Design, Implementation and Evaluation of a Multihop WiFi-based TDMA System

A Thesis Submitted

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF TECHNOLOGY

by

Nirav Uchat

06305906

under the guidance of

Prof. Bhaskaran Raman

and

Prof. Kameswari Chebrolu

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay

June 2009

Abstract

WiFi mesh networks with long distance outdoor links have become an attractive option to provide lowcost network connectivity to rural areas, especially in developing regions. Various studies shows severe drop in 802.11 performance on long distance links;primary reason being failure of carrier sensing. It is also well understood that a TDMA-based approach is necessary to provide good performance over such networks. While preliminary prototypes of TDMA-based MAC protocols have been developed, there is no implementation-based validation/evaluation in multi-hop settings.

In this work we present the implementation and evaluation of WiFi-based multihop TDMA system using off-the-shelf hardware and open-source drivers. To our knowledge there is no working implementation of such system complete with multihop schedule dissemination and time synchronization. In our implementation, we carefully account for the various overheads such as the synchronization error, guard time, header overheads, etc. Through detailed evaluation we show that the achieved throughput is close to what we expect theoretically. And the delay/jitter values are small even over multiple hops and good enough to support real-time voice and video-conferencing applications. We think that such a setup could provide avenues for e-learning through video conferencing, low cost telephony and internet access desirable for rural areas.

Acknowledgments

I would like to thank my faculty mentors **Prof. Bhaskaran Raman** and **Prof. Kameswari Chebrolu** for their constant guidance and support during the course of my research work. The excellent insights provided by them from time to time and the constant motivation have been pivotal in the completion of my thesis.

I would also like to thank my thesis partner **Ashutosh Dhekne** for helping me in understanding madwifi device driver. I remember, we used to had discussion on thesis status in lab, in hostel, on mess table and even while riding bicycle. For me, working with him was one of the best experience I had till now.

Further, I would like to thank **Synerg Group** and **Computer Science Department at IIT Bombay** for providing excellent working environment and giving 24x7 lab access. I would also like to thank **Computer Center at IIT Bombay** for giving me opportunity to work as Research Assistant.

Finally, I take this opportunity to thank my parents and my brother and sister, who have always been there for support and inspiration.

Nirav Uchat

CSE, IIT Bombay

24^{th} June 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Gone are the days when connectivity to the Internet was considered to be a luxury reserved for the urban citizens of a country. Governments around the world have realized the importance of providing outside connectivity to the rural population with the aim of providing better living conditions, brighter livelihood and better knowledge of the world. Consequently, providing low cost, easy to deploy wireless solutions to rural areas has emerged as an important research area and is gaining attention form governments and researchers alike. But what is so different about a rural setting that needs special attention? Can't we directly use the solutions that have already been proposed and deployed in urban areas? Why can't a 802.11b/g wireless network which is widely used in offices and colleges be used in rural areas also? Well, unfortunately, there are several reasons which make these solutions unsuitable for the problem at hand - which is providing Internet connectivity to rural areas without incurring large infrastructural costs.

Firstly, the standard 802.11 protocol gives satisfactory performance only over a small distance of 50-100 metres. However, it is important to note that to reduce the cost of initial setup it is essential that the wireless communication takes place over a larger distance (say through an antenna at a a distance of 20-25 km from the village - so that the same antenna can cater to several villages in the vicinity). This requirement of effective communication over such long distances makes 802.11 unsuitable as there are several studies [9] which show

that there is a severe drop in the performance of 802.11 over such long distances. The primary reason for this drop in performance is the failure of carrier sensing on long distance link. Another alternative would be to use WiMAX [6]. But this again beats the purpose of having a low cost solution as WiMAX uses a licensed band and it requires large investment in deploying base station hardware. Other solutions like laying fiber optic lines to these areas are obviously costly and further considering the low user base in a specific rural area, an ISP provider would definitely be reluctant to provide such a service.

The above discussion points at a clear and present need for finding ways of communication using unlicensed frequency spectrum and inexpensive off-the-shelf hardware. It has been observed that using sectorized or directional antennas, it is possible for WiFi signals to travel few tens of kilometers. In this report, we demonstrate a TDMA implementation that can be used over such long distance multihop links using off-the-shelf inexpensive hardware. To the best of our knowledge, there is **no working implementation of a multihop TDMA system complete with synchronization and centralized schedule dissemination**. By testing it over links in an indoor setup we show that the throughput achieved is close to what we expect theoretically. During experimentation, we observed UDP throughput for 4 hop linear topology with 2ms slot size to be 6.93 Mbps, while theoretical maximum for the same setting is 7.16 Mbps. We also observed that he delay/jitter values are of the order of few milliseconds and are considerably small for multiple hops setting and good enough to support voice and video conferencing applications and streaming media. During testing we also played video and made voice call between two PCs connected through linear topology. Such a setup could provide avenues for e-learning through video conferencing, low cost telephony and internet access desirable for rural areas particularly in developing nations.

In the rest of the report, we first discuss other similar efforts and contrast our work with what has been already done. We will then discuss the approach we are taking and give a detailed account of our implementation in chapter 2 and chapter 3 respectively. In chapter 4, we present our experimental results and show that the throughput obtained is close to what can be predicted by theoretical calculations. Finally, in chapter 5, we conclude with a discussion on applicability and future work that can be done in this field.

1.1 Problem Statement

Motivated by the necessity of providing low cost internet connectivity to rural areas we aim to design, implement and evaluate a multihop TDMA system using inexpensive off-the-shelf WiFi hardware and open-source drivers. The proposed system will have multihop schedule and data dissemination mechanism. The system should ensure QoS guarantees and should be capable of handling real-time audio and video traffic.

1.2 Related Work and Our Approach

There has been considerable effort in the area of software configurable radio using open source drivers which facilitates implementing various protocols over inexpensive WiFi hardware. We present some of these which have demonstrated TDMA implementations. However, none have discussed a multihop implementation.

SoftMAC [8] provides a generic software-defined radio to experiment with MAC protocols. It disables RTS/CTS, MAC level hardware acknowledgements and facilitates custom frame header formats by setting the card in monitor mode. To demonstrate the utility of the platform, the authors have implemented a TDMA system between two nodes. The insights about disabling certain aspects of CSMA are useful for our implementation. In addition, we also need very precise timing control for a multihop TDMA protocol.

The authors of MadMAC [13] also implement an example TDMA system between two machines with slot sizes of 20ms - 60ms and guard bands of 4ms - 12ms. However, since we envision a multihop system, increased slot size has a detrimental effect on the achievable TCP throughput. We have used slot sizes and guard bands much smaller than those proposed in MadMAC and still maintained tight synchronization between nodes.

Building over SoftMAC, FreeMAC [12] provides a generic platform exposing many more configurable parameters. It also demonstrates a TDMA system, however it synchronizes nodes using out of channel Ethernet links. It also implements channel switching in the TDMA system but does not implement multiple hop communication. FreeMAC uses the

hardware beacon timer and indicates that the timer works well under both low load and heavy load conditions. However, we found that the hardware timer is very sloppy with an increased number of RX interrupts. FreeMAC gives insights into various aspects of MadWifi, including the existence of hardware timer and has served as a starting point for our work.

Overlay MAC [11] uses the Click Router system and implements a configurable module between the MAC layer and the network layer. However, it does not have precise control over packet transmission times and implements a distributed algorithm for allocating slots. The 2P protocol [10] demonstrates a synchronous operation of TX and RX in a bipartite topology. However, this work has been done only for single hop scenarios. WILDNet [9] extends 2P and uses Click Router and loose time synchronization. However, WILDNet provides justifications for poor performance of 802.11 on long distance links and motivates the use of TDMA systems for this purpose.

We build upon the idea proposed in [7], which suggests implementing TDMA MAC in mesh network consisting of both long as well as short distance wireless links for the problem under consideration. Our work involves synchronization of nodes in a tree topology along with schedule and data transfer over a multihop network. Whereas most previous work demonstrate their TDMA systems through a single hop, we support TDMA over many more hops. Table 1.1 gives comparison between previous work and our approach.

Table 1.1 Comparison of our approach with previous related work

	Related Work						Our Approach
	SoftMAC	MadMAC	FreeMAC	WiLD Net	Overlay	2P	FRACTEL
Implemented at	MAC Layer	MAC Layer	MAC Layer	click router	Above MAC Layer	MAC Layer	MAC Layer
Multi-Channel	No	No	Yes	No	No	No	Yes
Timer type	Software	Software	Hardware	Software	Software if used	Software	Software
Multi-Hop Time Sync.	No	No	No	No	No	No	Yes
Multi-Hop Schedule Diss.	No	No	No	No	No	No	Yes
Multi-Hop TDMA Example	No	No	No	No	No	No	Yes

To summarize, our contribution is the implementation and performance characterization of a multi-hop TDMA-based MAC for WiFi mesh networks on off-the-shelf hardware.

Chapter 2

Design of a Multi-Hop TDMA System

The randomness inherent in CSMA based protocols for wireless access makes it difficult to support QoS for real-time traffic on multihop wireless links. On the contrary, a centralized TDMA protocol has the potential to ensure that all flows allowed in the network can be sustained in terms of QoS guarantees. In this section we describe in detail our TDMA system for multihop communication.

2.1 Terminology

During the design discussion we will come across various terminology. Its better to understand it beforehand to simplify the understanding. The envisioned system consists of one resource rich node named **root node**. It is responsible to create the schedule and routing tree information. All other nodes in the network are termed as **non-root nodes**. All information in the proposed network flows in custom packets[1]. The system has two types of packets, the schedule packet and the data packet.

Schedule Packet: is used for sending TDMA schedule along with synchronization and routing tree information across the network. The schedule packet is logically divided into

[1]there are no 802.11 headers attached to any packets in the network

- *Schedule header:* containing multihop synchronization information along with routing tree length and number of scheduling elements.
- *Scheduling elements:* indicating transmitter, receiver and flow id for a given data slot. Every data slot will have its own scheduling element.
- *Routing tree:* containing parent-child information in a given tree topology.

The information stored in schedule packet is used by all non-root nodes for synchronizing itself with the root node and to know its own parent in the given topology. The routing tree information is sufficient enough to recreate complete topology at each node.

Data Packet: is the actual network layer data that needs to be transmitted across the network. All packets except schedule packets are termed as data packet in our implementation. Data packet is logically divided in to

- *Data header:* which holds all necessary information for a packet to reach its final destination. All data packets will have data header attached to it. Data headers are added/modified/removed at each node depending on the requirement.
- *Payload:* is the network layer data which is received by the MAC layer. The payload is kept intact throughout its lifetime in the network. Only data header is added or removed while processing a packet.

Schedule Header and Synchronization: root node has exclusive rights to create the schedule and send it on air. Along with the schedule, it also sends routing tree information and scheduling elements. All non-root node on receiving such schedule first find its parent by referring to the routing tree information. If the received schedule is indeed from its intended parent, it synchronizes with root node using information present in schedule header.

Multihop Schedule Dissemination: the only way for non-root to send schedule is if it receives one from its parent. All non-root nodes store such schedule. When a transmission

opportunity comes, they modify some information in stored schedule and send it on air. The modified information is used for multihop synchronization purpose.

Control, Contention and Data Slots: at any time, every node in the network will be running in one of these slots. The multihop synchronization mechanism ensures that every node sees exact same slot at any given time. A slot defines a unit of operation. In general term it is the TDMA slot. Depending on type of slot different operations will be carried out by each node. The control slot is used for sending control information such as schedule packet while data slot is used for sending actual data packet.[2]. The contention slots are reserved as of now and will be used for implementing node join operation.

How Schedule and Data packets are generated?: the schedule packets are generated from MAC layer itself to incur minimum packet generation delay, while data packets are those packets which are received from network layer or through air. Note that, we never change any information in received data packet. Only data header is attached or removed while processing a data packet.

Routing: data headers are used for routing packets from one node to another in multihop scenario. Section 3.4.3 explains it in more detail.

TDMA Queuing: all data packets will be buffered in TDMA queue and will be removed[3] when node's transmission opportunity comes.

One might find that things said above gets repeated in later sections. The sole purpose of this section is to get acquainted with the working.

[2]In our implementation packet (including header and payload) is the physical entity which goes on air

[3]number of removed packets will be equal to the number of packets that can be send in a given slot

2.2 System Architecture

We envision the network as shown in Figure 2.1 to be a tree topology with a root node that decides the schedule, permits flows and controls node admission to the network. In our

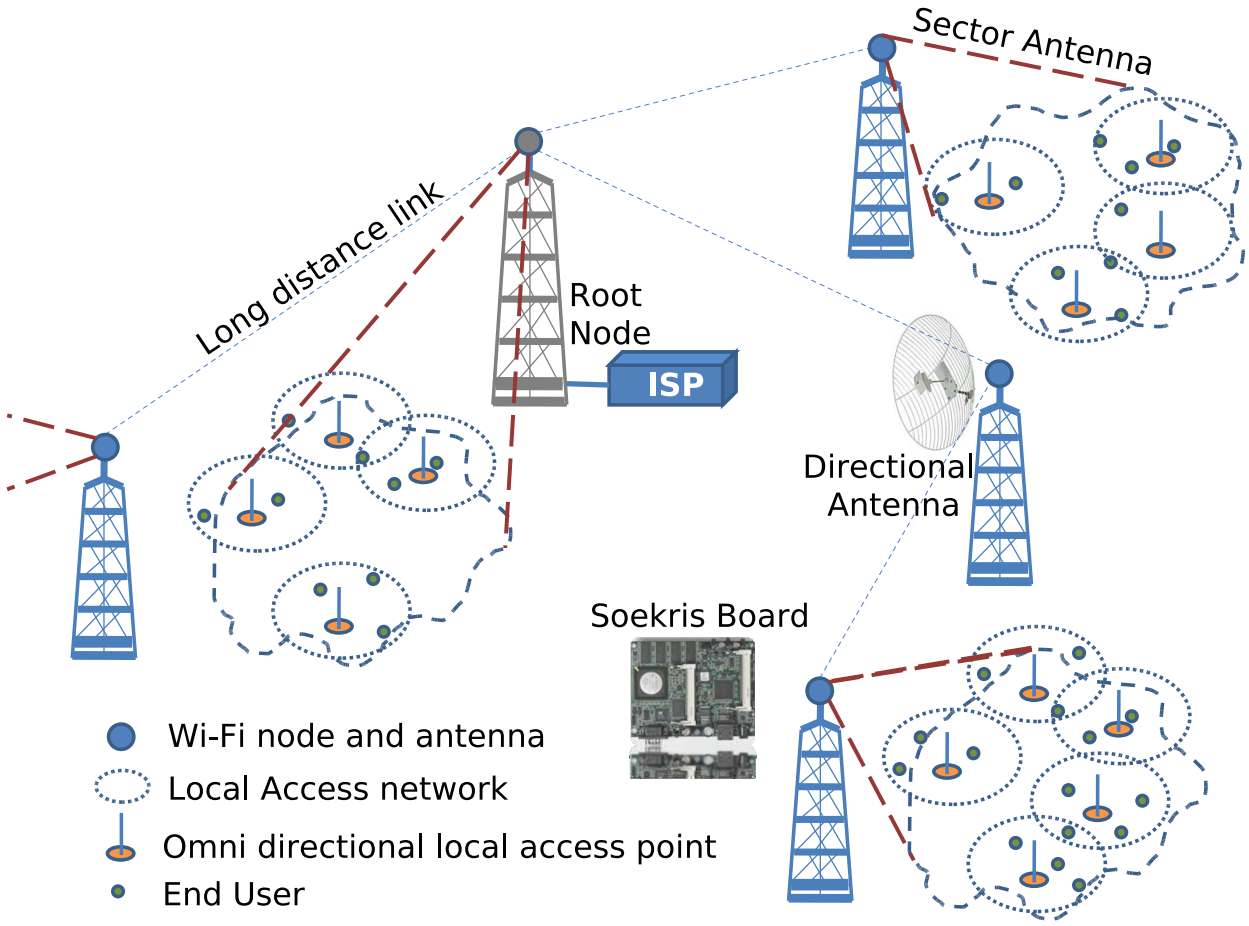


Figure 2.1 FRACTEL Architecture

implementation, the unit of work is a slot[4]. Depending on slot type, different operations are carried out at each node. There are three types of slots in our design.

