SYSTEM ARCHITECTURE FOR MULTICHANNEL MULTI-INTERFACE WIRELESS NETWORKS

BY

CHANDRAKANTH CHEREDDI

B.E., Osmania University, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

System Architecture for Multichannel Multi-interface Wireless Networks

Approved by
Dr. N. H. Vaidya

____

____

ABSTRACT

Wireless ad hoc networks typically make use of a single radio interface on a fixed channel to communicate with neighboring nodes. Previous research has advocated the use of multiple wireless channels for enhancing the capacity of such networks. IEEE 802.11 based wireless network interfaces are capable of communicating over multiple frequency separated channels. In existing system architectures the use of multiple channels can be supported provided every channel has one dedicated interface. However, in many scenarios, hosts will be equipped with fewer interfaces than available channels, requiring interfaces to be switched between channels. Implementing protocols, which require frequent switching, is nontrivial.

In this thesis, we developed a system that supports the implementation of protocols which require frequent switching. Initially, the feasibility of implementing such protocols using currently available wireless interfaces and system software was studied. The feasibility study identified several shortcomings in present day operating systems for implementing multichannel protocols. We developed a system for the Linux kernel which can handle these shortcomings. Our system includes modifications to the wireless interface device drivers that improve performance of multichannel protocols. We discuss the details of the modifications made to the device driver used in our testbed. Following this, an example multichannel multi-interface protocol interaction with our system is presented. Finally, the performance of our system is analyzed, and suggestions for selecting system parameters are made.

To my Parents, Grandparents, Hari and Suma

ACKNOWLEDGMENTS

This thesis would not have been possible without the support of my adviser, Dr. Nitin Vaidya, my family members, group members, and friends. I would also like to acknowledge my fellow researcher and friend, Pradeep Kyasanur, who has helped me in every stage of my research. A special thanks to Romit Roy Choudhury for the many long hours of insightful discussions.

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

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Wireless technologies, such as IEEE 802.11a [1], are widely used for building wireless ad hoc networks. IEEE 802.11a provides for 12 different channels that are separated in frequency. Several researchers have proposed the use of multiple wireless channels for enhancing network capacity. Multiple channels have been exploited in infrastructure-based networks by assigning different channels to adjacent access points, thereby minimizing interference between access points. However, in ad hoc networks, two hosts can exchange data only if they both have a radio interface tuned to a common channel. Currently available off-the-shelf interfaces can operate only on any one channel at a time, though over time, an interface can be switched across different channels. Typically, hosts are equipped with one or a small number of radio interfaces, but the number of interfaces per host is expected to be smaller than the number of channels. However, by exploiting their ability to switch between channels over time, it is possible to develop new protocols, where nodes can communicate over all possible channels, leading to increased network capacity. This scenario is expected to be more likely as additional channels become available, yet there are few protocols for this scenario.

One protocol design approach when hosts have fewer interfaces than channels is to assign the interfaces of different nodes in a neighborhood to different channels [2,3]. In this approach, interfaces do not switch channels, but collectively the interfaces of all the nodes in a region are distributed across the available channels. An alternate approach that is more flexible is to allow each node to potentially access all the channels by switching some of its interfaces among the available channels [4–7]. This *interface switching* approach allows *dynamic* channel assignment based on node density, traffic, channel conditions, etc., and has shown to be a good choice in theory [8, 9]. However, frequently switching[1] interfaces introduces extra implementation complexity.

Multichannel wireless protocols also assume that wireless transmissions by radio interfaces on different channels do not interfere. Though this is true when the interfaces have considerable separation (greater than 10 m), at shorter distances they may experience some interference, especially when the channels are adjacent. Experimentation with 802.11a based hardware has shown that transmissions on *suitably* separated channels do not interfere even when the radio interfaces are close to each other (less than 6 in), as explained in Chapter 3. This study was useful in determining which 802.11a channels to use for communication when interface separation is small.

Building multichannel protocols that require frequent switching in current operating systems raises several challenges. The use of multiple channels and multiple interfaces, as well as switching interfaces among channels, has to be insulated from existing user applications. As described in Chapter 3, implementing frequent switching also requires new features in the operating system kernel. In Chapter 4, we present the design and implementation of a new *channel abstraction* module in the kernel that simplifies the implementation of multichannel protocols that require interface switching.

First generation 802.11a based radio interfaces implemented a majority of the 802.11 protocol in hardware. Hence it was impractical to implement research protocols which changed fundamental assumptions made by the protocol. Recent wireless hardware chipsets implement time *insensitive* parts of the 802.11 Medium Access Control (MAC)

[1]By "frequent switching," we imply potentially switching an interface, between channels, multiple times every second.

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, giving us the flexibility to change some parts of the MAC protocol. Switching channels on an interface incurs a delay, and the magnitude of the delay influences the frequency of interface switching. Modifications, such as those required to reduce channel switching delay, were made to the device driver to improve the performance of multichannel protocols, and are presented in Chapter 5.

The development of the channel abstraction module was motivated by our efforts to implement a *hybrid multichannel* protocol [7, 10] that was proposed earlier. The hybrid protocol requires two interfaces per host. One interface at each node is tuned to one "fixed" channel, and different nodes use different fixed channels. The second interface at each node can be switched among different channels, as necessary. A node transmits a packet to a neighbor on the fixed channel of its neighbor. This technique of tuning one of the hosts' wireless interfaces to a fixed channel reduces the complexity of coordination required to maintain connectivity. The hybrid protocol was shown to be fairly effective in utilizing multiple channels when there are fewer interfaces per host than channels [10]. By using the *channel abstraction* module that we have developed, the hybrid protocol was implemented as a simple user space daemon. In Chapter 6, we will describe the programming interface a multichannel protocol implementation can use to control the behavior of our channel abstraction layer. Following this, the hybrid protocol implementation is detailed, which serves as one example interaction with the channel abstraction module.

The channel abstraction module running the hybrid protocol was analyzed under topologies which require frequent switching. This is compared with the expected performance when there is little or no switching, and comparison shows that multichannel protocols can significantly improve network performance. The study also makes recommendations on the choice of two important parameters used by the channel abstraction layer. These details are presented in Chapter 7.

Chapter 8 concludes the thesis, followed by some ideas for future work. It also makes a few suggestions for future radio interface hardware design, which can improve the performance of multichannel protocols.

CHAPTER 2

RELATED WORK

Several protocols have been proposed for utilizing multiple channels. Most protocols [11–17] require each node to have one interface per channel (only as many channels as the number of interfaces per host are utilized in the network). Some of these protocols have been implemented in real systems [16, 17], but the challenges faced in our implementation are largely different. The few protocols proposed for the scenario where hosts have fewer interfaces than channels [2–7] have mostly been evaluated in a simulation environment.

Raniwala and Chiueh [3] have proposed an approach where different nodes are assigned to different channels, and interfaces rarely switch channels. This approach has been implemented in a testbed. However, as their approach does not require frequent interface switching, their implementation was feasible with existing operating system support.

Chandra and Bahl [18] have developed an interface switching architecture called "Multinet." Multinet is designed for nodes with a single interface, and virtualizes the single physical interface into two virtual interfaces. One virtual interface is used to connect to an infrastructure network, while the second virtual interface is used to connect to an ad hoc network. Multinet has some similarity to the channel abstraction module that we propose, but does not offer all the features necessary to allow an interface to switch among multiple channels. Multinet exports one virtual interface per channel, which *exposes* the available channels to the user applications, and may necessitate modifying these applications. In contrast, our work *hides* the notion of multiple channels from user applications and therefore does not require any modifications to existing applications.

A feature of the channel abstraction module is to export a single virtual interface to abstract out multiple interfaces. There are other testbed works that can also abstract multiple real interfaces into a single virtual interface [19–21]. However, with those approaches it is harder to support the notion of using multiple channels or interface switching between channels.

There have been many other testbed implementations for single channel, single radio networks, such as MIT Roofnet [22], Uppsala University ad hoc protocol evaluation testbed [23], system services for ad hoc routing [21], and similar other implementations presented in [24–26]. However, to the best of our knowledge, all those implementations would require nontrivial changes to support the use of multiple channels by switching interfaces.

CHAPTER 3

BACKGROUND

In this chapter we will make the case for the feasibility of multichannel protocol implementation. After this we will discuss the shortcomings in present day operating systems to implement such protocols, which motivated the development of the system presented in this thesis.

3.1 Feasibility Study

In this section we will discuss the two main requirements to implement any multichannel protocols which require interface switching multiple times a second. The first is to have a reasonably low *"switching delay"* (defined later). The second is the simultaneous usability of frequency separated channels. We are interested in knowing which channels may be used for simultaneous transmission/reception, i.e., interference characteristics of signals on these channels needs to be understood. Both these measures are properties of the hardware and are specific to the hardware used in developing such a system.

Note that it is possible to develop hardware which aims at performing better with respect to both these measures. Such hardware may be of particular interest in ISM bands at higher frequencies, which provide for multiple frequency separated channels.

3.1.1 Switching delay measurement

Multichannel protocols in the literature often assume that it is possible for wireless interfaces to switch between channels, and start transmission and reception on a new channel in a relatively short period of time. The assumption is that the "switching time" is small in comparison with the time taken to transmit a given number of packets. Switching time varies based on the hardware design, and it is beneficial to have a relatively short switching time. Such a use of the hardware was not a design goal for hardware designers with the current wireless cards,[1] and hence there was no particular motivation for them to keep switching delay small. Fortunately, this switching time was discovered to be small enough for certain hardware models.

As a first step, we decided to quantify the switching delay we would experience in using the Atheros card deployed on our testbed. The hardware on which these measurements were performed is based on the Atheros chipset [27] running the Madwifi [28] driver. The card came in a Personal Computer Memory Card Interface Association (PCMCIA) bus form factor and used the Atheros 5212 chip, capable of 802.11 a/b/g communication. The wireless interface was configured to the 802.11a wireless channels in the ad hoc mode. In the next few sections, details of our analysis and steps in reproducing them will be presented.

Before explaining the measurement technique, it is necessary to first understand the madwifi driver. The madwifi driver is divided into four main parts. The first part is the hardware abstraction layer (HAL), which 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. Next is the IEEE 802.11 stack which 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

[1] The terms "wireless card," "card," "radio interface," and "wireless network interface" are used interchangeably, and imply the same meaning.

performed by this part of the driver. Third is the layer which glues the HAL and the 802.11 stack parts of the driver. This part takes care of complexities related to the device interrupts, buffer management, beacon generation, etc. The last portion is where the rate control algorithms are implemented. At driver load time, one can select the rate control algorithm to use by choosing the correct module to be loaded.

In the unmodified madwifi device driver, when a channel switch command is issued from user space via ioctl[2] calls, the call first reaches the 802.11 stack part of the madwifi driver. Specifically, the call is traced to the `ieee80211_ioctl_siwfreq()` function. This function immediately requests for *reinitialization* of the Atheros hardware. Reinitialization request is handled by the `ath_init()` function, which prepares the driver and device for a reset and finally requests the HAL to reset the device to tune to a new channel via the `ath_chan_set()` function. After device reset, the driver performs some postprocessing and sets up the requisite driver state before returning control to the caller.

In the above procedure, some of the steps may be unnecessary for typical multichannel protocols, which break the assumption that all neighbors are tuned to a specific channel. Specifically, some parts of the preprocessing before device reset, and postprocessing are extraneous, and were removed. Details on which parts were by-passed and how this was achieved are explained in Chapter 5.

In the modified version of the device driver, where parts of preprocessing and postprocessing have been removed, the ioctl call to switch channel is routed directly to the `ath_chan_set()` function via `ieee80211_ioctl_swifreq()`. Hence, we shall look at this function in detail to quantify switching delay.

[2] "Ioctl" calls are a mechanism for communicating with kernel space devices from user space. Ioctl informally stands for "input and output control."

