Net-X: System eXtensions for Supporting Multiple Channels, Multiple Interfaces, and Other Interface Capabilities*

Technical Report

August 2006

Pradeep Kyasanur Chandrakanth Chereddi Nitin H. Vaidya
University of Illinois at Urbana-Champaign
kyasanur@crhc.uiuc.edu, chereddi@gmail.com, nhv@uiuc.edu

***Abstract*— There are several *interface capabilities* that may be available in next generation wireless networks. Some examples of interface capabilities include the ability to set the channel of operation and data rate on a frequent basis, and the ability to equip devices with multiple radio interfaces and multiple antennas. It may be possible to significantly improve the performance of wireless networks by exploiting the interface capabilities through carefully designed protocols. However, current operating systems have poor support for implementing protocols that need to use the available interface capabilities. The goal of the *Net-X* project is to develop operating system support for utilizing interface capabilities. As a first step in this direction, we have developed an architecture in Linux to support the use of multiple channels, multiple interfaces, and interface switching. This support has been used to implement a set of multichannel protocols that we had previously developed. In this report, we will describe the new architectural support, implementation of the multichannel protocols, and the use of the protocols in a mesh network. We also describe how the architecture may be extended to support the use of other interface capabilities as well.**

I. Introduction

Newer generation wireless radio hardware provide support for setting several radio parameters, such as the channel of operation, the data rate, and the transmission power, on a frequent[1] basis. In addition, with reducing hardware costs, it is possible to increase the *resources* available at each host by providing for multiple interfaces, multiple antennas, etc. Collectively, we may view the different radio parameters and hardware resources as *interface capabilities* that are available in the network.

Common operating systems have strived to *hide* the interface capabilities from higher layers of the protocol stack (e.g., network layer is usually unaware of the notion of channels and data rates). While the ability to hide interface capabilities simplifies the design of higher layer protocols, it may also severely limit the ability to exploit the available capabilities. For example, because the network layer is only aware of the notion of interfaces, kernel routing tables typically only allow for specifying the interface to use over a route. As a result, there is no support for explicitly associating routes with other kinds of interface capabilities, such as channels and data rates. However, our past work [1] on utilizing multiple channels has demonstrated the need for channel-aware link and routing protocols for effective utilization of channels, and we believe that such "aware" protocols are needed to effectively exploit other interface capabilities as well. Therefore, there is a need to develop *extensions* to the existing operating systems to allow higher layer protocols to utilize interface capabilities.

The goal of the *Net-X* project is to develop generic support for utilizing interface capabilities, such that the support is cleanly integrated into the network stack. As a first step in this direction, we

*This work was funded in part by National Science Foundation.

[1]The radio parameters could be potentially set several times every second, and for some parameters could be set on a per-packet basis.

have focused on providing support for utilizing multiple channels and multiple interfaces. We have developed generic architectural support in Linux that provides higher layers fine-grained control over the channels and interfaces used for sending out data. Although our initial implementation has focused on the use of multiple channels, the architecture itself can support other interface capabilities, such as multiple data rates and multiple transmission powers.

In this report, we will describe our efforts on building a multichannel multi-interface testbed. Our work was motivated by the lack of kernel support for implementation of a set of multichannel protocols [2] that we had developed. The testbed implementation includes new architectural extensions for supporting the use of multiple channels and multiple interfaces, and one set of multichannel protocols [2] to demonstrate the use of the extensions. We will also describe how our implementation can be used to support other interface capabilities as well.

The rest of this report is organized as follows. We present related testbed work to support multichannel protocols in Section II. Section III provides an overview of the multichannel protocols that we have developed. Section IV identifies the need for new operating system support to exploit interface capabilities. Section V presents the implementation architecture, and the details of the implementation are in Section VI, Section VII, and Section VIII. We describe extensions to the implementation for building a mesh network in Section IX. Sample results are presented in Section X, and we conclude in Section XI.

II. Related work

There have been several research initiatives on building wireless network testbeds [3]–[9]. However, there have been relatively fewer attempts at building multichannel wireless testbeds [10]–[13]. Of these, [10], [12], [13] assume that interfaces are fixed to a channel for long intervals of time. Therefore, in those implementations, switching interfaces from one channel to another can be done infrequently, possibly using user-space scripts, without requiring support from the kernel. In contrast, our solutions require more fine-grained interface switching, which requires additional architectural support.

"VirtualWifi" [11], [14] is a virtualization architecture that abstracts a single wireless interface into multiple virtual interfaces. VirtualWifi provides support for switching the physical interface across the channels used by each virtual interface. VirtualWifi has some similarity to our implementation, 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 (by exposing one IP address per channel) 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 our implementation is that it exports 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 [15]–[17]. However, those approaches are not designed to support the notion of using multiple channels or interface switching between channels.

Architectural changes have been proposed by wireless researchers to support other protocols in ad hoc networks. One well studied architectural problem is to support on-demand routing [3], [4], [17], which requires mechanisms to buffer data packets while a route is being discovered. We implement the on-demand discovery component of the proposed multichannel routing protocol using the same approach as in [4]. However, implementing the other aspects of the multichannel protocols, such as interface management, requires additional support, as they require close interaction with the device drivers.

We are not aware of any testbed works that allow higher layer protocols to control the use of data rates, transmission powers, antennas on a frequent basis.

III. Overview of supported multichannel protocols

The Net-X testbed is used to implement a set of multichannel protocols that we have previously de-

veloped [1]. Here, we will provide a brief overview of the protocols.

The focus of our multichannel protocols is the effective use of multiple channels when there are fewer interfaces per node than channels. We develop two protocols: an interface management protocol to assign channels to interfaces, and a routing protocol to select good routes in a multichannel network. The testbed implementation of the protocols requires each node to have two interfaces, though we will later describe extensions to support the use of a single interface at each node.

One interface at each node is called a "fixed interface" and is assigned for long intervals[2] of time to a "fixed channel". The second interface at each node is called a "switchable interface" and can be switched between any of the remaining channels, as necessary. The fixed channel of a node is selected using a fixed channel selection protocol, which tries to equally distribute the fixed interfaces of different nodes on different channels. All data sent to a node must be over the fixed channel, because the node is guaranteed to always listen to the fixed channel. A node may send data to a neighbor using the fixed interface if both nodes use a common fixed channel; otherwise, the switchable interface is tuned to the neighbor's fixed channel, and data is sent out over the switchable interface. Each node periodically sends out broadcast "hello" messages. Hello messages include the fixed channel of the node, and the fixed channels of all 1-hop neighbors. Using this mechanism, a node can eventually learn about the fixed channels used by all nodes in its 2-hop neighborhood, and this information is used in balancing channel assignment.

The routing protocol supports a new multichannel routing metric (MCR). The metric is incorporated into an on-demand routing protocol. More details about the routing metric, as well as other aspects of the interface management and routing protocols are in [1].

[2]The intervals are long when compared to packet transmission time and interface switching delay.

IV. Architectural support for higher layer control over multiple channels

The multichannel protocols described in the previous section require (switchable) interfaces to potentially switch on a frequent basis. Although existing interface hardware allows for switching on a frequent basis, kernel support for higher layer control over frequent switching is absent. In this section, we motivate the need for new architectural support in kernel for supporting multichannel protocols that require interface switching.