- **Control slots:** are used to convey information from root node to other node. The schedule packet containing scheduling elements and routing tree information are sent in control slots. The control slots enables us to implement,

[4]In general term, slot is a TDMA slot. A node can send multiple packets on air in a single slot

- multihop schedule dissemination
- sending routing tree information across the network
- multihop time synchronization

- **Contention slots:** are used by non-root nodes to convey information to the root node. The contention slots are not assigned to any specific node. The non-root nodes have to content for them. Objective achieved using contention slots are
  - node join mechanism
  - new flow setup operation

- **Data slots:** are used for actual data flow across the network. Every data packet along with data header and payload are sent in data slots.

A **frame** consists of few control and contention slots and many more data slots. The number of slots in a frame is fixed for a network, but, in general, is a configurable parameter. A frame is repeating pattern of such slots. The frame structure is shown in Figure 2.2.

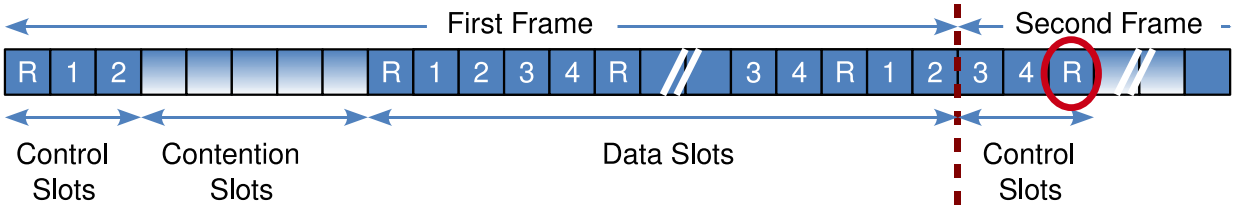


Figure 2.2 Components of a Frame.

The number of control slots in a frame, the frame length and the depth of the network determine the time required to propagate the schedule to all nodes. The number of control slots in each frame are constant, but they are repeatedly numbered from 0 (shown as R in Figure 2.2) to $n-1$ where n is the number of nodes in the network and can span over many frames.

Referring to the example in the Figure 2.2, given three control slots in each frame, and five nodes in the topology, first three nodes send their schedules in the first frame and

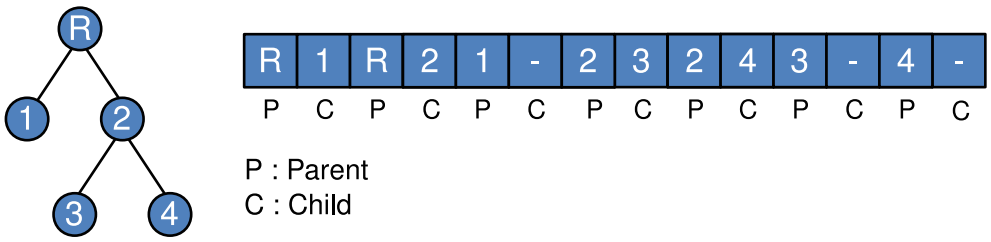


Figure 2.3 Structure of the routing tree communicated through a schedule packet.

the remaining two send their schedules in the consecutive frame. After the second slot in the second frame, the schedule transmission opportunity rotates again to the root node(as marked by circle in Figure 2.2). Each node transmits the schedule packet in a control slot determined by its position in the routing tree. Similarly, the data slots are numbered from 0 to a maximum number given in the schedule.

2.2.1 Schedule, Routing and Data Header

All packets sent on air has a custom header attached to them depending on the type of the packet. Schedule packets are constructed in the driver by the root node. It consists of a schedule header, a (possibly zero) number of scheduling elements and (possibly null) routing tree information. The schedule header has synchronization information and has the number of scheduling elements and routing tree elements contained in the packet. A scheduling element contains the transmitter, receiver and flow_id for a data slot. All scheduling elements together describe the path of all data flows in the network.

The *routing tree* information is simply a parent-child relationship described linearly. Each non-root node must appear at least once as a child node in this tree. A node may be a parent for multiple other nodes. Figure 2.3 shows an example topology and its routing tree. This centralized routing facilitates the root node to keep complete control over the bandwidth usage in the entire network enabling QoS guarantees.

Data packets are attached a data header that help in routing the packets. In addition to the next hop and end-to-end source and destination fields, it also has the flow_id field. This field enables a relay node to keep the received packet under a separate queue for each

flow-destination pair. Implementation of headers are covered in Section 3.2.

2.2.2 Node Join and New Flow Request

Nodes join the network by first listening to the schedule packet, thus getting synchronized with the network, and then requesting the root node to allocate a place in the routing tree. The requests for node join are sent in contention slots. On receiving such request, the root node either allow new node to join the network by placing appropriate parent-child pair in routing tree or reject it. On receiving the routing tree a new node seeing its ID understands that its node join request has been granted. If required, other nodes that are already present in the network may update their routing entries.

A node which wants to start new data flow conveys this information through contention slots. On receiving such request, the root node either create new scheduling element and send it in next schedule or reject the request. The node will only be allowed to sent data packets if there is a scheduling element with transmitter assigned as its own ID.

2.2.3 Multi-hop Schedule Dissemination

The root node periodically publishes the scheduling information along with routing tree in to the network in his control slots. The routing tree is used by all non-root nodes to determine their parent node in the given topology. On receiving any schedule packets, the non-root node first check if the received packet is from its intended parent node, if it is so, it stores such schedule for multi-hop transmission. When non-root nodes control slot comes, it first modify the stored schedules and sends it on air. The modified information are used by other non-root nodes for multi-hop synchronization.

2.2.4 Multi-hop Time Synchronization

To handle clock drift, synchronization is done in every schedule packet using a hardware timestamp. The synchronization information propagates down the tree to all nodes in the network using control slots. Each packet contains its own offset from the global time and

the exact global time of the beginning of this slot. Together, these three entities enable the receiving node to synchronize to the current global time and also calculate the next slot time. The multi-hop schedule dissemination and time synchronization are both achieved using schedule packets.

2.3 Packet Flow

The root node as well as every non-root node has a different flow for schedule packets. The non-root node stores the received schedule from its parent for multihop transmission while root node creates new schedule every time. The data packet has identical path in both type of nodes. The Figure 2.4 explains the packet flow at both root and non-root node.

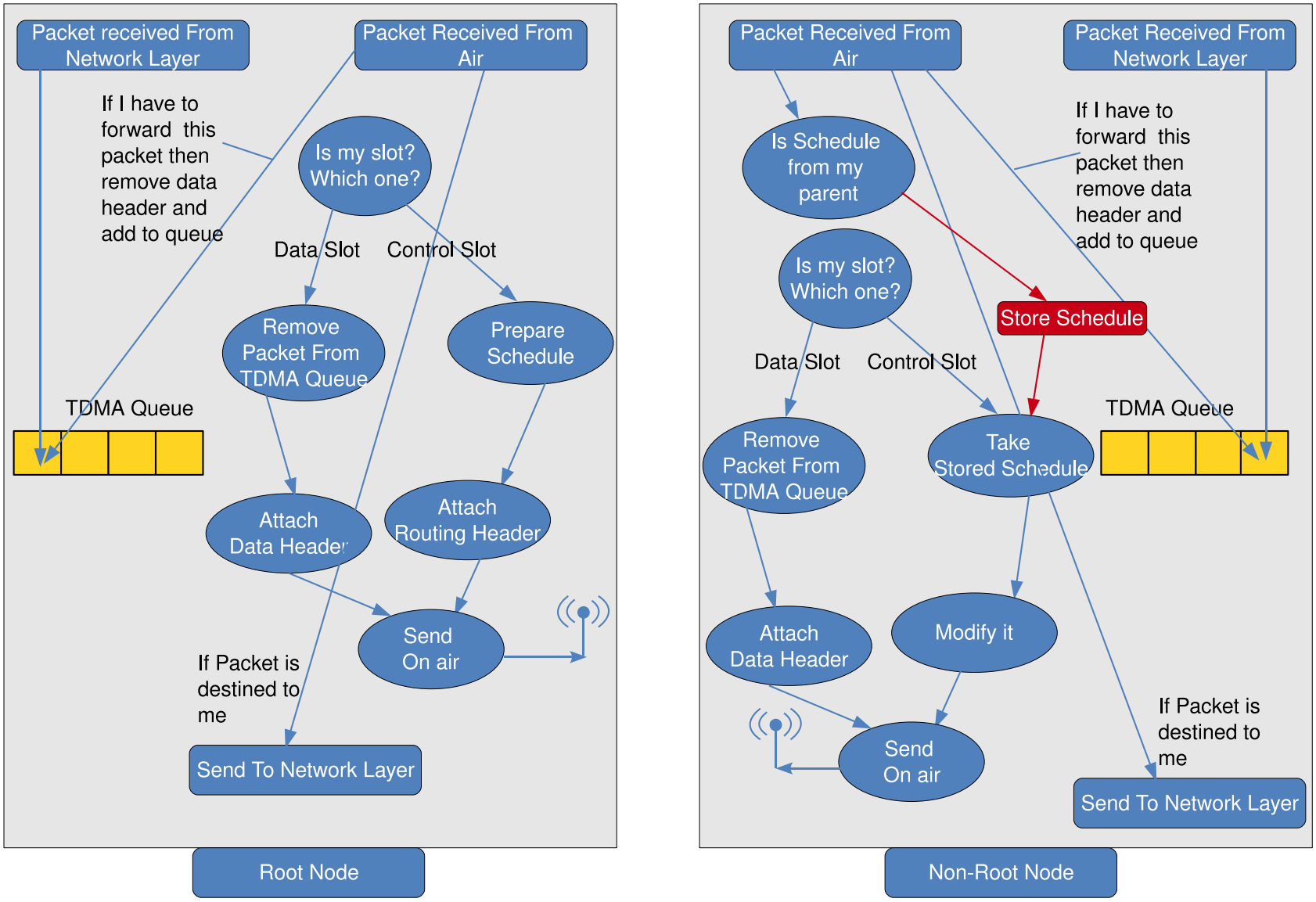


Figure 2.4 Flow of packets in multihop TDMA system

Chapter 3

Implementation

Implementing a multihop TDMA system at MAC layer requires extensive modification to wireless driver. In our implementation, we have used Mikrotik router board RB433AH [2] [3] with Ubiquity SR5 mini-pci wireless cards. The Wireless card has Atheros AR5212 chipset and is fully compatible with madwifi driver [5]. The RB433AH comes with proprietary Routerboard OS and wireless kernel modules. It has Atheros AR71xx family CPU with clock speed of 680MHz. For our implementation, we have removed Routerboard OS and reflashed RB433AH with OpenWRT Kamikaze 8.09 [4].

The OpenWRT Kamikaze 8.09 is a stripped down version of linux and has been ported to several different hardware platform, including Atheros AR71XX. OpenWRT is no different from any other linux distribution except it has very low memory footprint. The complete kernel takes 6 Megabytes of memory when loaded. With such streamlined kernel and powerful 680MHz CPU, the RB433AH has great potential to be used as off-the-shelf embedded wireless system. Naive user might find it difficult to install OpenWRT on RB433AH. It requires basic knowledge of kernel compilation and installing it through TFTP over network onto target system(RB433AH). For detail installation instruction refer appendix A.

This chapter is primarily focused on implementation aspect of our work. We start with set of requirements for implementing multihop TDMA system. We will look at working of madwifi device driver and how packets are processed in it. We will then discuss how we have

modified madwifi driver to suite our needs. Specifically, we will discuss,

- reasons for choosing monitor mode for implementation
- how do we enable two way communication in monitor mode
- how do we generate custom packets from MAC layer
- how do we timestamp a schedule packet for enabling multihop time synchronization

We will also talk about various custom headers and how they are attached/detached to/from data and schedule packets. We will then shift our attention to understand core modules of our implementation. Along with these modules, we will look at few helper functions that are needed for implementing multihop TDMA system.

3.1 Madwifi Device Driver

Madwifi is an open source wireless device driver for Atheros chipset. It has support for Station, AP, Adhoc, Ahdemo and Monitor modes. All operating modes has different functionality. Like, when we set wireless card in monitor mode, the device driver allows us to receive all frames seen on air. Whereas Adhoc, Station, AP and Ahdemo modes are used for normal communication. The madwifi device driver implements CSMA as MAC layer protocol while in our TDMA implementation we need a raw packet transmitter and receiver, with strict control over packet transmission timing. Given such requirements, we first need to disable CSMA and then proceed with implementing multihop TDMA system. In [8], the author has list out six tasks in order to disable CSMA in madwifi device driver and they are as follows,

1. Disable MAC level ACK's
2. Disable RTS/CTS
3. Sending custom frame format (no 802.11 frame)

4. Disable transmission backoff

5. Disable virtual carrier sensing

6. Disable CCA mechanism

Given the task at hand, we found that AP and Adhoc mode were not useful, since they send periodic beacons and follows 802.11 state machine. On other hand, Ahdemo mode does fit in for our requirements, but we found it unstable. We also found that, when we set card in monitor mode, it disables MAC layer ACK, RTS/CTS mechanism and allows us to send custom frames on air which directly acheives first three task in process of disabling CSMA mechanism. Given the kind of flexibility, we decided to use **monitor mode** for our implementation. But the problem was, one cannot communicate between two devices running in monitor mode. That is to say, we cannot ping one machine from another running in monitor mode. The monitor mode functionality is coded in such a way that it attaches prism header[1] to all incoming packets, to be processed by sniffer softwares. The network layer ignores such packets. The implication of it was, the machines running in monitor mode were not able to resolve ARP request in first place. By going through bit-by-bit packet information at different function in monitor mode, we were able to recreate the received packet payload and pass valid information to network layer to enable monitor mode communication. Section 3.3.1 explains these modifications in more detail.

