RATE CONTROL FRAMEWORK FOR NET-X

BY

NISTHA TRIPATHI

B.E., Shri G S Institute of Technology and Science, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Rate Control Framework for Net-X

Approved by
Dr. N. H. Vaidya

_____

_____

ABSTRACT

All the different PHY layers for the IEEE 802.11 WLAN (a/b/g) support multi-rate capabilities. There have been several initiatives in the area of exploiting these capabilities to maximize network performance. Implementation of these algorithms poses a considerable challenge and only few have been implemented in real world scenario. Most of the existing implementations are done entirely at the wireless device driver level. The notable shortcoming of the driver approach is that it does not take into account the particular needs of an application. For example, multimedia streaming applications have higher sensitivity to delay and jitter and more tolerance to packet losses. However, since most statistical rate control algorithms are designed to maximize throughput, they are not conducive to good quality multimedia applications. Another demerit is that they work wholly on link level and do not take into account the information available at higher layers such as network topology. There has been research that showed that the maximum aggregate throughput for different networks varies with the topology and density. This information is present above the link layer and cannot be used in the conventional link layer based rate control algorithms. Hence, we need a complete framework that spans beyond the link layer for a more effective rate control.

These factors motivated us to develop a rate control framework that supports high level control over rate adaptation. A minimal support for transmit power control is also implemented but a detailed design has not been provided since changing the power level at the interface requires card reset and results in high latency. We have implemented the system in existing Net-X framework which supports multiple interface multiple channel protocols.

To my parents, brother and sister

ACKNOWLEDGMENTS

I would like to express my gratitude to all those people who helped me directly or indirectly in completion of this thesis. I am deeply indebted to my advisor Dr. Nitin Vaidya under whose guidance and encouragement, I chose this topic and enjoyed working on my thesis. His stimulating suggestions, invaluable time and experience in the field helped me in overcoming the problems I faced and putting up my best.

I want to extend a special thanks to Pradeep Kyasanur and Chandrakanth Chereddi, whose work I am extending in my thesis. If it was not for them, I wouldn't be working on this topic. Through my thesis, I have realized the value of their original work. I also thank my fellow researcher Vartika Bhandari who gave invaluable insights into many practical problems I faced at work. Her positive critical comments have helped more than any knowledge. I am truly grateful to everyone of them. On a different note, I would like to thank Anthony for all the wonderful coffee hours, light discussions and helping me go through the thesis-trauma we shared in the last year.

Finally, I thank my parents and other family members because of whom I am here today. Their faith and inspiration is the only reason I have survived. No words can express my gratitude but it is the least I can do.

I also acknowledge NSF (National Science Foundation), for financially supporting my thesis research and implementation.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

This thesis presents a Rate Control Framework for the Net-X project [1]. We refer to the design of our rate control framework as *minmax* to emphasize on the information that needs to be passed on to the wireless driver, i.e., min-rate and max-rate. Net-X project aims at developing generic support for utilizing interface capabilities and integrating this support cleanly into the network stack. Its original implementation focuses primarily on multiple channels multiple interface support but the design is amenable for rate and power control. In this thesis, we extend the support for multiple data rates and transmit powers in Net-X. We have developed generic architectural support in Linux that provides a clean mechanism for a user level application to control the use of transmission rates. Our system provides a two-level joint rate control between higher layers and the link layer. It can be used to implement any protocols that require such a higher level rate control.

Wireless technologies, such as Institute for Electrical and Electronics Engineers (IEEE) 802.11a [2] support multiple bit-rates for transmissions. The data rates are decided as a trade-off between throughput (or any other performance metric) and loss probability, the final aim being best performance in terms of that metric. The desire for correct rate selection becomes all the more critical given the volatility of wireless medium used by IEEE 802.11 standard. This volatile nature is caused by fading, attenuation, interference from other wireless sources, etc. These variations can be either short-lived or long-lived.

Therefore, we need algorithms which can dynamically adapt the transmission parameters according to the channel conditions.

The main goal of our rate control framework is to provide the user with an ability to make a high-level decision about the rates to be used. We could find very few efforts of such a user-level rate control architecture. Most of the work in the literature is confined to the rate control at driver level. This has been deemed adequate till now because the driver has the necessary information. The driver is aware of transmission failures statistics, physical parameters information such as received signal strength indication (RSSI), and medium access control (MAC) level parameters like packet fragmentation threshold, request to send / clear to send (RTS/CTS) threshold, and frame error rate (FER). However, we argue that these factors cannot alone govern the optimum rate. There are other factors which need to be considered such as:

1. *Application:* Different applications have different requirements. For example, streaming applications cannot tolerate higher jitter and delay but they can compromise on overall throughput.

2. *Topology:* There are some works in literature [3], that suggest topology awareness helps in maximizing the throughput by exploiting spatial reuse via joint control of carrier sense threshold and transmission rates.

This suggests that there is a need for user defined rate control policies since the higher layers are better aware of the factors like network topology and application requirements. Hence, it can make intelligent decisions about which set of rates should be used for a communication. Of course, the ultimate finer grain control about rate selection is done at the driver level. The user level policy may use feedback from the driver about link quality and transmission statistics if needed. Together, the link layer and the higher layers can provide the best bit-rate selection which is more effective than rate control done wholly at the device driver.

Net-X Rate Control Framework lets the user policy choose a range of rates (by selecting a minimum and maximum rate). This *minmax* architecture is simple yet sufficiently elegant to implement protocols like [3]. We did not see any particular benefit in selecting non-contiguous sets of rates and so, we use a contiguous set of rates supported by the physical (PHY) layer for the IEEE 802.11 standard (a/b/g) in use. In the driver, we made modification so that it will select the best transmission rate to a destination within this range. Thus, the coarse control of rate selection is done by the user-space policy whereas driver selects the exact bit-rate to be used for the actual packet transmission.

We also provide the ability to set transmission rate for broadcast packets at the application level. In the existing systems, routing packets such as route discovery packets, control frames such as RTS/CTS and many other broadcast packets are sent at the base rates so that they are delivered reliably since there are no retransmissions for broadcast packets. However, this may create problems in some scenarios. For example, consider the Net-X network running multi-channel protocol [4], hello packets are broadcasted at base rate 6 Mbps in 802.11a network. They will go farther than the data packets traveling at higher data rates. Hence, it may happen that even though a node B exists in neighbor table of another node A but A is not able to talk to B directly. Therefore, we believe that a control over broadcast rate is beneficial.

We notice that of the many rate control algorithms available in the literature, only few talk about the real implementation issues. These have laid the foundation of the real-world rate control capabilities in actual wireless chipsets such as Atheros [5] AR5xxx. Net-X architecture was implemented using AR5212 chipset and we shall be referring to the same for the rate control framework described in this thesis. However, the changes are generic and can be exported to other drivers as well.

As discussed in [6], first generation 802.11a based radio interfaces implemented a majority of the 802.11 protocol in hardware. Recent wireless hardware chipsets implement time

insensitive parts of the 802.11 MAC protocol in software and perform only time critical tasks in hardware. The software and hardware components of such interfaces communicate using a hardware abstraction layer (HAL). This approach to interface design is also known as the "thin" MAC approach, where the components of the MAC protocol implemented in hardware are kept at a minimum (thin). The wireless interfaces that we used in our testbed were based on such a design. Since we introduce the userspace control over rate selection, we need to make changes to the device driver so that it can understand the user policy instructions. These shall be discussed in detail in Chapter 6.

We would like to point out that we can control transmit power in similar fashion at the user level. The difference in controlling rate and power is that switching rate does not require resetting the card whereas changing the transmit power does. This makes power control costlier in terms of latency. We have provided a minimal framework for choosing the transmit power at the user level application and it is desirable that transmit power levels are defined for longer periods of time. This simplistic framework just exposes the necessary application programming interface (APIs) to the user to switch the transmit power and can be utilized by an intelligent policy which switches the transmit power carefully based on some feedback. On the other hand, switching transmit rate is less disruptive and can be done even on a per-packet basis. In this thesis, it has been our primary focus. We have designed a complete rate control framework including a controlled co-ordination between user level and the driver. The details shall be described in the Chapter 4.

Our architecture and implementation is portable across various rate control algorithms in the driver. For example, Net-X uses Madwifi [7], the Linux kernel device driver for Atheros chipsets. Our rate control framework is also built using Madwifi which currently supports three rate control algorithms. We have implemented the framework such that it can be used with any of these algorithms. The implementation level details are described

in Chapter 5.

The organization of this thesis is as follows. Chapter 3 describes the background of IEEE 802.11 standard and Net-X that is relevant to our work. We talk about the existing algorithms that address the rate adaptation and their shortcomings in Chapter 2. Then, we present the architectural details of our system in Chapter 4. It also discusses various design choices and our decisions. In Chapter 5, we present the implementation details of our system describing the interactions between the key components. In Chapter 6, we describe the modifications made to the Madwifi device driver. We present some results and analysis from the experiments done in our Net-X testbed in Chapter 7. Finally, we conclude and discuss future work in Chapter 8.

CHAPTER 2

RELATED WORK

A lot of algorithms have been proposed to perform rate adaptation. Auto Rate Fallback (ARF) [8] is the earliest work in this area. In this, a sender probes a higher transmission rate when a certain number of transmissions succeed at current rate. On failure, it switches back to a lower rate. The main drawback of this approach is that it fails to adapt reasonably with channel quality. It waits for 10 or so consecutive successes to try a higher rate and it reverts to a lower rate more quickly. Also, it does not stop probing when the channel is stable, leading to expensive retransmission attempts at higher rate every 10th packet.

Adaptive Auto Rate Fallback (AARF) [9] overcame these shortcomings of ARF by introducing dynamic adaptation of threshold used to decide when to probe at higher rate. It uses Binary Exponential Backoff mechanism for this adaptation. When a transmission fails, it not only switches to next lower rate but also doubles the threshold for switching to higher rate. This improves the performance over ARF by preventing unnecessary probing and spending too much time on retransmissions. RBAR [10] is another popular algorithm in literature but is impractical to implement due to its incompatibility with existing IEEE 802.11 standard implementation. It has been mostly used as a performance reference.

From the implementation point of view, few algorithms are used in existing wireless device drivers. The most common ones are Sample Rate [11], Onoe [7], and Adaptive Multi Rate Retry (AMRR) [9]. All of these are a part of current implementation of Madwifi wireless driver. AMRR extended the original Multi Rate Retry (MRR) mechanism found in older Madwifi. Madwifi allows up to 4 different rates to be used for any transmission. There are FIFO queues of transmission descriptors for scheduling transmission packets. Each descriptor contains 4 pairs of rate and count fields (r, c). The packet is first sent at rate r_0. If it fails, it is tried at same rate for $c_0 - 1$ times. If it still fails, rate r_1 is tried c_1 times and so on. This handles the short term variations. In long term, (r, c) pairs are changed periodically. AMRR added the capability to dynamically adapt this period and thus, improves the performance.

Onoe is credit based algorithm which evaluates the credits per second. It ultimately selects the highest rate which gives less than 50% loss. Credits are maintained per destination for current bit rate. If less than 10% packets are retransmitted at the current rate, the credit points are increased else decreased. Whenever the credits reach 10, the rate is switched to next higher. If a rate fails, it is not probed till 10 sec. This conservative approach makes it less sensitive to individual packet losses.

Sample Rate is the newest and default algorithm in Madwifi. It maintains a set that contains all rates except the ones which have experienced 4 successive failures or whose loss-less transmission time is higher than the transmission time of current rate. It probes a higher rate randomly from this set every 10th packet. It calculates average transmission time for various rates over 10 second intervals and chooses the rate with lowest average transmission time. It allows the probing of random rates instead of switching to only the next higher or next lower rates. It also uses few strategies such as not probing 9 Mbps since 9 Mbps never performs better than 12 Mbps.