Existing off-the-shelf hardware do allow interfaces to be switched by the driver, but common operating system kernels are not designed to utilize this feature. Operating systems have always tried to abstract out the details of the underlying hardware from higher layer applications. We want to continue to preserve this design principle of abstracting out non-essential features, and provide a clean mechanism for higher layer protocols to control the use of channels and interfaces. For example, user applications need not be aware of the notion of multiple channels, multiple interfaces, and interface switching. Routing and link-layer protocols may be aware of the notion of channels and interfaces, but need not be aware of the detailed procedures used to implement interface switching. These requirements necessitate changes to the operating system kernel. As we argue next, supporting multiple interfaces and interface switching in multichannel networks requires non-trivial changes to the kernel. Our approach is to develop a generic *channel abstraction layer* to support interface switching. The channel abstraction layer could be used to implement other multichannel protocols that require interface switching, in addition to the protocols that we have implemented. The abstraction layer can also be extended to support other interface capabilities, such as data rate and transmission power.

A. Need for new support

We identify the features needed to implement interface switched multichannel protocols (that are missing in current operating systems) by using Linux as an example. The key features that are lacking in Linux are as follows:

1. Specifying the channel to use for reaching a neighbor:

Common operating systems do not allow the network layer to control the channels (or other radio features, such as data rates, transmission powers, and antennas) used to reach a neighboring node. Instead, the kernel routing tables only provide control over the interface to use to reach a neighboring node. In a single channel network, there is no benefit in explicitly selecting channels because all nodes have to use a common channel. This lack of explicit channel support is not a problem in those multichannel networks where each interface is associated with exactly one channel, i.e., there is an one-to-one mapping between interfaces and channels. For example, there is an one-to-one mapping between interfaces and channels in a network where each node has m interfaces, and the interfaces of a node are always fixed on some m channels. In this setting, the channel to use to reach a neighboring node can be indirectly specified by the specifying interface to use, since each interface is associated with a unique channel.

However, we are interested in the scenario where the number of interfaces per node could be significantly smaller than the number of channels. Therefore, there is no longer an one-to-one mapping between channels and interfaces[3]. Under this scenario, we have shown that a good strategy is to use the same (switchable) interface to send data to different neighboring nodes over possibly different channels. To support the notion of switchable interfaces, we need to to control the channels to use to reach a node.

For example, consider the scenario shown in Figure 1. In the figure, suppose that each node has a single interface. Also suppose that node B is listening to channel 1 and node C is listening to 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, when it is so required). This example shows that the channel to use for transmitting a packet may vary based on destinations, even if the same interface is used for all destinations. In general, without support for specifying the channels to use to reach a neighboring node, it is difficult to implement those multichannel protocols that use a *single* interface to send data to different neighboring nodes over *different* channels.

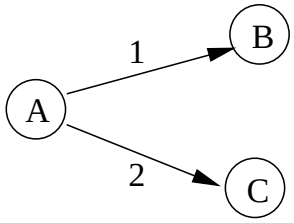

Fig. 1. Example illustrating the lack of kernel support for multichannel protocols.

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 multichannel network, different nodes may be listening to different channels. Therefore, to allow broadcast packets in a multichannel network to reach 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[4]. 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 that its neighbors B and C can 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 which channels broadcast packets have to be sent out on, or to actually

[3]Similarly, a single interface could be used with different data rates, or with different transmission powers, to reach different nodes. Therefore, there may not be a one-to-one mapping between an interface, and the data rates and transmission powers that are used.

[4]A similar feature may be needed if an interface is equipped with multi-beam antennas. A beam may only reach a subset of neighbors, and copies of packets may have to be sent over each beam.

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. Buffering and scheduling support

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 packets that are awaiting transmission in the interface queue. 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 with the interfaces used in our testbed), and switching too frequently 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.

B. Design choices

The earlier discussions 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. 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. Once we have decided to add support for multiple channels and interface switching, the next question is to identify the layer where the support can be added. Handling multiple channels and interface switching requires close interaction with the interface device driver. Based on this requirement, we have three possible locations for adding support:

1) Add the required 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 the required support into the network layer (for example, as a "Netfilter" hook [18]). 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 the required support as a new module that operates between the network layer (as well as 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 presenting 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 the need to know the details of managing multiple interfaces. We choose this approach, and implement a new *channel abstraction layer* module.

The option we have chosen 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 the device drivers. The bonding driver is typically used for grouping multiple Ethernet-based devices into a single virtual device. The bonding driver 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 have implemented the channel abstraction layer as a new feature of the bonding driver. In the next section, we describe the implementation of the multichannel protocols proposed in [1] using the interface switching support provided by the channel abstraction layer.

V. Implementation

In this section, we will first describe the implementation architecture, and then describe the implementation of each of the key components.

A. Implementation architecture

The multichannel implementation architecture is shown in Figure 2. Our implementation has three main components, which collectively implement one set of multichannel protocols.

- Channel abstraction layer: This kernel component manages multiple channels and interfaces, and provides support for fast interface switching. This component is generic enough to support other multichannel protocols, and other interface capabilities, such as data rates and transmission powers. The channel abstraction layer abstracts the details of multiple channels and interfaces from the higher layers, and is controlled by "IOCTL" commands from the userspace daemon.
- Kernel multichannel routing support: This component is used to provide kernel support for on-demand routing. The component informs the userspace daemon when a route discovery has to be initiated, and buffers data packets while the route discovery is pending.
- Userspace daemon: The userspace daemon implements the less time-critical components of higher layer multichannel protocols (our multichannel protocols include two an interface management protocol and a routing protocol). Most of the higher layer protocol functionality is implemented in this component.

The kernel components interact with the Linux TCP/IP implementation and the interface device drivers, while the userspace daemon is built using standard userspace networking libraries. Most of the multichannel protocol has been built into the userspace daemon. The kernel components support only a small set of essential features. In later sections, we describe the implementation of each component, as well as the interaction between components.

VI. Channel abstraction module

In this section, we will describe the channel abstraction module (CAL). The module is implemented as a new feature of the bonding driver present in the Linux kernel. Figure 2 shows the key components of CAL:

- 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.

In addition, we modify the madwifi drivers to better support channel switching. The details of the components and driver modifications are presented below.

A. 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 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 an user space multichannel protocol via IOCTL calls (entries can be added or deleted).

When the CAL 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.

The unicast component can be easily extended to support other interface capabilities. For example, the tuple associated with each destination could

