© 2011 by Lu-chuan Kung. All rights reserved.

UNIFIED CROSS-LAYER FRAMEWORK: A GENERIC PLATFORM FOR CROSS-LAYER DESIGN EXPERIMENTATION

BY

LU-CHUAN KUNG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:
Professor Nitin Vaidya

Abstract

Cross-layer designs in wireless network systems have been an active research area. Numerous cross-layer schemes are proposed to improve overall system performance by allowing information to be shared and controlled across protocol layers. However, much of the previous research work in this area is simulation-based. The main obstacle which hinders researchers from real implementation is the complexity involved in lower-level driver modification and kernel programming. Moreover, common pitfalls of cross-layer scheme implementation can lead to unexpected system performance degradation.

In this work we propose unified cross-layer framework (UCF), a generic mechanism for OS to support cross-layer schemes. Through this mechanism protocol components export protocol-specific information through parameters and events to protocol components at other layers. Cross-layer extensions are activated only at necessary times according to their assigned priorities so that they can react to events and make decisions at different time scales with minimal overhead. We implement and evaluate UCF on embedded systems running NetBSD to demonstrate the modularity, ease of programming, and utility of UCF.

To my parents, my wife, and my family, for their love and support.

Acknowledgments

The majority of the work of this thesis was performed under the guidance of the late Professor Jennifer Hou. I am grateful to her inspiration and encouragement. I am also very thankful to my advisor, Professor Nitin Vaidya, for his support and patience to help me finish this thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A multi-hop wireless network is a collection of wireless nodes which cooperatively establish communication, without use of fixed infrastructure or centralized administration. It has gained tremendous attention in recent years because of its wide applications in civilian and military areas, and its ability to provide connectivity without the need for a pre-existing infrastructure. An example multi-hop wireless network is the *wireless mesh network* (a.k.a. the *community wireless network*) for providing broadband access [1, 2, 3].

Although initial success has been reported, a number of performance-related problems have also been identified. Excessive packet losses [4, 5, 6], unpredictable channel behaviors [5, 6], inability to find stable and high-throughput paths [5, 6], throughput degradation due to intra-flow and inter-flow interferences [7, 8, 4], and lack of incentives (and a pricing mechanism) to forward transit packets are among those most cited.

All the above problems (except perhaps for the incentive issue) are rooted in the fact that *the notion of a link is different in wireless environments*. In network theory and practice, a link is usually characterized by its bandwidth, latency, and loss rate. However, in the case of wireless networks, a wireless medium is *shared* among nodes, and the *sharing range* is determined by (i) several PHY/MAC attributes such as transmit power and carrier sense threshold, and intra-/inter-flow interference and (ii) several environmental factors such as multi-path fading, shadowing, scattering, presence of obstacles, and temperature and humidity variations. As a result, all the definitive metrics that characterize a link are no longer well defined in the wireless context.

Because PHY/MAC attributes and environmental factors have a profound impact on higher-layer protocols [5, 9], the notion of cross-layer design and optimization has been proposed to optimize the overall performance of wireless networks. In spite of a large amount of theoretical research results that demonstrate the advantages of cross-layer design and optimization [10, 11, 12, 13], there has *not* been extensive device driver support that *exports PHY/MAC characteristics via a set of rich, well-defined APIs* and help realizing the notion of cross-layer design and optimization. Very often, research results on cross-layer design and optimization have been (in-)validated, via either analytical reasoning built upon abstract models or simulations using over-simplified PHY/MAC models. For the few efforts that go beyond theoretical derivation

and simulation [6, 1, 2], the software (such as customized device drivers, address resolution modules, routing daemons, and name servers) is often implemented in an ad-hoc manner, lacks in structural modularity, and does not come with well-defined APIs for experimentation and performance tuning. An open, modular programming environment is crucial to the understanding of whether or not, and to what extent, the performance of wireless networks benefits from cross-layer design and optimization.

In this work, we propose, in compliance with the guidelines given in [10, 14], the notion of *unified cross-layer framework* (UCF). UCF serves as a general interface to pass messages and control signals between protocol modules at *any* layer. It allows a protocol module to export certain attributes and events, thus other protocol modules can access their interested exported attributes and and register callback functions which are triggered in the case of event occurrence. In UCF, exporting and dynamic control of low-level attributes at varying granularities, such as per-packet, per-link, per-session or per-interface, are enabled through well-defined APIs to facilitate cross-layer design and optimization.

To demonstrate the utility of UCF, we have implemented several cross-layer schemes that tunes bit-rates and transmit power using cross-layer information. We also present an example in which the chosen event delivery mechanism have a large impact on system performance. In addition, we measure the overhead of event delivery mechanisms of UCF. The results showed that with slight increase of processing time, UCF provides simple but effective building blocks for implementing cross-layer schemes.

The remainder of the work is organized as follows. In Chapter 2, we introduce the background and explain the problem and challenge. In Chapter 3 we review related work. In Chapter 4, we elaborate on how we design and implement the unified cross-layer framework. In 4.4, we present the wireless extension interface designed to export PHY/MAC parameters and events on an IEEE 802.11 interface. In Section 5.1, we demonstrate the utility of UCF by giving examples of cross-layer control modules that adapts transmit power and bit-rate, as well as a schedule-based medium access algorithm. Following that, we present performance evaluation and empirical results in Section 5.3. Finally, we conclude the work in Section 6.

Chapter 2

Background

2.1 Layer versus Cross-Layer

Network protocols are usually designed with a *layered* architecture. One layer of protocol provides only a subset of network functionalities and delegates remaining functionalities to other layers. For example, TCP provides reliable transmission while IP deals with routing packets. Such design reduces the complexity of protocol design and implementation. It also greatly influences the design of network software. As a result, network software are usually organized as a stack of software layers, in which one layer only directly communicates with its adjacent layers through some interfaces.

However, with the emergence of wireless networks the notion of cross-layer approach is introduced. The idea is that in wireless network, the problem of packet routing and scheduling has to be considered jointly to obtain a theoretically optimal solution. Consequently, in order to improve network performance, the barriers that we imposed intensionally between non-adjacent layers need to be broken. The question is: do we really have to sacrifice software modularity for performance? In this work, we argue that we can avoid this trade-off by designing a clean interface for cross-layer protocols. Thus a cross-layer protocol can be implemented in a way that is modular, maintainable, and portable.

2.2 Models of Cross-Layer Schemes

As summarized by Fu at el. [15], based on which layer is making control decisions, cross-layer schemes can be categorized in four basic types: application-centric, middle layer-centric, centralized, and autonomous. In general, in order to make cross-layer decision, a decision maker must gather information from other layers with the right *scope* and *timescale*. Scope defines *which* entities that should report their status to a decision maker. For example, at the TCP layer, we could be interested in either the throughput of one session, or the aggregate throughput of all active sessions. At the wireless data link layer, we might be interested in the number of failed packet transmissions for a wireless NIC, or the number of failed transmissions for

a particular link, or even the number of failed packets on a link, with a particular modulation. Timescale defines *how often* should the decision maker be notified with the update of status from the interested entities. Different layers usually have a different requirement of the freshness of feedback. For example, link layer adaptation such as bit-rate adaptation and packet retransmissions usually requires immediate feedback. On the other hand, application-level schemes are usually more interested in statistics of a longer time duration, such as a round-trip time (RTT), therefore a generic cross-layer framework should provide a decision making module the flexibility to choose the timescale in which it gets status updates.

2.3 IEEE 802.11 Implementation in NetBSD

In this section, we briefly introduce how the 802.11 networking modules are organized in NetBSD 4.x/5.x kernel. NetBSD was chosen because it is highly customizable and the size of a complete NetBSD image can be crammed into a 16MB compact flash card. Also at the time this work started, NetBSD had the most robust driver support of Atheros NIC and 802.11 networking in kernel.

The organization of 802.11 networking stack in NetBSD is shown in Figure 2.1. At the top is the generic network device layer. The next layer is a 802.11-specific layer, which deals with all the 802.11-specific functions. To the bottom is the device driver layer, which handles all the hardware-specific operations of a wireless NIC.

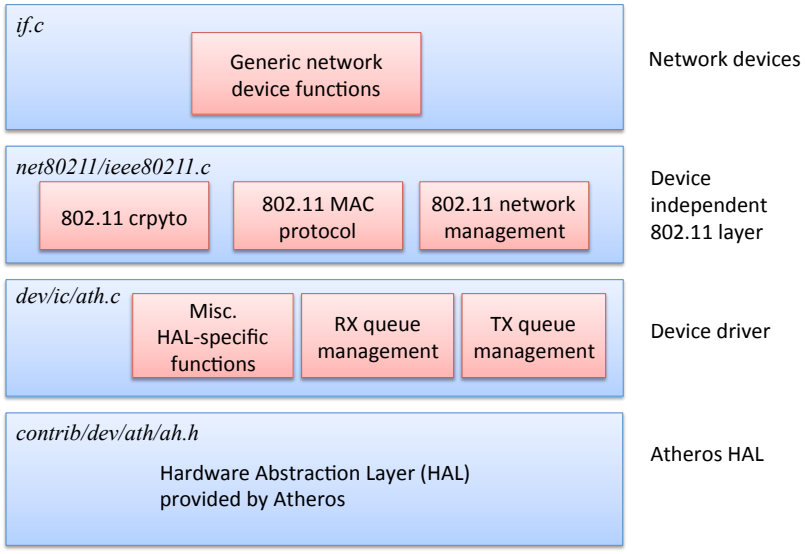


Figure 2.1: 802.11 module and device driver architecture in NetBSD.

