Design and Implementation of a Multi-Channel Multi-Interface Network

Chandrakanth Chereddi, Pradeep Kyasanur, and Nitin H. Vaidya
University of Illinois at Urbana-Champaign
cchered2@uiuc.edu, kyasanur@uiuc.edu, nhv@uiuc.edu

ABSTRACT

The use of multiple wireless channels has been advocated as one approach for enhancing network capacity. In many scenarios, hosts will be equipped with fewer radio interfaces than available channels. Under these scenarios, several protocols, which require interfaces to switch frequently, have been proposed. However, implementing protocols which require frequent interface switching in existing operating systems is non-trivial. In this paper, we identify the features needed in the operating system kernel for supporting frequent interface switching. We present a new *channel abstraction* module to support frequent interface switching. We identify modifications to interface device driver to reduce switching delay. The channel abstraction module, and an example multi-channel protocol that uses the module, have been implemented in a multi-channel multi-interface testbed. We also present results to quantify the overheads of frequent switching.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*

General Terms

Measurement, Design, Experimentation

Keywords

Wireless Testbed, Multiple Channels, Multiple Interfaces, Interface switching, Mesh networks

1. INTRODUCTION

Wireless technologies, such as IEEE 802.11a [8], provide for several non-overlapping channels. Several researchers have proposed the use of multiple wireless channels for enhancing network capacity. 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. 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 [19,20]. 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 [3,14,21,25]. This *interface switching* approach allows channel assignment to be dynamically done based on node density, traffic, channel conditions, etc., and has shown to be a good choice in theory [12,13]. However, frequently switching interfaces[1] introduces extra implementation complexity.

Building multi-channel protocols that require frequent interface 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. Implementing frequent switching requires new support in the operating system kernel. In this paper, we present the design and implementation of a new *channel abstraction* module in the kernel that simplifies the implementation of multi-channel protocols that require interface switching. Switching an interface incurs a delay, and the magnitude of the delay impacts the viability of frequent interface switching. We will describe the driver modifications we made to reduce the switching delay to around 5 milliseconds, so as to make frequent switching feasible, and present results to quantify the impact of switching delay on network throughput.

The development of the abstraction module was motivated by our efforts to implement a *hybrid multi-channel* protocol [14, 15] that we had 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 nec-

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

essary. A node transmits a packet to a neighbor on the fixed channel of its neighbor. This protocol was shown to be fairly effective in utilizing multiple channels when there are fewer interfaces per host than channels [15]. By using the *channel abstraction* module that we have developed, the hybrid protocol was implemented as a simple user space daemon. We will outline the hybrid protocol implementation, which serves as one example use of the channel abstraction module.

The rest of the paper is organized as follows. We present related work in Section 2. Section 3 identifies the new features needed for supporting multi-channel protocols that require interface switching. Sections 4 and 5 present the system architecture and implementation details of the channel abstraction module. Section 6 describes modifications made to the device driver to reduce switching delay. Section 7 presents the implementation of the hybrid multi-channel protocol using the channel abstraction module. Section 8 presents experimental results, and we conclude in Section 9.

2. RELATED WORK

Several protocols have been proposed for utilizing multiple channels. Most protocols [1, 7, 10, 16–18] require each node to have one interface per channel (otherwise, 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 [7, 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 [3, 14, 19–21, 25] have mostly been evaluated in a simulation environment.

Raniwala et al. [19] 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.

“VirtualWifi” [6, 24] is an virtualization architecture that abstracts a single wireless interface into multiple virtual interfaces. VirtualWifi has support for switching the physical interface across the channels used by each virtual interface. VirtualWifi has some similarity to the channel abstraction module that we propose, but does not offer all the features necessary for controlled switching among multiple channels. VirtualWifi 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 [5, 11, 23]. However, those approaches are not designed 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. However, to the best of our knowledge, all those implementations require non-trivial changes to support the use of multiple channels by switching interfaces.

3. KERNEL SUPPORT FOR MULTIPLE CHANNELS

Implementing multi-channel protocols that require interfaces to switch frequently is non-trivial. Our testbed was developed with the goal of using off-the-shelf IEEE 802.11 hardware. Existing hardware does not provide sufficient support for effective interface switching, as elaborated later. Furthermore, we wanted 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.

In this section, we identify the features needed in the kernel, with Linux as an example, for implementing multi-channel protocols with interface switching. In subsequent sections, we present details of our implementation that provides the requisite kernel support.