It can be observed that most algorithms try to maximize the throughput. There are few approaches which look into the other problems such as one of hidden terminals causing collisions that trigger rate decrease [12]. Some works have looked to specifically improve real-time capabilities in multimedia applications [13, 14]. To the best of our knowledge, there are very few works that integrate the application needs with the lower level rate control algorithm. Application directed automatic 802.11 rate control [15] provides a design to allow the application to steer the rate controlling algorithm. They use FER feedback from the radio and provide the application with an interface to modify the percentage of packets to be probed at higher or lower rates. Their control structure can be used as a guideline to couple the application requirements and the link level rate control process. However, they have no concrete implementation or results.

In the field of power control implementation, Kawadia et al. [16] modify the socket buffer, the *skb* structure in Linux kernel networking stack, to include a field specifying transmit power to be used for the packet. This approach, though efficient, needs modifications in Linux kernel networking stack which we wanted to avoid in our system. It is noted that Net-X does not modify the networking stack in the Linux kernel.

CHAPTER 3

BACKGROUND

In this chapter, we discuss the features of 802.11 standard relevant to rate and power control. We take a brief look on rate control mechanism in Madwifi and discuss the background of original Net-X architecture pertinent to our work. We finally present an overview of Net-X testbed deployed indoor in our office building.

3.1 802.11 background

IEEE 802.11 standard supports both multiple rate and multiple power values. We shall discuss the physical (PHY) layer features corresponding to rate and power control separately. At the medium access control (MAC) layer, it uses the medium access mechanism as defined by carrier sense multiple access with collision avoidance (CSMA/CA). The 802.11 family had 3 main amendments namely a, b and g. 802.11b uses the 2.4 GHz band. Being an unregulated band, 802.11 devices can encounter interference from other appliances using the same band. 802.11a supports bandwidth up to 54 Mbps and signals in a regulated frequency spectrum around 5 GHz. This higher frequency compared to 802.11b limits the range of 802.11a networks. The higher frequency also means 802.11a signals have more difficulty penetrating walls and other obstructions. Because 802.11a and 802.11b utilize different frequencies, the two technologies are incompatible with each other. 802.11g supports bandwidth up to 54 Mbps, and it uses the 2.4 GHz band for

greater range. 802.11g is backwards compatible with 802.11b. We will see the rate control specific MAC layer functionalities in the next subsection.

3.1.1 Rate adaptation

802.11 inherently supports per-packet rate control. The PHY layer sends a PHY layer convergence procedure (PLCP) preamble and header before every data packet which contains the rate that will be used to transmit the packet. This is sent at the fixed base rate (1Mbps for 802.11b and 6 Mbps for 802.11a/g). Whenever any other node detects the transmission, it tunes its hardware to the base rate to receive the preamble. After receiving the PLCP information, it tunes its hardware to the rate specified for the data payload. PLCP can be considered as a unicast overhead.

Table 3.1 SNR requirements for 802.11a and 802.11b rates for BERs less than 1E-5 [17].

802.11 *std*	$Rate(Mbps)$	$SNR(dB)$
b	1	-2.92
b	2	1.59
b	5.5	5.98
b	11	6.99
a	6	6.02
a	9	7.78
a	12	9.03
a	18	10.79
a	24	17.04
a	36	18.8
a	48	24.05
a	54	24.56

The multi-rate capability is supported in IEEE 802.11 by dynamically selecting the most appropriate modulation and channel coding mechanisms, which decide the bit-rate for transmission. Denser encoding of data bits results in higher data rates, i.e., they are more bandwidth efficient since they pack more data bits per signal pulse (symbol). However, faster rates require higher signal to noise ratio (SNR) for a given bit error rate (BER)

as shown in Table 3.1 [17]. Thus, there is a trade-off between throughput and error rate when choosing a particular bit rate. In general, transmission range and performance of any bit-rate are influenced by noise, transmit power and interference. All these factors should be considered while evaluating the performance of any system.

Originally IEEE 802.11 DSSS (direct sequence spread spectrum) had only 1 Mbps or 2 Mbps rates supported. Later, two high rate extensions for 802.11b were defined and this allowed for additional rates of 5.5 Mbps and 11 Mbps. Also, 802.11a was introduced in the 5 GHz band, which is based on OFDM (orthogonal frequency division multiplexing). This supported data rates up to 54 Mbps. The latest addition is 802.11g which extends 802.11b PHY to support rates up to 54 Mbps in 2.4 GHz band. The standard rates are summarized in Table 3.2.

Table 3.2 802.11 rate summary.

802.11 *std*	*radio modulation*	*modulation*	$bit-rate$
b	DSSS	BPSK	1
b	DSSS	QPSK	2
b	DSSS	CCK	5.5
b	DSSS	CCK	11
a/g	OFDM	BPSK	6
a/g	OFDM	BPSK	9
a/g	OFDM	QPSK	12
a/g	OFDM	QPSK	18
a/g	OFDM	QAM-16	24
a/g	OFDM	QAM-16	36
a/g	OFDM	QAM-64	48
a/g	OFDM	QAM-64	54

In 802.11 MAC layer, there is a mechanism of link-level retransmissions to provide reliability and hide the packet losses from upper layers. The destination is supposed to send an acknowledgment (ACK) on successfully receiving a packet. If the sender does not receive the ACK (either due to packet failure or ACK loss), it has to retransmit the packet. However, the caveat is that it has to keep doubling its back-off interval for

every ACK loss. Therefore, it becomes very expensive after few retries. This reinforces the need of a good rate selection algorithm to prevent packet losses and retransmissions. Sometime, the back-off penalty may exceed the transmissions time itself.

3.1.2 Power control

Power control is a cross-layer problem. It affects the PHY and MAC layers in non-trivial fashion.

1. Transmit power affects the signal quality at the receiver.
2. It determines the transmission range as well as interference caused to the neighbors.
3. Choice of power settings on different nodes (if not uniform) may lead to loss of symmetry and bi-directionality. This, in turn, affects the basic functioning of mechanisms like MAC level ACKs, which assume bidirectional links.
4. It directly affects the contention at MAC layer since higher the transmit power, higher is the interference.
5. It can also cause disconnections in the network because some links may break at lower transmit power levels.

While these are the fundamental protocol issues affected by power control, changing of power levels have practical repercussions too. It causes resetting of the wireless card which introduces delay and loss of buffered packets. Therefore it necessitates a careful design of power control mechanism.

3.2 Madwifi support

Madwifi stands for Multiband Atheros Driver for Wireless Fidelity (WiFi). It is an Open Source device driver for *Atheros* wireless chipsets. Before going into details, it is necessary to first understand the Madwifi driver. It is divided into four main parts.

- *Hardware abstraction layer (HAL)* - This is the lowest level of detail exposed to us. For example, commands such as *reset device* are translated to device and chip-specific calls and delivered to the device via this layer.

- *IEEE 802.11 stack* - It has been written entirely in software. This portion of the driver handles 802.11 protocol-specific functionality which is not time-critical. For instance, all scanning, association etc. are performed by this part of the driver.

- *The ath module* - It glues the 802.11 stack with the HAL. This part takes care of complexities related to the device interrupts, buffer management, beacon generation, etc.

- *The rate control algorithms*- It does the rate adaptation inside the driver. At driver load time, one can select the rate control algorithm to be used by choosing the correct module to be loaded.

Madwifi supports 802.11 stack ioctl calls for setting/getting transmission rate and transmit power for an interface and these can be used on a per-packet basis as well. These are the standard ioctls relevant to this thesis:

1. *ieee80211_ioctl_siwrate()* - set rate

2. *ieee80211_ioctl_giwrate()* - get rate

3. *ieee80211_ioctl_siwtxpow()* - set txpower

4. *ieee80211_ioctl_giwtxpow()* - get txpower

Madwifi currently supports three rate control algorithms-

1. AMRR

2. Onoe

3. Sample Rate

The details of these algorithms are discussed briefly in Chapter 2. We are mainly interested in the way Madwifi supports various rate control mechanisms. The `ath` module exposes an interface (`if_athrate.h`) which a rate control algorithm has to adhere to. It defines certain functions like `ath_rate_findrate()`, which each algorithm implements independently. Such a clean interface makes it easy to plug-in new algorithms.

3.3 Net-X background

Since this thesis is a continuation of Net-X work, it is important to be familiar with its original architecture and implementation. We are presenting a brief summary here. For details, the reader is referred to the report [1]. Figure 3.1 is taken from [1] to show the multichannel implementation architecture of Net-X.

3.3.1 Channel abstraction layer

Channel Abstraction Layer (CAL) operates between the network layer and device driver, but logically belongs to the link layer. Its fundamental utility is handling multiple channels and abstracting multiple physical interfaces into a virtual interface to avoid higher layers having to manage multiple interfaces. Since a similar feature of virtualizing the interfaces was available in *bonding driver*, already a part of Linux, CAL functionalities were integrated into the bonding driver. The key components of CAL are shown in Figure 3.1:

1. *Unicast component*

   It is the most important part from the viewpoint of our thesis. Originally it specifies the channel and interface needed to reach a neighbor. A typical Unicast table would look like Table 3.3.

   The thing to be noted is that Unicast table does not contain the next hop neighbor IP or MAC but the remote destination's IP. Though it seems counter-intuitive, the choice comes from another choice of keeping the Linux-kernel unmodified. To

