2006-07-19

Characterizing Dynamic Power and Data Rate Policies for WirelessUSB Networks

Jeffrey L. Barlow
Brigham Young University - Provo

CHARACTERIZING DYNAMIC POWER AND DATA RATE POLICIES FOR WIRELESSUSB NETWORKS

by

Jeffrey L. Barlow

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2006

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Jeffrey L. Barlow

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Charles D. Knutson, Chair
Date	James K Archibald
Date	Mark Clement

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Jeffrey L. Barlow in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Charles D. Knutson
Chair, Graduate Committee

Accepted for the Department

Parris K. Egbert
Graduate Coordinator

Accepted for the College

Thomas W. Sederberg, Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

CHARACTERIZING DYNAMIC POWER AND DATA RATE POLICIES FOR WIRELESSUSB NETWORKS

Jeffrey L. Barlow

Department of Computer Science

Master of Science

Wireless communication is increasingly ubiquitous. However, mobility depends intrinsically on battery life. Power can be conserved at the Media Access Control (MAC) layer by intelligently adjusting transmission power level and data rate encoding. WirelessUSB is a low-power, low-latency wireless technology developed by Cypress Semiconductor Corporation for human interface devices such as keyboards and mice. WirelessUSB devices conserve power by employing power-efficient hardware, dynamic power level adjustment and dynamic data rate adjustment.

We characterize the effects on power consumption of dynamically adjusting node power using two dynamic power negotiation techniques as well as two reactive techniques. We also characterize the effects of dynamically adjusting data rate using three rate adjustment techniques. We further characterize the effects of collaboratively adjusting both power and data rate. We validate our techniques through simulation and find that such collaboration yields the greatest energy conservation for a wide variety of conditions and usage models.

ACKNOWLEDGMENTS

I want to thank Cypress for their graceful support of this research; To my parents, my colleagues, my professors, and most of all, to my wonderful wife Becca, who has always believed in me, and whose constant encouragement has furnished this great work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Wireless communication is increasingly ubiquitous. However, mobility depends intrinsically on battery life. Power can be conserved at the physical layer via efficient, adjustable hardware and at the Media Access Control (MAC) layer by intelligently adjusting transmission power levels. Power consumption is also dependent on the rate of communication. If the time to transmit a given message is reduced, energy is conserved because the transmitter is active for a shorter period of time [1].

Most modern wireless technologies specify multiple data rates and power levels. However, few performance analysis studies have been conducted to determine the optimal approach to managing these parameters.

Prior to this research, the precise interplay between power adjustment and rate adjustment in power conservation for a WirelessUSB device was unknown. If a transmission fails (due to interference or node movement) it was unknown what should be adjusted to guarantee success on a subsequent transmission. Should the rate be adjusted, the power increased, or both? These two important factors are interrelated, and both are adjustable in a WirelessUSB device. We demonstrate that optimum mechanisms for adjusting the transmission power and data rate can be combined to produce even greater savings for WirelessUSB devices.

1.2 Thesis Statement

We characterize the power consumption effects of dynamically adjusting node power and data rate within the WirelessUSB transceiver and protocol. This research discovers the benefits and problems of various adjustment techniques within the context of WirelessUSB.

We demonstrate that RSSI sensing and link quality assessment can effectively control power usage; that beaconing, SNR Threshold Adaptation, and link quality assessment can effectively control data rate; and that such algorithms can be combined to further reduce power consumption for WirelessUSB devices.

1.3 Thesis Layout

This thesis comprises four stand-alone papers that build on each other to collectively address the stated goal of this research.

Chapter 2 is a paper entitled ***Characterizing Dynamic Power Optimization Policies for the WirelessUSB Protocol***. The research results demonstrate the effectiveness of employing power negotiation policies for various interface devices and remote controls in reducing power consumption by up to 50%. These results assume no interference nor loss of communication, and are in the context of one single-hop connection.

Chapter 3 is a paper entitled ***Minimizing Power Consumption for Dense WirelessUSB Networks***. The research approach described in this paper relaxes interference constraints and extends our previous work to many-to-one (N:1) single-hop connections. These results demonstrate effectiveness in conserving energy through reactive power policies.

Chapter 4 is a paper entitled ***Characterizing Dynamic Data Rate Policies for WirelessUSB Networks***. The research described in this paper explores dynamic data rate policies in the context of a constant power level. These results demonstrate the effectiveness of several data rate adjustment techniques.

Chapter 5 is a technical report entitled ***Maximizing Battery Life of WirelessUSB Devices Using Dynamic Power and Data Rate Policies*** in which

the optimal policies for power and data rate adjustment are combined in order to further increase performance.

Appendix A provides an overview of the simulation model as well as selections of source code written to facilitate the experiments in this thesis.

Chapter 2

Characterizing Dynamic Power Optimization Policies for the WirelessUSB Protocol

The WirelessUSB[1] *protocol provides low-power wireless connectivity for peripheral devices and sensor applications. Dynamic policies for the WirelessUSB protocol enable devices to conserve power by operating at the lowest power level necessary for communication. Simulation results show that a power savings of as much as 50% can be achieved via dynamic power management policies.*

2.1 Introduction

Wireless communication is increasingly ubiquitous, with mobility dependent on battery life. Efficient power management is essential to the usefulness of wireless devices.

This paper characterizes the benefits of employing power optimization policies, using the WirelessUSB protocol as a case study. WirelessUSB is a low-power wireless technology, employing both power-efficient hardware and dynamic power levels.

The rest of this paper is organized as follows. Section 2.2 discusses other low-power wireless technologies and related research on power management. Section 2.3

[1] *WirelessUSB is a trademark of Cypress Semiconductor Corporation.*

discusses methods of power negotiation. Performance metrics used in the evaluation of these methods are described in section 2.4 and theoretical savings are estimated in section 2.5. Our simulation methodology is discussed in section 2.6, and the results of our tests are presented and analyzed in sections 2.7 and 2.8.

2.2 Related Work

In this section we briefly describe Bluetooth and ZigBee, low-power wireless technologies that are technically similar to WirelessUSB. We then review other proposed power saving mechanisms that can be employed by low-power wireless technologies.

2.2.1 Bluetooth

Bluetooth is a low-cost, low-power, short-range radio technology for peripheral devices. Output power of a Bluetooth device is divided into three classes: The output power of class 1 is 100 mW (20 dBm); Class 2 is 2.5 mW (4 dBm); Class 3 is 1 mW (0 dBm) [2]. Power is controlled by measuring the Received Signal Strength Indication (RSSI) and then employing Link Manager Protocol (LMP) commands such as `LMP_incr_power_req` and `LMP_decr_power_req` to report the need to increase or decrease transmission power.

2.2.2 ZigBee

ZigBee is intended for use in sensor and control networks (such as thermostats and security sensors) that require power-sensitive devices with target battery life measured in years.

ZigBee conserves power by implementing the 802.15.4 standard [3], designed for low-power devices. Devices conserve power by spending most of their operational life in a sleep state, periodically listening to the RF channel to determine whether a message is pending.

Chips designed for ZigBee RF transceivers (such as the EM2420 [4] and CC 2420 [5]) divide output power into eight levels. Similar to Bluetooth and WirelessUSB,

ZigBee devices adjust output power among these eight levels according to the RSSI. However, ZigBee's physical layer also employs receiver energy detection (ED) and link quality indication (LQI) to aid in power level adjustment [3].

2.2.3 Power Optimization Mechanisms

Self-Tuning Power Management (STPM) [6] is a dynamic power algorithm that considers the "time and energy costs of changing power modes" to determine an optimum energy saving solution. This algorithm adapts to the characteristics of the network interface, mobile computer, and associated applications. Each application discloses hints about its intended use of the wireless network, allowing STPM to disable the network interface when it is not in use, thereby ensuring that application delay constraints are satisfied.

Dynamic Voltage Scaling (DVS) is an algorithm that emphasizes graceful energy scalability as the key to wireless node longevity [7]. This approach defines "knobs" that when tuned, allow energy to scale gracefully. Such knobs include the amplifier's transmission power, the convolutional code, and the processor's voltage. DVS describes a basic API, allowing an application to control the knobs, thereby communicating the energy requirements of the application to the power management system.

Varying transmission power at the physical layer has also been shown to yield energy savings. A physical layer study in [8] proposes a mathematical model to express the relationship between distance and transmission power. Based on this model, simulations revealed that by varying transmission power at the physical layer, a 40% energy savings can be achieved.

In [9], a general model is proposed in which power consumption can be estimated based on the distance between two nodes. We use this model in section 2.5 to mathematically estimate power savings.

STPM and DVS provide complex mechanisms for power management that require information from the application level. The power saving strategy employed by WirelessUSB is similar to [8] and [9] in that it analyzes the signal strength to

estimate the distance and adjust transmission power accordingly.

2.3 Power Negotiation

Battery power is lost unnecessarily when wireless devices transmit at a higher signal strength than necessary. Wasteful transmissions are reduced by negotiating transmission power. Optimal power negotiation entails minimizing negotiation overhead while maximizing power savings.

As mentioned previously, WirelessUSB devices transmit at one of eight power levels, 0 being the weakest, and 7 being the strongest. Table 2.1 illustrates the tradeoff between power and range.

Table 2.1: Performance tradeoff between power and range for a WirelessUSB device.

PA Level	Device Power	Max Range
7	99.0 mW	132 m
6	72.6 mW	100 m
5	59.4 mW	47 m
4	52.8 mW	22 m
3	49.5 mW	12 m
2	49.5 mW	9 m
1	49.5 mW	5 m
0	49.5 mW	3 m

To understand the energy saved by employing the appropriate power level and the overhead associated with negotiation, we compare two negotiation techniques while using a non-negotiated approach as a control. In the following sections we discuss these two negotiation techniques in detail.

2.3.1 Incremental Negotiation

The WirelessUSB protocol uses an incremental power negotiation model in which the receiver instructs the transmitter to increment or decrement its transmission power level. When a packet arrives, a received signal strength index (RSSI) is calculated. Based on this value, if the receiver determines that the sender is transmitting with a signal power greater than necessary, it sends back a negotiation packet instructing the sender to decrement its power level by one. Likewise, if the receiver detects that the signal power is below a certain threshold, the receiver sends a negotiation packet requesting that the sender increment its power level by one.

Incremental negotiation may not be optimal. In a worst-case scenario the receiver may instruct the sender to decrement 6 separate times to reduce the transmission power from level 7 to level 1.

2.3.2 Single-Step Negotiation

The Single-Step power transmission policy requires that the receiver determine and report the precise level the transmitter should adopt. This policy attempts to determine the lowest power level suitable for effective communication, and transmits that value to the sender. For example, the receiver may request that the sender jump from a power level of 7 directly to 4, or from 3 directly to 7. This policy makes it possible to jump to a more optimal power level more quickly, with less negotiation overhead.

2.4 Performance Metrics

We employ two metrics to evaluate the various power optimization policies: The total amount of energy used, and the number of negotiation packets. These metrics are discussed in the sections that follow.

2.4.1 Power

The amount of power consumed for a given topology and usage model is the driving factor in estimating battery life. It is therefore important that power (measured in Joules) be minimized for any given workload. The total power conserved can be found by comparing the power consumption of each policy and mobility model.

2.4.2 Number of Negotiation Packets

The energy overhead due to negotiating power levels is determined by the number of messages sent. Each message sent and received requires a certain amount of power, therefore, minimizing the number of messages sent leads to improved power performance.

The total power consumed may increase due to frequent negotiation, however, negotiating infrequently may lead to excessive power use—a reduction in messaging may save on power used to transmit messages, but more packets may be transmitted at higher than necessary power levels before a more optimal power level is established. On the other hand, if sufficient negotiation is performed, more data packets can be transmitted at an optimal power level, even though energy is expended in negotiation.

2.5 Mathematical Model

We estimate the reduction in power consumption due to power policies by exploiting the relationship between the transmission power level and the effective signal strength at a desired range. Assuming the actual transmission power is proportional to the distance, we employ the relationship $u(d) \propto (distance)^n$. The relationship between distance and transmission power of an ad hoc network [9] is given by

$$u(d) = ad^{\alpha} + c \tag{2.1}$$

where a denotes the physical environment, u denotes the transmission power, d denotes distance, α $(2 \leq \alpha \leq 5)$ denotes the distance index and the constant factor c denotes the total energy consumed.

If we consider the influence of free space loss, however, we can modify this formula to make it more accurate. Free space loss is $\propto d^2$, which is significantly smaller when the distance is small than when the distance is large. Therefore, if the distance exceeds a certain threshold, the required power may increase sharply. c is generally assumed to be a constant, although theoretically when the distance reaches a certain threshold, c starts to increase. Based on these considerations, we use an enhanced model to more accurately model the characteristics of WirelessUSB:

$$u(d) = a_1 d^{\alpha} + a_2 d^{\alpha-1} + ... + a_{n-1} d + a_n \tag{2.2}$$

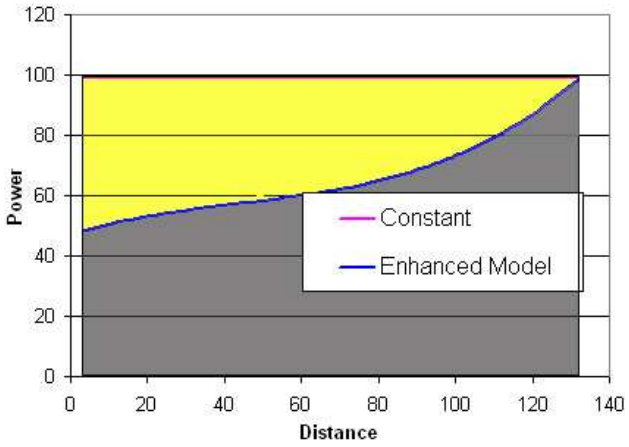


Figure 2.1: Comparison of energy usage between constant output power and the enhanced model.

Based on the data in Table 2.1 and this enhanced model, the efficiency of power adaptation in the WirelessUSB protocol can be estimated. Figure 2.1 shows the comparison of energy usage between a constant output power (no power adjustment) and the enhanced model. The straight line in Figure 2.1 indicates constant, maximum power use and the curved line indicates the power use of the enhanced model. Assuming there is an equal probability of transmitting at all distances, the overall power usage is calculated by computing the area under each line. Power usage for the constant model is 12,771 mJ and the power usage for the enhanced model

is 8,388.34 mJ. Therefore, the enhanced model shows a 34.3% reduction in power consumption.

2.6 Simulation Method

In this section we describe simulations that were performed to validate our mathematical model. For each simulation, power levels 0-7 are used. In practice, a power level lower than 3 would not be used because the power savings are negligible—at power levels below 3 the constant microchip power draw is the only factor. The only reason to switch to a lower power level is for co-location purposes.

2.6.1 ns-2

The protocol and mechanisms tested in this research were written and simulated in ns-2 [10], a reliable simulator with extensive libraries to facilitate wireless simulations. ns-2 is a discrete event simulator with a reliable scheduler which ensures that simulations are accurate. It also has a robust framework that is ideal for writing and testing new protocols.

2.6.2 Usage Models

Target usage models for this study include wireless keyboards, mice and remote controls. All of these devices are unilateral in their communication—they only transmit data, and rarely receive data themselves.

The wireless mouse is the most demanding in terms of the frequency with which data are sent. Constant use of a mouse necessitates the transmission of 10-15 bytes of positional information every 8 ms. Wireless mice are somewhat mobile, although less than a remote control.

The wireless keyboard is used in more intermittent bursts, but requires more total data to be sent. If keyboard data are encrypted, constant keyboard use warrants the transmission of as much as 15-20 bytes every 40 to 60 ms. A wireless keyboard may move during use, although minimally and infrequently.

Wireless remote controls are the least demanding in terms of the amount of data that must be sent, but are the most interesting in terms of mobility. A remote control for a home entertainment system may be used to control the TV, stereo system or DVD player, at distances ranging from less than 1 m to another room at the opposite end of a house. It may also be common to use such a device to control outside speakers from the backyard.

In addition to these traditional usage models, WirelessUSB is also used to operate radio controlled cars. In this model, the rate of transmission is similar to a keyboard, transmitting accelerate, decelerate, and directional information every 40 ms. The demand for range is as great as the remote control, but the speed at which distances are traveled is much greater, since high-power RC cars can accelerate from 0 to 30 mph in under two seconds.

2.6.3 Simulation Methodology

We employed simulations to test the power negotiation techniques in conjunction with these mobility models, further described in this section.

Topology

We consider two nodes communicating in varying proximity. For the purpose of these simulations, the receiving node is assumed to be a fixed node attached to a constant power source. The transmitting node is a mobile, power-sensitive node, and as such, is the only node considered in all power calculations.

Mobility Models

One mobility scenario involves a wireless keyboard or mouse that is repeatedly transported from a desk to a couch and back again every few minutes. We call this mobility model "less-frequent close-range proximity fluctuation." It is simulated by moving the communicating node repeatedly back and forth from 0.1 meters to 10 meters, with pseudo-random time from 1 to 3 minutes between moves.

Another usage scenario involves transporting a remote control from room to room within a house, as well as into the backyard. The remote bearer may also be walking or running. We call this a "random walk" employing distances from 0 to 130 m. We simulate this scenario by setting up a pseudo-random walk from 0 m to 130 m, while varying the movement speed.

A third usage scenario attempts to exploit a potential worst case in which the user is stationary with a wireless keyboard or mouse at a range precisely between two power levels. At this distance, the WirelessUSB device may constantly adjust power levels due to mild intermittent interference. This interference may be caused by a number of factors such as a person walking between the devices, or by a small tilt in device orientation. For the purposes of this paper we deem this mobility model the "borderline" model, which is simulated by moving the communicating node repeatedly back and forth between 5 and 6 meters (or alternatively between 9 and 10 meters).

A fourth scenario involves a radio controlled car. This scenario has the same throughput requirements as a wireless keyboard, but a significantly greater mobility demand. The movement of an RC car produces rapid differences in location and hence in proximity between devices. We call this a "random run," and simulate it by moving the communicating node rapidly through a broad proximity range (0 m to 130 m) in a pseudo-random manner, rarely stopping at any fixed location.

2.6.4 Negotiation Techniques

Three separate negotiation techniques are used in conjunction with each of the four usage models. The maximum power transmission policy always transmits packets at maximum power and is equivalent to not using any dynamic policy at all. Hence, maximum power transmission constitutes a control that can be compared with the Incremental and Single-Step power transmission policies.

2.7 Results

This section presents the NS-2 simulation results for each usage model. In each of these simulations, data were collected for 60 seconds. The details of each

simulated workload follows.

2.7.1 Power

A keyboard was simulated with a workload of 15 bytes every 60 ms. Figure 2.2 shows the results of this workload on the less-frequent close-range mobility model as well as the borderline mobility model.

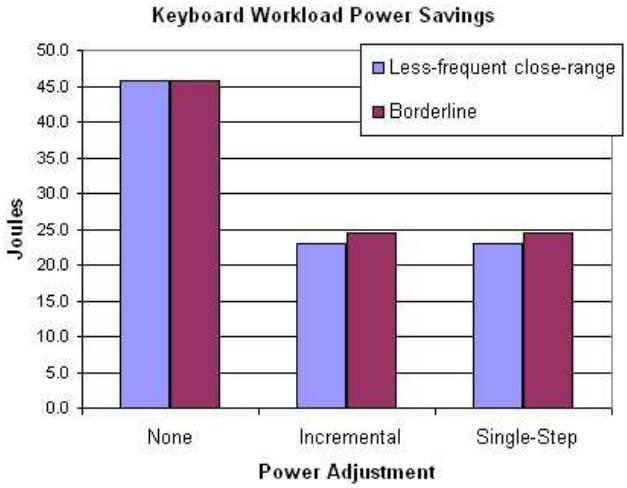


Figure 2.2: Power savings on the keyboard workload with both the less-frequent close-range proximity fluctuation and borderline mobility models.

Both the Incremental and Single-Step power policies for the keyboard workloads produced a power savings of 49.8%. There is little difference between the Incremental and Single-Step policies due to the nature of the usage model. The movement is slow enough and the packet transmission high enough that the Single-Step policy acted almost precisely like the Incremental policy, never jumping to a different power level greater than one step away.

A mouse was simulated with a workload of 10 bytes every 8 ms. The results of this workload on both the less-frequent, close-range and borderline mobility models are similar to the keyboard. A power savings of 50.0% was obtained by either power adaptation policy, with little variation.

Surprisingly, the borderline simulations for both the keyboard and mouse turned out to be fairly benign. The mouse workload produces a 3.1% increase in the number of packets sent and the keyboard workload produces a surprising 23.2% increase over the normal mobility model. However, since negotiation packets are small, the borderline simulation only marginally increases the total power used.

It is also interesting to note that the amount of data transmitted does not significantly effect the node's ability to conserve power. Both the keyboard (sending 15 bytes every 40 ms) and the mouse (sending 10 bytes every 8 ms) achieved nearly identical power savings.

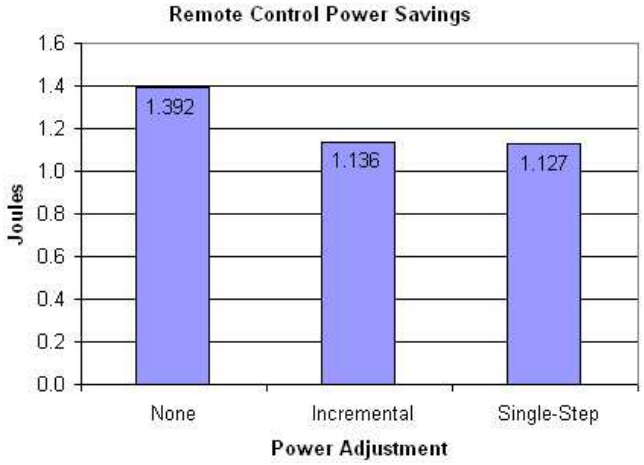


Figure 2.3: Power savings on the remote control workload with both the Incremental and Single-Step power policies.

A remote control was simulated with a workload of 15 bytes every 2 seconds. Figure 2.3 shows the results of such a workload under the Random Walk mobility model. The unique workload of the remote reveals the most significant difference between power policies. The Incremental policy reduces power consumption by 18.4% while the Single-Step policy reduces power consumption by 19.1%. Amortized over a six month battery lifetime, this difference suggests an increase in longevity of 2 days.

A RC Car was simulated with a workload of 10 bytes every 40 ms. Figure 2.4

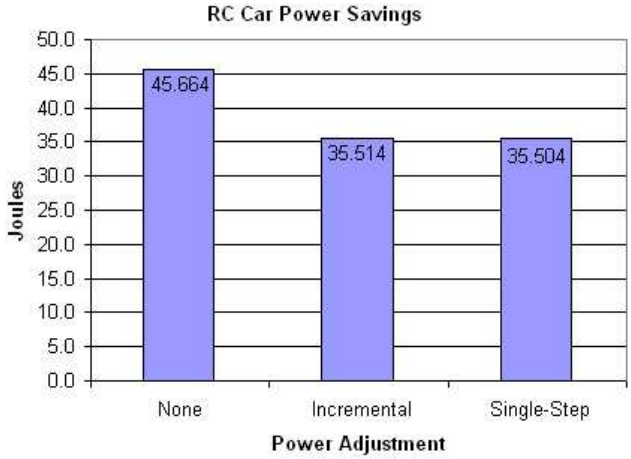


Figure 2.4: Power savings on the RC Car workload with both the Incremental and Single-Step power policies.

graphs the result of this workload under the Random Run mobility model. A 22.2% reduction in power consumption is achieved by both power policies.

Figure 2.5 summarizes the power savings for all models using both the Incremental and Single-Step power policies compared to no policy. This suggests that fewer savings are achieved with lower sending rates or faster node movement—greater savings are achieved with slower node movements and higher sending rates.

2.7.2 Number of Packets

The number of packets increases with the use of any power adjustment policy because both nodes must send negotiation packets to notify the other to change power levels. An example of the number of packets transmitted in a given simulation is 7,375 packets for the borderline mouse simulation using no power adjustment, 7,617 packets for Incremental, and 7,614 packets for the Single-Step power adjustment policy. This reveals a 3.3% increase in the total number of packets transmitted during power negotiation compared with no policy. There is a negligible difference in the number of packets for Incremental vs. Single-Step, because in all usage models, power levels are rarely skipped.

Negotiation packets add to the total power cost, but since these packets are

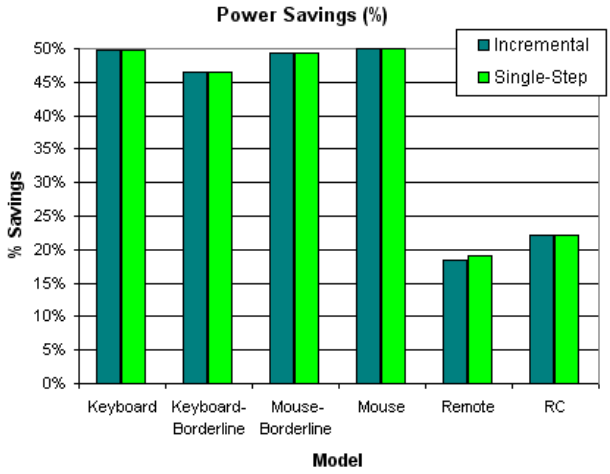


Figure 2.5: Power savings for all models using both Incremental and Single-Step power policies when compared against no policy.

extremely small, in order for their power draw to nullify the power saved, the number of negotiation packets on average would have to be three times the number of data packets. Figure 2.6 summarizes the packet increase for all models due to negotiation. The number of negotiation packets never exceeds 30% for any model.

2.8 Conclusion

Low-power devices such as WirelessUSB can achieve significant power savings by using power adaptation. Simulation demonstrated as much as a 50% decrease in power consumption when either the Incremental or Single-Step policy is employed. This is in line with our mathematical estimate since node positions in the simulation were not evenly distributed. Nodes in our simulation models spent more time at lower power levels, and thus achieved a a 20% greater savings over our mathematical estimate.

The savings difference between Incremental and Single-Step is statistically insignificant. A difference may only be seen in two cases: when the frequency of transmitting is low or the movement is incredibly fast. Only in the remote simulation was a small difference seen because the node had time to travel and jump power levels before the next communication (every 2 seconds).

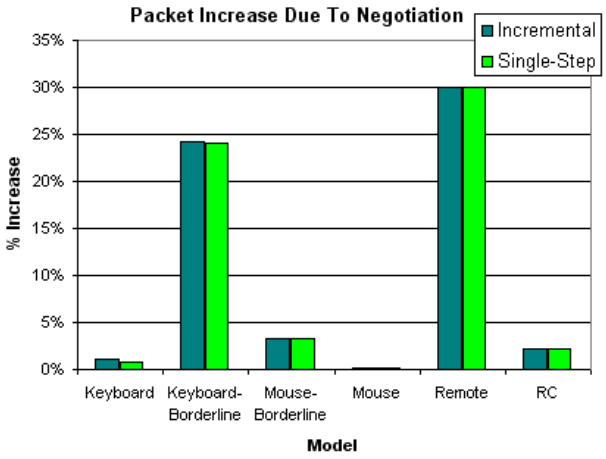


Figure 2.6: An increase in the number of packets for all models using both the Incremental and Single-Step power policies when compared against no policy.

For both power policies, the total number of packets never increased by more than 30% due to negotiation. Even then, sending many negotiation packets did not appear to make a significant difference in the total power consumed.

In order to save power, a low-power wireless device such as WirelessUSB should employ a power management policy. For typical usage scenarios and mobility patterns, it is advised that doing anything more complicated than a simple incremental power policy is unnecessary.

2.9 Future Work

In this study, we assumed that all traffic is UDP-like, transferred in a "best-effort" manner. If a packet is lost while transferring below maximum power, it may be re-transmitted at a higher power level. Further studies will explore the negative impact of such retransmissions on power consumption.

A study on packet loss will allow the exploration of other policies in which nodes cannot communicate. When the transmitting node cannot communicate at its current power level, it may either jump to maximum power and subsequently back off as necessary, or increment its power level by a certain amount until communication resumes. We plan on studying the power costs of such communication reestablishment

policies.

The WirelessUSB protocol specifies that multiple peripherals in an N:1 topology communicate to the base node at the same frequency. This increases collisions, and consequently retransmissions. Future work will include studying the effects of power optimization policies on such complex topologies.

Chapter 3

Minimizing Power Consumption for Dense WirelessUSB Networks

The WirelessUSB[1] *protocol provides low-power wireless connectivity for peripheral devices and sensor applications. Dynamic policies for the WirelessUSB protocol enable devices to conserve power by operating at the lowest power level necessary for communication. This paper presents simulation results that show that even in densely populated networks, power conservation can be achieved while maintaining throughput via reactive power management policies.*

3.1 Introduction

Achieving effective power adaptation in a dynamic, continuously evolving environment is a very difficult problem, exacerbated by high levels of contention in a noisy, lossy medium such as RF. In such an environment, retransmissions increase as more nodes join the same physical space and contend for the same spectrum resource. The goal of this research is to characterize the effects of varying the transmission power of a WirelessUSB device in a crowded environment in which retransmissions are necessary.

In the following sections we describe related work on power adaptation in wireless environments. We then describe two fundamental power adaptation mechanisms

[1] *WirelessUSB is a trademark of Cypress Semiconductor Corporation.*

that are targets for this study. Next, we describe the performance metrics used to rate these mechanisms. We then describe the simulation methodology used during this research. Next we present our results. Finally, we offer conclusions and suggestions for future research.

3.2 Related Work

In this section we discuss related technologies, negotiation techniques, and power control mechanisms.

3.2.1 Related Technologies

Mobile wireless technologies, such as 802.11 [11], ZigBee [3], Bluetooth [2], and WirelessUSB [12], operate under significant power constraints. However, there are fundamental differences between WirelessUSB and these other technologies in their approaches to power conservation. 802.11 devices tend to be less power-constrained while ZigBee devices conserve power by using deep sleep techniques. Bluetooth is perhaps the technology most similar to WirelessUSB as a cable-replacement technology. Woodings and Pandey [13] demonstrate the superiority of the WirelessUSB protocol over Bluetooth in terms of capacity, interference immunity, latency, and power. However, their study does not address node mobility.