The `ath_chan_set()` function makes a series of calls to the HAL layer before and after issuing the device reset call. The total delay experienced by a switch channel call is the amount of time it would take for all of these calls to complete. Each of these steps is detailed below.

1. **Disable Interrupts**: The device has to disable all receive and send interrupts to avoid any interruptions from the device. The function `ath_hal_intrset()` is called to achieve this.

2. **Drain Pending Transmission Packets:** In preparation for a device reset, any packets pending transmission are flushed, by calling `ath_draintxq()`. Note that care has to be taken before issuing a switch channel call because of this step. The interface does not wait for any pending packets to be transmitted before resetting, but discards them altogether.

3. **Disable Packet Transmissions:** Once we have disabled all interrupts, we should instruct the card to stop any ongoing packet direct memory access (DMA) for transmission. A call to the function `ath_hal_stoptxdma()` is made to achieve this. Following this step, the driver instructs all higher layers to queue any packets that are to be transmitted by this card, using the `netif_stop_queue()` call.

4. **Disable Packet Reception**: Call `ath_hal_stoppcurecv()` to disable the receive functions in the program control unit (PCU). This is followed by a call to `ath_hal_setrxfilter()` to clear the receive filter, and finally, to disable any further receiver DMA, `ath_hal_stopdmarecv()` is called. Following this the driver waits for 3 ms in a *busy wait* loop to complete any reception DMA which may have already started. This turns out to be a major portion of the total switching delay.

5. **Reset to New Channel:** Call `ath_hal_reset()` to reset device and configure to new channel. This is the most important step in the channel switch procedure, which actually tunes the wireless interface to a new frequency.

6. **Enable Packet Transmission:** The `ath_hal_putrxbuf()` function is called to re-enable packet transmission logic and DMA buffers.

7. **Enable Packet Reception:** Call `ath_hal_rxena()` to enable receiver descriptors, followed by a call to `ath_hal_startpcurecv()` to enable the receive functionality of the program control unit.

All the above calls, including the call to perform a device reset, do not have any portions of code which may sleep, i.e., they are synchronous calls which do not give up control until the process is complete. Investigation into a reverse engineered hardware abstraction layer, called OpenHAL [29], revealed that, most likely, all the above listed calls are synchronous. Hence, to measure delay, we instrumented the function `ath_chan_set()` with `do_gettimeofday()` calls at the start and end of the function.

Such instrumentation measured the time required to complete the call and return to the caller to be about 5 ms. Out of the measured 5 ms, 3 ms are spent waiting for any ongoing receive DMA to complete, and may be further optimized. In multichannel protocols, it is possible to achieve performance improvement with this switching delay, and we validate this claim with measurements of a protocol implementation.

3.1.2 IEEE 802.11a orthogonal channel usability

The 802.11a specification provides for 12 nonoverlapping[3] channels, 4 each in the U-NII lower band (5.15 to 5.25 GHz), U-NII middle band (5.25 to 5.35 GHz), and U-NII upper band (5.725 to 5.825 GHz). Each of these channels is 20 MHz wide. The channels in the lower band are numbered 36, 40, 44, and 48. Similarly the channels in the middle band are numbered 52, 56, 60, 64; and 149, 153, 157, and 161 in the upper band. In theory it should be possible to have simultaneous reliable transmission on these channels.

[3]IEEE 802.11a allows for the use of 12 different channels in the United States. These channels are each 20 MHz wide, and their center frequencies are separated by 20 MHz, so they do not overlap in frequency. We refer to these channels as "nonoverlapping" channels.

In practice though, due to signal leakage from one channel to another, transmissions on these nonoverlapping channels may still interfere with one another. The interference level is affected by many factors such as the hardware used, distance between transmitters, antennas used, etc. A pair of channels are *orthogonal*, if simultaneous communication on those channels do not interfere with each other.

We conducted a small study using two nodes, node A and node B, in our lab corridor. The nodes were single board computers from Soekris Engineering [30] (model net4521), equipped with two 802.11a wireless interfaces. One wireless interface made use of the MiniPCI bus and external antennas, while the other was a PCMCIA bus based card with an antenna directly attached to the interface. Both cards were based on the Atheros 5212 chipset, and made use of the open-source madwifi driver.

For the experiment, two nodes were placed at the two ends of our laboratory corridor. An ad hoc wireless network was setup, and the PCMCIA wireless interface on node A was setup to transmit data to the MiniPCI wireless interface on node B. Node B was directly connected to a laptop over Ethernet for data collection and control. The MiniPCI wireless interface on node A was used in 802.11b mode to communicate with the laptop for data logging and control. Since the nodes used were based on an Intel 486 microprocessor at 100 MHz with 64 Mbytes of memory, they were unable to handle transmission and reception at high data rates. To avoid processing bottlenecks from skewing the experiment, we forced the cards to use the 6 Mbps bit rate. For transmission and reception, the Iperf [31] measurement tool was used. The second wireless interface on node B (PCMCIA) was now used to create interference for the reception happening on node B's MiniPCI based interface. The interference data traffic was created using the *ping* program, by transmitting packets to the broadcast address in flood mode. The scenario we were testing here is known to create the worst possible interference at any node; i.e., when a node has two interfaces, one involved in transmission, and the other in reception, the interface which is receiving traffic will experience low signal to interference-plus-noise ratio (SINR). For

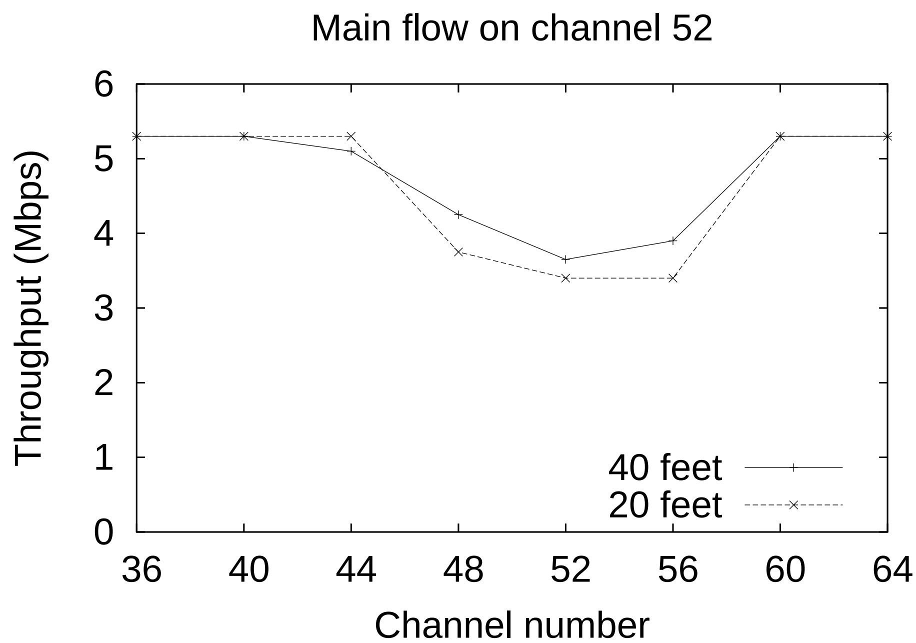

Figure 3.1 Throughput with the presence of interference on each lower and middle band channel, nodes are separated by a distance of 20 ft and 40 ft. Throughput with no interference is at 5.3 Mbps.

other scenarios, where both interfaces are either receiving or transmitting data, the SINR tends to be much better.

The experiment was conducted at distances of 80 ft, 60 ft, 40 ft, and 20 ft. The main data flow between node A and node B was set to channel 52. The interfering flow was set to different channels, one after the other, from 36 to 64, to cover the lower and middle bands, as shown in Figures 3.1 and 3.2. After this, the main flow was shifted to channel 149 and the interfering traffic was generated on channels starting from 149 to 161, to cover the upper band, as shown in Figures 3.3 and 3.4. This clearly showed one trend, viz., as the node distance was reduced, the tolerance to interference caused due to power leakage from adjacent channels increased. That is, when nodes are closer there may be more channels which can be used.

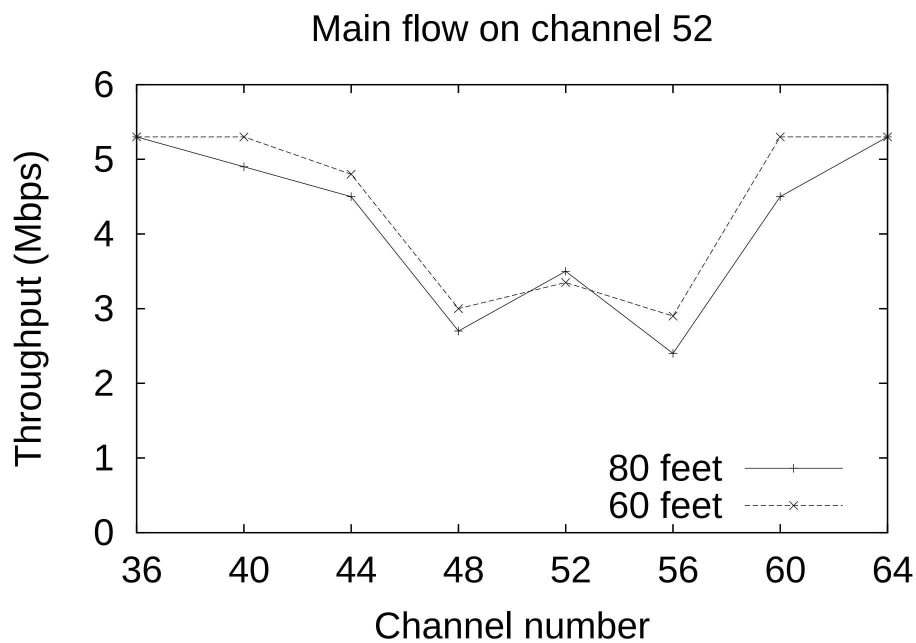

Figure 3.2 Throughput with the presence of interference on each lower and middle band channel, nodes are separated by a distance of 60 ft and 80 ft. Throughput with no interference is at 5.3 Mbps.

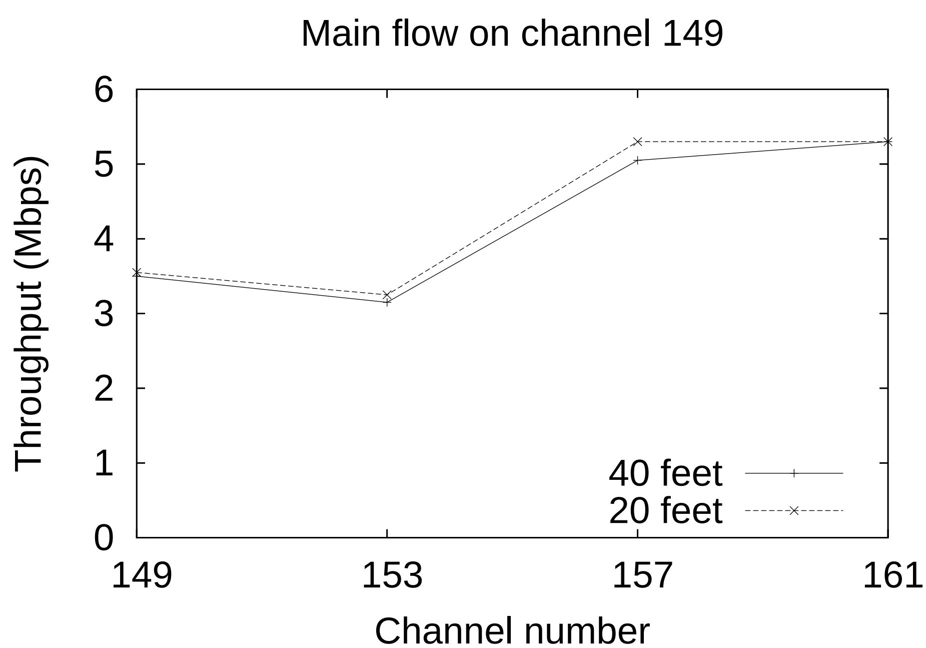

Figure 3.3 Throughput with the presence of interference on each upper band channel, nodes are separated by a distance of 20 ft and 40 ft. Throughput with no interference is at 5.3 Mbps.