Before doing any changes in device driver, it is very important to understand the flow of packets in it. The transmit and receive path of a packet in madwifi is different for each mode of operation. Appendix B explains the packet flow in monitor mode. To give brief overview, all incoming packets from network layer comes to `ath_hardstart()` function. It then calls `ath_tx_startraw()`, which setup packet descriptor[2]. It then calls `ath_txqaddbuf()` function which enqueue packet on specific hardware queue. Once queued

[1]Prism header is inserted into the packet by the driver while sniffing wireless packets. Prism header contains the information like the time at which driver received the packet, the channel on which the packet was received, the signal strength and the noise level etc.

[2]packet descriptor defines a packet metadata such as packet type, packet length, its transmit rate etc..

on hardware queue, it calls `ath_hal_txstart()` function to instruct HAL to dequeue specific hardware queue and send all dequeued packets on air. On receiving side, all received packets from air triggers call to `rx_poll()` function. The `rx_poll()` function process each received packet and calls `ath_capture()` function to attach prism header to it. It then calls `ieee80211_input_monitor()` function, which passes received packet to network layer.

To better understand the proposed system, we divide our implementation in two parts. First part, explained in Section 3.3, deals with framework preparation for multi-hop TDMA implementation. It includes monitor mode communication, MAC layer packet sending, channel switching etc.. These changes are not related to core TDMA system, but are required to implement it. The second part, discussed in Section 3.4, explains the use of this framework in implementing the multihop TDMA system.

Before going in to details of implementation, we will first look at structures of schedule header, data header and routing tree elements that are used in our system. These headers, apart from facilitating multihop schedule dissemination and time synchronization, plays important role in enabling monitor mode communication and disabling virtual carrier sensing as explained in section 3.2.

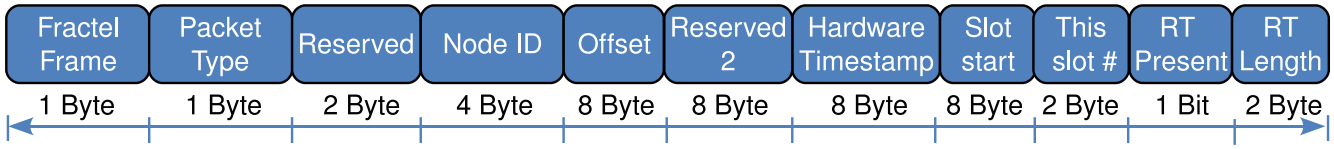
3.2 Packet Header Formats

In our implementation, we are using custom packet format. There are two types of packets, the schedule packet and the data packet. Each packet has header associated with it. Figure 3.1 describe these headers in more detail. Each byte position in packet header has some meaning associated with it.

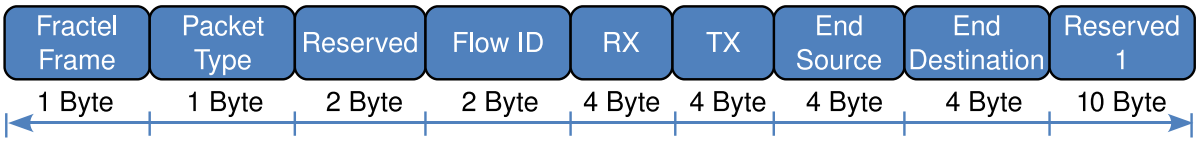
Field common in Schedule and Data header

Fractel frame: is used to indicate every packet as `FRACTEL_PACKET` by writing `0xFF` in first byte position. The standard defines `0xFF` as reserved bytes and hence will not affect the working of other WiFi devices in vicinity.

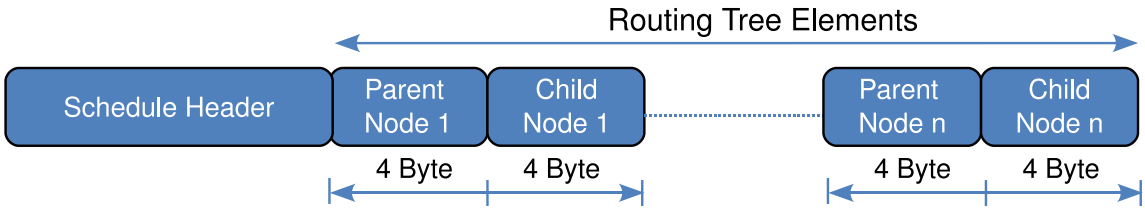
Packet type: field is used to indicate packet as either schedule packet or data packet. This



(a) Schedule Header



(b) Data Header



(c) Routing Tree Elements

Figure 3.1 Custom Packet Headers

field starts from bit position 8 and ends at bit position 15. Referring to standard IEEE 802.11 MAC header, the bit position 11 is used to indicate packet as retry packet. Now when we set 11^{th} bit position to 1 i.e mark it as retry packet, the modification done by hardware(wireless card) on packet data are avoided[3]. We use this technique and mark all custom packet as **retry packet**. We do this by writing `0x08` for schedule packet and `0x0C` for data packet. We use this field to identify packets on receiving side.

Reserved: field is used to suppress the effect of NAV field. Referring again to standard IEEE 802.11 MAC header, the 2^{nd} byte in MAC header represents the NAV field and is used for virtual carrier sensing. When set with some value, if instruct receiving device to backoff its transmission. In our implementation, we want precise control over packet transmission

[3]if retry flag is **not set**, the hardware will write sequence number at byte position 22 and 23 of the packet. Since our custom header spans more then 23 bytes, the sequence number will overwrite our valid header infomation on those two bytes. Refer section 3.3.2 for more explanation.

timing and hence **we write zero** in this field[4].

Schedule header fields

Node ID: holds the IP address of node that sent the schedule packet on air. This field is used by receiving node to know whether the schedule it received is from its intended parent in the topology.

Offset, Slot start, This slot #: are used for multihop time synchronization purpose.

Hardware timestamp: is used for multihop time synchronization purpose. This field starts at byte position 24 and ends at 31. We change schedule packet type to beacon type by setting `atype = HAL_PKT_TYPE_BEACON`. This will instruct hardware(wireless card) to write 8 byte hardware timestamp at byte position 24 to 31 of the schedule packet before sending it on air[5].

Reserved2: is reserved for future use.

RT Present and RT length: has 1 bit and 2 byte length respectively. When *RT present* is set to 1, it indicates that schedule packet has routing tree elements attached to it after schedule header. The receiving side will use this information to fetch routing tree from schedule packet. The length of routing tree is stored in *RT length* field.

Data header fields

Flow ID: is used to indicate flow id of data packet. The flow ID along with end-to-end source and destination enables the receiving node to store it in different TDMA queue. This feature has not yet been implemented in current multihop TDMA system.

TX and RX: is used to store IP addresses of sender and receiver node of the data packet.

End source and End destination: is used to store IP addresses of end-to-end source and destination of a data packet. This along with **TX** and **RX** fields are used for implementing centralized routing mechanism in our system.

Reserved1: is a 10 byte field and is used to make data header length big enough to nullify

[4]Section 3.3.2 explains it in more detail.

[5]Refer section 3.3.4 for detail explanation.

the changes made be device driver between byte position 22 to 31(both inclusive) in data packet. This field ensure that our network payload remains intact. We fill this field with zeros.

Routing tree element: consists of parent-child pair. There can be many routing tree element present in a schedule packet. The length of routing tree is stored in schedule header. The routing tree elements are sufficient enough to recreate complete network topology at any node.

3.3 Framework for Multihop TDMA Implementation

To implement TDMA MAC in madwifi, we first need to disable default CSMA mechanism. As explained in Section 3.1, out of six tasks for disabling CSMA mechanism, we were able to accomplish first three tasks by setting wireless card in monitor mode. Continuing from were we left, in this section we will discuss how we achieve remaining three tasks. We will also talk about how we enable monitor mode communication, how schedule packets are generated from MAC layer(without any packet injector utility) and how we timestamp the schedule packets. These functionality will provide framework for out multihop TDMA implementation. We will use these hooks extensively during core implementation of TDMA system discussed in Section 3.4.

3.3.1 Monitor Mode Changes for Two Way Communication

When card is in monitor mode, it not only dumps all the packets that it sees on air, but also allows us to send packets on air. The difference is, the packet sent in monitor mode does not have 802.11 header attached to it. In a sense, it allows us to send raw packets. But the problem is, we can not ping from one device to another(both running in monitor mode). By following packet path in monitor mode we found that the ARP requests were not getting resolved. Section 3.3.1.1 explains hows we enabled ARP resolution in monitor mode.

3.3.1.1 ARP Resolution

We found that during ARP resolution, **valid ARP request** packets were being sent by sending node but on receiving side the 26 byte prism header was getting attached to every ARP request packet and was passed to network layer. This behavior was corrupting ARP request packet and so receiving side was not generating **ARP reply packet**. To fix it, **we removed 26 byte prism header from head and 4 byte CRC from tail** of all ARP request(broadcast packets) in `ieee80211_input_monitor()` function as explained in Figure 3.2.

After doing above changes, the receiver node were generating ARP reply packet on response to ARP request packet, but now for ARP reply packet, the receiving side, apart from attaching 26 byte prism header, was also attaching 14 byte ethernet header. In all, the ARP reply packet was having 40 extra byte attached to it. We also found that byte position 56 and 57 of the ARP reply packets were getting replicated at byte position 58 and 59 respectively(these bytes were modified between call to `ath_capture()` and `ieee80211_input_monitor()` functions) and this modified ARP reply packet was passed on to network layer. This behavior was corrupting ARP reply packet and this was the main reason why we were not able to resolve ARP request in monitor mode. To fix it, **we removed first 40 byte from head and 4 byte CRC from tail of ARP reply packet**. We also **remove byte 56 and 57** as explained in Figure 3.2. After doing above changes we were able to resolve ARP in monitor mode, but when we tried to ping from one machine to another, we could not. Section 3.3.1.2 explains how we achieved monitor mode ping in more detail.

3.3.1.2 Ping in Monitor Mode

After doing changes described in section 3.3.1.1, we were able to resolve ARP entry in monitor mode but still, we were not able to ping between two machine running in monitor mode. By going through byte-by-byte data of ping packet we found that, sending side was writing sequence number at byte position 22 and 23, which for us was valid ping data.

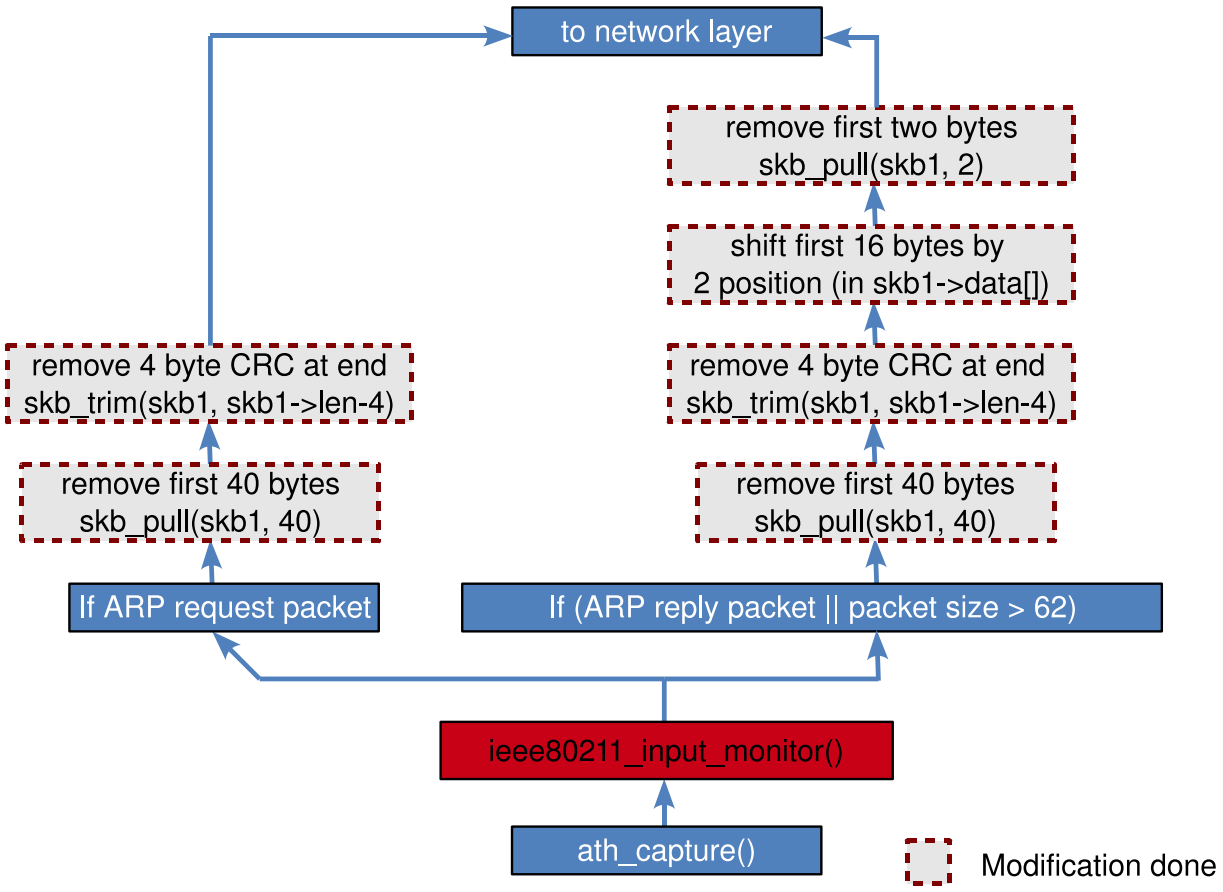


Figure 3.2 Changes made in ieee80211_input_monitor() for ARP Request, ARP Reply and all other Data Packets

To prevent such printing, we set retry flag for all packet, by attaching data header to all outgoing packets[6] in `ath_hardstart()` function as explained in Figure 3.3. This ensured us that sending side in not modifying any valid ping data byte. The **data header** is explained in section 3.2.

On receiving side, we found that, it replicates byte 30 and 31 at byte position 32 and 33 respectively. So we removed byte 30 and 31 from received packets in `rx_poll()` function. We also stripdown custom data header attached to all data packet. Apart from these changes, we also found that like ARP reply packet it also attaches 26 byte prism header and 14 byte ethernet header to every non-broadcast packet and also replicate byte position 56 and 57

[6] we attache data header to all outgoing data packet in our system

at byte position 58 and 59 respectively. We fix it the same way we did it with ARP reply packet as shown in Figure 3.2.