3.1 Specifying the channel to use for reaching a neighbor

An implicit assumption made in many operating systems is that *each interface is associated with exactly one channel*, i.e., there is an one-to-one mapping between interfaces and channels. This assumption is satisfied in a single channel network where an interface is fixed on the single channel used throughout 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. 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 across channels, the assumption that there is an one-to-one mapping between channels and interfaces is broken.

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 1. 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 (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, 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 multi-channel protocols which often use a *single* interface to send data over *multiple* channels.

3.2 Specifying channels to use for broadcast

In a single channel network, broadcast packets sent out on the wireless channel are typically 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 multi-channel network, different nodes may be listening to different channels. Therefore, to ensure that broadcast packets in a multi-channel network reach (almost) all the nodes that would have received the packet in a single-channel net-

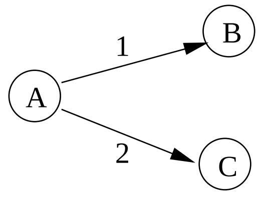

Figure 1: Example illustrating the lack of kernel support for multi-channel protocols.

work, copies of the broadcast packet may have to be sent out on multiple channels. For example, in Figure 1, node A will have to send a copy of any broadcast packet on both channel 1 and channel 2 to ensure its neighbors B and C receive the packet.

There are several existing applications that 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 existing kernel to specify on which channels broadcast packets have to be sent out on, 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 multi-channel broadcast.

3.3 Support for interface switching

As we discussed earlier, interfaces may have to be switched between different channels to enable communication among neighboring nodes that are 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 interface tuned to channel *c*. Suppose 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, then an immediate switch may result in the loss of 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. Switching an interface incurs a non-negligible delay (around 5 ms in our testbed), and too frequent switching may significantly degrade performance. Therefore, there is a need for a queuing algorithm to buffer packets, as well as a scheduling algorithm to transmit buffered packets using a policy that reduces frequent switching, yet ensures queuing delay is not too large.

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

4. SYSTEM ARCHITECTURE

In this section, we present the design choices we made to provide in-kernel support for interface switching, and outline the system architecture.

4.1 Design choices

The Linux kernel's networking stack is organized into multiple layers to ease implementation and improve extensibility. For example, IP belongs to the network layer, while the device drivers that control access to the interface hardware are part of the link layer. The key design question was to identify the layer where interface switching support could be added. Interface switching support requires close interaction with the interface device driver. Based on this requirement, we had three possible locations for adding interface switching support:

1. Add interface switching support directly into the device driver. This approach offers the most control in accessing the interfaces, but has two main drawbacks. First, this approach ties in our implementation with a specific device driver. Second, multiple interfaces cannot be cleanly handled within the device driver of a single interface.

2. Add interface switching support into the network layer (for example, as a "Netfilter" hook). This approach insulates the implementation from the specifics of device drivers. However, multiple interfaces are visible to the network layer, and this may require modifications to some protocols that are at (or below) the network layer (such as ARP).

3. Add interface switching support as a new module that operates between the network layer (and ARP) and the device drivers. The module may be logically viewed as belonging to the link layer. This approach has the benefit of being insulated from device driver specifics, while allowing us to present a single virtual interface to the network layer. The virtual interface can abstract multiple interfaces that may be actually available, and insulates the network layer from knowing the details of the number and types of interfaces. We choose this approach, and implement a new *channel abstraction* module.

The option we chose has some additional benefits. Linux already has the ability to "bond" multiple interfaces into a single virtual interface using a link layer "bonding driver" that resides between the network layer and device drivers. The bonding driver is typically used for grouping multiple ethernet-based devices into a single virtual device. The bonding drivers offers features that allow for load balancing (striping) over the available interfaces, interface fail-over support, etc. There is also a set of user space tools which support management operations, such as specifying which real interfaces to group into a single virtual interface. We implemented the *channel abstraction* module as a new feature of the bonding driver.