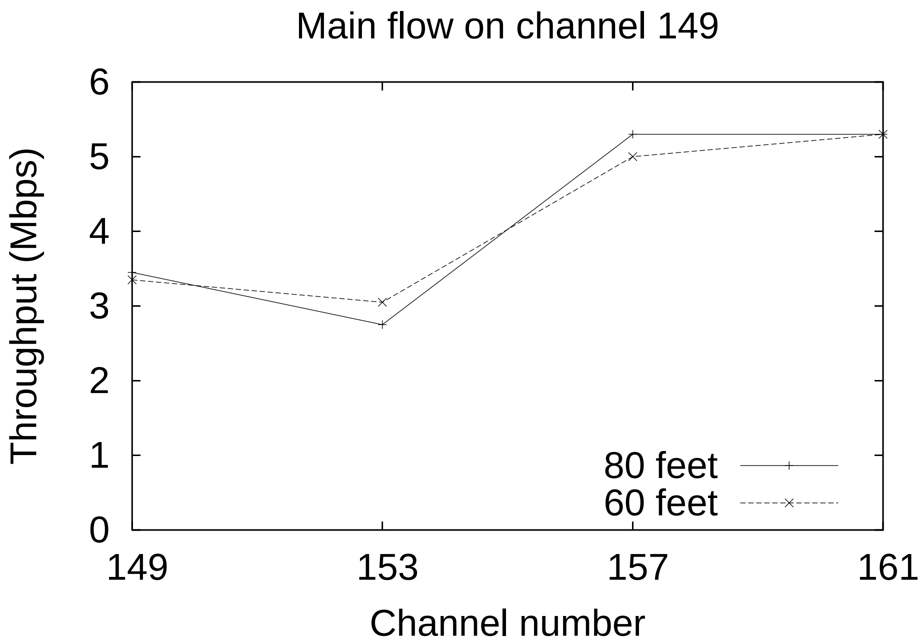

Figure 3.4 Throughput with the presence of interference on each upper band channel, nodes are separated by a distance of 60 ft and 80 ft. Throughput with no interference is at 5.3 Mbps.

At higher distances, the observed throughput shows an interesting phenomenon. When using the 802.11 Distributed Coordination Function (DCF) for channel access, it is possible to achieve throughputs of 3.5 Mbps for the main flow, even when a completing flow (ICMP ping traffic in our case) is contending for the channel, as shown in Figures 3.2 and 3.4. Contrary to what is expected, a transmission on a channel adjacent to the current channel causes more performance degradation than in the case when the interfering flow was on the same channel, as seen in Figures 3.2 and 3.4. We expect that this is because when a transmission is happening on an adjacent channel, the carrier sense mechanism would fail to detect it, and hence results in more collisions and lower performance.

In essence, we feel that it is possible to use three channels in the lower and middle band without any noticeable interference when at distance of about 80 ft, and similarly two channels are usable in the upper band. We are unsure about the behavior at higher distances, or at network edges when the normal channel itself may be unstable. At lower distances, it may be possible to use an extra channel in the lower and middle bands, depending on specific channel conditions.

3.2 Kernel Support for Multiple Channels

Implementing multichannel protocols that require interfaces to switch frequently is non-trivial. Existing hardware does not provide sufficient support for very fast interface switching, as previously explained. Furthermore, we want to ensure that the use of multiple channels would be transparent to user applications and higher layers of the network stack. This constraint implies that changes are needed in the kernel to hide the complexities of using multiple channels with interface switching.

We identify and present the features needed in the kernel, with Linux as an example, for implementing multichannel protocols with interface switching. In Chapter 4, we present details of our implementation that provides the requisite kernel support.

3.2.1 Specifying the channel to use for reaching a neighbor

An implicit assumption made in most operating systems is that *there is a one-to-one mapping between channels and interfaces.* This assumption is satisfied in a single channel network because an interface is fixed on the single channel used in the network. This assumption continues to be met in a network where each node has m interfaces, and the interfaces of a node are always fixed on some m channels, or switching is infrequent enough to seem like a static channel assignment. However, in the scenario we address, the number of interfaces per node could be significantly smaller than the number of channels. When interfaces have to switch channels, the assumption that there is a one-to-one mapping between channels and interfaces is invalid.

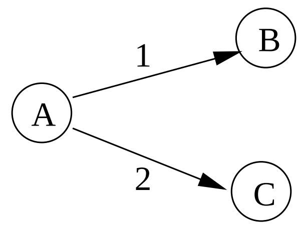

Figure 3.5 Example illustrating the lack of kernel support for specifying the channel to use to reach a neighbor.

The one-to-one mapping assumption is evident in the kernel routing tables, which specify only the interface to use for reaching a neighbor. For example, consider the scenario shown in Figure 3.5. In the figure, suppose that each node has a single interface. Suppose node B is on channel 1 and node C is on channel 2. Under this scenario, when A has to send some data to B, it has to send the data over channel 1, and similarly data to C has to be sent over channel 2. To achieve this, the interface at A has to be switched between channels 1 and 2, as necessary.

This implies that the channel to use for transmitting a packet may depend on the destination of the packet. However, in the standard Linux kernel, the routing table entry for each destination is associated only with the interface to use for reaching that destination, and has no information about the channel to use. Without this information in the kernel tables, it is hard to implement multichannel protocols which often use a small number of interfaces to send data over more channels than interfaces.

3.2.2 Specifying channels to use for broadcast packets

In a single channel network, broadcast packets sent out on the wireless channel are received by nodes within the transmission range of the sender. The wireless broadcast property is used to efficiently exchange information with multiple neighbors (for example, during route discovery). In a multichannel network, different nodes may be listening to different channels. Therefore, to ensure that broadcast packets in a multichannel network reach (almost) all the nodes that would have received the packet in a single-channel network, copies of the broadcast packet may have to be sent out on multiple channels. For example, in Figure 3.5, node A will have to send a copy of each broadcast packet on both channel 1 and channel 2 to ensure its neighbors B and C receive the packet.

Several existing applications use broadcast communication, for example, the address resolution protocol (ARP). To ensure that the use of multiple channels is transparent to such applications, it is necessary that the kernel send out copies of broadcast packets on multiple channels, when necessary. However, there is no support in the kernel to specify the channels on which broadcast packets have to be sent out, or to actually create and send out copies of broadcast packets on multiple channels. Therefore, there is a need to incorporate mechanisms in the kernel for supporting multichannel broadcast.

3.2.3 Support for interface switching

As we discussed earlier, an interface may have to be switched between different channels to enable communication with neighboring nodes on different channels, and to support broadcasts. A switch is required when a packet has to be sent out on some channel c, and at that time there is no suitable interface tuned to channel c. Assume that the kernel can decide whether a switch is necessary to send out some packet. Even then, the kernel has to decide whether an immediate switch is feasible. For example, if an interface is still transmitting an earlier packet, or has buffered some other packets for transmission on the present channel, then an immediate switch may flush those earlier packets. Therefore, there is a need for mechanisms in the kernel to decide if earlier transmissions are complete, before switching an interface.

When an interface cannot be immediately switched to a new channel, packets have to be buffered in a channel queue until the interface can be switched, as depicted in Figure 3.6. Switching an interface incurs a nonnegligible delay (around 5 ms in our testbed

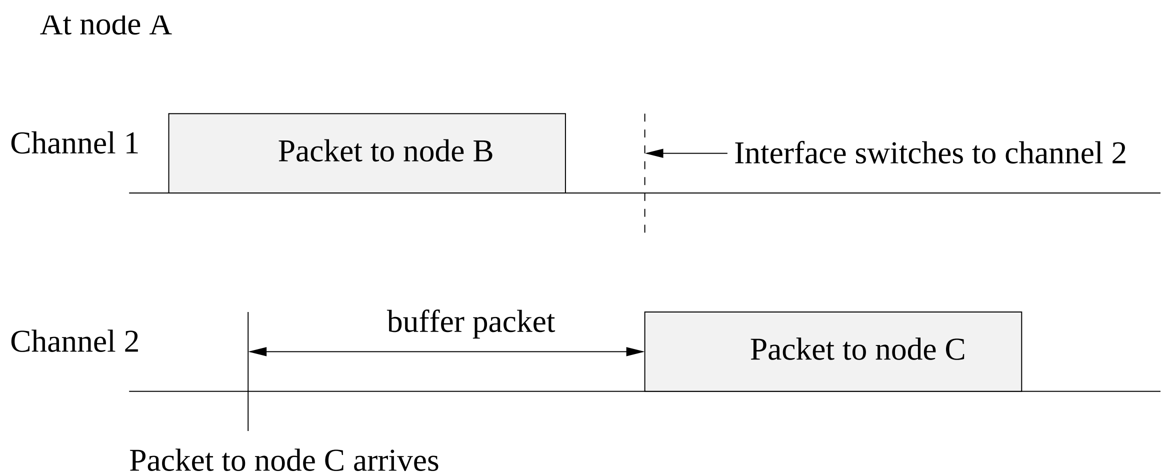

Figure 3.6 Example illustrating support required for queueing and switching.

as previously explained), and too frequent switching may significantly degrade performance. Therefore, there is a need for a queueing algorithm to buffer packets, as well as a scheduling algorithm to transmit buffered packets using a policy that reduces frequent switching, yet ensures that queueing delay of packets is not too large. Scheduling policies are protocol-specific, and in our implementation we implemented one such policy specific to the protocol explained in [10].

The discussion in this section identified the need for several new features in the kernel for supporting the use of multiple channels, especially when interfaces have to switch between channels.

CHAPTER 4

SYSTEM ARCHITECTURE

In this chapter we will detail the architecture of our system which enables multichannel multi-interface protocol implementation. Initially, we shall discuss the various design possibilities in implementing our target system. Later, a more detailed discussion about the actual implementation and various tradeoffs will be presented.

4.1 Design Alternatives and Decisions

In Chapter 3, we presented the major challenges which had to be addressed in implementing our protocols. There are potentially many alternative solutions to these. In this section we shall discuss such alternatives, and argue in support of the choices we made for our system.

In most current systems, the operating system is divided into multiple address spaces. The main divisions are "kernel space" and "user space," which are differentiated by the privilege level. It is well known that programming the required software in user space is a much easier task given the current system design, and isolation provided by the kernel. Given the privilege level restrictions of user space, and the requirement to abstract system level details such as existence of multiple cards, it was infeasible to implement our system fully in user space. Therefore, we made the choice of implementing most of our system in kernel space. This also had the advantage of providing better timing response when implementing the queueing and switching module.

4.1.1 Kernel space layers

The Linux kernel is a large and complex system, running in a single address space. It has many complicated interactions, and various layers at which clean "extensions" may be written. The Linux kernel's networking stack is layered in a fashion very similar to that of the OSI stack. This approach was used to ease implementation and improve extensibility. Given that our protocols mainly involved MAC and network layer modifications, we had to choose between the two layers to extend. The Linux network layer is very well developed and allows for extensions to be written via the "Netfilter" architecture of the Linux kernel. However, implementing our extensions at the network layer in the kernel posed three main problems. First, the ability to abstract multiple interfaces as a single interface would not be possible here. Second, we would not have tight control over the packet delivery delay from the network layer to the actual device. Though we only give best-effort guarantee as far as switching and timing requirements go, it is better done at the MAC layer to reduce the levels of indirection. Third, broadcast data which is generated below the network layer, such as ARP, will be out of our control and will have to be routed back to the network layer for multichannel broadcast. Though intercepting ARP traffic may be possible via MAC layer netfilter hooks, they are at best cumbersome and unclean extensions.