3.2.2 Power Negotiation

Our preliminary research results (see Chapter 2) demonstrate the effectiveness of employing power policies for various interface devices and remote controls in reducing power consumption by up to 50% through simple negotiation techniques. These negotiation techniques require that the bridge or master node communicate to the transmitting node the need to increase or decrease its power level based on the current receiver signal strength indicator (RSSI). However, these results assume no interference nor loss of communication, and apply only to the context of a two-node network.

3.2.3 Transmission Power Control Mechanisms

One way to determine the appropriate power level is through location awareness [8, 9, 14]. Since a high transmit power level is needed to transmit at significant distances, a device can determine its position relative to its intended receiver, either through the use of GPS [9] or triangulation [14], and set the appropriate power level.

Due to the low-cost nature of WirelessUSB, adding GPS capability to an enabled device may not be feasible. The problem with triangulation is that it requires either multiple antennae on the radio (increasing the cost per unit), or multiple nodes in a network, and a WirelessUSB device must operate efficiently even when no other devices are connected to the same network.

Another somewhat radical approach to power adjustment involves applying game theory to wireless networks [15, 16, 17, 18]. The basic idea behind a game theory approach is that all nodes are assumed to act rationally and maximize their utility function (a preference level associated with performing a certain action). When applied to power use, this utility is a function of the current cost associated with transmitting at a given power level. A central node may act as a broker and dynamically assign prices to power levels based on network conditions such as node density. In some cases a Nash equilibrium [16, 17] can be achieved when all participating nodes have no incentive to change power levels. Game theory, although potentially applicable to this research, is impractical since not all nodes in an enclosed area participate in the same communal network where incentive to change for the good of the whole exists.

3.3 Power Adaptation Through Link Quality Assessment

This research follows a transmission power adjustment scheme that involves monitoring link quality. In this approach, a device's power level is increased or decreased based on packet success and failure. This method is promising for use in WirelessUSB because no a priori environmental information is required, and each device can act independently and react quickly to environmental changes.

WirelessUSB devices transmit at one of eight power levels, 0 being the weakest, and 7 being the strongest [12]. Table 2.1 illustrates the tradeoff between power and range.

If a node transmits a packet and then fails to receive an acknowledgement the node should act to ensure a successful retransmission. For the purpose of this research, this action could involve either conservatively incrementing the power level, or aggressively jumping to maximum power. In the following sections we describe these two reactive techniques.

3.3.1 Incremental Reestablishment

Incremental reestablishment is a reactive power adaptation mechanism in which the power level is incremented until successful communication is again achieved. This approach may consume significant power and time since numerous power-stepping attempts may be required before successful connection reestablishment.

3.3.2 Aggressive Reestablishment

Aggressive reestablishment is a reactive technique in which nodes retransmit lost packets at maximum power and then execute a power level backoff once communication resumes in order to achieve the most efficient power level necessary to guarantee successful communication.

3.4 Performance Metrics

We employ four metrics to evaluate the various power optimization policies: 1) Energy per node; 2) Throughput per node; 3) Packet delivery ratio; and 4) Packet latency. These metrics are discussed in the following sections.

3.4.1 Energy Per Node

The amount of power used for a given topology and usage model is the main factor in determining battery life. It is therefore important that power consumption be minimized for any given workload to extend battery life. Since we will be dealing with

networks of more than one transmitting node, we use the average energy consumed per participating node, which is more accurate and effective than the total power of a given simulation.

In our simulation, we assume that all slave nodes are power-constrained devices, but do not make the same assumption for master nodes. As a consequence, power consumption of master nodes is not included in this metric.

3.4.2 Throughput Per Node

Throughput is defined as the average rate at which data flows through a system. We model effective throughput (or "goodput") from an application-level perspective. One of our goals is to minimize the potential negative impact of power policies on throughput.

3.4.3 Packet Delivery Ratio

The ratio of lost or retransmitted packets to successful packets aids in determining the reliability and robustness of the protocol. It is anticipated that this ratio may prove significant when packets are lost due to improper power settings.

3.4.4 Packet Latency

Packet latency is measured as user-perceived latency — if a packet takes 1 ms to transmit, and has to be sent twice in order to be received successfully, then the user-perceived latency would be 2 ms or more, depending on the time taken to determine the need to retransmit. Measuring latency in this manner allows us to better gauge the suitability of the proposed policies and the degree to which application-level constraints can still be satisfied.

3.5 Simulation Method

In this section we describe the methods used in developing and executing the simulations for this study.

3.5.1 ns-2

The protocol and mechanisms tested in this research were written and simulated in ns-2 [10], a reliable simulator with extensive libraries to facilitate wireless simulations. ns-2 is a discrete event simulator with a reliable scheduler that ensures simulation accuracy. It also has a robust framework that is well-suited for writing and testing new protocols.

For these studies, we modified the Berkeley Network Simulator to more accurately sample individual node power. We adjusted the physical layer so that variation in the output power affects the transmission range. Also, the carrier sense signal strength thresholds were modified and verified to accurately reflect the maximum theoretical ranges of WirelessUSB at each power level (see Table 2.1). The WirelessUSB routing layer was also modified to handle intelligent power adaptation resulting from transmission failures.

3.5.2 Simulation Methodology

We employed simulations to test the power adaptation techniques in conjunction with these mobility models. In each simulation, data were collected for 5 minutes. Results of each simulation were averaged over 30 iterations of random topologies and random node traffic start times.

Two separate negotiation techniques were used in conjunction with each of the three usage models. The maximum power transmission policy (also referred to as the constant power policy) always transmits packets at maximum power and is equivalent to not using any dynamic policy at all. Hence, maximum power transmission constitutes a control to which the Incremental and Aggressive power transmission policies can be compared.

Topology

We consider various sizes of star topologies composed of many slave nodes communicating in varying proximity to a master node. For the purpose of these simulations, the master receiving node is assumed to be a fixed node attached to a

constant power source. The transmitting nodes are mobile, power-constrained nodes, and as such, are the only nodes considered in all power calculations.

The simulation environment is a 180 m $\times$ 180 m square room. The fixed, non-power-constrained master node is placed in the center of the room, and all other slave nodes are given a random starting location. This guarantees that all slave nodes remain within the maximum power communication distance from the master at all times ($d_{max} = \sqrt{(90^2 + 90^2)} = 127.28m < 132m$).

Results were obtained by varying the number of slave nodes from 1 to 30. The random movement of all slave nodes is described as follows. All slave nodes relocate at random times to random locations. The velocity of each node varies from 1 to 8 meters per second. Once nodes reach their destination, they remain stationary for a random amount of time (between 1 to 5 seconds) before proceeding to move to another random location at a random velocity.

Once we generate the random topologies, we store them and reuse them in all subsequent experiments. This is vital to ensure that statistical differences in power policies are not the result of differing topologies. The same random topologies are used for each experiment while varying the power policy.

Workloads

We consider three relevant workloads based on typical usage models. The workloads for each simulation are designed to accurately model normal utilization of various devices such as keyboards, remote controls, and PDAs.

The first workload under study is referred to as the *HID workload*, characterizing a human interface device, such as a mouse or keyboard. For the purpose of this characteristic workload, a mouse is simulated using a workload of 4 bytes every 60 ms.

The second workload under consideration is referred to as the *remote workload*, which characterizes a TV/stereo remote control. This is simulated using a workload of 4 bytes every 5 seconds. Such a workload is characteristic of a typical channel-surfer, who presses the channel up/down button on the remote approximately every

five seconds.

The third workload under consideration is the *file transfer workload*, characterized by the transfer of a file from one computing device to another. Many usage scenarios apply this type of workload, such as a firmware update, a PDA synchronization, or an image transfer from a digital camera. A file transfer maximizes the available bandwidth of a WirelessUSB device, which, depending on the encoding employed, is about 250 Kbps. For the purpose of this research, a file transfer is simulated with a workload of 32 bytes every 10 ms.

3.6 Results

This section presents the ns-2 simulation results for each usage model. First, we describe the results of several preliminary studies performed to determine proper settings of key parameters. Next, we present the results of each simulated workload. Finally, we diagram the results of a static velocity study as described below.

3.6.1 Upgrade Backoff and Tolerance Studies

We first determine the values for two key parameters in any reactive power policy: the manner in which upgrade attempts are performed, and the tolerance to packet failures. Manipulation of the upgrade mechanism as well as the tolerance threshold may lead to subtle improvements in the way these power policies operate, and will thus be described and studied below.

A successful decrease in the transmission power level is considered an "upgrade" as it allows the communicating node to further conserve energy. For the purpose of this research, all power upgrades are considered a single decrement of the transmitting node's power level.

There are several ways to determine the rate of upgrade attempts. In a time-based upgrade policy, a certain amount of time must transpire before a node attempts another upgrade. We chose to use a packet-success approach where a node attempts to upgrade after a certain number of successful packets at its current power level.

In practice, tracking packet success on an inexpensive embedded system is easy to implement whereas it is difficult to maintain an accurate timer.

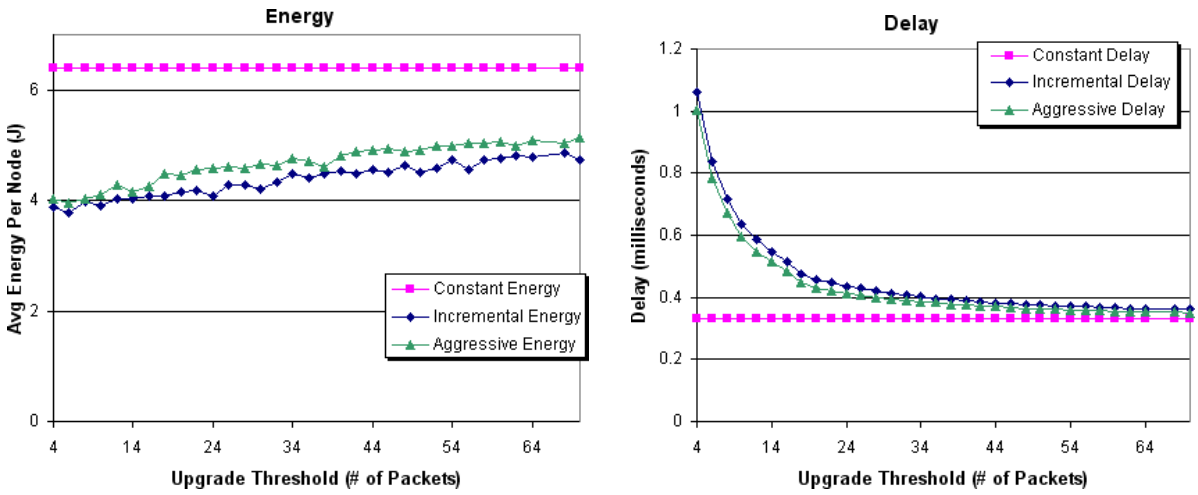
There are also various forms of adapting such an upgrade strategy. The simplest form involves no adaptation and uses a fixed upgrade threshold that is a statically-assigned value based on prior knowledge or experience. We also explore two forms of *upgrade backoff* in this research in which the upgrade attempt threshold dynamically backs off in response to upgrade failures in either a linear or exponential fashion. The purpose of any backoff strategy is to minimize energy waste due to too-frequent upgrade attempts. If a more intelligent node can determine that upgrade attempts are constantly failing, then that node should not attempt to upgrade as frequently.

Figure 3.1 shows the results of a fixed upgrade threshold study in which network size is held constant at 4 nodes and the threshold value is varied. Results show that little is gained by increasing the static upgrade threshold except for a decrease in delay. The delay is diminished with larger thresholds because of a decrease in the need to retransmit packets due to failed upgrade attempts.

Notice also that the average node energy use increases as the threshold increases. This is most likely due to the fact that a node's ability to respond to more favorable conditions diminishes. As an example, if a node must wait for 30 successful packets instead of 10 before upgrading, then that node has potentially operated under less-than-optimal conditions for a greater number of packets.

Figure 3.2 shows the results of varying the backoff mechanism. Fixed, linear, and exponential backoff mechanisms were compared by rating each one's ability to conserve energy as compared to always transmitting at maximum power. Results demonstrate a meager 0.2% to a maximum 4% increase in energy savings by moving from a fixed upgrade mechanism to an exponential mechanism.

Although in this situation energy savings are minuscule, the exponential backoff mechanism was found to reduce the delay by 16.04% (see Figure 3.3). Figure 3.3 shows the results obtained when using the Incremental power policy, although similar results are obtained from the Aggressive power policy.



(a) Energy

(b) Delay

Figure 3.1: Fixed Upgrade Threshold Study (a) Energy and (b) Delay Results. This study reveals that little is gained by increasing the static upgrade threshold except a decrease in delay.

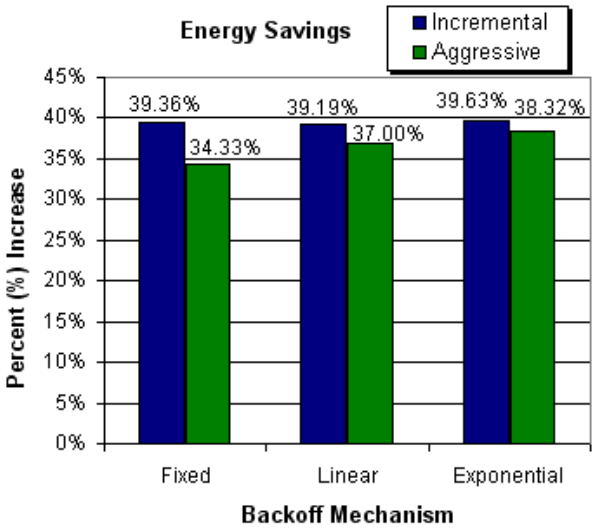


Figure 3.2: Backoff Mechanism Study Results: Meager 0.2 - 4 percent increase in energy savings when using Linear or Exponential compared to no policy.

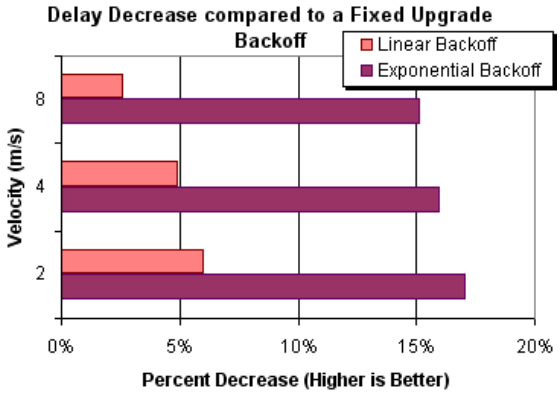


Figure 3.3: Incremental Power Policy Delay Results for the Backoff Mechanism Study: Less node movement improves delay performance when using a linear or exponential upgrade backoff technique.

Tolerance refers to a node's allowance of failed packets before reacting. In the context of this research, a tolerance threshold refers to the number of failed transmission attempts before a node reacts by increasing the power level as dictated by the employed policy.

The tolerance threshold under which a WirelessUSB node should operate was unknown prior to this study. Therefore, a tolerance threshold study was performed on three network densities (low, medium, and high) using the HID workload. This study reveals that energy consumption is not improved significantly by increasing the tolerance level beyond 0 (no tolerance whatsoever or immediate reaction) for low density networks. Any tolerance value above 0 increases delay dramatically.

However, as node density increases, tolerance becomes a factor, and significant performance gains are achieved by increasing node tolerance from 0 to 3 with minimal repercussion. This is due to the fact that more failures arise due to congestion than to movement. In this case, it is more beneficial to attempt a retransmission at the same power level in case the dropped packet was due to contention. If this retransmission is successful without an increase in power level, then the node has conserved power. Figure 3.4 summarizes the tolerance study results for the Incremental Power Policy under the HID workload (other policies and workloads exhibit similar behavior). A

tolerance of 3 in a dense network for example, provides an average energy savings per node of 17.12% with a 45% increase in delay. Above a tolerance threshold of 5, packet delivery and throughput are reduced considerably, and hence such thresholds are never recommended.

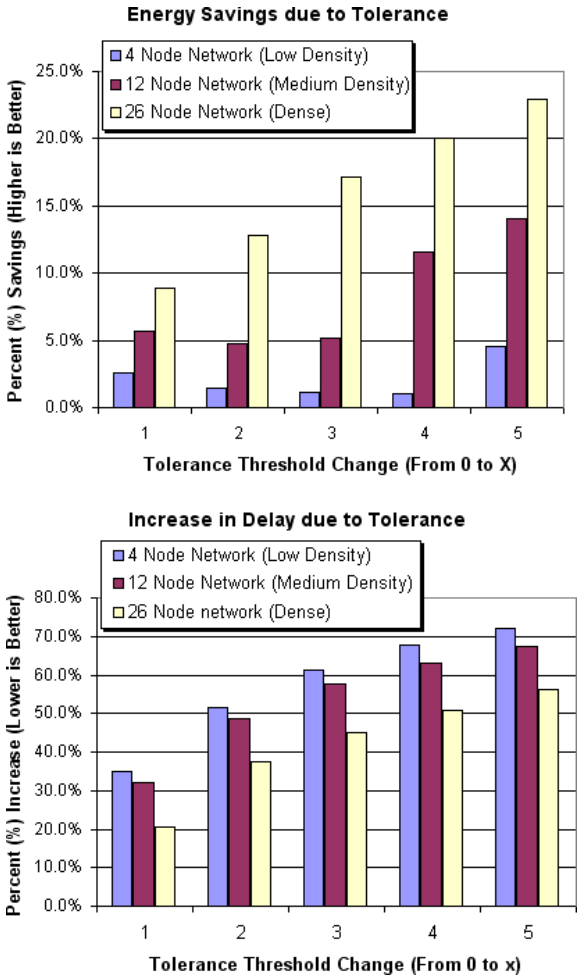
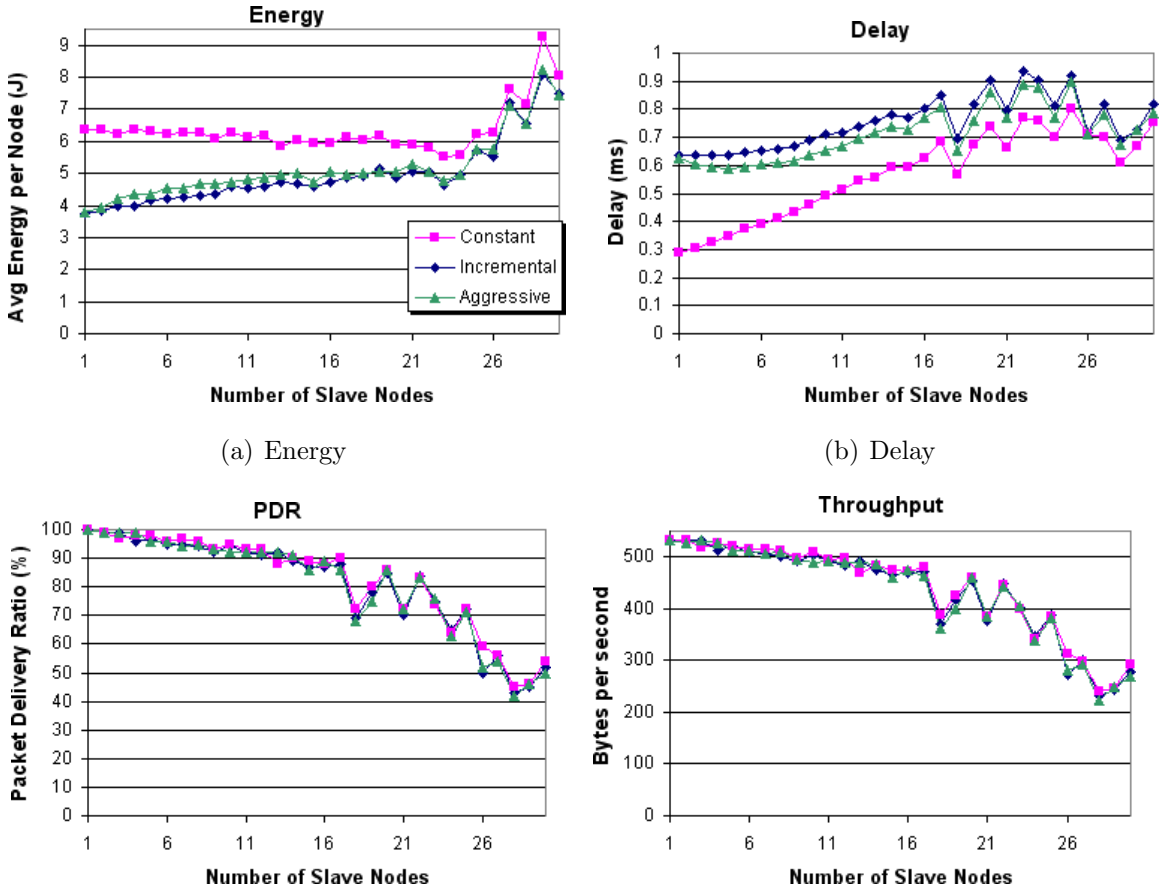


Figure 3.4: Tolerance Study Results: Power savings increase with tolerance for denser networks.

In light of our findings regarding upgrades and tolerance, all future studies were performed using a fixed upgrade threshold of 10 and a tolerance of 0, unless otherwise stated.

3.6.2 HID Workload



(a) Energy

(b) Delay

(c) Packet Delivery Ratio

(d) Throughput

Figure 3.5: HID Workload (a) Energy, (b) Delay (c) Packet Delivery Ratio and (d) Throughput Results.

The results of the HID workload (see Figure 3.5) show an initial energy savings of 31% for the Incremental power policy versus no policy, and 30% for the Aggressive power policy. These savings over no policy decrease gradually as the number of nodes increases, until around 25 slave nodes when the delivery ratio plummets to an unacceptable 50% (half of the transmitted packets do not succeed). At this point there is so much contention that there is little difference between power policies because all

nodes are forced to operate at maximum power to communicate. Thus we see that for the HID workload, the network becomes saturated around 25 nodes.

The Incremental power policy has the worst delay (an increase over constant by 63% on average), followed by the Aggressive power policy (an increase of 52% on average). The constant power policy has the lowest latency.

It is interesting to note the increase in delay when any power policy is employed. This is evident in all workloads and represents a fundamental tradeoff between energy consumption and packet delay. If delay were the only concern, then we would do nothing more than always transmit at the highest power level possible. At that point the only delay in transmission would be due to traffic collisions. However, since we are interested in minimizing energy consumption, we are constantly adjusting the power level, which introduces delay. This is caused by an increase in retransmissions due to incorrect or insufficient power settings as a result of mobility, interference, or upgrade attempts.

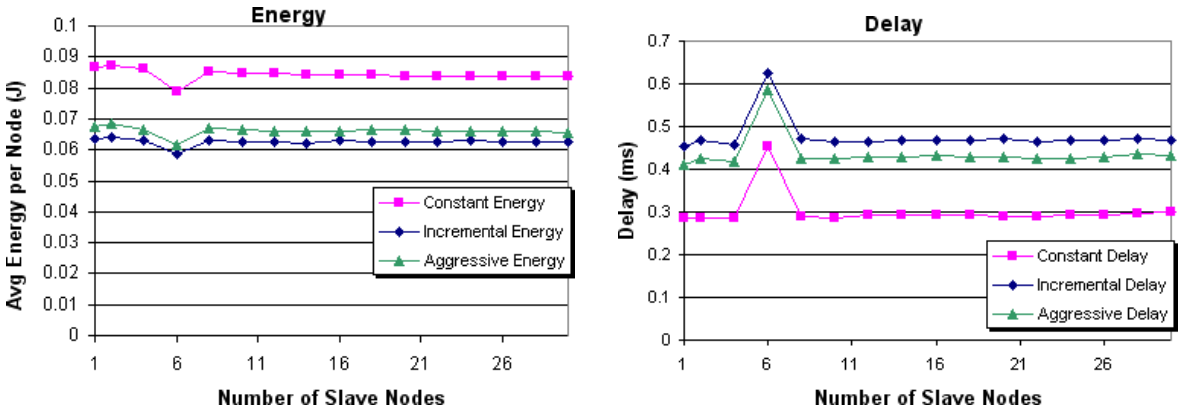
For this workload, all power policies have no significant effect on the packet delivery ratio nor the throughput. All packets that can be delivered when using no power policy are still deliverable when employing a power policy.

3.6.3 Remote Workload

Results of the remote workload (see Figure 3.6) also show the characteristic tradeoff between energy and delay described previously. Average energy savings for the Incremental power policy is 25.75%. Average energy savings for the Aggressive power policy is 21.49%.

Remote workload delays remained consistent throughout all tests. The average increase in delay when using the Incremental power policy is 58.96%, while the average increase in delay for the Aggressive power policy is 45.44%.

This workload obviously does not saturate the network at around 25 nodes like the HID workload, and can handle much greater node densities. Packet delivery ratio remains at around 100% and throughput values stay consistent throughout all tests and are therefore not shown. Due to constraints in ns-2, we were unable to



(a) Energy

(b) Delay

Figure 3.6: Remote Workload (a) Energy and (b) Delay Results.

saturate the network under this workload; however, it seems reasonable to assume it is irrelevant because network densities above 30 slave nodes are unrealistic in practice.

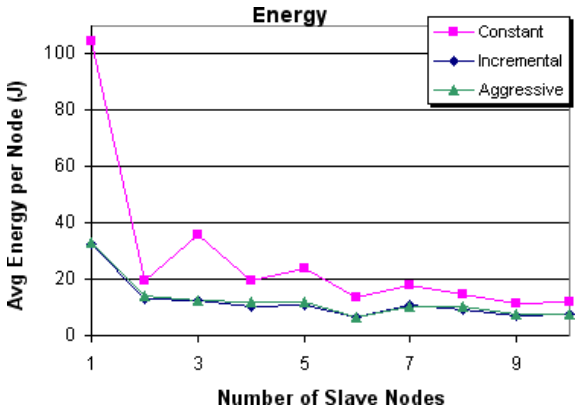
3.6.4 File Transfer Workload

The file transfer workload (see Figure 3.7) saturates the network at 2 slave nodes. Packet delivery ratio and throughput levels plummet when the number of slave nodes is greater than 1. The average packet delay shows no significant difference between all policies even though they show a linear increase with the number of slave nodes.

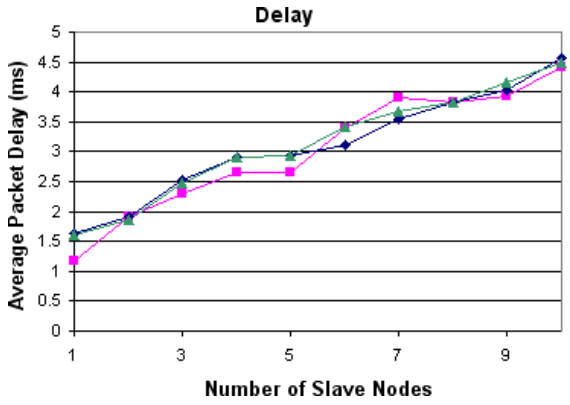
The file transfer workload shows an average energy savings of 46.93% for the Incremental power policy, and 40.16% for the Aggressive power policy. This workload yields the greatest power savings because of the larger packet sizes which increase transmission duration.

3.6.5 Velocity Study

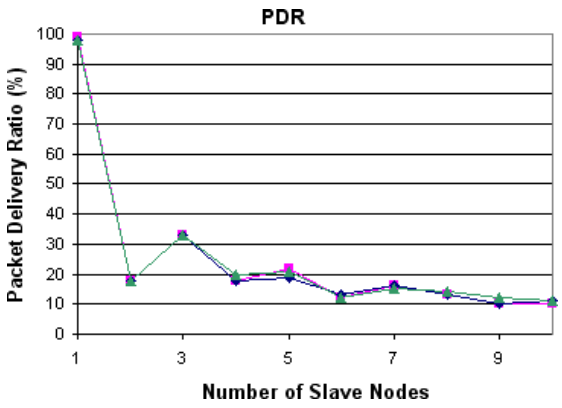
In an effort to pinpoint the precise condition in which the Aggressive power policy may be more cost effective in terms of power, a velocity study was also conducted. The only situation in which the Aggressive power policy may be more cost



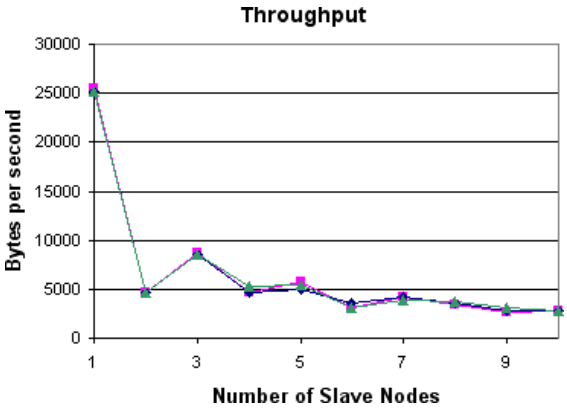
(a) Energy



(b) Delay



(c) Packet Delivery Ratio



(d) Throughput

Figure 3.7: File Transfer Workload (a) Energy, (b) Delay (c) Packet Delivery Ratio and (d) Throughput Results.

effective is when a node travels at such a velocity between packet transmissions that an increment of more than one would be necessary for successful transmission of the next packet. We define *Mobility per packet potential* as the velocity (in m/s) over the packet sending rate (p/s), which reduces to meters per packet (m/p).

This study is performed with a single transmitting node under a workload of 1 packet per second. Node velocity is varied through 10 m/s yielding a mobility per packet potential ranging from 1 to 10 m/p. Node movement consists of a node starting 1 m from the master, and moving to a maximum transmission distance of 130 m and then back again twice. Because an increase in velocity decreases the amount of time necessary to travel a fixed distance, the simulation time is also varied. Since simulation times vary, savings at each velocity are compared.

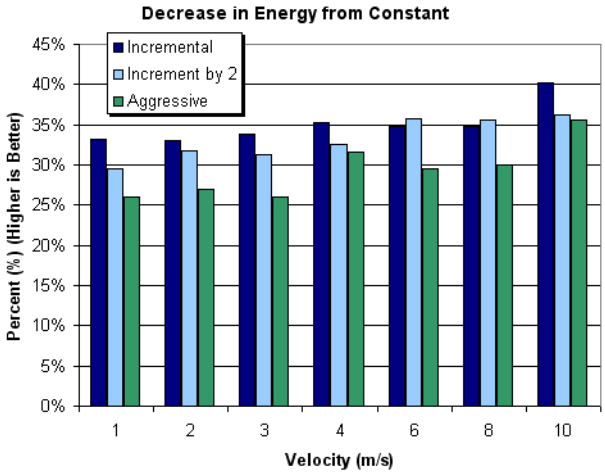


Figure 3.8: Velocity Study Results.