The data structures that actually store the device information at each layer are linked together as shown in Figure 2.2. Structure `ieee80211com` stores all the vendor-independent 802.11 information of a NIC, such as channel information, available base stations, RTS/CTS threshold, ...etc. Per-link information such as available bit-rates for the link and statistics for the link, RSSI of received packets is stored in struct `ieee80211_node` . Hardware-specific data structure `ath_softc` stores hardware-specific data such as interrupt handlers, memory descriptors for DMA transfer, vendor-specific statistics of the interface.

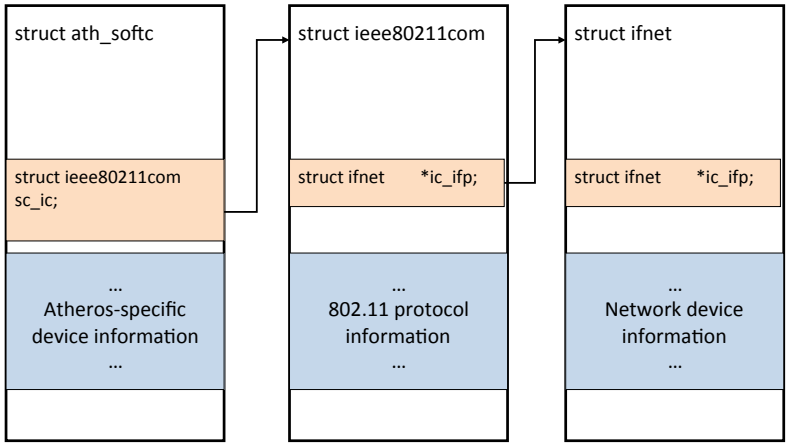


Figure 2.2: 802.11 module and device driver data structures in NetBSD.

2.3.1 Packet Transmission

Figure 2.3 diagrams how data link layer serves a request to send a 802.11 data packet from an upper layer in NetBSD. An upper layer protocol first put a packet `mbuf` into the queue of the interface through `ifq_enqueue`. It then calls the `if_start` function of that interface to start transmission. In our example, `if_start` function will points to `ath_start` function, which calls several helper functions at the 802.11 layer to obtain the necessary information in order to set up hardware-specific transmission descriptors. It then calls `ath_start` which puts the per-packet descriptors into a transmit queue. The SoC on the card then initiates transmission as long as it thinks the channel is available.

2.3.2 Packet Reception

The flow diagram of packet reception in NetBSD is shown in Figure 2.4. When a packet is received by the wireless NIC, it triggers a hardware interrupt, which is handled by the registered interrupt handler

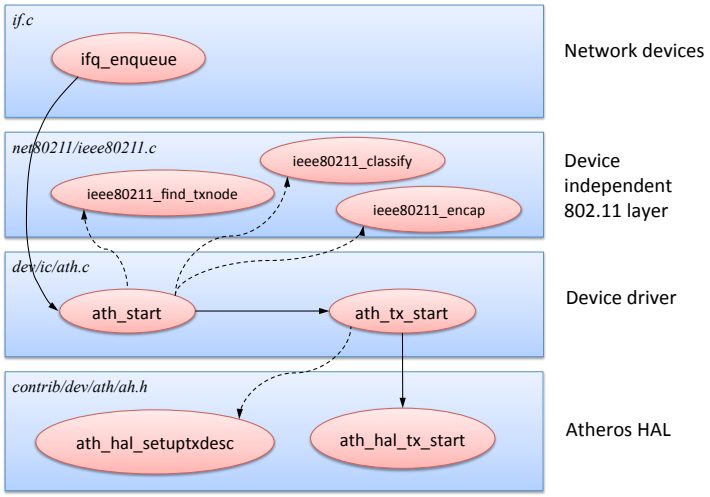


Figure 2.3: Flow graph of packet transmission in NetBSD.

`ath_intr`. This generic handler then calls a reception-specific handler `ath_rx_proc`. This handler's main job is to maintain DMA descriptors and prepare a `m_buf` structure of the received packet before passing the packet to the 802.11 layer. After some 802.11 protocol processing, the packet is passed to the `if_input` handler, which is usually the IPv4 packet input handler, if the NIC was set up as an IPv4 interface.

2.4 802.11e Differentiated Service in NetBSD

IEEE 802.11e defines quality of service (QoS) extensions to 802.11. In the enhanced distributed channel access (EDCA), four *access categories*, are defined for background, best effort, video, and voice traffic. In NetBSD, the access category of a packet is determined by either the 802.11d tag of the Ethernet header or the TOS field of the IP header of the packet. The Atheros NIC maps each access category to a hardware queue. Each queue has its own QoS parameters, as defined in 802.11e. Users can change the parameters of a particular access category by system call `ioctl`, which is dispatched to function `ieee80211_ioctl_setwmeparam` in the 802.11 layer.

2.5 Motivating Examples

Rate adaptation algorithms have been an active research topic [16, 17, 18]. The open-source Atheros device driver in NetBSD includes a few rate selection schemes: AMRR[19], ONOE, and SampleRate[20]. These

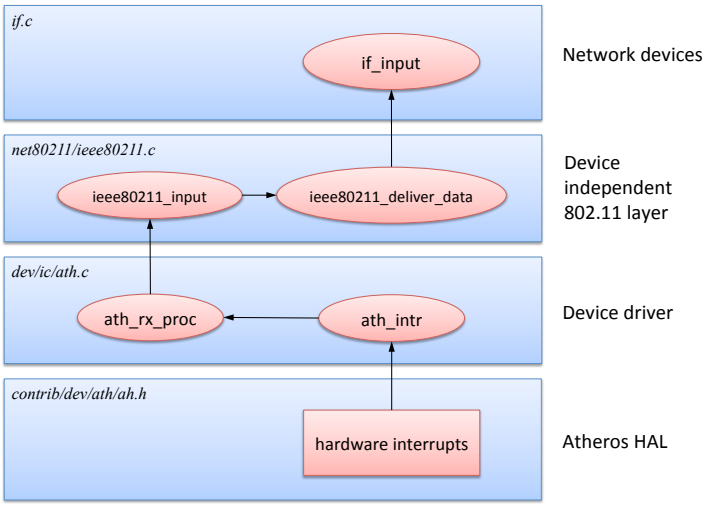


Figure 2.4: Flow diagram of packet reception in NetBSD.

schemes are implemented according to an API specifically designed for rate selection. A rate control module provides certain callback functions to the device driver so that the module is notified when certain event of interest occurs. For example: before transmitting a frame, the driver consults the rate control module for the bit-rate to use. After the transmission is completed, the driver sends the feedback of the transmission back to the rate control module. A simplified diagram that illustrates the interactions between the Atheros driver and rate control module is shown in Figure 2.5.

As shown in the example, a common approach to implementing cross-layer design/optimization and enable interaction between protocols in different layers is to modify the device driver and insert callback functions of a higher-layer protocol module wherever desirable. There are a number of drawbacks in this callback architecture with respect to the system design, implementation complexity and performance:

- **Lack of priority differentiation**. Callback functions are executed in the same thread context of a device driver. In many cases the calling device driver is in the process of handling an interrupt when it invokes a callback function. Calling a callback function in an interrupt handler may increase the interrupt processing time and may thus decrease the overall system throughput. To demonstrate this problem, we insert a delay procedure into one of the rate control module in the Atheros driver to simulate a computational intensive adaptive procedure. We then measure the TCP throughput between a sender and a receiver node where the receiver node has the long-delay rate control module. As shown in Figure 2.6, the overall system performance is affected by the cross-layer callback function as an unintended side-effect.