The Linux networking layer is implemented in software, while the MAC layer may be partly implemented in hardware. Most complexities involving the MAC protocol are dealt with by the device hardware or by the device driver. However, the MAC layer exposes a common interface between the complex device driver and hardware and the other parts of the kernel. This layer also deals with the data structures which expose device and driver related information to other parts of the kernel. This gives us the ability to abstract multiple devices as a single virtual interface. The last layer of buffering and queue maintenance is performed at this layer, thus minimizing any further queueing delays. Also, since all kernel traffic is generated above this layer, both unicast and broadcast data traffic can be handled without any special changes.

The Linux kernel has functionality to abstract multiple interfaces as one interface, using a special driver known as "bonding driver." 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.

Given that the bonding driver already achieves one of our main goals of multi-interface abstraction, we chose to develop other capabilities 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.

4.2 Bonding Driver

Our system is closely coupled with the functionality of the bonding driver, so we must present its functionality before explaining our extensions.

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.

At module load time bonding driver allows us to set the *bonding mode*. Every bonding mode has different semantics. Except for a few common functions, such as setting up the basic data structures and device enslaving routines, the actual scheduling and packet delivery routines to the real devices are implemented independently as bonding modes. Bonding modes are large sections of code and are the main part of the bonding driver. In essence a new bonding mode can be used to cleanly extend the current functionality of the bonding driver. By default the bonding driver is loaded in the balanced round robin mode. As the name suggests, in this mode packets are scheduled in a round robin fashion to each device.

Our extensions were implemented as a new mode, named the *Multi Channel Multi Interface* mode. During initial development, we have also implemented two different modes, namely, *Mutli Interface* mode and *Channel Broadcast* mode. They were mainly used for testing different functionalities and have been integrated in a more polished fashion into our final *Multi Channel Multi Interface* mode.

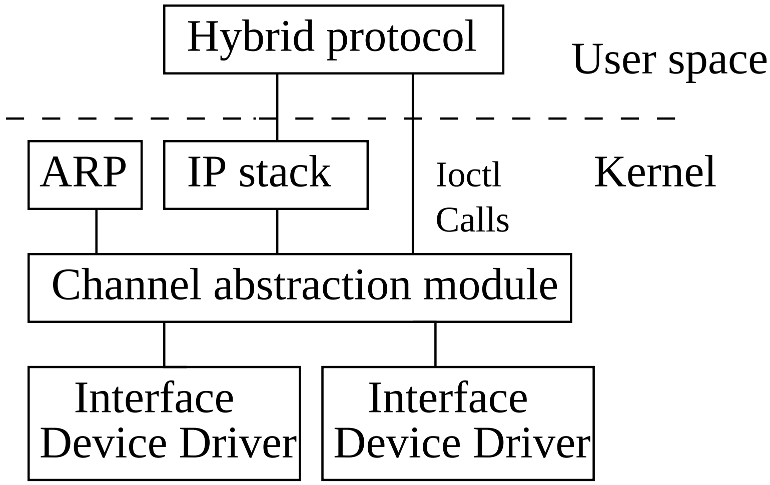

Figure 4.1 System architecture.

4.3 Channel Abstraction Module

In this section we will discuss in detail the most important part of our system, the Channel Abstraction Module (CAM). Figure 4.1 depicts the system architecture showing the CAM. As we can see from the figure, the channel abstraction module resides between the network layer and the interface device drivers. The user space daemon is an example implementation of a multichannel protocol which interacts with the channel abstraction module using ioctl calls. The user space daemon deals with complexities of multichannel protocols which are not time sensitive, and are hence best done at user space. A more detailed discussion of this will follow in Chapter 6.

Using this system architecture, we mask all complexity of underlying protocols from user applications. Such an abstraction of complexity is an important design goal, which facilitates all applications, including kernel space protocols such as ARP, to function without any changes. Therefore, applications are oblivious to multiple interfaces, channel switching, buffering, etc. Also, the system has been implemented above the link layer, making it independent of the underlying device driver.

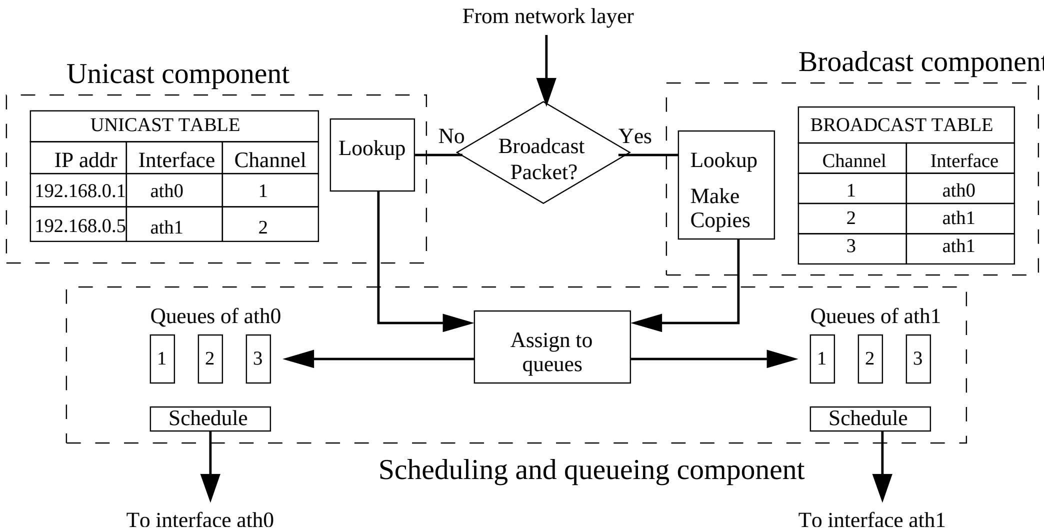

Figure 4.2 Components of channel abstraction module: Tables are filled with the assumption that there are two interfaces "ath0" and "ath1," and three channels.

Channel Abstraction Module can be best explained as a combination of four key components, which address different challenges. Though these components are closely coupled in implementation, they are explained independently for conceptual clarity. Figure 4.2 shows the key components of the channel abstraction module:

- Unicast component: Enables specifying the channel to use to reach a neighbor/ destination.
- Broadcast component: Provides support for sending broadcast packets over multiple channels.
- Scheduling and queueing component: Supports interface switching by buffering packets when necessary, and scheduling switching across channels.
- Statistics component: Maintains various counters and statistics which may be used for debugging, route selection, etc.

4.3.1 Unicast component

The unicast component provides support for specifying the channel to use to reach a neighbor. The unicast component maintains a table called the "Unicast table" as shown in Figure 4.2. The unicast table is composed of tuples. Each tuple has a destination IP address, a channel the destination is expected to be listening on, and a real interface to use to transmit to the neighbor. The unicast table is populated by a user space multichannel protocol via ioctl calls (entries can be added, deleted or updated). We will describe in Chapter 6 an example of the approach used by the hybrid multichannel protocol to populate the unicast table.

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. After this, the packet is handed off to the queueing component for subsequent transmission.

4.3.2 Broadcast component

The broadcast component provides support for sending out copies of a broadcast packet on multiple channels. The broadcast component maintains a table called the "Broadcast table" as shown in Figure 4.2. The broadcast table maintains a list of channels over which copies of a broadcast packet have to be sent, and the interfaces to use for sending out the copies. The table is populated by an user space multichannel protocol. This table structure offers protocols the flexibility of changing the set of channels to use for broadcast over time, as well as controlling the specific interface to use for broadcast. Therefore, protocols that use a common channel for broadcast, and protocols that send copies of broadcast packets over all the available channels can all use this broadcast architecture.

When the channel abstraction module receives a broadcast packet from the network layer, it hands the packet off to the broadcast component. The broadcast component creates a copy of the packet for each channel listed in the table, and hands off the copies of the packet to the queueing component.

4.3.3 Scheduling and queueing component

The scheduling and queueing component is the most complex part of the channel abstraction module. For each available interface, the component maintains a separate set of channel queues as shown in Figure 4.2. The user space multichannel protocol, on startup, can specify the list of channels supported by each interface using ioctl calls. This architecture allows different interfaces to support possibly different sets of channels.

The queueing component receives a packet, from either the unicast or the broadcast component, along with information about the channel and interface to use for sending out the packet. Using this information, the packet is inserted into the appropriate channel queue for subsequent transmission. Each interface runs a separate scheduler to send out the packets. In our current implementation, we use identical round-robin schedulers on all interfaces.

The scheduler is responsible for controlling interface switching. Since interface switching delay is not negligible (around 5 ms for our hardware), we want to amortize the switching cost by sending multiple packets on each channel (if possible) before switching to a new channel. However, waiting for too long on a channel increases packet delay. Once the interface is switched to a channel, it stays on that channel for a duration of at least T_{min}. If the channel is continuously loaded, then the scheduler decides to switch to a different channel (only if another channel has packets queued for it) after a duration of at most T_{max}. Both T_{min} and T_{max} are module load time parameters and can be suitably modified as per network requirements.

Figure 4.3 describes the scheduler operation. The scheduler maintains an estimate T_{fin} of the time needed to transmit packets it has already delivered to the interface device driver (these packets are stored in a separate queue within the device driver). T_{fin} is the sum of the expected transmission time for each packet delivered to the device driver. The time required to transmit a packet T_t, is estimated based on packet size as follows:

$$T_t = \frac{Packet\ Size + Ack\ Packet\ Size}{Bit\ Rate} + SIFS\ Time + Preamble\ Time$$

Initially, after a switch, T_{fin} is set to zero. For each packet that is sent to the device driver, T_{fin} is incremented by an estimate of the time needed to transmit that packet. We ignore channel contention in the calculations as it is not critical to have very accurate estimates, as explained later. Also, the bit rate used is assumed to be a fixed value (802.11a rate of 6 Mbps for our testbed), which is true in our experiments. This limitation is because all wireless drivers do not use the same bit rate selection algorithms. Ready to send (RTS) packet and clear to send (CTS) packet transmission times are ignored in our estimation, as they are not used in our testbed, as well as in most wireless network deployments. The scheduler sends packets to the interface driver until either the channel queue is empty (in which case T_{fin} is set to the maximum of its current value and T_{min}), or T_{fin} exceeds T_{max}. At this time, a timer is set to expire after T_{fin} duration, if packets are pending for any other channel. When the timer expires, if some other channel has queued packets, then the interface may have to be switched to service that channel.

Before the interface is actually switched, the device driver is queried to see if all packets that had earlier been delivered to the driver, have been transmitted. Such a querying interface is not common in most wireless drivers, and we have built a custom querying interface in the device driver that we use (details are in Chapter 5). If some packets are still pending, the actual switch is deferred for some more time (for T_{defer} time). The driver flushes its queue when a switch is requested. Therefore, deferring switching allows

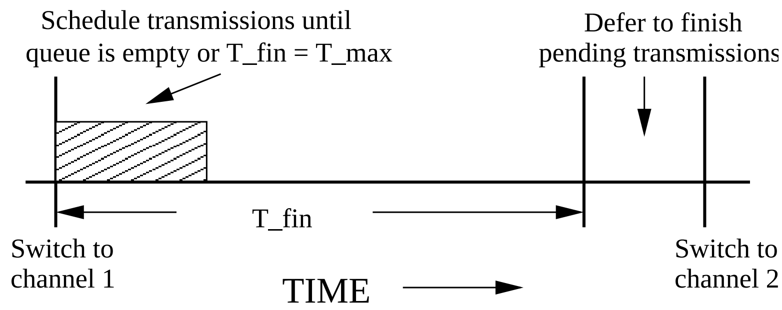

Figure 4.3 Timeline of scheduling operations: Example shows scheduling process after switching into channel 1.

any pending packets to be sent out. This technique is required to compensate for transmission time estimation inaccuracies and variable network conditions. After deferral, the interface is switched, in round-robin order, to the next channel that has buffered packets. The number of deferrals was limited because of conditions which may delay transmissions beyond reasonably expected time. For example, if a particular channel experiences heavy interference due to an external signal source, the interface may try multiple times to transmit each packet, thus keeping all other packets in queue waiting indeterministically. Also, in the event of a node failure, head of line blocking by data destined to such a host may cause extremely long waits. Though such conditions are uncommon, we nevertheless limited deferrals to avoid possibly long waits. By default, the maximum number of channel switch deferrals has been fixed at two, and may be optimized further.