The results of the velocity study are shown in Figure 3.8. In no case does the Aggressive power policy conserve the greatest amount of energy because the effect of jumping to maximum power during periods of low power needs (close range) negates gains at other times due to the high cost of transmitting at maximum power. However, the Increment by 2 power policy (included for completeness, as a slight variant of the Increment power policy) does conserve more energy when node velocity is between 6

and 8 meters per second.

3.7 Conclusion

At this point, the reasons for implementing a power policy that allows a transmitting node to adjust its power level are clear. Nodes implementing a simple reactive power policy that requires no negotiation overhead can save upwards of 60% for small network sizes with no significant degradation in packet delivery ratio or throughput. However, packet latency does increase with power adaptation.

A definite tradeoff exists between the energy conserved and the latency incurred when employing any kind of power adaptation policy. Both the Incremental and Aggressive power policies have been shown to increase average packet latencies. However, this negative effect can be reduced by employing an exponential upgrade backoff technique.

The Aggressive reestablishment power policy in general delivers a nice balance between the Incremental and constant policies. If power is the only concern, however, there is no need to use any other policy other than Incremental. If delay is a concern, and higher velocities are expected, then an increment by 2 power policy may be beneficial.

Tolerance of packet failure becomes increasingly important as the number of nodes in a network increases. Our results show that until a network is saturated, no loss in power is incurred by having zero tolerance. However, when a network is saturated (for example, around 25 slave nodes for the HID workload) then a tolerance increase actually improves energy consumption, because packet failure increases due to contention rather than node movement. Therefore, in these scenarios, nodes should not be too quick to increase their power level.

Backoff mechanisms for reactive power policies show little, if any, improvement over statically assigning the upgrade threshold. An upgrade threshold should be chosen that balances both the energy and the delay needs of the transmitting node.

3.8 Future Work

All simulations performed in this research were performed at a constant data rate. Future work includes adjusting the data rate, which should have a similar affect on communication range as varying the transmission power. Likewise, adjusting the data rate also affects transmission duration, which directly affects power consumption. Transmission duration may also affect contention in the spectrum in a multi-node environment.

We believe that an investigation of the combinatory synergy of transmission power and data rate adjustment will yield significant energy conservation for a wide variety of conditions and usage models.

Chapter 4

Characterizing Dynamic Data Rate Policies for WirelessUSB Networks

The WirelessUSB[1] *protocol provides low-power wireless connectivity for peripheral devices and sensor applications. Dynamic data rate policies for the WirelessUSB protocol enable devices to conserve power while operating at the highest data rate possible. Our simulation results show that dynamic data rate policies increase energy savings significantly while also minimizing delay and maintaining throughput as network densities increase.*

4.1 Introduction

Power-constrained mobile devices must conserve energy in order to maximize their usefulness. Power conservation strategies for mobile devices have historically focused on sleep state management and transmission power control. In this paper, we approach power conservation for WirelessUSB [12] nodes from the perspective of data rate management.

Network nodes generally adjust data rates with the primary goal of increasing throughput. However, lowering the data rate may improve the robustness of a transmission, as data encodings provide varying tolerance to bit errors. Lowering the data rate may also extend the effective range of a wireless signal.

[1] *WirelessUSB is a trademark of Cypress Semiconductor Corporation.*

Dynamically adjusting data rate may be utilized to conserve power. Power consumption is dependent on the rate of communication. If one reduces the time to transmit a given message, energy is conserved because the transmitter remains active for a shorter period of time [1]. Changes in transmission durations also affect collision frequency in a multi-node environment.

In this paper we demonstrate the use of various data rate adaption policies in minimizing energy consumption and optimizing latency and throughput in small to medium sized homogeneous wireless networks. We use Cypress Semiconductor Corporation's WirelessUSB radio and protocol as a case study for this research.

The primary usage model for WirelessUSB involves the connection of multiple devices to a single master device, forming a many-to-one (star) network, such as a keyboard and a mouse connected to a PC. WirelessUSB devices encode packets in such a way that the communicating data rate can be altered on a per-packet basis. This provides an excellent venue for studying the effects of dynamically adjusting a radio's data rate. Table 4.1 lists all of the possible data rates at which WirelessUSB packets may be sent.

Table 4.1: WirelessUSB Data Rates.

Name / Mode	Data Rate	Range	Description
64-chip DDR	31.25 Kbps	130 m	A pair of 64-chip PN codes using Dual Data Rate Mode. Each 64-chip symbol represents two data bits.
32-chip DDR	62.5 Kbps	68 m	A pair of 32-chip PN codes using Dual Data Rate Mode. Each 32-chip symbol represents two data bits.
64-chip 8DR	125 Kbps	22 m	A pair of 64-chip PN codes using 8DR Mode. Each 64-chip symbol represents eight data bits.
32-chip 8DR	250 Kbps	7 m	A pair of 32-chip PN codes using 8DR Mode. Each 32-chip symbol represents eight data bits.
RAW	1 Mbps	1.5 m	Raw data. Each chip represents one data bit.

Prior research suggests that effective data rate adjustment techniques include beaconing, channel quality assessment, threshold adaptation and game theory. We apply beaconing, link quality assessment, and signal-to-noise ratio threshold techniques to the WirelessUSB protocol, and study the results under three characteristic workloads. We find that significant energy savings can be achieved without a compromise in throughput or latency by applying dynamic data rate policies.

4.2 Data Rate Adjustment Techniques

In the following sections we present prior research on data rate adjustment techniques including beaconing, link quality assessment, and signal-to-noise ratio threshold.

4.2.1 Beaconing

Nodes may use beaconing [19] to determine the most effective data rate for a given communication link. In the beaconing method (sometimes called sub-beaconing), the host node broadcasts periodic beacons at all data rates starting from the highest and continuing down to the lowest. The first beacon that the receiving device interprets successfully determines the data rate necessary for further communication. This method allows the receiving node to use periodic cross-communication from the master to gauge the quality of the transmission medium. If a device successfully interprets a master's beacon at a given data rate, then that device can theoretically transmit to the master at that data rate.

Depending on the beaconing interval, this primitive rate-changing mechanism may hinder a device's ability to rapidly adapt to changing conditions. However, the simplicity of its implementation makes it ideal for use in memory-constrained WirelessUSB devices. In addition, its master-centric approach to handling beacon transmissions is a good fit for use in star topology usage models.

4.2.2 Link Quality Assessment

Nodes can adjust their data rates by monitoring channel quality [19, 20, 21, 22, 23]. In this method, nodes determine the appropriate data rate based on packet success and failure. Kim and Bambos [22] determine the maximum allowable data rate by adaptively probing the channel. Sheu, Lee, and Chen [19] propose a Hybrid Handshake Protocol that monitors transmission status. If a transmission fails, the protocol assumes that movement occurred and mandates that subsequent transmissions employ a lower level modulation scheme (lower data rate).

Heusse et. al. [23] propose a method of determining channel quality, called *Idle Sense*, an access method based on CSMA that senses the mean number of idle slots between transmission attempts in order to control the contention window and estimate the frame error rate requirements. Devices then utilize this information to determine the appropriate data rate.

Lacage et. al. [20] propose a low-latency rate adaption mechanism called Adaptive Auto Rate Fallback (AARF). In this mechanism, a device periodically attempts to upgrade to a higher level modulation (higher data rate). If such an upgrade fails, the device resumes transmission using the previous lower level modulation. After a period of time expires, the device again attempts to upgrade. AARF adaptively changes the upgrade attempt interval in an effort to minimize energy consumption due to excessive upgrade attempts.

4.2.3 Signal-to-Noise Ratio Threshold

Another method of rate adjustment requires nodes to periodically calculate the signal-to-noise ratio of the communication link [15]. Since each data rate tolerates a certain bit error rate (BER), each data rate has a certain SNR threshold at which communication becomes possible. Communicating nodes therefore determine their appropriate data rate by calculating the current SNR at runtime, and then selecting the data rate whose SNR threshold the current SNR does not exceed.

This method requires some initial analysis and a priori assumptions about the propagation model of the transmission medium which may affect accuracy in practice.

Likewise, fixed thresholds prevent communicating devices from adapting to changing conditions at runtime. Also, the signal strength readings used to calculate the SNR at runtime may vary as much as 5% between readings, making it difficult to ascertain the correct data rate.

4.3 Dynamic Data Rate Techniques for WirelessUSB

In this study, we selected three representative techniques to implement in the WirelessUSB protocol: 1) Beaconing; 2) Link Quality Assessment; and 3) SNR Threshold Adaptation. In the following sections we discuss each of these within the context of WirelessUSB, including specific implementation details.

4.3.1 Beaconing

In practice, WirelessUSB devices switch to a low-power idle mode when inactive, and cannot remain in a costly receive mode to accept these periodic beacons. Neither can tight synchronization with the master be guaranteed to allow them to switch to receive mode during beacon periods. As such, the slaves themselves must perform the beaconing. We refer to this policy as *Slave Beaconing* to differentiate it from Host Beaconing, in which the host sends beacons. Although not practical for WirelessUSB, we include Host Beaconing in all results for comparison.

4.3.2 Link Quality Assessment

The link quality assessment policy determines data rate based entirely on packet success, measured by the receipt of acknowledgements. This policy mandates that the transmission data rate drop to the next lower encoding after each unsuccessful packet. This method attempts to exploit the fact that WirelessUSB devices can change data rates on a per-packet basis by modifying the encoding. Such a policy may facilitate a lower latency by placing priority on maintaining communication as opposed to maintaining a high data rate.

The link quality assessment policy attempts a data rate upgrade after 20 successful packets at the current data rate. If this upgrade is unsuccessful, the device immediately returns to the previous data rate.

4.3.3 Signal-to-Noise Ratio Threshold Adaptation

The signal-to-noise (SNR) ratio threshold policy employs pre-computed SNR thresholds for each data rate. Devices compute the runtime signal-to-noise ratio each time they receive an ACK from the master. They then determine the highest data rate to employ based on which data rate threshold the current SNR exceeds.

Receiver Signal Strength Indicator (RSSI) readings occur in a WirelessUSB radio at two distinct times. The first occurs when a radio receives a Start of Packet (SOP) signal. The RSSI hardware engages, calculating the RSSI of the SOP. This, for all intents and purposes, is the RSSI of the actual data being received. The second time a WirelessUSB radio calculates an RSSI occurs when a radio is in receive mode and the RSSI hardware engages, but does not receive an SOP signal. This results in a background noise level RSSI. WirelessUSB devices can then compute the runtime SNR by dividing the data RSSI reading by the latest noise RSSI reading.

This method of rate adjustment is similar to other pre-computed threshold rate selection methods, except that the thresholds may change over time. This method keeps track of the SNR and packet success for each data rate in order to adjust or "adapt" to changing conditions.

4.4 Performance Metrics

We employ four metrics to evaluate the various data rate policies: 1) Energy per node; 2) Throughput per node; 3) Packet delivery ratio; and 4) Packet latency. These metrics are discussed in the following sections.

4.4.1 Energy Per Node

The amount of power used for a given topology and usage model is the main factor in determining battery life. It is therefore important that power be minimized

for any given workload to extend battery life. Since we deal with networks of more than one transmitting node, we use the average energy consumed per participating node, providing a more accurate and useful metric than the total simulation power.

In this study, we assume that all slave nodes are power-constrained devices, but do not make the same assumption for master nodes. As a consequence, we do not include power consumption of master nodes in this metric.

In addition, we do not include idle mode energy in this metric since its effect is negligible (approximately 1 mA) during the constant transmission periods simulated in this research.

4.4.2 Throughput Per Node

We model the average rate at which data flows through a system as effective throughput (or "goodput") as seen from an application-level perspective. This metric empowers us to quantify the negative impact that dynamic power policies may have on throughput.

4.4.3 Packet Delivery Ratio

The ratio of the number of successfully delivered packets to the total number of packets transmitted aids in determining the reliability and robustness of the protocol. This ratio proves significant when packets are lost due to incorrect data rate settings.

4.4.4 Packet Latency

Packet latency is measured as user-perceived latency — if a packet takes 1 ms to transmit, and has to be sent twice before successful reception, then the user-perceived latency would be 2 ms or more, depending on the time taken to determine the need to retransmit as well as any other random retransmission delays. Measuring latency in this manner allows us to better gauge the suitability of the proposed policies and the degree to which application-level constraints can still be satisfied.

4.5 Simulation Method

In this section we describe the methods used in developing and executing the simulations for this study.

4.5.1 ns-2

The protocol and mechanisms tested in this research are written and simulated in ns-2 [10], a reliable simulator with extensive libraries to facilitate wireless simulations. ns-2 is a discrete event simulator with a reliable scheduler that ensures simulation accuracy. It also has a robust framework that is well-suited for writing and testing new protocols.

For these studies, we modified the Berkeley Network Simulator to enable individual nodes to dynamically adjust their transmission data rate. We adjusted the physical layer so that variation in the data rate affects the transmission range due to bit errors. Also, the carrier sense and retransmission timeouts were modified to account for much slower data rates than were employed in previous studies (see Table 4.1). The WirelessUSB routing layer was also modified to handle intelligent data rate adaptation.

4.5.2 Simulation Methodology

We employed simulations to test these data rate adaptation techniques in conjunction with the mobility models described in this section. We ran each simulation for a duration of 5 minutes and obtained the final results by averaging the intermediary results of 30 iterations of random topologies and random node traffic start times.

The constant data rate policy (also referred to as "No Policy") always transmits packets at the lowest, most robust data rate and is equivalent to not using any dynamic policy at all. No Policy, therefore, constitutes a control to which the other dynamic data rate policies are compared.

Topology

We consider various sizes of star topologies composed of slave nodes communicating in varying proximity to a master node. For each individual simulation, however, the number of slave nodes remains constant. In other words, slave nodes do not leave or join the network during a simulation.

The simulation environment is a 70 m $\times$ 70 m area. A fixed, non-power constrained master node is placed in the center of the room, and all other slave nodes are placed at a random starting location.

We vary network densities by repeating the simulations with a different number of slave nodes, ranging from 1 to 30. The random movement of all slave nodes is described as follows. All slave nodes relocate at random times to random locations. The velocity of each node varies from 1 to 6 meters per second. Once nodes reach their destination, they remain stationary for a random amount of time (between 1 to 5 seconds) before proceeding to move to another random location at a random velocity.

Once we generate the random topologies, we store them and reuse them in all subsequent experiments. This is vital to ensure that statistical differences in power policies are not the result of differing topologies. The same random topologies are used for each experiment while varying the data rate policy.

Workloads

We consider three relevant workloads based on characteristic usage models. The workloads for each simulation are designed to accurately model normal utilization of various devices such as keyboards, remote controls, and PDAs.

The first workload under study is referred to as the *HID workload*, characterizing a human interface device, such as a mouse or keyboard. For the purpose of this characteristic workload, we simulate a mouse using a workload of 6 bytes every 30 ms.

The second workload under consideration is referred to as the *remote workload*, which characterizes a TV/stereo remote control. We simulate a remote control using

a workload of 4 bytes every 2 seconds. Such a workload is characteristic of a typical channel-surfer, who presses the channel up/down button on the remote approximately every two seconds.

The third workload under consideration is the *file transfer workload*, characterized by the transfer of a file from one computing device to another. Many usage scenarios apply this type of workload, such as a firmware update, a PDA synchronization, or an image transfer from a digital camera. A file transfer maximizes the available bandwidth of a WirelessUSB device, which, depending on the encoding employed, is about 250 Kbps. For the purpose of this research, we simulate a file transfer using a workload of 32 bytes every 20 ms.

4.6 Results

This section presents the ns-2 simulation results for each usage model. First, we describe the results of several preliminary studies performed to determine proper settings of key parameters. Finally, we present the results of each simulated workload.

Retransmission delay is a key parameter. If a communicating node does not receive an ACK after a sufficient ACK timeout, the node must wait for a random period of time, from one to X times the duration of a single packet transmission before attempting a retransmission. Since transmission times vary depending on the data rate employed, we must ascertain the appropriate timeout interval before a node should attempt a retransmission. Figure 4.1 shows the results of varying the random retransmission delay range from 1 to 2 packet transmission durations through 1 to 8 packet transmission durations.

Results of this study reveal that sufficient throughput is obtained by waiting no longer than a random time between 1 to 4 packet transmission durations before retransmitting a packet. Waiting any longer needlessly increases average packet delay, and should not be considered.

The operation of both the Beaconing and Slave Beaconing data rate policies requires the specification of a beacon interval, which is the time between successive beacons. The precise value for the beacon interval was unknown prior to this research,

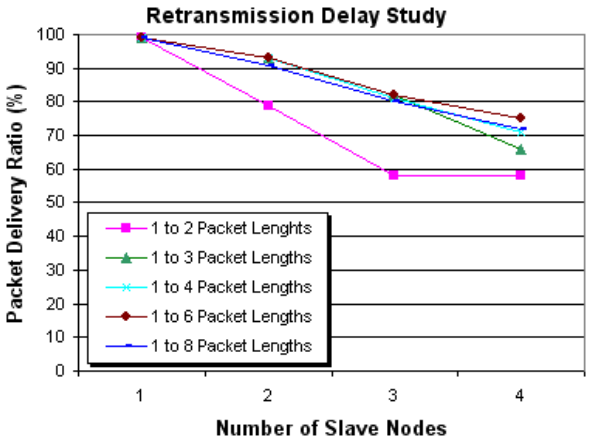


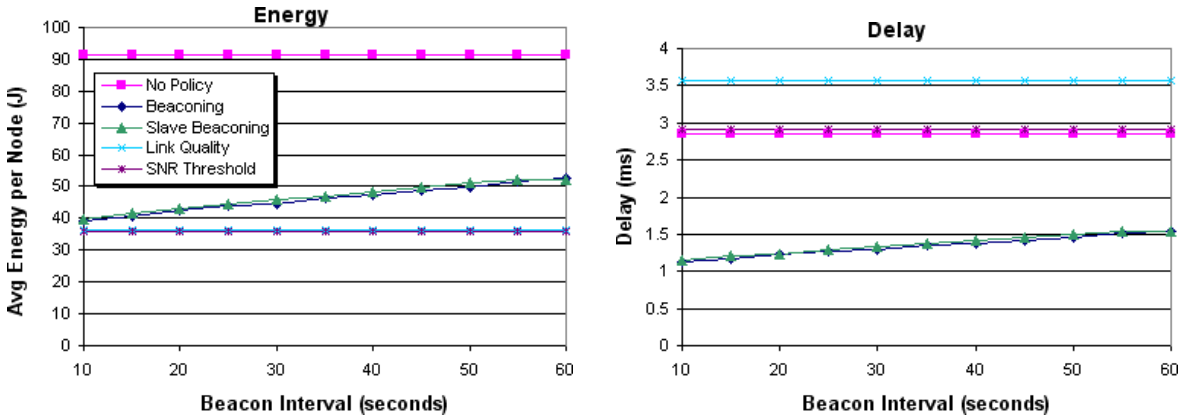
Figure 4.1: Retransmission Delay Study for a 180 m × 180 m Area, 1-4 Transmitting Nodes under the HID Workload.

and may vary considerably due to mobility. Frequent beaconing may lead to overuse of the spectrum and increased power consumption. Infrequent beaconing may lead to sub-optimal data rates for extended periods of time. If nodes remain stationary, and network conditions remain constant, a long beacon interval is desirable. However, if network conditions change, and mobility is present, then a short beacon interval is desirable.

Figure 4.2 shows the results of the beacon interval study for a 70 m × 70 m area, a maximum node velocity of 6 m/s, and a network size of 3 slave nodes operating under HID workloads. These results show that shorter beacon intervals (around 10 to 20 seconds between beacons) produce the greatest energy savings and the lowest latencies as maximum node velocities approach 6 m/s. The packet delivery ratio remains close to 99% and throughput remains unaffected by beacon interval changes.

A final key parameter is packet tolerance, which refers to the number of unsuccessful packets before reacting, or in this case, downgrading the data rate.

Figure 4.3 shows the results of the tolerance study for a 70 m × 70 m area, a maximum node velocity of 6 m/s, and a network size of 3 slave nodes operating under HID workloads. The throughput in all cases behaves similar to the packet delivery ratio, and is therefore not shown. These results show that a tolerance of 1 actually



(a) Energy

(b) Delay

Figure 4.2: Beacon Interval Study for a 70 m × 70 m Area, 3 Transmitting Nodes under the HID Workload.

reduces energy consumption and delay with no significant decrease in throughput for beaconing policies. However, increasing the tolerance threshold for both the Link Quality and SNR Threshold policies immediately produces an undesirable increase in delay and decrease in throughput. However, as seen in previous power policy studies, a tolerance of 1 may be desirable for dense networks, and thus we choose to proceed with a tolerance threshold of 1 packet for all future studies.

In light of our findings regarding retransmission timeouts and beaconing, all future studies are performed using a random timeout of 1 to 4 packet lengths and a beacon period of 20 seconds, unless otherwise stated. Tolerance to packet failure is fixed at 1 packet.

4.6.1 HID Workload

The results of the HID workload are shown in Figure 4.4. We observe that overall the best performing data rate policy under this workload is SNR Threshold. A close second is Link Quality, followed by Beaconing, then Slave Beaconing. Both beaconing policies, however, provide superior latency for networks of fewer than 8 nodes. This is explained by their passive approach to data rate upgrades, which are

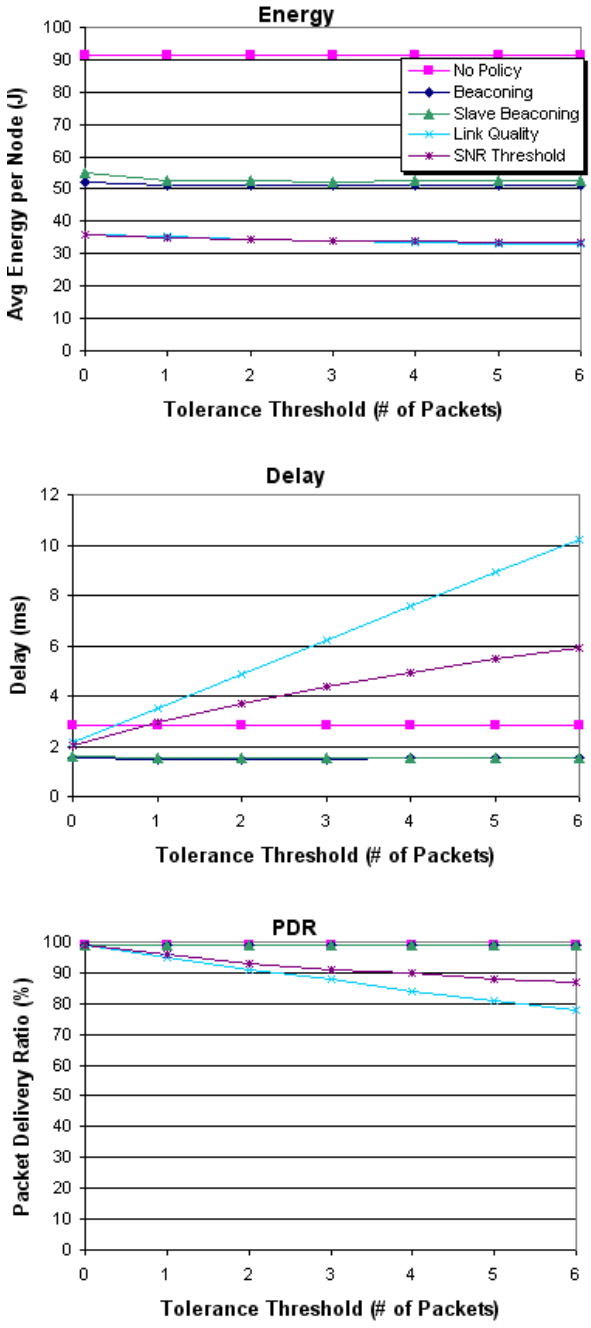


Figure 4.3: Tolerance Study on a 70 m × 70 m Area, 3 Transmitting Nodes under the HID Workload.

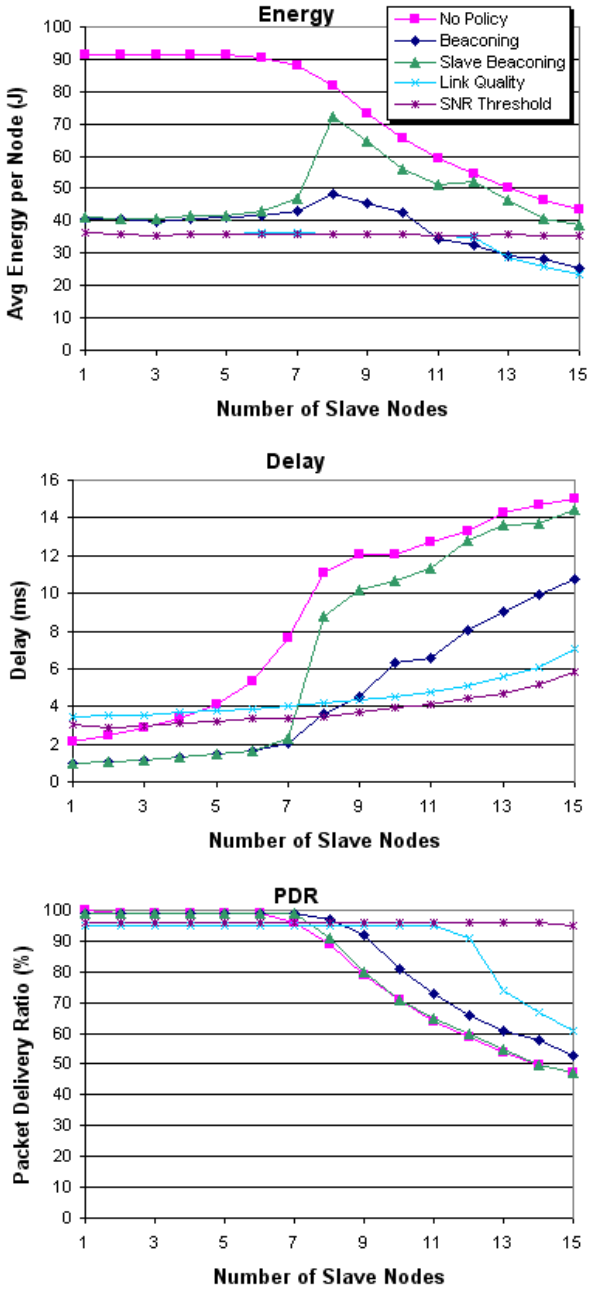


Figure 4.4: HID Study on a 70 m × 70 m Area.

performed only when slave nodes successfully receive a beacon for a data rate higher than their operating data rate, and only as often as the beacon interval permits. For these results, this occurs every 20 seconds.

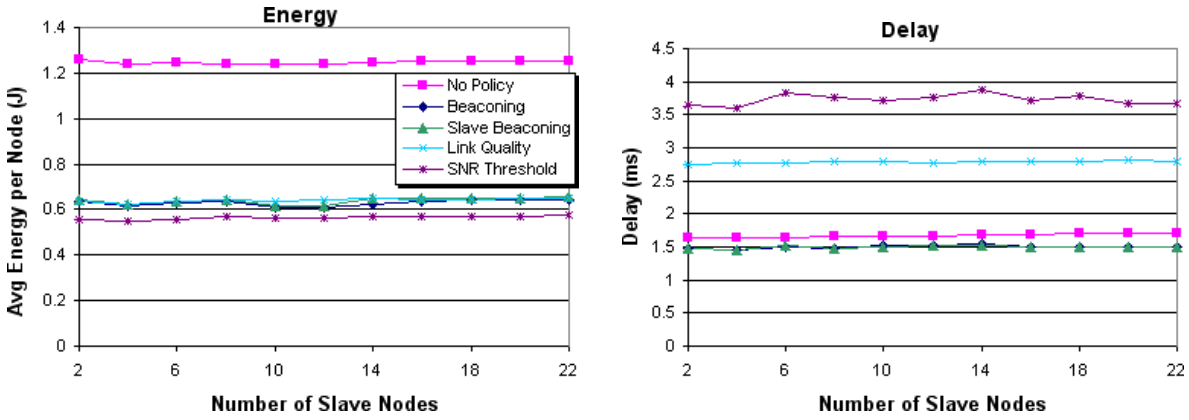
Also interesting to note is each policy's capacity to support networks of varying densities. For Slave Beaconing the packet delivery ratio begins to fall after 7 slave nodes and does not perform any better than No Policy in this regard. However, Beaconing is able to support 1 additional slave node (for a total of 8) before delivery performance suffers. Link Quality is able to support 4 additional nodes (11 total), and SNR Threshold is able to support 8 additional nodes (for a total of 15).

Another point of interest is the latency improvement over No Policy for all other policies when the network size is greater than 4 slave nodes. This is the first time we see not only the energy use for all policies remaining well under the values obtained by No Policy, but also the delay and throughput. Prior research reveals that when a node employs a power policy to conserve power, latency is compromised (although policies generally attempt to minimize this negative effect). Here, we find that because an increase in the data rate also reduces transmission time, we can achieve a decrease in energy consumption without compromising other performance metrics such as latency and throughput.

4.6.2 Remote Workload

The results of the remote workload study are shown in Figure 4.5. PDR and throughput (not shown) remain high throughout all tests since this workload does not begin to saturate the total available bandwidth. Unlike the HID workload, the remote workload shows the characteristic tradeoff between energy savings and delay that we observe in pure power studies. The SNR Threshold policy reduces power consumption by an average of 54.88% but increases delay by 104.85%. Likewise, the Link Quality Assessment policy reduces power consumption by 48.68% but increases the average delay by 57.28%.

Both Beaconing and Slave Beaconing policies, however, reduce energy consumption by 49.20% and 48.56% respectively, while also reducing average delay by



(a) Energy

(b) Delay

Figure 4.5: Remote Workload (a) Energy and (b) Delay Results.

13.98% and 13.86%. This result demonstrates that Slave Beaconing is the preferred policy for the remote workload.