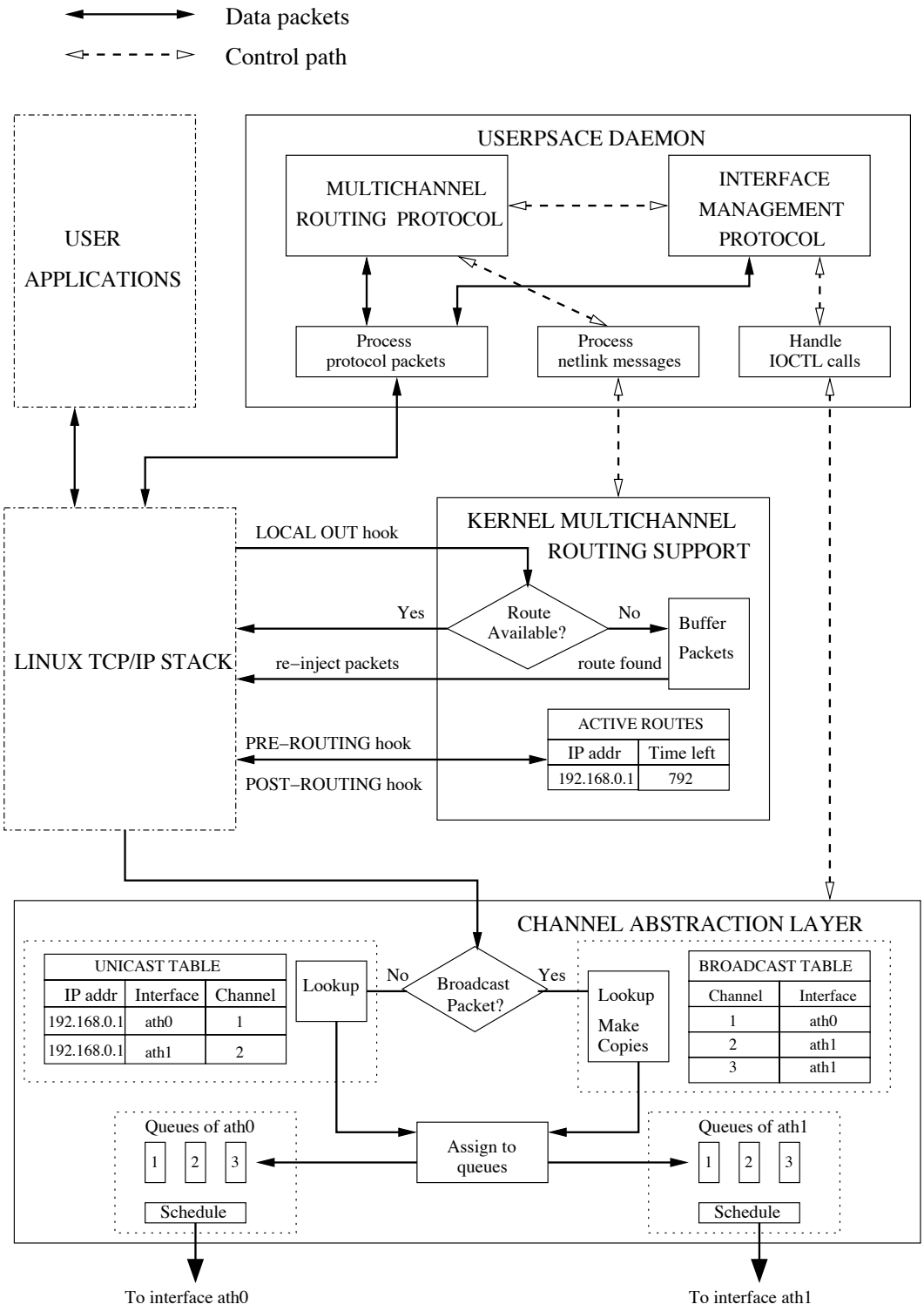

Figure 3.1 Net-X architecture for implementing multichannel protocols. The figure assumes that two interfaces, "ath0" and "ath1" are available. It is taken from [1].

Table 3.3 Unicast table original structure.

Dest. IP	*Interface*	*Channel*
192.168.30.1	ath0	1
192.168.30.3	ath1	2

understand this, we shall consider the flow of packets through Linux network stack. By the time a packet comes down from the IP layer, it only has the information of destination IP and next hop MAC address (unless one modifies the stack itself). This is the only information available in the packet when it reaches the CAL. It may be argued that next hop MAC address is a better choice since CAL belongs to the link layer. But the userspace daemon functions such that it populates the unicast table on receiving the *hello* packets and it does not have the information of neighbor's MAC addresses.

Therefore, it is a trade off that was made while designing the Net-X architecture. It is important to understand the assumptions and reasons why choices were made because the original architecture decided many of the choices we made in this thesis.

To continue the discussion on the unicast component functionality, when a unicast packet reaches the CAL, it is handed over to this component. The destination IP is looked up in the table to identify the channel and interface required. Accordingly, the packet is enqueued for subsequent transmission. It supports ioctl calls mentioned in Table 3.4.

2. *Broadcast component*

   It specifies the list of channels and corresponding interfaces on which the broadcast packets have to be sent.

3. *Scheduling and queuing component*

   Since switching channel for every packet to be transmitted is highly inefficient, a mechanism was developed to enqueue the packets on a per-channel basis and for

Table 3.4 List of ioctl calls exposed by CAL.

Ioctlcall	*Function*
AddValidChannel	Specify the channels that may be used by an interface.
UnicastEntry	Add, update, or modify an entry in the unicast table.
BroadcastEntry	Add or remove an entry in the broadcast table.
SwitchChannel	Explicitly switch an interface to a new channel.
GetStatistics	Return per-channel usage statistics.

scheduling the switching process. This component supports the interface switching by buffering packets when needed and scheduling switching of the channels.

3.3.2 Kernel multiple channel routing support

Kernel Multiple Channel Routing (KMCR) is the module which provides on-demand routing for the multichannel protocol. It communicates with the Linux TCP/IP stack through Netfilter hooks. It provides the support for buffering the packets when a new route discovery is required. It communicates the need of route discovery to the userspace daemon using netlink messages.

3.3.3 Userspace daemon

It implements the less time-critical components of higher layer protocol by utilizing the features exported by CAL and KMCR. It handles initialization, hello packet communication and route discovery/maintenance. We will not be discussing the details of the multichannel protocol implemented using the userspace daemon. From our viewpoint, userspace daemon corresponds to any application (multi or single channel) using our rate control framework. It makes the ioctl calls to the CAL to keep the unicast and other tables updated. It communicates with KMCR for routing support. Our interest lies in its interaction with CAL only. When it receives hello packets containing the fixed channel information from its neighbors, it adds/updates the corresponding entry in the unicast table in CAL. Absence of hello messages lead to timing out the unicast table entries.

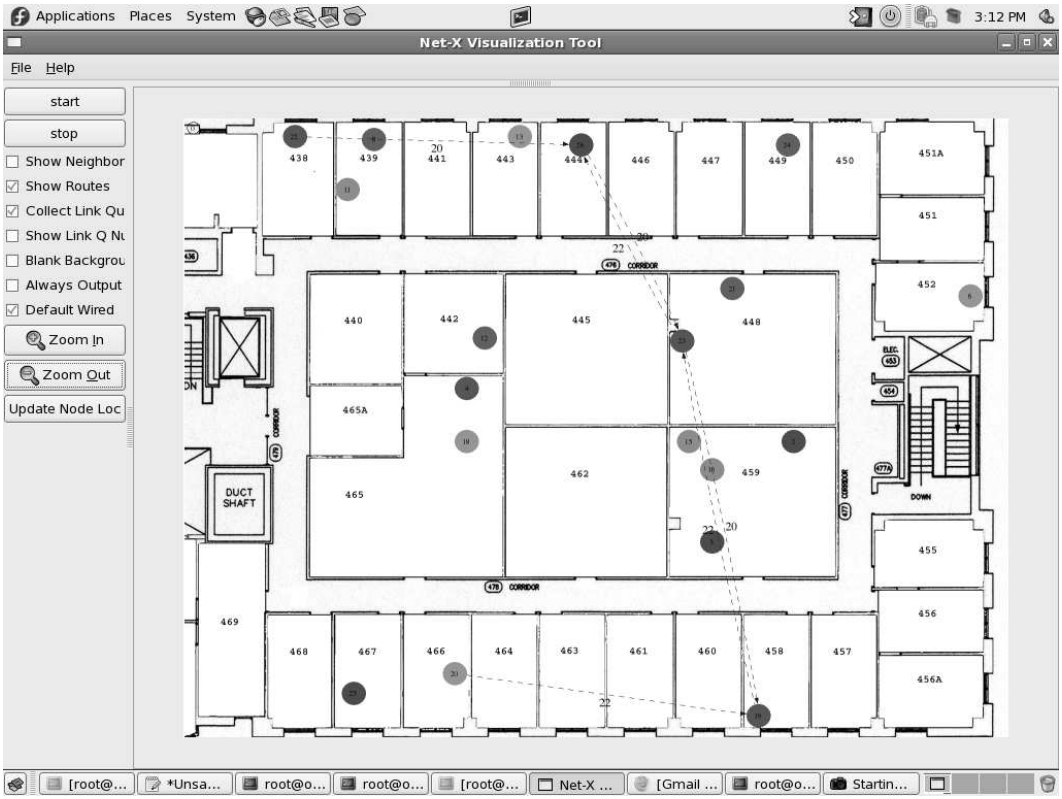

Figure 3.2 A map of 4th floor indoor network showing the testbed nodes location.

Since hello packets can contain one-hop neighbor information of the sender, remote entry information can also be added to unicast table(for which the sender was next hop). This explains the presence of final destination IP information in the CAL unicast table.

3.4 Net-X testbed deployment

Net-X multichannel protocol has been implemented in a testbed consisting of 26 nodes currently deployed on fourth floor at Coordinated Science Lab (CSL), UIUC. The nodes comprise of *Net 4521* Soekris [18] boxes. All nodes are equipped with two wireless interfaces (one pcmcia and one mini-pci interface each). The wireless cards are based on *Atheros* [5] chipsets. We use 256 MB compact flash cards for hard-disk.

Figure 3.2 shows a snapshot of the testbed with 19 nodes up and running. The links and routes are visualized dynamically using the Net-X visualization tool developed for Net-X system. Each circle represents a node and shows its position in the deployment. For clarity, the individual links are not shown. The dotted lines represent the hops in an active multi-hop route. Though the thesis mainly talks about implementation of rate control framework, a substantial time was devoted to have a stable deployment of the testbed. We used *pebble* [19], which is a compact debian-based Linux distribution specifically designed for embedded devices on a *vanilla* 2.4.26 kernel. Each node has watchdog installed to initiate the automatic reboot if it crashes. Currently we run the testbed on *802.11a* standard to prevent interference with *CSL wireless* which uses *802.11b*.

CHAPTER 4

SYSTEM ARCHITECTURE

In this chapter we motivate the need for new architectural support in existing Linux kernel for facilitating our two-level rate control framework. We also discuss various design alternatives and our final choices in implementation.

The main goals of Net-X Rate Control Framework are as follows:

1. Provide the user with a control over data rates in the system.
2. Couple the rate control at the driver level with that at the application level, thereby, making the rate control process application-aware.
3. Keep the rate control framework transparent to the rate control algorithm being used at the driver level.
4. Keep the interface and design generic so as to facilitate the control of other interface capabilities such as carrier-sense threshold, RSSI threshold, etc., in the future.

4.1 Rate control design alternatives and decisions

The original architectural design of Net-X system was presented in Chapter 3. Our framework is integrated in the existing Net-X architecture and it abides by many of the original design choices. Any operating system is divided into multiple address spaces. The main divisions are "kernel space" and "user space," which are differentiated by the

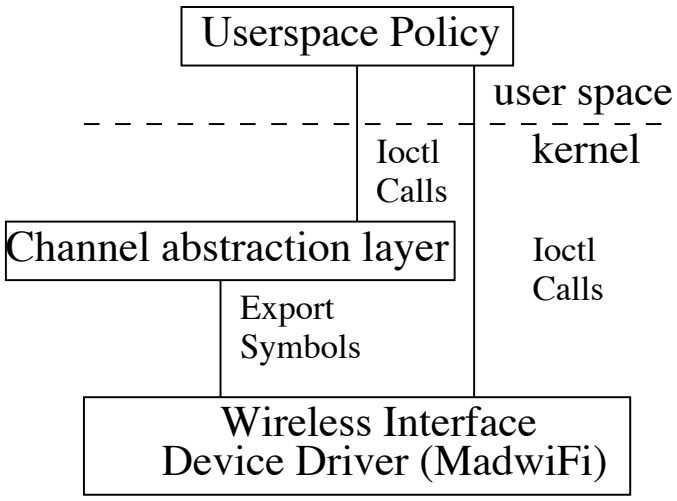

Figure 4.1 Proposed architecture for the rate control framework.

privilege level. While programming in user-space is easier, it is inefficient and infeasible to implement complex systems entirely in it. On the other hand, implementing things in kernel might be harder but is speedier and more efficient. Net -X implements most of the system in kernel space. This also provided better timing response when implementing the queuing and switching module in channel abstraction layer (CAL). The need for communication between user-space and kernel was fulfilled using ioctl and netlink library calls. We stick to this design in our architecture, as will be discussed shortly.

Our proposed architecture is shown in Figure 4.1. The main components that needed modification in the kernel are channel abstraction layer (CAL) and the wireless device driver, Madwifi in our case. We needed a new form of communication between the modules implemented within the kernel space itself. This was achieved using *export_symbols* mechanism. The details will be discussed in Chapter 5. Userspace policy is the implementation of any protocol that uses two-level rate control functionality. The two kind of ioctls shown are for setting the unicast minimum and maximum rates using CAL and broadcast rates directly to the wireless driver.