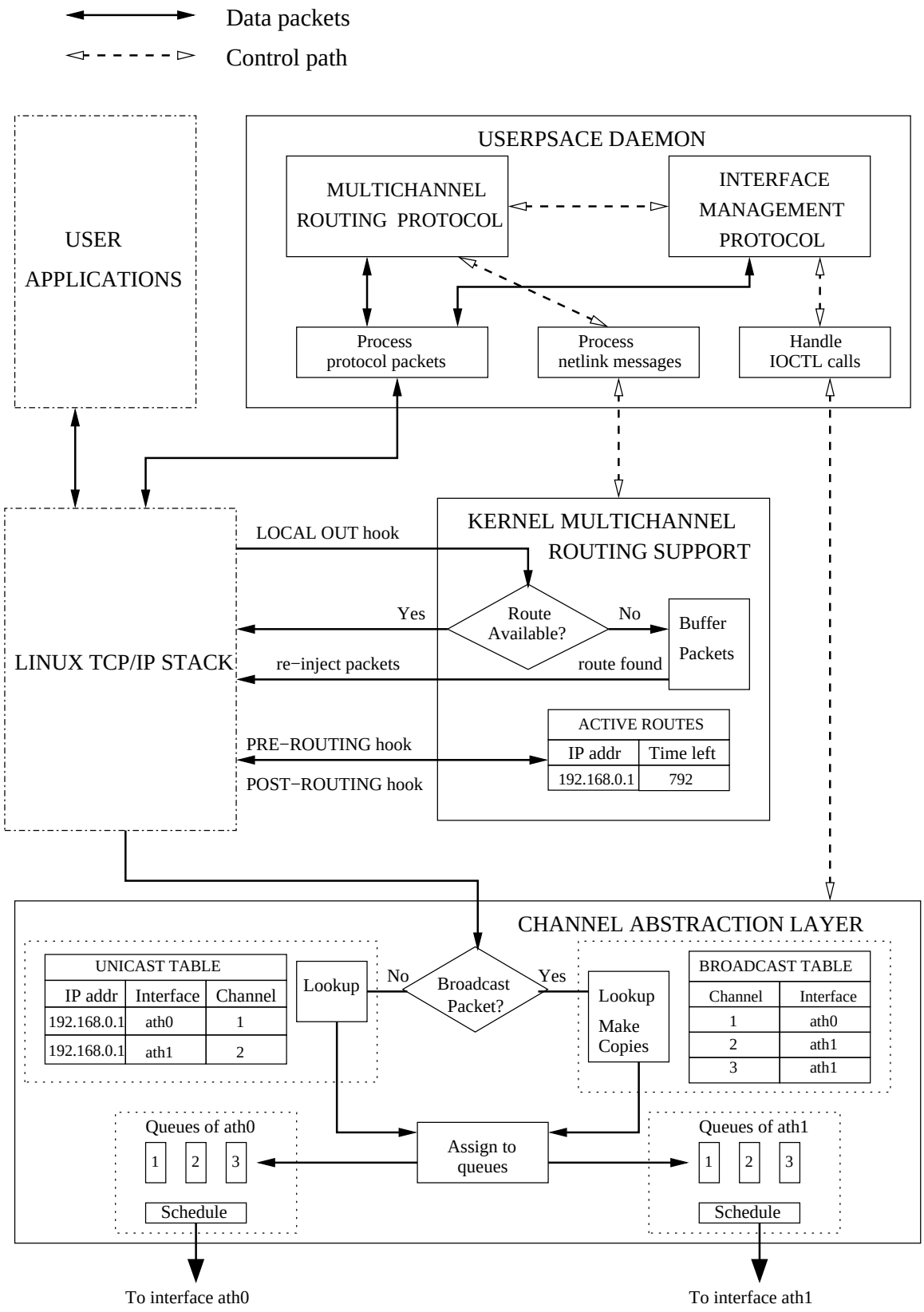

Fig. 2. Architecture for implementing multichannel protocols. The figure assumes that two interfaces, “ath0” and “ath1”, are available.

include additional entries to specify the data rate and the transmission power. The IOCTL calls can be extended (or new IOCTL calls added) to include data rate and transmission powers. When the packet is handed down to the device driver, the rates and transmission powers to use can be given to the driver.

B. 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 2. 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 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, protocols that send a copy of broadcast packet over all the available channels, can all use this broadcast architecture.

When the CAL 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.

A similar mechanism can be used to support other interface capabilities that may require multiple transmissions to support broadcast. For example, the broadcast list can be extended to specify the antenna beam on which a packet has to be sent out, to support broadcasts with multi-beam antennas.

C. Scheduling and queuing component

The scheduling and queuing component is the most complex part of CAL. For each available interface, the component maintains a separate set of channel queues as shown in Figure 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 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 queuing procedure is useful to support interface capabilities that incur a non-negligible delay for switching from one mode[5] to the other. For example, multi-beam antennas may incur a non-negligible beam switching delay. Similarly, certain hardware may not allow changing the data rate or transmission power on a per-packet basis. In such scenarios, a queue can be associated with each mode of an interface capability, thereby reducing the frequency of switching the mode of operation (and hence reduce the switching cost).

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 3 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

[5]We use the term "mode" to refer to the different values of an interface capability. For example, the different data rates correspond to the different modes available with the data rate capability.

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 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 VI-D). 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 that has buffered packets, using a round-robin service policy.

Although our current implementation has used a round-robin scheduling policy, it is fairly simple to provide alternate scheduling policies. For example, the scheduler can be easily modified to provide higher priority to certain nodes, or certain channels. Similarly, different scheduling policies may be appropriate for different interface capabilities.

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 multichannel protocols to do intelligent channel assignment, route selection, etc.

D. Driver modifications

CAL 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 [19] 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.

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 multichannel 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 [20]. Using this technique, we have reduced the interface switching delay to about 5 ms.

Some capability-specific modifications may be required for other interface capabilities as well. However, with the generic CAL architecture, we expect such modifications to be fairly small and restricted to minor aspects of the driver functionality.

2) Query support: As we discussed in Section VI-C, 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

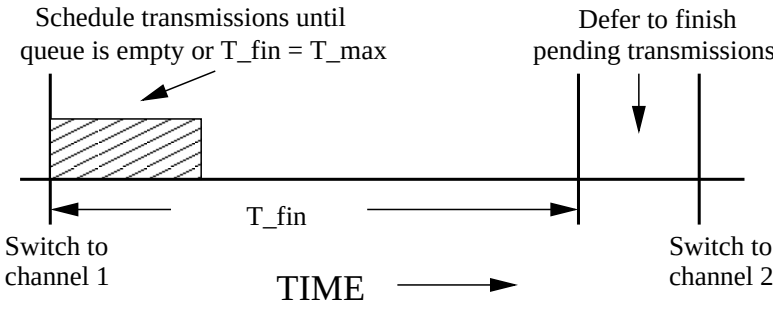

Fig. 3. Example time line of scheduling.

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.

E. Functionality exported by CAL

The userspace daemon communicates with CAL using a set of IOCTL calls. IOCTL calls are a common way in Linux for interaction between kernel and userspace components. The list of IOCTL calls provided by CAL and their functionality is listed in Table I. We will describe later the sequence in which these IOCTL calls are invoked by the userspace daemon. More details about the CAL implementation are in Chereddi's thesis [21].

The same IOCTL calls can potentially be extended to take additional parameters to support other interface capabilities. Alternatively, it is also possible to provide new IOCTLs to support other interface capabilities.

VII. Kernel multichannel routing support

The kernel multichannel routing (KMCR) module provides support for on-demand routing. For example, when an application initiates communication to a destination that is not a direct neighbor, a new route may have to be setup if no route to the destination is already available. The route discovery protocol is implemented as part of the userspace daemon. However, a mechanism is needed to invoke the discovery process when a new route is desired by the application. Clearly, the only place where access to all application packets is available is in the kernel. Therefore, several earlier routing implementations [3], [4], [17] have included support in the kernel to initiate route discovery.

The on-demand route discovery process should be transparent to applications. While the on-demand route discovery is in progress, any packets sent by the application have to be buffered, and later sent out once a new route is available. This buffering is required to prevent packet drops. Note that higher-layer protocols such as TCP are severely affected by the loss of the initial packets in a connection. For example, TCP incurs a large timeout if the initial SYN packet, used for connection establishment, is lost. Packet buffering has been implemented in past works in two different ways. In one approach [17], application packets are sent up to the userspace and stored by a userspace daemon, and re-injected once the route is discovered. In the second approach [4], packets are buffered in the kernel itself. We follow the second approach because it avoids the context switching overheads of sending a packet up to the userspace.