4.6.3 File Transfer Workload

The results of the file transfer workload study are shown in Figure 4.6. We observe that the delivery ratio (and consequently the throughput) for No Policy using this workload immediately drops for network densities greater than 1 transmitting node. This is due to the increase in transmission duration when transmitting at the lowest data rate.

Overall, the Link Quality policy saves the greatest amount of energy (43.22%). However, the SNR Threshold policy provides comparable energy savings (39.01%) while preserving a high packet delivery ratio and maintaining low latency for network sizes greater than 2 slave nodes. Delay is reduced by an average of 54.37% by employing the SNR Threshold policy and only 42.26% for the Link Quality Assessment policy. However, beaconing policies continue to provide the lowest latency under a certain network size, as observed in the other workload studies. For the file transfer workload, this threshold is 3 slave nodes.

The effectiveness of the SNR Threshold policy is due to its ability to react to

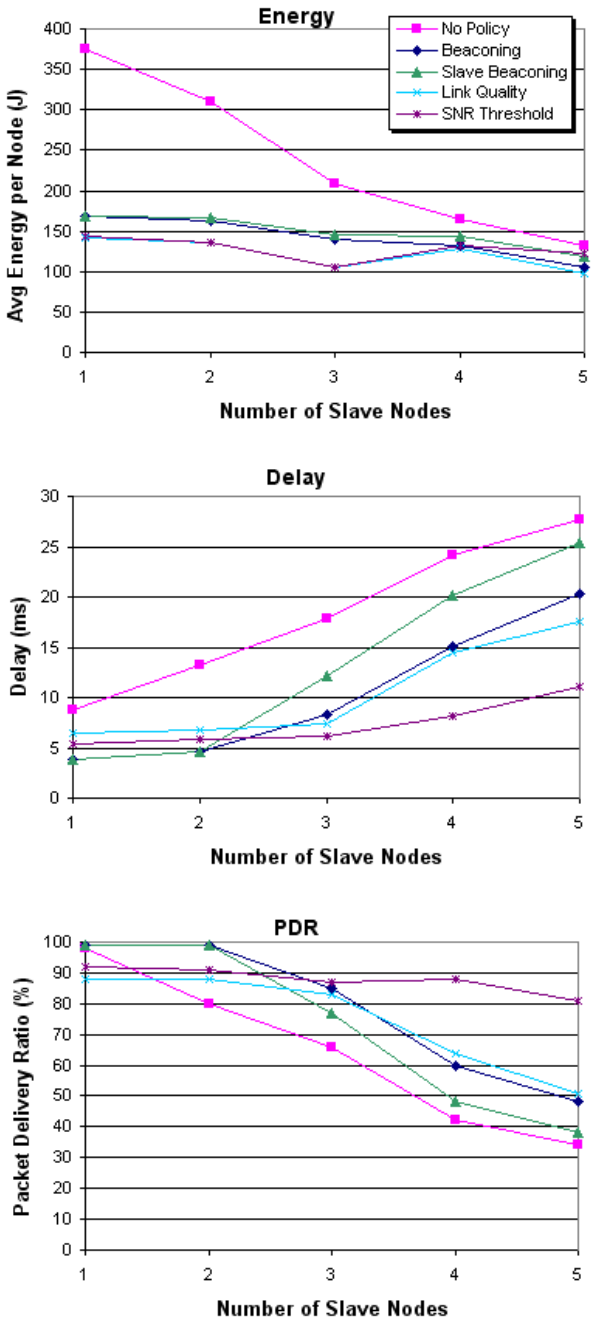


Figure 4.6: File Transfer Study on a 70 m × 70 m Area.

changing conditions without the need to generate retransmissions. The SNR Threshold policy requires no negotiation overhead — instead of reacting to packet failures, it reacts to changes in the RSSI readings each time a device receives an ACK from the master.

The packet delivery ratio of the SNR Threshold policy never reaches 100%, even for a single transmitting node. This is a result of RSSI jitter, causing retransmissions due to incorrect data rate settings. However, this shortcoming of the SNR Threshold policy proves to be worthwhile for greater network densities.

4.7 Conclusion

This study demonstrates that using a data rate policy not only improves energy consumption, but also delay and throughput.

Studying three characteristic workloads reveals that the best policy to use for the remote workload is a beaconing policy, and for all other workloads, the SNR Threshold policy. The reason beaconing policies outperform all others for the remote workload may be due to the fact that the beacon interval relative to the transmit interval is a factor of 10, whereas for all other workloads, this factor is closer to 1000.

Employing a dynamic data rate policy allows WirelessUSB devices to conserve a significant amount of their energy over no data rate policy while also decreasing latency and preserving throughput as network densities increase.

4.8 Future Work

For each simulation performed in this research, the power amplitude at each node was held constant. We propose to further explore the combinatory synergy of transmission power and data rate adjustment. We anticipate that this will yield the greatest energy conservation for a wide variety of conditions and usage models.

Chapter 5

Maximizing Battery Life of WirelessUSB Devices Using Dynamic Power and Data Rate Policies

Dynamic power and data rate policies for the WirelessUSB protocol enable devices to conserve power while operating at the highest data rate possible. Simulation results show that even greater performance can be achieved by combining both power and data rate policies.

5.1 Introduction

Power-constrained mobile devices must conserve energy in order to maximize their usefulness. Both transmission power and data rate adjustment strategies contribute to this end. In this paper we demonstrate the simultaneous use of various power and data rate policies in further minimizing energy consumption and optimizing latency and throughput in small to medium sized homogeneous wireless networks.

We employ Cypress Semiconductor Corporation's WirelessUSB radio and protocol as a case study for this research. WirelessUSB devices may employ eight transmission power levels, and five data rate encodings. Table 5.1 lists the maximum theoretical transmission ranges at varying transmission power amplitude (PA) levels and data rates.

Table 5.1: WirelessUSB Theoretical Transmission Ranges (meters).

		31.25 kbps				1 Mbps
	PA / Rate	0	1	2	3	4
Highest	7	130	68	22	7	2
	6	108	42	12	4	1
	5	76	23	7	2	0
	4	42	12	4	1	0
	3	23	7	2	0	0
	2	12	4	1	0	0
	1	7	2	0	0	0
Lowest	0	4	1	0	0	0

A WirelessUSB device has multiple options for transmitting at a given distance. For example a device has two options for transmitting a distance of 50 m; It may transmit at either a PA of 5 and a data rate of 0, or a PA of 7 and a data rate of 1. Which settings should the device utilize to maximize performance in terms of energy, latency and throughput? In this paper we address these issues.

In the following sections we outline our simulation method, describe our combined power and data rate policies, and then present results.

5.2 Simulation Method

We modified ns-2 [10] to simulate the WirelessUSB protocol with random topologies and random movement patterns in a 70 m × 70 m area (see Section 4.5.2). We analyzed the effects of our proposed policies using the performance metrics described in Section 4.4.

We analyze the results of each policy against 3 controls: No Policy, Power Only (PA Only) and Data Rate Only (DR Only). The specific policies used for the controls are optimal policies as described in the following paragraphs.

In Chapters 2 and 3 we demonstrated effective use of the eight transmission power levels available in a WirelessUSB device (see Table 2.1). For all workloads, the Incremental power policy demonstrated the best performance. We therefore use the Incremental Reestablishment power policy (see Section 3.3.1) as the power adjustment strategy in all combined policies.

In Chapter 4 we demonstrated effective use of the five data rates supported by WirelessUSB (see Table 4.1). For both the HID and file transfer workloads, the SNR Threshold Adaptation policy (see Section 4.3.3) performed best. We therefore use the SNR Threshold Adaptation policy when we only adjust data rate. For the same reason, the Slave Beaconing policy (see Section 4.3.1) is used as the Data Rate Only policy for the remote workload.

5.3 Combined Techniques

For this study, we selected two representative techniques to implement in the WirelessUSB protocol: 1) Link Quality - Data Rate First, and 2) Combined Incremental and SNR Threshold Adaptation. In the following sections we discuss each of these within the context of WirelessUSB, including specific implementation details.

5.3.1 Link Quality - Data Rate First (LQDRF)

This policy combines the reactive link quality assessment policies for both power and data rate adjustment. This policy adjusts transmission power using the

Incremental Reestablishment policy, and adjusts data rate using the Link Quality Assessment policy.

Preliminary study of the relationship between power and data rate reveals that it is more cost-effective in terms of energy consumption to reduce the data rate than the transmission power level. For this reason, this policy coordinates power and data rate adjustment by upgrading the data rate first, followed by upgrading transmission power. Reaction to a failed transmission, therefore occurs in reverse order. First this policy relaxes the transmission power level, followed by the data rate until communication resumes.

5.3.2 Combined Incremental and SNR Threshold Adaptation (I+SNR)

This policy is a combination of the Incremental Reestablishment power policy for transmission power control, and the SNR Threshold policy for data rate adjustment. This policy does not require collaboration between the policies since the SNR Threshold policy determines the optimal data rate upon receiving an ACK and the Incremental policy determines the optimal transmission power based on link quality assessment.

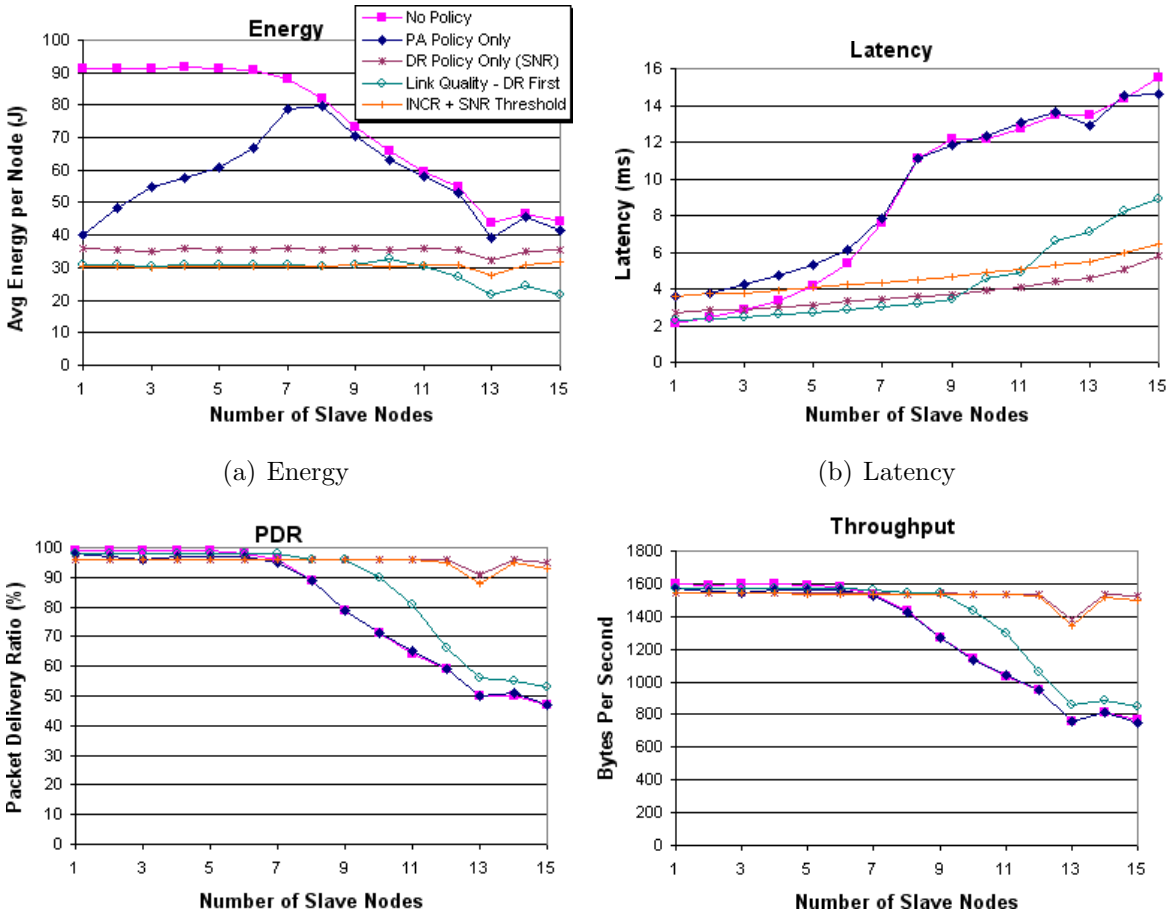
5.4 Results

This section presents the ns-2 simulation results for each workload.

5.4.1 HID Workload

The results of the HID workload are shown in Figure 5.1. For 1 to 9 slave nodes, the LQDRF policy obtains the lowest latency and energy consumption. For 10 to 15 slave nodes, the Data Rate Only policy obtains the lowest latency. However, since we are primarily interested in maximizing battery life, the I+SNR Threshold policy exhibits similar latencies, maintains a high throughput and low energy consumption, and therefore should be used for network sizes greater than 10 slave nodes. The lower amount of energy consumed by the LQDRF policy for greater than 11 slave nodes is due to a reduction in throughput and is not considered optimal.

For network sizes up to 10 slave nodes, the LQDRF policy decreases energy consumption over the previous best Data Rate Only policy by 13.22% on average, while also decreasing the average latency by 10.09%. For network sizes above 10 slave nodes, the I+SNR Threshold policy decreases energy consumption over the previous best Data Rate Only policy by 12.80% on average, but incurs a latency increase of 18.10%.



(a) Energy

(b) Latency

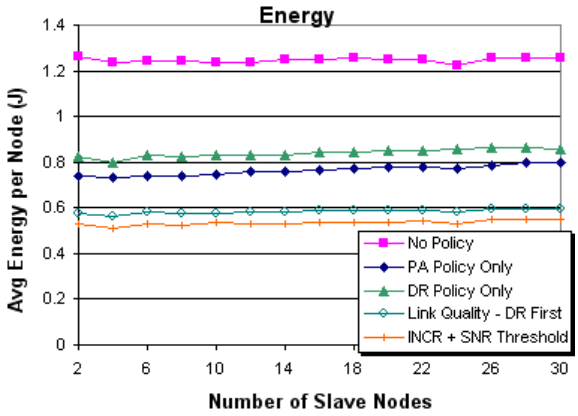
(c) Packet Delivery Ratio

(d) Throughput

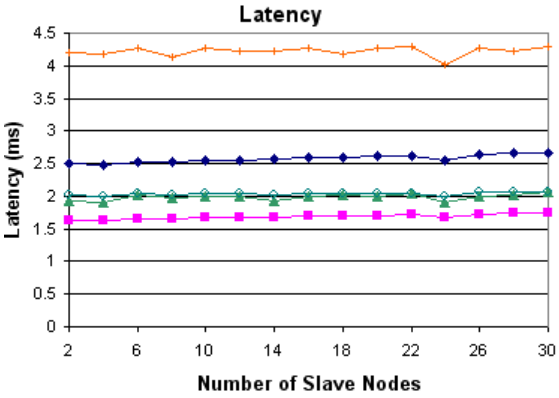
Figure 5.1: HID Workload Results. (a) Energy, (b) Latency, (c) Packet Delivery Ratio, and (d) Throughput.

5.4.2 Remote Workload

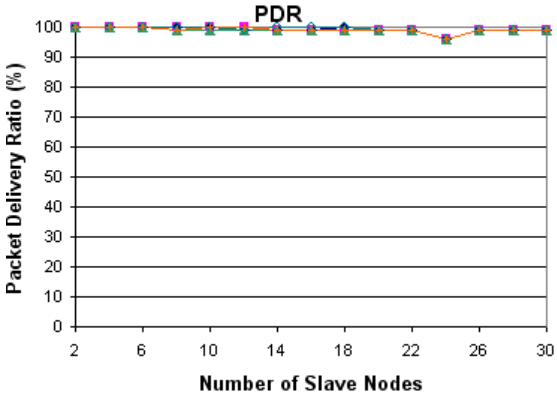
The results of the remote workload study are shown in Figure 5.2. Packet delivery ratio and throughput (not shown) remain high for all policies and are not a factor. The LQDRF policy decreases energy consumption over the previous best Data Rate Only policy by 23.46% while only increasing the average latency by 2.52%.



(a) Energy



(b) Latency



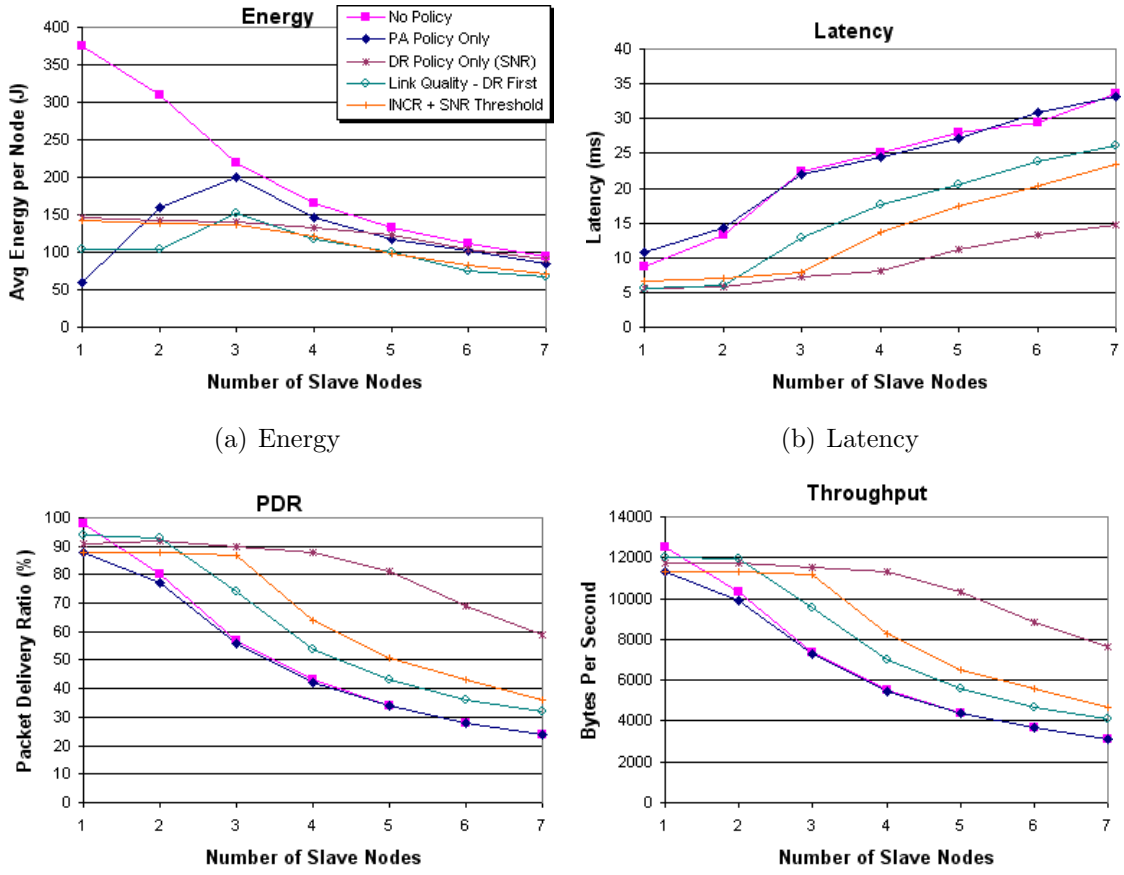
(c) Packet Delivery Ratio

Figure 5.2: Remote Workload Results. (a) Energy, (b) Latency, and (c) Packet Delivery Ratio.

The I+SNR Threshold policy decreases energy consumption over the previous best Data Rate Only policy by 36.29% (an additional 5.92% beyond the LQDRF

policy). Unfortunately, this policy increases latency by 112.54%.

5.4.3 File Transfer Workload



(a) Energy

(b) Latency

(c) Packet Delivery Ratio

(d) Throughput

Figure 5.3: File Transfer Workload Results. (a) Energy, (b) Latency, (c) Packet Delivery Ratio, and (d) Throughput.

The results of the file transfer workload study are shown in Figure 5.3. The intensity of this workload causes node throughput to plummet at 2 slave nodes, even when no policy is in force. The LQDRF policy staves off this decline for a network of 2 slave nodes, but then begins to plummet as well. Both the I+SNR Threshold and

the Data Rate Only policy manage to mitigate this negative effect for network sizes up to 3 slave nodes.

We see a crossover in latency performance at 2 slave nodes. For network sizes of 1 to 2 slave nodes, the LQDRF policy provides the lowest latency, decreasing overall latency by 43.19%. The I+SNR Threshold and Data Rate Only policy provide the lowest latency for network sizes above 2 slave nodes, decreasing overall latency when compared against No Policy by 41.08% and 40.94% respectively.

The file transfer workload demonstrates that the LQDRF policy provides the greatest energy conservation (40.63% over No Policy). The I+SNR Threshold policy decreases energy consumption by 36.88%. The PA Only policy demonstrates the lowest average energy consumption for a single slave node due to its lower packet delivery ratio, and is therefore not considered optimal.

5.5 Conclusion

Employing a combined power and data rate policy yields improved energy conservation over both a data rate only policy as well as a power only policy. In some instances, latency is also improved.

The I+SNR Threshold policy is the best policy to use for dense networks. The LQDRF policy provides maximum energy savings while also minimizing latency for sparse networks.

We summarize the results of all workloads in Figure 5.4. These graphs show the energy and latency savings over the optimal Data Rate Only policy. Similar results are obtained when compared against the optimal Power Only policy. Overall, the LQDRF policy is superior in performance in terms of these two metrics for both the HID and remote workload. For the file transfer workload, there is no clear dominant combinatory policy if we only observe energy and latency, since latency increases with energy savings.

However, if latency is not a concern, and we are more interested in energy efficiency, then the data-to-energy ratio summary found in Figure 5.5 demonstrates that the I+SNR Threshold policy is the most energy efficient for both the HID and

Figure 5.4: Workload Comparison of Energy and Latency Savings for Combined Policies.

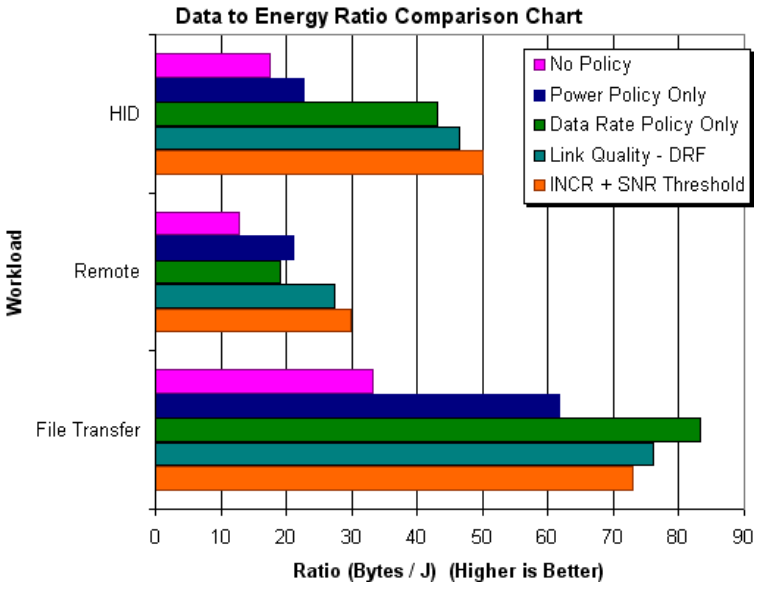


Figure 5.5: Data-to-Energy Ratio Comparison for all Workloads.

remote workloads. The most energy efficient policy for the file transfer workload is the Data Rate Only policy.

The combinatory synergy of transmission power and data rate adjustment yields the greatest energy conservation for a wide variety of conditions and usage models.

Appendix A

Simulation Model and Source Code

A.1 ns-2 Code Samples

Figure A.1 diagrams the ns-2 simulation model for WirelessUSB. We modified the original WirelessUSB implementation by Woodings and Pandey [13] to include intelligent power and data rate adjustment.

The protocol stack of the simulation model emulates the OSI protocol stack using a separate class file for each layer. Implementation of the WirelessUSB protocol requires customizing the operation of the lowest three layers, specifically, the network, data link, and physical layers. Each layer's code contains a `recv()` and `send()` method in order to pass the current packet object up or down the protocol stack. Some classes, such as `wireless-phy.cc` have helper methods, such as `sendUp()` and `sendDown()` that specifically send the packet object up or down the protocol stack respectively.

Section A.2 is an example ns-2 TCL script used to drive the simulations for this research thesis. Power and data rate policies are set on the routing agent of each node, which corresponds to the network layer.

Section A.3 contains source code samples from `wu.h` and `wu.cc` (located in the `/wu` directory) which comprise the Network layer of the OSI stack for the WirelessUSB protocol implementation. This layer is responsible for packet retransmissions and for dynamically adjusting a node's power level and data rate.

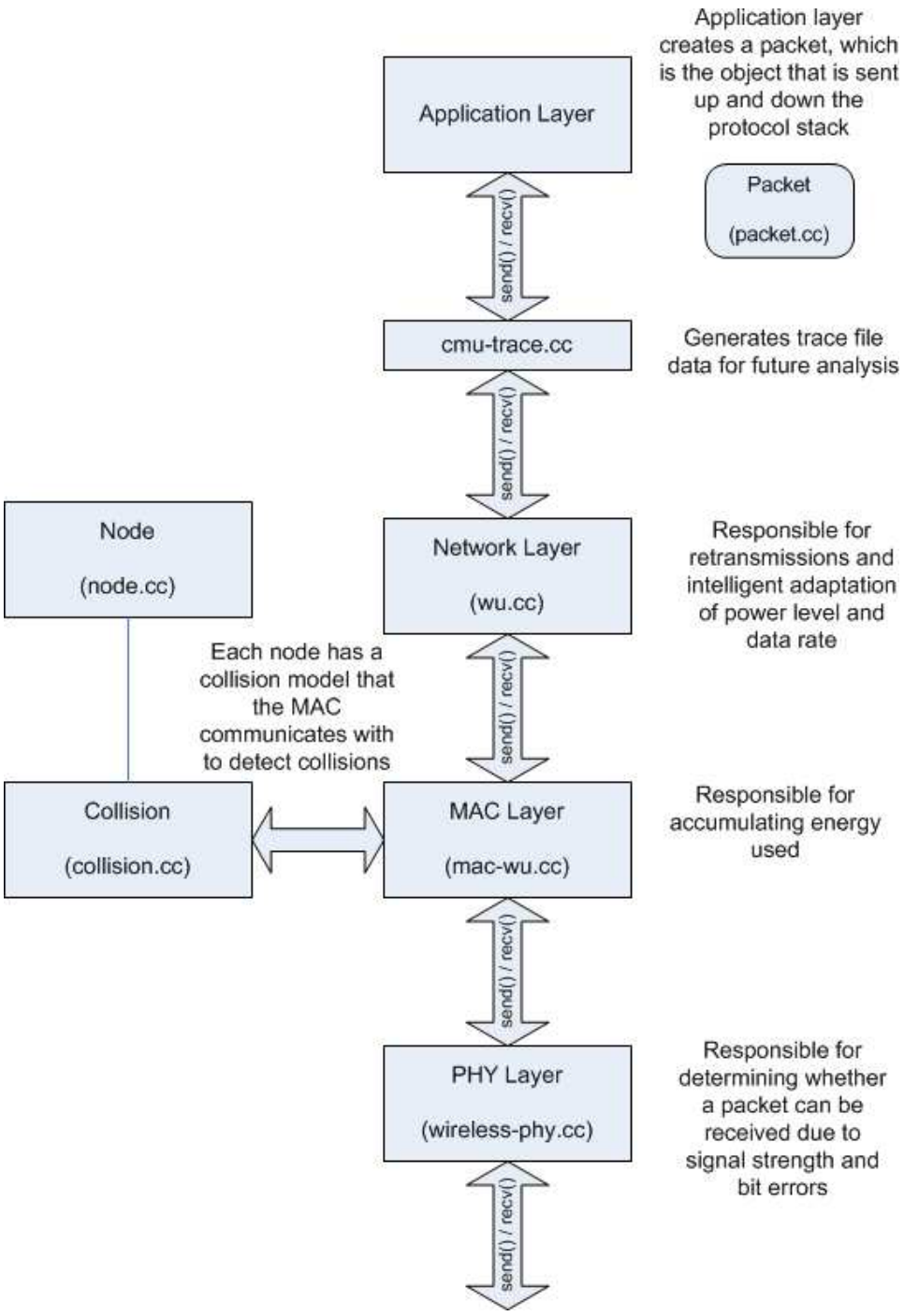


Figure A.1: WirelessUSB Simulation Model.

Section A.4 contains source code samples from `mac-wu.cc` and `mac-wu.h` (located in the `/mac` directory), which comprise the MAC layer functionality for the WirelessUSB protocol implementation. This layer is responsible for the accumulation of energy totals and RSSI sensing.

Section A.5 contains source code samples from `wireless-phy.cc` (located in the `/mac` directory), which comprises the physical layer functionality for the WirelessUSB protocol implementation. This layer is responsible for determining whether or not a packet can be received due to signal strength and bit errors.

Section A.6 contains the modifications made to `packet.h` (located in the `/common` directory) for the WirelessUSB protocol implementation. These modifications include the addition of a data rate member variable to track the data rate at which the packet was transmitted. The master then uses this information to transmit acknowledgements at the same data rate.

Section A.7 contains additions to the collision model found in `collision.cc` (also located in the `/common` directory) to facilitate calculating signal-to-noise ratios.

