
# Prototyping a Packet Scheduler from Time Driven Priority Network to 802.11 Access Network

Master of Science Thesis*

**Dipankar Biswas**

KTH, Royal Institute of Technology, Stockholm, Sweden.
November, 2006

Examiner/Advisor: Dr. Johan Montelius
Supervisor: Prof. Yoram Ofek
Prof. Renato Lo Cigno

* This work was a part of European Union R&D project, IP-Flow, supported by Marie Curie Chair Excellent (EXC) Actions at University of Trento, Italy.

## Abstract of The Masters Thesis

Internet traffic continues to grow exponentially due to steady expansion of its service areas and it is foreseen that it will be dominated by stream media flows, such as, audio/video telephony or conferencing, distributed gaming, virtual reality and many others. Additionally, since data and telecom network are merging to the dream trend '*all-IP'*, the use and the presence of 802.11 network is expanding beyond corporate offices and hotspot to home users and is becoming one of the access networks of choice. So there is a real need to improve forwarding scalability of IP packets to provide Quality of Service, especially for stream and real-time traffic from core network to mobile user. A lot research is being carried out in this field from data link layer to application layer. However, very few have researched the use of '*global time'* to solve the stated scalability problem. It has already been realized, implemented and experimented that (universal coordinated time) UTC-based packet forwarding is able to solve the scalability issue.

This thesis endeavors to find an optimal and cost-effective solution for the wireless extension of time-driven packet forwarding to the 802.11 network. It also aims to implement the idea and divulge the experimental results. This work presents a kernel based prototype solution of synchronous scheduler for 802.11 network for an access network interface to time-driven network. It has been implemented directly in kernel space of Linux operating system that manages network layer and partially MAC layer of an Access Point.

The problem is of great complexity due to the non-modifiable device dependent routines that manage MAC and PHY layer of 802.11 stack and unavailability of device specification from vendors. However, this work has devised and implemented two versions of packet scheduler. First one is open-loop that shows only plausibility of synchronous time-driven scheduling but experimented that, it is hard to implement on existing hardware. The second one is close-loop approach, where the local clock generated by the access-point is aligned periodically with the UTC-based time from the externally connected time-driven network. It's feasible to implement this approach on existing hardware.



**Keywords:** IP packet switching, 802.11 wireless network, time-driven network, network layer protocol stack of Linux kernel.

# Acknowledgements

At the outset I would like to thank Prof. Yoram Ofek, Renato Lo Cigno and Dr. Johan Montelius for giving me the opportunity to carry out this research project. Their encouragement and advice during the entire process is appreciated. I would like to specially thank to the phd student of the project Yury Audzevich for his help, suggestions, feedbacks and support without which this thesis would not have been possible. I am grateful to the members of the *IP-Flow* project for making my stay a memorable one. It was a pleasure working with the team. My sincere thanks to all those generous individuals of different open source project who replied promptly to my queries. Finally, I would like to thank my parents for their constant support and encouragement.

# Table of Contents

## List of Figures:

## List of Tables:

## *Chapter One:*

# Introduction

Internet traffic is growing steadily, about doubling in every year. Online interactive entertainment, multimedia streaming, voice/video/TV over IP, various real-time application are already deployed up to the massive number of end-user. Now it is expanding services. So service-wise and traffic-wise growth of internet is yet to come, which will be dominated by distributed 3D gaming, high quality video-conferencing, virtual reality and many more application and services. These streaming multimedia applications need not only high speed data transferability, but also to be ensured Quality of Service (QoS).

On the other hand, telecom industry and data network industry are merging to the dream-trend *"all-IP"*. Moreover, mobile devices are spreading their versatility and usages in multi-tier information system in heterogeneous network platform to the telecom service provider and manufacturer. For example, *Skype* is on your mobile hand device. So wide deployment of wireless LAN, wireless MAN that enable omnipresent service provisioning to mobile user anywhere, any time, in any context.

So there are lots of issues to the research people to solve the problem of immense growth of Internet traffic over wired network as well as wireless network. In particular, there are two issues concerning IP packets forwarding [5]:

- `routing`- determining the path a packet travels on from source to destination, and
- `flow control`- how a packet is forwarded (or stored), primarily with respect to universal time, along with the selected destination path.

Timing and flow control of IP packet over the Internet are important and more critical issue especially when IP packet is forwarded (or stored) between wired and wireless environment back and forth primarily with respect to time, along with the selected path. It is already realized, implemented and experimented that, to use ***`global common time reference`*** (CTR) derived from GPS (global positioning system) to control flow of IP packets and forwarding them over wired network is more

beneficial than traditional *'only'* destination address based forwarding [1],[2],[3]. UTC-based (Universal Coordinated Time) *pipeline forwarding* [4] for IP packet scheduling, flow control is particularly suitable for stream application, since it:

- guarantees a maximum per-hop queuing delay below one ms, independent of the flow rate and the network load, also in bandwidth mismatch points [4];
- is already realized and implemented efficient packet switch architecture that increases the scalability of switches and eliminates the electronic switching bottleneck [2].
- guarantees Quality of service (deterministic delay and jitter, no loss) for (UDP-based) constant bit rate (CBR) and variable bit rate (VBR) streaming applications — as needed, while preserving the TCP-based "best-effort" traffic(compatible with existing application) [1],

In general, we called this network as *time-driven network*. Maximum benefit can be achieved form UTC-based *pipeline forwarding,* if it is deployed in the end-to-end user basis. Due to enormous growth of wireless LAN, the number of mobile end users is growing exponentially. Moreover, UTC with pipeline forwarding can solve the bandwidth mismatch problem between high capacity core network optical/ethernet and low speed wireless network, which is a link bottleneck problem. Therefore it should be worth highlighting to take advantage from time-driven pipeline forwarding to wireless network for IP packet forwarding and experiment the performance. Figure 1.1 shows the general overview of architecture for deploying time driven network with wireless extension.

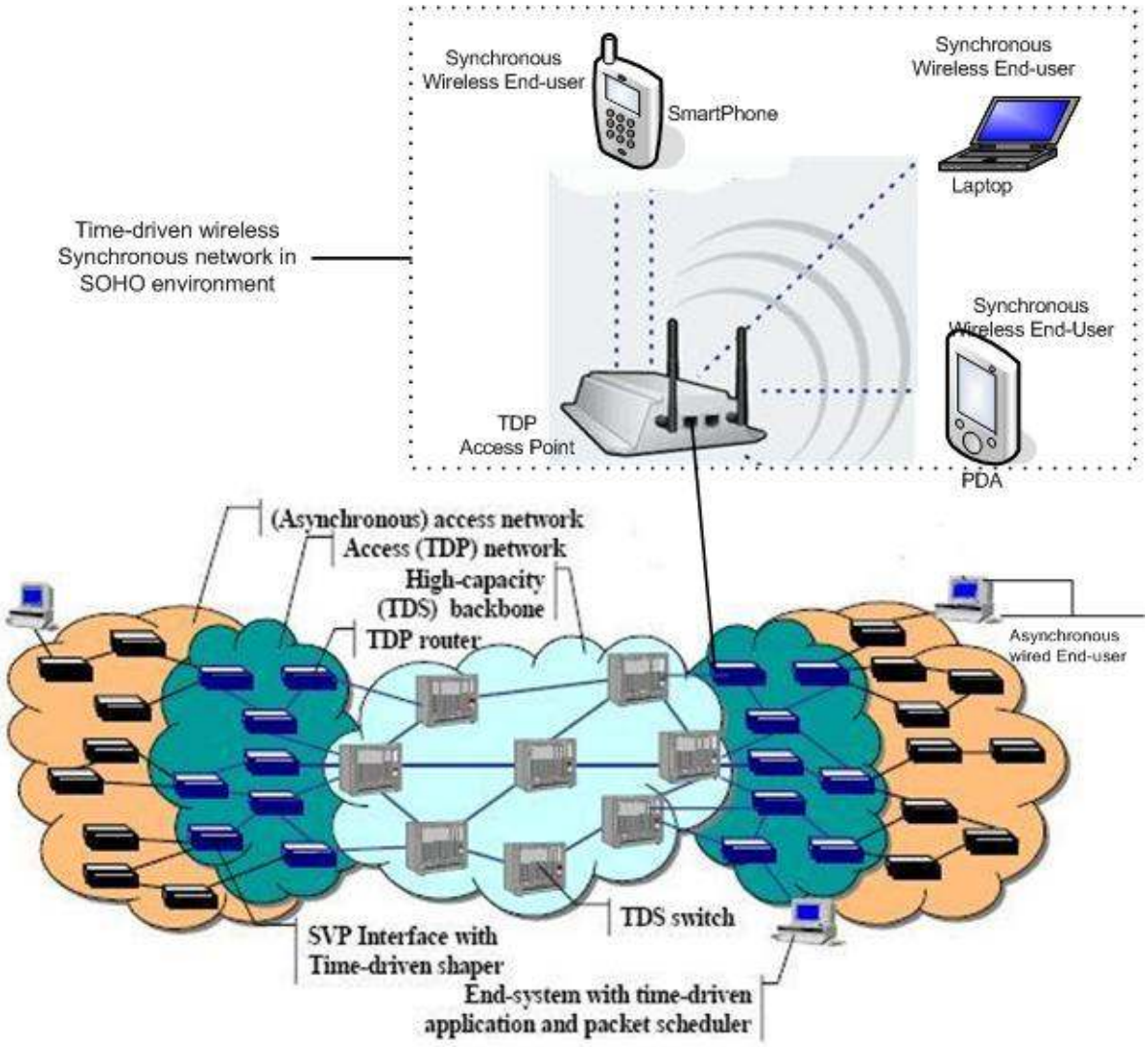



Figure 1.1: Architecture for deploying wireless extension of Time Driven Switching (TDS) network.