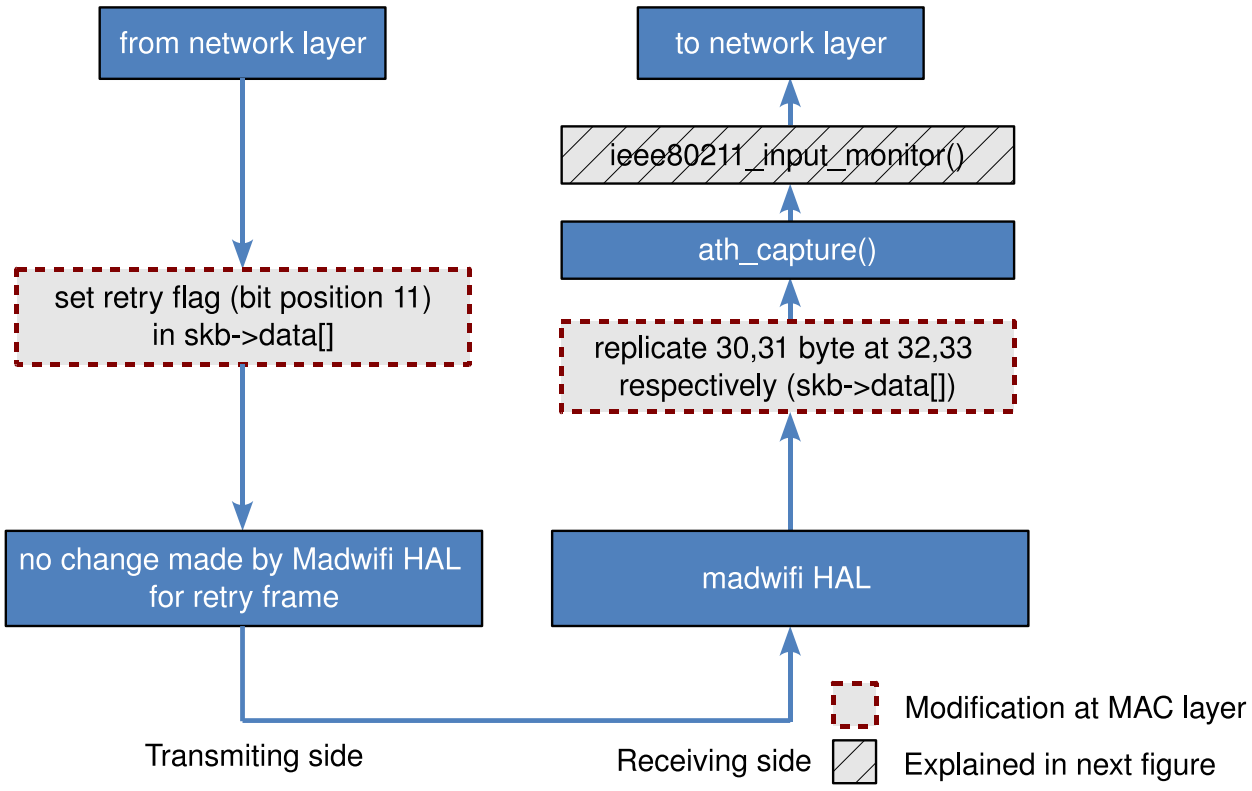


Figure 3.3 Changes made in rx_poll() function for all data packets

After doing the changes described in section 3.3.1.1 and section 3.3.1.2, we were able ping from one device to another in monitor mode.

3.3.2 Effect of NAV and Sequence Number field

The IEEE802.11 frame structure contains a NAV field that causes other nodes to backoff while the current transmission is under way. We do not need this field because the trans-

mission of packets happens strictly in a node's own transmit slot. However, the value of the NAV field is used by the receiving devices to perform virtual carrier sensing and backoff their own transmissions in hardware. Since our headers, described in Section 3.2, replace the standard IEEE802.11 header, the value present in this field must be zeroed to prevent other devices from unnecessary backoff. Also, the hardware stamps a sequence number at byte 22 and 23; the driver does not have control over what value will be written in this field. Since our custom header exceeds 23 bytes, stamping of the sequence number corrupts the header. As a work-around, we set the `RETRY` flag to suppress the stamping of the sequence number field by the hardware, as suggested in [8].

3.3.3 Generating RAW Packet at MAC Layer

We do not use the standard IEEE 802.11 [1] frame structure while sending schedule or data packets. Data packets arrive to the MAC layer from the network layer and are attached with a custom data header. Schedule packets, on the other hand, are generated in the madwifi driver itself. We are not using any packet injector utility so as to minimize packet generation delay. We have written a function similar to `ieee80211_send_qosnulldata()` that uses `ieee80211_getmgtframe` to allocate `skb`, fills in the schedule data-structures and then sends the packet on air by calling the `ath_startraw()` function. Both the schedule header and the data header contain `0xFF` as the first byte so that all receivers can clearly distinguish valid IEEE802.11 packets from ours.

3.3.4 Hardware Timestamping

The schedule packets contain a hardware timestamp to maintain synchronization of all nodes. This time stamping, if done in software, is inaccurate, because we cannot be sure when the packet will leave the hardware. A similar requirement is present for beacon packets in default 802.11 MAC. All beacons are timestamped with a 64bit microsecond granularity value by the hardware at bytes position 24 to 31(of beacon packet). The Atheros hardware can be made to timestamp any packet by setting the **packet type** flag to `HAL_PKT_TYPE_BEACON` in

the call to the `ath_setup_txdesc()` function. This instructs Atheros HAL to write hardware timestamp on such packets. In our implementation all schedule packets are sent with this flag set. This along with other information present in schedule header are use by all non-root nodes in implementing multihop time synchronization.

3.3.5 Channel Switching From Driver

Using multiple channels for transmission on different link will definitely improve the link utilization and overall system throughput. One way to change channel is to use `iwconfig` command from shell. In our implementation, we required it be done from driver itself. This enables us to switch channel from our TDMA protocol itself(through driver module) rather then invoking iwconfig using system command. Since `iwconfig` had a way to change it, we looked at iwconfig source code and found a hook into madwifi driver through IOCTL. What we need is to set `ic->curchan` structure with required channel information and calling `ic->ic_set_channel(ic)` with channel structure as argument. The `ic->ic_set_channel(ic)` is the function which changes the operating frequency.

There are few pre-conditions for channel switching. The driver will flush the current queue and will reset the hardware before switching it to the new frequency. This process incurs delay of the order of 3-5msec [12]. Though in our implementation we are not queuing any packets on hardware queue and hence this delay might be less then what is stated [12][7]. In our implementation, we have not used channel switching yet.

3.3.6 Configuration Through `proc` Filesystem

The madwifi wireless modules runs in kernel space. To change the numerical values of a variable from user space, one way is to recompile the driver again and do unload-load procedure. The other way is to use `/proc` file system. The `/proc` file system allows communication from user space to kernel space, without recompilation. We have used it for making TDMA system specific configuration such as assigning different node_id and IP address for each node. We

[7]We have not yet characterized channel switching delay ourself

now flash single code on all devices and make device specific changes through /proc entry.

3.4 Multi-Hop TDMA System

Once the framework is prepared, we are in a position to implement the proposed system. The system consists of three core logical modules and each module has set of helper functions at its disposal.

Implementation of TDMA Queue: The TDMA queue is used to buffer all incoming packet at MAC layer. Packet can arrive from network layer or from air. Depending on routing entry and packet content, packet is either passed to network layer or added to TDMA queue for multihop transmission. It is a software queue and is implemented as singly linked list.

Core Slotting Structure: The `fractel_event_handler()` is the core function which handles slotting structure. It is called periodically through timer interrupt with periodicity equal to slot size(interval). The `slot counter` is used as indicator of active slot. The slotting structure uses variables such as number of control, contention and data slots which are configurable through /proc entry.

Centralized Routing: It uses routing tree and routing map information to enable communication between different nodes. This functionality can be extended for dynamically changing topology either through schedule dissemination or through `/proc` entry.

3.4.1 TDMA Queuing Mechanism

All packets arriving at the madwifi driver enter through the `ath_hardstart()`. Depending on whether the device is in the monitor mode or not, the `ath_hardstart()` sends the packet to the `ath_tx_startraw()` or `ath_tx_start()` function respectively. Since we need

precise control over packet transmission times, we buffer all incoming packets in a software queue instead of allowing them to flow through the `ath_hardstart()`. During the node transmission slot(data slot), packets are dequeued and handed over to the `ath_hardstart()` function which attaches the data header and sends the packet on air. The number of packets sent during a transmission slot is equal to the lesser of the number of packets that can be transmitted in the slot interval and the number of packets present in the buffer.

On the receiver side, an arriving packet will have one of the following three destinies. It may either be intended for consumption by the receiving node, or may be required to be forwarded to another node (this node is a relay) or may have nothing to do with this node, in which case, it must be dropped. Specifically, a packet is thought to be destined to a node if its ID appears in the `end_destination` field or if both the `next_hop_dest` and the `end_destination` fields have broadcast address. In such a case, the packet is sent to the network layer for its consumption. If the node's ID appears in the `next_hop_dest` field, but not in the `end_destination` field, the packet must be forwarded to another node. Such packets are enqueued in the TDMA buffer. All other packets are dropped. Currently, we have implemented static routing inside the driver code. Thus, a relay node always knows the next hop destination for an arriving packet that must passes through it. Figure 3.4 shows the flow of TDMA queuing mechanism.

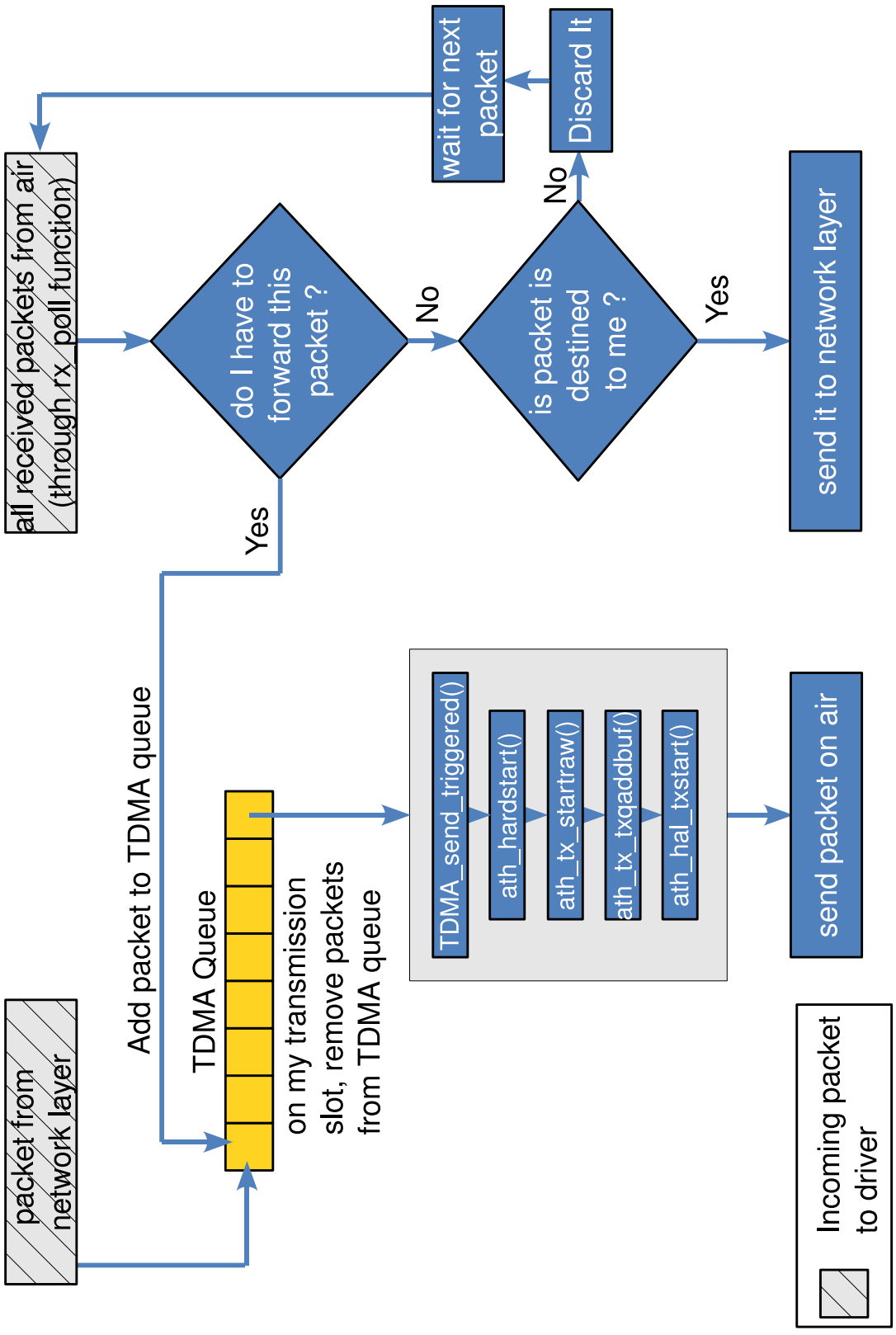


Figure 3.4 Implementation of TDMA Queue

3.4.2 Implementation of Slotting Structure

Our implementation has three types of slots. There are fixed number of **control slots**[8]. The control slots are used for sending schedule information. There can be multiple **contention slots**, which is used for sending information back to root node. The **data slots** are used for sending normal data packets at each node. Figure 2.2 depicts the logical frame structure. The slot interval is fixed for a topology but can be changed through `/proc` entry.

The `fractel_event_handler()` is responsible for carrying out different tasks for different types of slots. This function can be called from three places,

- when any node other then root node receives the schedule from its parent
- the root node calls it for sending first schedule packet
- once it is called, it calls itself repeatedly through software timer interrupt. The periodicity of timer is equal to slot interval

The root node calls it through bootstrap timer[9]. Once the timer is triggered it calls `fractel_event_handler()`. As shown in Figure 3.5, this function keeps track of current slot number and depending on its type it sends schedule or data packet respectively. The flow of schedule packet for root and non-root node is different. The root node will prepare new schedule each time whereas non-root node will forward received schedule packet[10]. The root node also sends routing tree information along with schedule packet. The routing tree will be used by non-root node to know its assigned parent. Note that, before sending the schedule, both root and non-root node will change the `offset` and `node_id` information, which will be used for multihop synchronization as shown in Figure 3.5.

The data slots are used for sending normal traffic. Each node knows its own transmission opportunity i.e data slots. Each node has been assigned node_id, zero for root node and 1 to (max_device - 1) to non-root device as shown in Figure 2.3. In our five

[8]Number of control slots is equal to number of nodes in the topology

[9]Bootstrap timer is used for setting up `/proc` entry

[10]non-root node stores the schedule it receives from its parent for multi-hop transmission

node topology, shown in Figure 4.1 each node gets every fifth data slot for transmission. When a node's transmission opportunity comes, it removes packets from TDMA queue using `TDMA_send_triggered()` and sends it on air. The number of data packets sent in a single slot is a function of *slot size* and *transmission rate*[11].