4.2 Architecture overview

Figure 2 depicts the system architecture. As we can see from the figure, the channel abstraction module resides between the network layer and the interface device drivers. We have implemented the hybrid multi-channel protocol [14,15] as an user space daemon. The user space daemon interacts with the channel abstraction module using *ioctl* calls. We also made a few modifications to the interface device

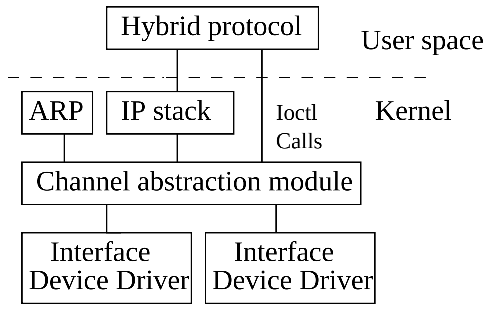

Figure 2: System architecture.

driver to reduce switching delay, and to improve scheduling efficiency (details are in Section 6). All existing user applications, and protocols in the kernel above the link layer, are unaware of the use of multiple channels, multiple interfaces and interface switching.

5. CHANNEL ABSTRACTION MODULE

In this section, we will describe the channel abstraction module. The module is implemented as a new feature of the bonding driver present in the Linux kernel. Figure 3 shows the key components of the channel abstraction module:

- Unicast component: Enables specifying the channel to use to reach a neighbor.
- Broadcast component: Provides support for sending broadcast packets over multiple channels.
- Scheduling and queuing component: Supports interface switching by buffering packets when necessary, and scheduling switching across channels.

The details of the components are presented below.

5.1 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 3. 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 an user space multi-channel protocol via *ioctl* calls (entries can be added or deleted). We will describe in Section 7, with an example, the approach used by the hybrid multi-channel 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 the interface to use for reaching the destination. After this, the packet is handed off to the queuing component for subsequent transmission.

5.2 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 3. The broadcast table maintains a list of channels on which copies of a broadcast packet have to be sent out on, and the interfaces to use for sending out the copies. The table is populated by an user space multi-channel 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, protocols that send a copy of broadcast packet 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 queuing component.

5.3 Scheduling and queuing component

The scheduling and queuing 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 3. The user space multi-channel protocol, on startup, can specify the list of channels supported by each interface using *ioctl* calls. This architecture allows different interfaces to support a possibly different set of channels.

The queuing 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 at least T_{min} duration. 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 T_{max} duration ($T_{max} > T_{min}$).

Figure 4 describes the scheduler operation. The scheduler maintains an estimate T_{fin} of the time needed to transmit packets it has already given to the interface device driver (these packets are stored in a separate queue within the device driver). 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. The estimate is derived based on the size of the packet and the transmission data rate (we ignore channel contention as it is not critical to have very accurate estimates). The scheduler sends out 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

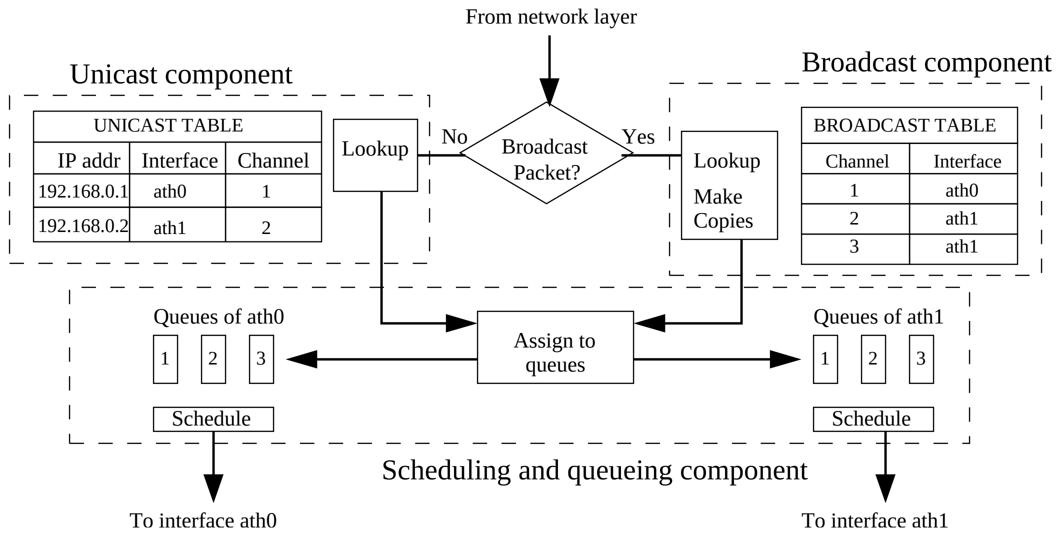

Figure 3: Components of channel abstraction module: Tables are filled with the assumption there are two interfaces "ath0", and "ath1" and 3 channels.