All switches, a packet forwarder, in time-driven network require UTC time directly from GPS. So it would be simpler solution to use a GPS receiver in an access point in wireless network (802.11, Infrastructure). Unfortunately, that will be expensive in comparison with current market value of AP.

.This thesis work endeavors to find an optimal, cost-effective solution for wireless extension of time-driven network, implement the idea and divulge the experimental result. This work presents a kernel based prototype solution of synchronous scheduler for 802.11 network for an access network interface to time-driven network. It has been implemented directly in kernel space of Linux operating system that manages network layer and partially MAC layer of an Access Point.

The report is organized such a way that, next chapter describes background of the project, i.e. basics of underlying technology `time-driven switching network` and existing timing principle of wireless network (IEEE802.11), chapter three views theoretical idea of proposed approaches, chapter four provides details of prototypal implementation, chapter five evaluates the performance of current implementation followed by (in chapter six) a critics- the difficulty we faced, how far this work meet the goal and what to do next.

*Chapter Two:*

# Background

This thesis work is based on an innovative concept to use *'time'* to forward packet with existing network architecture. So it would be helpful to understand the basic principal of the underlying technology. This chapter also gives a sufficient background to understand following chapters. Experienced reader may safely skip some sections.

## 2.1 Underlying Principle of Time-Driven Network

Internet traffic can be synchronized, minimum delay bounded, congestion free by using *UTC-based pipeline forwarding* of IP packets. Underlying idea is following.

### 2.1.1 Common time reference

In time-driven networks all switches maintain a common time reference *(CTR)* typically aligned with UTC (coordinated universal time) [1] [2] [4]. The granularity of the CTR is refereed as *time frame (TF)* of predefined fixed duration (typically between 12.5 µs and 125 µs) and the TFs are used to schedule packet forwarding from all sources throughout the network. Note that different links can have different TF duration (e.g., 12.5 µs for high capacity links and 125 µs for low capacity ones), for example, for forwarding to wireless link it can be used longer TF. Packets are not transmitted in a specific time, but rather within this predefined TF. Thus, this method is called pseudoisochronous packet switching.

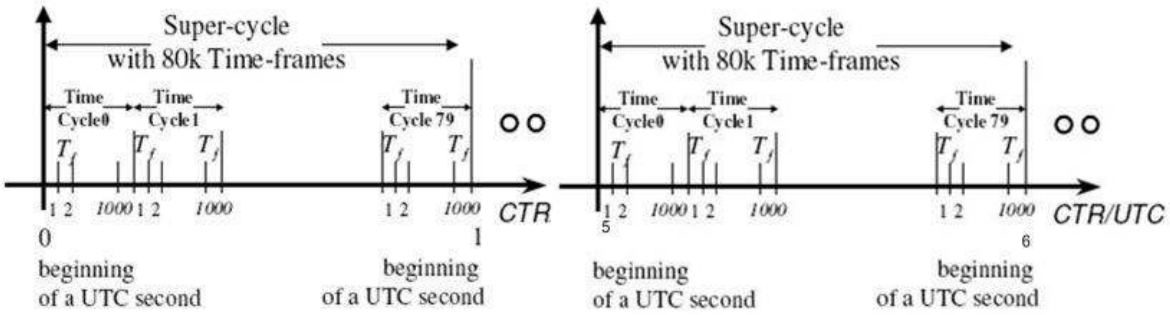


Figure 2.1: Definition of the common time reference [4].

The CTR is organized in the following manner: *k* TFs are grouped into a *time cycle(TC)* and *l* contiguous TCs are grouped together into a *super cycle*. A typical duration of a super cycle is one UTC second, as shown in Figure 2.1 (for $T_f$ = 125 μs), with *k* = 100 and *l* = 80.

The underlying principle of forwarding packet using CTR derived from the idea pipeline forwarding, used in computing, manufacturing, which is called *UTC-based pipeline forwarding*. In UTC-based pipeline forwarding, all switches, getting CTR from GPS, is forwarding packet utilizing TFs in a sequential, synchronized and increasing manner of TFs. Two implementations of the pipeline forwarding were proposed: Time-Driven Switching (TDS) and Time-Driven Priority (TDP) in [1] [2]

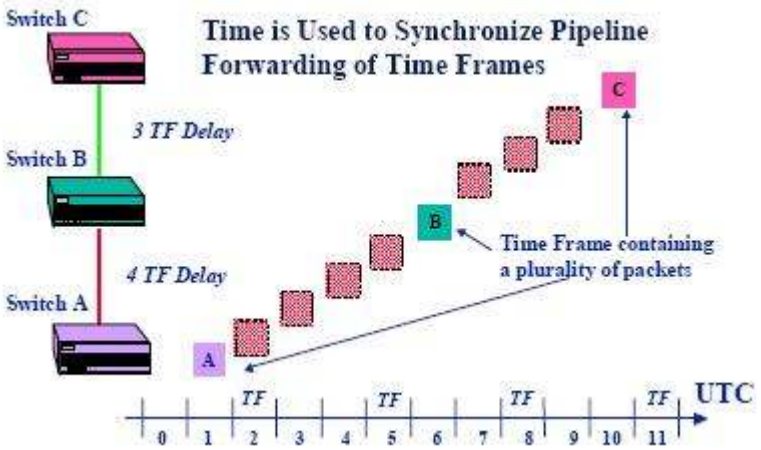


Figure 2.2: UTC-based pipe line forwarding [2]

*Time-driven switching* (TDS) was proposed to realize sublambda or fractional lambda switching (FΛS) in highly scalable dynamic, high speed (optical) backbone networking with minimal optical buffer. In TDS all packets in the same TF are switched in the same way.

### 2.1.2 Packet forwarding with Time-Driven Priority

Wireless extension of time-driven network, at the edge of the network, requires flexibility, e.g. conventional IP destination based routing. *Time-driven priority* (TDP) is a synchronous packet scheduling technique

that implements UTC-based pipeline forwarding and can be combined with conventional IP routing [2][4].

Packets are forwarded along TDP switches one hop every TF, as shown in Figure 2.3, for example a video frame. This figure shows very particular case. General idea is following:

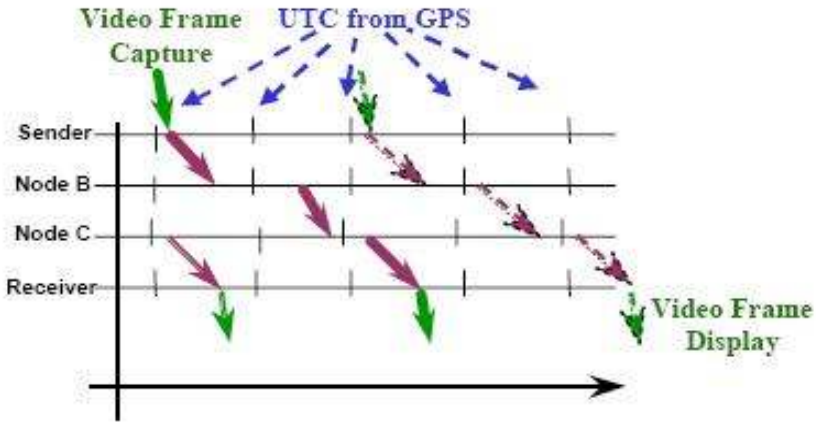



Figure 2.3: IP packet forwarding with TDP in the case of constant delay forwarding [1]

During each TF, one or more packets can be transmitted; for example, if $T_f$ = 125 μs and the link capacity is 1Gb/s, about 300 ATM cells can be transmitted in every TF. If all packets that must be sent during TF *i* by a node are in the correct output port of that node before the beginning of that TF, and the delay between an output port of one node and the output port of the next node is a constant integer multiple of $T_f$,, referred as `forwarding delay`, the traffic in the network is said to be `TDP paced or shaped` and these two conditions are the main points that allow TDP to control the delay experienced by the packets in the network.