[11] with slot size of 5msec and transmit rate of 54 Mbps, one can send 20 packets of 1470 byte each in single slot. This calculation is independent of the underlying network topology

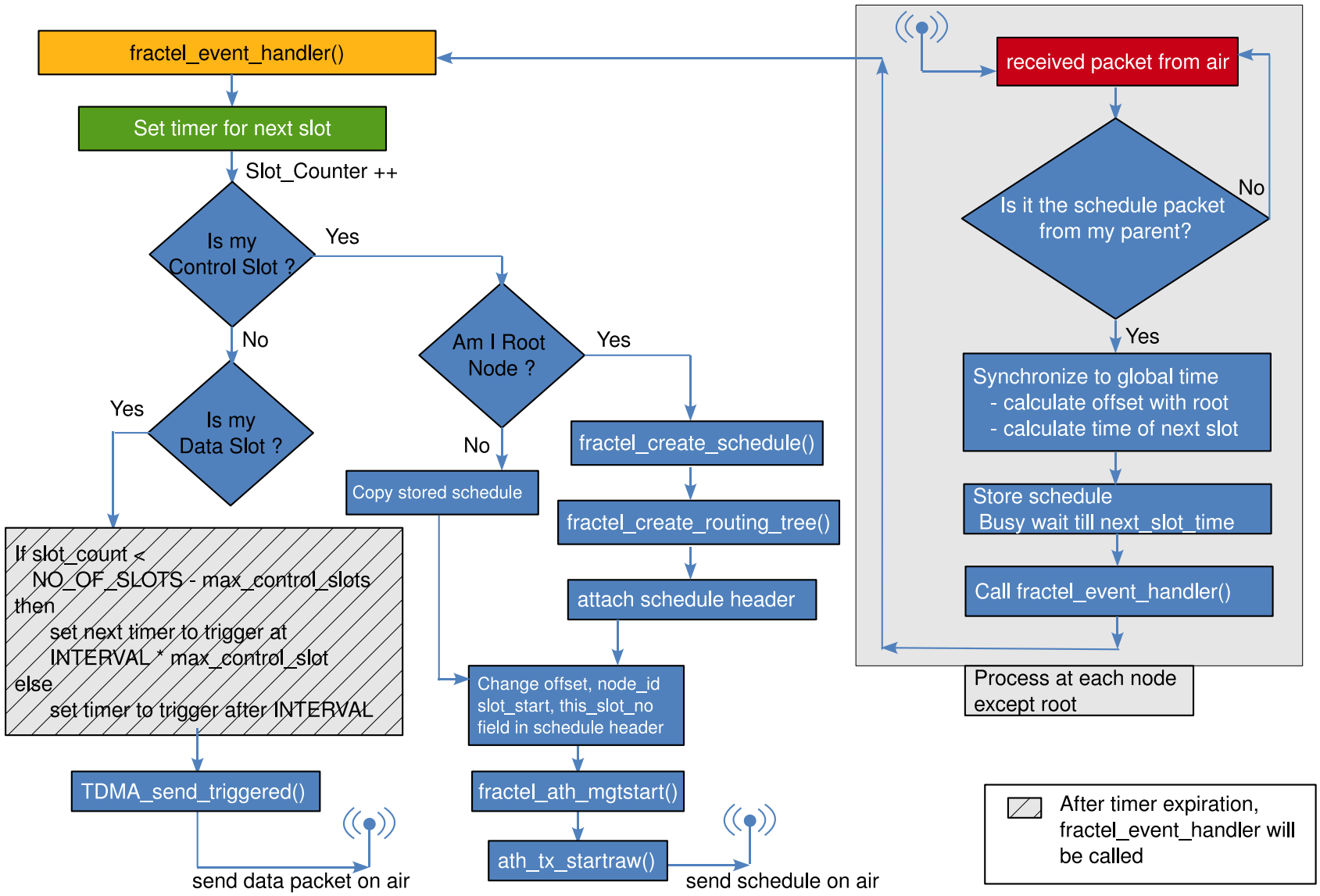


Figure 3.5 Fractel event handler routine for implementing core TDMA slotting structure

3.4.3 Implementation of Centralized Routing

Every packet in our implementation is either schedule packet or data packet. The schedule packet has schedule header and number of routing tree elements, while data packet has data header. The schedule header has sufficient information for multihop dissemination of schedule across all nodes. While for data to be transmitted from intended source to destination, we use data header as placeholder for routing information. As we are running in monitor mode and are using custom frame format, the linux inbuilt routing will not work in our case.

We have used two data structures for filling routing entry in data header. One is **MAC-to-IP table**, used for mapping MAC address to IP address and another is **routing table**, used for filling `next_hop_ip` address. Both tables are explained in Figure 3.6 along with working of centralized routing mechanism. Every packet has end-to-end source and end-to-end destination MAC address attached in ethernet header. We used this MAC addresses to fill corresponding IP address using MAC-to-IP data structure. This two IPs goes into `end_source_ip` and `end_destination_ip` field in data header. To send data packet to other node, every node must attach next_hop_ip. This field indicates that, for this data packet next_hop_ip should forward it ahead. Referring to Figure.3.6, when 192.168.0.1 sends data to 192.168.0.5, it fills next_hop_ip as 192.168.0.2 . This, next_hop_ip entry is stored in routing table data structure. Every node refers to routing table for finding valid match in it.

When a node receives any data packet[12], it checks whether it is

- **destined to me** - *if* (end_destination == me and next_hop_ip == me) **OR** it is broadcast packet *then* pass it to network layer
- **destined to other node through me** - *if* (end_destination != me) **AND** (next_hop_ip == me) *then* remove data header and add it to my TDMA queue for multi hop transmission[13].
- **not my packet** - discard it and free the resource, do not take any action.

[12]Data packet is identified using `skb->data[1]==FRACTEL_DATA`

[13]Data packet still has ethernet header attached to it. We will use it again while transmitting it from TDMA queue to fill end-to-end source and end-to-end destination IPs.

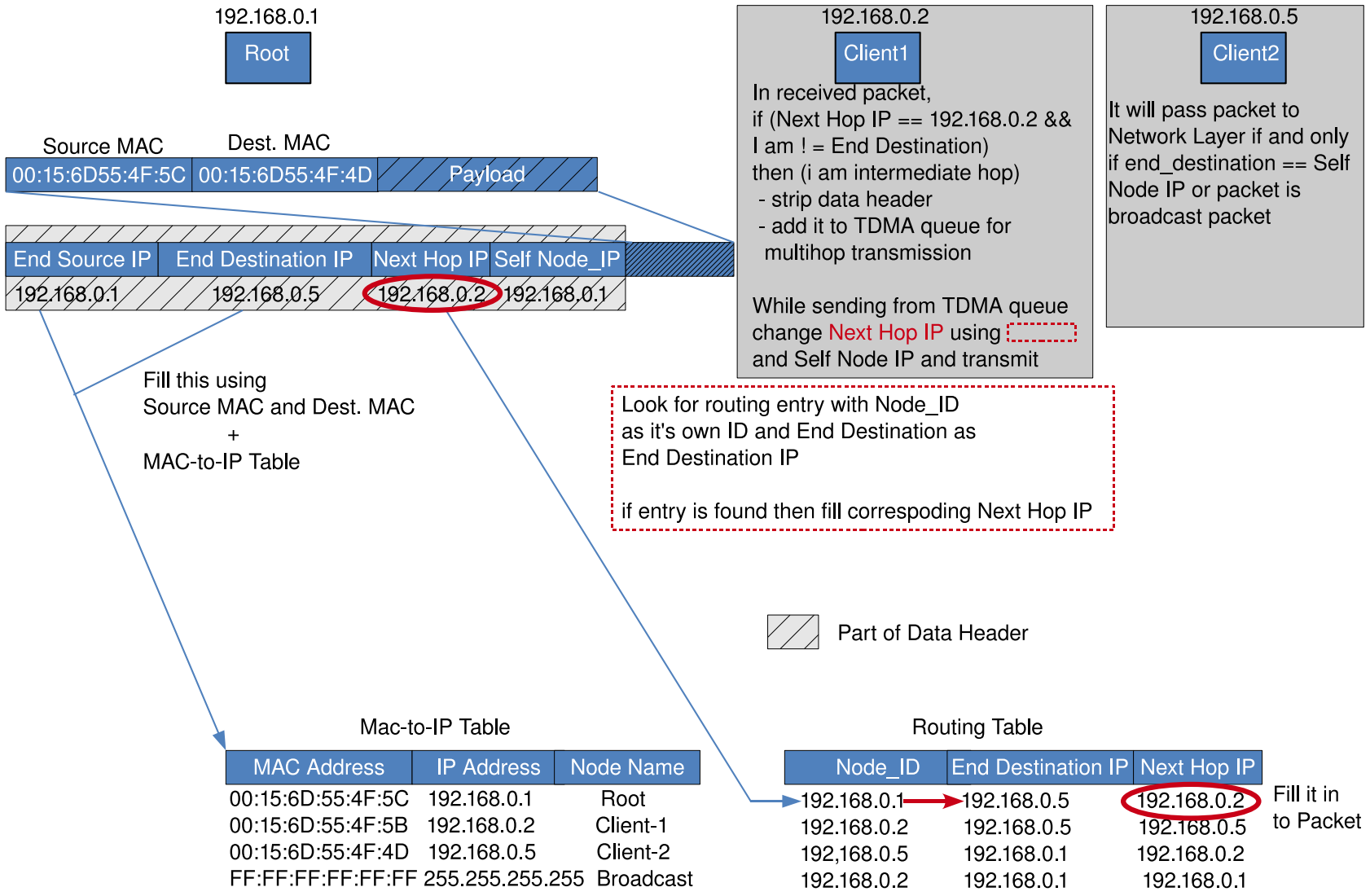


Figure 3.6 Centralized Routing Implementation

3.4.4 Small Slot Size and MTU

In our implementation, we buffer all the packets until transmission slot occurs. Once transmission slot starts, we check whether current packet at the head of the queue is small enough to send in current slot. If not, we stop transmission. This causes problem at small data rate with small slot size. For example, consider 1 Mbps data rate with 1 msec slot size. In such setting maximum data that can be sent in one slot is 125 Bytes. If packet with length more than 125 bytes comes to queue, the TDMA transmission logic will not send it and such packet will never get out of queue which will stall complete TDMA system. The work around for this is to set MTU such that MAC layer never gets such large packet for current slot size and transmission rate.

3.4.5 Multihop Time Synchronization

The efficiency of a TDMA system depends on the simultaneous triggering of same slot at each node. The requirement is, at any given time all nodes in the network should see exactly same slot. Apart from this, we also have to account for clock drift[14]. We answer these question by designing simple yet novel method of multihop time synchronization. The design and implementation of multihop time synchronization has been done by ***Ashutosh Dhekne*** as a part of his master thesis. We have incorporated this mechanism in to our multihop TDMA system.

To give brief overview, in our implementation, synchronization is done in every control slot by sending schedule packet with a hardware timestamp. The synchronization information propagates down the tree, one hop at a time, to all nodes in the network. Each schedule packet contains its own offset from the global time and the exact global time of the beginning of this slot[15]. Together, these three entities enable the receiving node to synchronize to the current global time and also calculate the next slot time. Once synchronized, every node sets

[14]when we start two clock exactly at same time, after some time both will show different time. This behavior is called as clock drift.

[15]in which it receives schedule packet

periodic timer to enable slot triggering. Through experimentation, we found that the clock drift between different card pairs is different and is of the order of $15\mu s/sec$. To counter this, we re-synchronize every node at regular interval and also place $100\mu s$ guard band in each slot.

3.4.6 Understanding Complete Flow

We have looked at detailed implementation of various modules in previous sections. The modules interact with each other to implement multihop TDMA system. Let's look at complete picture of what we have discussed in this chapter. Figure 3.7 shows complete data flow implemention of multihop TDMA system. Packets can arrive to MAC layer either from network layer or through air. Depending on type of packet we attach either schedule header or data header to each packet. The schedule packets are created only in root node while non-root nodes stores such schedule packet when it receives one from its assigned parent. They sends stored schedule during their control slots. Every schedule packets are timestamped for multihop time synchronization. The data headers are attached to every outgoing data packets in `ath_hardstart()` function.

On receiving side, packets are first checked to be of fractel packet, by checking packet type field in both schedule and data header. The schedule header has routing tree elements, which is used by receiving node to recreate complete tree topology. If schedule packet received by any node is from its intended parent, then receiving node uses information present in schedule header to synchronize itself with global clock. We use data header to route packet from one node to another in multihop TDMA system.

Data packets received from air at any node has three possible flow

- It is destined to me OR it is a broadcast packet then pass it to network layer
- If I am a relay node for this packet then remove data header and 4 CRC and add it to TDMA queue
- If it is not destined to me then discard it

As show in Figure. 3.7, both data and schedule packet has different flow in our implementation. The `fractel_event_handler()` function(explained in Section 3.4.2 and Figure 3.5) implements core slotting structure and with the help of various helper functions(explained in appendix C) implements multihop TDMA system.