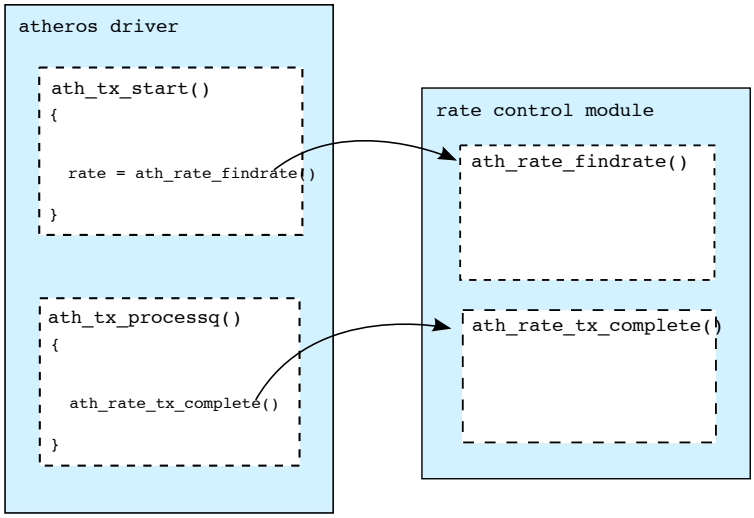


Figure 2.5: The rate control module in the Atheros driver on NetBSD

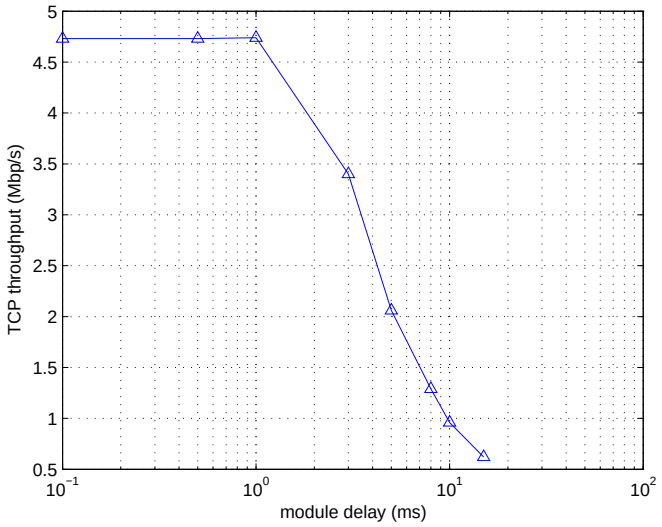


Figure 2.6: TCP throughput with different module delay

- **Lack of asynchronous events**. As shown in the example, sometimes a device driver needs certain feedback from a cross-layer module, but at some other times it does not. In the latter case, the call can be handled in an asynchronous manner (e.g., buffering). The aforementioned ad-hoc approach, however, does not differentiate the two cases and thus the device driver has to wait for the completion of a callback function every time when an event occurs. This incurs a larger overhead.

- **Lack of dynamic binding**. Because a callback function is directly invoked by a device driver, the address of the callback function must be known at compile time. This implies that, each time the device driver switches to another cross-layer control module or a new cross-layer control module is added, the device driver has to be re-compiled and reloaded.

- **Tight coupling of the device driver and higher-layer protocol modules**. Each time a cross-layer control module is implemented, the device driver has to be modified. This involves tracing the source code of the device driver and locating the right places to insert callback functions. This compromises the design goal of the layered network architecture.

- **Lack of an unified open API.** The lack of an open API makes it difficult to integrate cross-layer schemes in a systematic manner. Ad-hoc insertion of callback functions in the device driver may eventually make the code difficult to debug and maintain. It also makes it difficult to simultaneously load and run multiple cross-layer control algorithms, even if they control orthogonal PHY/MAC parameters.

2.5.1 Rate Adaptation Using Unified Cross-Layer Framework

Figure 2.7 shows how the above rate-control module can be implemented using the proposed unified cross-layer framework(UCF). Essentially the framework provides two major interfaces for both the device driver and cross-layer control modules. It provides (i) for the device driver an API to export events and register parameter handlers; and (ii) for the cross-layer control module an API to subscribe events of interest and register event handlers. These interfaces provide a clean separation between the driver and the modules. The driver exports a certain set of events and parameters which constitute part of an extension. A cross-layer control module that understands the extension can subscribe to events of interest and access parameters through the *unified cross-layer manager*. We will elaborate on the UCF and its features in Section 4.2.

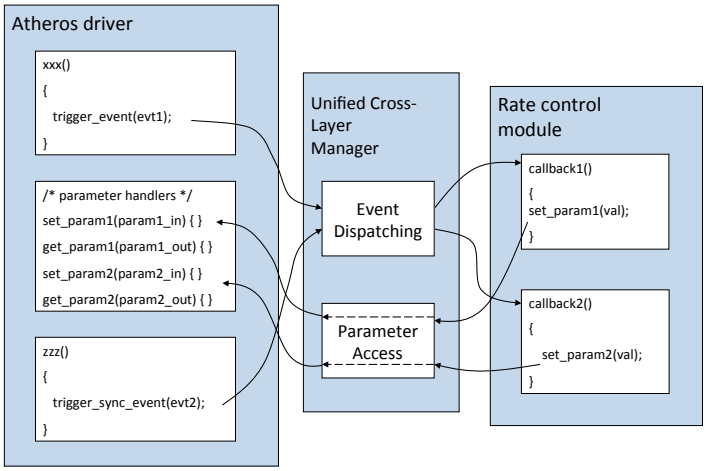


Figure 2.7: Rate control module using the proposed unified cross-layer framework.

Chapter 3

Related Work

3.1 Accessing Information from Drivers and Network Stacks

Common operating systems usually provide some mechanisms to configure parameters of software components such as TCP/IP protocol stacks or hardware devices like a 802.11 network interface card (NIC). In Linux for example, *sysfs*[21], introduced in version 2.6, is a virtual file system that exports device information and configuration options as simple text files. Users can discover and explore all devices in the system by traversing the directories and symbolic links of sysfs.

Linux Wireless Extension is the initial interface for supporting 802.11 devices. It defines a common set of `ioctl` system calls for configuring 802.11 devices. However, `ioctl` is not flexible enough to support complicated configurations, nor does it support notifications from kernel. Therefore *cfg80211* and *mac80211* [22] are introduced as a replacement of the Linux Wireless Extension. Building on top of *netlink*, which is a generic interface for userspace process to communicate with kernel in Linux, cfg80211 provides a unified interface to configure 802.11 interfaces. Sitting on top of cfg80211 is the mac80211 layer, which deals the details of 802.11 protocol, such as authentication, processing and generating management frames, and supporting ad hoc and mesh modes.

Similarly on Windows systems, Network Driver Interface Specification (NDIS) of 802.11 WLAN drivers defines a set of configuration and indication objects for 802.11 device drivers. Such OS support of 802.11 focuses on configuration of 802.11 devices but does not consider the issues with cross-layer designs such as latencies and priorities. UCF, on the other hand, enables fine-grained control of the tradeoff between event latency and performance overhead.

3.2 Cross-Layer Control

The IEEE 802.11 standard [23] defines a layer management interface, which allows a layer-independent management entity to gather information from MAC and PHY layer and to set the values of layer-dependent

parameters. While this abstract interface can be seen as a subset of the UCF API, our framework extends this interface and provides more functionalities and considers practical operating system issues such as the context switching, locking issues, latencies and event buffering.

3.3 Software MAC

The notion of implementing MAC layer functions entirely in software, i.e. *software MAC*, is not a new concept. SoftMAC [24] explores the possibilities to implement new MAC, on top of Atheros wireless NIC. It is an extension to the Madwifi driver for the Atheros 802.11a/b/g networking cards that provides the following controls over the MAC layer: (i) overriding 802.11 MPDU frame format; (ii) eliminating automatic ACK and retransmission; (iii) eliminating RTS/CTS exchange; (iv) eliminating virtual carrier sense (NAV); and (v) controlling PHY clear channel assessment (CCA) and transmission backoff. Inspired by this work, we extend the notion of software MAC to allow tuning a *complete* suite of PHY/MAC characteristics for cross-layer design and optimization: transmit power, carrier sense threshold, channels, data rates, and frame transmission schedule.

Click modular router [25] is an effort to enables a normal PC to become an efficient router. In Click, routers are constructed by inter-connecting *click elements*, which perform simple computing tasks as part of routing. From the point of view of an operating system, Click bypasses the whole networking stack. It intercepts packets from network devices right after reception and sends packets directly to devices for transmission, after the actual processing is done entirely inside Click. While Click also promotes modularity and reusability of software components, it does not reuse any of the existing, well-tested and fine-tuned protocol modules of OS kernel. On the contrary, UCF lets developers to build cross-layer-aware protocols, on top of existing kernel modules, which greatly reduces the complexity of such task.

3.4 Cross-Layer Signaling

Abu et al. [26] discuss how to pass information across layers in layered network system. While the goal of facilitating cross-layer signaling of their work is similar to ours, their discussion is only conceptual and lacks consideration of practical issues and actual implementation.

NS-Miracle[27] is a ns2 library that provides a mean for cross-layer modules to communicate with each other. Conceptually similar to UCF, NS-Miracle emphasizes the importance of a standard mechanism for cross-layer messaging. However, NS-Miracle is only a simulation library while UCF is implemented on a real system that deals with all the complicated kernel programming issues: interrupt handling, locking, memory

management, and DMA. UCF greatly lightens the burden of protocol designers to implement and evaluate their cross-layer protocols on a real testbed.

Chapter 4

Unified Cross-Layer Framework

In this chapter we discuss the design and internals of Unified Cross-Layer Framework (UCF). First we summarize the design principles of UCF. We then introduce the architecture and major components of UCF. We then discuss the implementation details of UCF, before delving into the details of the 802.11 extension interface.