It is easy to understand that a resource reservation protocol is needed by this queuing algorithm to keep these conditions true. Reserving resources for a connection requires solving a scheduling problem to find a feasible sequence of TFs, called *schedule*, on links on the route from source to destination. TDP needs the establishment of virtual circuits over the network, but as well as where sending a packet of a particular flow, TDP switches must know when sending it. There are three different forwarding schemes to choose when sending an incoming packet (*k* is the number of TFs in a TC):

- `immediate forwarding`: packets arriving at an output port in TF *i* must be sent out in TF (*i*+1) mod *k*;

- *2-frame choice*: packets arriving at an output port in TF *i* can be sent out either in TF (*i*+1) mod *k* or in TF (*i*+2) mod *k*, but the choice must be the same for all the next packets of the same flow that will arrive at this output port in TF *i* in the next TC;
- *arbitrary-frame choice*: packets arriving at an output port in TF *i* can be sent out in any of the TFs (*i*+1) mod *k* to (*i*+*k*) mod *k*, but the choice must be the same for all the next packets of the same flow that will arrive at this output port in TF *i* in the next TC.

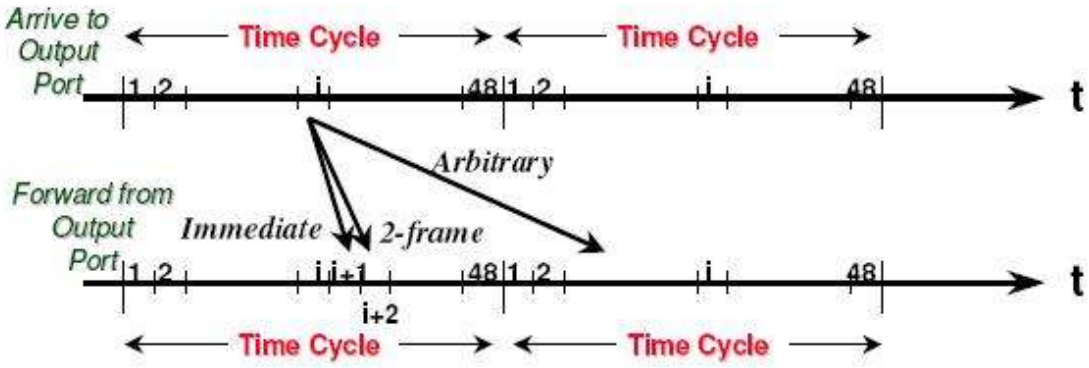



Figure 2.4: Packet forwarding scheme, immediate, non-immediate and arbitrary forwarding [2].

It would be, therefore, clear the main implication of TDP forwarding: the number of TFs it takes for a cell to be forwarded from one node to the next node is predefined in a deterministic manner and, in order to do it, the forwarding schedule of a given connection must be determined during set-up and must be kept fixed for the connection duration. TDP forwarding with the proper resource reservation therefore provides QoS guarantees in terms of bandwidth, constant bound on delay, jitter of one TF (because packets could be transmitted in a period of time that goes from the beginning to the end of a TF) and no loss due to congestion for real-time traffic. In the same time a best-effort strategy is possible: best-effort packets can be transmitted anyway with lower priority during any unused part of any TF. Furthermore, large best-effort IP packets can be sent during multiple TFs in which case the packet will be fragmented by a time-driven nondestructive preemptive priority.

## 2.2 TDP Interface of Time-Driven Wireless Network

Figure 2.5 shows end-to-end prototypal setup of a typical time-driven network with wireless extension. All switches in the network are getting

UTC time using attached GPS receiver. However, in wireless extension, AP doesn't have any GPS receiver due to non-feasibility of market value of AP. But we believe that, it is certainly still possible to make wireless network (802.11) time-driven and implement TDP packet scheduling if we consider TDP router as *timing master* of wireless network. AP which is directly connected with TDP router, eventually distributes the timing information among the associated wireless client. We referred the AP that performs TDP scheduled packet forwarding as *TDP Access Point (TDP AP).*

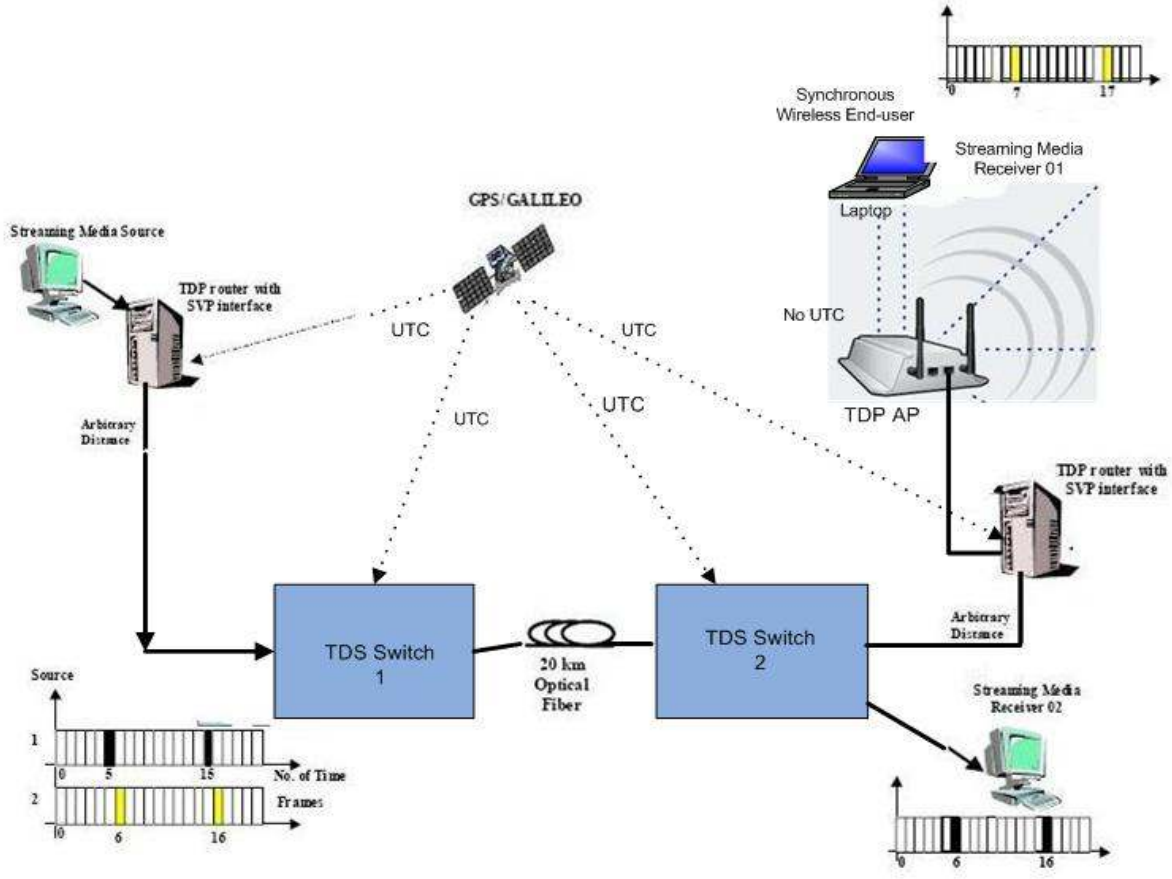



Figure 2.5: End-to-end prototypal test bed setup with time-driven wireless extension

A prototypal implementation of a *Time-Driven Priority (TDP)* router has already been realized for shaping packet forwarding inside the time-driven network [23] and integrated with TDS optical backbone. It experimented the performance in [1]. TDP router is realized using a personal computer with FreeBSD operating system. Details of hardware configuration can be found in table 4.1. Although a non-real time architecture such as a PC is used to

implement TDP scheduling, it already experimented that, TDP router with SVP interface can provide deterministic QoS over a packet switching network, with a bounded end-to-end delay and jitter. Furthermore, it also minimizes the buffering requirements inside networks switches, while ensuring no packet loss due to congestion.

TDP router is to be considered as timing distributor of the *Time-Driven Wireless Network* (TDWN), should send a special packet called *delimiter* aligned with UTC time to the TDWN at pre-defined time interval. Delimiter may either

*(i)* contain timing information, e.g. as timestamp value equals to the time when delimiter packet start to send output interface buffer of TDP router or

*(ii)* be empty body frame as timing indicator that will represent as TDP timer within AP.

All APs connected with TDP router '*simply*' accept the timing information of delimiter and adjust (synchronize) its own time or consider as time to forward timing value to all associated wireless client. Delimiter may be implemented as either:

*i)* a special control/management frame, or

*ii)* TDP shaped ordinary UDP packet since delimiter doesn't need any acknowledgement.