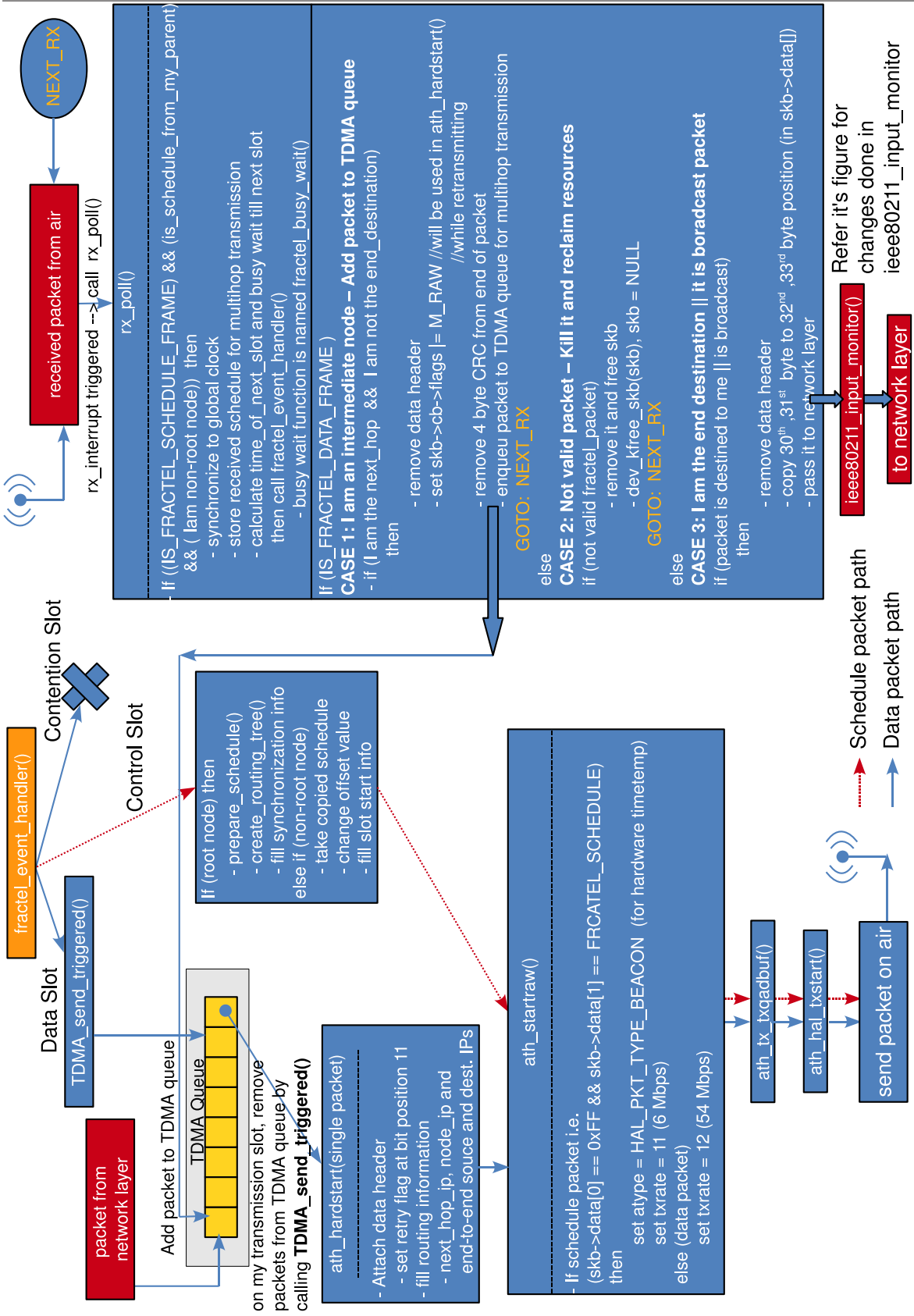


Figure 3.7 How packet flows in Fractel TDMA

Chapter 4

Experiments and Results

Through various experiments, we seek to answer the following questions.

1. What is the impact of changing the number of hops on UDP and TCP throughput?
2. What is the impact of changing the slot size on UDP and TCP throughput?
3. How are number of hops and slot size related in terms of UDP and TCP throughput?
4. What is the impact of the number of hops and the slot size on the round trip delay of packets?

All the experiments done here are in an interference free 802.11a frequency range (channel 160). Though these experiments were conducted in an indoor setting, we believe that the results can be extended to interference free outdoor links. Nevertheless, such claim should be made only after careful experimentation. Long distance links also cause propagation delay ($\sim$83μs/25km link), which as of now we have have ignored in our implementation.

4.1 Experimental Setup

In order to answer the above questions, we conducted a number of experiments on a linear topology consisting of five nodes as shown in Figure 4.1.

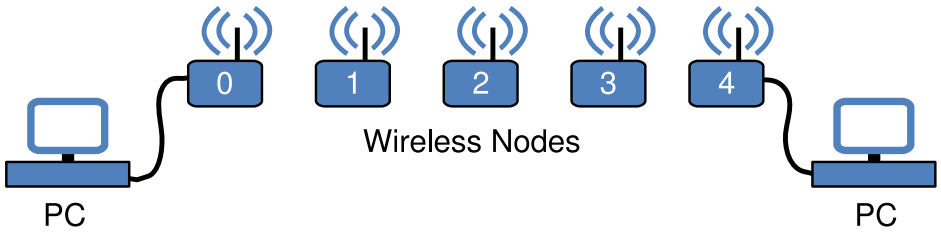


Figure 4.1 Linear topology used in our experiments.

One node is designated as root node and generates control packets. The routing tree contains information about the linear topology. Each node is numbered starting at 0 from the root node downwards in the topology. Each node sends packets when (slot number) *modulo* (number of nodes) matches its own node id. The contention slots are unused. All data packets are destined either to the root node or the leaf node, and routing entries facilitate routing of data. The number of control, contention, and data slots, and the slot interval are all configurable in the user space through a `/proc` entry. We have used 3 control, 5 contention and 92 data slots(total 100 slots/frame) in this setup and the slot interval is varied as described in individual experiments. UDP and TCP throughput is calculated using *iperf* tool running between the two PCs.

4.1.1 Theoretical Expected Throughput

All nodes are set to transmit at 54Mbps and can transmit only in their own transmission slots. With the configuration described in Section 4.1, with five nodes transmitting, we use 87 of the 92 available data slots[1] in a round-robin fashion, so that each node gets $87/5 = 17.4$ slots per frame. The number of packets sent in each slot depends on the slot size and the size of the packet. Table 4.1 shows the transmit time for the various parts of a packet at 54Mbps. Equation 4.1 calculates the theoretical throughput for the 4-hop case with the slot size of 2ms and a 100μs guard band giving 87 slots per second to each node. Similar calculations are performed to derive the theoretical maximum throughput for any number of hops.

[1]The last x data slots in a frame are not used so that the control slot timer for the next frame is triggered precisely. x is equal to the number of nodes in the network.

Table 4.1 Time taken to transmit various portions of the packet at 54Mbps

Description	Bytes	Time (μs)
UDP Payload	1470	217.77
UDP Header	8	1.185
IP Header	20	2.962
Ethernet Header	14	2.074
CRC Trailer	4	0.592
Fractel Data Header	32	4.740
PLCP Header	-	20.444
Total	-	249.767

$$\begin{aligned} Transmit\,time(slotsize) &= 1900\mu s\,(2000 - 100\mu s\,guard\,band) \\ Packets/slot &= 1900/249.767 = \lfloor 7.607 \rfloor \\ Packets/sec &= (frames/sec) * (\#ofslots/frame) * (packets/slot) \\ &= 5 * (87/5) * 7 = 609 \\ Throughput &= 609 * 1470 * 8/(10^6) \\ &= 7.16Mbps \end{aligned} \tag{4.1}$$

4.1.2 Number of Hops and Throughput

We conducted experiments for TCP and UDP throughput on linear topology with varying number of hops from 1 to 4 and slot size 2ms. The results are shown in Figure 4.2. UDP throughput decreases with increasing number of hops since the available bandwidth is time-divided by the number of hops a packet has to cover. TCP throughput decreases much faster than UDP throughput because an increase in number of hops means an increase in end-to-end error probability and also an increase in the round trip delay. Moreover, since we have disabled per link retransmissions, TCP throughput suffers drastically. When we use multiple TCP connections, depicted in the graph as TCP-2 and TCP-4, the total available

bandwidth is shared between them. Thus the cumulative bandwidth approaches that shown by UDP.

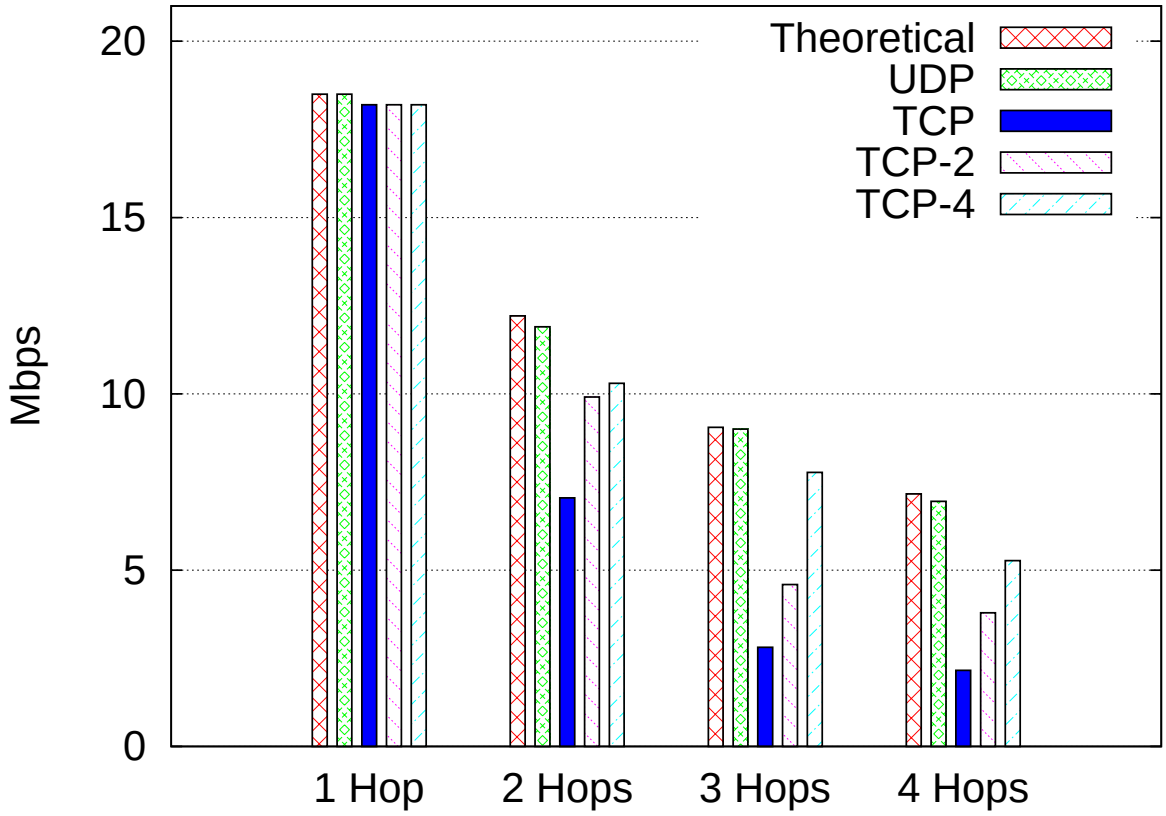


Figure 4.2 UDP throughput decreases with increasing number of hops. TCP throughput decreases much faster due to no link-to-link retransmissions.

4.1.3 Slot size and Throughput

The slot size variations should ideally have no impact on the UDP throughput. However, since we do not fragment packets at the MAC layer, an increase in slot size causes lesser overhead. Also, we implement a small guard band of 100μs for every slot. UDP performs better with increasing slot size. TCP throughput is adversely affected by increased slot size because it causes a higher delay in receiving acknowledgements as shown in Figure 4.3. Since it never fully uses the available bandwidth even in smaller slot sizes, the reduced overheads

in larger slot sizes do not affect TCP throughput. Since a single TCP connection does not fully utilize the available bandwidth, we experimented with multiple connections. We found an equivalent increase in the cumulative throughput as shown by 4-TCPs-3Hops readings in Figure 4.3.

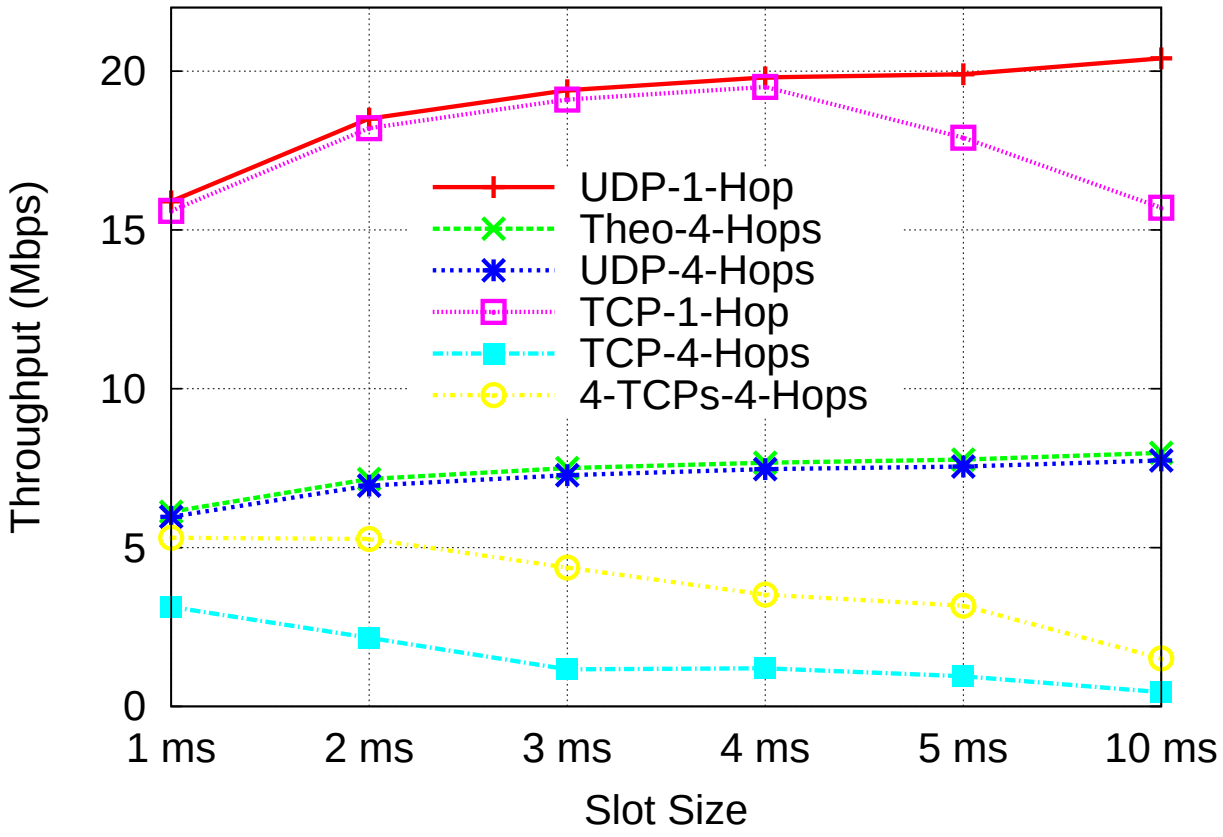


Figure 4.3 UDP throughput increases with increasing slot size. TCP performs worse with increasing slot size due to increased RTT.

4.1.4 Slot Size and Number of Hops

As expected from the discussion above, UDP shows better performance with *increasing* slot size and decreasing number of hops. TCP, on the other hand, shows better performance with *decreasing* slot size and with decreasing number of hops. This is evident form the Figure 4.3.