4.3.4 Statistics component

This component collects channel usage statistics for different channels. Statistics are collected on number of packets sent on each channel, average service time per channel, and so on. Most of this information is exported through the *proc* filesystem. A fixed subset of the statistics, which may be of interest to higher layer multichannel protocols, is also exported via an ioctl call. By default, when queried for a specific wireless interface (via an ioctl), channel usage in terms of milliseconds per channel for all valid channels is returned. Choice of the information to export can be easily incorporated as per requirements of the higher layer protocol, with a small change in the code. These statistics can be used by higher layer protocols to do intelligent channel assignment, route selection, etc.

CHAPTER 5

DRIVER MODIFICATIONS

In this chapter we will discuss modifications which were made to the wireless driver used in our testbed. These changes are required to derive reasonable performance from multichannel protocols which rely on frequent channel switching. The modifications and additions made should be of interest for other multichannel protocol implementations.

The channel abstraction module's design was made to allow for its use with any existing Linux wireless network device driver. However, without making some driver modifications, the switching delay could be excessive. Some changes were required to the 802.11 protocol implementation for suitable use in the multichannel scenario. Furthermore without specific feedback, it is possible for the scheduling component to prematurely switch to a new channel, while still having some packets pending to be delivered in the previous channel. To overcome these challenges the driver had to be suitably modified. In the next few sections we will present more details about these changes.

As explained in Chapter 3, the wireless interfaces used in our testbed make use of the "madwifi" open source driver, and 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 devices which make use of a software implementation of the time insensitive portions of the 802.11 protocol ("thin" MACs), in the device driver, provided they are available in open source.

5.1 Reducing Channel Switching Delay

An IEEE 802.11 wireless interface operating in the ad hoc mode is associated with two identifiers called the Extended Service Set Identifier (ESSID) and Basic Service Set Identifier (BSSID). BSSID is a 48-bit address used at the MAC layer to uniquely identify a wireless network. ESSID is an alphanumeric string set by an administrator and is informally referred to as the *network name*. On startup, once a node is assigned an ESSID, it searches all possible channels or a specific channel for nodes which may be advertising the same ESSID, by listening for *"beacons."* Beacons are messages periodically transmitted by nodes in ad hoc mode. These messages contain network information such as the ESSID, BSSID, network age, channel number, etc. On successfully receiving a beacon with the same ESSID, the new node learns about the network's BSSID. The same BSSID is used by the nodes for all further communication with nodes in the same network. The ESSID is not used in data communication after determining the network's BSSID. In case of an unsuccessful search, where no nodes advertise the same ESSID, the new node selects a random 48-bit address for the BSSID and starts advertising this via beaconing. The process discussed so far is referred to as 802.11 ad hoc mode network association process. However, due to ambiguity in the 802.11 specification about the ad hoc mode association process, some wireless interfaces may slightly differ or improve on this process.

Association is an infrequent process and is time insensitive. Hence, the madwifi driver implemented most of this functionality in software. This gave us the flexibility to modify it for our needs. When a channel switch command is issued to a wireless interface, it assumes that since we are changing to a new channel, we will be breaking communication with the present wireless network. Hence, it restarts the association phase and listens for beacons advertising the same ESSID. Most wireless interfaces send out beacons at the rate of 10 per second, with a 100-ms gap between each beacon. Therefore, wireless interface cards have to wait for at least 100 ms in anticipation of a beacon. Depending on when a suitable beacon is heard, the amount of time the association phase would take to complete varies from less than a millisecond to hundreds of milliseconds.

In multichannel networks, the assumption that we would disassociate from the present network when we switch to a new channel is inaccurate. Given that such a use was not envisioned by the 802.11 designers, the assumption was previously justified. Because of this, we modified the driver to not change its internal 802.11 "state," other than the channel information, on switching to a new channel. Details on how this was achieved are explained in the next paragraph. Following this change, association is invoked only at interface bring-up time.

In Chapter 3 we briefly described the channel switch call and how it is routed to the actual channel switch function. As explained previously, when a higher kernel layer or user space program issues a channel switch command via an ioctl call, it reaches the function named `ieee80211_ioctl_swifreq()`. In the unmodified version of the driver, this function first checks that the requested channel is within possible frequency range, and finally invokes the `ath_init()` function, which is also invoked on system initialization. Hence, the whole association procedure is restarted after completion of the channel switch. The channel switch is still achieved by a single function named `ath_chan_set()`, which is not exposed to other kernel modules. In the madwifi driver design, the 802.11 protocol is implemented and loaded as a separate module, to which the `ath_chan_set()` function is invisible. This required the implementation of a wrapper in the madwifi driver, named `ath_chan_wrapper()`, responsible only for passing the arguments to `ath_chan_set()` function. This function is now exposed to the 802.11 protocol module in a fashion similar to exposing other device related functions. Following this modification, all channel switch requests are handed directly by the `ath_chan_set()` function. Details of this function have been explained in Chapter 3.

Thus, by by-passing the association process on switching to a new channel, the overall channel switching delay has been reduced in our testbed.

5.2 Avoiding Network Partitioning

IEEE 802.11a defines two modes of operation, managed/infrastructure and ad hoc mode. In the managed mode, there is a clear understanding of the requirements of a BSSID, ESSID and their use in the association phase. Since the managed mode always has a central authority in the form of an access point, mobility will not cause complications. However, in the ad hoc mode, the association process was complicated due to the lack of a central authority. Ideally all nodes, using the same ESSID, in communication range should use the same BSSID value. Under practical network conditions, nodes using the same ESSID (network name) may end up using different BSSIDs, leading to network partitioning. Many reasons such as changing channel conditions, node mobility, faulty beacon processing, etc., cause this problem. In the multichannel network case, nodes which are in communication range may use different channels on powering on, causing network partitioning. Specific changes were required to overcome this problem, as explained in the next few paragraphs.

Commonly proposed solutions for network partitioning involve either totally disregarding the BSSID value, or making sure that all nodes somehow use a preset BSSID value. Some real world experience on this from other testbed projects such as MIT Roofnet were also published [22]. In MIT Roofnet the BSSID value in the 802.11 frame is set to null, and is disregarded. However, to avoid too many changes to the driver, we used the process where all nodes on startup choose a known BSSID value. All nodes which run our multichannel protocols are loaded with this modified driver.

Since BSSID and ESSID values are of no functional significance in our present setup, the requirement to advertise these in beacon packets was also unnecessary. Beaconing involves sending about 10 packets per second per interface. We removed all beaconing from our driver to avoid these unnecessary transmissions. These modifications avoid any possible network partitioning.

5.3 Query Support

In the scheduling policy discussed in Section 4.3.3, we mentioned that we query the underlying driver to know the number of packets yet to be transmitted. This information is used in deciding whether to defer switches. Such information is not available as a directly readable counter. Even if such a counter was present, no standard query interface was defined for such information in the current Linux wireless extensions.

This required us to first extend the driver to maintain a single counter which has information on the number of packets yet to be transmitted. To support this, a counter was incremented every time a packet was sent to the hardware. Similarly, the counter was decremented on the completion, i.e., on the reception of a transmission completion descriptor from the hardware (transmission could have succeeded or failed on the card). This seemingly trivial extension, however, required instrumentation in carefully selected places in the driver to account for all possible driver/device conditions and outcomes of transmissions.

Linux wireless extensions define a function called `get_wireless_stats()` for querying basic interface statistics. This function normally returns wireless specific statistics from the device driver. In the returned data structure, there was an unused field, initially designed to return the number of packets which have been discarded due to fragmentation errors. We now use this field to return the value discussed above. This information is used by the scheduling component to prevent packet losses due to premature channel switching.

CHAPTER 6

MULTICHANNEL PROTOCOL IMPLEMENTATION

In this chapter we will discuss the programming interface a multichannel protocol may use to communicate with our channel abstraction module (CAM). As shown in Figure 4.1, user space multichannel protocols can communicate with CAM to set up various tables and control its behavior. Initially we will discuss the available ioctl interfaces, and then discuss in detail an example interaction between a multichannel protocol and CAM.

6.1 Communicating with the Channel Abstraction Module

In this section we will discuss in detail the available ioctl calls which a user space protocol can use to communicate with our 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. We implemented five new private ioctls in CAM, detailed below.

1. *Add Valid Channel:* On startup, every multichannel protocol is expected to inform the channel abstraction module about the channels it expects to service. This ioctl call reads in a slave interface name, channel number, validates the inputs, and initializes internal data structures and queues for that particular interface and channel pair. At this point, we do not have functionality to delete a valid channel. One can set up the channel abstraction module to not service a channel by controlling entries in the broadcast and unicast tables. However, implementing the delete functionality is trivial.

2. *Unicast Entry:* This ioctl is used to set up the unicast table. It expects a destination IPv4 address, destination channel, and the interface to use to transmit on this channel. Unicast entries can be added, deleted, and updated using this call. Recall that an IPv4 address is the unique key for the unicast table. When an IPv4 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.

3. *Broadcast Entry:* This ioctl is used to set up the broadcast table. It expects a valid channel and the interface to use to service that channel for broadcast packets. For the broadcast table, the wireless channel is the unique key, and semantics to add, update or delete an entry are similar to that of the unicast ioctl.

4. *Switch Channel:* The switch channel ioctl passes a channel switch call to the underlying interface device driver. However, since the channel abstraction module maintains local state on current channel, it is suggested that all multichannel protocols, which would like to request a channel switch on a specific wireless interface, do so using this ioctl call via the channel abstraction module, instead of directly invoking a switch request.

5. *Get Statistics:* Channel abstraction module has a statistics component which exposes channel usage information via the Linux proc filesystem and also via this ioctl call. On passing an interface name, this ioctl returns channel usage in milliseconds per second for each valid channel.

6.2 Hybrid Protocol Implementation

In this section we will discuss the interaction between a multichannel protocol and the channel abstraction module. A hybrid multichannel protocol, proposed earlier in [7, 10], will be used to demonstrate the use of the channel abstraction module. The hybrid protocol requires at least two interfaces at each node. One interface is tuned to a specified "fixed" channel, and the second interface can switch between the remaining channels. All incoming traffic to a node is expected to arrive on the fixed channel, while most outgoing transmissions (except on the fixed channel) are through the switchable interface. Broadcast is supported by sending a copy of each broadcast packet on every channel. Each node advertises its fixed channel using broadcast "hello" packets. The hybrid protocol tries to ensure that the number of nodes using each fixed channel is balanced. More details of the protocol are in [10].