In the current implementation of TDP router, former method is taken on, i.e. TDP will send empty bodied delimiter as an ordinary TDP shaped UDP packet at a pre-defined interval called *delimiter interval*. So TDP AP simply and safely accept the delimiter as timing indication to adjust its own timer by measuring drift or just forward timing information to all associated. However, it is important to understand that, since delimiter doesn't contain any timing value, TDP AP can not just adopt the time from the received delimiter packet. TDP classifier categorizes delimiter by using *Differentiated Services (DS)* field [27] (previously known as type of service (TOS)) of IP header [23]. Four unallocated DS codepoints are used for identify TDP packets, but note that the DS codepoint **xxxx11xx** identifies a TDP packet, other bits that distinguish odd/even TFs and odd/even TCs. Therefore, it should be easy to distinguish delimiter packet by TDP AP.

## 2.4 Time Synchronization of IEEE802.11

This section reviews the state-of-art of timing synchronization of 802.11 network. We limit our discussion in only infrastructure mode

as this is the initial stage to introduce time-driven priority to the wireless network.

Like other wireless network, 802.11 depends on the distribution of time information to all stations which is used by the medium reservation mechanism and other purpose as well. A `time synchronization function` (TSF) keeps the timer for all stations, in addition to a local TSF timer for each station. The TSF timer (a modulus $2^{64}$ counting, 8 byte) is based on a 1-MHz clock and increase in *'ticks'* µs In infrastructure network, the Access Point (AP) will be timing master, responsible to transmit a special management frame called beacon periodically containing *"now"* in a timestamp (TSF timer). AP should set timestamp value in beacon frame so that, it is equal to the TSF time at the time when first bit of timestamp field hits the physical layer plus the transmission delay from its MAC-Physical interface to its physical interface with wireless medium [21]. This timing accuracy is so rigid that, 802.11 standard defines to maintain the synchronization of TSF timer with all stations in a BSS within 4 µs plus maximum propagation delay of physical layer.

Every station associated with should simply accept the time received beacon frame sent by AP, If a station's TSF timer different from received timestamp, should update its local timer, but they may add a small local offset time. The interval at which AP sends beacon frame is referred as `beacon interval` that measured in Time Unit (TU) which is equals to 1024 µs. Beacon interval is also included in beacon frame.

*Chapter Three:*

# Theory of Proposed Approaches

This work is the very initial stage research- how to introduce UTC-based pipeline packet forwarding concept in the wireless network and experiment the result. We have proposed two possible approaches to achieve the goal. Section 3.1 and 3.2 describes approaches in a more *'general'* way, however, it can be found more specific description in section 3.3 for 802.11 network considering the implementation feasibility on the existing hardware.

## 3.1 Open Loop Approach

Most of wireless network depends on timing distribution since medium can be shared among the nodes on the basis of time. Moreover, one node of the network should act as timing master (as central coordinator point) or all node exchange timing information to each other. The node (for example, coordination point) that directly connected with TDP interface should act as timing *distributor* among the all nodes in the wireless network. We called the node connected with the TDP interface as `TDP node`. Each TDP node has its local timing function, e.g. its local clock.

The main idea of open loop solution is the central coordinator (for example in 802.11, infrastructure, access point) of TDWN is designed such a way that, internal clock is initialized and incremented with TDP timer remotely. Virtually, it is expected that, there will be no propagation delay to get timing signal from TDP router, like an electrical signal.

## 3.2 Close Loop Approach

Idea of the close loop method is similar to the `time adjustment process` of a station in infrastructure mode of 802.11 network. In the case of OLA, a TDP node should adjust or align its own clock after receiving delimiter from TDP interface *'if necessary'*. It important to define, when it is necessary to align own clock timer with TDP timer and

how TDP timer can be implement within TDP node. TDP timer can be implemented within TDP node by following ways depends on content of delimiter, for example:

*i)* if delimiter contains TDP timing value TDP timer of TDP node will be added value of TDP timestamp with local offset of TDP AP, see figure 3.1

*ii)* in the case of empty bodied delimiter, receiving a delimiter cause increment TDP timer one unit depends on local timer of TDP node incrementing in which unit and delimiter is receiving at which interval.

Now it should explain when it necessary to make alignment. Since in the TDP node there will be two timer, one its local clock timer, another is TDP timer. It is important to point out that, the TDP timer represent UTC time in the TDP getting via delimiter. The purpose of close loop method is to synchronize between two timers. There should be drift between two timers which is measured at each delimiter arrival time. It may be problematic and time consuming to make an alignment. Moreover, it is still possible to implement TDP traffic scheduling drift between two timers don't cross a certain time. We defined this upper limit of the drift as *drift threshold*. Hence when the amount of drift is just crossed drift threshold.

Close loop method is not ideally time-driven as this method lives always with a small drift. However, this drift never be greater than drift threshold. That's why this method may be referred as pseudo time-driven method.

## 3.3 Modification Needed by Time Driven Wireless Network (TDWN)

If it is possible to send timing information from TDP router to TDP AP using lower layered frame exchanging, more timing accuracy can be achieved. In the current implementation of TDP router, experimented to send timing information using an ordinary data packet, UDP packet, which goes into the inherited upper layer processing and adds some software latency. However, the initial research of TDWN, for experimental purpose indeed, we are interested to describes the both approaches in consider with existing network architecture and network device that can enable to implement the solutions to make wireless network time-driven. As the research on TDWN is very initial stage, we only consider 802.11 network (infrastructure mode)

### 3.3.1 For Open Loop Approach

We have reviewed (section 2.4) the state-of-the-art time synchronization of IEEE802.11 (infrastructure) wireless network that reveals AP maintains time synchronization functionality (TSF) with its associated station by sending a beacon frame (special management frame) periodically. To implement OLA in 802.11, it is necessary to control sending beacon frame using TDP timer remotely instead of TSF timer. So TDP AP should send beacon frame as soon as it gets the delimiter. Virtually difference between these two times, when delimiter has received by the TDP AP and time when beacon frame start to send, should be (tends to) zero. AP (WLAN chipset) suppose to set timestamp value at the time when first bit of beacon frame hits the physical layer of the device to transmit to the air. On the other hand, stations simply adjust it own local TSF time with timestamp value received from beacon. Hence, stations suppose to get the network time *'perfectly'* aligned with UTC time. But practically it is not possible to keep the time difference between delimiter receiving time and start to send beacon frame to (tends to) zero. It may be useful to add some offset referred as `local offset` determined by TDP AP This procedure can be useful only in case if delimiter contains timing value when start sending the delimiter packet by TDP router.

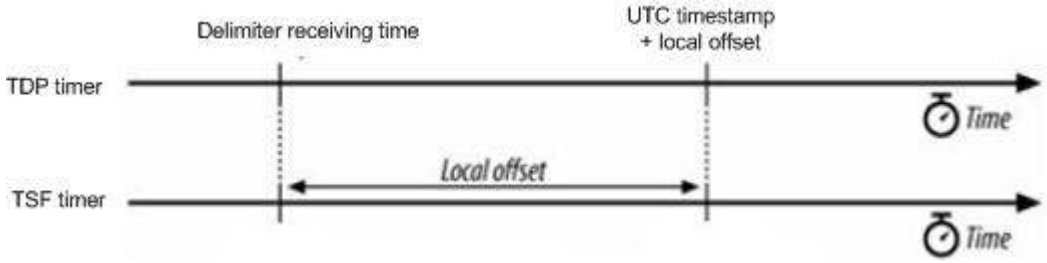



Figure 3.1: local offset adding

Although it is not possible to get device specification of WLAN chipset from respective vendor, but what we have learned from the experiment and open source community that, most of the chipset control (by firmware) time-critical MAC functionality like beacon transmission, since 802.11 standard stressed to maintain timing accuracy to sending beacon frame within 4 μs plus propagation delay of MAC-Physical layer interface [21]. This is certainly hard to maintain by device driver software. Most of the WLAN device contain different hardware transmission queue for different kind of traffic. Generally for beacon frame, AP also may use a special queue only for beacon. So it important to set chipset to transmit beacon at *'exact'* time if it is already queued into the beacon queue.

### 3.3.2 For Close Loop Approach

It is easy to implement CLA on the exiting hardware that's design was followed 802.11 standard. In the this method we are trying to align TSF timer with TDP timer but device is still controlling to start sending beacon frame. TSF timer of the device is increasing every μs. that implies, granularity of TSF timer is μs. If we configure TDP router with delimiter interval is (e.g.) 1 ms, arriving event of every delimiter causes 1000μs increment of TDP timer. At each beacon frame transmission time, TDP AP needs to measure the drift between TSF timer and TDP timer when drift is greater than drift threshold, according to the method, an alignment of TSF timer with TDP timer is needed.

It is not possible to reduce or increase time value of chipset clock, but alignment can be possible if we can shift ahead or backward beacon transmission equal to drift threshold then alignment will be success.

We have called close loop solution as pseudo time-driven method, as this method is not ideally time-driven with TDP timer. Once an `alignment` happened, sending beacon frame 'aligned' with TDP timer with some μs before or after because of drift. However, this difference will never cross `drift threshold`. This phenomenon was described in figure 3.2 and 3.3.