4.1.5 Delay Characteristics

In our experimental setup described in the section. 4.1, a packet sent by the root node, will be transmitted over consecutive data slots to reach the leaf node. Since the data slots are numbered from 0 to 4 and then the numbering is restarted from 0, a packet sent from the leaf node to the root node, will be transmitted by the intermediate nodes only when their transmission turn occurs. The round trip of a ping packet is depicted in the Figure 4.4 and formalized in equation 4.2, where x is the number of nodes. We note that the equation shown here is a function of the way we have numbered the slots. It will be different for different slot numbering patterns.

$$Best\,case\,RTT = x + ((x-1) * (x-2)) \tag{4.2}$$

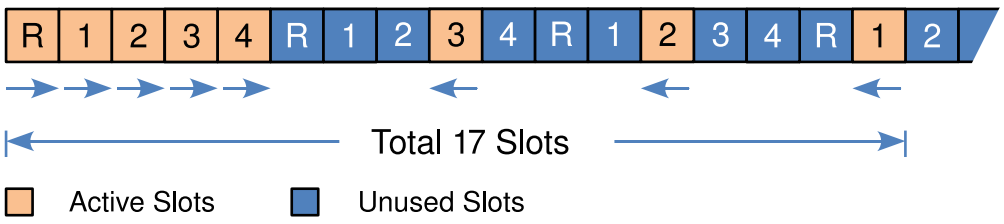


Figure 4.4 The best case round trip of a ping packet.

In addition to the delay, the packet delay variation (jitter) is also an important parameter for good quality of service for audio and video communication. A jitter below 100ms is generally believed to be good enough for such applications. In our experimental setup, we observed the jitter to be less than 3ms. Figure 4.5 shows a frequency plot of the observed jitter values for the 4-hop setup with slot size set to 5ms. The best case round trip time for this network is 85ms as calculated from Equation 4.2 which matches the RTT value we observed during our experimentation. Jitter is independent of the slot numbering pattern.

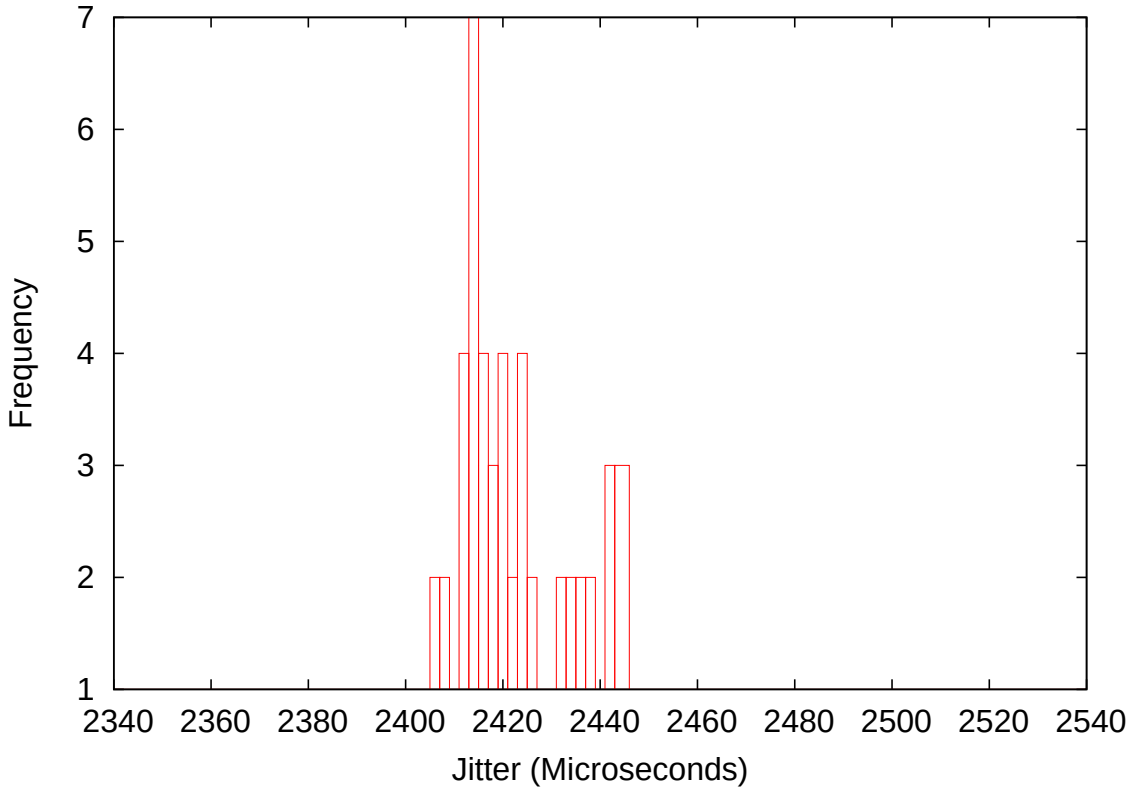


Figure 4.5 The jitter observed is under 2.5ms for 4 hop topology with slot size of 5ms.

4.2 Implications of Results

After doing detail experimentation we found that

- The observed UDP throughput is very close to the theoretically predicted value.
- The throughput for *single-TCP* connection is quite low for multiple hops but ***multiple-TCP*** connections together can provide good performance.
- the observed jitter value is very small even for multihop topology.

During the testing of our prototype, we played videos from one PC to another in the topology shown in Figure 4.1 and also made voice calls using Ekiga between the two machines. Both these applications showed very good performance. This implementation is analogous to a

situation where a remote village is connected to a computer in a large town and an e-learning video conference application is running over long distance multihop links.

Though our experimentation was based on linear topology, we are confident that it will work for tree topology, since the multihop schedule dissemination and time synchronization mechanism are implemented for tree topology. Our linear topology is just a simple case of general tree topology. Our experimental setup can be thought of as tree with depth 5. The performance of tree topology will largely depend on the efficiency of TDMA scheduler and how data flow are schedule in the network. Though the design of an efficient/optimal TDMA scheduler is in itself a challenging task, assuming we have efficient TDMA scheduler, our system will work for any generic tree topology.

Chapter 5

Conclusion

We have shown that a multihop TDMA protocol can be built on top of commodity hardware by modifying the madwifi driver. We have also designed a complete multihop TDMA system to support multiple flows, allow new nodes to join and leave dynamically and request for change in bandwidth. However, in this first proof-of-concept, we have implemented a linear topology with statically allocated transmission opportunities and static routing entries. Nevertheless, the implementation is very close to what we expect from the full-fledged protocol since we have built placeholders for almost all aspects of it–we have control slots, contention slots, data slots and perform multihop synchronization and data transfer between nodes. In a real deployment with long distance links, channel switching and spatial reuse is possible. In our indoor setting, spatial reuse was not possible but avenues exist for channel switching. We defer it to future work.

Our work has paved a path for future intensive work in the area of long distance communication over wireless multihop links. When completed, the system will also be useful for providing cellular connectivity using the long distance WiFi links as back-haul connecting a rural base station with the other base stations at different location.

Bibliography

[1] IEEE P802.11, The Working Group for Wireless LANs. http://grouper.ieee.org/groups/802/11/.

[2] Mikrotik RB433AH Decumentation. http://routerboard.com/comparison.html#powerSeries.

[3] Mikrotik Website. http://www.mikrotik.com.

[4] OpenWRT Website. http://openwrt.org/.

[5] The MadWifi project Website. http://madwifi-project.org/.

[6] WiMAX Technology. http://www.intel.com/technology/wimax/.

[7] K. Chebrolu and B. Raman. Fractel: a fresh perspective on (rural) mesh networks. In *NSDR '07: Proceedings of the 2007 workshop on Networked systems for developing regions*, pages 1–6, New York, NY, USA, 2007. ACM.

[8] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald. Softmac - flexible wireless research platform. In *Fourth Workshop on Hot Topics in Networks (HotNets-IV)*, November 2005.

[9] R. Patra, S. Nedevschi, S. Surana, A. Sheth, L. Subramanian, and E. Brewer. Wildnet: Design and implementation of high performance wifi based long distance networks. In *Fourth USENIX Symposium on Networked Systems Design and Implementation*, April 2007.

[10] B. Raman and K. Chebrolu. Design and evaluation of a new mac protocol for long-distance 802.11 mesh networks. In *ACM MOBICOM*, August 2005.

[11] A. Rao and I. Stoica. An overlay mac layer for 802.11 networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 135–148, New York, NY, USA, 2005. ACM.

[12] A. Sharma and E. M. Belding. Freemac: framework for multi-channel mac development on 802.11 hardware. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 69–74, New York, NY, USA, 2008. ACM.

[13] A. Sharma, M. Tiwari, and H. Zheng. Madmac: Building a reconfiguration radio testbed using commodity 802.11 hardware. pages 78–83, Sept. 2006.

Appendix A

Installation Howto

A.1 Installing OpenWRT on Mikrotik RB433AH

During the course of report, we talked lot about OpenWRT, kamikaze 8.09 and Router board 433AH. The obvious question is how to install openwrt on RB433AH. Before that we need to understand why we are using Router board and openwrt in first place.

As explain earlier the goal of the project is to use low cost off-the-shelf WiFi hardware. Router board RB433AH is such hardware with 680 MHz CPU, one of it's kind given its price and features [2]. The router board comes with proprietary router board OS. Since it has licenced OS, one can not play around with it's firmware. While complete opposite to it is OpenWRT, A tiny linux distribution which gives complete control over underlying hardware. Once installed, it feels as if running any normal linux distribution. The OpenWRT supports various hardware from different OEMs. Each hardware has it's own installation method.

A.1.1 Step 1: Setting up serial console

The RB433AH has a serial port, which gives command line interface. One can use linux tool such as *minicom* for accessing it. RB433AH uses 115200Bps and 8N1 setup for serial connection. Below are the steps for configuring RB433AH serial console.

```
sudo apt-get install minicom
```

start minicom using

```
minicom -s
```

select `Serial port setup` and set

```
A - Serial Device    /dev/tty*  //where * is your serial port number
E - Bps/Par/Bits     115200 8N1.
F - Hardware Flow Control     NO
G - Software Flow Control     NO
```

Refer appendix A.2 for detailed information on `/dev/tty*` i.e knowing serial port number on your computer. Now save you setting and start minicom. Assuming you have connected NULL modem serial cable to RB433AH, you should see Router Board command prompt on your console.

A.1.2 Step 2: Building OpenWRT-Kamikaze 8.09

Once RB433AH is powered on, press any key to enter setup mode. Once in to setup menu, it will show you various configuration option. To install OpenWRT, you first need to erase proprietary firmware by typing *e*. At this point RB433AH has no bootbale operating system. The only way to install OpenWRT is through network boot.

Assuming you are in setup menu, press *o* and select ethernet boot by pressing *e*. This will set RB433AH to boot from network and not from local NAND flash. Now save this changes by pressing *x* and reboot RB433AH. After reboot, it will search for kernel on network. Since we haven't made any kernel lets make one for RB433AH.

A.1.2.1 Download OpenWRT-Kamikaze 8.09

Download OpenWRT latest build using

```
wget http://downloads.openwrt.org/kamikaze/8.09/kamikaze_8.09_source.tar.bz2
```

Extract kamikaze 8.09 using

```
bzip2 -d kamikaze_8.09_source.tar.bz2  //will create .tar file
tar -xvf kamikaze_8.09_source.tar //extract complete source to kamikaze_8.09
                                  // directory
```

now goto kamikaze directory by typing,

```
cd kamikaze_8.09
```

A.1.2.2 Building OpenWRT-Kamikaze 8.09

OpenWRT can be build by non-root user only. Make sure you carry out all further steps as non-root user.

```
make menuconfig V=99   // V=99 will give debug messages
```

it will ask to install few dependencies. Run apt-get to install all dependencies.

```
apt-get install zlib, gawk, perl, path, libncurses5-dev, autoconf
```

depending on your linux distribution, you might have to insall other dependencies. Again type

```
make menuconfig V=99
```

If all dependencies are met, it will try to fetch various packages from internet. Note that, this process might take long time, depending on your internet connection speed. Once done, run following command

```
make kernel_menuconfig
```

This command is used to change software timer resolution to 1000 Hz. This is required to get 1ms software timer.

```
select Kernel Type --> Timer Frequency 1000HZ
```

Now it's time to create kernel image for RB433AH. We will create ramdisk image for network boot and tgz and ext2 for installing OpenWRT on RB433AH local flash drive. Note that this is the one time process. One can boot OpenWRT from local flash once it is installed. Type following command to create `ramdisk`

```
make menuconfig
select Target System --> Atheros AR71xx [2.6]
select Target Images --> ramdisk
```

save config file and exit from menuconfig and run

```
make V=99
```

Since you are executing make for first time, this may take long time to complete. If everything goes fine, it will create ramdisk image in `bin` folder.

```
[atlantis@localhost kamikaze_8.09]\$ ls bin/openwrt-ar71xx-vmlinux-initramfs.elf
bin/openwrt-ar71xx-vmlinux-initramfs.elf
[atlantis@localhost kamikaze_8.09]\$
```

Now to create tgz and etx2 run following command

```
make menuconfig
select Target Images --> tgz and ext2
```

save config file and exit from menuconfig and run

```
make V=99
```

This will create list of files in `bin` directory. We are now ready with two kernel image which we will use for installing OpenWRT on RB433AH.

A.1.3 Step 3: Installation of OpenWRT on RB433AH

To install OpenWRT on RB433AH, We need to setup `tftp` and `dhcp` server. Run following commands,

```
apt-get install atftpd
apt-get install dhcp3-server
```

now create `/tftpboot` directory and copy `ramdisk` image in to it.

```
mkdir -p /tftpboot
cp bin/openwrt-ar71xx-vmlinux-initramfs.elf /tftpboot
```

Start `dhcp` and `tftp` process by typing

```
/etc/init.d/aftpd start
/etc/init.d/dhcpd start
```

Assuming dhcp working properly, power on RB433AH. As explained A.1.2, RB433AH will boot from network and will get IP address from `dhcp server`. After booting up over ethernet, make sure RB433AH and your machine are able to communicate over ethernet. For this move ethernet cable in RB433AH from eth0 to eth1 and run following commands.

```
ifconfig br-lan down
ifconfig eth0 down
ifconfig eth1 192.168.1.10 netmask 255.255.255.0 up  // Assuming dhcp server is
                                                     // on 192.168.1.X segment
```

we need to reset the root password on RB433AH in order to transfer kernel image from dhcp server.

```
passwd    // type new password
```

We will now transfer kernel image to RB433AH. Run following command from machine running dhcp server.

```
 scp openwrt-ar71xx-vmlinux.elf openwrt-ar71xx-rootfs.tgz  root@192.168.1.10:/tmp/
```

At this point RB433AH has kernel image in its `/tmp` directory. To install OpenWRT on RB433AH run following set of commands

```
mount /dev/mtdblock1 /mnt/
mv /tmp/openwrt-ar71xx-vmlinux.elf /mnt/kernel
umount /mnt
```

```
mount /dev/mtdblock2 /mnt/
cd /mnt/
rm * -rf
tar -xzvf /tmp/openwrt-ar71xx-rootfs.tgz
cd ..
umount /mnt/
sync
```

At this point, RB433AH has OpenWRT on it's local flash drive and is ready to boot from it. As given in A.1.2 change boot device to flash drive from ethernet boot. Reboot RB433AH, it will boot from local flash drive and you will have openwrt prompt on console.