Figure 6.1 presents an example of the interaction between the hybrid protocol, implemented in user space, and the channel abstraction module. The example assumes that three channels (1,2,3) and two interfaces (ath0 and ath1) are available. Initially, the hybrid protocol informs the channel abstraction module of the set of valid channels for each interface through *Add Valid Channel* ioctl call. The hybrid protocol sets up the available interfaces identically, but other multichannel protocols could use different interfaces on different channels. Next, the broadcast table is set up using the *Broadcast Entry* ioctl call. In this example, channel 1 is used as the fixed channel, and interface ath0 will be assigned to channel 1. Therefore, on channel 1, interface ath0 is used to send broadcast

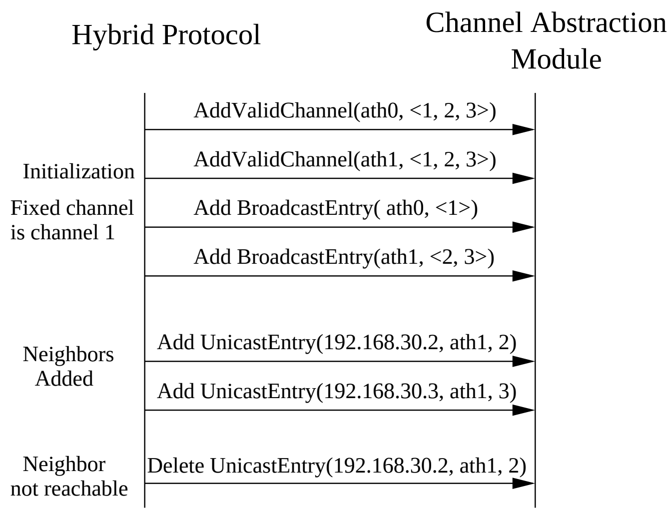

Figure 6.1 Interaction between user space hybrid protocol and kernel channel abstraction module through ioctl calls. Example assumes that three channels (1, 2, 3) and two interfaces (ath0, ath1) are available.

packets, while on channels 2 and 3, interface ath1 is used.

After initialization, when the node receives “hello” packets from a neighbor, it populates the unicast table in the channel abstraction module with the channel to be used to reach the neighbor by invoking the *Unicast Entry* ioctl call. Later, if a neighbor is no longer reachable, then the neighbor’s entry is deleted from the unicast table using *Unicast Entry* ioctl call with a suitable request. The channel abstraction module uses the *Broadcast Entry* ioctl to change the interfaces used to service each channel, should it change its “fixed” channel to a different one. The *Get Statistics* ioctl is used by the routing protocol [10] in optimizing multihop routes.

Using the channel abstraction module significantly simplified the hybrid protocol implementation. We believe that using the generic support offered by the channel abstraction module can simplify the implementation of other multichannel protocols as well.

CHAPTER 7

PERFORMANCE ANALYSIS

In this section we will look at some performance analysis of the scheduling component of our system. The system is running the hybrid multichannel protocol explained in Chapter 6.

7.1 Throughput Measurements

We present results from a 4-node topology, running the Hybrid Protocol, to quantify the impact of switching (Figure 7.1). Every node has two wireless interfaces operating in IEEE 802.11a mode, and the data rate of both interfaces is set to 6 Mbps. One interface at each node is fixed to a channel, and the fixed channels used are shown next to the node labels in Figure 7.1, for instance, node A has one interface fixed on channel 60. The Hybrid protocol is disabled once initial neighbor discovery is complete to avoid any interference with the experimentation. In reality the Hybrid protocol adds very little broadcast traffic overhead (less than one packet per second), but we chose to disable it to prevent any effect on results.

As explained in Section 3.1.2, five channels can be simultaneously used (channels 36, 52, 64, 149, 161) in our testbed in 11a mode. For results presented below, we have used only channels 36, 60, and 149.

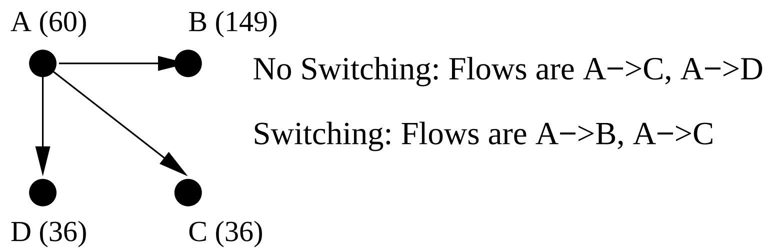

Figure 7.1 Experimental setup.

We use two scenarios to quantify the cost of switching, as shown in Figure 7.1. In the first scenario (called "No switching" scenario), node A sets up two flows to nodes C and D. We perform two experiments, one with flows using UDP, and the other with flows using TCP. The flows are created using Iperf tool [31]. C and D are receiving on channel 36, while A has its first interface fixed to channel 60. Therefore, node A has to use its second interface to send data to C and D. However, since C and D are both on channel 36, the second interface does not need to switch at all, and can stay on channel 36 for the entire duration of communication. In the second scenario (called "Switching" scenario), node A sets up two flows to nodes B and C. Now, since B receives data on channel 149, while C receives data on channel 36, the second interface A has to switch between channels 149 and 36 to service the two flows. This scenario creates frequent interface switching. The difference in the aggregate throughput achieved between the two scenarios is a measure of overheads of switching.

The overhead of switching depends on how frequently interfaces are switched, which in turn depends on the scheduling parameters T_{min} and T_{max} (see Section 4.3.3 for parameter descriptions). Recall that T_{min} specifies the minimum time spent on a channel before a switch can be made, while T_{max} specifies the maximum time allowed on a channel if another channel has pending packets.

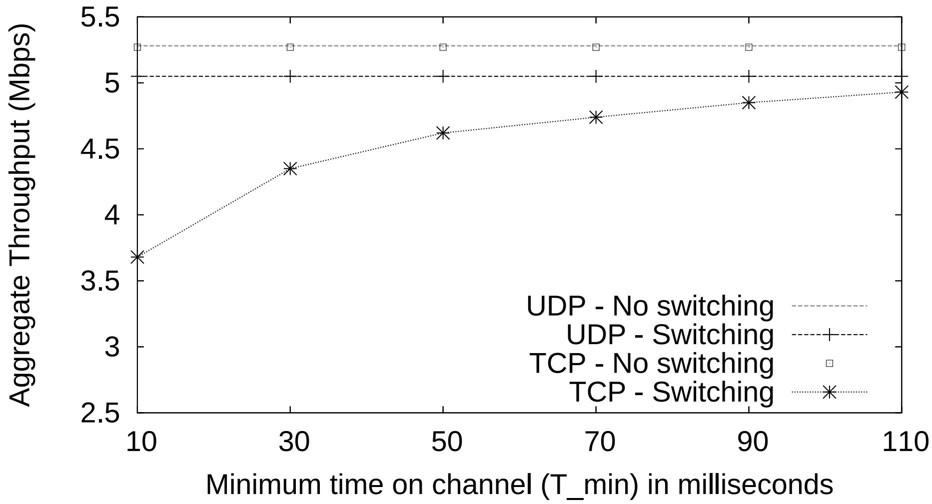

Figure 7.2 Throughput when minimum time spent on a channel is varied. Note that the curves for UDP and TCP with no switching overlap.

Figure 7.2 plots the aggregate throughput with varying T_{min} (T_{max} is set to 130 ms), for both TCP and UDP traffic. Figure 7.3 plots the aggregate throughput with varying T_{max} (T_{min} is set to 10 ms). As we can see from the figures, in the "No Switching" scenario, both TCP and UDP get approximately the same aggregate throughput (the curves overlap in both figures). TCP throughput is close to UDP throughput because the TCP ACK packets use a different channel than the data packets. Also, because interfaces do not have to switch, throughput does not change when either T_{min} or T_{max} is varied.

Note in Figures 7.2 and 7.3 that when there is no switching the aggregate throughput observed is 5.3 Mbps, for both UDP and TCP traffic. That is, when node A is sending data to nodes C and D, each flow gets approximately the same throughput, together adding to 5.3 Mbps. From simple throughput measurement experiments, where Node A is talking directly to Node D, using an unmodified driver, *without* our system in place, the throughput observed for UDP is 5.3 Mbps. TCP throughput observed for the same unmodified case was less than 5.3 Mbps since TCP acknowledgments are being sent on the same channel, while for our system, TCP acknowledgments are sent on another non

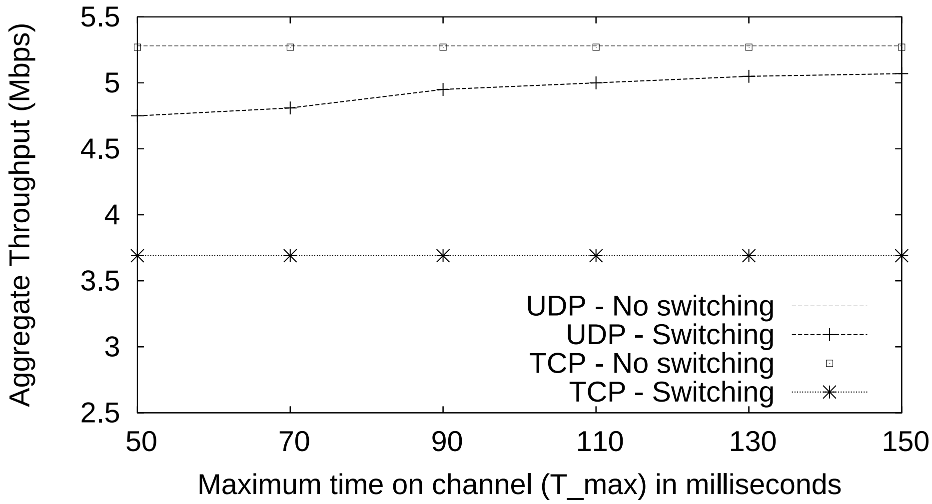

Figure 7.3 Throughput when maximum time spent on a channel is varied. Note that the curves for UDP and TCP with no switching overlap.

interfering channel. This shows that our system *does not add any overhead when not switching channels.*

In the "Switching" scenario with UDP flows, throughput is unaffected when T_{min} is varied. However, UDP throughput varies when T_{max} is varied. The UDP flows that we have set up have sufficient load to saturate the channel. By *saturate*, we mean UDP load is high enough to force the scheduling protocol to stay for maximum possible time (T_{max}) on that particular channel. Therefore, when the second interface switches to a channel to service its flow, there are enough buffered packets to keep the interface on that channel for T_{max} duration. Hence, the number of switches per second, and therefore the switching overhead, depends on the value of T_{max}, as shown in Figure 7.4. Hence, UDP traffic is not affected by T_{min}, and only depends on the value of T_{max}. Note that UDP throughput in the "Switching" scenario is within 5% of the throughput in the "No Switching" scenario when T_{max} is sufficiently high as seen in Figure 7.3. Earlier measurements showed that each switch takes roughly 5 ms. When UDP flows are fully loaded, a switch happens every T_{max} duration. Therefore, the overhead due to switching is proportional to $5/T_{max}$,

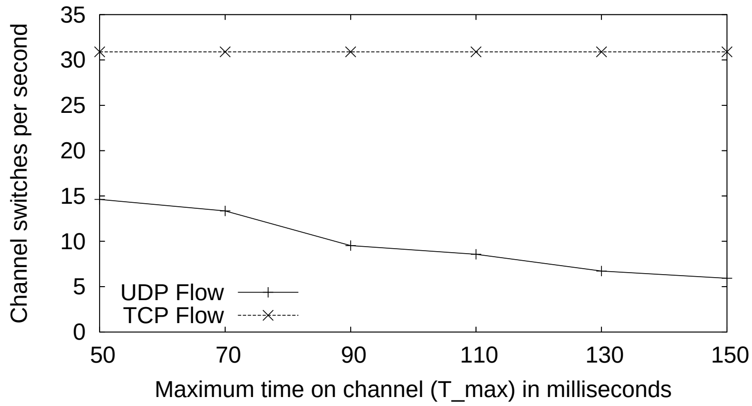

Figure 7.4 Channel switches per second when maximum time spent on a channel is varied.

and this is approximately the observed switching overhead. This shows that our scheduling algorithm is efficient for saturated UDP traffic, implying that our system *does not add a measurable overhead when switching.*

In the "Switching" scenario with TCP flows, throughput is quite sensitive to the value of T_{min}, but not as sensitive to the value of T_{max}. In steady state, TCP sends a new packet only after receiving the ACK of an earlier packet, spacing out packet transmissions over time. Upon switching to a channel, there may only be a few packets buffered for transmission and most switches happen after T_{min} duration. Therefore, with TCP traffic, the number of switches depends on T_{min}, as shown in Figure 7.5, and using larger T_{min} reduces the number of switches, thereby improving aggregate throughput.