4.1 Design Guidelines

The UCF is designed to provide the following features:

- Controlled transparency: The UCF provides a transparent and generic interface for higher-layer protocol modules to access, through well-defined APIs, a rich set of PHY/MAC attributes and functionalities in the device driver. Through an event subscription mechanism, higher-layer protocol modules can also receive timely update of the channel status, without directly inserting callback functions in various places of the device driver.

- Flexibility: The design philosophy of the UCF is to provide minimum but crucial functionalities that enable implementation of complicated cross-layer design/control algorithms. The event subscription mechanism is simple, elegant, and allows *multiple* higher-layer protocol modules to subscribe, and be alerted of, PHY/MAC events of interest. They can also register with the event subscription mechanism their callback functions, allowing adequate actions to be taken upon occurrence of the event of interest. Moreover, the UCF allows the time granularity at which PHY/MAC properties are controlled to be on a *per-packet* or *per-link*, *per-interface*, or *per-system* basis.

- Easy integration and portability: Each extension interface register its exported parameters events to the UCF independently. Multiple extension interfaces can be dynamically loaded. An cross-layer protocol module (e.g., a routing daemon) can be extended to subscribe events of interest (e.g., the frame reception status upon frame arrival), and realize cross-layer design/optimization by adequately

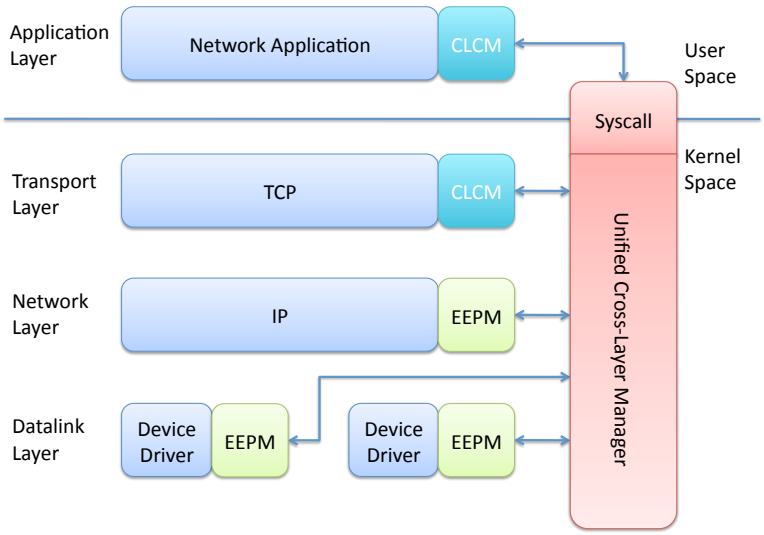


Figure 4.1: The architecture of the unified cross-layer Framework

gaining access to/control PHY/MAC parameters and being timely informed of important events. A cross-layer control scheme falls back to its normal operation if the required extension is not supported by the UCF. This ensures backward compatibility.

4.2 Architecture and Major Components

Figure 4.1 shows the architecture of the unified cross-layer framework (UCF). Different from the traditional layered approach, an extension-enabled device driver exports PHY/MAC parameters and events to higher-layer protocol modules. The major components in the UCF are as follows:

- Unified Cross-Layer Manager (UCLM): UCLM is the major component of the UCF. It is responsible for (i) loading and unloading extensions, (ii) providing an API for cross-layer control modules to register events of interest and the associated callback functions; (iii) allowing control modules to read and write the exported attributes of a protocol module via handlers registered by the protocol module; (iv) maintaining event definition and subscription, and (v) dispatching events to subscribing control modules. We will elaborate on its internals later in Section 4.3.

- Extension-enabled protocol module (EEPM): A protocol module such as device driver registers itself to the UCLM to export a set of PHY/MAC attributes and events in the form of *extension specification*.

The specification serves as service agreements between a protocol module and a cross-layer control module that uses it. To implement an extension, a protocol module implements the get/set handlers of the attributes it exports. It also defines events, provides the event information to the UCLM, and notifies the manager upon occurrence of events.

- Cross-layer control module (CLCM) : A cross-layer control module implements a cross-layer design/optimization algorithm. As a client to the UCLM, it registers itself with the UCLM in order to use the facilities provided by a EEPM. Through a generic interface, a control module can read and write PHY/MAC parameters exported by an EEPM. Also, it can subscribe to events of interest defined in an extension specification and provide callback functions to be invoked whenever certain events occur.

- Kernel-mode proxy: For user-space programs to gain access to the UCF in the kernel, we introduce a *kernel mode proxy* that serves as a bridge between the two entities. Each UCF API function exported is assigned an unique system call number. The kernel mode proxy is responsible for: (i) translating a UCF-related system call and invoking the corresponding UCF function, and (ii) delivering events to the handler in the user space.

4.3 Internals of the Unified Cross-Layer Manager

The unified cross-layer manager maintains (i) the definition record of all the supported events in an event definition tree; and (ii) the list of subscribers of each event. To provide access to one parameter (or one set of parameters), an EEPM registers one *getter function* and one *setter function* with the UCLM via RegisterSetHandler() and RegisterGetHandler().

A cross-layer control module (un-)subscribes to an event with a callback function by calling AddEventHandler() (RemoveEventHandler()). An EEPM generates and delivers an event to the UCLM (and subsequently CLCM that are interested in the event) by calling TriggerEvent(). Table 4.1 lists the APIs exported by the UCLM.

4.3.1 Accessing Exported Parameters

Parameters exported by an EEPM are controllable parameters which affect the behavior and state of the protocol module. At the first glance, parameter access in UCF may seem similar to the traditional operation of `ioctl()`. The UCLM keeps a table of function pointers for each extension interface. (Recall that the table is updated by an extension interface via RegisterSetHandler() and RegisterGetHandler().) When a

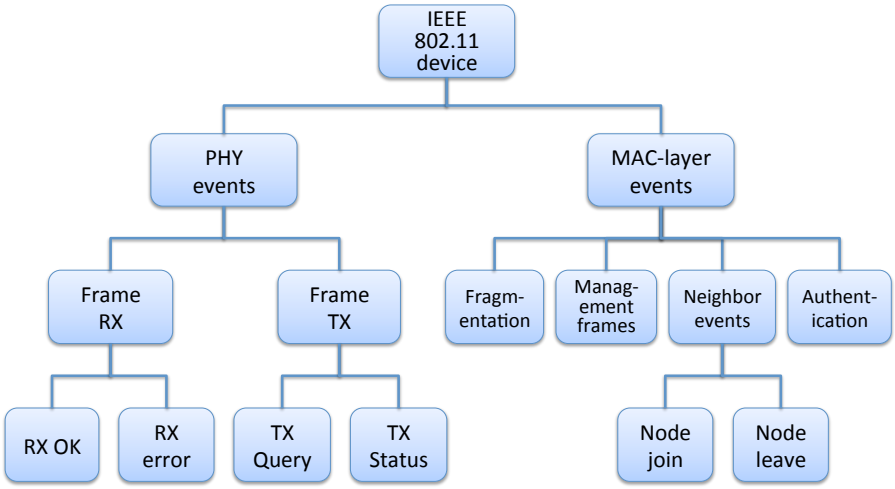


Figure 4.2: The event definition tree of the 802.11 wireless extension.

request to read/write a parameter arrives, the manager invokes the corresponding handler in the table. For example, the 802.11 wireless extension defines a parameter 'channel' to specify the wireless channel on which a wireless interface operates. A driver that provides the 802.11 wireless extension implements (and registers) two handler functions, e.g., setDevChannel() and getDevChannel(). A CLCM can then use the API functions SetExtParam() /GetExtParam() to set/read the current setting of wireless channel, respectively.

What differentiates UCF from ioctl-like functions is that it also supports dynamic and fine-grained access to objects that are usually not accessible through `ioctl()` calls. The lifespan of these objects is usually shorter and associated with some events. Often, one event indicates creation of an object, another indicates its termination, while some other events might indicate change of the state of the object. These events all include an identification of the object, which can be used in the parameter access function to specify the object. For example, in the 802.11 wireless extension (Section 4.4), a neighbor is a data structure associated with each direct neighbor of a node. The neighbor object is created when a new neighbor node is detected and destroyed when the node moves away or leaves the network. A CLCM can use SetExtParam() and GetExtParam() to specify the properties associated with that particular neighbor.

4.3.2 Events Registration and Subscription

An event in the framework triggers certain extension-specific state change. The state change may result from a hardware interrupt, a timer timeout, or a function call from upper layers. Each extension specification defines a number of events. In the UCF, events are organized using a MIB(Management Information

Base)-style naming scheme, in which an event name consists of a list of words, connected by dots. All events supported in UCF (which may come from different extensions) are organized in an internal data structure called the *event definition tree*. Figure 4.2 shows an example of the event definition tree for the 802.11 wireless extension (meaning of each event will be described in Section 4.4). UCF provides functions CreateEventNode(), RemoveEventNode(), and LookupEventNode() to create, remove, and lookup event definition, respectively.