The features needed to implement unicast rate control in our Rate Control Framework are as follows:

1. **User level rate adaptation:** Desired rate range has to be selected at the user level. This can take into account, the application demands and other factors such as topology.

2. **Updating the driver:** The specified rate range has to be communicated to the wireless device driver.

3. **Driver level rate adaptation:** Final per-packet bit rate selection for the unicast packets at the driver.

We shall discuss requirement and design possibilities for first two points separately. Third part constitutes the modifications needed in the driver itself and shall be described in Chapter 6.

4.1.1 User level rate adaptation

The motivation for user level control over rate selection comes from the argument that different applications have different demands in terms of performance metrics and a common rate control algorithm at the driver level trying to maximize a certain metric (such as throughput) irrespective of application is undesirable. Researches such as done by Yang et al. [3] showed that the throughput achieved for a given topology are a function of rate and carrier-sense threshold combination. The topology knowledge can help in selecting the appropriate operating point in terms of carrier-sense threshold and transmission rate. Such protocols cannot be implemented using existing approaches.

The earlier discussion clearly identifies the need for a higher level rate control mechanism. Next, we consider few options for the basis of user level rate control:

- Rate control based on next hop. Doing rate adaptation on a link basis is justified since the link quality to next hop decides how much data can be pumped to a

remote destination. This is the approach taken in most of the driver rate control algorithms which try to maximize the throughput at each individual link and thus, hope to achieve the maximum aggregate throughput in the network .

- Rate adaptation based on final end destination. This is the ideal scenario where we try to meet the possibly different requirements of each remote destination irrespective of next hop itself. Though, none of the existent algorithm take this approach, it may be advantageous to provide such a capability in future. This needs more analysis and is considered as a possible extension in future work.

Of the two possibilities, we choose the first one since Net-X uses similar notion in selecting the channel to reach a remote destination based on the fixed channel of next hop. Deciding on how precise the rate selection should be, there are following choices for the design:

1. Do a very precise rate selection, i.e., determine the exact value of rate to be used. This approach is easily implementable through standard ioctl calls exposed by IEEE 802.11 stack in the Madwifi driver (`80211_ioctl_siwrate`), but there are drawbacks. It overrides the driver rate control completely and is undesirable at user level because we do not have the sufficient knowledge of the channel quality, transmission statistics and some other factors that help the driver in adaptive rate selection. Duplicating the functionality of already existing intelligent link layer algorithms at higher layers is neither feasible nor intuitive.

2. It became evident that we want to choose a set of rates instead of one particular rate. A natural choice is to select a subset of the rates supported by the IEEE 802.11 standard in use. Either a non-contiguous set of rates can be chosen or a contiguous set. For example, 802.11a supports {6, 9, 12, 18, 24, 36, 48, 54} Mbps rates. We can pick either a non-contiguous subset such as {6, 18, 36} or a contiguous subset like {18, 24, 36, 48}. We did not see any particular advantage in selecting the first option and to improve simplicity, we decided upon the latter. The advantage with

this scheme is that user needs to specify only the minimum rate and the maximum rate.

The rate control of broadcast packets is simpler and only a single value of rate needs to be passed to the device driver. This process is independent of actual destination or next hop or any other parameter. So, a user-defined ioctl call directly to the wireless driver can be used.

It is understood that the above discussion determines only the design of the framework and a policy is needed to utilize it intelligently to the benefit of particular application or network.

4.1.2 Bonding driver

Once the user level policy makes the decision about the rates, the necessary information needs to be passed on to the wireless driver. We have following options for this:

1. Making calls directly from the user space to the device driver. This is not entirely feasible since driver uses link level information such as mac addresses whereas this information is not available at user level. We talk about the driver details in Chapter 6 which explains why this approach cannot be used. This method is suitable for passing global information such as setting broadcast rate or transmit power but not for updating data on a per next-hop node basis.

2. Pass rate information inside the packet through modification of *skb* data structure inside Linux kernel itself. This required kernel modification which we wanted to avoid and hence, we rule out this option even though it would have been the most efficient way of implementing the things.

3. Use an intermediate layer such as Channel Abstraction Layer in Net-X. This is the choice we make to adhere to Net-X architecture.

Since we already had *bonding driver* as part of Net-X architecture, we chose to develop rate adaptation as extensions to it. The bonding driver is implemented as a loadable module in the Linux kernel; thus kernel recompilations and unnecessary system reboots can be avoided, easing the development and testing process. Our system is closely coupled with the functionality of the bonding driver and its role in original Net-X, so we must present its functionality before explaining our extensions.

Bonding driver provides functionality to abstract multiple interfaces as one interface in Linux kernel. This driver was mainly developed for Ethernet (802.3) based devices to perform load balancing (striping), fail over, adaptive shaping, etc. The bonding driver exposes a single virtual interface and has the ability to "bond" together many interfaces into one. It also has a set of user space tools which can bond or debond real networking devices together. The bonding driver is well integrated into the Linux kernel and has been made part of the main kernel source tree.

The bonding driver is a dummy networking interface not linked to a real networking device. A bonding device can be invoked by loading the *bonding.o* module which creates a virtual interface, named starting from *bond0* onwards. There can be multiple instances of the bonding driver, by loading the bonding module multiple times. The bonding driver has a user space helper program, known as *"if-enslave,"* which can *bond* multiple real network interfaces into one. The devices which are bonded together are referred to as a *slave* devices, and the process of bonding them together is also known as *enslaving* the devices. Once enslaved, a *slave* flag is set up in the real devices' kernel data structure to mean that the devices are now *incapable* of receiving any data *directly* from user programs. Any data to be transmitted using these devices has to be routed via the bonding device. Typically each network interface has one or more network addresses assigned to them. However, once enslaved, the network addresses are automatically released, and all devices will now share the same network address or addresses assigned to the bonding device. All bonded devices also share the same 48-bit MAC address. The first slave's

(first device to be enslaved) network address is used for all other enslaved devices.

We talked about Channel Abstraction Module (CAM) in Chapter 3. It resides between the network layer and the interface device drivers. In the Net-X architecture, CAM consists of the Unicast component, Broadcast component and Scheduling & queuing component.

Our interest lies in the Unicast component mainly since it already contains one of the physical parameter needed to reach a destination, i.e., the channel information. It was an ideal place to add the required rate information per destination. As discussed in Subsection 4.1.1, we maintain a minimum rate and maximum rate for each destination. It is pointed out that even though information is maintained per remote destination, in reality it is defined only by the next hop neighbor. This is same as the notion of channel needed to reach a remote destination in original Net-X architecture since that also depended only on the channel of next hop node. Thus, all remote destinations with same next hop neighbor use same channel and *minmax* rate information.

Though we populate the transmit power entries in Unicast table, we have not developed the hooks in wireless driver. We discuss power control architecture in Section 4.2. We saw the older unicast table design in Table 3.3. The new unicast table will look like Table 4.1.

Table 4.1 New unicast table structure.

Dest. IP	*Interface*	*Channel*	*Min rate*	*Max rate*	*Txpower*
192.168.30.1	ath0	1	6	36	65
192.168.30.3	ath1	2	18	54	50

The unicast table is populated by a user space protocol via ioctl calls (entries can be added, deleted or updated). Though we do the rate control for broadcast packets, we did not need to change the broadcast table since it contains the channel to interface mappings

which remain as before. Besides, we do the broadcast rate control directly between the user space and the wireless device driver using user-defined private ioctl calls in Madwifi.

When the channel abstraction module receives a unicast packet from the network layer, it hands the packet off to the unicast component. The destination address of the packet is looked up in the unicast table to identify the channel and interface to use for reaching the destination. The rate range required for the destination is also looked up and passed on to the driver. After this, the packet is handed off to the queuing component for subsequent transmission.

For passing on the rate information to the driver, we need inter-module communication since both bonding driver and wireless device driver are loaded as modules in the kernel. Ioctl calls work only between user-space and kernel-space. We used the mechanism of *exporting symbols* for this. We will talk about the details in Chapter 5.

4.2 Power control architecture

Power control framework can be designed on same guidelines as the rate control architecture. Doing a per-packet power control is expensive compared to the per-packet rate control. This is because the switching latency can be very high due to the resetting of card required for changing the transmit power. It is a similar problem to changing a channel on an interface. Net-X dealt with the channel switching problem by using queuing mechanism, i.e., it maintains queues for every channel and process all the packets enqueued for a channel in one schedule. Doing a similar thing for transmit power would have required sub-queues and may have complicated the design. We were not sure if the extra queuing overhead was justified and we decided not to go for per-packet power control. However, we believe that card resetting required for changing power is a firmware limitation and not a limitation of the electronics since fast power control is already used commonly in code division multiple access (CDMA) networks. So we are hopeful that

this limitation would be eliminated in future and the power control mechanism would be more usable.

We have currently provided the support for setting transmit power at the user space. Also, we have extended the unicast table to hold the information for transmit power per destination but currently, we are not using it to change the power settings in the driver. We are leaving it as a hook that can be used in future designs of detailed power control for Net-X.

CHAPTER 5

IMPLEMENTATION

Our rate control framework is implemented on a Linux-based testbed that runs the Net-X system. We shall explain the implementation architecture in this chapter. The current support for power control will also be discussed.

5.1 Implementation architecture

The rate control framework implementation architecture is shown in Figure 5.1. The figure does not show details of KMCR and broadcast components at CAL which exist in Net-X multichannel protocols since these remain same as seen in Chapter 3. The main components of rate control architecture are:

- **Userspace policy:** This implements the less time-critical components of any protocols that utilize two-level rate control provided by our framework.
- **Channel abstraction layer:** As discussed earlier, this kernel component abstracts the details of multiple interfaces from the higher layers. It is used to maintain rate information on a per next-hop basis and communicate it to the driver.
- **Customized wireless device driver:** This constitutes the modification made to the wireless driver for supporting the rate control mechanism.