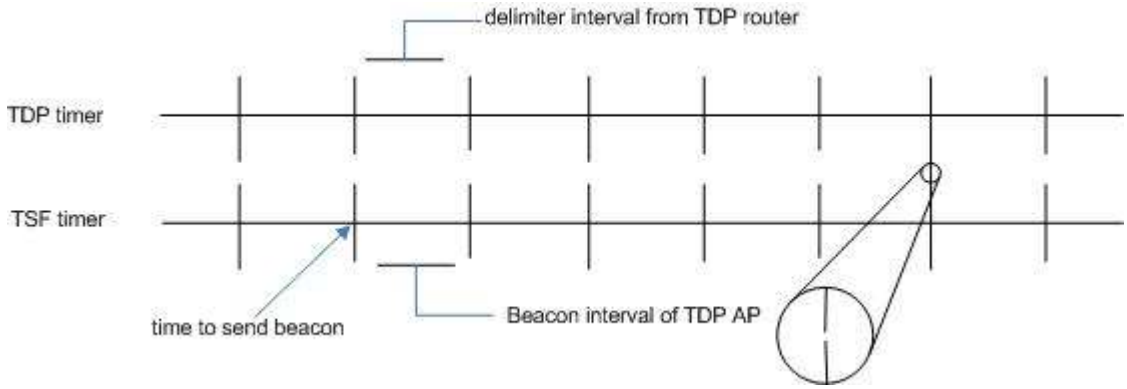



Figure 3.2: Ideally TDP time and beacon frame sending time line

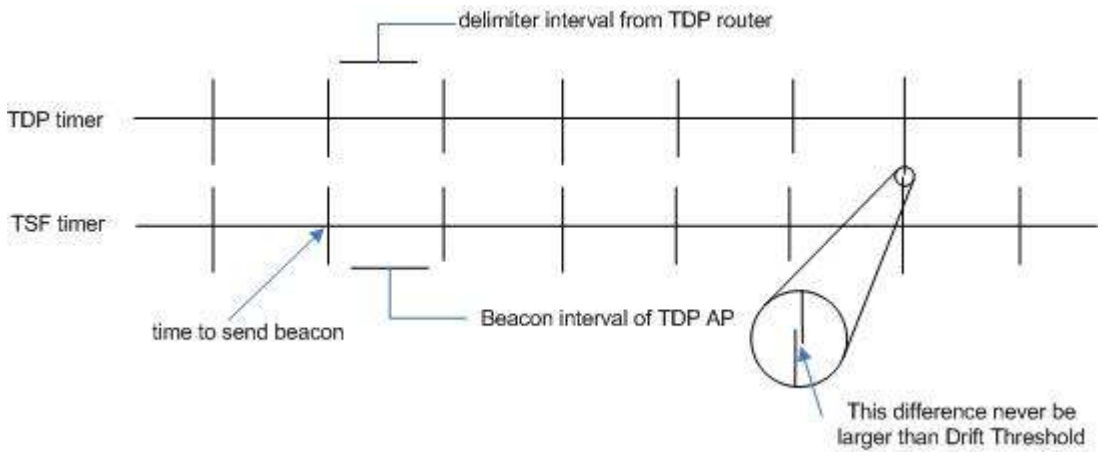



Figure 3.3: Pseudo time-driven of CLA

*Chapter Four:*

# Implementation and Platform

From the chapter two and three, we have studied sufficient theoretical description of the problem and possible and proposed solutions. This chapter describes the implementation details, the development platform, utility and environment in which the implement and experiments were carried out.

## 4.1 Introduction

Before going to details description of implementation, we are interested to focus on motivation and reason to choose the implementation environment. There was sufficient time had been spent in the study period of this thesis work to choose appropriate software platform and hardware to implement this project , since almost no vendor is kind enough to open source community. We needed to consider the availability, chipset vendor specification and financial support that project can provide and some trade off between several alternatives.
We have studied following alternatives:

### a. OpenWrt:

OpenWrt [7] is Linux system distribution for embedded device e.g. Wireless Router or Access Point that runs on Linux kernel. It provides fully writable filesystem, JFFS2, along with read-only (embedded in the firmware) known as SquashFS. This fully writable filesystem and package management facility enables developers and users application selection and configuration provided by vendor as well as allows you to customize and add functionality through the use of existing packages and added packages to make suit any preferred application. OpenWrt, a framework, that allows developers to full customize and develop packages to enrich feature without building complete firmware around the wireless router or AP, those functionality was not envisioned by the vendors.

Initial stage of this project work, we have studied the feasibility of OpenWrt as the development platform to implement the idea as described in section 3.1 and 3.2 , by developing a package and ultimately cross compile for the target device (see chapter 6 ) Shortly it has been realized

that, implementation of the proposed solution requires to access to some lower level driver code of wireless LAN chipset which has vendor proprietary and licensing issue. But OpenWrt is certainly important and necessary platform for the future stage of this project work.

**b. HostAP:**

Host AP [8] is a Linux driver software as kernel module for wireless LAN cards based on Intersil's PRISM® ( 2, 2.5, 3 ) WiFi chipset [9]. This driver was developed basically to support master mode ( AP mode) without having any special manufacturer provided firmware for wireless LAN card along with its normal station operation in BSS and also in IBSS. Host AP endows with functionalities in the host computer or embedded system required to initialize, configure, attach, de-attach Prism based cards to transmit and receive frames and to gather statistics. It also allows bridging through the regular Ethernet bridge driver of Linux kernel which might be useful for the implementation of this project work (see section 4.3.1). Moreover, it includes most of IEEE802.11 management and control functions such as authentication and de-authentication, association with re-association, disassociation, power saving mode operation, frame buffering for power saving stations. Although, this driver code has still lack of development debugging, accessing hardware configuration records, I/O registers. This driver software has been designed and optimized such a way, it can be used as kernel module to customized Linux kernel for embedded system.

According to the latest release notes [8], however, the firmware of Intersil's PRISM® chipset for station (supplicant) takes care of `time critical` features of IEEE802.11 protocol stack, e.g. beacon frame sending, frame acknowledgement. As described in section 3.3, in both approaches, OLA and CLA, basically we need to control over sending beacon frame from AP in shifted or adjusted time. This "`time critical`" task is mostly built-in inside the firmware of the device. There fore, choosing Hos AP driver as the implementation framework of this project was in question.

**c. Bcm43xx:**

We already know that, implementation of this project will be sufficient modification and adding of code that control lower MAC layer for particular wireless LAN chipset. No vendor kind to open source community. `bcm43xx` is Linux kernel driver platform for Broadcom `bcm43xx` wireless chipset. This driver is based on reverse engineered specification of binary release of world leading wireless chipset provider, Broadcom after refusal to release any specs of their chips or any code of

their driver. Open source community reverse engineered Broadcom bcm43xx chipset family by analyzing disassembled code from other Braodcom driver , partially hand translated assembly to C followed by better understanding hardware and a creative MAC-On-Linux hacking to allow PCI proxying [10].

Compare with leading WLAN chipset provider Atheros provided Madwifi, closed hardware access layer (HAL), `bcm43xx` should be more useful platform to implement two solutions for this project. We would have chosen this platform, however, while I started to implement, `bcm43xx` was not succeeded to work with.

## 4.2 Hardware

The implementation environment consists of a testbed containing two TDP router as interface of TDP wired and wireless network, TDP AP (with linux kernel). Table shows a quick view of test bed. ( see figure 5.4 experimental setup)

| Name | Work as | Processor | Main memory | Operating system (kernel) |
|---|---|---|---|---|
| TDP router | Network interface of TDP network | Intel® Xeon(TM) 2.8 GHz, 4 processors in each router | 2 GB of RAM | FreeBSD 5.3 |
| TDP AP | Access Point of TDP wireless network | Atheros WLAN chipset, Intel® Pentium4 (system µp) | 1 GB of RAM | Linux 2.6.12 |
| Station | Synchronous wireless client (as source or destination ) associated TDP AP | Intel® Pentium 4 3.06 Ghz (notebook) | 1 GB of RAM | Linux 2.6.12 |
| Fluke network packet analyzer | Synchronous wireless client associated with TDP AP to capture and analyze frame | Intel® Pentium M 1,1 GHz | 504 MB of RAM | Windows XP |

Table 4.1: Implementation and experimental hardware description

### 4.2.1 Atheros WLAN Chipset as AP

We have chosen madwifi (see section 4.3.2) driver as our implementation platform. This driver, lives in Linux kernel space, is for WLAN device based on Atheros WLAN chipset. In particular we have

selected Proxim WLAN card based on Atheros AR5212 chipset act as access point. So it would be good for the reader having some knowledge of chipset specification

Atheros AR5212 chipset support 802.11a,b,g is the second generation chipset derived from initial AR5210 which has been first full 802.11a standard.