For a cross-layer module to actually receive any event, it must subscribe to its interested events. AddEventHandler()/RemoveEventHandler() is provided to allow an extension interface to register/cancel an event handler for a particular event. Internally the UCLM maintains a subscriber list for every leaf node in the event definition tree to keep track of subscribers and maintain pointers to their event handlers.

4.3.3 Event Delivery and Callback Invocation

Depending on the type of events, there are two possible paths for delivering an event to the manager. A *synchronous event* is an event for which an EEPM requires immediate feedback from its subscribers. When a synchronous event is triggered, it is delivered by the dispatcher in UCLM immediately and the EEPM that triggers the event waits until all the subscriber handlers are finished. An example of a synchronous event is a *transmit query* (Table 4.6), in which prior to the transmission of a frame, the extension-enabled device driver queries registered subscribers for recommendations of a few per-packet transmit parameters such as transmit power, channel, and bit-rate. Thus per-packet control of transmit power, channel, and bit-rate can be implemented. Synchronous events make it possible for cross-layer control modules to make decisions upon occurrence of certain events. An *asynchronous event*, on the other hand, is a *notification* message sent by the EEPM to the subscriber(s) of that event. Upon reception of an asynchronous event, the UCLM inserts the event into the event queue and wakes up the dispatcher. The dispatcher then delivers the event to the corresponding callback functions.

Figure 4.3 shows the internals of the UCLM and the data paths of the two event delivery mechanisms. Synchronous events are delivered directly to the subscribing modules using the original thread which calls TriggerEvent(). As many of the events are triggered by interrupts, TriggerEvent() is likely to be invoked by an interrupt handler. This implies that the event handler for a synchronous event must complete within a short time to ensure that it will not degrade the system performance. Asynchronous events, on the other hand, are buffered in event queues before being delivered by the dispatcher thread. Again, because TriggerEvent() may be invoked by an interrupt handler and delivering events in the context of interrupt handlers is inefficient, we split the task into event creation and callback invocation. The TriggerEvent() function creates an event

object and puts it into event queue. It is now dispatcher's job to invoke the registered callbacks with the event object. Running on its own kernel thread, the dispatcher constantly monitors the event queue and is awakened whenever there is a new event. In this manner, the overhead incurred in interrupt handlers is greatly reduced.

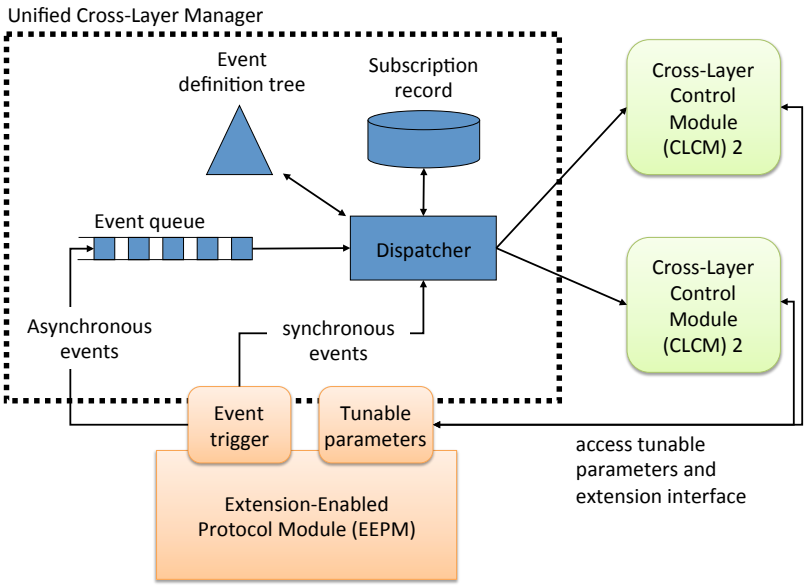


Figure 4.3: Unified cross-layer manager and the flow graph of event delivery.

4.3.4 Implementing Device Drivers as Loadable Kernel Modules in NetBSD

In our NetBSD implementation of UCF, both EEPM and CCLM are implemented as *loadable kernel modules* (LKM). LKMs are dynamic libraries that can be loaded/unloaded at run time to provide new functionalities in NetBSD. Using LKMs in an experimental testbed is beneficial for supporting fast system prototying, allowing drivers/modules to be updated/modified at runtime, allowing the system to fall back to the original default driver after a system crash.

To realize extension interfaces as LKMs in the framework, we modularize the Atheros driver in NetBSD as a LKM. This is more complicated than modularizing other kinds of kernel modules, because it involves the auto-configuration (autoconf(9)) framework of NetBSD. Basically we have to instruct the OS how to load a device by providing two data structures: `struct cfdriver` points to the driver to be loaded and `struct cfattach` instructs OS how to attach the driver in a LKM.

Category	Function name	Function description
Extension management	RegisterExtension() UnregisterExtInterface()	Register/unregister an extension interface module.
	FindExtension()	Query whether an extension interface identified by a unique name or id exists. Return a handle to the interface if it exists.
Register parameter set/get handlers	RegisterSetHandler() UnregisterSetHandler()	Register or unregister a set handler to the unified cross-layer manager.
	RegisterGetHandler() UnregisterGetHandler()	Register or unregister a get handler to the unified cross-layer manager.
Access to extension parameters	GetExtParam()	Get the value of an extension parameter by invoking the registered get handler.
	SetExtParam()	Set the value of an extension parameter by invoking the registered set handler.
Event definition and lookup	CreateEventNode() RemoveEventNode()	Define or remove an event definition in the event definition tree.
	LookupEventNode()	Look up an event definition by its name and return the pointer of the event definition node.
Event subscription and delivery	TriggerEvent()	Generate an event and deliver it to the subscribers.
	AddEventHandler() RemoveEventHandler()	Subscribe to an event with a callback handler function.

Table 4.1: The APIs defined in the unified cross-layer manager.

4.3.5 Accessing PHY/MAC Parameters From the User Space

One design goal of the UCF is to provide uniform access to PHY/MAC parameters, no matter in which layer the cross-layer control module resides. In many cases, programs in the user space (e.g., routing daemons) can benefit from utilizing the cross-layer PHY/MAC information. Because most driver modules reside in the kernel space where user-space programs cannot access, a mechanism is required for user-space programs to gain access to or control PHY/MAC parameters.

There are two reasonable design options for providing access to extensions from the user space: (i) adding new system calls to access extensions; and (ii) enabling user-space programs and the unified cross-layer manager to communicate through IPC mechanisms such as local socket and shared memory.

In the first approach, new system calls are added into the system call table of the operating system. User programs can therefore access extensions through standard system calls. Note that on NetBSD, system calls can be extended through loadable kernel modules. Therefore the entire UCF can be implemented as a LKM. This way it is easy for an application to link with the UCF library and the overhead incurred in this approach is minimal.

In the second approach, the UCLM communicates with cross-layer user-space programs through IPC mechanisms. This approach incurs additional overhead of encoding and decoding function calls and argument list into messages. However, it has the potential to support remote control modules because the socket interface can support both local and remote processes transparently. In our current implementation, we

focus on a local system and hence take the first approach for a lightweight implementation.

4.4 Wireless Extension Interface

In principle, one can design any type of extension interfaces that export attributes and events of interest. In this section, we elaborate in particular on the wireless extension interface, which is designed to export a rich set of PHY/MAC parameters and events on an IEEE 802.11 wireless interface.

4.4.1 PHY/MAC Parameters Exported by the Wireless Extension Interface

We list PHY/MAC parameters supported in the proposed wireless extension. The listing is by no means a complete set of 802.11 configurations but to provide a flavor of the supported parameters. We categorize them into three groups: *per-device*, *per-neighbor*, and *per-packet* parameters. Note that several of the per-device parameters are related to network configuration and thus can also be set by `ifconfig` via `ioctl()` calls. However, `ioctl()` does not support settings of dynamic objects such as per-neighbor and per-node parameters. In contrast, the proposed wireless interface supports per-neighbor and per-packet parameter settings through events that specify the lifetime of such dynamic objects.

Per-Device Configuration

Table 4.2 lists the parameters on a per-device level of a 802.11 wireless interface. In addition to the *setter* commands listed, there are also the corresponding *getter* commands that read the current values of these parameters. For brevity, the getter commands are not listed.

Per-neighbor Transmit Parameters

Table 4.3 gives the parameters that control the transmission behavior on a *per-neighbor* basis. The device driver keeps a table of neighbors and maintains the status and parameter settings for each neighbor. These settings can be retrieved and set using the proposed wireless extension. In particular, one can realize a specific power (rate) control algorithm at the per-neighbor granularity by setting the `txPower` (`rate`) parameter. Before transmission of a frame, the driver searches the table to retrieve the parameter setting for the destination node.