These experiments suggest that for improving TCP performance larger T_{min} is suitable, while for improving UDP performance larger T_{max} is suitable. Since these two requirements are not contradictory, it may seem like the optimal choice is to use large T_{min} and T_{max} values. However, end-to-end delay goes up as T_{min} and T_{max} are increased, and large values may not be appropriate with delay-sensitive traffic, and in certain cases for TCP traffic as well. Detailed delay measurements and identifying good parameter values

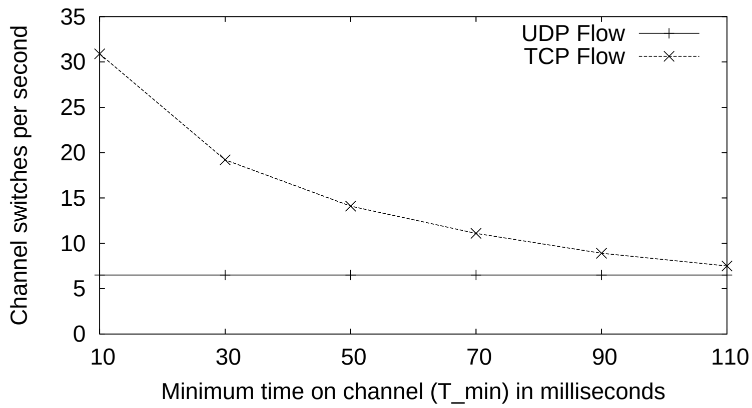

Figure 7.5 Channel switches per second when minimum time spent on a channel is varied.

for delay-sensitive traffic is ongoing.

Results from the same 4-node topology, using different flows, are presented below. These results illustrate the throughput benefits of using multichannel protocols. In one set of experiments, two channels are available (IEEE 802.11a channels numbered 36 and 149), while in the second set of experiments, four channels are available (IEEE 802.11a channels numbered 36, 60, 149, and 161). This increase in number of channels used shows an increase in performance while not breaking connectivity, as seen below.

Figure 7.6 plots the aggregate network throughput in a 4-node topology for 2-channel and 4-channel scenarios. All nodes are in the transmission range of each other. We set up UDP flows using the Iperf tool [31]. For each channel scenario, we run experiments with 4-flows and 8-flows. The flows used for the 4-flow case are A to B, B to C, C to D, and D to A as shown in the figure. In the 8-flow case, each node is sending data to two of its neighbors, as shown in the figure. The hybrid protocol balances fixed channel assignment of nodes (after a few iterations of the protocol). Therefore, in the 2-channel scenario, there are two nodes per channel, while in the 4-channel scenario there is one node per channel. In the 2-channel scenario, since there are 2 interfaces per node, in-

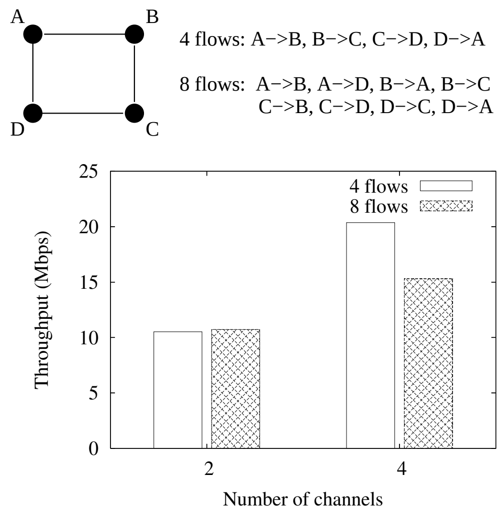

Figure 7.6 Testbed throughput in a 4-node scenario with varying number of channels and flows.

terface switching is not required. In the 4 channel scenario, when each node has one outgoing flow (4-flow scenario), again interface switching is not required (except for occasional broadcast packets). However, in the 4-channel, 8-flow scenario, each node has to frequently switch interfaces to send data along its two outgoing flows.

As we can see from Figure 7.6, in the 4-flow scenario, using 4 rather than 2 channels, doubles the aggregate network throughput. In this scenario, interface switching is not required, and the result indicates that our architecture does not add unnecessary overheads when switching is not needed. For the 8-flow scenarios, using 4 instead of 2 channels, provides approximately 50% throughput improvement. The throughput improvement in this scenario is lower than the expected 100% for two reasons. First, the 8-flow setup requires frequent interface switching as T_{max} is set to 50 ms even though the UDP flows can saturate the channel. Second, each node has only one channel for it to *receive* on; that is, when two nodes, say B and D, would like to send data to A, they need to do so on A's receiving channel, causing contention. Since the Hybrid protocol does not control when nodes can talk on a channel, it is possible for two nodes to contend for a channel for a given period of time, while later both may switch channels to service their other flows. This can create periods of heavy traffic on each node's *incoming channel*, followed by *possible* idle periods. Though this specific case shows the Hybrid protocol in a bad light, it still gives a 50% increase in aggregate throughput. Therefore, this result indicates that multichannel protocols can in practice *increase overall throughput of a system* by using multiple channels.

7.2 Delay Measurements

In the previous section, we saw how throughput is affected by switching, and the value of T_{min} and T_{max} that may be suitable for better performance. In this section, we will study the same topology, from the perspective of delay experienced by the traffic while using our system. Delay measurements are important for real time and multimedia traffic, which is typically very delay-sensitive.

Before we present the results, we would like to explain the expected delay when using our system with two flows, requiring switching. Of the two flows, assume that one, a UDP flow, is saturating and can force the scheduling component to stay on its destination channel for T_{max} duration. The second flow, an ICMP echo request and reply flow (generated using "ping"), is a low rate flow and requires between T_{min} and T_{max} time to service it. A timeline is shown in Figure 7.7. In the diagram, the packets P_i and P_j are used to refer to the first and the last ICMP packets queued because the interface is currently servicing a different channel. The ICMP packets received from user space after packet P_i and before P_j all experience some queueing delay. Observe that the extra queueing delay is highest for packet P_i, and least for P_j, decreasing in that order. The amount of delay added to packets arriving between P_i and P_j depends on the value of T_{max}. Therefore, maximum delay is expected to be affected mainly by the T_{max} parameter. Packets arriving after P_j and before switching channel to service the UDP flow

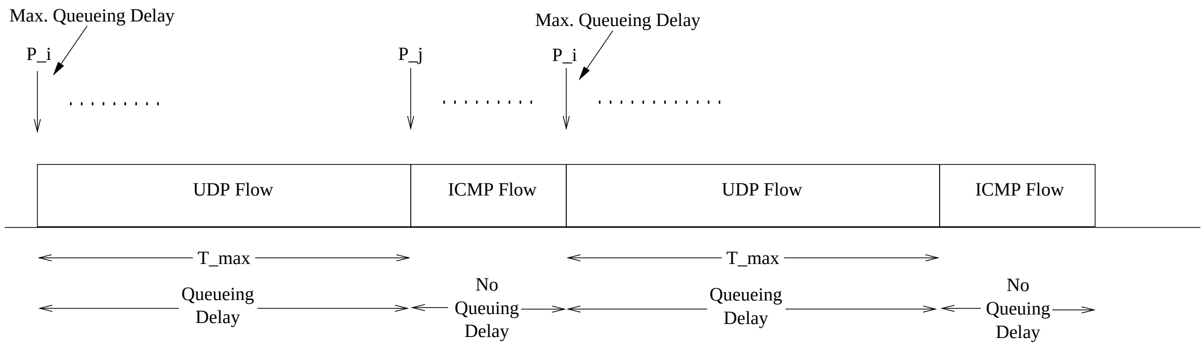

Figure 7.7 Timeline of queueing to understand experienced packet delay.

again, do not experience extra queueing delay, given that all queued ICMP packets can be transmitted before switching. The amount of delay experienced by ICMP packets which arrive while servicing this flow is the least possible delay. Therefore, decreasing the chance of queueing ICMP packets decreases the average delay of this flow. More packets are likely to be transmitted directly without any queueing, when the value of T_{min} is large. Thus, average queueing delay is effected by the T_{min} parameter.

To understand the effect of T_{min} and T_{max} parameters on delay, we studied the same 4-node topology using one heavily loaded UDP flow along with one lower load ICMP flow. Node A, as shown in Figure 7.1, has to switch between channels 36 and 149 to communicate with nodes C and B, respectively. We set up a UDP flow from node A to node B, which is capable of saturating the channel, i.e., the flow is heavy enough to force node A to stay on channel 149 for T_{max} duration. Such a flow would create the worst possible delay for any flow on channel 36, i.e., the flow to node C. We used the common Unix utility *ping* to generate ICMP traffic on channel 36. Typically, ping gives us a measure of round-trip time. However, in our multichannel protocol, the communication to C from A uses channel 36, while the reply packet comes on channel 60, which has no traffic on it, and has a delay of approximately 1 ms. Hence, in this case the round-trip time reported by ping is dominated by the delay from A to C.

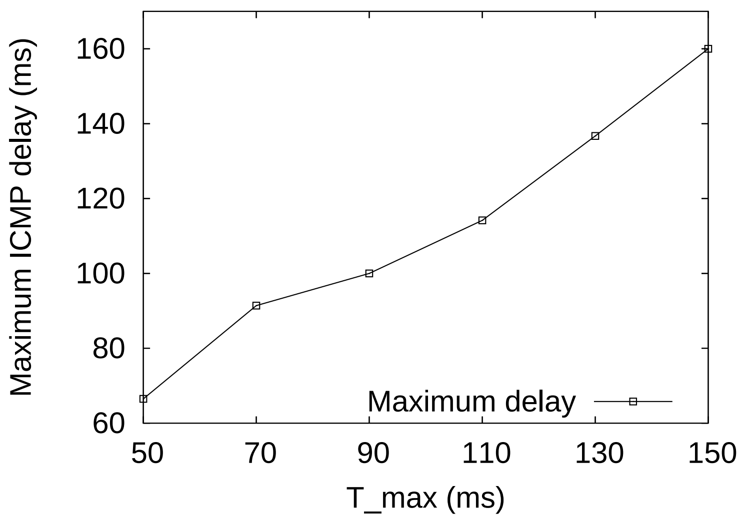

Figure 7.8 Packet delay when varying maximum time on a channel.