Atheros WLAN chipset has multi-protocol MAC or baseband processor supports Radio-on-Chip (RoC) that can operate dual band 2.4/5 GHz. The radio modem use OFDM in 5 GHz band in 8 different channels with throughput of up to 54Mb/s rates (depending of range).

The Atheros chipset has some proprietary features, e.g. Atheros turbo G mode (super AG® mode) that allows to make twice of bit rate (say 108 Mb/s) by using two channels in parallel. Downside is, it helps to increase sensitivity of interferences, certainly decreases number of channels and lead to incompatible 802.11a. Host interface of the chipset are MiniPCI, Cardbus ( 32 bit PCMCIA), PCI with direct DMA access.

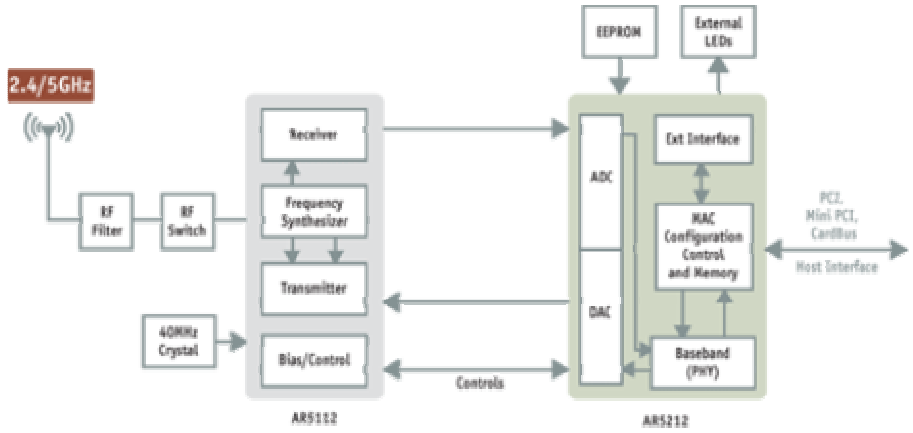



Figure 4.1: Atheros 5002X chipset architecture with AR5212 [22]

This chipset support wireless multimedia extension (WME), defined in IEEE802.11e standard, Quality of Service (QoS) enhancement that consider delay-sensitive application, voice over wireless IP by extended 802.11 MAC layer [11]. It can manage additional hardware transmit (TX) queue for WME classified traffic, e.g. voice over wireless IP (VoWIP) classified traffic can be assigned high priority queue. In one of our experimental cases of OLA, we used a hardware TX queue of WME traffic to transmit beacon frame (see section 5.1).

## 4.3 Software and Utility

This section describes details of customization and added functionality to the driver software for WLAN chipset. Moreover, in the beginning, a details discussion of how delimiter packet traverses inside the kernel followed by a timing resolution of Linux kernel that has affected the result of our implementation.

### 4.3.1 Traversing Delimiter in the Kernel:

Two approaches have been proposed in section 3.3, to achieve the objective of this project. And in section 2.3, architecture of TDP router has been depicted. It has been clear that, delimiter, a UDP packet, will be act as indicator to send beacon frame from AP for OLA or to measure the drift between two timer, TDP timer and TSF timer .Since Linux is not real time operating system, moreover its timing resolution is low, so it is important to realize that, how much time is being spent to traverse delimiter inside kernel to be routed from Ethernet interface to PCMCIA interface. To prepare this section, we have followed OSI layer (appendix A.1) sequence physical layer, MAC layer and then IP layer.

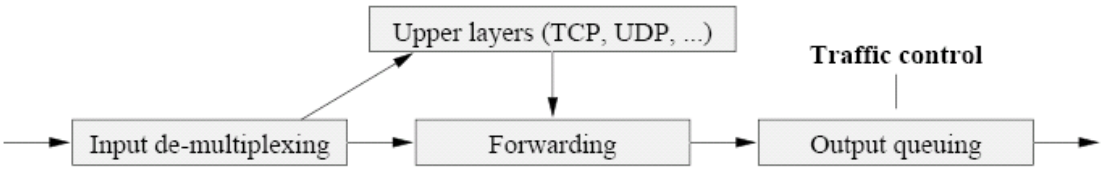



Figure 4.2 : Abstract view of network packet processing by Linux Kernel

#### *4.3.1.1 Layer-1: Packet From NIC to the Network Buffer:*

Linux kernel supports various network interfaces, for example, Ethernet, 10/100baseT, gigabit Ethernet etc. In our implementation, Ethernet has been used. So lets limit this study in the case of Ethernet interface as receiver of delimiter packet from border interface router of time driven network. Typically, the on board memory of Ethernet divided as receiving buffer (Rx) and transmitting buffer (Tx). For instance, the 3c509B from 3Com Ethernet card has 8kB on board memory buffer divided any of combination of 4kB Rx, 4kB Tx or 5kB Rx, 3kB Tx or 6kB Rx, 2kB Tx [12].

The packet is received into this on board memory buffer in FIFO basis. This structured is called *rx-ring*. After reception of a packet, hardware issues a hardware interrupt *(irq)* to make attention of cpu execution and an interrupt handler is invoked which has been registered during *open()* method when device attached with the kernel using *net_device* kernel interface. The interruption mechanism will be disabling as long as *irq* will be acknowledged by the cpu.

Linux kernel network buffer structure called *sk_buff* declared in */Linux-src/include/linux/skbuff.h* is then allocated and then packet is copied into the kernel memory that can be mapped to the Direct Memory Access (DMA) region, shared memory etc. Once packet copied to the *sk_buff*, it needs to be queued into the kernel network queue declared as *softnet_data* structure in */Linux-src/include/linux/netdevice.h*. this structure is unique for all interfaces for single cpu machine. Packets enter and leave to and from this queue in FIFO basis.

We should describe briefly *sk_buff* an important control data structure with block of attached memory, which has been used frequently in this implementation. This control structure is used basically to contain frame content, has several methods to maintain doubly linked list of *sk_buff* e.g. *skb_put(), skb_append()*, *kfree_skb()* etc.

Once packet is queued, now it's time to handle by kernel. So interrupt handler of interfaces driver leaves a warning to kernel to dequeue packet at its convenient time. Some times this mechanism is referred as *software interrupt*.

#### *4.3.1.2 Layer-2: From Network Buffer to Appropriate Protocol Handler*

Layer 2 is further divided into logical link layer (LLC) and MAC layer by IEEE, that makes more complicated. Software interrupt is generally scheduled for execution to dequeue the frame from network buffer to appropriate protocol handler. Protocol type is supposed to identify by the device driver followed by finding encapsulation type which gives information how extract layer 2 header and eventually set *sk_buff->protocol* . Once protocol handler is determined, frame should go layer 3. Details description is out of the scope of this project work.

#### *4.3.1.3 IP Layer: Packet Forwarding*

We have called this section as IP layer since we limit our study only for IP packet that should be forwarded through the Linux kernel ( layer 3 ) followed by some kernel filtering. In one of our experiments, it was taking considerable time for delimiter packet to be routed from Ethernet interface to PCMCIA interface. The destination of the delimiter was a station associated with access point in that PCMCIA interface. (see section 5.1). We tried to analyze the reason for our implementation in this section and some suggestion to avoid the delay time to traverse the delimiter will be depicted in chapter 6. Implementation was beyond of this project work.

The packet, to be routed or dropped, enter layer 3 processing through the method called `ip_rcv()` implemented in `net/ipv4/ip_input.c`, then packet has to enter any hooks of Linux netfiler [13]. Netfilter is a framework, started from Linux kernel 2.4.X, that enables packet (statefull or stateless) filtering, network address (or port) translation (NAT, or NAPT) packet mangling or manipulation, beyond the normal Berkeley socket interface. Netfilter, the successor of `ipchains` and `ipfwadm` from previous kernel, is a set of hooks associated with kernel modules to register callback functions those are called back for each packet that is going to forward through respective hook.

`iptables`, we are more familiar with, is a userspace command line program helps to configure kernel netfiltering rule set. It is basically packet selection system built over netfilter framework.

All supported protocol of netfilter defines some 'hooks', e.g. IPv4 defines five hooks. The respective protocol stack will call the netfilter framework with the packet to be traversed and hook number [14]. Netfilter consist of sequence of hooks with some entry points for a particular protocol stack. For instance, abstract diagram of IPv4 packet traversal looks like:

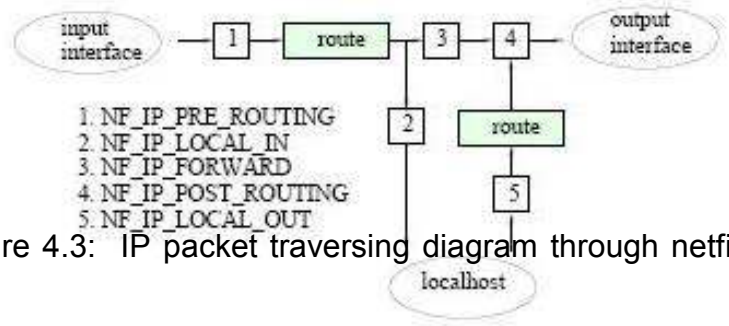



Figure 4.3: IP packet traversing diagram through netfilter framework [14].