Command	Description and parameters	
SetOpMode	Set the operating mode.	
	opMode	Operating mode. Valid values are: `AP` (access point), `STA` (station), `ADHOC` (ad-hoc mode), `MONITOR` (monitoring mode)
SetBssID	Set the ID of BSS (basic service set defined in 802.11).	
	bssID	BSS ID of the network to join
SetChannel	Set the 802.11 channel.	
	channel	New Channel
	phyMode	Physical mode of the new channel, eg. 11A, 11B, 11G.
SetPreamble	Set the preamble format.	
	shortPreamble	0(default): long preamble, 1: short preamble
SetDevTxPower	Set the per-device transmit power limit.	
	devTxPower	The maximum transmit power level of this device.
SetAntenna	Select the antenna.	
	antSelect	Selection of the antenna being used.
SetEdcaParam	Set the IEEE 802.11e EDCA(enhanced distributed channel access) parameters of an access category.	
	ac	Access category
	aifsn	AIFS in slottimes.
	logCwMin	Minimum congestion window size.
	logCwMax	Maximum congestion window size.
	txOpLimit	Limit of transmit opportunities.
	noAckPolicy	True if no ack is used for this access category
SetRtsThreshold	Set the minimum threshold of the frame size to enable RTS/CTS.	
	rtsThreshold	RTS threshold.
SetFragThreshold	Set the minimum threshold of the frame size to enable fragmentation.	
	fragThreshold	Fragmentation threshold.

Table 4.2: Per-device parameters in the wireless extension interface.

Command	Description and parameters	
SetNodeTxParam	Set the transmit parameters of a neighbor node.	
	node	The neighbor node
	txPower	Transmit power level.
	rate	Rate to transmit this packet.
	retries	Limit of the MAC-level retransmission
	priority	WME access category of the packet

Table 4.3: Per-neighbor parameters in the wireless extension interface.

Per-Packet Transmit Parameters

Table 4.4 lists the parameters that control the transmission of a frame. Each request includes an input argument packetPtr that identifies the packet of which the properties are being set. One can, for example, implements a specific power control algorithm at the per-packet granularity by setting the txPower parameter. Note that many of the per-packet transmit parameters can also be set in a per-neighbor or per-interface setting. In such case, settings at the finer granularity take precedence. In other words, for the same parameter, per-packet setting overrides per-neighbor or per-interface setting.

Command	Description and parameters	
SetPktTxParam	Set the transmit parameters of a packet.	
	pktPtr	Pointer to the packet
	txPower	Transmit power level.
	rate	Rate to transmit this packet.
	retries	Limit of the MAC-level retransmission
	priority	WME access category of the packet
	antenna	Antenna to use for transmission

Table 4.4: Per-packet parameters in the wireless extension interface.

4.4.2 PHY/MAC Events Exported by the Wireless Extension Interface

In this section, we present the PHY/MAC events supported by the wireless extension interface. An event is specified by a name, the scenario when it is generated, and the event data that comes with it. We categorize all the events into three groups: *receiving events*, *transmission events*, and *MAC-associated events*.

Reception Events

Table 4.5 gives events that are related to, and triggered upon, reception of a frame.

Event Name	Event description and data fields	
FrameRxOk	A frame is received successfully by the interface.	
	recvFrame	The received frame.
	senderNode	Sender of this frame.
	recvStatus	Reception status. Reception status includes rate, signal strength, and timestamp of the received frame.
FrameRxError	A reception error has occurred and its cause is indicated by the error code.	
	errorCode	Error code of the receiving error.

Table 4.5: Reception events in the wireless extension interface.

Transmission Events

Table 4.6 gives events that are generated when a frame is to be transmitted. Note that the transmit query event is a *synchronous* event (Section 4.2). Through this event, a cross-layer control module can be inquired, prior to the time of transmitting a frame, to provide transmit parameters (as listed in section 4.4.1) on a per-packet basis.

Event Name	Event description and data fields
TxQuery (synchronous)	This event is generated right before a frame is to be transmitted. The transmit descriptor of the frame is to be filled. framePtr Pointer to the frame being transmitted
TxInterrupt	An interrupt is generated when a marked frame is transmitted. markedFrame The marked frame.
TxStatus	The event is generated after a frame transmission attempt. It carries the status of the transmission, including the retry counts, the rate used, the virtual collision count and the hardware-assigned timestamp. framePtr Pointer to the transmitted frame. txStatus Result of the transmission.

Table 4.6: Transmission events in the wireless extension interface.

MAC-Layer Events

Table 4.7 gives events that are related to the IEEE 802.11 MAC protocol.

Event name	Event description and data fields
MgmtFrame	The event is generated when a 802.11-related management frame is received. frameType Type of the received frame. frame The received frame.
FragmentReq	The event is generated before a frame is fragmented at the MAC layer. outgoingFrame The frame to be fragmented.
StateChanged	The event is generated when the MAC layer changes its state. There are 5 states defined: initial, scanning, authenticating, associating, and associated. oldState Previous state. newState New state.
NodeJoin	The event is generated when a new node joins the network. A corresponding entry is created in the node table. node Pointer to the node structure in the node table.
NodeLeave	The event is generated when a node leaves the network. The corresponding entry in the node table is invalidated. node Pointer to the node structure in the node table.

Table 4.7: MAC-Layer events in the wireless extension interface.

Chapter 5

Performance Evaluation

5.1 Examples of Cross-Layer Control Modules

With the availability of the unified cross-layer manager and the wireless extension interface, numerous cross-layer design/optimization applications can be devised, ranging from power control, interference mitigation, channel assignment, and scheduling-based MAC (rather than contention-based MAC) both on a per-connection or per-packet basis. In this section, we give illustrative examples of several cross-layer control modules and demonstrate how they can be implemented using the proposed framework.

5.1.1 Rate and Power Control Algorithms

Topology control and management – how to determine the transmit power of each node so as to mitigate interference, improve spatial reuse, while consuming the minimum possible power – is one of the most important issues in wireless multi-hop networks. Several rate and power control algorithms have been proposed, and the principle they employ is essentially to decide the transmit rate and power level dynamically based on several PHY/MAC parameters: the frame error rate, the frame loss statistics on the wireless link, and/or the estimate of interference level. To demonstrate the utility of the proposed framework, we implement a few different rate adaptation schemes. Listed in Figure 5.1 is an adaptation algorithm that adapts transmit power in a way similar to how rate is adapted in the Auto Rate Fallback (ARF) algorithm[28]. Another example is Power-controlled Auto Rate Fallback (PARF) algorithm proposed in [29]. Figure 5.2 outlines how PARF can be implemented using UCF.

The above algorithms can be readily implemented as a cross-layer control module in our framework. They listen to the TxStatus event, update the number of of successful transmissions, and call SetNodeTxParam to set the transmit power txPower on a per-neighbor basis. We have implemented these algorithms and evaluate their performance in Section 5.3.3.

```
PowerAdaptation
On L consecutive packets losses :
   txPower ← min ( txPower + step, maxTxPower)
On S consecutive successful transmissions:
   txPower ← max ( txPower - step, minSafeTxPower)
On timeout for being idle longer than IDLE_TIME:
   txPower ← maxTxPower
```

Figure 5.1: Pseudo-code of power adaptation.

```
PowerControlledAutoRateFallback
On L consecutive packets losses :
   rate ← Lower(rate)
   if IsLowestRate(rate)
       txPower ← min ( txPower + step, maxTxPower)
On S consecutive successful transmissions:
   rate ← Higher(rate)
   if IsHighestRate(rate)
       txPower ← max ( txPower - step, minTxPower)
On timeout for being idle longer than IDLE_TIME:
   rate ← HighestRate()
```

Figure 5.2: Pseudo-code of the PARF power and bit-rate adaptation algorithm.

5.1.2 Scheduling-based MAC

Another dimension of improving the network capacity is through *joint temporal and spatial diversity.* Specifically, the overall capacity can be increased by exploiting spatial diversity that exists among a number of multi-hop paths. Packets that are routed along these paths can be *scheduled* to take place simultaneously if their transmissions do not interfere with each other significantly.

To realize the notion of joint temporal and spatial diversity, a device driver has to support scheduling-based medium access (rather than contention-based medium access). A scheduler should also be able to gain access to PHY/MAC attributes and schedule packets according to the channel status and packet delay constraints. To enable a scheduler to transmit packets at deterministic time instants, we can leverage the features provided by the wireless extension interface:

- Disable exponential backoff mechanism by setting both logCwMin and logCwMax to 0. In this manner, all the frames will be transmitted once they arrive at the transmit queue of the firmware/hardware.
- Disable MAC-layer retransmission by setting retries=0 in the per-neighbor/per-packet transmit parameter.

In addition, to disable link-level ACKs, we can leverage the SoftMAC methodology [24] and "undo" several

802.11 functions implemented in the Atheros interface hardware/HAL as follows:

- Instrument via the device configuration parameter. Set the interface to operate in the *monitor* mode (in which all the messages, including those meeting the 802.11 PHY format but not necessarily the 802.11 MAC message types, will be received).
- Exploit a IEEE 802.11 MAC feature that it does not acknowledge multicast frames to eliminate automatic ACKs in the case of unicast. That is, the *multicast* bit in the destination address of a unicast frame is set to lure the hardware/HAL into believing that the destination address is not a valid unicast address. (The frame will still be received because it meets the 802.11 PHY format.)