The kernel routing support is implemented as a module which can be loaded into the Linux kernel. The module utilizes the Linux Netfilter support [18], and the implementation was based on the AODV implementation from Uppsala university [4], [22]. Figure 2 shows the structure of KMCR module. The KMCR module maintains a "active route table" containing a list of nodes to which routes are available. Each node in the list is also associated with a "time left" field that represents the time

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

TABLE I

LIST OF IOCTL CALLS EXPOSED BY CHANNEL ABSTRACTION LAYER.

period after which the route to that node is deemed to be inactive.

The Netfilter library offers "hooks" to intercept packets traversing through the networking stack. The "LOCAL OUT" hook intercepts packets that originate in the node, before the routes to be used by the packets are computed. The KMCR module adds itself to the LOCAL OUT hook. If a packet received on the LOCAL OUT hook is destined for a node (using the wireless interface) that does not currently have a route, then a new route has to be discovered. Otherwise, if a route exists, the packet is returned without any modifications. When a route is not available, KMCR requests the userspace daemon to find a new route. After that, any packets to that destination are captured from the LOCAL OUT hook and buffered in the KMCR module until a new route is discovered. After a route is discovered, the KMCR module is notified by the userspace daemon, which then re-injects the buffered packets. If the route discovery fails, then any packets that had been buffered, pending route discovery, are dropped.

The KMCR module also adds itself to two other netfilter hooks; the "PRE-ROUTING" hook and the "POST-ROUTING" hook. The PRE-ROUTING hook intercepts packets received by a node from an external node, while the POST-ROUTING hook intercepts packets that are destined for an external node. Packets received on the PRE-ROUTING hook over the switchable interface are dropped (as the switchable interface is not intended for receiving data). Suppose X is a node corresponding to either the source of a packet intercepted on the PRE-ROUTING hook (that is received over the fixed interface), or the destination of a packet intercepted on the POST-ROUTING hook. Then, the time left for the route to X, contained in the active route table, is reset to a maximum *LifeTime* value (we set the *LifeTime* value to 30 seconds in our implementation). Essentially, when packets are intercepted on the PRE-ROUTING and POST-ROUTING hooks, it indicates that the routes used by the packets are active. If no packet is intercepted along a route over a time longer than the *LifeTime* value, then the route is assumed to be not in use. The userspace daemon periodically checks which routes in the active route table are no longer in use, and removes them.

The communication between the KMCR module and the userspace daemon is implemented using "netlink" messages. Netlink library is a feature in Linux to support communication between kernel and userspace. Netlink offers more flexibility than IOCTL calls by allowing two way communication, and is especially useful when more than a few bytes of information have to be exchanged. The list of netlink messages implemented by the KMCR module, and their functionality is listed in Table II.

VIII. USERSPACE DAEMON

The userspace daemon implements the interface management and routing protocols by utilizing the features offered by the CAL module and the KMCR module. Recall that an overview of the protocols was provided in Section III. In this section, we focus on issues specific to the implementation of the protocols. The implementation currently supports at most two interfaces per node, but can be easily extended to support more than two interfaces. The userspace daemon should be started only after the CAL and KMCR modules have already been loaded into the kernel. Our current userspace implementation assumes that two interfaces are available at each node (and can be easily extended to handle more than two interfaces). In the next section, we describe extensions to handle nodes with a single interface.

Netlink message	Function
AddRoute	Add an entry into active table. Implies route discovery was successful.
DiscoveryFailed	Inform KMCR that route discovery failed.
DeleteRoute	Remove an entry from active route table.
InitiateDiscovery	Request sent by KMCR to userspace for initiating route discovery.
IsRouteActive	Query KMCR if the requested route is active.
RouteStatus	Response from KMCR to userspace on the status of the requested route.

TABLE II

LIST OF NETLINK MESSAGES SUPPORTED BY THE KERNEL MULTICHANNEL SUPPORT MODULE.

Initialization: The userspace daemon is provided with a list of valid channels and a list of available interfaces in a configuration file. Using this information, the daemon initializes the kernel modules as shown in Figure 4. The initialization protocol first informs CAL of the set of valid channels associated with each interface. Next, using the fixed channel selection protocol (Section III) one of the available channels is randomly chosen as the fixed channel, and one of the interfaces is switched to the fixed channel, while the second interface is switched to any of the remaining channels. After that, the broadcast list is initialized such that the fixed interface is used to transmit on the fixed channel, and the switchable interface is used to transmit on the remaining channels. Once these initialization steps are complete, the hello packets announcing the fixed channel can be sent out. After that, whenever the fixed channel of the node is changed, a request is sent to the CAL to switch the fixed interface.

Managing received hello packets: As described in Section III, hello packets received from a neighbor enables a node to discover the channels used by its neighbor. When the fixed channel being used by a neighbor is first discovered, a new entry is added into the unicast table of CAL by sending a UnicastEntry message. Later, if the channel used by the neighbor changes, CAL is updated by another UnicastEntry message. Similarly, if no hello messages have been received from a neighbor for more than a timeout duration, then the entry corresponding to the neighbor is removed from the CAL using a UnicastEntry message.

Route discovery and maintenance: The route discovery process is initiated by the KMCR module, as we described earlier. Figure 5 shows the interaction between the KMCR module, the userspace daemon, and the CAL module to set up a new route. When the KMCR module at some node S discovers the need for a new route to some node D, it sends an InitiateDiscovery request to the userspace module. The userspace module then queries the CAL module to obtain the switching cost of using different channels (recall that switching cost of channels is used by the routing metric). After that, a route request (RREQ) packet is sent out. Intermediate nodes on receiving the RREQ again query their CAL to obtain the switching cost, which is included while forwarding the RREQ. The destination on receiving the RREQ responds with a route reply (RREP) packet. The source node S on receiving the RREP adds a new entry in the unicast table of the CAL for node D, and the channel to use for node D is set to the channel used to reach the first hop node on the route to node D. After that, the KMCR module is informed of the successful route discovery through an AddRoute message. If a route discovery fails, then the KMCR module is informed of the failure using the DiscoveryFailed message.

Intermediate nodes that forward the RREP containing the route from S to D have to also add information about the route into their kernel tables. Typically, after a route is set up from S to D, the route is used for bi-directional communication. Therefore, intermediate nodes have to add a route to both the source S and the destination D to ensure that data packets between S and D, sent in either direction, are forwarded. The process of adding a route is similar to the procedure followed by node S, which was described above (though the process has to be invoked twice to add routes to both S and D).

Route maintenance involves removing routes from the KMCR table that have not been active for a

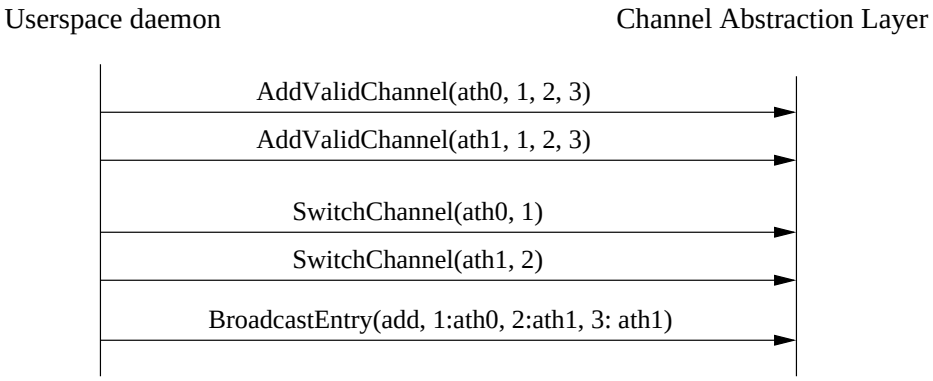

Fig. 4. Commands invoked at initialization. The figure assumes that two interfaces (ath0, ath1) and three channels (1, 2, 3) are available. The values in parenthesis are parameters included in the IOCTL calls. For example, the values in the BroadcastEntry call list the interface to use for each broadcast channel.