In Figure 7.8 we set the T_{min} value to 30 ms on node A and plot the observed *maximum* delay for different values of T_{max} at node A. Ping returns various statistics from observed delays including maximum delay experienced. However, due to wireless errors, some packets may experience delays which are significantly above the expected value. Such numbers are outliers, and do not reflect the delay introduced by the protocol. Hence, such values are discarded. After discarding these outliers, there are multiple delay values which are within 10 ms of the maximum delay value, thus validating our confidence in the maximum delay value. Those maximum delay values are plotted in the Figure 7.8 for each T_{max}. Recall that maximum queueing delay is experienced by ICMP packets which arrive just after switching to service the UDP channel, as shown in Figure 7.7. If the UDP load is too high, the queues in the kernel may become too large, and the kernel starts dropping packets. To avoid this, multiple trials were made to choose an appropriate UDP load to saturate the time on channel. As expected, the consistently observed maximum delays, plotted in Figure 7.8, show a direct relation to T_{max}. This result indicates that *higher T_{max} results in higher maximum queueing delay.*

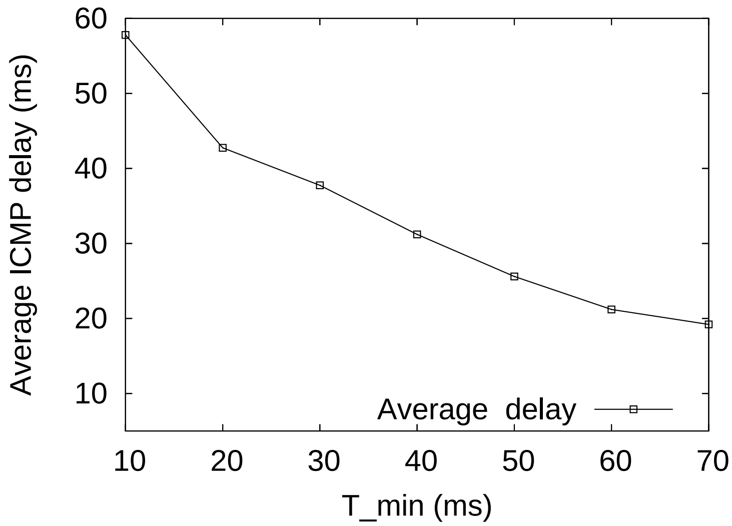

Figure 7.9 Packet delay when varying minimum time on a channel.

In the second experiment, we tried to measure the average ICMP delay experienced while varying the value of T_{min}. On node A we set T_{max} to 100 ms and varied T_{min}, as seen in Figure 7.9. An increase in T_{min} decreases the average delay experienced by the ICMP packets. The maximum experienced delay remains constant at 120 ms (not shown in figure), as T_{max} is not varied. As stated above, UDP load was chosen to saturate the time on channel, while ensuring ICMP packets were not dropped. In conclusion, average queueing delay is effected primarily by T_{min}, as long as the difference between T_{min} and T_{max} is not too large.

7.3 Summary

In summary we would like to restate the main observations made in this chapter. From initial throughput results, it can be concluded that our system does not add any extra overhead when not switching between channels, suggesting that our system does not perform worse than an unmodified 802.11 based system. Under heavy switching, given suitable values of T_{min} and T_{max}, the overheard added by our system can be less than

10%. Even under heavy switching, and a suboptimal scenario where the multichannel protocol may not achieve best performance, our system still achieves a 50% gain over the base case when number of channels is doubled, supporting its practical use. Delay experiments indicate that average packet delay and maximum packet delay are dependent on T_{min} and T_{max}.

A question which has been repeatedly posed in our analysis was the suitable choice of T_{min} and T_{max}. Throughput experiments, under symmetric loads, seem to be suggest that larger T_{min} and T_{max} values are better. However, under asymmetric loads, such as those used in the delay experiments, the flow with lower load may not be able to keep the channel busy for T_{min} duration, leading to channel under-utilization. Hence, a very large T_{min} may also be a poor choice. From our measurements presented in Chapter 3, it is clear that every channel switch costs roughly 5 ms. Therefore, if a switch happens too frequently, say, more than once every 20 ms, the switching overhead will be high. Hence, a T_{min} of 20 ms to 40 ms is suggested, depending on traffic patterns. Under sufficiently high load, the value of T_{max} is of importance, and larger values have a lower overhead with respect to throughput. However, in the delay experiments, higher T_{max} results in higher maximum delay. Hence, a value of T_{max} between 100 ms to 140 ms is suggested, depending on the network load expected.

CHAPTER 8

CONCLUSION AND FUTURE WORK

We designed and implemented an architecture for supporting multichannel multi-interface protocols. This thesis identified shortcomings in the present day operating systems in implementing multichannel protocols. A software architecture for overcoming has been presented. In the implementation process, many related issues pertaining to the 802.11 protocol and its incompatibilities when used in a multichannel scenario were resolved. Experimentation has shown that the extra layer of functionality does not worsen network performance, even when using devices with low processing power such as those in our study. An example multichannel protocol proposed by Kyasanur and Vaidya in [10] was implemented, and its interaction with our system was explained. This thesis showed that it is possible to implement multichannel protocols which require interface channel switching with relatively low overheads using existing hardware.

Goal of the system presented in this thesis was to enable multichannel protocol research. However, no initial architecture would be holistic enough to support all possible multichannel protocols. Our system was developed to address the issues presented in Section 3.2. Other multichannel protocols may present a different set of requirements. A few such protocols may require changes to hardware, which is out of the scope of software architectures like ours, but some may warrant for extensions to our system. Such extensions could be a possible future direction for our system.

Future wireless chipsets are expected to support faster switching between wireless channels. This feature is informally referred to as "*fast switching.*" The ability to switch channels considerably faster than what is possible at present could change some of the assumptions made, and impact both our system architecture and multichannel protocol design. It is necessary to carefully follow this development and exploit such abilities to full potential in the future.

8.1 Suggestions for Future Hardware Design

In our study of multichannel protocols we made use of 802.11a based wireless interfaces, which provide for 12 different channels. But experimentation proved that when the interfaces are very close (within a few inches) to each other, it would be impractical to use more than 3-5 channels depending on the distance between the transmitter and the receiver. This is because of the leakage of power into side-bands with filters used in current cards. Communication with other practitioners suggests that the filter used was considerably better than what is recommended by the 802.11a specification. This showed that the design of the 802.11 protocol never envisioned such an use, and hence manufacturers had no reason or advantage in implementing better filters. It may however be possible to allow for much better channel diversity with the use of better filters.

REFERENCES

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

[2] A. Raniwala, K. Gopalan, and T. Chiueh, "Centralized channel assignment and routing algorithms for multi-channel wireless mesh networks," *Mobile Computing and Communications Review*, vol. 8, pp. 50–65, April 2004.

[3] A. Raniwala and T. Chiueh, "Architecture and algorithms for an IEEE 802.11-based multi-channel wireless mesh network," in *INFOCOM 2005. Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, March 2005, pp. 2223– 2234.

[4] S.-L. Wu, C.-Y. Lin, Y.-C. Tseng, and J.-P. Sheu, "A new multi-channel MAC protocol with on-demand channel assignment for multi-hop mobile ad hoc networks," in *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN)*, 2000, pp. 232–237.

[5] J. So and N. H. Vaidya, "Multi-channel MAC for ad hoc networks: Handling multi-channel hidden terminals using a single transceiver," in *MobiHoc '04: Proceedings of the 5th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, 2004, pp. 222–233.

[6] P. Bahl, R. Chandra, and J. Dunagan, "SSCH: Slotted seeded channel hopping for capacity improvement in IEEE 802.11 ad-hoc wireless networks," in *MobiCom '04: Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, 2004, pp. 216–230.

[7] P. Kyasanur and N. H. Vaidya, "Routing and interface assignment in multi-channel multi-interface wireless networks," in *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, vol. 4, March 2005, pp. 2051–2056.

[8] P. Kyasanur and N. H. Vaidya, "Capacity of multi-channel wireless networks: impact of number of channels and interfaces," in *MobiCom '05: Proceedings of the 11th Annual International Conference on Mobile Computing and Networking*, 2005, pp. 43–57.

[9] M. Kodialam and T. Nandagopal, "Characterizing the capacity region in multi-radio multi-channel wireless mesh networks," in *MobiCom '05: Proceedings of the 11th Annual International Conference on Mobile Computing and Networking*, 2005, pp. 73–87.

[10] P. Kyasanur and N. H. Vaidya, "Routing and link-layer protocols for multi-channel multi-interface ad hoc wireless networks," *Sigmobile Mobile Computing and Communications Review*, vol. 10, pp. 31–43, Jan. 2006.

[11] M. A. Marsan and F. Neri, "A simulation study of delay in multichannel CSMA/CD protocols," *IEEE Transactions on Communications*, vol. 39, pp. 1590–1603, November 1991.

[12] A. Nasipuri, J. Zhuang, and S. Das, "A multichannel CSMA MAC protocol for multihop wireless networks," in *IEEE Wireless Communications and Networking Conference (WCNC)*, vol. 3, Sept. 1999, pp. 1402–1406.

[13] A. Nasipuri and S. Das, "Multichannel CSMA with signal power-based channel selection for multihop wireless networks," in *Proceedings of the 52nd IEEE Vehicular Technology Conference*, vol. 1, Sept. 2000, pp. 211–218.

[14] N. Jain, S. Das, and A. Nasipuri, "A multichannel CSMA MAC protocol with receiver-based channel selection for multihop wireless networks," in *Proceedings of the 10th International Conference on Computer Communications and Networks*, October 2001, pp. 432–439.

[15] A. Adya, P. Bahl, J. Padhye, A. Wolman, and L. Zhou, "A multi-radio unification protocol for IEEE 802.11 wireless networks," in *Proceedings of the First International Conference on IEEE International Conference on Broadband Networks (Broadnets)*, 2004, pp. 344–354.

[16] R. Draves, J. Padhye, and B. Zill, "Routing in multi-radio, multi-hop wireless mesh networks," in *MobiCom '04: Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, 2004, pp. 114–128.

[17] A. Miu, H. Balakrishnan, and C. E. Koksal, "Improving loss resilience with multi-radio diversity in wireless networks," in *MobiCom '05: Proceedings of the 11th Annual International Conference on Mobile Computing and Networking*, 2005, pp. 16–30.

[18] R. Chandra and P. Bahl, "Multinet: Connecting to multiple IEEE 802.11 networks using a single wireless card," in *INFOCOM 2004. Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 2, Hong Kong, March 2004, pp. 882–893.

[19] P. Stuedi and G. Alonso, "Transparent heterogeneous mobile ad hoc networks," in *Proceedings of the 2nd Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, 2005, pp. 237–246.

[20] N. Boulicault, G. Chelius, and E. Fleury, "Experiments of ana4: An implementation of a 2.5 framework for deploying real multi-hop ad-hoc and mesh networks," in *REALMAN '05: Proceedings of the 2005 Realman Workshop*, 2005, pp. 19–26.

[21] V. Kawadia, Y. Zhang, and B. Gupta, "System services for ad-hoc routing: Architecture, implementation and experiences," in *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003, pp. 99–112.

[22] J. Bicket, D. Aguayo, S. Biswas, and R. Morris, "Architecture and evaluation of an unplanned 802.11b mesh network," in *MobiCom '05: Proceedings of the 11th Annual International Conference on Mobile Computing and Networking*, 2005, pp. 31–42.

[23] H. Lundgren, D. Lundberg, J. Nielsen, E. Nordstrm, and C. Tschudin, "A large-scale testbed for reproducible ad hoc protocol evaluations," in *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, vol. 1, Mar. 2002, pp. 412–418.

[24] D. Raychaudhuri, I. Seska, S. G. M. Ott, K. Ramachandra, H. Krem, R. Siracus, H. Liu, and M. Singh, "Overview of the orbit radio grid testbed for evaluation of next-generation wireless network protocols," in *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, vol. 3, Mar. 2005, pp. 1664–1669.

[25] B. Raman and K. Chebrolu, "Design and evaluation of a new MAC protocol for long-distance 802.11 mesh networks," in *MobiCom '05: Proceedings of the 11th Annual International Conference on Mobile Computing and Networking*, 2005, pp. 156–169.

[26] K. Ramachandran, E. Belding-Royer, and K. Almeroth, "Overview of the orbit radio grid testbed for evaluation of next-generation wireless network protocols," in *Proceedings of the First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks(SECON)*, Oct. 2004, pp. 601–609.

[27] "AR5212 Chipset from Atheros Communications Inc.," 2005, http://www.atheros.com.

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

[29] "Openhal," October 2005, http://pdos.csail.mit.edu/~jbicket/openhal/.

[30] "Hardware model Net4521 from Soekris Engineering, Inc.," 2005, http://www.soekris.com/net4521.htm.

[31] "Iperf version 2.0.2," May 2005, http://dast.nlanr.net/Projects/Iperf/.