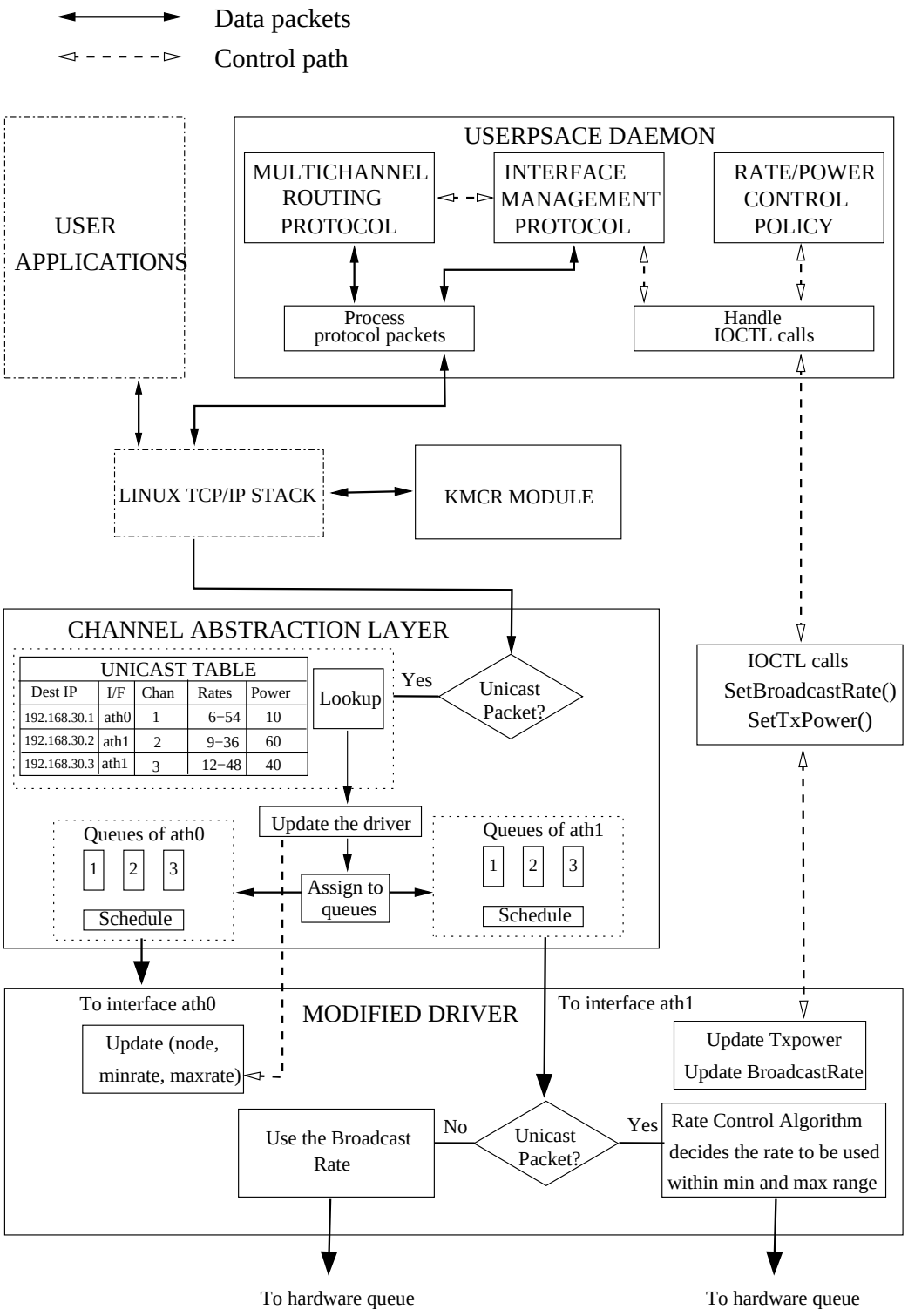

Figure 5.1 Architecture for implementing rate control framework. The figure only shows the main components of proposed rate control framework. It assumes two interfaces 'ath0' and 'ath1'.

5.2 User space interaction with the channel abstraction layer

In this section we present the details of relevant ioctl calls used by the user space protocol to communicate with channel abstraction module. "Ioctl" is an acronym used in operating systems to informally mean "Input Output Control." An ioctl call from user space is sent to a specific device, identified by an ioctl number, along with an user request. Since our channel abstraction module is built into the bonding driver, which exposes a pseudo-networking device, we found ioctls to be a suitable communication mechanism with user space applications.

In the Linux kernel, each device can define and implement up to 16 private ioctls. Original Net-X implemented five new private ioctls in CAL. For the rate control part, we extended one of these:

- *Unicast Entry:* This ioctl is used to set up the unicast table. We added the interface capabilities that can be exploited to reach a particular destination. These interface capabilities include transmit rates (defined by minimum and maximum rates) and transmit power. The user policy can decide upon the suitable values for these and the unicast table can be populated accordingly. Unicast entries can be added, deleted, and updated using this call. Recall that the Destination IP address is the unique key for the unicast table. When an IP address passed is not found in the table, and the channel and interface are valid, a new entry is created. If the address passed is already found in the table, the new channel and interface information is used to update the entry. A unicast entry is deleted when its IP address is passed with a blank interface name.

  For the reasons discussed in Chapter 4, the rate and power information is, in reality, pertinent to the next hop neighbor rather than the end destination. It is implied

that the rate and power control are tied down with the next hop neighbor and hence, all the end destinations with same next hop will be governed by same min and max rate values and transmit power value.

5.3 Channel abstraction layer interaction with wireless driver

In this section we will discuss the interaction between the channel abstraction module and the wireless device driver, Madwifi in our case. The channel abstraction layer has the necessary rate information per next hop neighbor, which needs to be conveyed to the wireless driver. To decide when to update the Madwifi, following options can be considered.

- Ideally, the driver should be updated whenever the unicast table itself is updated from the user space policy. User space policy uses the end destination IP address but we need mac addresses of next hop neighbor to update the wireless driver since it is not IP aware. An alternative is to use reverse address resolution protocol (rarp) lookups to map the next hop IP (which is known to us) to the next hop mac address. But *rarp* is not available by default on all systems, incurs a lookup overhead and possibly can generate *rarp* requests if the entry is not in the cache. To avoid these, this approach is ruled out.

- The node's mac address could be passed inside the *hello* packets. However, this approach breaks up the layering paradigm and it is not desirable to hard code hardware addresses inside the protocol packets.

- The third approach is to update the driver only when we have a packet to transmit. Once the packet reaches the CAL, it contains final destination IP address and next hop mac address. The corresponding destination IP is looked up in the unicast table, the next hop mac address is picked up from the packet and passed to the

Madwifi along with the *minmax* rate information. This approach prevents *rarp* overheads and updates the driver only when needed. Therefore, it is used in our implementation.

This process of updating the driver incurs some extra overhead and therefore, we do the updates periodically instead of on every packet. We use a period of 20 packets in our current implementation. Also, further optimization is done by using a 'driver-updated' flag in unicast table. Whenever *minmax* information for some next hop neighbor corresponding to a remote destination is updated in the driver, all the entries with same next hop neighbor are marked as updated. This is because in current architecture, *minmax* information pertains to next hop neighbor only.

The communication done by CAL to update the driver cannot use ioctl or netlink calls that were used between user-space. The interaction between CAL and Madwifi is different because both are loaded as kernel modules. We use another form of communication, which is suitable for modules. In this, we make Madwifi expose a function interface and export it through the macro EXPORT_SYMBOL so that bonding driver can call it when needed. Thus, we implemented the function `ath_handle_unicast()` in Madwifi which can be called from the bonding driver. The details are discussed in Chapter 6.

5.4 User space interaction with wireless driver

We use direct interactions between the user space and the Madwifi for setting up the transmission rate for broadcast packets and transmission power. The transmission power can be set using a standard ioctl call `80211_ioctl_siwtxpow`. However, for setting the broadcast rate, we implemented a private ioctl in Madwifi. Details will be presented in Chapter 6.

CHAPTER 6

DRIVER MODIFICATIONS

In this chapter we discuss the modifications that were made to the wireless driver used in our testbed. These changes are not merely optional but mandatory to make the driver compatible to the rate control framework design.

Wireless device driver is mainly concerned with whether an interface is using fixed rate or it supports multi-rate features. In first case, it simply uses the single fixed rate for all transmissions irrespective of link quality. A fixed rate can be set for an interface using the `iwconfig` command or by using a standard ioctl `ieee80211_ioctl_siwrate` at the user level application. If the fixed rate option is not specified, the driver uses its own rate control algorithms to determine the best rate per packet. In our rate control framework for Net-X, we wanted the driver to support multi-rate within a given subset and hence, it needs suitable modifications to the driver.

Our power control architecture does not require any changes in the driver since it uses a standard ioctl exposed by the Linux kernel, i.e., `ieee80211_ioctl_siwtxpow`. The user level application can change the transmit power value using this ioctl irrespective of the driver. Hence, the discussion that follows in this chapter is pertinent only to the rate control architecture.

The wireless interfaces used in Net-X testbed make use of the "Madwifi" open source driver, and we use the same. Therefore, our modifications are based on this driver. We have not yet looked at the feasibility of making these modifications to other wireless network drivers. Similar changes should be possible for other devices too which are available in open source.

6.1 Overview of MadWiFi

Madwifi is a partial open source project supported by Atheros. The only closed source part of Madwifi is the hardware abstraction layer (HAL). This comes available only as a compiled binary for few architectures. Other parts of the driver are provided as open source and hence, are ideal for use in research environment. The Madwifi code consists of various modules:

1. `net80211` or `ieee80211` stack - This was originally derived from FreeBSD and implements the generic IEEE 802.11 stack functionality in software. It contains WLAN authentication, beaconing, and cryptographic parts along with other features of 802.11.

2. `ath` module - It describes the Atheros WLAN controller specific callbacks for `net80211` module access to the hardware through `hal` module. It contains critical part of 802.11 management such as beacon management, device's ioctls, configure and setup transmit(TX) and receive(RX) queue, and bus controlling connected with CPU, etc.

3. `hal` module - This is the hardware abstraction layer, responsible to access the hardware. It is a closed source component and acts like the firmware of the card, though is not exactly the firmware since firmware is meant to be hardwired program executable on-board microcontroller. It is not provided as open source to enforce the limit on usage of transmit power, licensed bands of frequency, etc.

4. `ath_rate` module - This is the rate control module and chooses the best transmission rate according to the current rate control algorithm. In the current version, Madwifi supports three rate control algorithms:

   (a) AMRR

   (b) Onoe

   (c) Sample Rate

   We have seen these in the Chapter 2.

As we discussed in Chapter 4, passing the required rate range information to the Madwifi driver from the bonding driver needs a form of inter-module communication. We use the technique of *exporting symbols* from the Madwifi which the bonding driver can use.

6.2 Kernel symbols and modules

Kernel modules are object code, which is added to the kernel at runtime. Once it has been embedded, the module is in the kernel address space. Before the embedding of a module, however, several aspects have to be observed. As the module will probably have to call functions of the actual kernel and want to use its data structures, we first have to resolve the addresses of these functions and data structures. The Linux kernel includes a table, the ksym symbol table, for this purpose. This table includes all required information. Each row of the table contains the name and memory address of a function or variable. Information about the data type or parameters is not saved to the table. A module can access only functions and data structures saved in the kernel's symbol table. Other parts of the kernel are not accessible to a module. This has the benefit that modules cooperate with the kernel exclusively over defined interfaces, as is true for the microkernel architectures.

The instruction EXPORT_SYMBOL(xxx) from the file `kernel/ksyms.c` adds a function or variable of the kernel to the symbol table. From then on, each module can access these

variables or call functions. In addition, modules can export references to functions and variables from the module into the symbol table. The macro EXPORT_SYMBOL can be used to allow modules to export selected function and data pointers into the symbol table of the kernel.

A module can normally access only those symbols that are listed in the symbol table when the module loads. For this reason, a situation where two modules loaded consecutively into the kernel want to access each other's symbols may cause problems. The module loaded first cannot access the symbols of the second module, because they are not yet known. We faced this problem when we wanted to update the bonding driver from the Madwifi. We will discuss this later in subection 6.3.2.

6.3 Minmax rate adaptation

We refer to the design of our rate control framework as *minmax* to emphasize on the information that needs to be passed on to the wireless driver, i.e., min-rate and max-rate. Madwifi maintains the interface device information in a structure `ieee80211com`. This structure maintains the associated neighbor table structure called `ieee80211_node_table`. This table contains the stations associated to the interface device when it is operating in infrastructure mode and contains the neighboring nodes otherwise. The neighboring nodes include all the nodes which can be detected in the channel.