A.2 Example TCL Script

```
# WirelessUSB script for power and data rate policy experiments
# Copyright (c) 2006 JLB Software
# All rights reserved.

# ======================================================================
# Define Global Options
# ======================================================================

set val(chan)       Channel/WirelessChannel
set val(prop)       Propagation/TwoRayGround
set val(netif)      Phy/WirelessPhy
set val(mac)        Mac/MAC_WU
#set val(mac)        Mac/802_11
set val(ifq)        Queue/DropTail/PriQueue
set val(ll)         LL
set val(ant)        Antenna/OmniAntenna
set val(x)  70;
set val(y)  70;
set val(ifqlen)  50    ;#max packet in ifq
set val(seed)   30;
set val(adhocRouting)  WU
#set val(adhocRouting) DSR
set val(nn) 16;
set val(cp) "cbr.in"
set val(sc) "dest.in"
set val(stop)   300;
```

```
# ======================================================================
# Main Program
# ======================================================================

#
# Initialize Global Variables
#

# create simulator instance
set ns_     [new Simulator]

# setup topography object
set topo    [new Topography]

# create trace object for ns and nam
set tracefd [open output.tr w]
set namtrace    [open wireless-out.nam w]

$ns_ trace-all $tracefd
$ns_ namtrace-all-wireless $namtrace $val(x) $val(y)

# define topology
$topo load_flatgrid $val(x) $val(y)

#
# Create God
#
set god_ [create-god $val(nn)]

#
# define how nodes should be created
#

#global node setting

# ======================================================================
set numInterferingStations  0;

# Create channels (can be more than one with this new type. Not using -channelType)
set chan_1_ [new $val(chan)]

# ======================================================================
#Now, set up WU nodes.
# ======================================================================

$ns_ node-config -adhocRouting $val(adhocRouting) \
                 -llType $val(ll) \
                 -macType $val(mac) \
                 -ifqType $val(ifq) \
                 -ifqLen $val(ifqlen) \
                 -antType $val(ant) \
                 -propType $val(prop) \
                 -phyType $val(netif) \
                 -channel $chan_1_ \
                 -topoInstance $topo \
                 -agentTrace ON \
                 -routerTrace OFF \
                 -macTrace OFF

#  Create the specified number of nodes [$val(nn)] and "attach" them
#  to the channel.
```

```
for {set i $numInterferingStations} {$i < $val(nn) } {incr i} {
    set node_($i) [$ns_ node]
    $node_($i) random-motion 0 ;# disable random motion
}

for {set i $numInterferingStations} {$i < $val(nn) } {incr i} {
    set mac_($i) [$node_($i) getMac 0]
    $mac_($i) hasBluetooth 0
    $mac_($i) dataRate 0 31250 ;#250 kbps
#    $mac_($i) dataRate 0 250000 ;#250 kbps
    set rag_($i) [$node_($i) getRagent]
    $rag_($i) base-channel-number 10
    $rag_($i) node $node_($i);  #Create a reference to routing agent in node
}

#Bind initiation: Both, master and slave, should, ideally, be initiated at
#the same time. Corresponds to pressing of a button manually by the user.

set numMasters  1;
set numSlaves   15;

#set the power policy to 0 = NONE, 1 = INCREMENTAL 2 = AGGRESSIVE 3 = INCREMENT BY 2
set powerPolicy 1;
#set the data rate policy to 0 = NONE, 1 = BEACONING 2 = SLAVE_BEACONING 3 = LINK_QUALITY 4 = SNR_THRESHOLD
set dataRatePolicy  4;
#set the combined power and data rate policy to 0 = NONE, 1 = LINK_QUALITY_DR_FIRST
set combinedPolicy  0;

#The The manner in which upgrades are backed off 0 = FIXED (uses BackoffValue), 1 = LINEAR, 2 = EXPONENTIAL
set dataRateUpgradeBackoffMechanism 0;
#The number of successfull packets in succession before adjusting power level down
set dataRateBackoffValue    10;
#The number of retransmissions before adjusting power level up
set dataRateTolerance   1;

set totNodesWu  [expr {$numMasters} * [expr 1 + {$numSlaves}]]
puts "============================================="
puts "Total number of active WirelessUSB nodes is $totNodesWu"

set masterNode $numInterferingStations
for {set i $numInterferingStations} {$i < [expr $numInterferingStations + $totNodesWu]} {incr i} {
#    puts "i is $i"

    if {$i == $masterNode} {
        puts "Master node is $i"
        $mac_($masterNode) isAMaster 0
        $rag_($masterNode) isAMaster
        $rag_($i) pnCode [expr 1001 + $i]
    $rag_($i) beaconInterval_ 20
    } else {
        $rag_($masterNode) bind-initiate-time [expr 1.0 + [expr 0.1 * $i]]
        $rag_($i) bind-initiate-time [expr 1.0 + [expr 0.1 * $i]]

        #set the power policy, and data rate policy, and associated params
        $rag_($i) combinedPolicy $combinedPolicy
        $rag_($i) powerPolicy $powerPolicy
        $rag_($i) dataRateBackoff_ $dataRateBackoffValue
        $rag_($i) dataRateTolerance_ $dataRateTolerance

# Optional parameters to manually set the specific power level or data rate of a node
```

```
#        $mac_($i) powerLevel_    7
#        $mac_($i) dataRateLevel_ 2
    }

    #Settings for All nodes
    $rag_($i) dataRatePolicy $dataRatePolicy

    #enable / disable debugging at the router / mac level:    (NEVER BOTH AT THE SAME TIME)
    #$rag_($i) hasDebug 1 1.0
    #$mac_($i) hasDebug 1 2 21.00

    #$node_($i) dontPrintToTrace
    #$node_($i) timeToHalt $val(stop)
}

# Define traffic model
puts "Loading traffic model '$val(cp)'..."
source $val(cp)


# Define node movement model
puts "Loading mobility model '$val(sc)'..."
source $val(sc)

# Define node initial position in nam
for {set i 0} {$i < $val(nn)} {incr i} {

    # The function must be called after mobility model is defined
    $ns_ initial_node_pos $node_($i) 20
}

# Tell nodes when the simulation ends
for {set i 0} {$i < $val(nn) } {incr i} {
    #$ns_ at $val(stop).0 "$node_($i) reset";
}

# Optional display each node's energy usage
for {set i 0} {$i < $val(nn)} {incr i} {
    $ns_ at $val(stop).0001 "$mac_($i) energy"
}

$ns_ at $val(stop).0002 "puts \"NS EXITING...\n \" ; $ns_ halt"

puts $tracefd "M 0.0 nn $val(nn) x $val(x) y $val(y) rp $val(adhocRouting)"
puts $tracefd "M 0.0 sc $val(sc) cp $val(cp) seed $val(seed)"
puts $tracefd "M 0.0 prop $val(prop) ant $val(ant)"

puts "Starting Simulation..."
$ns_ run
```

A.3 WirelessUSB Network Layer

A.3.1 wu/wu.h

```
/*
Copyright (c) 2006 Cypress Semiconductor Corporation.  All Rights Reserved.
*/

...
```

```
#define RSSI_MIN_THRESHOLD -70.00
#define RSSI_MIN_SENSING_DBM -100.00
#define WU_RSSI_MAX 31              //Maximum RSSI value

...

#define WU_MAX_PA 7
#define WU_MIN_PA 0
#define POWER_POLICY_NONE 0
#define POWER_POLICY_INCREMENTAL 1
#define POWER_POLICY_AGGRESSIVE 2
#define POWER_POLICY_INCREMENTAL_PLUS_2 3

#define POWER_LEVEL_UPGRADE_FIXED 0
#define POWER_LEVEL_UPGRADE_LINEAR 1
#define POWER_LEVEL_UPGRADE_EXPONENTIAL 2

#define POWER_LEVEL_UPGRADE_BACKOFF_INCREMENT 3

#define MIN_POWER_LEVEL_UPGRADE_BACKOFF_VALUE 4
#define MAX_POWER_LEVEL_UPGRADE_BACKOFF_VALUE 64

#define WU_MAX_RATE_LEVEL 4
#define WU_MIN_RATE_LEVEL 0
#define WU_DATA_RATE_POLICY_NONE 0
#define WU_DATA_RATE_POLICY_BEACONING 1
#define WU_DATA_RATE_POLICY_SLAVE_BEACONING 2
#define WU_DATA_RATE_POLICY_LINK_QUALITY 3
#define WU_DATA_RATE_POLICY_SNR_THRESHOLD 4

#define WU_DATA_RATE_THRESHOLD_CONSTRAIN 0
#define WU_DATA_RATE_THRESHOLD_RELAX 1

#define WU_COMBINED_POLICY_NONE 0
#define WU_COMBINED_POLICY_DR_FIRST 1

#define WU_UPGRADE_POWER 1
#define WU_UPGRADE_DATA_RATE 2

////////////////////////////////////////////////////////////////////////////////

...

// Timer definitions for beaconing approaches
class wuBeaconTimer: public TimerHandler {
public:
        wuBeaconTimer(WU* a) : agent(a) {agent=a;}
        void    handle(Event*);
        void    expire(Event*);
        WU    *agent;
private:
        Event   intr;
};

class wuSubBeaconTimer: public TimerHandler {
public:
        wuSubBeaconTimer(WU* a) : agent(a) {agent=a;}
        void    handle(Event*);
        void    expire(Event*);
        WU    *agent;
```

```
private:
        Event   intr;
};

...

/*
  The Routing Agent
*/
class WU: public Agent {

  /*
   * make some friends first
   */

...

        friend class wuBeaconTimer;
        friend class wuSubBeaconTimer;

 public:
        WU(nsaddr_t id);

        void    recv(Packet *p, Handler *);
        void    retransmit();
        double  getRetransmitTO(Packet *packet, bool addRandomness);
        double  getMaxRoundTripTxTime(double psz, double drt, bool addRandomness);

        double  getTxPowerValueFromMac();
        double  getTxPowerValue(int powerLevel);
        double  getTxDataRateValue(int dataRateLevel);
        double  getMACEnergyUsed();
        inline int getDataRatePolicy() { return dataRatePolicy; };
        int     ifNeedToPrintWu(int);

...

        int num_times_retx;

        //Sub Beaconing variables
        bool subBeaconSent[WU_MAX_RATE_LEVEL+1];
        Packet * subBeaconPacket[WU_MAX_RATE_LEVEL+1];

 protected:
        int command(int, const char *const *);

        void decideWhatDataRateToUseUsingSNR();
        int powerPolicy;
        int powerLevelBackoff_; //Backoff value for fixed upgrade backoff attempt. # of successful packets before attempting upgrade
        int powerLevelTolerance_;
        int powerLevelUpgradeBackoffMechanism;  // Either FIXED (uses powerLevelBackoff_ value), LINEAR, or EXPONENTIAL

        int dataRatePolicy;
        int dataRateBackoff_;   //Backoff value for fixed upgrade backoff attempt. # of successful packets before attempting upgrade
        int dataRateTolerance_;
        int dataRateUpgradeBackoffMechanism;    // Either FIXED (uses powerLevelBackoff_ value), LINEAR, or EXPONENTIAL

        int combinedPolicy;     //Defines the manner in which both power and data rate policies should be combined

        double beaconInterval_;     //Time between successive beacons (for data rate adaptation)
```

```
        void            send(Packet *p, double delay);
        virtual void    sendBeaconResponse(nsaddr_t ipdst);

        void            recvDataRateBeacon(Packet *p);

...

        wuBeaconTimer           beaconTimer;
        wuSubBeaconTimer        subBeaconTimer;

private:
    int successfulPacketsAtCurrentTxPowerLevel;
    int successfulPacketsAtCurrentTxDataRate;
    bool attemptingToUpgradePowerLevel; // Will be set when there is an attempt to upgrade (decrease) power level
    bool attemptingToUpgradeDataRate;   // Will be set when there is an attempt to upgrade (increase) data rate
    int beaconResponseDataRate;
    int upgradePowerOrDataRate;     //Should power or data rate be upgraded next?
};
```