A.2 Setting up Serial PCI Card in Linux

Our testbed was having five RB433AH and each one was connected to PC through serial cable. For that, we attached serial PCI card on two machine. But to no surprise, they were giving us trouble in Linux. Below are the steps to fix it for all. Once PCI card is attached, run following command

```
lspci -v
```

It will give you detail information about all devices. Look for serial controller in it. E.g on my PC, I got output as

```
04:03.0 Serial controller: Unknown device 4348:3253 (rev 10) (prog-if 02 [16550])
Subsystem: Unknown device 4348:3253
Flags: medium devsel, IRQ 16
I/O ports at 1008 [size=8]
I/O ports at 1000 [size=8]
```

Note down `IRQ` and `I/O port addresses`. In my case, `IRQ` is 16 and two serial ports are at `1008` and `1000` respectively. Now type

```
setserial -g /dev/ttyS*   // install setserial if command is not found
```

Above command will list all serial port detected by PC. On my PC I got

```
/dev/ttyS0, UART: 16550A, Port: 0x03f8, IRQ: 4
/dev/ttyS1, UART: 16550A, Port: 0xbc00, IRQ: 20
/dev/ttyS2, UART: 16550A, Port: 0x1008, IRQ: 16
/dev/ttyS3, UART: 16550A, Port: 0x1000, IRQ: 16
```

Now the problem is, sometime there is `IRQ number` mismatch. To fix it, write down the `/dev/ttyS*` number corresponding to what we got while running `lspci`. For this example the mapping is,

```
1008 --> /dev/ttyS2
1000 --> /dev/ttyS3.
```

Now do appropriate changes as shown below, using `setserial` command

```
setserial /dev/ttyS2 port 0x1008 UART 16550A irq 16 Baud_base 115200
setserial /dev/ttyS3 port 0x1000 UART 16550A irq 16 Baud_base 115200
```

Now one can use serial device connected to `/dev/ttyS2` and `/dev/ttyS3` seamlessly. Moreover, one can add this command to `/etc/rc.local` to be executed at boot time.

Appendix B

Overview of Madwifi Driver

During the course of our project, we have modified madwifi driver at various places. The driver mailing list does give some help, but not to the point, where a naive user after reading it, can actually modify the driver. This Appendix helps in understanding madwifi driver in better way. The primary focus is on understanding transmit and receive path of a packet in *monitor mode*.

Our goal is to design TDMA system ensuring QoS guarantees. To accomplish it, we first need to disable various CSMA dependent functionality. As listed in [8], by setting card in *monitor mode*, one can disbale MAC layer acknowledgments, can send custom frame format and can disable RTS/CTS mechanism. Moreover, it also disable periodic beaconing mechanism.

Every packet received from network layer goes through various function in madwifi, before it goes on air. Every received packet from air also goes through series of checking, before it is handed over to network layer. There are few structure, that are frequently used in madwifi driver and they are `net_device`, `ath_hal`, `ath_softc`, `sk_buff` and `ieee80211vap`. Table B.1 shows use of that structures. Almost all structure has back pointer to get one from another. We will now look at functional flow of packet in both direction.

As shown in Figure B.1, every packet sent by network layer triggers the `ath_hardstart()` routine in madwifi. It then calls `ath_tx_startraw()`. In it, drivers prepare transmit de-

Table B.1 Important Structures in Madwifi

Structure	Description
struct net_dev *dev	One structure for each network device. Store all information regarding it
struct ath_hal *ah	Hardware Access Layer (HAL) API
struct ath_softc *sc	Stores software configuration parameter for device
struct sk_buff *skb	Each send/received packet is handled by it. `skb->data[]` contains the payload data
struct ieee80211vap *vap	Stores WiFi interface related statistics

scriptor for every packet using `ath_hal_setuptxdesc()`. One can change the transmit_rate, packet_type, tx_power for each packet. We have set *packet_type* of every schedule packet to be of `HAL_PKT_TYPE_BEACON` type. Since every beacon packet has hardware timestamp in it. By forcing schedule packet to be of beacon type, we were able get hardware timestamp on each schedule packet. We are using this timestamp for multi-hop synchronization mechanism.

After preparing transmit descriptor, `ath_tx_startraw()` calls `ath_tx_txqaddbuf()` to add packet in hardware transmit queue. Every hardware queue is defined by `tx->axq_qnum`. Once added to specific hardware queue `ath_tx_txqaddbuf()` calls `ath_hal_txstart(queue_number)` function to enable transmission of packets residing in given hardware queue.

In transmission side, `ath_hal_txstart()` is the last call from madwifi driver to instruct HAL to enable packet transmission. Once handle is given to HAL, it's up to the HAL to transmit the packet.

Referring to Figure B.1 again, every received packet from air generates RX (receive) interrupt, which is handled by `rx_poll()` function in `if_ath.c` file. Input to `rx_poll()` is `net_dev` structure or container of `soft_sc`. The `rx_poll()` does some processing on received packet and calls `ath_capture()`, which attache 26 byte header to received packet. Once header is attached, it calls `ieee80211_input_monitor()`, which again attach/remove some byte from header and pass packet to network layer by calling `netif_receive_skb()`.

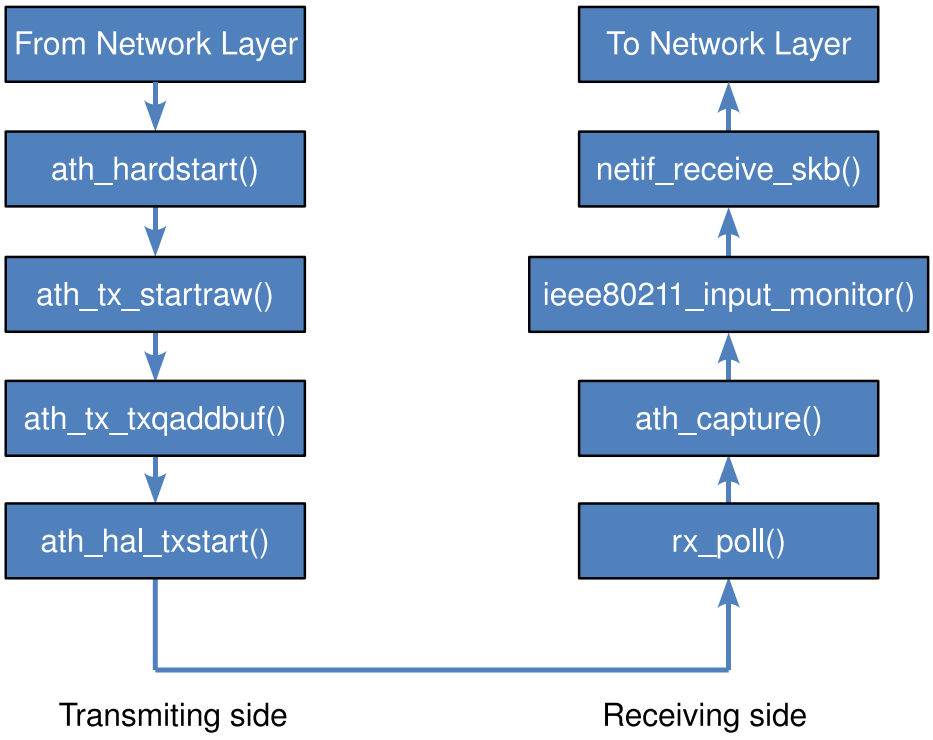


Figure B.1 Packet Flow in Monitor Mode

Appendix C

Code Explanation

Our implementation consists of three main modules. They are used for **queuing all incoming packets from network layer**, **for implementing TDMA slotting structure** and **to handle packets received from air**. Every module has set of helper function to carry out its operation.

TDMA Queuing Module: is responsible for buffering all incoming packets from network layer. On nodes transmission opportunity, it removes packets from the queue and sends it on air. In madwifi, all incoming packets from network layer comes to `ath_hardstart()` routine and goes through set of function as explained in appendix B, before going on to air. In our case, we required a place to buffer all incoming packets and send it only on nodes transmission slot. The way we implement it is, rather than calling `ath_hardstart()` for every incoming packet, we call `fractel_TDMA_add_to_buffer()` to buffer them in software queue. This queue is implemented as singly linked list with global head and tail pointers. We remove packets from the queue using `TDMA_send_triggered()`, which uses `TDMA_get_packet_params()` and `TDMA_remove_from_buffer()` to fetch required number of packets from TDMA queue. The `TDMA_send_triggered()` then calls `ath_hardstart()` routine for sending each packet on air. The data header is also attached to every packet in `ath_hardstart()`. The schedule packet type is changed to HAL_PKT_TYPE_BEACON in `ath_tx_startraw()` routine for hardware

timestamping.

TDMA Slotting Module: is implemented in `fractel_event_handler()` routine. Depending on slot type, it either sends schedule packet or calls `TDMA_send_triggered()` to send data packets. This function calls itself through software timer interrupt. In our implementation, We do not take any action in contention slots. The root node makes first call to it through `bootstarp_timer()`. All non-root nodes calls it through `fractel_busy_wait()` routine.

Received Packet Handling: is done in `rx_poll()` routine. The mikrotik board that we are using is big-endian and atheros hardware hardware-timestamp is in little-endian format. To handle it, every synchronization packet's hardware timestamp is first converted in to big-endian fromat in `rx_poll()` routine. The non-root node synchronizes with root node using schedule packet information. Every data packet has three possible flow in `rx_poll()` function and is explained in Function 9.

Function 1 `TDMA_send_triggered()`

Input: available slot transmit time and TDMA→head pointer

Output: send packets to `ath_hardstart()` for transmission. The number of packets transmitted is limited by slot interval.

```
1: calculate current transmission rate
2: while (TDMA_get_packet_params(&pcklen)) do
3:    calculate the time required to send this packet
4:    if (packet_transmit_time < slot_interval) then
5:       remove packet from queue by calling TDMA_remove_from_buffer()
6:       send packet on air using ath_hardstart()
7:    else
8:       my slot is over
9:       break
10:   end if
11: end while
```

Function 2 `fractel_TDMA_add_to_buffer()`

Input: a packet to be queued

Output: TDMA queue length + = 1 and tail pointer pointing to newly added packet

```
1: acquire spin-lock on TDMA queue data-structure
2: if (TDMA→head == NULL) i.e. it is a first packet to arrive then
3:    create TDMA queue
4:    allocate space for new packet
5:    modify TDMA→head and TDMA→tail pointers to point at newly added packet
6: else
7:    allocate space for new packet
8:    modify TDMA→tail pointer to point at newly added packet
9: end if
10: release spin-lock
```

Function 3 `TDMA_get_packet_params()`

Input: address of the variable to store length of the packet pointed by TDMA→head
Output: returns 1 if packet is found, otherwise returns 0

1: **if** (TDMA→head == NULL) **then**
2: return 0
3: **else**
4: write length of the packet pointed by TDMA→head in to the address passed as argument
5: return 1
6: **end if**

Function 4 `TDMA_remove_from_buffer()`

Input: address of variable to store removed packet
Output: returns 1 if packet is removed from queue else returns 0

1: **if** (TDMA→head == NULL) **then**
2: return 0
3: **else**
4: acquire spin-lock on TDMA queue data-structure
5: write packet content in address (passed as argument)
6: modify TDMA→head to point to next packet in queue
7: free the resources held by removed packet
8: release spin-lock
9: **end if**

Function 5 `fractel_prepare_schedule()`

Input: slot start time for multihop synchronization
Output: sends schedule on air

1: calls `fractel_create_schedule()`
2: `fractel_create_schedule()` will return new schedule for root node and stored schedule for non-root node.
3: send schedule on air

Function 6 `fractel_event_handler()`

Input: expected_next_slot_time, current slot number and TDMA→head pointer
Output: send Schedule or Data frames on air

1: **while** (current_time < expected_next_slot_time) **do**
2: nothing {busy wait}
3: **end while**
4: **if** ((current_time - expected_next_slot_time) > 100) **then**
5: reduce next slot size by 1msec {to compensate for late firing of current slot}
6: next_slot_interval = TDMA_SLOT_INTERVAL(INTERVAL - 1)
7: **else**
8: next_slot_interval = TDMA_SLOT_INTERVAL(INTERVAL)
9: **end if**
10: set timer for next slot using mod_timer(&fractel_event_timer, next_slot_interval)
11: slot_counter++
12: slot_counter = slot_counter % total_slots {we are using 3 control, 5 contention and 92 data slots, Total 100 slots}
13: **if** (my control slot) **then**
14: **if** ((I am the root node) OR (I am non-root node and has received schedule from my parent)) **then**
15: both node will call `fractel_prepare_schedule()`
16: non-root node will take stored schedule and modifies offset and node_field
17: root node will create new schedule by attaching schedule and routing tree header
18: both will then call `ic→ic_fractel_mgtstart()` to send schedule on air {the `ath_startraw()` will set packet type to beacon to enable hardware timestamping in schedule packet}
19: **end if**
20: **else if** (contention slot) **then**
21: do nothing {not yet implemented, might be used for nod join operation}
22: **else if** (my data slot) **then**
23: call `TDMA_send_trigerred()` for transmitting data packets on air
24: **end if**

Function 7 `ath_hardstart()`

Input: all data packet arriving from MAC layer
Output: sends data packet on air when called by `TDMA_send_triggered()` routine

1: take packet to be transmitted on air as input
2: attach data header to it
3: modify routing entry in data header
4: send it on air by calling `ath_tx_startraw()`

Function 8 `ath_tx_startraw()`

Input: data or schedule packet to be transmitted on air from `ath_hardstart()`
Output: transmit packets on air

take packet to be transmitted on air as input from `ath_hardstart()`
if (it is schedule packet) **then**
  set atype = HAL_PKT_TYPE_BEACON {to tell hardware to timestamp it}
  txrate=11 {transmit rate of 6Mbps for schedule packet}
end if

Function 9 `rx_poll()`

Input: all schedule and data packets arriving from air (through RX-Interrupt)
Output: take appropriate action depending on packet type.

if (valid schedule frame) **then**
  change byte ordering of hardware timestamp in received packet from little-endian to big-endian
end if
if (it is a Fractel schedule frame and i am a non-root node) **then**
  call `find_control_slot()` to check whether it is from my assigned parent
  set is_schedule_from_my_parent = 1 if received schedule if from my parent
end if
if (is_schedule_from_my_parent == 1) **then**
  synchronize to root node using information present in schedule packet
  activate tasklet for making initial call to `fractel_event_handler()`
  the call to `fractel_event_handler()` will enable slotting mechanism in non-root node
end if
if (it is a fractel data frame) **then**
  it has three path to choose from
  if ((it is broadcast_packet) OR (i am the expected receiver and the end destination of this packet)) **then**
    remove data header and pass it to network layer
  else if (I am the next hop for this packet and I am not the end destination) **then**
    remove data header
    add packet to TDMA queue by calling `fractel_TDMA_add_to_buffer()` for multihop transmission
  else
    Not my data packet
    kill it and reclaim the resource
  end if
end if