6.3.1 Updating the rate information

Each node information is maintained in a structure `ieee80211_node`, which contains all the attributes of any IEEE 802.11 compliant node. This is the generic class from which `ath_node` inherits. `ath_node` describes an atheros node that maintains some extra driver-specific information apart from the IEEE 802.11 compliant features. There are structures in the rate control modules which further extend the `ath_node` structure by adding rate control specific information. `ath_node` is the ideal place where *minmax* rate information

can be maintained. This is because any rate control module inheriting from it will automatically have this information. However, we faced some implementation issues if we tried adding anything in the structure `ath_node`. Finally, we had to shift this information to the rate control node structures. Since, we are using sample rate currently, we modified the structure `sample_node` to include *minrate* and *maxrate* parameters. The details can be seen in Figure 6.1. For simplicity, `ath_node` structures are omitted. `ath` modules are shown on left and right separately to avoid complexity and differentiate *minmax* rate update process for the unicast packets from the rate selection process.

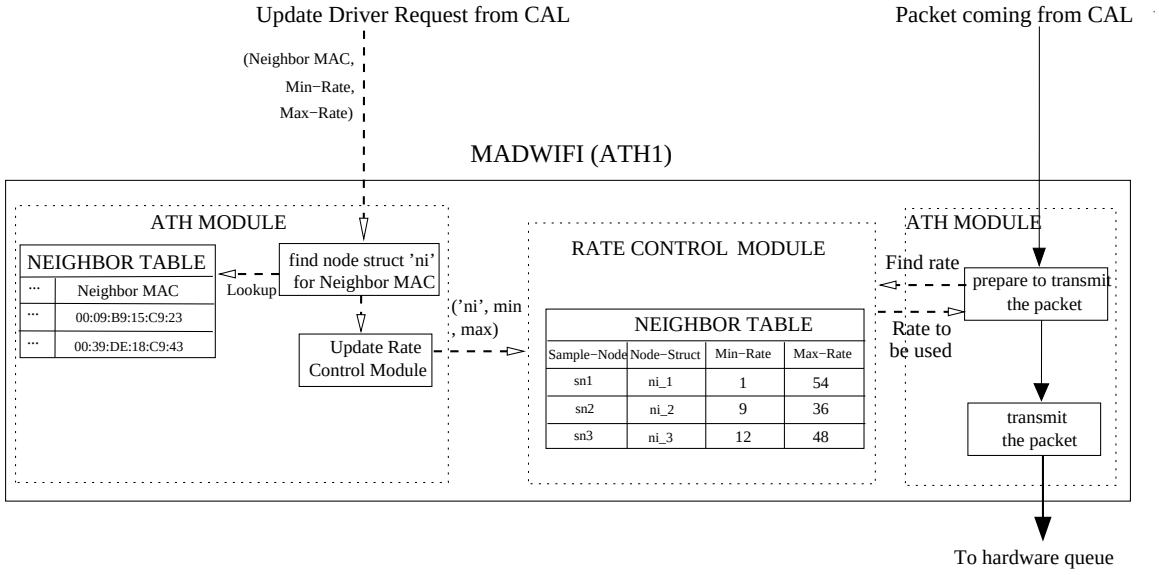

Figure 6.1 Rate control interaction inside Madwifi. The figure shows the rate update and selection process. It assumes Sample Rate algorithm as the rate control algorithm being used.

Since we are below the IP layer in the Madwifi driver, everything is stored on basis of MAC layer information or MAC addresses in particular. As we discussed in Chapter 5, we pass on the MAC address of the next hop neighbor and the *minmax* rate information from the bonding driver to the Madwifi driver. This was achieved by implementing `ath_handle_unicast()` in `ath` module. We do `EXPORT_SYMBOL (ath_handle_unicast)`

so that bonding driver can call this function when needed. When a packet comes down to bonding driver, it looks up the destination IP in the unicast table and finds the minimum and maximum rate information corresponding to the remote destination and calls `ath_handle_unicast()`, passing on next hop MAC address and *minmax* rates. The `ath` module finds the node structure corresponding to the next hop MAC address using `find_txnode()` function which looks up the specified MAC address in the neighbor table of the interface device. Once a match is found, it passes on the node reference and *minrate* and *maxrate* information to the rate control module by calling `ath_rate_handle_unicast()`, so that it can update the node structure. Once a 'Delete UnicastEntry' ioctl is issued from user level policy, CAL informs the Madwifi to use the default standard rates eg. 6-54 Mbps for 802.11a.

The basic interactions used in *minmax* rate update process are summarized in Figure 6.2.

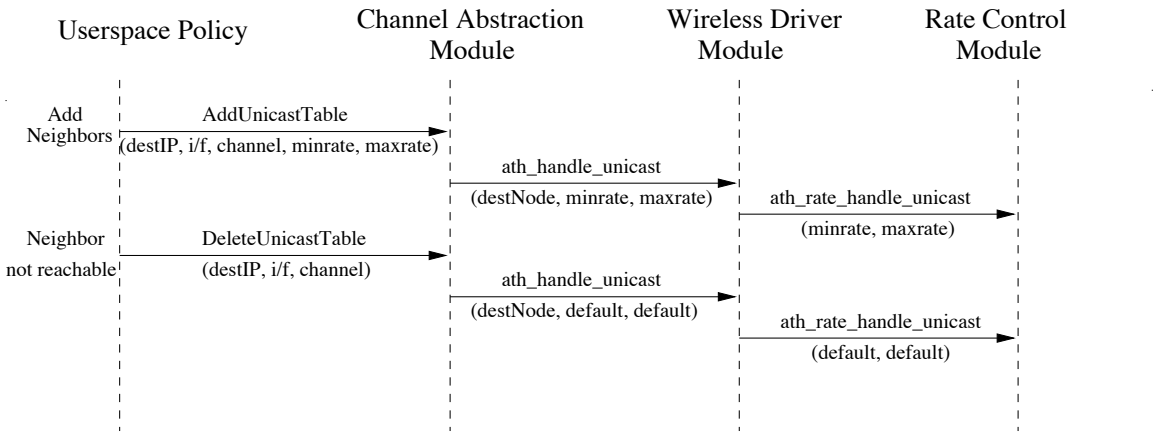

Figure 6.2 Rate control update process.

As we discussed briefly in Chapter 5, it is evident that the whole process of updating the Madwifi driver incurs an overhead and is not desirable on a frequent basis. All the destination entries with same next-hop neighbor are updated simultaneously as an optimization because *minmax* rate information is determined by the next-hop neighbor.

6.3.2 Neighbor deletion in Madwifi

It can be seen from the method of implementation that the *minmax* rate is stored in the node corresponding to next hop MAC address present in the neighbor table of our interface device. There may be a scenario when a neighbor is removed from the neighbor table maintained in the structure `ieee80211_node_table` of the interface device. Whenever the neighbor node reappears in the neighbor table, it contains the default minrate and maxrate. To avoid the situation where the corresponding entry persists in the unicast table while the Madwifi driver flushes the entry from neighbor table, the bonding driver needs to be informed to prevent any false assumption about the *minmax* rate information.

The best approach would have been to send a signal up to the bonding driver and set the *updated* flag back to 0 whenever the entry is flushed. However, this called for one more inter-module communication, in this case, from Madwifi driver to bonding driver. But, as we pointed out in Section 6.2, this created the problem of *circular dependency* between bonding driver and Madwifi driver. Therefore, an alternative approach of periodically updating the Madwifi driver at the bonding layer is used. Currently, we keep updating the Madwifi driver every 20 packets to prevent obsolete information. The details were discussed in Chapter 5.

6.3.3 Rate control inside Madwifi

The final rate control is done inside the module `ath_rate_control` according to the rate control algorithm selected by the user amongst AMRR, Onoe or SampleRate in the current implementation. Sample rate is the default and achieves better results than the others. So it was selected for the base of our modifications. As it was mentioned in Subsection 6.3.1, `sample_node` is modified to include *minrate* and *maxrate*. The modifications to sample rate are minimal and can be easily adapted for any other rate control algorithm inside the wireless device driver. The function `ath_rate_handle_unicast()` is implemented to set the *minrate* and *maxrate* values in the `sample_node` corresponding

to the node reference passed on by the function `ath_handle_unicast()` .

`ath_pci` module exposes an interface which defines the functions that the rate control module has to implement. The function of our interest is `ath_rate_findrate()`, through which rate control module returns the rate it considers best. The function `ath_rate_findrate()` internally calls `best_rate_ndx()` which iterates through various rates supported by the current IEEE 802.11 standard in use (a/b/g) and finds the index to the supposed best rate. It was modified to pick an index within the range of *minrate* and *maxrate* always.

There is another function `pick_sample_ndx()` which picks a random index for the rate to be sampled. This was also modified to always pick the sample index within the range specified. Thus, it is ensured that the transmission rate selected for a particular destination will always lie between minimum rate and maximum rate set in the unicast table.

CHAPTER 7

EXPERIMENTS

In this chapter, we present the results of some of the experiments we performed in our testbed mentioned in Chapter 3.

7.1 Topology variation

The foremost impact of changing broadcast rate of control packets or transmit power of the interfaces can be seen in the variation of network topology itself. Since transmit rate and power determine how far the packet will travel and its reliability, it is interesting to observe how the topology of our testbed changed as we varied these parameters.

7.1.1 Experimentation methodology

In this section, we run the multichannel protocol [4] implemented in Net-X on 22 nodes in an indoor testbed. Each node uses IEEE 802.11a with 5 channels. A snapshot of the routing table is taken every 10 sec on each node. We plot the node degrees per node for different broadcast rates at a transmit power of 18 dBm. Node degree is the term used for number of one-hop neighbors seen in the routing tables. The experiment was run for 400 sec and there was no other traffic within the network. Each value represents an average over two runs. The same experiment is later run with varying transmit power levels using a broadcast rate of 6Mbps.

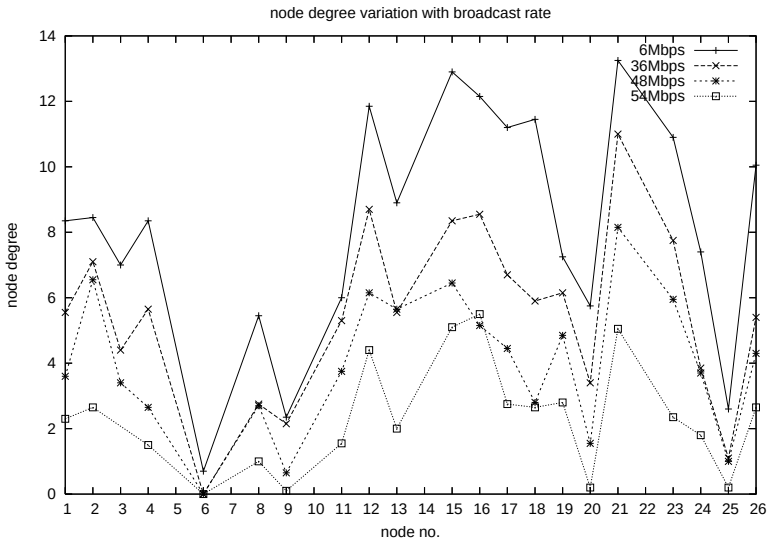

Figure 7.1 Variation of node degree with increase in broadcast rate.