A.3.2 wu/wu.cc

```
/*
Copyright (c) 2006 Cypress Semiconductor Corporation.  All Rights Reserved.
@Manoj Pandey, Cypress Semiconductor Corp.
@Jeff Barlow, Cypress Semiconductor Corp.
*/

/*
 *  This is the routing layer module for WirelessUSB
 *
*/

...

/*  Constructor  */
WU::WU(nsaddr_t id) : Agent(PT_WU), retxtimer(this), rptimer(this), rssitimer(this),
    cstimer(this), bindtimer(this), gptimer(this), beaconTimer(this), subBeaconTimer(this)
{
    index = id;
    base_channel = 0;
    calculateSubChannels(base_channel);

    logtarget = 0;
    ifqueue = 0;
    isAMaster = 0;
    pn_code = PN1;
    masters_pn_code = PN1;
    numBadRSSIChannelsBeforeDuringRegularDataMode = 0;
    numBadRSSIChannelsBeforeDuringChannelSelect = 0;
    numPingsSentOnThisChannel = 0;
    numRoundsOfSubsetForPing = 0;
    mac_ = 0;
    node_ = 0;
    need_to_send_ack = 0;

    setMode(START_MODE);
    setPktTxMode(WU_STATUS_IDLE);
    hasDebug = 0;
    debugStartTime = 0.0;
```

```
    hasTrace = 0;
    pktTx_ = 0;

    for (int i=0; i < MAX_SLAVES_SUPPORTED; i++) {
        bind_times[i] = -1000.0;
    }

    gptimer.purpose = PURPOSE_INIT_BIND;
    gptimer.resched(DELTA_AFTER_ZERO);

    my_masters_network_id = -1; //Not bound to a master yet

    //Power dynamics variables
    powerPolicy = POWER_POLICY_NONE;
    successfulPacketsAtCurrentTxPowerLevel = 0;
    powerLevelBackoff_ = 10;
    powerLevelTolerance_ = 0;
    powerLevelUpgradeBackoffMechanism = POWER_LEVEL_UPGRADE_FIXED;
    attemptingToUpgradePowerLevel = false;

    //Data rate dynamics variables
    dataRatePolicy = WU_DATA_RATE_POLICY_NONE;
    successfulPacketsAtCurrentTxDataRate = 0;
    dataRateBackoff_ = 10;
    dataRateTolerance_ = 0;
    dataRateUpgradeBackoffMechanism = POWER_LEVEL_UPGRADE_FIXED;
    attemptingToUpgradeDataRate = false;

    //Start the beacon
    beaconInterval_ = 1.0;

    double randomness = id * 0.1;
    beaconTimer.resched(beaconInterval_ + randomness);

    //Combined Power and Data rate policy
    combinedPolicy = WU_COMBINED_POLICY_NONE;
    upgradePowerOrDataRate = WU_UPGRADE_DATA_RATE;  //Start with Data Rate

}


int WU::command(int argc, const char*const* argv)
{

...

    else if (strcmp(argv[1], "powerPolicy") == 0) {
        //Only Slave Nodes have power policies
        if (!isAMaster)
            powerPolicy = atoi(argv[2]);

        //printf("wu.cc: node_(%d) The power policy is set to %d \n", index, powerPolicy);
        return TCL_OK;
    }
    else if (strcmp(argv[1], "powerLevelBackoff_") == 0) {
        powerLevelBackoff_ = atoi(argv[2]);
        //printf("wu.cc: node_(%d) The power policy backoff is set to %d \n", index, powerLevelBackoff_);
        return TCL_OK;
    }
    else if (strcmp(argv[1], "powerLevelTolerance_") == 0) {
        powerLevelTolerance_ = atoi(argv[2]);
```

```
        //printf("wu.cc: node_(%d) The power policy tolerance is set to %d \n", index, powerLevelTolerance_);
        return TCL_OK;
    }
    else if (strcmp(argv[1], "powerLevelUpgradeBackoffMechanism") == 0) {
        powerLevelUpgradeBackoffMechanism = atoi(argv[2]);
        //printf("wu.cc: node_(%d) The power policy upgrade backoff mechanism is set to %d \n", index, powerLevelUpgradeBackoffMechanism);

        if (powerLevelUpgradeBackoffMechanism != POWER_LEVEL_UPGRADE_FIXED)
        {
            powerLevelBackoff_ = MIN_POWER_LEVEL_UPGRADE_BACKOFF_VALUE;
        }

        return TCL_OK;
    }
    else if (strcmp(argv[1], "dataRatePolicy") == 0) {
        //Set the data rate policy
        dataRatePolicy = atoi(argv[2]);

        //Make sure the master never uses a data rate policy other than 0 or 1 (beaconing)
        if (isAMaster && (dataRatePolicy != WU_DATA_RATE_POLICY_BEACONING))
        {
            dataRatePolicy = WU_DATA_RATE_POLICY_NONE;
//          printf("wu.cc: node_(%d) setting data rate policy to none.\n", index);
        }

//      printf("wu.cc: node_(%d) The data rate policy is set to %d \n", index, dataRatePolicy);
        return TCL_OK;
    }
    else if (strcmp(argv[1], "dataRateBackoff_") == 0) {
        dataRateBackoff_ = atoi(argv[2]);
        //printf("wu.cc: node_(%d) The data rate backoff is set to %d \n", index, powerLevelBackoff_);
        return TCL_OK;
    }
    else if (strcmp(argv[1], "dataRateTolerance_") == 0) {
//      if ((dataRatePolicy == WU_DATA_RATE_POLICY_BEACONING))
//          dataRateTolerance_ = 1;
//      else
            dataRateTolerance_ = atoi(argv[2]);
        //printf("wu.cc: node_(%d) The data rate tolerance is set to %d \n", index, powerLevelTolerance_);
        return TCL_OK;
    }
    else if (strcmp(argv[1], "dataRateUpgradeBackoffMechanism") == 0) {
        dataRateUpgradeBackoffMechanism = atoi(argv[2]);
        //printf("wu.cc: node_(%d) The power policy upgrade backoff mechanism is set to %d \n", index, powerLevelUpgradeBackoffMechanism);

        if (dataRateUpgradeBackoffMechanism != POWER_LEVEL_UPGRADE_FIXED)
        {
            dataRateBackoff_ = MIN_POWER_LEVEL_UPGRADE_BACKOFF_VALUE;
        }

        return TCL_OK;
    }
    else if (strcmp(argv[1], "beaconInterval_") == 0) {
      beaconInterval_ = atoi(argv[2]);
//      printf("wu.cc: node_(%d) The beacon interval is set to %.1f\n", index, beaconInterval_);

      return TCL_OK;
    }
    else if (strcmp(argv[1], "combinedPolicy") == 0) {
        //Only Slave Nodes have power policies
        if (!isAMaster)
```

```
            combinedPolicy = atoi(argv[2]);

        printf("wu.cc: node_(%d) The combined power & dr policy is set to %d \n", index, combinedPolicy);
        return TCL_OK;
    }


    //This case provides a link to this routing layer at the node level - BarlowJ
    else if (strcasecmp(argv[1], "node") == 0)
    {
        TclObject *obj;
        if ( (obj = TclObject::lookup(argv[2])) == 0)
        {
            fprintf(stderr, "wu.cc: %s lookup of %s failed\n", argv[1], argv[2]);
            return TCL_ERROR;
        }

        //printf("wu.cc: Setting node pointer in node to me!\n");
        Node *nodePtr_;
        nodePtr_ = (Node *) obj;
        nodePtr_->routingLayer = this;

        return TCL_OK;
    }

...

}

...

void wuPktRetxTimer::handle(Event*)
{
    if (agent->ifNeedToPrintWu(2)) {
        printf("wu.cc: [%lf] node_(%d) is handling a retransmit timer to see if this packet needs to be retransmitted \n",
                NOW, agent->index);
    }

    agent->retransmit();

    return;
}

void wuPktRetxTimer::expire(Event*)
{
    return;
}

void wuBeaconTimer::handle(Event*)
{
    //Only masters send out beacons using regular beaconing mode
    if ((agent->isAMaster) && (agent->dataRatePolicy == WU_DATA_RATE_POLICY_BEACONING))
    {
        if (agent->ifNeedToPrintWu(2)) {
            printf("wu.cc(%d): node_(%d) is handling a beacon timer at time %f \n", __LINE__, agent->index, NOW);
        }

        if (NOW > 5.0)  //CANNOT BEACON UNTIL ALL NODES ARE BOUND. Prevent any Beaconing until at least 5 seconds
            //Send out a sub-beacon at each data rate
            agent->prepareAndSendDataRateSubBeacons(IP_BROADCAST);      //Send to All nodes on this frequency!)
    }
```

```
    //Only slaves send out beacons in slave beaconing mode
    if ((!agent->isAMaster) && (agent->dataRatePolicy == WU_DATA_RATE_POLICY_SLAVE_BEACONING))
    {
        if (agent->ifNeedToPrintWu(2)) {
            printf("wu.cc(%d): Slave node_(%d) is handling a beacon timer at time %f \n", __LINE__, agent->index, NOW);
        }

        if (NOW > 5.0)  //DON'T BEACON UPON INITIAL SETUP. WAIT FOR NEXT INTERVAL.
            //Send out a sub-beacon at each data rate
            agent->prepareAndSendDataRateSubBeacons(agent->my_masters_network_id);
    }

    //Mutual code
    if ((agent->dataRatePolicy == WU_DATA_RATE_POLICY_BEACONING) ||
        (agent->dataRatePolicy == WU_DATA_RATE_POLICY_SLAVE_BEACONING))
    {
        //Reschedule every beaconInterval_
        if (NOW < 5.0)  //DON'T BEACON UPON INITIAL SETUP. WAIT FOR NEXT INTERVAL.
            agent->beaconTimer.resched(10.0);
        if (NOW < 12.0) //DON'T BEACON UPON INITIAL SETUP. WAIT FOR NEXT INTERVAL.
            agent->beaconTimer.resched(10.0);
        else
            agent->beaconTimer.resched(agent->beaconInterval_);
    }

}
void wuSubBeaconTimer::handle(Event*)
{
    //Only masters send out beacons using this mode and the data rate policy must warrant it
    if ((agent->dataRatePolicy != WU_DATA_RATE_POLICY_BEACONING) &&
        (agent->dataRatePolicy != WU_DATA_RATE_POLICY_SLAVE_BEACONING))
    {
        return;
    }

    if (agent->ifNeedToPrintWu(2)) {
        printf("wu.cc(%d): node_(%d) is handling a sub beacon timer at time %f \n", __LINE__, agent->index, NOW);
    }

    double nextPossibleSendingTime = 0.0005;
    bool haveSentAllSubBeacons = true;

    //For Regular Master Node Beaconing
    if (agent->isAMaster && (agent->dataRatePolicy == WU_DATA_RATE_POLICY_BEACONING))
    {
        for (int i = WU_MAX_RATE_LEVEL; i > 0; i--)
        {
            if (!agent->subBeaconSent[i])
            {
                haveSentAllSubBeacons = false;  //We found one that hasn't been sent

                if (agent->mac_ && (agent->mac_->mode == WU_IDLE))
                {
                    struct hdr_cmn *ch = HDR_CMN(agent->subBeaconPacket[i]);
                    ch->seq_no = agent->get_seq_no();

//                  printf("wu.cc: [%lf] SUBBEACON- Scheduling the transmission of beacon (seq: --, rate: %d) NOW!\n", NOW, i);
                    Scheduler::instance().schedule(agent->target_, agent->subBeaconPacket[i], 0.0);
                    agent->subBeaconSent[i] = true;
                    nextPossibleSendingTime = 0.001;
```

```
                }
                else
                {
//                  printf("wu.cc: [%lf] SUBBEACON- TRIED to schedule beacon %d but failed: MAC isn't ready.\n", NOW, i);
                }
                break;
            }
        }
    }
    //For Slave Node Beaconing
    else if (!agent->isAMaster && (agent->dataRatePolicy == WU_DATA_RATE_POLICY_SLAVE_BEACONING))
    {
        //Make sure we have a ref to mac
        if (!agent->mac_)
        {
            printf("wu.cc(%d): [%lf] node_(%d) ERROR: CANNOT CONTINUE. No Reference to MAC layer in wu.cc.\n",
                __LINE__, NOW, agent->index);
        }
        //Loop through data rate beacons, starting with one above that which we are on.
        int nextPossibleDataRateLevel = agent->mac_->getDataRateLevel() + 1;
        if (nextPossibleDataRateLevel > WU_MAX_RATE_LEVEL)
        {
            //Nothing to do. Already at max data rate.
//          printf("wu.cc(): [%lf] node_(%d) SUBBEACON- No more data rates to try beaconing with\n", NOW, agent->index);
            haveSentAllSubBeacons = true;
        }
        else    //We've still got an acceptable range of beacons to send out
        {
            for (int i = nextPossibleDataRateLevel; i <= WU_MAX_RATE_LEVEL; i++)    //Beacon from Lowest DR to Highest
//          for (int i = WU_MAX_RATE_LEVEL; i >= nextPossibleDataRateLevel; i--)    //Beacon from Highest DR to Lowest
            {
                if (!agent->subBeaconSent[i])
                {
                    haveSentAllSubBeacons = false;  //We found one that hasn't been sent

                    if (agent->mac_ && (agent->mac_->mode == WU_IDLE))
                    {
                        struct hdr_cmn *ch = HDR_CMN(agent->subBeaconPacket[i]);
                        ch->seq_no = agent->get_seq_no();

//                      printf("wu.cc: [%lf] SUBBEACON- Transmitting beacon (seq: %d, rate: %d, rate: %d) NOW!\n",
//                          NOW, ch->seq_no, i, agent->subBeaconPacket[i]->data_rate);
                        Scheduler::instance().schedule(agent->target_, agent->subBeaconPacket[i], 0.0);
                        agent->subBeaconSent[i] = true;
//                      nextPossibleSendingTime = 0.001;
                        nextPossibleSendingTime = agent->getMaxRoundTripTxTime(1.0, agent->getTxDataRateValue(i), false);
//                      printf("\t\tnext possible sending time(1.0, %d): %lf in the future.\n", i, nextPossibleSendingTime);
                    }
                    else
                    {
//                      printf("wu.cc: [%lf] node_(%d) SUBBEACON- TRIED to schedule beacon %d but failed: MAC isn't ready.\n",
//                          NOW, agent->index, i);
                    }
                    break;
                }
                else
                {
//                  printf("wu.cc: [%lf] node_(%d) SUBBEACON- Beacon has already been sent at level %d\n", NOW, agent->index, i);
                }

            }
```

```
        }
    }

    //Reschedule if we still have more sub-beacons to send out!
    if (!haveSentAllSubBeacons)
    {
//      printf("wu.cc: [%lf] node_(%d) SUBBEACON- Rescheduling for time %lf\n", NOW, agent->index, NOW+nextPossibleSendingTime);
        agent->subBeaconTimer.resched(nextPossibleSendingTime);
    }
    else
    {
//      printf("wu.cc: [%lf] node_(%d) SUBBEACON- we've sent all subbeacons!\n", NOW, agent->index);
    }
}

void wuBeaconTimer::expire(Event *) { return; }

void wuSubBeaconTimer::expire(Event *) { return; }

void WU::handle_ack_notification(int senderAddr)
{
  //struct hdr_ip *ih = HDR_IP(p);

  //Let us keep the 4th field as the current frequency.
    if (hasTrace == 1)
        printf("ANALYSIS-TRACE: %d  %d  %ld %d  %d\n", IS_AN_AUTO_ACK, index, index, getChannelNumber(), int(NOW*1000));

    if (isAMaster == 1)
    {
        if (current_mode == MASTER_BIND_WAIT_FOR_AUTO_ACK_AFTER_SENDING_RESPONSE_MODE)
        {
            setChannelCounter(previous_channel_counter);
            changeToDataMode();

            if (ifNeedToPrintWu(1)) {
                printf("wu.cc: I am a master in _wait_for_auto_ack_after_sending_response_mode, Setting mode to DATA_MODE,
                    Channel to %d; return \n", getChannelNumber());
            }
        }
        else if (current_mode == MASTER_CHANNEL_SELECT_MODE)
        {
            current_mode = MASTER_CHANNEL_SELECT_MODE_GOT_AN_ACK;
        }
        else if (current_mode == DATA_MODE)
        {
            if (ifNeedToPrintWu(1)) {
                printf("wu.cc: I am a master in DATA_MODE, simply return \n");
            }
        }

        return;
    }

    if (ifNeedToPrintWu(1)) {
        printf("wu.cc: [%lf] node_(%d) Handling ack notification \n", NOW, index);
//      printf("wu.cc: node_(%d) This is my own ping packet, which was sent to the node \n", index);
        printf("\tCurrent mode is %d. (%d = Bind mode, %d = Data mode) \n", current_mode, BIND_MODE, DATA_MODE);
    }

    if (current_mode == DATA_MODE)
    {
```

```
if (pkt_tx_mode == WAITING_FOR_ACK)
{
    //setPktTxMode(WU_STATUS_IDLE);
    if (ifNeedToPrintWu(1)) {
        printf("wu.cc: [%lf] node_(%d) Was waiting for this DATA ack. Got it, hence return \n", NOW, index);
    }

    if (pktTx_ != 0) {
        Packet *np = pktTx_;
        Packet::free(np);
        pktTx_ = 0; //Reset the packet to NULL
    }

    //Decide how to upgrade the power and or data rate
    if (combinedPolicy != WU_COMBINED_POLICY_NONE)
    {
        //Are we using a link quality mechanism?
        if (combinedPolicy == WU_COMBINED_POLICY_DR_FIRST)
        {
            //Make sure we're using the link quality schemes for both power and data rate
            if (upgradePowerOrDataRate == WU_UPGRADE_DATA_RATE) //Upgrade Data Rate
            {
                powerPolicy = POWER_POLICY_NONE;
                dataRatePolicy = WU_DATA_RATE_POLICY_LINK_QUALITY;
            }
            else    //Upgrade Power
            {
                powerPolicy = POWER_POLICY_INCREMENTAL;
                dataRatePolicy = WU_DATA_RATE_POLICY_NONE;
            }
        }
    }


    //Upgrade power level if we are using a policy
    if (powerPolicy != POWER_POLICY_NONE)
    {
        //Did we just succeed after an upgrade attempt?
        if (attemptingToUpgradePowerLevel)
        {
            attemptingToUpgradePowerLevel = false;
            if (powerLevelUpgradeBackoffMechanism != POWER_LEVEL_UPGRADE_FIXED)
            {
                powerLevelBackoff_ = MIN_POWER_LEVEL_UPGRADE_BACKOFF_VALUE;
            }
            //printf("wu.cc(%d): node_(%d) was successfull after a power upgrade attempt.
            //  Setting powerLevelBackoff_ back to: %d\n", __LINE__, index, powerLevelBackoff_);
        }

        //Increment the number of successful packets at this power level
        successfulPacketsAtCurrentTxPowerLevel++;

        //Decrement power level if we're successful over X times
        if (successfulPacketsAtCurrentTxPowerLevel > powerLevelBackoff_)
        {
            if (mac_)
            {
                //Attempt a power level UPGRADE by decrementing the power level
                attemptingToUpgradePowerLevel = true;
```

```
            //Reset success counter
            successfulPacketsAtCurrentTxPowerLevel = 0;

            int powerLevel = mac_->getPowerLevel();
            if (powerLevel != 0)
            {
                //printf("wu.cc: node_(%d) is upgrading its PA from %d to %d \n", index, powerLevel, powerLevel-1);

                powerLevel--;
                mac_->setPowerLevel(powerLevel);
            }
        }
    }
}

if (dataRatePolicy == WU_DATA_RATE_POLICY_SNR_THRESHOLD)
{
    mac_->updateDataRateSNRThreshold(mac_->getDataRateLevel(), WU_DATA_RATE_THRESHOLD_RELAX);
    decideWhatDataRateToUseUsingSNR();
}

//Upgrade data rate level if we are using a policy
else if (dataRatePolicy == WU_DATA_RATE_POLICY_LINK_QUALITY)
{
    //Did we just succeed after an upgrade attempt?
    if (attemptingToUpgradeDataRate)
    {
        attemptingToUpgradeDataRate = false;
        if (dataRateUpgradeBackoffMechanism != POWER_LEVEL_UPGRADE_FIXED)
        {
            dataRateBackoff_ = MIN_POWER_LEVEL_UPGRADE_BACKOFF_VALUE;
        }
        //printf("wu.cc(%d): node_(%d) was successfull after a rate upgrade attempt.
        //  Setting dataRateBackoff_ back to: %d\n", __LINE__, index, dataRateBackoff_);
    }

    //Increment the number of successful packets at this data rate
    successfulPacketsAtCurrentTxDataRate++;

    //Decrement power level if we're successful over X times
    if (successfulPacketsAtCurrentTxDataRate > dataRateBackoff_)
    {
        if (mac_)
        {
            //Attempt a data rate UPGRADE by incrementing the data rate level
            attemptingToUpgradeDataRate = true;


            //Reset success counter
            successfulPacketsAtCurrentTxDataRate = 0;

            int dataRateLevel = mac_->getDataRateLevel();
            if (dataRateLevel != WU_MAX_RATE_LEVEL)
            {
                //printf("wu.cc: node_(%d) is upgrading its Rate Level from %d to %d \n", index, dataRateLevel, dataRateLevel+1);

                dataRateLevel++;
                mac_->setDataRateLevel(dataRateLevel);
            }
        }
    }
```

```
            }


            setPktTxMode(RECEIVED_ACK);
            return;
        }

        if (ifNeedToPrintWu(1)) {
            printf("wu.cc: i am already bound, hence return \n");
        }

        return;
    }

...

    sendBindRequest(senderAddr);
    setMode(SLAVE_BIND_WAIT_FOR_RESPONSE_AFTER_SENDING_REQUEST_MODE);

  //Packet::free(p);

}

...

void WU::recv(Packet *p, Handler*)
{
    struct hdr_cmn *ch = HDR_CMN(p);
    struct hdr_ip *ih = HDR_IP(p);

    if (ifNeedToPrintWu(1)) {
        printf("wu.cc: [%lf] node_(%d) is receiving a packet here with wu_size as %d \n", NOW, index, ch->wu_size());
    }

    if (ch->ptype() == PT_WU) {
        ih->ttl_ -= 1;
        recvWU(p);
        return;
    }


 /*
  *  Must be a packet I'm originating...
  */
    if ((ih->saddr() == index) && (ch->num_forwards() == 0)) {
        /*
        * Add the IP Header
        */
//     ch->size() += IP_HDR_LEN;
//     ch->wu_size() += WU_NETWORK_HDR_LEN + WU_TRANSPORT_HDR_LEN;
        ch->wu_size() += ch->size();    //Keep the same as the regular size!
       // Added by Parag Dadhania && John Novatnack to handle broadcasting
        if ( (u_int32_t)ih->daddr() != IP_BROADCAST)
            ih->ttl_ = 1;

        if (ifNeedToPrintWu(1)) {
            printf("\tThis packet is being originated at application layer, must route/unicast this \n");
        }

        rt_resolve(p);
    }
```

```
    else {
        drop(p, DROP_RTR_TTL);
        return;
    }
}

void WU::recvWU(Packet *p)
{
    struct hdr_wu *wh = HDR_WU(p);
    struct hdr_ip *ih = HDR_IP(p);

    assert(ih->sport() == RT_PORT);
    assert(ih->dport() == RT_PORT);

 /*
  * Incoming Packets.
  */
    switch(wh->wh_type) {

        case WUTYPE_PING:
            if (ifNeedToPrintWu(1)) {
                printf("wu.cc: [%lf] node_(%d) is receiving a PING  \n", NOW, index);
            }

            recvPing(p);
            break;

        case WUTYPE_BIND_REQUEST:
            if (ifNeedToPrintWu(1)) {
                printf("wu.cc: [%lf] node_(%d) is receiving a BIND REQUEST \n", NOW, index);
            }

            recvBindRequest(p);
            break;

        case WUTYPE_BIND_RESPONSE:
            if (ifNeedToPrintWu(1)) {
                printf("wu.cc: [%lf] node_(%d) is receiving a BIND RESPONSE \n", NOW, index);
            }

            recvBindResponse(p);
            break;

        case WUTYPE_CONNECT_REQUEST:
            if (ifNeedToPrintWu(1)) {
                printf("wu.cc: [%lf] node_(%d) is receiving a CONNECT REQUEST \n", NOW, index);
            }

            recvConnectRequest(p);
            break;

        case WUTYPE_CONNECT_RESPONSE:
            if (ifNeedToPrintWu(1)) {
                printf("wu.cc: [%lf] node_(%d) is receiving a CONNECT RESPONSE\n", NOW, index);
            }

            recvConnectResponse(p);
            break;

        case WUTYPE_DATA_RATE_BEACON:
            if (ifNeedToPrintWu(1)) {
```

```
                printf("wu.cc: [%lf] node_(%d) is receiving a DATA RATE BEACON\n", NOW, index);
            }

            recvDataRateBeacon(p);
            break;

        default:
            if (ifNeedToPrintWu(1)) {
                printf("wu.cc: [%lf] node_(%d) is receiving a DEFAULT packet here \n", index);
                fprintf(stderr, "Invalid WU type (%x)\n", wh->wh_type);
            }

        exit(1);
    }
}

/** Gets the retransmit TimeOut value for packet */
double WU::getRetransmitTO(Packet *p, bool addRandomness)
{
    double t_;
    struct hdr_cmn *ch = HDR_CMN(p);

    double drt = getTxDataRateValue(mac_->getDataRateLevel());
    int psz = ch->wu_size();

    t_ = getMaxRoundTripTxTime(psz, drt, addRandomness);

    return t_;
}

/* Calculates the maximum round-trip tx time for packet of size "psz" bytes
 *        at rate "drt" bps, includes ack time and small random amount of increase for propigation delay.
 */
double WU::getMaxRoundTripTxTime(double psz, double drt, bool addRandomness)
{
    double t_;

    //Add time for which packet would be transmitted
    t_ = (8 * (psz + CRC_SEED_LENGTH + LENGTH_FIELD))/(drt);
    t_ += (8 * (NUMBER_STARTER_FRAMES * START_FRAME_SIZE))/(mac_->basicRate_);
    //Add time for which ack would be transmitted
    t_ += (8 * (MAC_WU_ACK_SIZE + CRC_SEED_LENGTH + LENGTH_FIELD))/(drt);
    t_ += (8 * (NUMBER_STARTER_FRAMES * START_FRAME_SIZE))/(mac_->basicRate_);
    //Adding a small value of ACK time to take care of propagation delay.
    //This is not accurate, but would work, since the min time for the next
    //ack to come would be ack_time, and here the field is 16th time smaller
    //hence, it should be okay.
//  printf("wu.cc(%d): node_(%d) Round-trip time prior to prop delay: %lf\n", __LINE__, index, t_);
    double propigationDelay = (8 * MAC_WU_ACK_SIZE )/(16 * drt);
//  printf("wu.cc(%d): node_(%d) Adding prop delay of: %lf\n", __LINE__, index, propigationDelay);
    t_ += propigationDelay;

    if (addRandomness)
    {
        //Add a random amount of time for uniqueness
//      double randomness = ((Random::random()) % 9) / 10000.0f;
        double packetLengthRandomness = ((Random::random() % 5)) * t_;  //Add  0 - 4 times the current time
        //Add small amount
        double smallRandomness = ((Random::random() % 10000) * 0.0000001);  //A small number of microseconds (from 0 to 1ms)

//      printf("wu.cc(%d): node_(%d) Adding random time for pkt lngth and uniqueness (%lf + %lf) = %lf\n",
```

```
//          __LINE__, index, packetLengthRandomness, smallRandomness, packetLengthRandomness+ smallRandomness);
        t_ += packetLengthRandomness + smallRandomness;
    }

    if (mac_->hasCarrierSensing) {
            //Add possible back_offs and carrier_sensing_timings as well.
        double carrierSenseBackoffTiming = MAC_MAX_BACK_OFF_LIMIT * (MAC_WU_CARRIER_SENSE_TIMEOUT + MAC_WU_MAX_BACK_OFF);

//      printf("wu.cc(%d): node_(%d) Adding carrier-sense backoff timing of %lf\n", __LINE__, index, carrierSenseBackoffTiming);
        t_ += carrierSenseBackoffTiming;
    }
    else {
        if (ifNeedToPrintWu(1)) {
            printf("wu.cc: since MAC has NO carrier sensing, let us NOT add backoff timings \n");
        }
    }

    if (ifNeedToPrintWu(1)) {
        printf("wu.cc: node_(%d) The total maximum round-trip TX time for this packet (at %d kbps) is %.09f \n", index, (int)drt, t_);
    }

    return t_;
}


/*
   Packet Transmission Routines
*/
void WU::retransmit()
{
    if (pkt_tx_mode == RECEIVED_ACK)
    {
        if (ifNeedToPrintWu(1)) {
            printf("wu.cc: [%lf] node_(%d) has received an Ack. No need to retransmit the data \n", NOW, index);
        }

        if (pktTx_ != 0)
        {
            Packet *np = pktTx_;
            Packet::free(np);
            pktTx_ = 0;
        }

        setPktTxMode(WU_STATUS_IDLE);
        return;
    }

    if (ifNeedToPrintWu(1))
    {
        printState();
    }

    if (current_mode == SLAVE_CHANNEL_SEARCH_AFTER_SUCCESSFUL_BIND)
    {
        if (ifNeedToPrintWu(1))
        {
            printf("wu.cc: node_(%d) is in channel_ search mode now. Do not retransmit. Ignore the timer.
                Packet would be sent once channel search is done \n", index);
        }

        num_times_retx = 0;
```

```
    return;
}

struct hdr_cmn *ch = HDR_CMN(pktTx_);

if (ifNeedToPrintWu(1))
{
    printf("wu.cc: node_(%d) RETRANSMITTING - number of times retransmitted is %d and the seq_no is %d \n",
        index, num_times_retx, ch->seq_no);
}

num_times_retx++;
if (num_times_retx > MAXIMUM_PKT_RETX)
{
    if (ifNeedToPrintWu(1))
    {
        printf("wu.cc(%d): [%lf] node_(%d) PROBLEM, did not receive ACK after sending %d retries. Start Channel Search now \n",
            __LINE__, NOW, index, num_times_retx-1);
    }

    Packet *np = pktTx_;
    Packet::free(np);
    pktTx_ = 0;

    //setPktTxMode(WU_STATUS_IDLE);
    startChannelSearch();

    return;
}

//COMBINED ADJUSTMENT - What should we adjust? Power or Data Rate?
bool shouldRelaxPower = false;
bool shouldRelaxDataRate = false;
if (combinedPolicy != WU_COMBINED_POLICY_NONE)
{
    //Adjust power first, then Data Rate (relaxing performes the reverse of upgrading)
    if (combinedPolicy == WU_COMBINED_POLICY_DR_FIRST)
    {
        int powerLevel = mac_->getPowerLevel();

        //If not at max PA, relax power!
        if (powerLevel != WU_MAX_PA)
            shouldRelaxPower = true;
        else
            shouldRelaxDataRate = true;
    }
}


//Adjust power level if we are using a policy
if ((powerPolicy != POWER_POLICY_NONE) || shouldRelaxPower)
{
    //Did we just fail after an upgrade attempt?
    if (attemptingToUpgradePowerLevel)
    {
        upgradePowerOrDataRate = WU_UPGRADE_DATA_RATE;  //Upgrade Data Rate Next
        //We failed, so backoff (if at all possible
        //Have we backed off as far as we can go?
        if (powerLevelBackoff_ < MAX_POWER_LEVEL_UPGRADE_BACKOFF_VALUE)
        {
            if (powerLevelUpgradeBackoffMechanism == POWER_LEVEL_UPGRADE_LINEAR)
```

```
            {
                powerLevelBackoff_ += POWER_LEVEL_UPGRADE_BACKOFF_INCREMENT;
            }
            else if (powerLevelUpgradeBackoffMechanism == POWER_LEVEL_UPGRADE_EXPONENTIAL)
            {
                powerLevelBackoff_ *= 2;
            }
        }

        //printf("wu.cc(%d): node_(%d) failed after a power upgrade attempt. Setting powerLevelBackoff_ to: %d\n",
        //  __LINE__, index, powerLevelBackoff_);
    }

    if (num_times_retx > powerLevelTolerance_)  //How soon should we adjust the power level?
    {
        if (mac_)
        {
            //Reset success counter
            successfulPacketsAtCurrentTxPowerLevel = 0;

            int powerLevel = mac_->getPowerLevel();

            if (powerLevel != WU_MAX_PA)
            {
                //Increment power level -   Or Aggressive after an upgrade attempt (acts like incremental)
                if ((powerPolicy == POWER_POLICY_INCREMENTAL) || (attemptingToUpgradePowerLevel) || shouldRelaxPower)
                {
                    //printf("wu.cc: node_(%d) is incrementing its PA from %d to %d \n", index, powerLevel, powerLevel+1);
                    powerLevel++;
                }
                else if (powerPolicy == POWER_POLICY_AGGRESSIVE)
                {
                    //printf("wu.cc: node_(%d) is aggressively setting its PA from %d to %d \n", index, powerLevel, WU_MAX_PA);
                    powerLevel = WU_MAX_PA;
                }
                else if (powerPolicy == POWER_POLICY_INCREMENTAL_PLUS_2)
                {
                    //printf("wu.cc: node_(%d) is incrementing (by 2) its PA from %d to %d \n", index, powerLevel, WU_MAX_PA);
                    powerLevel += 2;
                    //Make sure we haven't gone over WU_MAX_PA
                    if (powerLevel > WU_MAX_PA)
                        powerLevel = WU_MAX_PA;
                }

                mac_->setPowerLevel(powerLevel);

                //Reset backoff value if necessary
                if ((attemptingToUpgradePowerLevel == false) &&
                        (powerLevelUpgradeBackoffMechanism != POWER_LEVEL_UPGRADE_FIXED))
                {
                    powerLevelBackoff_ = MIN_POWER_LEVEL_UPGRADE_BACKOFF_VALUE;
                }
            }
        }
    }

    if (attemptingToUpgradePowerLevel)
        attemptingToUpgradePowerLevel = false;
}

//************ END OF POWER POLICY ADJUSTMENT ************************************************************************************/
```

```
    //Adjust rate if we are using a policy
    if ((dataRatePolicy != WU_DATA_RATE_POLICY_NONE) || shouldRelaxDataRate)
    {
        //Did we just fail after an upgrade attempt?
        if (attemptingToUpgradeDataRate)
        {
            upgradePowerOrDataRate = WU_UPGRADE_POWER;  //Upgrade Power Next
            //We failed, so backoff (if at all possible
            //Have we backed off as far as we can go?
            if (dataRateBackoff_ < MAX_POWER_LEVEL_UPGRADE_BACKOFF_VALUE)
            {
                if (dataRateUpgradeBackoffMechanism == POWER_LEVEL_UPGRADE_LINEAR)
                {
                    dataRateBackoff_ += POWER_LEVEL_UPGRADE_BACKOFF_INCREMENT;
                }
                else if (dataRateUpgradeBackoffMechanism == POWER_LEVEL_UPGRADE_EXPONENTIAL)
                {
                    dataRateBackoff_ *= 2;
                }
            }

            //printf("wu.cc(%d): node_(%d) failed after a data rate upgrade attempt. Setting dataRateBackoff_ to: %d\n",
            //  __LINE__, index, dataRateBackoff_);
        }

        if (num_times_retx > dataRateTolerance_)    //How soon should we adjust the power level?
        {
            if (mac_)
            {
                //Reset success counter
                successfulPacketsAtCurrentTxDataRate = 0;

                int dataRateLevel = mac_->getDataRateLevel();

                if (dataRateLevel != 0)
                {
                    //Decrement data rate level -   Or Aggressive after an upgrade attempt (acts like incremental)
                    //if (ifNeedToPrintWu(1))
                    printf("wu.cc: [%lf] node_(%d) is decrementing its Data Rate from %d to %d \n",
                        NOW, index, dataRateLevel, dataRateLevel-1);

                    dataRateLevel--;

                    mac_->setDataRateLevel(dataRateLevel);

                    //Reset backoff value if necessary
                    if ((attemptingToUpgradeDataRate == false) &&
                            (dataRateUpgradeBackoffMechanism != POWER_LEVEL_UPGRADE_FIXED))
                    {
                        dataRateBackoff_ = MIN_POWER_LEVEL_UPGRADE_BACKOFF_VALUE;
                    }
                }
            }
        }

        if (attemptingToUpgradeDataRate)
            attemptingToUpgradeDataRate = false;
    }

// ********** END OF DATA RATE POLICY ADJUSTMENT ********************************************************************/
```

```
    double t_retx_ = getRetransmitTO(pktTx_, true);
    if (ifNeedToPrintWu(1))
    {
        printf("wu.cc: Retransmitting after timeout value is %0.9f \n", t_retx_);
        //printf("%s: retransmitting()\n", __FUNCTION__);
    }

    if (ifNeedToPrintWu(1))
    {
        printf("\t\t\tFrom retransmit at wu.cc \n");
        pktTx_->print_header();
        pktTx_->print_cmn_header();
    }

    retxtimer.resched(t_retx_); //this is based on pkt size.

    if (hasTrace == 1)
        printf("ANALYSIS-TRACE: %d  %d  %ld %d  %d\n", IS_A_RETX, index, index, getChannelNumber(), int(NOW*1000));

    Scheduler::instance().schedule(target_, pktTx_->copy(), 0); //Don't use friggin copy method, try new Packet(pktTx_)
}


void WU::send(Packet *p, double delay)
{
    struct hdr_cmn *ch = HDR_CMN(p);
    struct hdr_ip *ih = HDR_IP(p);

    if (hasTrace == 1)
        printf("ANALYSIS-TRACE: %d  %d  %ld %d  %d\n", IS_A_SOURCE, index, index, getChannelNumber(), int(NOW*1000));

    if (ifNeedToPrintWu(1)) {
        printf("wu.cc: [%lf] node_(%d) SENDING () a packet with seq_no as %d \n", NOW, index, ch->seq_no);
    }

    if (ch->ptype() != PT_WU && ch->direction() == hdr_cmn::UP &&
        ((u_int32_t)ih->daddr() == IP_BROADCAST)
        || ((u_int32_t)ih->daddr() == here_.addr_)) {
            dmux_->recv(p,0);
            return;
    }

    ch->next_hop_ = ih->daddr();
    ch->addr_type() = NS_AF_WU;
    ch->direction() = hdr_cmn::DOWN;       //important: change the packet's direction

    if (ifNeedToPrintWu(1)) {
        printf("wu.cc: [%lf] node_(%d) We have the RTF_UP and the next hop is %d \n", NOW, index, ch->next_hop_);
    }

    p->freq1 = p->freq2 = getChannelNumber();
    p->pn_code = pn_code;
    if (mac_)
        p->data_rate = mac_->getDataRateLevel();
    else
        p->data_rate = 0;

    if ((pkt_tx_mode != WU_STATUS_IDLE) && (pkt_tx_mode != RECEIVED_ACK)) {
        if (ifNeedToPrintWu(1)) {
```

```
            printf("wu.cc: [%lf] node_(%d) The current mode is data mode, but tx mode is not idle,
                hence drop the packet and return \n", NOW, index);
        }

        return;
    }

    pktTx_ = p->copy(); //new Packet(p);


 //When you get an ack, you should see if there are more packets that need to be sent.
    double t_retx_ = getRetransmitTO(p, false);

    if (ifNeedToPrintWu(1)) {
        printf("wu.cc: node_(%d) The timeout value is %0.9f \n", index, t_retx_);
    }

    retxtimer.resched(t_retx_); //This is based on pkt size + maximum number of retransmission at the MAC layer.


    num_times_retx = 0;
    setPktTxMode(WAITING_FOR_ACK);

    if (delay > 0.0) {
        Scheduler::instance().schedule(target_, p, delay);
    }
    else {
        //Not a broadcast packet, no delay, send immediately
        Scheduler::instance().schedule(target_, p, 0.0);
    }

}

void WU::sendBeaconResponse(nsaddr_t ipdst)
{
    Packet *p = Packet::alloc();
    struct hdr_cmn *ch = HDR_CMN(p);
    struct hdr_ip *ih = HDR_IP(p);
    struct hdr_wu_beacon *wb = HDR_WU_BEACON(p);

    p->freq1 = p->freq2 = getChannelNumber();
    p->pn_code = pn_code;
    ch->xmit_gotAnAck_ = wu_rt_gotAnAck_callback;
    ch->xmit_failure_data_ = (void*) this;

    wb->rp_type = WUTYPE_DATA_RATE_BEACON;
    wb->rp_dst = ipdst;
    wb->rp_src = index;
    wb->pn_code = PN1;
    wb->network_id = 0;
    wb->data_rate = beaconResponseDataRate;

    ch->ptype() = PT_WU;
    ch->size() = IP_HDR_LEN + wb->size();
    ch->wu_size() = CRC_SEED_LENGTH;        //SIZE OF ONE FOR NOW
    ch->iface() = -2;
    ch->error() = 0;
    ch->addr_type() = NS_AF_WU;
    ch->next_hop_ = ipdst;
    ch->prev_hop_ = index;
    ch->direction() = hdr_cmn::DOWN;
```

```
    ch->seq_no = get_seq_no();

    ih->saddr() = index;
    ih->daddr() = ipdst;
    ih->sport() = RT_PORT;
    ih->dport() = RT_PORT;
    ih->ttl_ = 10;

    if (ifNeedToPrintWu(1)) {
        printf("wu.cc(%d): [%lf] node_(%d) is sending a BEACON RESPONSE for data rate %d to node_(%d)
            on channel number [%d-%d] with seq_no as %d \n",
            __LINE__, NOW, index, wb->data_rate, ipdst, p->freq1, p->freq2,  ch->seq_no);
    }

    Scheduler::instance().schedule(target_, p, 0.0);
}

void WU::recvDataRateBeacon(Packet *p)
{
    struct hdr_cmn *ch = HDR_CMN(p);
    struct hdr_ip *ih = HDR_IP(p);
    struct hdr_wu_beacon *wb = HDR_WU_BEACON(p);

    //Master needs to ack back the successful reception of this beacon (that originated from a slave)
    if (isAMaster)
    {
        //Acknowledge beacon
        beaconResponseDataRate = wb->data_rate;     //Internal variable to know what to send the beacon response out at
        waitForAckToFinishAtRadioLayer(wb->rp_src, PURPOSE_SEND_BEACON_RESPONSE);
    }
    else    //Slave Nodes get Updated
    {
        if (mac_)
        {
            int newDR = wb->data_rate;
            int oldDR = mac_->getDataRateLevel();

//          printf("wu.cc(%d): [%lf] node_(%d) Rec'd DR Beacon - wb->data_rate is %d\n", __LINE__, NOW, index, newDR);

            //Only Upgrade
            if (newDR > oldDR)
            {
                mac_->setDataRateLevel(newDR);

                if (ifNeedToPrintWu(1)) {
                    printf("wu.cc(%d): [%lf] node_(%d) Rec'd DR Beacon - Changing data_rate to %lf\n",
                        __LINE__, NOW, index, getTxDataRateValue(newDR));
                }
            }
        }
    }

    Packet::free(p);
}

void WU::prepareAndSendDataRateSubBeacons(nsaddr_t ipdst)
{
    Packet *p = Packet::alloc();
    struct hdr_cmn *ch = HDR_CMN(p);
    struct hdr_ip *ih = HDR_IP(p);
    struct hdr_wu_beacon *wb = HDR_WU_BEACON(p);
```

```
    p->freq1 = p->freq2 = getChannelNumber();
    p->pn_code = pn_code;
    ch->xmit_gotAnAck_ = wu_rt_gotAnAck_callback;
    ch->xmit_failure_data_ = (void*) this;

    wb->rp_type = WUTYPE_DATA_RATE_BEACON;
    wb->rp_dst = ipdst;
    wb->rp_src = index;
    wb->pn_code = PN1;
    wb->network_id = 0;
    wb->data_rate = -1; // WILL BE CHANGED

    ch->ptype() = PT_WU;
    ch->size() = IP_HDR_LEN + wb->size();
    ch->wu_size() = CRC_SEED_LENGTH;        //SIZE OF ONE FOR NOW
    ch->iface() = -2;
    ch->error() = 0;
    ch->addr_type() = NS_AF_WU;
    ch->next_hop_ = ipdst;
    ch->prev_hop_ = index;
    ch->direction() = hdr_cmn::DOWN;
    ch->seq_no = -1;    // WILL be changed before it is sent!!!

    ih->saddr() = index;
    ih->daddr() = ipdst;
    ih->sport() = RT_PORT;
    ih->dport() = RT_PORT;
    ih->ttl_ = 10;

    int beaconLevel = WU_MAX_RATE_LEVEL;    //Start with the highest data rate

    double delay = 0.0;

    //Set up a beacon for each data rate except the last one (useless) (max - 1)
    while (beaconLevel > 0)
    {
        Packet *newPacket = p->copy(); //new Packet(p);
        struct hdr_wu_beacon *wb_new = HDR_WU_BEACON(newPacket);
        wb_new->data_rate = beaconLevel;    //OLD WAY USING CUSTOM BEACON HDR
        newPacket->data_rate = beaconLevel;

//      printf("wu.cc(%d): BEACON - node_(%d) is temporarily setting its data rate to level %d for sending beacon.\n",
//              __LINE__, index, beaconLevel);

        if (ifNeedToPrintWu(1)) {
            printf("wu.cc: node_(%d) is scheduling the sending of a DATA RATE BEACON (#%d) on channel number [%d-%d]
                at time %f with seq_no as --\n", index, beaconLevel, newPacket->freq1, newPacket->freq2, NOW);
        }

        //Save the packet for transfer later
        subBeaconSent[beaconLevel] = false;
        subBeaconPacket[beaconLevel] = newPacket;

        beaconLevel--;
    }

    subBeaconTimer.resched(0.0);    //Call the subBeacon Timer to send off these beacons!
}

/** This function gets the appropriate power value for use in phy-layer calculations when determining
```

```
 * the propagation range. Called from wireless-phy.cc.
 */
double WU::getTxPowerValueFromMac()
{
    int powerLevel = -1;
    double powerValue = 0.0;    //To be returned

    //Query the mac layer to find out what power level it is currently operating at
    if (mac_)
    {
        powerLevel = mac_->getPowerLevel();
    }

    //Get the mWatt value
    powerValue = getTxPowerValue(powerLevel);

    //Adjust for more realistic conditions (Lower by two orders of magnitude)
    //Since this is purely for range issues at the phy layer, this is the same as changing the rx thresholds
    //I chose to keep the thresholds the same, and modify this Pt_ value. -BarlowJ
    powerValue = powerValue / 100.0f;



    //Adjust for DATA RATE
    powerValue = powerValue / pow(10, mac_->getDataRateLevel());    //Each data rate reduces the power level by an order of magnitude

//  printf("wu.cc: getTxPowerValueFromMac() - node_(%d) Power Level is %d; returning powerValue %e\n",
//      index, powerLevel, powerValue);

    return powerValue;
}

/**  This function returns the appropriate power value in mWatts for use in energy calculations.
 */
double WU::getTxPowerValue(int powerLevel)
{
    double powerValue = 0.0;    //To be returned

    //Just as a reference: pow(10, 2.45) * 1e-3;         // 24.5 dbm, ~ 281.8mw
    switch (powerLevel)
    {
        case (0) :
                powerValue = pow(10, -3.00); // .001 (-30dBm)
            break;
        case (1) :
                powerValue = pow(10, -2.50); // .0031623 (-25dBm)
            break;
        case (2) :
                powerValue = pow(10, -2.00); // .01 (-20dBm)
            break;
        case (3) :
                powerValue = pow(10, -1.50); // .031623 (-15dBm)
            break;
        case (4) :
                powerValue = pow(10, -1.00); // .1 (-10dBm)
            break;
        case (5) :
                powerValue = pow(10, -0.50); // .31623 (-5dBm)
            break;
        case (6) :
                powerValue = pow(10, 0.00); // 1.00 (0dBm)
            break;
```

```
        case (7) :
                powerValue = pow(10, 0.40); // 2.51189 (4dBm)
            break;
        default :
            powerValue = 0.00;
    }

    return powerValue;
}
/**  This function returns the appropriate power value in mWatts for use in energy calculations.
 */
double WU::getTxDataRateValue(int dataRateLevel)
{
    double dataRate = 0.0;  //To be returned

    switch (dataRateLevel)
    {
        case (0) :
                dataRate = 31250.0;
            break;
        case (1) :
                dataRate = 62500.0;
            break;
        case (2) :
                dataRate = 125000.0;
            break;
        case (3) :
                dataRate = 250000.0;
            break;
        case (4) :
                dataRate = 1000000.0;
            break;
        default :
            dataRate = 0.00;
    }

//  printf("wu.cc(%d): Node(%d) data rate level is %d, returning value of: %f\n", __LINE__, index, dataRateLevel, dataRate);
    return dataRate;
}
double WU::getMACEnergyUsed()
{
    //Query the mac layer to find out what power level it is currently operating at
    if (mac_)
    {
        return mac_->debugTxEnergy + mac_->debugRxEnergy;
    }

    return -1;
}

void WU::decideWhatDataRateToUseUsingSNR()
{
    //What data rate can we use based upon this reading?
    int highestDataRate = mac_->getHighestDataRateBasedOnSNR(mac_->getLastSNR());
    int currentDataRate = mac_->getDataRateLevel();
    printf("mac-wu.cc(%d): [%lf] node_(%d) SNR of (%lf) can use a data rate of %d.\n",
        __LINE__, NOW, addr(), mac_->getLastSNR(), highestDataRate);
    if (highestDataRate > currentDataRate)
    {
        printf("mac-wu.cc: [%lf] node_(%d) is upgrading its data rate level to %d\n", NOW, addr(), highestDataRate);
        mac_->setDataRateLevel(highestDataRate);
```

```
    }
}
```