Next, to enable the scheduler to receive packets from an upper-layer protocol module and to schedule packets based on PHY/MAC characteristics, the following events can be used to notify the scheduler:

- Deliver request: A scheduler can be notified by this event of a packet sent from an upper layer protocol module for delivery. A packet may be associated with a tag that specifies its scheduling deadline, i.e., the relative time by which the packet has to be transmitted.
- Timer event: A scheduler can invoke itself periodically by listening to timer events. These timers can be used to schedule packets according to their (relative) transmit delay constraints.
- Transmission feedback: A scheduler is notified by event TxStatus of whether or not a packet transmission is successful.
- Dequeue request: A scheduler is notified by this event when the hardware has completed the previous transmission and is ready to accept the next packet(s).

The pseudo code in Figure 5.3 elaborates how a scheduling-based MAC can be implemented using UCF.

5.2 Implementation

We implement UCF on a development branch of NetBSD 4.0. The whole UCF framework is implemented as a loadable kernel module (LKM) which can be loaded or unloaded dynamically. We also modify the Atheros driver extensively to support the 802.11 MAC/PHY parameters and events mentioned in Section 4.4. Our customized NetBSD distribution is based on CUWiN 0.7.0[1]. The customized NetBSD distribution is deployed to a 20-node testbed consisting of Soekris Net4526 (with 64MB RAM and 64MB flash) and Soekris Net4801 (128MB RAM and 256MB flash) embedded systems. Each wireless node is equipped with two Wistron CM9 MiniPCI cards.

```
DeterministicMACScheduler
On deliveryRequest(packet, constraints):
   if constraints can be satisfied:
      enqueue packet with priority inferred by constraints
      schedule timer event(s) if necessary
   else
      reject the delivery request

On timer event:
   packet ← first packet in the queue
   put packet into hardware transmission queue for transfer

On transmission feedback event:
   if transmission error occurs
      if delivery constraint is still satisfiable
         reschedule packet delivery
      else
         notify failure of packet delivery

On dequeueRequest event:
   packet ← first packet in the queue
   put packet into hardware transmission queue for transfer
```

Figure 5.3: Pseudo code of a deterministic MAC scheduler using UCF.

5.3 Performance Evaluation

In this section we evaluate UCF by measuring its performance and showcase a few cross-layer modules implemented in UCF.

5.3.1 Micro-Benchmarks

To understand how much overhead is introduced by the event notification mechanism of UCF, we benchmark the packet transmission function of Atheros driver with cross-layer modules subscribing to different events. For each packet transmission, we breakdown the processing task in device driver into 3 stages: (i) pre-DMA (direct memory access) processing, such as filling hardware-dependent frame descriptor and (ii) setup of DMA descriptors and enqueuing to hardware transmission queue, and (iii) cleanup of DMA descriptors. In this experiment, three cross-layer modules subscribe to no event, TxStatus, and both TxStatus and TxQuery events, respectively. The results are shown in Figure 5.4. It can be seen that TxStatus event incurs more overhead in the DMA cleanup stage, while TxQuery event incurs more overhead in the DMA setup and hardware queue enqueuing stage.

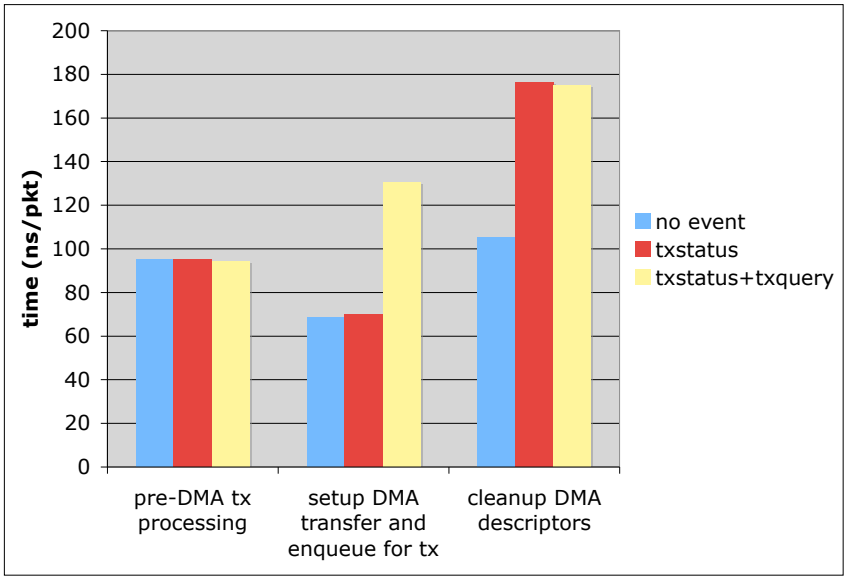


Figure 5.4: Breakdowns of the processing time of packet transmission tasks with different event subscriptions.

5.3.2 Synchronous v.s. Asynchronous Events

In this experiment, we run ping tests on two nodes to study the effect of different event delivery methods (synchronous or asynchronous) on system performance. The receiver node runs a cross-layer module which simulates a computationally intensive procedure by running a delay loop to delay a fixed amount of time (5ms). The cross-layer module subscribes to the "frame-received" (Table 4.5) event using either synchronous mode or asynchronous mode. We vary the intervals between two consecutive ping request packets on the sender and measure the packet loss rate and average round-trip time. The results are shown in Figure 5.5 and Figure 5.6.

In Figure 5.5 we see that in synchronous mode, when the ping interval approaches the processing delay of one packet (5ms), CPU is locked up in the interrupt handler most of time and cannot process interrupts fast enough, therefore causing packet losses when the hardware receiving buffer is overflowed. On the other hand in asynchronous mode, incoming packets are still processed and delivered to the network stack to return the ping reply packets even though the system spends most of time running the time-consuming cross-layer module. This is because the hardware interrupt handler and network stack handlers have higher priority than the cross-layer module. The results confirm that the ad-hoc way of implementing cross-layer module can incur unnecessary performance degradation. In contrast, UCF handles event triggering and delivery in the correct context (either interrupt or process), providing more flexibility and better interface for protocol

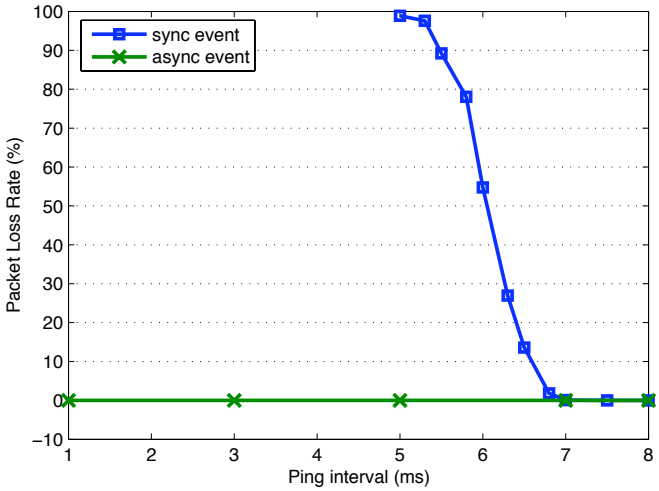


Figure 5.5: Packet loss rate of ping tests with different ping intervals using either synchronous or asynchronous event.

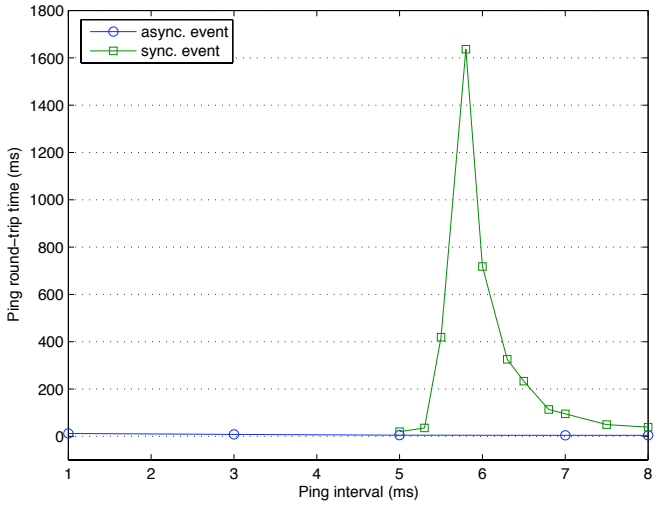


Figure 5.6: Round-trip time of ping tests with different ping intervals using either synchronous or asynchronous event.

designers to design and implement cross-layer schemes.