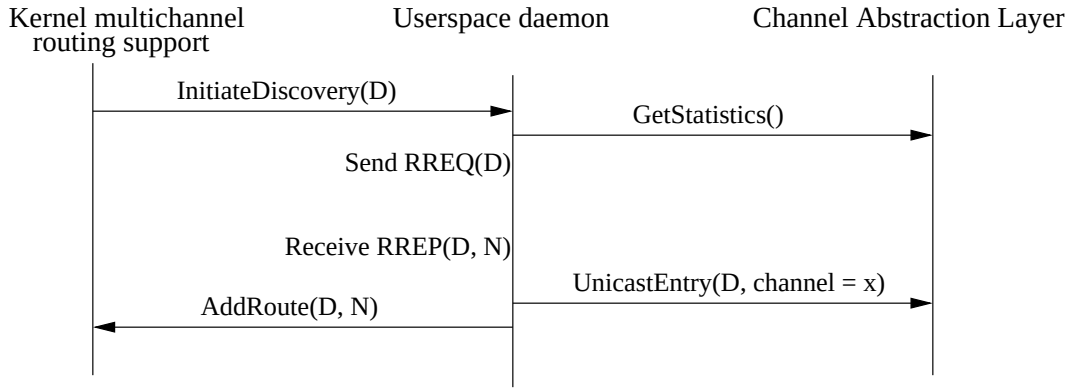

Fig. 5. Commands invoked during route discovery. The example assumes that a route is being discovered to node D, and the next hop on the discovered route is node N.

long time or have been identified as broken. Inactive routes are discovered by the userspace daemon by periodically querying the KMCR module with the IsRouteActive message (the response is received in the RouteStatus message). Broken routes are identified when the next hop is not reachable, or when a route error message is received. In all cases, the route is removed from the active route table of KMCR by sending the DeleteRoute message, and from the unicast table of CAL by sending the UnicastEntry message.

The multichannel protocol implementation described above provides most of the support required for running a multihop multichannel network, provided each node is equipped with two interfaces. The protocols allow any node in the network to communicate with any other node. In our work, we have assumed that there is a separate address assignment protocol to assign addresses to individual nodes. Any of the address assignment protocols proposed in the literature [23], or even manual assignment (if appropriate), could be used with our implementation. Our testbed experiments have used manual address assignment, and we defer to future work automated address assignment.

The current userspace daemon implementation may be used as a template to implement other multichannel protocols. In addition, we believe that the implementation may serve as a template for building higher layer protocols to exploit other interface capabilities as well. For example, it is possible to develop higher layer protocols for utilizing the multi-rate capabilities. Such a protocol may require neighbor discovery and routing mechanisms, and our implementation may be used to guide the multi-rate protocol implementation.

IX. Mesh networking extensions

Multihop wireless mesh is one network architecture for providing last mile wireless connectivity. This architecture has been used in building community wireless networks [24], [25] city-wide wireless networks [26], among others. In this section, we describe extensions to our implementation to support mesh networking. The mesh networking implementation demonstrates the feasibility of using multiple channels in a mesh network.

Figure 6 describes the mesh networking architecture that we support. Nodes in the network are classified into two types; "mesh nodes" that run the software we have implemented, and "client nodes" that are unmodified. Client nodes are equipped with one IEEE 802.11 interface, while mesh nodes have either one or two interfaces. Some of the mesh nodes are connected to the Internet and act as "gateways". Mesh nodes form a backbone network that is fully connected, and a mesh node is capable of communicating with any other mesh node. A client node connects to one of the mesh nodes that is in its direct communication range. The mesh nodes are viewed by the clients as access points. All mesh nodes and client nodes are capable of communicating with hosts in the Internet, but the client nodes do not communicate with any mesh node other than the node they are connected to directly.

Mesh nodes equipped with two interfaces support the multichannel protocols presented in Section III, and can potentially transmit on any channel. Single interface mesh nodes support a modified interface management protocol which requires them to be fixed on a specific channel, but support the default multichannel routing protocol used by two interface nodes. In the rest of this section, we describe extensions to the implementation to provide support for gateways, support for mesh nodes that have a single interface, and support for allowing unmodified client nodes to connect to the Internet through a mesh node.

A. Gateway support

A gateway node uses a wired interface to connect to the Internet, while using two wireless interfaces to participate in the mesh backbone. In our testbed, the mesh network uses a private address space that is not visible to the Internet. Therefore, to allow nodes in the mesh network to communicate with nodes in the Internet, network address translation (NAT) is required at the gateway node[6]. Standard Linux distributions provide NAT support, and the translation can be set up using the "iptables" tool.

When an application on a mesh node initiates communication with a node in the Internet, the packets have to be first routed from that node to the gateway node, and from there on to the Internet. IP packets have a destination field in the IP header containing the address of the destination, and this is used by the routing tables in the mesh nodes for deciding the channel and next hop node to use while forwarding a packet. When a packet is destined to the Internet, the gateway is an intermediate destination, with the actual destination located in the Internet. If the destination address in a IP packet is set to that of the final destination, then there is no information in the packet to guide it toward the gateway node. On the other hand, if the destination address is set to that of the gateway node, then the final destination address is not contained in the packet.

To ensure that a packet contains the address of both the gateway node and the final destination (without requiring modifications to IP), we use a *packet encapsulation* approach (a similar approach was used by [22]). At the source mesh node, in the first step, an IP packet is created with the destination address in the IP header set to the address of the node in the Internet. After that, the packet is encapsulated with a second IP header[7], and the destination address in the second header is set to that of the gateway. When the packet reaches the gateway, the gateway node decapsulates any packets that are encapsulated, and then forwards the packet on to the Internet. The process of encapsulation/decapsulation and network address translation is shown for an example scenario in Figure 7.

[6]The address translation ensures that packets originating in any mesh node appear to the nodes in the Internet to be originating from the gateway node, on the wired interface

[7]This mechanism is provided for in the IP protocol, and is called the IP in IP mode. The mode is often used to tunnel packets over multiple IP hops, for supporting mechanisms such as mobile IP.

