitted before the message exchange window. This change would also have no impact on delay, but would help to conserve energy through the drastic minimization of idle listening.

7.1.3. Dynamically Accommodate Varying Population Size

In its current state, WPI-MAC does not support nodes joining or leaving after the initial setup period. This is another area in which ContikiMAC and X-MAC currently outperform it as they are significantly more flexible in this regard. This would be accomplished through the use of control packets that would be used to inform new nodes of the current state of the network.

7.1.4. Selective Omission of Broadcast Slot

It is possible that WPI-MAC could reduce a portion of the energy wasted to over-hearing by allowing for the broadcast slot to be omitted during certain cycles. The protocol would declare ahead of time how frequently there would be a broadcast window. For example, by including a broadcast window only once every ten periods, this would help to keep more transceivers off for longer durations of time. However, in situations that experience heavy broadcast traffic, this could further increase the delay of such traffic types. Although in scenarios with low volumes of broadcast traffic, removing an entire transmission slot could slightly reduce latency.

7.1.5. Additional Testing With Larger Packets

The programs used to test these protocols all used very short packet payloads, none of them larger than 32 bytes. As a result, some of the protocols may behave differently when given larger packets to transmit. It is predicted that WPI-MAC would suffer higher delays than the ContikiMAC, which includes a mechanism for bulk transfer. This highly specific optimization allows a node to send multiple packets in rapid succession, something that currently cannot be achieved with WPI-MAC.

References

[1] Crossbow Technology, Inc., "TelosB Mote Platform Data Sheet," [Online]. Available: http://bullseye.xbow.com:81/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf.

[2] A. Dunkels, "The ContikiMAC Radio Duty Cycling Protocol," *SICS Technical Report,* vol. T2011, no. 13, pp. 1-11, 2011.

[3] A. Dunkels, B. Gronvall and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, 2004.

[4] B. Bates, A. Keating and R. Kinicki, "Energy Analysis of Four Wireless Sensor Network MAC Protocols," *Worcester Polytechnic Institute,* pp. 1-6, 2009.

[5] B. Bates, A. Keating and R. Kinicki, "Energy analysis of four wireless sensor network MAC protocols," in *6th International Symposium on Wireless and Pervasive Computing (ISWPC)*, Hong Kong, 2011.

[6] The Contiki Project, "Contiki: The Open Source Operating System For The Internet Of Things," July 2012. [Online]. Available: http://www.contiki-os.org/. [Accessed 11 December 2012].

[7] A. Dunkels, "Change MAC or Radio Duty Cycling Protocols," 27 November 2011. [Online]. Available: http://www.sics.se/contiki/wiki/index.php/Change_MAC_or_Radio_Duty_Cycling_Protocols. [Accessed 17 September 2012].

[8] S. Zacharias and T. Newe, "Competition at the Wireless Sensor Network MAC Layer: Low Power Probing interfering with X-MAC," *Journal of Physics: Conference Series,* vol. XVI, no. 307, pp. 1-6, 2011.

[9] A. Dunkels, "Radio duty cycling: The Contiki X-MAC," December 2012. [Online]. Available: https://github.com/contiki-os/contiki/wiki/Radio-duty-cycling#wiki-The_Contiki_XMAC. [Accessed 3 January 2013].

[10] The Contiki Project, "Contiki 2.6 Documentation," [Online]. Available: http://contiki.sourceforge.net/docs/2.6/a01787.html.

[11] G. P. Halkes and K. G. Langendoen, "Crankshaft: An Energy-Efficient MAC-Protocol for Dense Wireless Sensor Networks," *Lecture Notes in Computer Science,* vol. 4373, pp. 228-244, 2007.

[12] S. Pratapa, "Improving latency in Crankshaft - An energy-aware MAC protocol for Wireless Sensor Networks," *Worcester Polytechnic Institute,* 2009.