A.4 WirelessUSB MAC Layer

A.4.1 mac/mac-wu.h

```
/* -*-  Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*-
Write Copyright (c) info here
@Manoj Pandey, Cypress Semiconductor.
@Jeff Barlow, Cypress Semiconductor.
 */

...

/* ======================================================================
   Frame Formats
   ====================================================================== */

...

#define WU_RADIUS   100.0   //All nodes outside of this radius are not considered for collisions. Set high!
#define POWER_WU_RX 2.52    //mWatts -- This value is slightly higher than the highest mWatt for PA7

...

//The network layer sets a timeout based on this value.
#define MAC_WU_CARRIER_SENSE_TIMEOUT   0.000005 //50 us
#define MAC_WU_MAX_BACK_OFF     0.001224            //5 32kbps packets
#define MAC_WU_MAX_BACK_OFF     0.000600        //600 us
#define MAC_MAX_BACK_OFF_LIMIT  10              //300 us

...

/* ======================================================================
   The actual WirelessUSB (WU) MAC class.
   ====================================================================== */
class Mac_WU : public Mac {

...

public:
    Mac_WU();
    void        recv(Packet *p, Handler *h);
    void handle_tx_rx_timer();
...
    inline int getPowerLevel() { return powerLevel_; }
    inline int getDataRateLevel() { return dataRateLevel_; }
    inline void setPowerLevel(int newPA) { powerLevel_ = newPA; }
    void setDataRateLevel(int newDataRate);
    void printDataRateSNRThresholds();

    double txtime(double psz, double drt);
    double txtime(double psz);  //Helper function that wraps above function so you don't have to specify the rate
    void updateDataRateSNRThreshold(int dataRateLevel, int relaxOrConstrain);
    int getHighestDataRateBasedOnSNR(double snrValue);
    inline double getLastSNR() { return lastSNR; };
...
```

```
    int need_to_send_an_ack_to;
    int ack_data_rate_level;
...
    Packet *pktRx_;
    WU *networkLayer;
    double dataRate_;
    double basicRate_;

    double debugTxEnergy;
    double debugRxEnergy;

private:
    int     command(int argc, const char*const* argv);

    /*
     * Packet Transmission Functions.
     */
    void    sendACK(int dst, int ackDataRatelLevel);

    void    send(Packet *p, Handler *h);
    void    doCarrierSensingBeforeSendingDATA(Packet *p);
    void    sendDATAasCarrierSensingIsSuccessful(Packet *p);

    /*
     * Packet Reception Functions.
     */
    void    recvACK();
    void    recvDATA();

    void        collision(Packet *p);

...

protected:

...

    int powerLevel_;        //For power adjustment (WUSB has 8 different PA levels, 0 - 7)
    int dataRateLevel_;     //For power adjustment (WUSB has 5 different Data Rate levels, 0 - 4)
    double dataRateSNRThresholds[4+1];  //WU_MAX_RATE_LEVEL <---  SHOULD GO HERE!
    double lastSNR;         //The last or latest Signal To Noise Ratio Reading
};
```

A.4.2 mac/mac-wu.cc

```
/* -*-  Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*-
@Manoj Pandey, Cypress Semiconductor.
@Jeff Barlow, Cypress Semiconductor.
*/

...

void tx_rx_timer_class::expire(Event *) {
    return;
}

void tx_rx_timer_class::handle(Event *) {
```

```
    agent->handle_tx_rx_timer();
}

void Mac_WU::handle_tx_rx_timer() {

    if (ifNeedToPrint(2))
        printf("mac-wu.cc: node_(%d) is handling a tx_rx timer at %0.9f and mode %d \n",
                addr(), NOW, mode);

    //Set mode to idle now!


    if (mode == WU_TX) {
        if (ifNeedToPrint(2))
            printf("mac-wu.cc: node_(%d) has finished transmitting the packet at time %0.9f with pkt_rx_ as %d \n",
                addr(), NOW, pktRx_);

        setMode(WU_IDLE);
        return;

    }
    else if (mode == WU_RX) {
        if (ifNeedToPrint(2))
            printf("mac-wu.cc: node_(%d) has finished receiving the packet at time %0.9f \n", addr(), NOW);


        setMode(WU_IDLE);

        if (pktRx_->isCorrupt == 1)
        {
            if (ifNeedToPrint(2))
                printf("mac-wu.cc: node_(%d) PROBLEM, sorry the DATA received is corrupted. Simply return \n", addr());

            if (pktRx_ != 0)
            {
//            if (ifNeedToPrint(2))
//              printf("mac-wu.cc: loc4 node_(%d) is freeing the message here. pkt is corrput \n", addr());
                Packet *np = pktRx_;
                Packet::free(np);
                pktRx_ = 0;
            }

            return;
        }

        struct hdr_cmn *ch = HDR_CMN(pktRx_);
        hdr_mac_wu *mh = HDR_MAC_WU(pktRx_);
        u_int32_t src = ETHER_ADDR(mh->dh_ta);

        if (seq_no_counter[src] >= ch->seq_no)
        {
            if (ifNeedToPrint(2))
            {
                printf("mac-wu.cc: [%lf] IS A DUPLICATE from %d, seq: %d as my counter is at %d. Packet is out of order or
                    has already been received.\n", NOW, src, ch->seq_no, seq_no_counter[src]);
            }

            //if (pktRx_ != 0) {
            //printf("mac-wu.cc: loc5 node_(%d) is freeing the message here \n", addr());
            //Packet *np = pktRx_;
            //free(np);
```

```
            //pktRx_ = 0;
            //}

            return;
        }

        seq_no_counter[src] = ch->seq_no;


        Packet *newP = pktRx_->copy();  //new Packet(pktRx_);

        struct hdr_cmn *chNew = HDR_CMN(newP);
        chNew->ptype() = PT_WU;
        chNew->energyForTrace = debugTxEnergy + debugRxEnergy;

        uptarget_->recv(newP, this);     //this calls recv() of the next layer up

        backoff_counter = 0;

        //if (pktRx_ != 0) {
//        if (ifNeedToPrint(2))
//          printf("mac-wu.cc: loc6 node_(%d) is freeing the message here. pkt has been received \n", addr());

        Packet *np = pktRx_;
        Packet::free(np);
        pktRx_ = 0;
        //}

        if (need_to_send_ack == 1)
        {
//        if (ifNeedToPrint(2))
//          printf("mac-wu.cc: [%lf] node_(%d) is sending an ack to node_(%d) \n", NOW, addr(), need_to_send_an_ack_to);

            sendACK(need_to_send_an_ack_to, ack_data_rate_level);
        }
        else{
            if (ifNeedToPrint(2))
                printf("mac-wu.cc: NEED_TO_SEND_ACK flag is not set. NO need to send Ack. Return \n");
        }
    }
    else
    {
//      printf("mac-wu.cc: node_(%d) WHY ARE WE HERE with mode %d \n", addr(), mode);
    }

}



/* ======================================================================
   Mac Class Functions
   ====================================================================== */
//Constructor
Mac_WU::Mac_WU() :
    Mac(), phymib_(this), tx_rx_timer(this), ack_timer(this), cs_timer(this), bo_timer(this), debug_timer(this) {

    //dataPacket_ = 0;
    et_ = new EventTrace();

    //cache_ = 0;
    //cache_node_count_ = 0;
```

```
    Tcl& tcl = Tcl::instance();
    tcl.evalf("Mac/MAC_WU set basicRate_");
    if (strcmp(tcl.result(), "0") != 0)
        bind_bw("basicRate_", &basicRate_);
    else
        basicRate_ = bandwidth_;

    tcl.evalf("Mac/MAC_WU set dataRate_");
    if (strcmp(tcl.result(), "0") != 0)
        bind_bw("dataRate_", &dataRate_);
    else
        dataRate_ = bandwidth_;

    powerLevel_ = 7;    //set by default to +4dBm (also, so the master is always at 7. Slaves can decrement.)
    dataRateLevel_ = 0; //set by default to 31.25kbps (also, so the master is always at 0. Slaves can increment it.)

    EOTtarget_ = 0;
    bss_id_ = IBSS_ID;
    has_bt_ = 0;
    need_to_send_ack = 1;
    hasCarrierSensing = 1;

    hasDebug = 0;
    debugStartTime = 0.0;
    timeCounter = 0;
    debugTxEnergy = 0.0;
    debugRxEnergy = 0.0;


    for (int i=0; i < MAX_NODES_ALLOWED; i++)
        seq_no_counter[i] = -1000; //Set it to -ve value so that a pkt with

//    printf("mac-wu.cc: dataRate is %f and basicRate is %f \n", dataRate_, basicRate_);
    setMode(WU_IDLE);

    need_to_send_an_ack_to = 0;
    ack_data_rate_level = 0;

    debug_timer.purpose = PURPOSE_CHECK_MAC_STATE;
//    debug_timer.resched(21.006505);

    //Initialize Data Rate SNR Thresholds
/*  //FOR RSSI INDEX THRESHOLDS
    dataRateSNRThresholds[0] = 0.0;
    dataRateSNRThresholds[1] = 10.0;
    dataRateSNRThresholds[2] = 13.0;
    dataRateSNRThresholds[3] = 17.0;
    dataRateSNRThresholds[4] = 21.0;
*/
    //For DBm Thresholds
    dataRateSNRThresholds[0] = 100.0;
    dataRateSNRThresholds[1] = 0.6437;
    dataRateSNRThresholds[2] = 0.543;
    dataRateSNRThresholds[3] = 0.444;
    dataRateSNRThresholds[4] = 0.343;

//  printDataRateSNRThresholds();
    lastSNR = -1;
}
```

```
int Mac_WU::command(int argc, const char*const* argv)
{
...
        else if (strcmp(argv[1], "powerLevel_") == 0) {
            powerLevel_ = atoi(argv[2]);

            printf("wu.cc: node_(%d) The power level is set to %d \n", addr(), powerLevel_);
            return TCL_OK;
        }
        else if (strcmp(argv[1], "dataRateLevel_") == 0) {
            dataRateLevel_ = atoi(argv[2]);

            printf("wu.cc: node_(%d) The data rate level is set to %d \n", addr(), dataRateLevel_);
            return TCL_OK;
        }
    }
...
    return Mac::command(argc, argv);
}

/* ======================================================================
   Misc Routines
   ====================================================================== */

/** All ACK's are sent back to transmitting node at the transmitting node's data rate. Therefore, pass in data rate level */
void Mac_WU::sendACK(int dst, int ackDataRateLevel)
{
    Packet *p = Packet::alloc();
    hdr_cmn* ch = HDR_CMN(p);
    int myAddr = addr();
    struct wu_ack_frame *af = (struct wu_ack_frame*)p->access(hdr_mac::offset_);

    if (networkLayer->hasTrace)
        printf("ANALYSIS-TRACE: %d  %d  %ld %d  %d\n", IS_AN_AUTO_ACK, addr(),
            addr(), freq1, int(NOW*1000));

    if (ifNeedToPrint(1))
        printf("mac-wu.cc: [%lf] node_(%d) sending ACK to node %d at data rate %d with pktRx_ as %d \n",
                NOW, addr(), dst, ackDataRateLevel, pktRx_);

    ch->uid() = 0;
    ch->ptype() = PT_MAC;
    ch->size() = phymib_.getACKlen();
    ch->wu_size() = MAC_WU_ACK_SIZE;
    ch->iface() = -2;
    ch->error() = 0;

    bzero(af, MAC_HDR_LEN);

    af->af_fc.fc_protocol_version = MAC_ProtocolVersion;
    af->af_fc.fc_type = MAC_Type_Control;
    af->af_fc.fc_subtype = MAC_Subtype_ACK;

    STORE4BYTE(&dst, (af->af_ra));
    STORE4BYTE(&myAddr, (af->af_ta));

    //Make sure ack data rate level is a valid level
    if ((ackDataRateLevel < 0) || (ackDataRateLevel > WU_MAX_RATE_LEVEL))
        ackDataRateLevel = 0;
```

```
    //Calculate the data rate
    double drt;
    drt = networkLayer->getTxDataRateValue(ackDataRateLevel);

    /* calculate ack duration */
    ch->txtime() = txtime(ch->wu_size(), drt);

//  printf("mac-wu.cc(%d): node_(%d) - Sending ACK back to node %d at rate %d (txtime: %lf)\n",
//      __LINE__, addr(), dst, ackDataRateLevel, ch->txtime());


    /* store ack tx time */
    af->af_duration = (u_int16_t) ch->txtime();

    p->freq1 = freq1;
    p->freq2 = freq2;
    p->pn_code = pn_code;
    p->data_rate = ackDataRateLevel;

    netif_->node()->collision_container.add_entry_to_collision_table(netif_->node(), NOW, NOW + ch->txtime(),
        freq1, freq2, pn_code, networkLayer->getTxPowerValue(powerLevel_), WU_RADIUS, dst);

    setMode(WU_TX);
    tx_rx_timer.resched(ch->txtime());

    double addToEnergy = networkLayer->getTxPowerValue(powerLevel_) * ch->txtime();
    debugTxEnergy += addToEnergy;

    if (ifNeedToPrint(2))
    {
        printf("mac_wu_for_debug: node_(%d) is adding to the TX_energy [%0.6f, tot: %0.5f], sending ACK \n",
            addr(), addToEnergy, debugTxEnergy);
    }

    downtarget_->recv(p, this); //this calls recv() of the wireless-phy.cc, memcheck - manoj
    Packet::free(p);

    if (isAMaster) {
        if ((int(NOW))/10 > timeCounter ) {
            timeCounter++;
//          printf("mac-wu.cc: Time is %f \n", NOW);
        }
    }

}


void Mac_WU::sendDATAasCarrierSensingIsSuccessful(Packet *p)
{
    hdr_cmn* ch = HDR_CMN(p);
    struct hdr_mac_wu* dh = HDR_MAC_WU(p);


    int dst = ETHER_ADDR(dh->dh_ra);
    int src = ETHER_ADDR(dh->dh_ta);
    int macBroadcast = MAC_BROADCAST;

    if (ifNeedToPrint(1))
        printf("mac-wu.cc: node_(%d) sending a frame at time %0.9f with dst as %d and src as %d and mac_broadcast as %d with wu_size as %d \n",
            addr(), NOW, dst, src, macBroadcast, ch->wu_size());
```

```
//    if (ifNeedToPrint(1))
//      printf("mac-wu.cc: node_(%d) - Actual packet size is: %d \n",
//          addr(), ch->size());

    // HUGE HACK FOR BEACONING -- Save old Data Rate to restore later
    int oldDataRateLevel = getDataRateLevel();

    if (ch->ptype() == PT_WU)
    {
        struct hdr_wu_beacon *wb = HDR_WU_BEACON(p);
        if (wb->rp_type == WUTYPE_DATA_RATE_BEACON)
        {
            int newDataRateLevel = wb->data_rate;
//          printf("mac-wu.cc(%d): [%lf] node_(%d) is temporarily setting data rate to level %d\n",
//              __LINE__, NOW, addr(), newDataRateLevel);
            setDataRateLevel(newDataRateLevel);
        }
    }



    /*
     * Update the MAC header
     */
    ch->size() += phymib_.getHdrLenWu();

//  struct hdr_cmn *chNew = HDR_CMN(newP);
//    chNew->ptype() = PT_WU;


    if (ifNeedToPrint(1))
        printf("mac-wu.cc: node_(%d) - Actual packet size after adjusted: %d \n",
            addr(), ch->size());

    dh->dh_fc.fc_protocol_version = MAC_ProtocolVersion;
    dh->dh_fc.fc_type = MAC_Type_Data;
    dh->dh_fc.fc_subtype = MAC_Subtype_Data;

    /* store data tx time */
    ch->txtime() = txtime(ch->wu_size());
    dh->dh_duration = (u_int16_t) ch->txtime();

    netif_->node()->collision_container.add_entry_to_collision_table(netif_->node(), NOW, NOW + ch->txtime(),
        freq1, freq2, pn_code, networkLayer->getTxPowerValue(powerLevel_), WU_RADIUS, dst);

    setMode(WU_TX);

    double addToEnergy = networkLayer->getTxPowerValue(powerLevel_) * ch->txtime();
    debugTxEnergy += addToEnergy;

    if (ifNeedToPrint(2))
        printf("mac_wu_for_debug: node_(%d) is adding to the TX_energy [%0.6f, tot: %0.5f], sending DATA \n", addr(), addToEnergy, debugTxEnergy);

    tx_rx_timer.resched(ch->txtime());

    //Change the TX Power
//  double oldPower = p->txinfo_.getTxPr();
//  p->txinfo_.stamp(p->txinfo_.getNode(), p->txinfo_.getAntenna(), 0.111, p->txinfo_.getLambda());

//  printf("mac-wu.cc: Previous power level of packet: %e. New power level: %e \n", oldPower, p->txinfo_.getTxPr());
```

```
    //printf("mac_wu_for_debug: %0.9f \t %d \n", NOW, p->freq1);
    downtarget_->recv(p, this);  //this calls recv() of the phy.cc. memcheck -manoj
    Packet::free(p);

    //HUGE BEACONING HACK -- Restore old data rate
    setDataRateLevel(oldDataRateLevel);
}


/* ======================================================================
   Incoming Packet Routines
   ====================================================================== */
void Mac_WU::send(Packet *p, Handler *h)
{
    double rTime;
    struct hdr_mac_wu* dh = HDR_MAC_WU(p);

    EnergyModel *em = netif_->node()->energy_model();

    if (em && em->sleep()) {
        em->set_node_sleep(0);
        em->set_node_state(EnergyModel::INROUTE);
    }

    if (ifNeedToPrint(1)) {
        struct hdr_cmn *hdr = HDR_CMN(p);
        printf("mac-wu.cc(%d): node_(%d) is initiating the process of sending a data packet to node_(%d). carrier sensing
            flag is %d with seq_no as %d \n", __LINE__, addr(), ETHER_ADDR(dh->dh_ra), hasCarrierSensing, hdr->seq_no);
    }

    callback_ = h;
    backoff_counter = 0;

    if (hasCarrierSensing == 1) {
        //setMode(WU_CS);
        doCarrierSensingBeforeSendingDATA(p);
    }
    else{
        sendDATAasCarrierSensingIsSuccessful(p);
    }
}


void Mac_WU::doCarrierSensingBeforeSendingDATA(Packet *p)
{
    if (ifNeedToPrint(1))
        printf("mac-wu.cc: node_(%d) is starting carrier sensing for data node at time %0.9f \n",
            addr(), NOW);

    double rssi = netif_->node()->collision_container.getRSSI(netif_->node(), freq1, freq2, addr(), pn_code);

    if (rssi > RSSI_MIN_THRESHOLD)
    {
        if (networkLayer->hasTrace)
            printf("ANALYSIS-TRACE: %d  %d  %ld %d  %d\n", IS_A_BO, addr(), addr(), freq1, int(NOW*1000));

        backoff_counter++;

        if (ifNeedToPrint(2))
        {
```

```
            printf("mac-wu.cc: node_(%d) is starting carrier sensing at %0.9f. RSSI %f is still high. back_counter is
                increased to %d  and freq is %d \n", addr(), NOW, rssi, backoff_counter, freq1);
        }

        if (backoff_counter == MAC_MAX_BACK_OFF_LIMIT) {
            printf("mac-wu.cc: [%lf] node_(%d) Hit max backoff counter limit at %d (doCarrier)\n", NOW, addr(), backoff_counter);
            //if (isAMaster == 0) { //Let network take care of it.
            //networkLayer->startChannelSearch();
            //}
            setMode(WU_IDLE);
            Packet::free(p); //desp
            return;
        }

        //srand((unsigned)time(NULL));
        float random_val = (float)(rand())/RAND_MAX;
        if (ifNeedToPrint(2))
            printf("mac-wu.cc: random_val is %f \n", random_val);

        bo_timer.pktToTx_ = p;
        if (ifNeedToPrint(2))
        {
            printf("mac-wu.cc: [%lf] node_(%d) BACKOFF TIMER SET TO: %lf in dCarrier-\n",
                NOW, addr(), MAC_WU_MIN_BACK_OFF+random_val*MAC_WU_MAX_BACK_OFF);
        }
        bo_timer.resched(MAC_WU_MIN_BACK_OFF+random_val*MAC_WU_MAX_BACK_OFF);
        return;
    }
    else {
        if (ifNeedToPrint(2))
            printf("mac-wu.cc: for node_(%d) RSSI is %f and is acceptable \n", addr(), rssi);
    }

    cs_timer.pktToTx_ = p;
    cs_timer.resched(MAC_WU_CARRIER_SENSE_TIMEOUT);
}


void Mac_WU::recv(Packet *p, Handler *h)
{
    struct hdr_cmn *hdr = HDR_CMN(p);
    hdr_mac_wu *mh = HDR_MAC_WU(p);
    u_int32_t src = ETHER_ADDR(mh->dh_ta);
    u_int32_t rcv = ETHER_ADDR(mh->dh_ra);

    if (ifNeedToPrint(2))
    {
        printf("mac-wu.cc: node_(%d) is starting to receive a frame from node_(%d) with seq_no %d and is intended for
            node %d with addressType %d at time %0.9f \n", addr(), src, hdr->seq_no, rcv, hdr->addr_type_, NOW);
        printState();
    }

    if (hdr->addr_type_ == NS_AF_LOCAL_LOOPBACK) {
        if (ifNeedToPrint(1))
            printf("mac-wu.cc: This packet is a special packet for function update \n");

        networkLayer = (WU*)hdr->genPtr;
        xmit_update_node_pointer = hdr->xmit_update_node_pointer;
        update_node_pointer = hdr->xmit_failure_data_;
        xmit_update_node_pointer(update_node_pointer, netif_->node(), this);
```

```
        if (ifNeedToPrint(1))
            networkLayer->printState();

//        if (ifNeedToPrint(2))
//          printf("mac-wu.cc: loc7 node_(%d) is freeing the message here. ns_af_local_loopback \n", addr());

        Packet::free(p);

        return;
    }

    if (mode == WU_TX) { //Since the radio is sending now, there is no way I can send another frame here!!
        //I am already sending a frame. I can't handle a new frame either from higher layers or from lower layers
        if (ifNeedToPrint(2))
            printf("mac-wu.cc: node_(%d) is already sending a frame. Can't handle a new frame from higher layers nor lower layers\n",
                addr());
//        if (ifNeedToPrint(2))
//          printf("mac-wu.cc: loc8 node_(%d) is freeing the message here. I am transmitting the packet here \n", addr());

        Packet::free(p);

        return;
    }
    //else if (mode == WU_CS) { //Since the radio is doing carrier sensing. Let it do so.
        //if (ifNeedToPrint(2))
        //printf("mac-wu.cc: node_(%d) is doing a carrier sensing. No new frames can be accepted.\n",
            //addr());
        //Packet::free(p);
        //return;
    //}


    /*
     *  Handle outgoing packets.
     */
    if (hdr->direction() == hdr_cmn::DOWN) {
        //if (mode == WU_RX) {
            //printf("mac-wu.cc: node_(%d) is RECEIVING. MAJOR_PROBLEM. It cannot send any data now. \n",
            //addr());
            //Packet::free(p);
            //return;
        //}
        send(p, h);
        return;
      }

    /*
     *  Handle incoming packets.
     *
     *  We just received the 1st bit of a packet on the network
     *  interface.
     *
     */

/*  double addToEnergy = POWER_WU_RX * hdr->txtime();
    debugRxEnergy += addToEnergy;

    if (ifNeedToPrint(2))
        printf("mac-wu.cc: node_(%d) is adding to the RX_energy [%0.6f, tot: %0.5f], receiving DATA \n", addr(), addToEnergy, debugRxEnergy);
*/
    //Check if there were any bit errors
```

```
    if (hdr->error())
    {
        if (ifNeedToPrint(2))
            printf("mac-wu.cc: The packet is corrupt (has bit errors). Hence node_(%d) is dropping packet at MAC layer \n", addr());

        Packet::free(p);
        return;
    }

    if ((freq1 == p->freq1) && (freq2 == p->freq2))
    {
        if (ifNeedToPrint(2))
            printf("mac-wu.cc: Since the channel is same. Let me node_(%d) check for collision \n", addr());

        int if_collision = netif_->node()->collision_container.check_if_there_is_a_collision(netif_->node(), freq1, freq2, src);

        if (if_collision == 1)
        {
            if (ifNeedToPrint(2))
                printf("mac-wu.cc: node_(%d) PROBLEM, there is a collision \n", addr());

            if (networkLayer->hasTrace)
                printf("ANALYSIS-TRACE: %d  %d  %ld %d  %d\n", IS_A_COLLISION, addr(), addr(), freq1, int(NOW*1000));

            if (ifNeedToPrint(1))
                printState();

            if (mode == WU_RX) {
                //Simply corrupt the packet being received. In this way, when
                //it would be sent upwards, it would be detected and dropped.
                if (ifNeedToPrint(2)) {
                    printf("mac-wu.cc: node_(%d) PROBLEM, sorry, I am already receiving \n", addr());
                    printf("\tSimply corrupt the packet being received. In this way, when it is sent upwards, it will be detected and dropped. \n");
                }
                if (pktRx_ != 0)
                    pktRx_->isCorrupt = 1;
                //ack_timer.expire();
                //printf("mac-wu.cc: loc9 node_(%d) is freeing the message here \n", addr());
                //Packet::free(p);
                return;
            }

//            if (ifNeedToPrint(2))
//              printf("mac-wu.cc: loc10 node_(%d) is freeing the message here. A collision has occurred \n", addr());

            Packet::free(p);
        }
    }
    else
    {
        if (ifNeedToPrint(2))
            printf("mac-wu.cc: the frequency is not same. Hence node_(%d) is dropping packet at MAC layer \n", addr());

//        if (ifNeedToPrint(2))
//          printf("mac-wu.cc: loc11 node_(%d) is freeing the message here. outside the frequency range. \n", addr());

        Packet::free(p);
        return;
    }
```

```
    //Beyond this point, everyone has to listen to this packet now!
    if (ifNeedToPrint(2))
        printf("mac-wu.cc: This is an incoming packet with size %d and frequency width [%d-%d] and PN code %d \n",
            hdr->size(), p->freq1, p->freq2, p->pn_code);

    if (pn_code != p->pn_code)
    { //if isRecevingAValidFrame == 1, then that means you are listening to a valid frame. Do not bother
        //to add noise here.
        if (ifNeedToPrint(2))
            printf("mac-wu.cc: PN codes do not match. node_(%d) would drop after receiving it \n", addr());

        //setMode(WU_RX);
        //pktRx_ = p->copy();

        //tx_rx_timer.purpose = PURPOSE_WAS_RECEIVING_NOISE_DROP_THE_PACKET;
        //tx_rx_timer.resched(hdr->txtime());

//        if (ifNeedToPrint(2))
//          printf("mac-wu.cc: loc12 node_(%d) is freeing the message here. pn_codes are not same \n", addr());

        Packet::free(p);
        return;
    }

    u_int8_t  subtype = mh->dh_fc.fc_subtype;

    if (subtype == MAC_Subtype_ACK) {
        if (ifNeedToPrint(1))
            printf("mac-wu.cc: This is an ack with freq range [%d-%d], pn_code: %d \n", p->freq1, p->freq2, p->pn_code);

        //if ((dataPacket_ &&  isAMaster != 1 ) || (dataPacket_ &&  rcv == addr())) {
        if (rcv == addr()) {
            setMode(WU_RX);
            //pktRx_ = p->copy();
            pktRx_ = p;
            recvACK();
            //Packet::free(p);
            return;
        }

//        if (ifNeedToPrint(2))
//          printf("mac-wu.cc: loc13 node_(%d) is freeing the message here. ACK. But is not for me \n", addr());

        Packet::free(p);
    }
    else {
        if (ifNeedToPrint(1))
            printf("mac-wu.cc: This is NOT an ack with freq range [%d-%d], pn_code: %d \n", p->freq1, p->freq2, p->pn_code);

        if (isAMaster == 1) {
            if (ifNeedToPrint(1))
                printf("mac-wu.cc: However, I am a master and so can send an ACK as a reply to node %d \n", src);

            setMode(WU_RX);
            //pktRx_ = p->copy();
            pktRx_ = p;

            recvDATA();
            //Packet::free(p);
            return;
        }
```

```
        else if ((rcv == addr()) || (rcv == IP_BROADCAST)) {
            if (ifNeedToPrint(1))
                printf("mac-wu.cc: Though I am not a master, this frame is for me. Must respond with freq range [%d-%d], pn_code: %d \n",
                    p->freq1, p->freq2, p->pn_code);

            setMode(WU_RX);
            //pktRx_ = p->copy();
            pktRx_ = p;
            recvDATA();
            //Packet::free(p);
            return;
        }
        else {
            Packet::free(p);
            return;
        }
    }

    //printf("mac-wu.cc: loc13 node_(%d) is freeing the message here \n", addr());
    //Packet:free(p);
}



/*
 * txtime() - calculate tx time for packet of size "psz" bytes
 *        at rate "drt" bps
 */
double Mac_WU::txtime(double psz)
{
    double drt = dataRate_;

    //If possible, grab data rate from the network layer
    if (networkLayer)
        drt = networkLayer->getTxDataRateValue(dataRateLevel_);

    return txtime(psz, drt);
}

/*
 * txtime() - calculate tx time for packet of size "psz" bytes
 *        at rate "drt" bps
 */
double Mac_WU::txtime(double psz, double drt)
{
    if (drt == 0.0)
        printf("mac-wu.cc(%d): ERROR: DATA RATE IS ZERO! CANNOT DIVIDE HERE!\n", __LINE__);
    if (basicRate_ == 0.0)
        printf("mac-wu.cc(%d): ERROR: BASIC DATA RATE IS ZERO! CANNOT DIVIDE HERE!\n", __LINE__);

    double t = 8 * psz/drt;
    t += 8 * (CRC_SEED_LENGTH + LENGTH_FIELD)/drt;
    t += 8 * (NUMBER_STARTER_FRAMES * START_FRAME_SIZE)/basicRate_;

    if (ifNeedToPrint(1)) {
        printf("mac-wu.cc: + node_(%d) - Calculating txtime - size is %d and data rate is %d kbps; basicRate_ is %d kbps \n",
            addr(), (int)psz, (int)drt, (int)basicRate_);
        printf("mac-wu.cc:  txTime is %f \n", t);
    }
    return(t);
}
```

```
void Mac_WU::recvDATA()
{
    struct hdr_mac_wu *dh = HDR_MAC_WU(pktRx_);
    u_int32_t dst, src, size;
    struct hdr_cmn *ch = HDR_CMN(pktRx_);

    if (ifNeedToPrint(2))
        printf("mac-wu.cc: [%lf] node_(%d) is receiving data... \n", NOW, addr());

    dst = ETHER_ADDR(dh->dh_ra);
    src = ETHER_ADDR(dh->dh_ta);
    size = ch->size();
    /*
     * Adjust the MAC packet size - ie; strip
     * off the mac header
     */
    ch->size() -= phymib_.getHdrLenWu();
    ch->num_forwards() += 1;

    if (ifNeedToPrint(2))
        printf("mac-wu.cc: [%lf] node_(%d) has started receiving the packet with seq_no %d \n", NOW, addr(), ch->seq_no);

    setMode(WU_RX);
    need_to_send_an_ack_to = src;
    ack_data_rate_level = pktRx_->data_rate;

//  printf("mac-wu.cc: [%lf] node_(%d) is saving need_to_send_an_ack_to %d, ack_data_rate_level %d for pkt (seq: %d) will finish at %lf\n",
//      NOW, addr(), need_to_send_an_ack_to, ack_data_rate_level, ch->seq_no, NOW+ch->txtime());

    double addToEnergy = POWER_WU_RX * ch->txtime();
    debugRxEnergy += addToEnergy;

    if (ifNeedToPrint(2))
        printf("mac_wu_for_debug: node_(%d) is adding to the RX_energy [%0.6f, tot: %0.5f], receiving DATA \n",
            addr(), addToEnergy, debugRxEnergy);

    pktRx_->isCorrupt = 0;
    tx_rx_timer.resched(ch->txtime());

}

void Mac_WU::recvACK() {

    struct hdr_cmn *ch = HDR_CMN(pktRx_);
    struct hdr_mac_wu *dh = HDR_MAC_WU(pktRx_);
    int senderNode = ETHER_ADDR(dh->dh_ta);
    int rcv = ETHER_ADDR(dh->dh_ra);

    if (ifNeedToPrint(1))
        printf("mac-wu.cc: node_(%d) got an ACK from node_(%d) at time %0.9f, and is destined to %d \n",
            addr(), senderNode, NOW, rcv);

    //if ((dataPacket_ &&  isAMaster != 1 ) || (dataPacket_ &&  rcv == addr())) {
    if (ifNeedToPrint(2))
        printf("wu.cc: Let us listen to this ack for duration %0.9f \n", ch->txtime());
    //struct hdr_mac_wu* dh2 = HDR_MAC_WU(dataPacket_);

    pktRx_->isCorrupt = 0;
    setMode(WU_RX);
```

```
    double addToEnergy = POWER_WU_RX * ch->txtime();
    debugRxEnergy += addToEnergy;

    if (ifNeedToPrint(2))
        printf("mac_wu_for_debug: node_(%d) is adding to the RX_energy [%0.6f, tot: %0.05f], receiving ACK \n",
            addr(), addToEnergy, debugRxEnergy);


    //If we have a dynamic policy that requires RSSI values, sniff the air now during receive
    if (!networkLayer->isAMaster && (networkLayer->getDataRatePolicy() == WU_DATA_RATE_POLICY_SNR_THRESHOLD))
    {
        //Check the RSSI during the start of the RECEIVE
//      netif_->node()->collision_container.print_collision_table(addr());
        double dataRSSI = netif_->node()->collision_container.getDataRSSI(netif_->node(), freq1,
            freq2, addr(), pn_code);
        double noiseRSSI = netif_->node()->collision_container.getNoiseRSSI(netif_->node(), freq1,
            freq2, addr(), pn_code);

        //Calculate the SNR
        double snrValue = 0.0;
        if (noiseRSSI == 0)
        {
//          noiseRSSI = 1;  //Noise can never be zero
            noiseRSSI = -1;

        }
        if (noiseRSSI < RSSI_MIN_SENSING_DBM)
        {
            noiseRSSI = RSSI_MIN_SENSING_DBM;
        }
        snrValue = ((double)dataRSSI / (double)noiseRSSI);

        printf("mac-wu.cc(%d): [%lf] node_(%d) is receiving data with RSSI of %.03f (noise RSSI is %.03f, ratio: %.04f)\n",
            __LINE__, NOW, addr(), dataRSSI, noiseRSSI, snrValue);
        lastSNR = snrValue;
    }

    ack_timer.senderNode = senderNode;
    //ack_timer.dataPacketPtr_ = dataPacket_->copy();
    ack_timer.resched(ch->txtime());
    //}

    //Packet *np = dataPacket_;
    //if (ifNeedToPrint(2))
    //printf("mac-wu.cc: loc14 node_(%d) is freeing the message here at recv ack\n", addr());
    //free(np);
    //dataPacket_ = 0;
}

/** This function returns the int value of the highest data rate possible based on current snr thresholds.
 * @param double snrValue The Signal To Noise Ratio
 * @return int data rate level
 */
int Mac_WU::getHighestDataRateBasedOnSNR(double snrValue)
{
    int highestDataRate = 0;

    //FOR RSSI INDEX
/*  for (int i = WU_MAX_RATE_LEVEL; i >= 0; i--)    //Start from Highest DR to Lowest
    {
```

```
        if (snrValue > dataRateSNRThresholds[i])
        {
            highestDataRate = i;
            break;
        }
    }
*/

    //FOR DBm
    for (int i = WU_MAX_RATE_LEVEL; i >= 0; i--)    //Start from Highest DR to Lowest
    {
        if (snrValue < dataRateSNRThresholds[i])
        {
            highestDataRate = i;
            break;
        }
    }

    return highestDataRate;
}

/** This function updates a SNR threshold by relaxing it or constraining it based upon success or failure at that rate.
 *  @prarm int dataRateLevel The data rate level
 *  @param int relaxOrConstrain (0 = CONSTRAIN, 1 = RELAX)
 */
void Mac_WU::updateDataRateSNRThreshold(int dataRateLevel, int relaxOrConstrain)
{
    //The percent to increase or decrease
    double increment = 0.25;

    if ((dataRateLevel < 0) || (dataRateLevel > WU_MAX_RATE_LEVEL) ||
        ((relaxOrConstrain != WU_DATA_RATE_THRESHOLD_RELAX) && (relaxOrConstrain != WU_DATA_RATE_THRESHOLD_CONSTRAIN)))
    {
        printf("mac-wu.cc(%d): ERROR - Cannot update Data Rate SNR Threshold. Bad values passed in (dataRateLevel %d, relaxOrConstrain %d)\n",
            __LINE__, dataRateLevel, relaxOrConstrain);
        return;
    }

    if (relaxOrConstrain == WU_DATA_RATE_THRESHOLD_RELAX)
    {
        printf("\t\trelaxing...\n");
//      dataRateSNRThresholds[dataRateLevel] -= increment * (dataRateSNRThresholds[dataRateLevel]); //Decrease by this percent
    }
    else if (relaxOrConstrain == WU_DATA_RATE_THRESHOLD_CONSTRAIN)
    {
        printf("\t\tconstraining...\n");
//      dataRateSNRThresholds[dataRateLevel] += increment * (dataRateSNRThresholds[dataRateLevel]); //Decrease by this percent
    }

    printf("mac-wu.cc: Data Rate Threshold level %d updated (relaxed or constrained? %d) to:\n", dataRateLevel, relaxOrConstrain);
    printDataRateSNRThresholds();

    return;
}
void Mac_WU::printDataRateSNRThresholds()
{
    printf("=== [%lf] === node_(%d) DR Thresholds ===\n", NOW, addr());
    printf("\tRate\t|\tThreshold\t\n");
    for (int i = 0; i <= WU_MAX_RATE_LEVEL; i++)
    {
        printf("\t%d\t|\t%lf\t\n", i, dataRateSNRThresholds[i]);
```

```
    }
    printf("=========================================\n");
}

void Mac_WU::setDataRateLevel(int newDataRate)
{
//  printf("mac-wu.cc: [%lf] node_(%d) is setting its data rate level to %d\n", NOW, addr(), newDataRate);
    dataRateLevel_ = newDataRate;
}
```