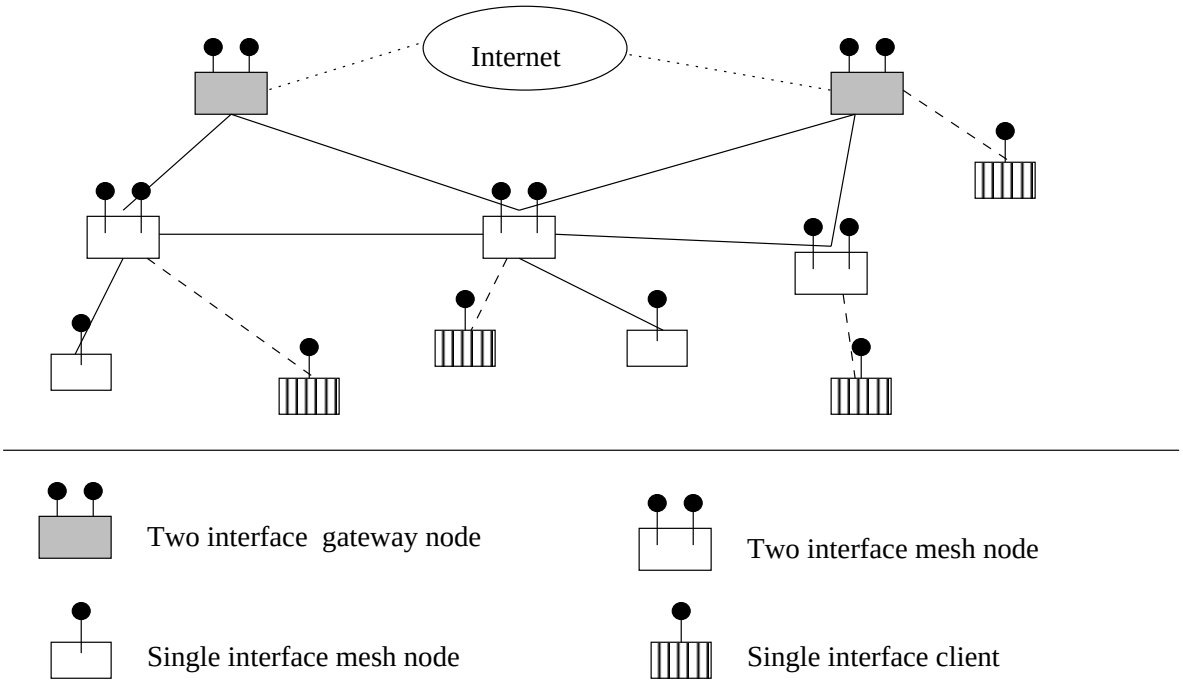

Fig. 6. The mesh networking architecture supported by our implementation.

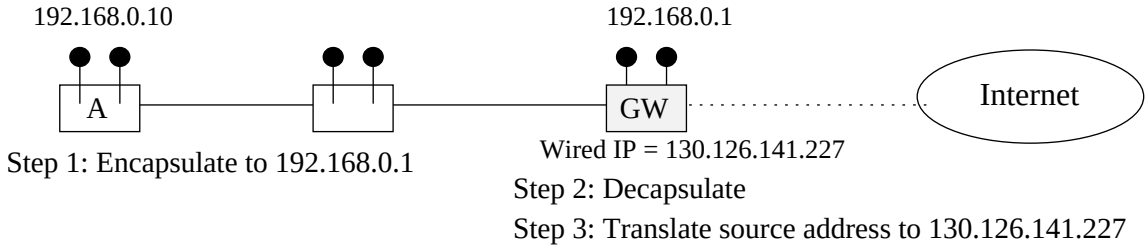

Fig. 7. Steps involved in sending a packet to the Internet. Source node A encapsulates the packet, which is forwarded to the gateway node GW. Gateway node decapsulates the packet, translates the address and forwards the packet to Internet.

The support for encapsulation and decapsulation is implemented in the KMCR module. The encapsulation is done in the LOCAL OUT netfilter hook at each mesh node (only for packets destined to the Internet), while the decapsulation is done in the PRE-ROUTING netfilter hook (only at gateway nodes). Note that packets that are coming in from the Internet to a mesh node do not need encapsulation, because the network address translation implicitly ensures that packets to mesh nodes are first routed to the gateway node.

Discovering gateway nodes: A gateway node sends out a broadcast advertisement informing other nodes that it is a gateway. The advertisement message has a “hop” field containing the number of hops to the gateway node. Initially, the field is set to zero, and every node that forwards the message increases the hop field by one. The message is forwarded only up to a pre-specified maximum number of hops which can be suitably chosen. A node receiving the advertisement message will learn the address of the gateway node and the number of hops to the gateway. Similar gateway discovery mechanisms have been proposed in the past as well [23].

Our implementation supports multiple gateways in the mesh network. Having multiple gateways provides resilience to failures, and allows traffic to the Internet to be distributed across nodes. When a node receives advertisement messages from multiple gateways, it selects one of the gateways as the

gateway it intends to use. In our implementation, the gateway that is fewest hops away is selected, with ties broken arbitrarily. The gateway selection mechanism can be easily extended to use other cost metrics, instead of basing the selection on the number of hops to the gateway.

B. Single interface support

Mesh nodes that are equipped with a single interface cannot support the hybrid interface assignment strategy which requires at least one fixed interface and one switchable interface. It is possible that a mesh network may be incrementally deployed, with some of the nodes having only a single interface. We aim to allow single interface nodes to participate in the mesh network, though at the cost of not being able to communicate directly with all neighbors.

A node S with a single interface keeps its interface fixed on one of the available channels (the algorithm for selecting the fixed channel is described below). The fixed channel is chosen such that the node S has at least one multi-interface node, say M, that can be directly reached on the fixed channel. Single interface nodes implement the full multichannel routing protocol presented earlier. Therefore, if the single interface node has to communicate with any other neighboring node that is not sharing a fixed channel with itself, it can do so by routing via node M. This ensures that node S can communicate with any other node in the network via node M.

During initialization, a single interface node S first randomly selects a channel and switches to that channel. After that, node S sends out a broadcast message on that channel advertising its presence. Any multi-interface mesh node M that receives this advertised message[8] responds back with a unicast acknowledgment announcing its presence. If node S receives an acknowledgment, node S learns the availability of a multi-interface node M on that channel. Node S then fixes its single interface to the channel it is already on, completing the fixed channel selection. On the other hand, if node S does not receive any acknowledgments, it learns that there are no multi-interface nodes on its current channel within its communication range. Node S then switches its interface to the next available channel and repeats the discovery procedure, and the process continues through the available channels till a channel with a multi-interface neighbor is found. After initialization, the fixed channel selection algorithm is repeated whenever no multi-interface nodes are reachable on the fixed channel. The discovery procedure could be extended to select fixed channels based on other metrics, such as the number of multi-interface nodes on a channel, or the load on a channel.

The fixed channel selection procedure described above allows a single interface node to be connected to the mesh network if it has at least one multi-interface node in its neighborhood. This requirement is conservative as it does not allow for a single interface node S to connect to the rest of mesh network through another single interface node, say X. Note that if X is connected to mesh backbone, then S could connect to the mesh backbone through X. We do not allow a single interface node to be connected through another single interface node because of the possibility of network partitions. For example, node X may believe it is connected to the mesh backbone through node S, while node S believes it is connected through node X. In that scenario, nodes S and X could be connected to each other, but not to other nodes in the mesh network. Then, even though the mesh nodes form a connected network when all nodes use a common channel, they may no longer be connected if some single interface nodes are only connected to other single interface nodes. It is possible to have a more elaborate protocol which ensures that a set of connected single interface nodes are also connected to at least one multi-interface node, but we defer such an extension to future work.

C. Support for client nodes

Client nodes do not run any of the software that we have implemented. Instead, client nodes view mesh nodes as “access points” and connect to them using standard IEEE 802.11 protocol rules. A client node is allowed to connect to any of its neighboring mesh nodes on the fixed interface of the mesh node. In typical wireless networks, clients connect to access points using the “managed” mode provided

[8]Note that a multi-interface node M would receive the message only if the channel on which the message has been sent happens to be M’s fixed channel.

for in IEEE 802.11. However, in our implementation mesh nodes are connected to each other using the "ad hoc" mode, and therefore a client will have to connect to the fixed interface of a mesh node in the "ad hoc" node. An alternate solution would be to have an extra interface at each mesh node, and set up the extra interface to operate as an access point in the managed mode. Then, clients could connect to the extra interface in the managed mode. Our approach (of connecting to the fixed interface) avoids the need for an extra interface.