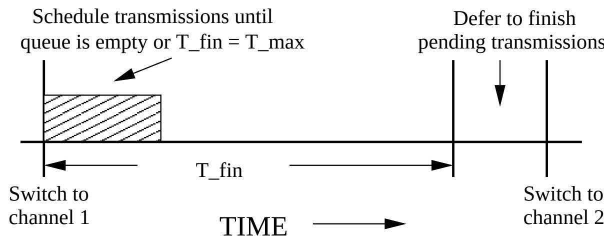
Schedule transmissions until queue is empty or T_fin = T_max
Defer to finish pending transmissions
T_fin
Switch to channel 1
Switch to channel 2
TIME

Figure 4: Example time line of scheduling.

other channel. When the timer expires, if some other channel has queued packets, then the interface may have to be switched.

Before the interface is actually switched, the device driver is queried to see if all packets, which had been given to the driver since the last switch, 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 Section 6). If some packets are still pending, the actual switch is deferred for some more time (for T_{defer} time, currently set to 10 ms). The driver flushes its queue when a switch is requested. Therefore, deferring switching allows any pending packets to be sent out. After deferral, the interface is switched to the next channel, in round-robin order, that has buffered packets.

The scheduling component also collects the channel usage statistics for different channels. This information is exported through the *proc* filesystem, and can also be accessed through *ioctl* calls. The statistics can be used by higher layer multi-channel protocols to do intelligent channel assignment, route selection, etc.

6. DRIVER MODIFICATIONS

The channel abstraction module has been designed for use with any existing driver. However, without making some driver modifications, the switching delay could be excessive, and many packets could be lost after a switch (the packets present in the interface driver queue). In this section, we describe the driver modifications that we have implemented to improve performance.

Our testbed uses wireless interfaces that are based on atheros chipsets [2] controlled by "madwifi" open source driver. Our device driver modifications have been made to the madwifi driver. We have not yet looked at the feasibility of making these modifications to other drivers.

6.1 Reducing channel switching delay

An IEEE 802.11 wireless interface operating in the ad hoc mode is associated with two identifiers called the ESSID (set by the administrator), and BSSID (chosen by the node that first came up with that ESSID), and these identifiers are sent out periodically in beacon packets. When a wireless interface, running in the ad hoc mode, switches to a new channel, it is expected to listen for networks which advertise the same ESSID as itself. If no advertisements are heard within a specified time period, then the interface is supposed to create a new network by advertising a different randomly chosen BSSID. This process of listening for beacons and advertising a new BSSID, if necessary, can take up to 100 ms (the time for only switching channels is about 5 ms). Therefore, the overall interface switching delay can be excessive when normal beaconing is used.

In multi-channel protocols, the beaconing procedure after a switch is not really required if all nodes belong to the same network. To reduce the channel switching delay, we changed the behavior of the interface after a channel switch request has been made, so as to not search for any beacons. Instead, at startup, all nodes are initialized with a pre-specified BSSID (in addition to the ESSID). This removes the need for scanning for beacons after the switch. Beacons have been disabled in a similar fashion in some other testbed projects as well [4]. Using this technique, we have reduced the interface switching delay to about 5 ms.

6.2 Query support

As we discussed in Section 5.3, there is a need for the scheduling component to estimate the queue size in the interface driver. To support this, we overloaded a statistics function already provided in Linux wireless device drivers called *get_wireless_stats()*. This function normally returns basic book keeping counters, which are wireless specific. In the returned data structure, there was an unused field, which we now use to return the number of packets which have been handed down to the driver, but have still not been transmitted. This information is used by the scheduling component to prevent packet losses due to premature channel switching.

7. IMPLEMENTATION OF HYBRID MULTI-CHANNEL PROTOCOL

We implement a hybrid multi-channel protocol that we had proposed earlier [14, 15], to demonstrate the use of the channel abstraction module. As we described in Section 1, the hybrid protocol requires two interfaces at each node. One interface is tuned to a specified "fixed" channel, and the second interface can switch between the remaining channels. Broadcast is supported by sending a copy of the broadcast packet on every channel. The hybrid protocol tries to ensure that the number of nodes using each fixed channel is balanced. Each node advertises its fixed channel using broadcast "hello" packets. When a node A wants to send a packet to some node B, then it has to first switch its second interface to the fixed channel of B (if B and A use different fixed channels), and then transmit the packet. More details of the protocol are in [15].