Packets are passed to the first hook of netfilter framework referred as `NF_IP_PRE_ROUTING` followed by some introductory check, e.g. IP checksum, truncated or not, receive promiscuous etc.

Packer enter, next step, routing code where routing decision will take place, i.e. the packets are destined another interface or get into processing of local host or getting dropped because of non-routable.

If the destination address is the local host itself, packet passed to the hook called `NF_IP_LOCAL_IN` followed by passing to the process of some userspace. On the other hand, if the packet is destined to the other interface, it is being passed to the `NF_IP_FORWARD` hook. This would be the case of our implementation and giving more description in the next paragraph. The packet may pass to the final hook of netfilter, `NF_IP_POST_ROUTING`, before going to be handled by the Linux traffic control and then driver module of another interface (see next section).

In the case of generating packet locally, the hook, `NF_IP_LOCAL_OUT` takes care of them. You can realize from the figure 4.3 , after passing `local_out` hook the packet has to passed into routing code again since it needs to check source destination address and some IP options. In some case, e.g. NAT coding, routing may need to be changed by altering *`skb->dst`* field.

Any kernel module can register itself to listen to any of these hooks. When any particular hook is called from core networking code of kernel, the module registered with the hook at some points is free to mangle the packet and responsible take action of any of the following five alternatives [14]:

- NF_ACCEPT: continue to forward as normal.
- NF_DROP: drop the packet; don't continue traversal.
- NF_STOLEN: someone have taken over the packet; don’t continue to forward.
- NF_QUEUE: queue the packet for handling the process of userspace.
- NF_REPEAT: call this hook again.

Routing decision needs a quite expensive look up operation into a complex structure refer as `Forwarding Information Base` (FIB) tables. The purpose of this look up is to find out an entry of route associated with the destination IP address. A next hop, where packet will be forwarded, will be associated with each of these route entry. Since this look up into the quite complex data structure, FIB tables, is reasonably expensive operation, a route cache is also used to make it faster. A hash function combination of source address, destination address, incoming device and TOS filed, is used to look up to the cache system.

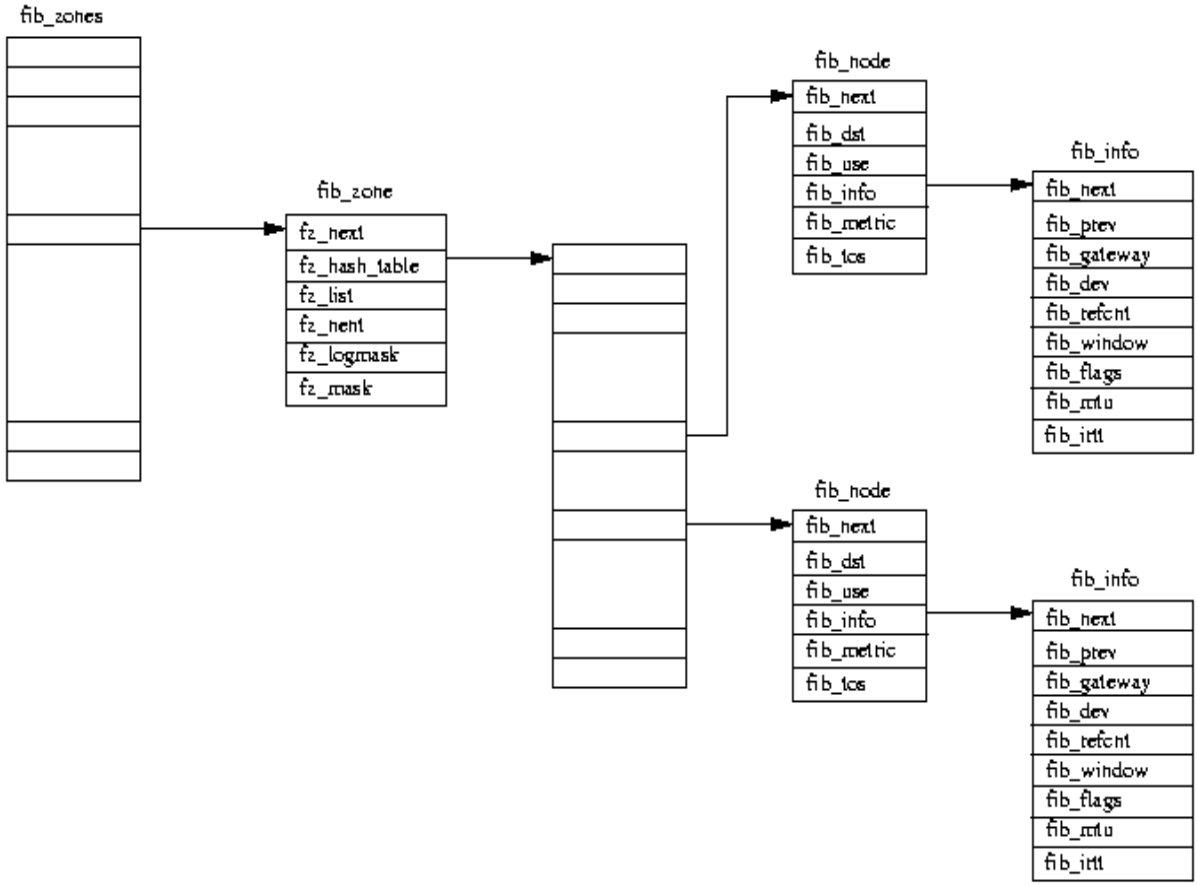



Figure 4.4: Typical view of FIB (forwarding information base) [16]

When the address of next hop is found, `skb->dst` variable is set with this value. Some works, however, to fulfill the requirement of IP routing [15] has to be completed before sending packet to the outgoing interface:

- TTL field of IP header suppose to decrement by one
- Checking the maximum transmit unit (MTU) of outgoing interface. If MTU is smaller than the packet size, packet should be fragmented otherwise let that be just transmitted.
- Depending on the requirement of IP routing ICMP message may generated
- If the hardware address of the interface of next hop is not known, an ARP packet has to be issued to get the hardware address.

In our test bed setup (see figure 5.4.), the destination address of the delimiter is one mobile station associated with the AP. So certainly delimiter

suppose to be forwarded inside Linux kernel. From the above explanation, delimiter has to be passed to first hook, `NF_IP_PRE_ROUTING`, of netfilter framework followed by routing stack decision to forward to the Access Point interface, PCMCIA. So hook `NF_IP_FORWARD` will next take care of. Since delimiter packet is being forwarded at a very short interval (few milliseconds), route entry may be in the cache system of FIB tables.

#### *4.3.1.4 Layer 2 : Packet Queuing into the Outgoing Interface*

Layer 3 packet forwarding was involved to find out the output interface, route entry of next hop, encapsulation etc. Once al of these works have been done, packet has to be queued for the outgoing interface. At this stage, Linux traffic control, complex queuing discipline, bandwidth shaping come to play. After releasing from the traffic control, device driver will take care of packet to transmit with the device to next network. Sometime device driver does additional vendor specific traffic classification, provide quality of service (QoS) oriented hardware queue of respective device for emitting the packet.

Every network device has its own queuing discipline and its own way to treat on the enqueued packet for that device interface. Queuing discipline may use *filtering* to classify among the different *classes.* For instance, this task includes to giving priority to the packets of one classes over other classes. High priority traffic may use *token bucket filtering* (TBF) which ensures data rate at most 5 Mbps, for example. On the other hand low-priority traffic is being queued by, say, FIFO discipline.

Queuing discipline provides following set of methods (related one mentioned here only) declared in *`struct Qdisc_ops`* in *`source/include/net/sch_generic.h`*

- `enqueu` – enqueues a packet with classified queuing discipline.
- `dequeue` – returns the next packet qualified for transmitting
- `requeue` – put a packet back into the queue after dequeuing for some reasons.