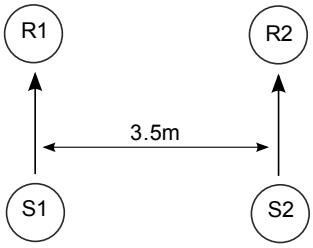
5.3.3 Rate and Power Control Schemes

In this section we show the performance evaluation results of the three cross-layer schemes described in Section 5.1.1. The topology contains two pairs of sender and receiver nodes and is shown in Figure 5.7(a). The sender and receiver of each pair are placed close to each other while the two pairs are separated by 3.5m. To show the influence of different rates and power levels within such a small area, we unplug the antennas of the nodes and only put antenna cables on them. In each run of the experiment, both senders send data to their receivers simultaneously for 10 seconds. We measure the aggregated UDP throughput of the two pairs. The offered load is 11Mbps which saturates the link (11Mbps for 802.11b). We repeat the experiments three times for every rate/power control scheme and present the average results.

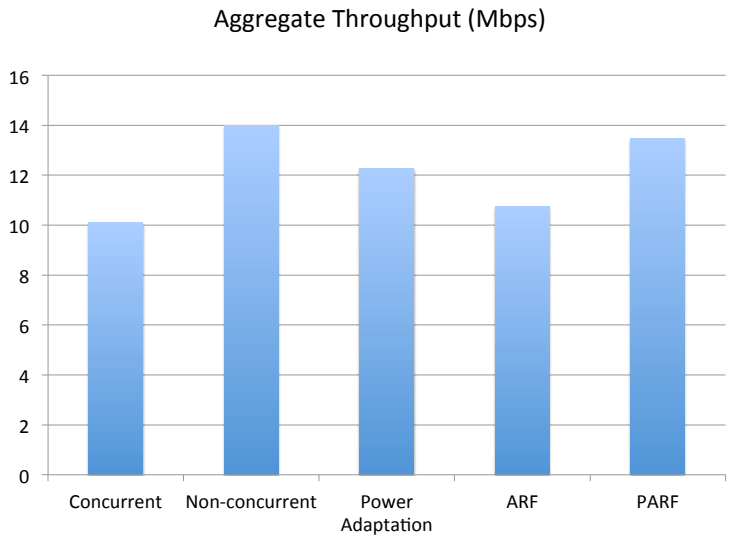
Since the two pairs are within each other's interference range under the maximum power control, there will be poor spatial reuse by using the maximum power. By applying power control, we can improve spatial reuse and boost the aggregated throughput. Also, since the channel condition is time-varying, the best data rate to use should also be time-varying. Figure 5.7(b) justifies these intuitions. From left to right, each bar shows the aggregate throughput of the following setups:

- Concurrent: aggregate throughput when each sender uses fixed data rate (11Mbps) and default transmit power.
- Non-concurrent: sum of the individual throughputs of the two flows using fixed data rate (11Mbps) and is therefore the maximum achievable throughput.
- Power Adaptation as described earlier in Figure 5.1 with minTxPower=0, maxTxPower=60, and step=10.
- ARF: each node use Auto Rate Fallback (ARF) [28]. Bit-rates are in the set of $\{1, 2, 5.5, 11\} Mbps$.
- PARF: each node use PARF as described in Figure 5.2 to adapt both transmit power and bit-rate (in the same range and set as Power Adaptation and ARF).

We find that both the rate control module and the power control module achieve better throughput than the default (fixed) scheme. Further, the joint rate/power control module can offer even higher throughput since it improves both spatial reuse and robustness against temporary bad channel conditions.



(a) Two-flow topology used in the experiment



(b) Aggregate throughputs of rate-control, power-control, and joint rate and power control algorithm

Figure 5.7: The topology and aggregate throughputs of three rate and power control schemes.

Chapter 6

Conclusions

In this work we present UCF as a generic framework to facilitate messaging between cross-layer components. UCF lightens the burden of protocol designers by hiding the details of low-level OS programming. The generic API interface of UCF enables protocols to be cross-layer aware, yet still maintains modularity and separation between cross-layer components. Possible future work includes using UCF to implement cross-layer aware TCP and cross-layer aware routing protocols. Another avenue for future work is to port UCF to Linux and other wireless drivers and make UCF available for more system developers and network researchers.

References

[1] "Champaign-urbana community wireless network," http://www.cuwireless.net/.

[2] "Seattle wireless," http://www.seattlewireless.net/.

[3] "Mit roofnet," http://pdos.csail.mit.edu/roofnet/doku.php.

[4] Z. Fu, H. Luo, P. Zerfos, S. Lu, L. Zhang, and M. Gerla, "The impact of multihop wireless channel on tcp performance," *IEEE Trans. on Mobile Computing*, vol. 4, no. 2, pp. 209–221, March/April 2005.

[5] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris, "Link-level measurements from an 802.11b mesh network," in *Proc. of ACM SIGCOMM*. ACM, September 2004.

[6] J. Bicket, D. Aguayo, S. Biswas, and R. Morris, "Architecture and evaluation of an 802.11b mesh network," in *Proc. of ACM Mobicom*. ACM, September 2005.

[7] K. Sanzgiri, I. D. Chakeres, and E. M. Belding-Royer, "Determining intra-flow contention along multihop paths in wireless networks," in *Proc. of Broadnets Wireless Networking Symposium*, October 2004.

[8] D. Berger, Z. Ye, P. Sinha, S. Krishnamurthy, M. Faloutsos, and S. K. Tripathi, "TCP-friendly medium access control for ad-hoc wireless networks: alleviating self-contention," in *Proc. of IEEE MASS*, October 2004.

[9] R. Draves, J. Padhye, and B. Zill, "Routing in multi-radio, multi-hop wireless mesh networks," in *Proc. ACM MobiCom 2004*. ACM, September 2004.

[10] V. Kawadia and P. R. Kumar, "A cautionary perspective on cross layer design," *IEEE Wireless Communication Magazine*, vol. 12, no. 1, February 2005.

[11] X. Liu, E. K. P. Chong, and N. B. Shroff, "Opportunistic scheduling: An illustration of cross-layer design," *Telecommunications Review*, December 2004.

[12] X. Lin and N. B. Shroff, "The impact of imperfect scheduling on cross-layer congestion control in wireless networks," *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 302–315, February 2006.

[13] X. Lin, N. B. Shroff, and R. Srikant, "A tutorial on cross-layer optimization in wireless networks," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 8, pp. 1452–1463, June 2006.

[14] X. Lin and N. B. Shroff, "Cross-layer design for multi-hop wireless networks: a loose coupling perspective," in *Proc. of IEEE INFOCOM*. IEEE Computer Soceity, March 2005, (invited for fast track publication to *IEEE/ACM Trans. on Networking*.

[15] F. Fu and M. van der Schaar, "A new systematic framework for autonomous cross-layer optimization," *Vehicular Technology, IEEE Transactions on*, vol. 58, no. 4, pp. 1887–1903, Mar. 2009.

[16] G. Holland, N. Vaidya, and P. Bahl, "A rate-adaptive MAC protocol for multi-Hop wireless networks," in *Proceedings of the 7th annual international conference on Mobile computing and networking - MobiCom '01*. New York, New York, USA: ACM Press, 2001, pp. 236–251. [Online]. Available: http://portal.acm.org/citation.cfm?doid=381677.381700

[17] S. H. Y. Wong, S. Lu, H. Yang, and V. Bharghavan, "Robust rate adaptation for 802.11 wireless networks," in *Proceedings of the 12th annual international conference on Mobile computing and networking - MobiCom '06*, vol. pp. New York, New York, USA: ACM Press, 2006, p. 146.

[18] M. Vutukuru, H. Balakrishnan, and K. Jamieson, "Cross-layer wireless bit rate adaptation," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 3–14, Aug. 2009.

[19] M. Lacage, M. Manshaei, and T. Turletti, "IEEE 802.11 rate adaptation: a practical approach," in *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*. ACM, 2004, pp. 126–134.

[20] J. C. Bicket, "Bit-rate selection in wireless networks," Master's thesis, MIT, 2005.

[21] P. Mochel, "The sysfs Filesystem," *Proceedings of the Linux Symposium*, vol. 1, 2005.

[22] The official linux wireless wiki. [Online]. Available: http://linuxwireless.org/

[23] *IEEE 802.11 Standard - Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE Computer Soceity, 2007.

[24] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald, "SoftMAC Flexible Wireless Research Platform 1 Introduction 2 Implementation and Design of the," in *Hotnets*, 2005.

[25] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[26] M. Abu-Rgheff, "Cross-layer signalling for next-generation wireless systems," *2003 IEEE Wireless Communications and Networking, 2003. WCNC 2003.*, pp. 1084–1089.

[27] N. Baldo, F. Maguolo, M. Miozzo, M. Rossi, and M. Zorzi, "ns2-miracle: a modular framework for multi-technology and cross-layer support in network simulator 2," in *ValueTools '07*. ICST, Brussels, Belgium, Belgium: ICST, 2007, pp. 1–8.

[28] A. Kamerman and L. Monteban, "Wavelan-ii: a high-performance wireless lan for the unlicensed band," in *Bell Labs Technical Journal*, August 1997, pp. 118–133.

[29] A. Akella, G. Judd, S. Seshan, and P. Steenkiste, "Self-management in chaotic wireless deployments," in *MobiCom '05: Proceedings of the 11th annual international conference on Mobile computing and networking*. New York, NY, USA: ACM, 2005, pp. 185–199.