Figure 5 presents an example of the interaction between the hybrid protocol, which has been implemented as an user space daemon, 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 *AddValidChannel* ioctl call. The hybrid protocol sets up the available interfaces identically, but other multi-channel protocols could use different interfaces on different channels. Next, the broadcast table is set up using the *AddBroadcastTable* 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 out broadcast 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 *AddUnicastTable* ioctl call. Later, if a neighbor is no longer reachable, then the neighbor's entry is deleted from the unicast table using *DeleteUnicastTable* ioctl call. The channel abstraction module also exports a *DeleteBroadcastTable* ioctl which may be used when the fixed interface is changed to a different channel. There is also ioctl support for getting channel usage statistics.

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 multi-channel protocols as well.

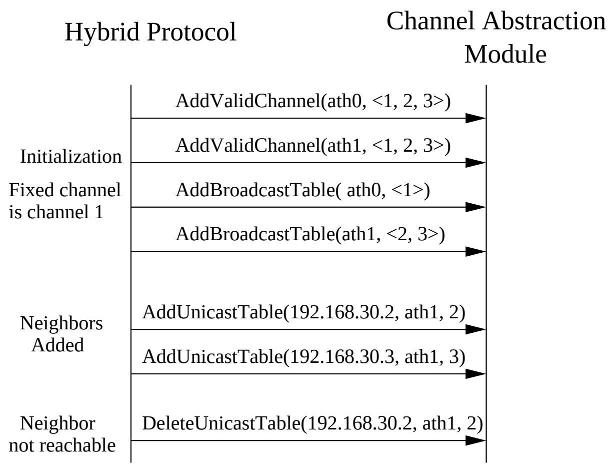
Hybrid Protocol
Channel Abstraction Module
Initialization
Fixed channel is channel 1
AddValidChannel(ath0, <1, 2, 3>)
AddValidChannel(ath1, <1, 2, 3>)
AddBroadcastTable(ath0, <1>)
AddBroadcastTable(ath1, <2, 3>)
Neighbors Added
AddUnicastTable(192.168.30.2, ath1, 2)
AddUnicastTable(192.168.30.3, ath1, 3)
Neighbor not reachable
DeleteUnicastTable(192.168.30.2, ath1, 2)

Figure 5: Interaction between user space hybrid protocol and kernel channel abstraction module through ioctl calls.

8. PERFORMANCE EVALUATION

We have deployed the channel abstraction module and the hybrid protocol on a multi-interface testbed. Currently, the testbed includes more than 20 nodes. The testbed nodes are based on *Net 4521* boxes from Soekris [22]. Each node is currently equipped with two wireless interfaces (one pcmcia interface and one mini-pci interface), and it is possible to have up to 3 wireless radios using this hardware platform. The wireless cards are based on *Atheros* chipset [2], and support IEEE 802.11a/b/g protocols. We are currently deploying the testbed nodes in multiple offices with the goal of doing multi-hop experiments. Here, we present preliminary results on measuring the switching delay, and quantifying the overheads of switching delay.

8.1 Measuring interface switching delay

We have made modifications to the madwifi device driver to reduce switching delay (as discussed in Section 6). Here, we present the methodology we used to measure the switching delay with the modified driver.

When a channel switch request is received via an *ioctl* call by the madwifi driver, it invokes the function *ath_chan_set()*. This function is responsible for switching the channel on the card. On analyzing this function, we found that all other calls made within this function are blocking calls (i.e., the calls do not sleep), and when the function returns, the channel switching has been fully completed. We wrapped this function call with two *do_gettimeofday()* calls, and the difference in the time returned by the two calls is the time elapsed in switching. Using this methodology, the interface switching delay for the atheros chipset-based cards that we use is approximately 5 milliseconds. We have validated the switching delay measurements with other indirect measurement experiments as well (the experiments look at the interarrival time between packets while switching an interface between packet transmissions).