7.1.2 Variation in broadcast rate

We first present the results of varying the broadcast rate. The rate at which hello packets are transmitted is determined by the current broadcast rate. The number of hello packets received by a node determines which neighbors it sees, i.e., its node degree. As we know, the hello packets travel farther as the broadcast rate is decreased. Therefore, the broadcast rate determines the node degrees. We can see in Figure 7.1, the node degrees are highest for 6 Mbps since hello packets are received most reliably at this rate. It was observed in our data that 6, 9, 12, and 18 Mbps results were not very differentiable. Hence, we did not include 9, 12, and 18 Mbps in the figure here. It does imply that broadcast packet delivery was not significantly affected till 18 Mbps. We see a clear dip in the node degrees as we increase the broadcast rates further. For 54 Mbps, the packet reliability would be clearly affected since broadcast packets are not retransmitted. The resulting loss in node connectivity can be noticed. Thus, overall we see a decrease in

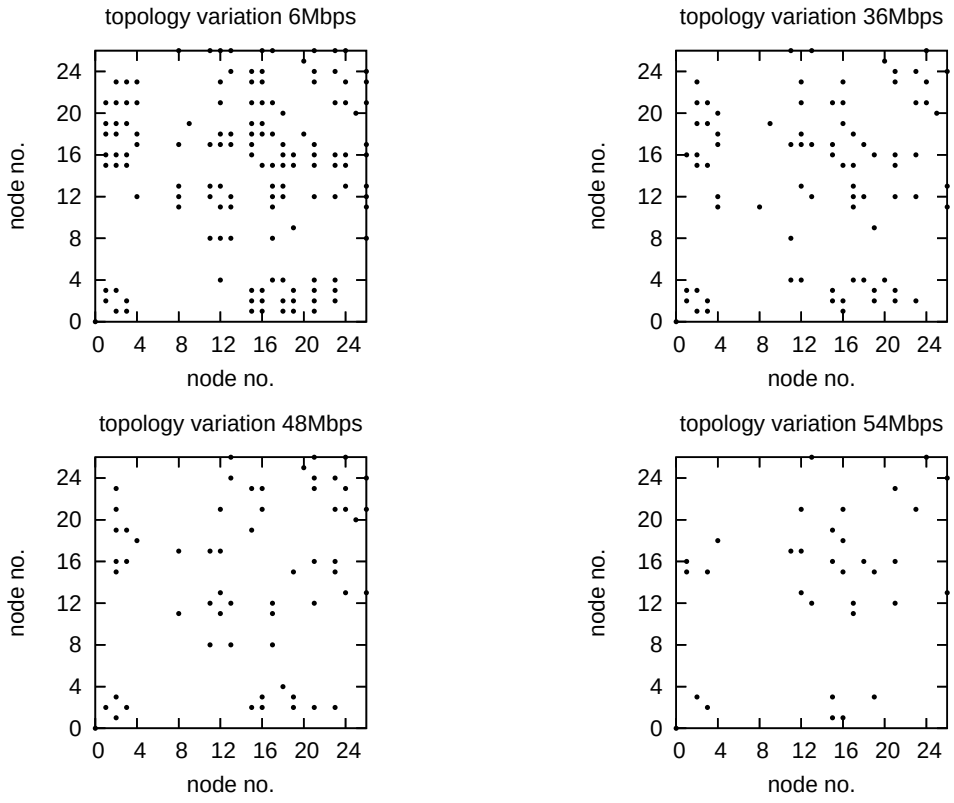

Figure 7.2 Scatter plot of the connectivity of nodes for various broadcast rates. Each point represents existence of a link between node on x-axis and y-axis.

node degrees as the broadcast rate is increased.

We present another result from the same experiment in Figure 7.2. Here, we plot the pairs of nodes between which a link is detected. Out of the 40 iterations in each experiment, we consider a node to be a valid neighbor only if it exists in the routing table more than 50% of the times for each of the two runs. Our protocol ensures the link symmetry as can be observed from the scatter plots. The plots are symmetric showing that the links are symmetric, i.e., a node adds another node in its neighbor table only if the other node

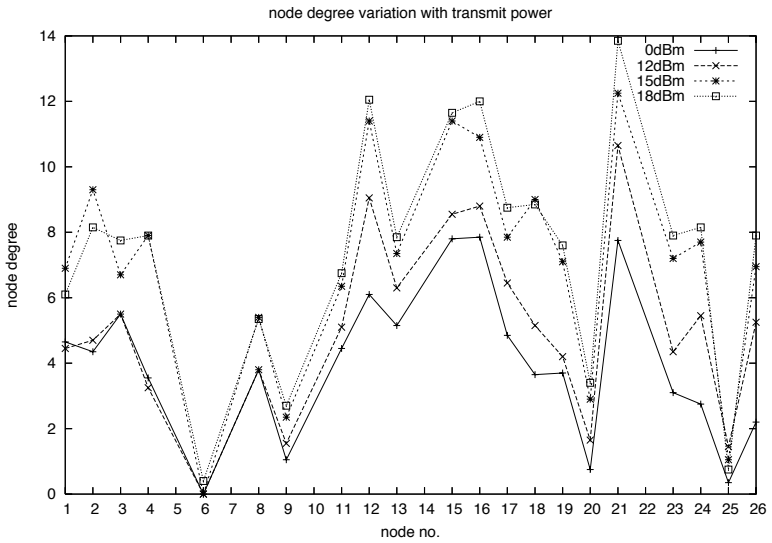

Figure 7.3 Variation of node degree with increase in transmit power.

is able to see this node too. At 54 Mbps, many nodes were isolated and the topology is visibly sparser.

7.1.3 Variation in transmit power

We performed the same experiment described in subsection 7.1.2 for various transmit power levels. We use Atheros 5212 chipsets with maximum transmit power of 18 dBm or 65 mW. The broadcast rate was fixed at 6 Mbps. The results are plotted in Figure 7.3. We did not plot the data for 5 and 10 dBm since the results are close to that for 0 dBm. It can be seen that node degree increases with the increase in transmit power. However, since we could not use much higher power levels, the increase is not as marked as seen with the variation in broadcast rate. Besides, our testbed deployment used in the experiment was fairly dense and hence, even lower power levels show significant node degrees. The scatter plots from this data are presented in Figure 7.4. The topology density decreases as the transmit power level of the interfaces is reduced. As pointed out

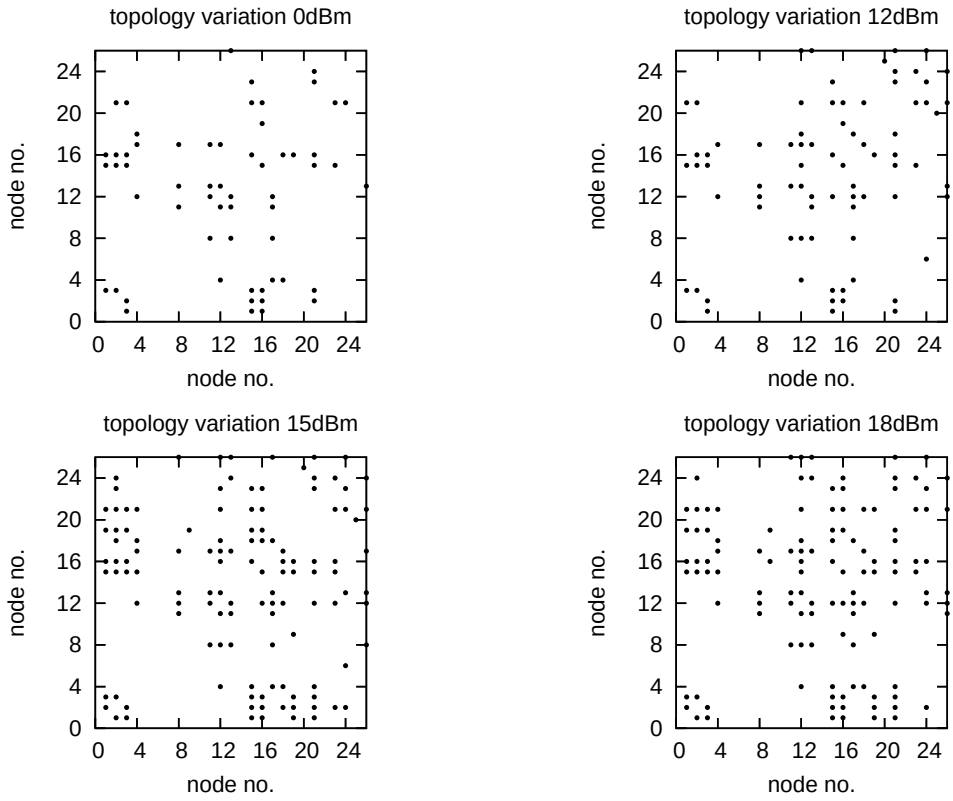

Figure 7.4 Scatter plot of the connectivity of nodes for various transmit power levels. Each point represents existence of a link between node on x-axis and y-axis.

earlier, the topology at 0 dBm is not as sparser as seen in the results of variation with broadcast rates in Figure 7.2.

7.2 Effect on routing

In this section, we will discuss how broadcast rate affects routing, in particular route lengths.

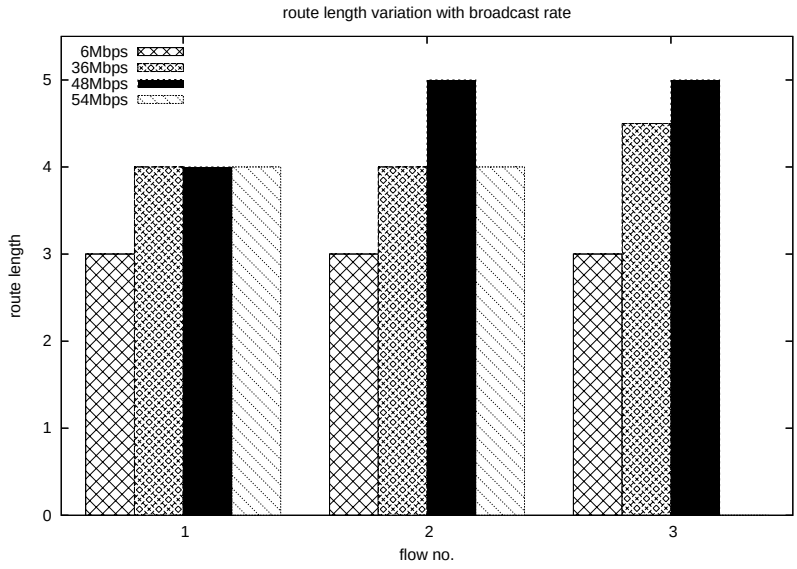

Figure 7.5 Variation in route lengths as the broadcast rate in changed.

7.2.1 Experimentation methodology

We chose three node pairs running the same hybrid multichannel protocol (such that the node pairs were not direct neighbors). Then, we ran ping sessions of 300 packets, 64 bytes each. The unicast packet rate range was set at 6-54 Mbps. The transmit power was fixed at 18 dBm. The route length used for these flows is plotted for various broadcast rate values in Figure 7.5. The route length refers to total number of nodes in the route including the source and destination. The data for 54 Mbps could not be collected for the flow between third node pair since no route could be discovered.

7.2.2 Observations

It was noticed that routes were frequently changed in higher broadcast rate cases. The route lengths shown are the average of lengths of most stable routes used for the ping sessions. It can be seen that route length for first flow increases from 6 to 36 Mbps

and remains high for 48 and 54 Mbps rates. For the second flow, route lengths increase consistently between 6 and 48 Mbps. The dip seen at 54 Mbps is deceptive since the link was highly lossy and 67% loss was seen in the ping packets. Hence, it cannot be considered as an actual 4 node (3 hop) long route. The third flow shows the same trend between 6 and 48 Mbps. At 54 Mbps, no route could be found at all. Thus, we can say that route lengths tend to increase as one increases the broadcast rates. This is consistent with the earlier results presented in Section 7.1 because denser topology at lower broadcast rates means more connectivity and hence, shorter routes. It is pointed out that the observation of route lengths was purely manual and hence, is prone to human errors. The routes were observed using the routing tables and route lengths calculated manually.

An advantage of using higher broadcast rate is that probability of transmissions succeeding at higher data rates increases. Hence, a combined control of broadcast rate along with our *minmax* rate adaptation should give better results than using the rate control for unicast transmissions alone.