A.5 WirelessUSB PHY Layer

```
/* -*-  Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*-
 * wireless-phy.cc
 */

...

/* ======================================================================
   WirelessPhy Interface
   ====================================================================== */
static class WirelessPhyClass: public TclClass {
public:
        WirelessPhyClass() : TclClass("Phy/WirelessPhy") {}
        TclObject* create(int, const char*const*) {
                return (new WirelessPhy);
        }
} class_WirelessPhy;


WirelessPhy::WirelessPhy() : Phy(), idle_timer_(this), status_(IDLE)
{
    /*
     *  It sounds like 10db should be the capture threshold.
     *
     *  If a node is presently receiving a packet a a power level
     *  Pa, and a packet at power level Pb arrives, the following
     *  comparion must be made to determine whether or not capture
     *  occurs:
     *
     *    10 * log(Pa) - 10 * log(Pb) > 10db
     *
     *  OR equivalently
     *
     *    Pa/Pb > 10.
     *
     */
    bind("CPThresh_", &CPThresh_);
    bind("CSThresh_", &CSThresh_);
    bind("RXThresh_", &RXThresh_);
    //bind("bandwidth_", &bandwidth_);
    bind("Pt_", &Pt_);
    bind("freq_", &freq_);
    bind("L_", &L_);

    lambda_ = SPEED_OF_LIGHT / freq_;

    node_ = 0;
    ant_ = 0;
    propagation_ = 0;
```

```
    modulation_ = 0;

    // Assume AT&T's Wavelan PCMCIA card -- Chalermek
        //  Pt_ = 8.5872e-4; // For 40m transmission range.
    //      Pt_ = 7.214e-3;  // For 100m transmission range.
    //      Pt_ = 0.2818; // For 250m transmission range.
    //  Pt_ = pow(10, 2.45) * 1e-3;         // 24.5 dbm, ~ 281.8mw

    Pt_consume_ = 0.660;  // 1.6 W drained power for transmission
    Pr_consume_ = 0.395;  // 1.2 W drained power for reception

    //  P_idle_ = 0.035; // 1.15 W drained power for idle

    P_idle_ = 0.0;

    channel_idle_time_ = NOW;
    update_energy_time_ = NOW;
    last_send_time_ = NOW;

    idle_timer_.resched(1.0);
}

int WirelessPhy::sendUp(Packet *p)
{
    /*
     * Sanity Check
     */
    assert(initialized());
    hdr_cmn *ch = HDR_CMN(p);

    #ifdef PRINT_ENERGY
    printf("wireless_phy_debug: node_%d at sendup , rx_energy %f, tx_energy %f \n",
        node()->address(), debugRxEnergy, debugTxEnergy);
    #endif

    PacketStamp s;
    double Pr;
    int pkt_recvd = 0;

    // if the node is in sleeping mode, drop the packet simply
    if (em())
        if (em()->sleep() || (em()->node_on() != true)) {
            pkt_recvd = 0;
            goto DONE;
        }

    // if the energy goes to ZERO, drop the packet simply
    if (em()) {
        if (em()->energy() <= 0) {
            pkt_recvd = 0;
            goto DONE;
        }
    }

    if (propagation_) {
        s.stamp((MobileNode*)node(), ant_, 0, lambda_);

        Pr = propagation_->Pr(&p->txinfo_, &s, this);
        //printf("wireless-phy.cc: receiving packet (sq: %d) - txPower: %lf rcvPower: %e \n", ch->seq_no, p->txinfo_.getTxPr(), Pr);
        //printf("wireless-phy.cc: CSThresh_ is: %e RXThresh_ is: %e \n", CSThresh_, RXThresh_);
        #ifdef DEBUG
```

```
        printf("wireless-phy.cc here in wireless-phy rcvPower: %lf \n", Pr);
        #endif

        if (Pr < CSThresh_) {
            //printf("wireless-phy.cc: node_(%d) Can't even hear packet. Discarding at rcvPower %e\n", node()->address(), Pr);
            pkt_recvd = 0;
            goto DONE;
        }
        if (Pr < RXThresh_) {
            //printf("wireless-phy.cc: node_(%d) Can hear packet, but can't discern it. Discarding at rcvPower %e\n", node()->address(), Pr);
            /*
             * We can detect, but not successfully receive
             * this packet.
             */
            hdr_cmn *hdr = HDR_CMN(p);
            hdr->error() = 1;
#if DEBUG > 3
            printf("SM %f.9 _%d_ drop pkt from %d low POWER %e/%e\n",
                   Scheduler::instance().clock(), node()->index(),
                   p->txinfo_.getNode()->index(),
                   Pr,RXThresh);
#endif
        }
    }
    if (modulation_) {
        hdr_cmn *hdr = HDR_CMN(p);
        hdr->error() = modulation_->BitError(Pr);
    }

    //printf("wireless-phy.cc: node_(%d) Received Packet successfullly at rcvPower %e ...\n", node()->address(), Pr);
    /*
    * The MAC layer must be notified of the packet reception
    * now - ie; when the first bit has been detected - so that
    * it can properly do Collision Avoidance / Detection.
    */
    pkt_recvd = 1;

DONE:
    p->txinfo_.getAntenna()->release();

    /* WILD HACK: The following two variables are a wild hack.
       They will go away in the next release...
       They're used by the mac-802_11 object to determine
       capture.  This will be moved into the net-if family of
       objects in the future. */
    p->txinfo_.RxPr = Pr;
    p->txinfo_.CPThresh = CPThresh_;

    /*
     * Decrease energy if packet successfully received
     */
    if (pkt_recvd && em()) {
        //double rcvtime = (8. * hdr_cmn::access(p)->size())/bandwidth_;
        double rcvtime = hdr_cmn::access(p)->txtime();
        // no way to reach here if the energy level < 0

        /*
          node()->add_rcvtime(rcvtime);
          em()->DecrRcvEnergy(rcvtime,Pr_consume_);
        */
```

```
        double start_time = MAX(channel_idle_time_, NOW);
        double end_time = MAX(channel_idle_time_, NOW+rcvtime);
        double actual_rcvtime = end_time-start_time;

        if (start_time > update_energy_time_) {
            em()->DecrIdleEnergy(start_time-update_energy_time_,
                P_idle_);
            update_energy_time_ = start_time;
            debugIdleEnergy += (start_time-update_energy_time_)* P_idle_;
        }

        em()->DecrRcvEnergy(actual_rcvtime,Pr_consume_);

        debugRxEnergy += actual_rcvtime * Pt_consume_;

        if (end_time > channel_idle_time_) {
            status_ = RECV;
        }

        channel_idle_time_ = end_time;
        update_energy_time_ = end_time;

        /*
          hdr_diff *dfh = HDR_DIFF(p);
          printf("Node %d receives (%d, %d, %d) energy %lf.\n",
          node()->address(), dfh->sender_id.addr_,
          dfh->sender_id.port_, dfh->pk_num, node()->energy());
        */

        if (em()->energy() <= 0) {
            // saying node died
            em()->setenergy(0);
            ((MobileNode*)node())->log_energy(0);
        }
    }

    //Randomly corrupt one in 10,000 of the packets
//    float random_val = (float)(rand())/700;
//  printf(" random value: %lf\n", random_val);
/*  if ((rand() % 10000) == 0)
    {
        hdr_cmn *hdr = HDR_CMN(p);
        hdr->error() = 1;
        printf("%s(%d): randomly corrupting a received packet!\n", __FILE__, __LINE__);
    }
*/
    ch->energyForTrace = debugRxEnergy + debugTxEnergy;
//  printf("wireless-phy.cc(%d): node_(%d) - Setting energyForTrace to: %lf\n", __LINE__, node()->address(), ch->energyForTrace);

    return pkt_recvd;
}

...

void WirelessPhy::UpdateIdleEnergy()
{
    if (em() == NULL) {
        return;
    }
    if (NOW > update_energy_time_ && em()->node_on()) {
          em()-> DecrIdleEnergy(NOW-update_energy_time_,
```

```
                    P_idle_);
            update_energy_time_ = NOW;
            debugIdleEnergy += (NOW - update_energy_time_)* P_idle_;
    }

    // log node energy
    if (em()->energy() > 0) {
        ((MobileNode *)node_)->log_energy(1);
    }
    else {
        ((MobileNode *)node_)->log_energy(0);
    }

    idle_timer_.resched(10.0);
}

double WirelessPhy::getDist(double Pr, double Pt, double Gt, double Gr,
                double hr, double ht, double L, double lambda)
{
    if (propagation_) {
        return propagation_->getDist(Pr, Pt, Gt, Gr, hr, ht, L,
                        lambda);
    }
    return 0;
}
```

A.6 WirelessUSB common/packet.h

```
/* -*-  Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
class Packet : public Event {

...

public:

...

    //Copy Constructor
    Packet(Packet *p) : bits_(0), data_(0), ref_count_(0), next_(0) {
        bits_ = new unsigned char[hdrlen_];
        if (p->bits())
            memcpy(bits_, p->bits(), hdrlen_);

        if (p->data_)
            data_ = p->data_->copy();

        txinfo_.init(&p->txinfo_);
        freq1 = p->freq1;
        //p->for_srp_random_toplogy = for_srp_random_toplogy;    // type of next_hop_ addr
        freq2 = p->freq2;
        pn_code = p->pn_code;
        data_rate = p->data_rate;

    }

private:
...

    // the pkt stamp carries all info about how/where the pkt
    // was sent needed for a receiver to determine if it correctly
```

```
    // receives the pkt
    PacketStamp txinfo_;
    short freq1;
    short freq2; //Two frequencies occupied by the packet.
    short pn_code;
    short data_rate;    //The rate at which this packet was sent (0 - MAX Data Rate)
    short isCorrupt;
    //Frequency spread is essentially freq2-freq1.

/*
        * According to cmu code:
        * This flag is set by the MAC layer on an incoming packet
        * and is cleared by the link layer.  It is an ugly hack, but
        * there's really no other way because NS always calls
        * the recv() function of an object.
    *
        */
    u_int8_t        incoming;

//monarch extns end;

};

struct hdr_cmn
{

...

    double energyForTrace;

...

        int seq_no; //For WU
        int freq1; //For internal messaging inside the WU
        int freq2; //For internal messaging inside the WU
        int pn_code; //For internal messaging inside the WU
        int data_rate; //For internal messaging inside the WU
...

};

...

inline Packet* Packet::copy() const
{
    Packet* p = alloc();
    memcpy(p->bits(), bits_, hdrlen_);
    if (data_)
        p->data_ = data_->copy();
    p->txinfo_.init(&txinfo_);
    p->freq1 = freq1;
    //p->for_srp_random_toplogy = for_srp_random_toplogy;    // type of next_hop_ addr
    p->freq2 = freq2;
    p->pn_code = pn_code;
    p->data_rate = data_rate;

    return (p);

}

...
```

A.7 WirelessUSB common/collision.cc

...

```
/** This function currently returns the current RSSI for a given frequency */
double collision::getNoiseRSSI(Node *node_, int freq1, int freq2, int address_, int pn_code)
{
    int rssi = 0;

    int jitterPercent = 7;  //7 = 7%, 5 = 5%, etc.

    int i = 0;
    double rssiDBm = -10e3;
    int entryFound = 0;
    //printf("==================================================\n");

    MobileNode *mn = (MobileNode *) node_;
    double x = mn->X();
    double y = mn->Y();
    int address = node_->address();
    double gainAntennaTx = 1.0;
    double gainAntennaRx = 1.0;
    double PI = 3.14159;
    double M, distance;
    double SPEED_OF_LIGHT = 3e+8;
    double FREQ = 914e+6;
    double lambda_ = SPEED_OF_LIGHT/FREQ;
    double originalPower = 0.0;
    double rxPower = 0.0;

    //printf("node.cc: printing collision table for node_%d with x:%f and y:%f\n", address, x, y);
    //print_collision_table(node_->address());


    for (i=i; i < TOTAL_COLLISION_ENTRY; i++)
    {
        if ((collision_container[i].endTime >= (NOW-.01))
            && (collision_container[i].nodeid != address)
            && (collision_container[i].freq1 <= freq1)
            && (collision_container[i].freq2 >= freq2))
        {
            if (((collision_container[i].receiver == address) || (collision_container[i].receiver == IP_BROADCAST))
                && (collision_container[i].pn_code == pn_code)){
                //printf("node.cc: This entry is for me, and with the same pn_code, must NOT add as a noise \n");
                continue;
            }

            distance = sqrt((collision_container[i].x - x)*(collision_container[i].x - x) +
                    (collision_container[i].y - y)*(collision_container[i].y - y));
            if (distance == 0.0)
            {
                distance = 1.0;
            }

            //printf("The nodes are %d and me %d \n", collision_container[i].nodeid, address_);
            M = lambda_ / (4 * PI * distance);

            //Not correct: - BarlowJ, using 2nd line because txpower is already converted from dbm to mWatts
            //originalPower = pow(10, (collision_container[i].txpower)/10);
            originalPower = collision_container[i].txpower;

            rxPower += originalPower * (gainAntennaTx * gainAntennaRx * (M * M))/1.0;
```

```
            //Add all the noise in mWatts to rxPower. Later on convert the sum
            //into decibels. You can't add decibels directly!

        }
    }
    //printf("================================================\n");

    if (rxPower != 0.0)
    {
        //printf("collision.cc: Total RxPowerMw:%f \n", rxPower);
        rssiDBm = 10 * log10(rxPower);
        //printf("collision.cc: Total RxPowerDb or rssi is %f \n", rssi);
    }

    //Add Jitter?
    if (jitterPercent > 0)
    {
        double percentToJitter = (rand() % (jitterPercent*10)) * 0.001;
        if ((rand() % 2) == 0)  //50/50 chance of being negative or positive
            percentToJitter = (1.0 - percentToJitter);
        else
            percentToJitter = (1.0 + percentToJitter);

        printf("collision.cc: adding JITTER to reading. Percent Jitter = %lf\n", percentToJitter);

        rssiDBm = rssiDBm * percentToJitter;
    }

    return rssiDBm;
}

/** This function currently returns the current RSSI for a given frequency during a receive
 * (doesn't use any other noise) */
double collision::getDataRSSI(Node *node_, int freq1, int freq2, int address_, int pn_code)
{
    int i = 0;

    int jitterPercent = 5;  //7 = 7%, 5 = 5%, etc.

    int rssi = 0;   //To be returned
    double rssiDBm = -10e3;
    int entryFound = 0;
    //printf("=====================================\n");

    MobileNode *mn = (MobileNode *) node_;
    double x = mn->X();
    double y = mn->Y();
    int address = node_->address();
    double gainAntennaTx = 1.0;
    double gainAntennaRx = 1.0;
    double PI = 3.14159;
    double M, distance;
    double SPEED_OF_LIGHT = 3e+8;
    double FREQ = 914e+6;
    double lambda_ = SPEED_OF_LIGHT/FREQ;
    double originalPower = 0.0;
    double rxPower = 0.0;

    //printf("node.cc: printing collision table for node_%d with x:%f and y:%f\n", address, x, y);
    //print_collision_table(node_->address());
```

```
for (i=i; i < TOTAL_COLLISION_ENTRY; i++)
{
    if ((collision_container[i].endTime >= NOW)
        && (collision_container[i].nodeid != address)
        && (collision_container[i].freq1 <= freq1)
        && (collision_container[i].freq2 >= freq2))
    {
        if (((collision_container[i].receiver != address) && (collision_container[i].receiver != IP_BROADCAST))
            || (collision_container[i].pn_code != pn_code)) {
            //printf("node.cc: This entry is not for me, must NOT add as signal strength during receive \n");
            continue;
        }

        distance = sqrt((collision_container[i].x - x)*(collision_container[i].x - x) +
                (collision_container[i].y - y)*(collision_container[i].y - y));
        printf("collision.cc: [%lf] node_(%d) is at distance %lf\n", NOW, address, distance);

        if (distance == 0.0)
        {
            distance = 1.0;
        }

        //printf("The nodes are %d and me %d \n", collision_container[i].nodeid, address_);
        M = lambda_ / (4 * PI * distance);

        //Not correct: - BarlowJ, using 2nd line because txpower is already converted from dbm to mWatts
        //originalPower = pow(10, (collision_container[i].txpower)/10);
        originalPower = collision_container[i].txpower;

        rxPower += originalPower * (gainAntennaTx * gainAntennaRx * (M * M))/1.0;
        //Add all the noise in mWatts to rxPower. Later on convert the sum
        //into decibels. You can't add decibels directly!

        //printf("collision.cc: TxPowerMw:%f, RxPowerMw:%f \n", originalPower, rxPower);

        //rxPower = collision_container[i].txpower + 10 * log10((gainAntennaTx * gainAntennaRx * (M * M))/1.0);
    }
}
//printf("====================================\n");

if (rxPower != 0.0)
{
    //printf("collision.cc: Total RxPowerMw:%f \n", rxPower);
    rssiDBm = 10 * log10(rxPower);
    //printf("collision.cc: Total RxPowerDb or rssi is %f \n", rssi);
}

//Add Jitter?
if (jitterPercent > 0)
{
    double percentToJitter = (rand() % (jitterPercent*10)) * 0.001;
    if ((rand() % 2) == 0)  //50/50 chance of being negative or positive
        percentToJitter = (1.0 - percentToJitter);
    else
        percentToJitter = (1.0 + percentToJitter);

    printf("collision.cc: adding JITTER to reading. Percent Jitter = %lf\n", percentToJitter);

    rssiDBm = rssiDBm * percentToJitter;
}
```

```
    return rssiDBm;
}
```

Bibliography

[1] D. Qiao, S. Choi, A. Jain, and K. G. Shin, "MiSer: an optimal low-energy transmission strategy for IEEE 802.11a/h," in *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking.* New York, NY, USA: ACM Press, 2003, pp. 161–175.

[2] "Bluetooth core specification, version: 1.2," Bluetooth Association, November 2003. [Online]. Available: http://www.bluetooth.org/spec/

[3] "IEEE 802.15.4 specific requirements," IEEE, October 2003. [Online]. Available: http://www.bluetooth.org/spec/

[4] "EM2420 2.4 GHz IEEE 802.15.4 / ZigBee RF transceiver," Ember, 2004. [Online]. Available: http://www.ember.com/downloads/pdfs/EM2420datasheet.pdf

[5] "CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF transceiver," Chipcon, June 2004. [Online]. Available: http://www.chipcon.com

[6] M. Anand, E. B. Nightingale, and J. Flinn, "Self-tuning wireless network power management," in *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking.* New York, NY, USA: ACM Press, 2003, pp. 176–189.

[7] R. Min and A. Chandrakasan, "A framework for energy-scalable communication in high-density wireless networks," International Symposium on Low Power Electronics and Design, August 2002.

[8] R. Kumar and R. Mahapatra, "Optimizing power at the physical layer in a wireless ad hoc network," IEEE International Conference on Network Protocols, 2002.

[9] V. Rodoplu and T. Meng, "Minimum energy mobile wireless networks," *IEEE Journal On Selected Areas In Communications*, vol. 17, no. 8, pp. 1333–1344, August 1999.

[10] UCB/LBNL/VINT, "The ns-2 network simulator," October 2004. [Online]. Available: http://www.isi.edu/nsnam/ns/

[11] "IEEE 802.11g. part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications: Further higher data rate extension in the 2.4 ghz band," Amendment to IEEE 802.11 Standard, June 2003.

[12] "WirelessUSB standard protocol v0.4," Cypress Semiconductor Corporation, Internal Document, January 2005.

[13] R. Woodings and M. Pandey, "WirelessUSB: A low power, low latency and interference immune wireless standard," in *Proceedings of the IEEE Wireless Communications & Networking Conference (WCNC)*, April 2006.

[14] L. Li, J. Y. Halpern, P. Bahl, Y.-M. Wang, and R. Wattenhofer, "A cone-based distributed topology-control algorithm for wireless multi-hop networks," *IEEE/ACM Trans. Netw.*, vol. 13, no. 1, pp. 147–159, 2005.

[15] W. Teerpabkajorndet and P. Krishnamurthy, "Rate and power control on a reverse link for multi-cell mobile data networks," in *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems.* New York, NY, USA: ACM Press, 2004, pp. 260–267.

[16] S. Koskie and Z. Gajic, "A nash game algorithm for SIR-based power control in 3G wireless CDMA networks," *IEEE/ACM Trans. Netw.*, vol. 13, no. 5, pp. 1017–1026, 2005.

[17] C. Saraydar, N. Mandayam, and D. Goodman, "Efficient power control via pricing in wireless data networks," *IEEE Vehicular Transactions on Communications*, vol. 50, pp. 291–303, 2002.

[18] S. Mohapatra and N. Venkatasubramanian, "A game theoretic approach for power aware middleware," in *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware.* New York, NY, USA: Springer-Verlag New York, Inc., 2004, pp. 417–438.

[19] S. Sheu, Y. Lee, and M. Chen, "Providing multiple data rates in infrastructure wireless networks," *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE*, vol. 3, pp. 1908–1912, 2001.

[20] M. Lacage, M. H. Manshaei, and T. Turletti, "IEEE 802.11 rate adaptation: a practical approach," in *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems.* New York, NY, USA: ACM Press, 2004, pp. 126–134.

[21] Z. Ji, Y. Yang, J. Zhou, M. Takai, and R. Bagrodia, "Exploiting medium access diversity in rate adaptive wireless LANs," in *MobiCom '04: Proceedings of the 10th annual international conference on Mobile computing and networking.* New York, NY, USA: ACM Press, 2004, pp. 345–359.

[22] J. Kim and N. Bambos, "Power-efficient MAC scheme using channel probing in multirate wireless ad hoc networks," *IEEE Vehicular Technology Conference*, vol. 4, pp. 2380–2384, 2002.

[23] M. Heusse, F. Rousseau, R. Guillier, and A. Duda, "Idle sense: an optimal access method for high throughput and fairness in rate diverse wireless LANs," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications.* New York, NY, USA: ACM Press, 2005, pp. 121–132.