8.2 Throughput measurements

We present results from a 4-node topology that has been setup to quantify the impact of switching (Figure 6). Every

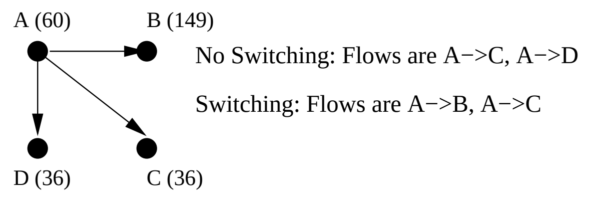
A (60)
B (149)
D (36)
C (36)
No Switching: Flows are A−>C, A−>D
Switching: Flows are A−>B, A−>C

Figure 6: Experimental setup.

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 6.

There are 12 non-overlapping channels in the IEEE 802.11a band, but past work has shown that when nodes are equipped with multiple interfaces, simultaneous transmission on one interface and reception on another interface, over adjacent channels, may interfere with each other [7]. Which pair of channels interfere depends on the frequency separation between channels, the distance separation between interfaces, and the distance separation between communicating nodes. We have conducted measurements which show 5 channels can be simultaneously used (channels 36, 52, 64, 149, 161) in our testbed. For results presented below, we have used only channels 36, 60, and 149.

We use two scenarios to quantify the cost of switching, as shown in Figure 6. 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 [9]. 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. 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 at 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 5.3 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 plots the aggregate throughput with varying T_{min} (T_{max} is set to 130 ms), for both TCP and UDP traffic. Figure 8 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

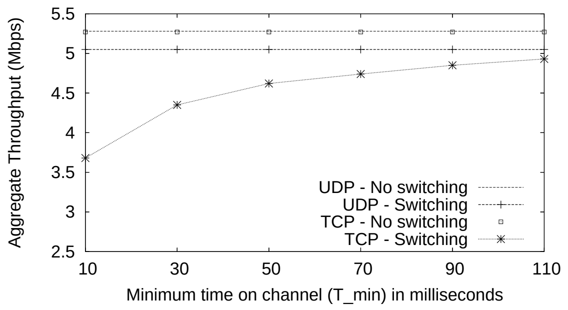

Figure 7: Throughput when minimum time spent on a channel is varied.

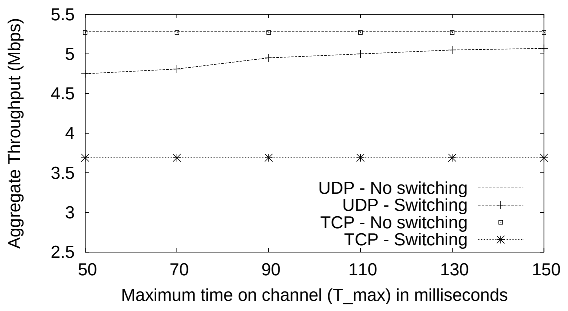

Figure 8: Throughput when maximum time spent on a channel is varied.

to switch, throughput does not change when either T_{min} or T_{max} is varied.

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 setup have sufficient load to saturate the channel. When the second interface switches to a channel to service one 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}. Therefore, 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. Theoretically, each switch wastes 5 ms of channel every T_{max} duration, and this is approximately the observed switching overhead. Therefore, our scheduling algorithm is efficient for saturated UDP traffic.

In the "Switching" scenario with TCP flows, throughput depends on the value of T_{min} but not on the value of T_{max}. In steady state, TCP sends a new packet only after receiving the ACK of an earlier packet. Therefore, packet transmissions are spaced out. On switching to a channel, there may only be a few packets buffered for transmission and most switches happen after only T_{min} duration. Therefore, with TCP traffic, the number of switches depends on T_{min}, 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. Detailed delay measurements and identifying good parameter values for delay-sensitive traffic is ongoing.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an architecture to support multi-channel protocols that require frequent interface switching. Our contributions in this paper include identifying the features needed in the kernel for supporting interface switching, designing and implementing a *channel abstraction* module that provides the requisite kernel support, and implementing a hybrid multi-channel protocol using the channel abstraction module. Preliminary results suggest that interface switching can be supported with moderate overheads, and this encourages further development of multi-channel protocols based on interface switching.

There are several directions for future work. We have already initiated a detailed evaluation of the architecture on a 20 node testbed. The architecture we presented here was partly biased by the requirement of the hybrid multi-channel protocol that we wanted to implement. To ensure our architecture is sufficiently general, we intend to implement other multi-channel protocols proposed in the literature as well. Further implementations may identify new features that may be necessary, and we plan to incorporate any other required features into the architecture as well. Although this paper has focused on supporting fast interface switching, the architecture may be useful to support per-packet power and rate control as well, and we intend to explore these possibilities as well.