7.3 Overhead measurements

When the packet reaches CAL, the *minmax* rate information is passed on to the driver. This process of updating the driver is done periodically after certain number of packets. This parameter is referred to as the frequency of updating the driver and is measured as number of packets. We use a simple one hop flow experiment to quantify the cost of updating the driver. This is to show that the overhead incurred by our implementation in Net-X does not significantly affect the performance of the hybrid multichannel protocol.

7.3.1 Experimentation methodology

We use two nodes, each having two wireless interfaces operating in IEEE 802.11a mode, and set up an Iperf [20] user datagram protocol (UDP) flow between them. The UDP flows run for 100 s each time with a different value of frequency of updates. Every data

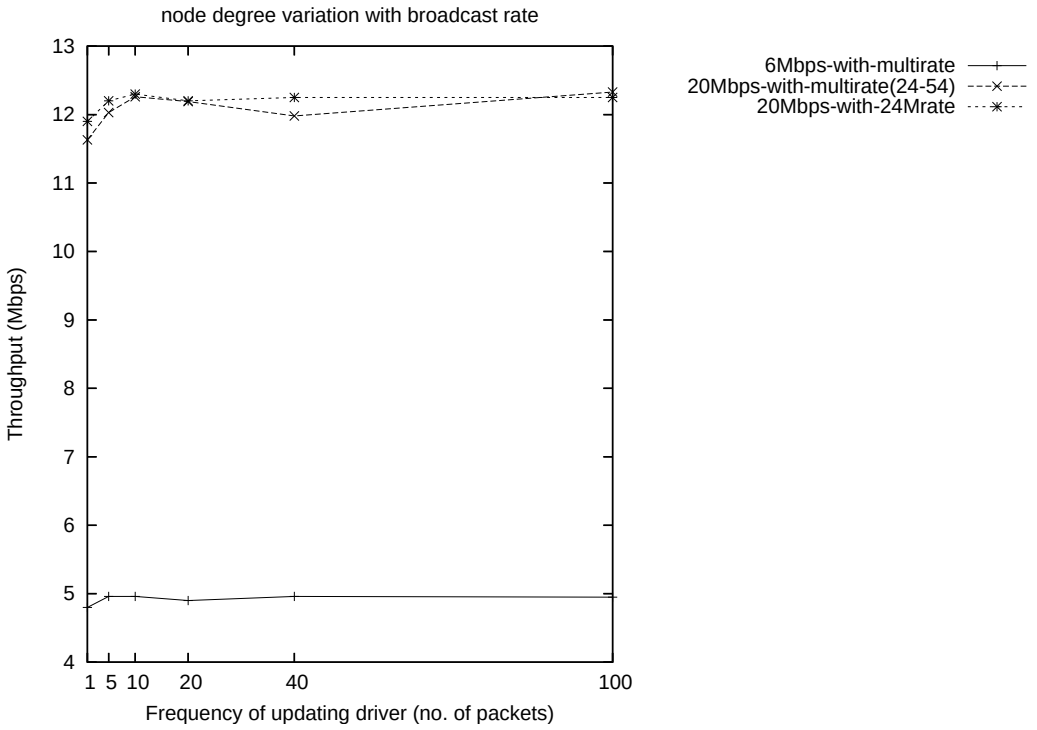

Figure 7.6 Throughput when frequency of updating the driver is varied.

point represents an average over six runs. We measure the throughput of the flows for different frequencies and plot the results in Figure 7.6. The broadcast rate is kept constant at 6 Mbps and transmit power is set to 18 dBm for all the flows.

We use three kinds of flow between the nodes, one flow at a time. Frequency of updating the driver is varied between 1, 5, 10, 20, 40, and 100 packets. Frequency of 1 packet denotes the per-packet update of the driver whereas frequency of 100 packets denote almost no update. In the first flow, traffic is generated at a rate of 5 Mbps and the unicast rate range is set to 6-54 Mbps. Traffic in second and third flows is generated at 20 Mbps while the data rate is kept constant at 24 Mbps for the second flow and varied between 24-54 Mbps for the third flow. We have kept the ratio of data rate to the traffic generation rate constant for all the flows, i.e., 5:6 for a fair comparison.

7.3.2 Observations

We see that per-packet (frequency = 1) updating of driver leads to small throughput loss and it becomes insignificant as the frequency is lowered beyond 5 packets. After 5 packets, the overhead appears negligible since the throughput achieved at a frequency of 5 packets is similar to that obtained at a frequency of 20 packets. Similarly, no throughput loss is seen in second and third flows for frequency lesser than 5. The drop in throughput of third flow with multi-rate at a frequency of 40 packets may have occurred due to the failed transmissions that were tried at higher rates. It can be noted that the flows used in our experiments are the only ones existing in the network at the time of experiment. Hence, there is no advantage of using higher rates since there are no competing flows in network to use the left-over time. But, any costly rate probing at higher rates may result in overall throughput loss.

These results show that our system does not add any significant overhead over a constant rate system which does not require the driver updating. Even per-packet driver updating incurs a very small overhead and hence, we can use per-packet rate or power control in future implementation. Currently, per-packet rate control will not be effective since the driver is updated before the packet is enqueued in bonding driver. The system will need to be modified so that the update process is done at the time of dequeuing the packet from bonding driver queues, to support per-packet rate adaptation. However, the overhead involved will remain same as we have demonstrated here. Besides, we argued in Chapter 5 that very frequent updating of driver is not required.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this thesis, we have presented a framework to support protocols that can exploit two-level joint rate control between user level application and wireless driver. We identified the shortcomings of existing rate control approaches, most of which work entirely at link layer. Our work tried to address the demerits of existing systems and implemented the proposed system in a realistic testbed. Implementing our system in a real world testbed was non-trivial given the practical difficulties faced in deployment and maintenance. Through our testbed, we have demonstrated the practical usability of our framework. Implementation of an intelligent protocol utilizing the high level rate control provided by our framework has been out of the scope of this thesis. But, we expect that such policies will be developed in future. We have also provided the minimal features to attain power control and these can be extended in future to support a full-fledged power control architecture.

Our system is integrated in the Net-X system [1] and our testbed does rate control over the multichannel protocol proposed by Kyasanur et al. [4]. The main concerns in any practical implementation are its extensibility and low overhead. Our framework adheres to the standard network stack layers in Linux kernel and is fully portable. The modifications made to the wireless driver are generic enough to be exported to any open-source driver that can be modified to expose the required functionalities. It was also made sure that our system does not affect the network performance in a non-trivial manner. Our

current architecture may need to be modified slightly to support per-packet rate control but we showed that even if one uses such rate control, the overhead incurred is not significant. We are hopeful about the hardware and firmware improvement in future that will reduce the card reset latency so that frequent power control can be used.

We identify the main contributions of our work as follows:

1. Provide a generic rate control framework at user level process that can tune the rate control at the driver level according to the application needs or any other heuristic. This includes the control over unicast as well as broadcast packets.

2. Present a simple power control interface to achieve dynamic power control.

Our rate control architecture is currently based on the next hop neighbor but it can be modified to use the remote destination or any other parameter as the basis. Similar modifications may be used for power control as well. Such extensions could be a possible future direction for our system.

With the commodity hardware becoming cheaper, we expect to see a greater number of multiple interface systems in future. In such scenario, Net-X architecture, developed for exploiting different kind of interface capabilities, provides a useful skeleton for implementing a variety of multi-rate, multi-channel and power control protocols. Some of the assumptions made in our system may be challenged as the hardware and requirements change in future. This may warrant modifications in existing architecture. However, we are hopeful that our design will prove to be flexible and easily extensible.

Currently, we do not support inter-operability between different IEEE 802.11 standards. Given our architecture, user-space application will have to be aware of the standard in use to specify accurate values of broadcast rates. This requirement may be eliminated in future by taking the input for broadcast rate in form of index into rate tables or some other heuristic. We may also integrate the current rate supported by a link as a measure

for routing metric in future. This will make our routing 'rate-aware'. Similar ideas can be used for transmit power too.

REFERENCES

[1] Pradeep Kyasanur, Chandrakanth Chereddi, and Nitin H. Vaidya, "Net-X: System eXtensions for Supporting Multiple Channels, Multiple Interfaces, and Other Interface Capabilities," Tech. Rep., University of Illinois at Urbana-Champaign, August 2006.

[2] *IEEE Standard for Wireless LAN-Medium Access Control and Physical Layer Specification, P802.11*, 1999.

[3] Xue Yang and N. H. Vaidya, "Spatial Backoff Contention Resolution for Wireless Networks," in *Second IEEE Workshop on Wireless Mesh Networks(WiMesh 2006)*, Sept 2006.

[4] Pradeep Kyasanur and Nitin H. Vaidya, "Routing and Link-layer Protocols for Multi-Channel Multi-Interface Ad hoc Wireless Networks," *Sigmobile Mobile Computing and Communications Review*, vol. 10, no. 1, pp. 31–43, Jan 2006.

[5] "Atheros inc," http://www.atheros.com.

[6] Chandrakanth Chereddi, "System Architecture for Multichannel Multi-interface Wireless Networks," M.S. thesis, University of Illinois at Urbana-Champaign, April 2006.

[7] "Multiband Atheros Driver for WiFi (MADWiFi), BSD-branch CVS snaphot," May 25th, 2005, http://madwifi.org.

[8] A.J. van der Vegt, "Auto rate fallback algorithm for the IEEE 802.11a Standard," Tech. Rep., Utrecht University, 2002.

[9] Mathieu Lacage, Mohammad Hossein Manshaei, and Thierry 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, 2004, pp. 126–134, ACM Press.

[10] Gavin Holland, Nitin Vaidya, and Paramvir Bahl, "A rate-adaptive mac protocol for multi-hop wireless networks," in *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, New York, NY, USA, 2001, pp. 236–251, ACM Press.

[11] John Bicket, "Bit-rate selection in wireless networks," M.S. thesis, Massachusetts Institute of Technology, February 2005.

[12] Starsky H. Y. Wong, Songwu Lu, Hao Yang, and Vaduvur Bharghavan, "Robust rate adaptation for 802.11 wireless networks," in *MobiCom '06: Proceedings of the 12th annual international conference on Mobile computing and networking*, New York, NY, USA, 2006, pp. 146–157, ACM Press.

[13] Jin Tang, G. Morabito, I.F. Akyildiz, and M. Johnson, "RCS: a rate control scheme for real-time traffic in networks with high bandwidth-delay products and high bit error rates," in *INFOCOM 2001.Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, Vol.1*, 2001.

[14] O.B. Akan and I.F. Akyildiz, "ARC: the analytical rate control scheme for real-time traffic in wireless networks," in *Networking, IEEE/ACM Transactions on, Vol.12, Iss.4*, August 2004.

[15] Ivaylo Haratcherev, Koen Langendoen, Inald Lagendijk, and Henk Sips, "Application-directed automatic 802.11 rate control, Project GigaMobile," 2002.

[16] V. Kawadia and P. Kumar, "Principles and protocols for power control in wireless ad hoc networks," *IEEE Journal on Selected Areas in Communications, 23(1):76–88*, Jan 2005.

[17] Jung Yee and Hossain Pezeshki-Esfahani, "Understanding Wireless LAN performance Trade-Offs," *Communications System Design*, p. 33, November 2002.

[18] "Net 4521 hardware from soekris," http://www.soekris.com/net4521.htm.

[19] "Pebble linux," http://www.nycwireless.net/pebble.

[20] "Iperf version 2.0.2," http://dast.nlanr.net/Projects/Iperf/.