Our implementation requires the mesh nodes to be assigned addresses with a common subnet prefix, while client addresses must belong to a different subnet prefix. The userspace daemon adds a default entry into the unicast table of the channel abstraction layer. The default entry ensures that packets to any destinations not in the unicast table are sent out on the node's fixed channel. This ensures that packets sent to clients are sent out on the fixed interface (because no entries are added into unicast table for clients). Clients have to be configured to connect to any of the mesh nodes in its vicinity on the fixed channel of the mesh node[9]. In our implementation, clients are allowed to only communicate with nodes in the Internet, or with other nodes in the mesh network. However, our implementation could be extended to allow mesh nodes to connect to client nodes as well. Packets sent out by a client are network address translated by the mesh node to which it is connected, and then sent on to the Internet or another mesh node. Figure 8 illustrates the process of a client communicating with a node in the Internet. Apart from the minimal configuration requirements, clients can run without requiring any other changes.

X. Testbed experimentation

The multichannel implementation has been evaluated on a testbed that we have built. Nodes in the testbed comprise of *Net 4521* boxes from Soekris [27]. Figure 9 shows a picture of a testbed node. 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 [19], and support IEEE 802.11a/b/g protocols.

[9]The standard IEEE 802.11 beacons continue to be sent out on the fixed channel in our implementation, and enables clients to discover the channel on which a mesh node is available.

Fig. 9. Picture of a testbed node. Each testbed node is equipped with two wireless interfaces.

The multichannel implementation has been carefully tested for correctness on the testbed. We have set up scenarios with gateway nodes, single interface mesh nodes and client nodes, and verified the correct operation of the protocols. The performance of channel abstraction layer has been extensively studied in Chereddi's thesis [21]. The CAL performance was measured while operating it as part of the multichannel implementation described here (we do not restate those results here), and the results there demonstrate significant performance improvements when multiple channels are used. In addition, [21] also includes a study on the interface switching delay and a study on the number of orthogonal channels available in our testbed. In the rest of this section, we present sample results to demonstrate the benefits of our implementation in single hop and multihop scenarios.

A. Single hop experiments

We measure the performance of the multichannel implementation under a single hop scenario. Five nodes are placed such that each node can directly communicate with any other node. The nodes are numbered from 0 to 4 and node i sets up a flow to node $(i+1) \bmod 5$. We vary the number of channels in the network from one to four. All nodes are equipped with two interfaces. The channel data

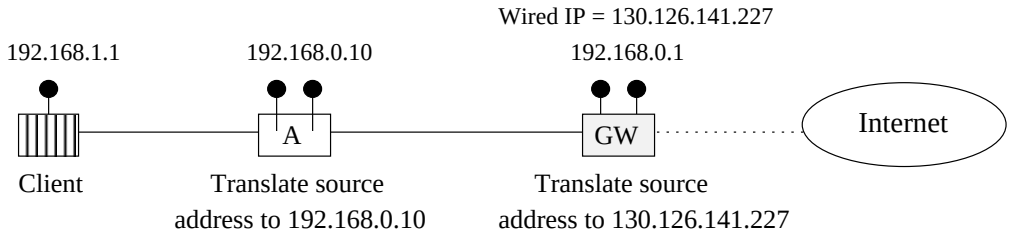

Fig. 8. Client communication process. Client packets are network address translated at its mesh router, node A. Example assumes that client address space is 192.168.1.*, while mesh node address space is 192.168.0.*.

rate is set to 6 Mbps. The experiments are run with both TCP and UDP traffic (flows are run for 100 s), and traffic is created using the iperf tool [28]. Iperf creates back-logged traffic.

Figure 10 plots the aggregate throughput for both TCP and UDP traffic. As we can see from the figure, when more channels are available, there is a substantial improvement in the throughput. With UDP traffic, each node has data to transmit on exactly one channel. Therefore, interface switching is not required at any of the nodes. However, with TCP traffic, each node has to send TCP data packets along the flow it is initiating, and send TCP ACK packets along the flow it is receiving data. These two data streams may potentially be on different channels, and therefore, may require interface switching (in experiments using three or four channels). However, we see that in spite of interface switching, TCP performance is comparable to UDP (up to three channels).

We have found that TCP performance starts degrading when more than three channels are used. Looking at the TCP traffic received over one second intervals, we see that in certain intervals the nodes receive no packets, implying that TCP timeouts are occurring in experiments with more than three channels. TCP timeouts arise when there are multiple packet losses. We speculate that packet losses are occurring because of cross-channel interference when more than three channels are used. Our interference experiments (see [21] suggested that five orthogonal channels may be available for use in our testbed. However, those experiments only considered aggregate throughput and did not consider packet losses. In addition, those results considered interference when traffic was present on only two channels at a time. In this testbed experiment, all nodes are close to each other, and traffic is present on all channels simultaneously, which may lead to increased cross channel interference. UDP does not slow down in the face of packet losses. Therefore, UDP throughput continues to improve even if a few packets are lost because of cross-channel interference.

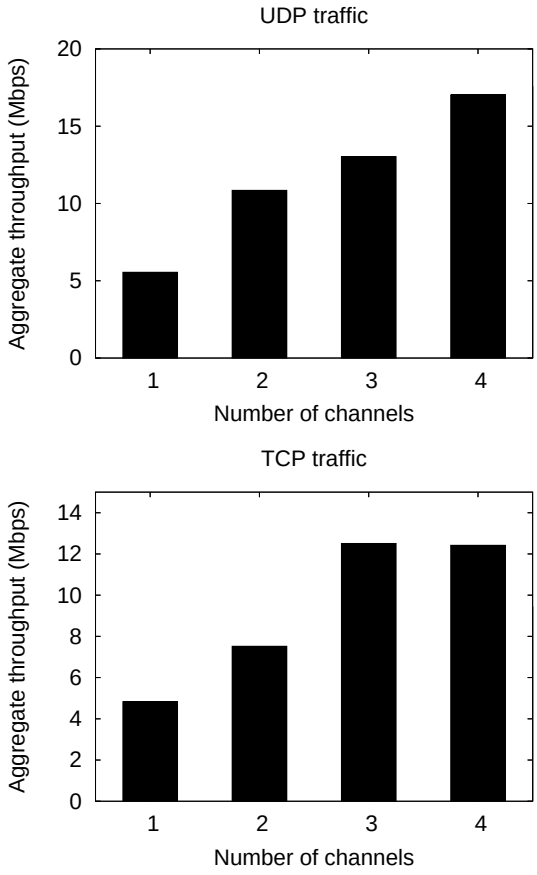

Fig. 10. Throughput in a single hop network with varying number of channels. Channel data rate is 6 Mbps.

B. Multihop experiments

The performance of the multichannel implementation under multihop scenarios is measured by setting up a chain of five nodes. A flow (TCP or UDP) is set up from the first node in the chain to the last node in the chain (i.e., a 4-hop flow is set up). The flow throughput is measured while the number of channels is varied from one to four. Channel data rate is set to 6 Mbps.