10. ACKNOWLEDGMENTS

This research was supported in part by US Army Research Office grant W911NF-05-1-0246, NSF grant ANI-0125859, and Vodafone Graduate Fellowship.

11. REFERENCES

[1] A. Adya, P. Bahl, J. Padhye, A. Wolman, and L. Zhou. A Multi-Radio Unification Protocol for IEEE 802.11 Wireless Networks. In *IEEE Broadnets*, 2004.

[2] Atheros inc. http://www.atheros.com.

[3] P. Bahl, R. Chandra, and J. Dunagan. SSCH: Slotted Seeded Channel Hopping for Capacity Improvement in IEEE 802.11 Ad-Hoc Wireless Networks. In *ACM Mobicom*, 2004.

[4] J. Bicket, D. Aguayo, S. Biswas, and R. Moris. Architecture and Evaluation of an Unplanned 802.11b Mesh Network. In *ACM Mobicom*, 2005.

[5] 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*, 2005.

[6] R. Chandra, P. Bahl, and P. Bahl. MultiNet: Connecting to Multiple IEEE 802.11 Networks Using a Single Wireless Card. In *IEEE Infocom*, Hong Kong, March 2004.

[7] R. Draves, J. Padhye, and B. Zill. Routing in Multi-Radio, Multi-Hop Wireless Mesh Networks. In *ACM Mobicom*, 2004.

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

[9] Iperf version 2.0.2. http://dast.nlanr.net/Projects/Iperf/.

[10] N. Jain, S. Das, and A. Nasipuri. A Multichannel CSMA MAC Protocol with Receiver-Based Channel Selection for Multihop Wireless Networks. In *IC3N*, October 2001.

[11] V. Kawadia, Y. Zhang, and B. Gupta. System Services for Implementing Ad-Hoc Routing Protocols. In *ACM Mobisys*, 2003.

[12] M. Kodialam and T. Nandagopal. Characterizing the capacity region in multi-radio multi-channel wireless mesh networks. In *ACM Mobicom*, 2005.

[13] P. Kyasanur and N. H. Vaidya. Capacity of Multi-Channel Wireless Networks: Impact of Number of Channels and Interfaces. In *ACM Mobicom*, 2005.

[14] P. Kyasanur and N. H. Vaidya. Routing and Interface Assignment in Multi-Channel Multi-Interface Wireless Networks. In *IEEE WCNC*, 2005.

[15] P. Kyasanur and N. H. Vaidya. Routing and Link-layer Protocols for Multi-Channel Multi-Interface Ad hoc Wireless Networks. *Mobile Computing and Communications Review*, 10(1):31–43, Jan 2006.

[16] M. A. Marsan and F. Neri. A Simulation Study of Delay in Multichannel CSMA/CD Protocols. *IEEE Transactions on Communications*, 39(11):1590–1603, November 1991.

[17] A. K. Miu, H. Balakrishnan, and C. E. Koksal. Improving Loss Resilience with Multi-Radio Diversity in Wireless Networks. In *ACM Mobicom*, 2005.

[18] A. Nasipuri, J. Zhuang, and S. Das. A Multichannel CSMA MAC Protocol for Multihop Wireless Networks. In *IEEE WCNC*, Sept 1999.

[19] A. Raniwala and T. Chiueh. Architecture and Algorithms for an IEEE 802.11-Based Multi-Channel Wireless Mesh Network. In *IEEE Infocom*, 2005.

[20] A. Raniwala, K. Gopalan, and T. Chiueh. Centralized Channel Assignment and Routing Algorithms for Multi-Channel Wireless Mesh Networks. *Mobile Computing and Communications Review*, 8(2):50–65, April 2004.

[21] J. So and N. H. Vaidya. Multi-channel MAC for Ad Hoc Networks: Handling Multi-Channel Hidden Terminals using a Single Transceiver. In *ACM Mobihoc*, 2004.

[22] Net 4521 hardware from soekris. http://www.soekris.com/net4521.htm.

[23] P. Stuedi and G. Alonso. Transparent Heterogeneous Mobile Ad Hoc Networks. In *ACM MobiQuitous*, 2005.

[24] Virtual wifi software page. http://research.microsoft.com/netres/projects/virtualwifi.

[25] 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 *I-SPAN*, 2000.