Figure 11 plots the flow throughput for both TCP and UDP traffic. As we can see from the figure, when more channels are available, there is a substantial improvement in the throughput, though the magnitude of improvement is less with TCP traffic. TCP throughput depends on the end-to-end delay in addition to the available bandwidth. The interface switching delay is 5 ms with our testbed nodes, and the delay experienced by packets could be higher because of the queuing introduced by the channel abstraction layer. We suspect that this increased delay could be limiting the performance improvements. Note that although a single TCP flow may not utilize all the available bandwidth, other flows in the vicinity could still utilize the unused bandwidth. Therefore, the aggregate network throughput is still expected to increase even with multihop traffic. Similar to one hop experiments, TCP throughput in a chain topology is also lower with more than three channels. As before, we speculate that this is because of the increased packet losses arising out of cross channel interference.

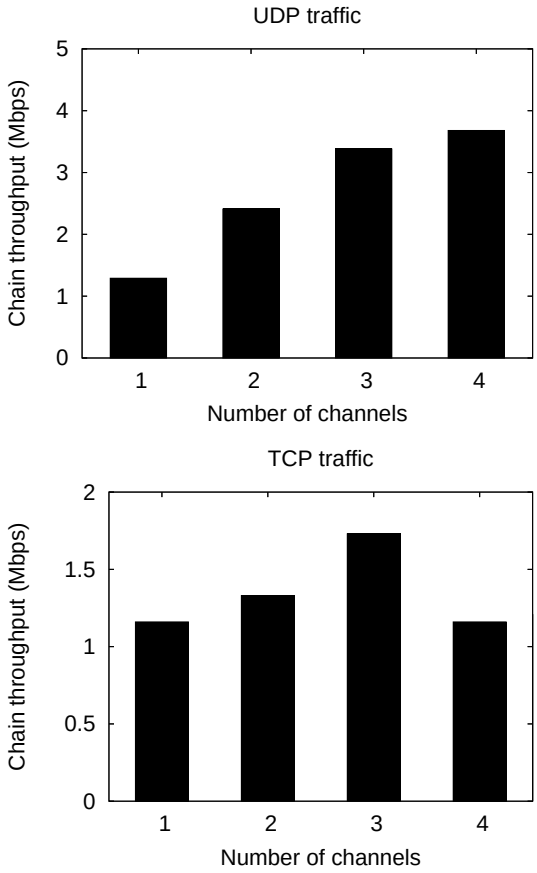

Fig. 11. Throughput in a 4-hop chain topology with varying number of channels. Channel data rate is 6 Mbps.

XI. Conclusions

In this report, we described the implementation of the one set of multichannel protocols on the Net-X testbed. The prototype implementation has demonstrated the feasibility of using multiple channels when only two interfaces are available. As part of the implementation, we developed a generic channel abstraction architecture to support multichannel protocols. The architecture and the protocol suite that we have developed may be useful to develop other multichannel protocols as well.

The channel abstraction architecture was designed to support the use of multiple channels, multiple interfaces, and frequent interface switching. Several commercially available radios now export a rich set of interface capabilities to the user, such as the ability to set the data rate and transmission power on a per-packet basis. However, to benefit from these features, new higher layer protocols may be required. We argued that the implementation of higher layer protocols that utilize other interface capabilities will require new kernel support as well, and provided examples to identify how the channel abstraction layer may be extended to provide the desired support. We believe that the channel abstraction layer design can serve as a blueprint for building support for most of the interface capabilities. It is part of our ongoing work to implement protocols to exploit other interface capabilities on the Net-X testbed.

References

[1] Pradeep Kyasanur, *Mutichannel Wireless Networks: Capacity and Protocols*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 2006.

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

[3] D. Maltz, J. Broch, and D. Johnson, "Experiences Designing and Building a Multi-Hop Wireless Ad-Hoc Network Testbed," Tech. Rep. TR99-116, CMU, 1999.

[4] H. Lunndgren, D. Lundberg, J. Nielsen, E. Nordstrom, and C. Tscudin, "A Large-scale Testbed for Reproducible Ad Hoc Protocol Evaluations," in *WCNC*, 2002.

[5] B. A. Chambers, "The Grid Roofnet: A Rooftop Ad Hoc Wireless Network," M.S. thesis, MIT, 2002.

[6] R. Karrer, A. Sabharwal, and E. Knightly, "Enabling Large-scale Wireless Broadband: The Case for TAPs," in *Hotnets*, 2003.

[7] B. White, J. Lepreau, and S. Guruprasad, "Lowering the Barrier to Wireless and Mobile Experimentation," in *Hotnets*, 2002.

[8] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh, "Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols," in *WCNC*, 2005.

[9] P. De, A. Raniwala, S. Sharma, and T. Chiueh, "MiNT: A Miniaturized Network Testbed for Mobile Wireless Research," in *Infocom*, 2005.

[10] Atul Adya, Paramvir Bahl, Jitendra Padhye, Alec Wolman, and Lidong Zhou, "A Multi-Radio Unification Protocol for IEEE 802.11 Wireless Networks," in *IEEE International Conference on Broadband Networks (Broadnets)*, 2004.

[11] Ranveer Chandra, Paramvir Bahl, and Pradeep Bahl, "MultiNet: Connecting to Multiple IEEE 802.11 Networks Using a Single Wireless Card," in *IEEE Infocom*, Hong Kong, March 2004.

[12] Richard Draves, Jitendra Padhye, and Brian Zill, "Routing in Multi-Radio, Multi-Hop Wireless Mesh Networks," in *ACM Mobicom*, 2004.

[13] Ashish Raniwala and Tzi-cker Chiueh, "Architecture and Algorithms for an IEEE 802.11-Based Multi-Channel Wireless Mesh Network," in *Infocom*, 2005.

[14] "Virtual Wifi software page," http://research.microsoft.com/netres/projects/virtualwifi.

[15] Patrick Stuedi and Gustavo Alonso, "Transparent Heterogeneous Mobile Ad Hoc Networks," in *ACM MobiQuitous*, 2005.

[16] Nicolas Boulicault, Guillaume Chelius, and Eric Fleury, "Experiments of Ana4: An Implementation of a 2.5 Framework for Deploying Real Multi-hop Ad-hoc and Mesh Networks," in *REALMAN*, 2005.

[17] Vikas Kawadia, Yongguang Zhang, and Binita Gupta, "System Services for Implementing Ad-Hoc Routing: Architecture, Implementation and Experiences," in *Mobisys*, 2003.

[18] "Netfilter: Linux packet filtering framework," http://www.netfilter.org/.

[19] "Atheros Inc," http://www.atheros.com.

[20] John Bicket, Daniel Aguayo, Sanjit Biswas, and Robert Moris, "Architecture and Evaluation of an Unplanned 802.11b Mesh Network," in *ACM Mobicom*, 2005.

[21] Chandrakanth Chereddi, "System architecture for multichannel multi-interface wireless networks," M.S. thesis, University of Illinois at Urbana-Champaign, 2006.

[22] "AODV-UU: AODV implementation from Uppsala University, version 0.9.1," http://core.it.uu.se/AdHoc/ImplementationPortal.

[23] Matthew J. Miller, William D. List, and Nitin H. Vaidya, "A Hybrid Network Implementation to Extend Infrastructure Reach," Tech. Rep., University of Illinois at Urbana-Champaign, 2003.

[24] "Champaign-Urbana Community Wireless Network," http://www.cuwireless.net/.

[25] "Seattle Wireless," http://www.seattlewireless.net/.

[26] "Municipal Wireless," http://www.muniwireless.com/.

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

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