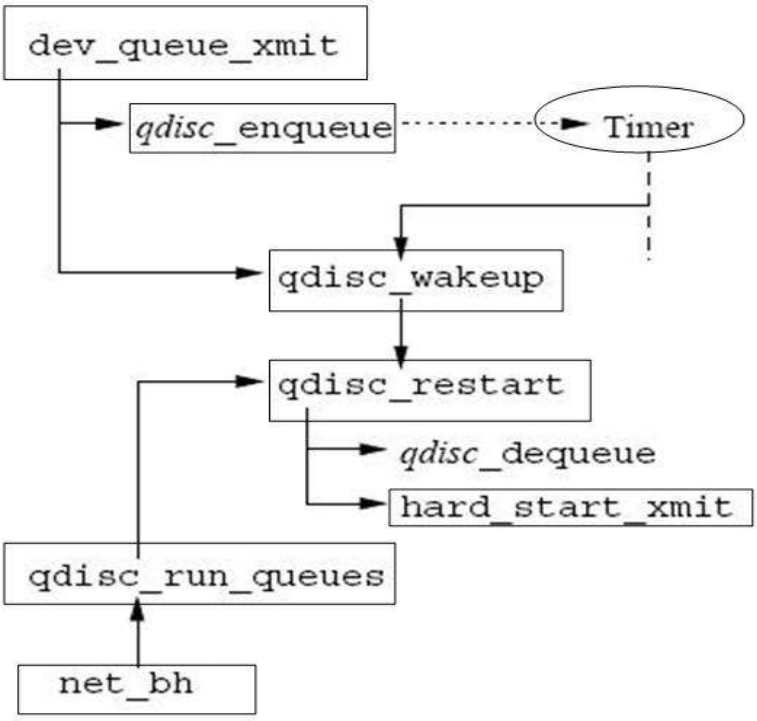



Figure 4.5: Flow of function calling to enqueue, dequeue and sending packets to device driver in Linux kernel

Figure 4.5 shows the flow of procedure call, how a eligible packet comes into hand of device (Access Point) driver code, our implementation platform can start processing with delimiter. Here details of called function invoked by queue discipline itself for classification etc. are not shown for make it simple. Detail study is beyond of this project work. When a packet is going to be enqueued into the device interface queue, *enqueue* function of *dev_queue_xmit()* in */net/core/dev.c* is invoked. Here it should be mentioned that, device's queue discipline is referenced by a pointer to the corresponding *struct Qdisc* for all of its functions. *qdisc_wakeup()* immediately call function named *qdisc_restart()*, which is the important function responsible to poll queuing discipline. *qdisc_restart()* is also responsible to take release packet from queuing discipline of the device. On success, it invokes net_device's *hard_start_xmit()* declared in */include/linux/netdevice.h*. This is the function pointer which is unique for every network device, responsible to hand over the packet for that net device's code to transmit. So for every packet suppose to transmit via that device, referenced function

of that device driver to *hard_start_xmit()* function pointer will be called. In our implementation, we have pointed to our own transmit function, *ath_xmit()* by replacing its driver own xmit function *ieee80211_hardstart()*. The details of next implementation phase will be described in section 4.3.2

#### 4.3.1.5 Packet Queuing and Timing Resolution

The queuing policy and discipline may not allow to send packet instantly as soon they are enqueued [13]. A timer (see figure 4.5) may need to configure to schedule the transmission of the packet. So packet may be left in the queue for certain time. Linux has very low timing resolution, timing is obtained by sampling internal clock tick unit (the constant HZ) which is only operating at 100 HZ ( 1 tick = 10 ms ) for Intel based architecture [17]. This low timing resolution of Linux may be the reason to impose restriction to configure parameter of scheduler for departure of packet. For instance, a packet in *token bucket filtering* (TBF) queue may need to wait 1/HZ seconds. Since, as describe earlier, 1 tick = 1/HZ seconds which equivalent to 10 ms for Intel based architecture, 1 ms for alpha. In TBF queuing discipline, say B bytes is possible to send by this buckets in any time interval (1 clock tick unit). So maximum bandwidth can be achieved B*HZ . It, however, of course, is possible to make wake up queuing asynchronously by entering other packets into the system, which lead to positive timing effect.

#### 4.3.1.6 Shorewall

To enable IP routing inside Linux kernel, to make traverse delimiter packet from Ethernet interface (eth0) to wireless interface (wifi0), we have used a user space tool for configuring `netfilter's` hook `NF_IP_FORWARD` called shorewall [26] after the motivation of users of implementation platform madwifi. It gives possibility to describe the firewall/gateway requirements using entries in a set of configuration files. `shorewall` studies those files and with the help of the `iptables` utility configures Netfilter to match the requirements. Shorewall is not a daemon. Once `shorewall` has configured Netfilter, its job is complete. After that, there is no `shorewall` code running although the */sbin/shorewall* program can be used at any time to monitor the Netfilter firewall. Moreover, `shorewall` does not use Netfilter's `ipchains` compatibility mode and can thus take advantage of Netfilter's connection state tracking capabilities.

The `shorewall` utility has been used in our case for configuration route between Ethernet and Wireless interfaces, previously configured and up, in particular the Two-interface Shorewall configuration scheme (the usage of specific configuration files depend on the network design) was

used. The content of each reconfigured file (Interfaces, Policy, and Zones) represented in appendix A.2.

Shorewall views the network where it is running as being composed of a set of ZONES, moreover it recognizes the firewall system as own zone when the specific */etc/shorewall/zones* file is processed. We consider the Wireless as one zone, and the Ethernet network as another zone.

In order to make route between two boundaries the association between Linux INTERFACES and ZONES is accomplished in /etc/shorewall/interfaces file. There is another zone that is not put in this zones file, called the "firewall zone" or "$FW". This is already defined in /etc/shorewall.conf. By default any traffic originating from the machine (fw) to the Wireless and to the local Ethernet network is accepted. For $FW and all the other zones defined beforehand the corresponding POLICY for interconnections can be configured in */etc/shorewall/policy* file. The POLICY field can be build up from a number of actions ("ACCEPT", "DROP", "REJECT", "CONTINUE" or "NONE") in order to regulate traffic patterns. See appendix A.2 configuration file of shorewall.

In additional to firewall rulesets, the */proc* filesystem offers some significant enhancements to network security settings. The pseudo file (It does not contain “real" files but rather runtime system information) structure within proc provides a file-system like interface to the kernel. This allows applications and users to fetch information from and set values in the kernel using normal file-system I/O operation. In our case to regulate IP forwarding inside Linux kernel some modifications in */proc/sys/net/ipv4* directory should be made (in particular insert “1” in */proc/sys/net/ipv4/ip_forward* file).

### 4.3.2 Linux Kernel Driver of Atheros WLAN Chipset

It has been realized that, we need to customize and add time-driven functionality to WLAN chipset driver code to implement the idea. No vendor (e.g. leading WLAN chipset provider Atheros, Broadcom, Texas Instrument) is kind enough to open source community. This phenomena makes complicated the implementation of this project. However, we have chosen madwifi [19], a Linux kernel device driver for wireless LAN chipset from Atheros Communications [20]. This is partially open source project supported by the vendor Atheros itself. This section, gives the details of implementation, is organized by starting a brief of network device driver, then details of time synchronization of madwifi and lastly specific description of both approaches, OLA and CLA.

#### 4.3.2.1 Basics of Network Device Driver for Linux Kernel

This section gives a brief introduction of network device driver for Linux kernel. This will only describe the related part that may help to understand our implementation. The experienced reader, however, can safely skip this section.

Linux kernel is inherently considered as the efficient and secure for networking. Modularity, granularity of recent kernel provides concise well-organized, and efficient and solely higher layer protocol independent coding interface that enable programmer to develop network device driver as kernel module, instead of part of monolithic kernel. As a kernel module, driver of WLAN chipset, request resources needed for the operations of the device such as I/O port, interrupt (IRQ) number etc. Kernel maintains a global list of network devices those have been detected. Each interface defined by *struct net_device,* declared in */include/linux/netdevice.h*. Whenever a device driver register itself, using *register_netdevice()*, it initializes the hardware and allocate the resources it needs by filling up *net_device's* items. Following methods are common for each network device interface in *init_module()* when a driver as kernel module is loaded into the kernel [18].

- *open()* – this opens the interface identifying as *name* filed, whenever *ifconfig* makes it up. This method also responsible to allocate system resources it needs (I/O port, IRQ, bus number, DMA, start queuing etc.)
- *stop()* – it closes the interface, release the resources, when *ifconfig* makes it *down*
- *hard_start_xmit()* – whenever kernel release a packet for transmission, this method is invoked to send packet by that device. Packet should contain in *sk_buff* (socket buffer) skb.
- *void *priv* - private data for the driver. This is the base of all data structure used for the driver development.

#### 4.3.2.2 Introduction of Implementation Platform, MadWifi

Madwifi is partially open source project supported by the vendor Atheros itself. We called it 'partially' open source since some part of this driver, provided by the vendor, is closed source referred as hardware access layer (HAL), comes available only as compiled binary for couple of architectures. We will see soon why it is closed source according to vendor.

MadWifi, *Multiband Atheros Driver for Wireless Fidelity,* a framework, provides development environment as Linux kernel (started from 2.4.x ) device driver that your wireless card will appear as multipurpose network interface in your system. It support wireless extension kernel API that

allows to configure the device using common wireless tools, `ifconfig`, `iwconfig`. A rich supported operational modes such as station, i.e. managed mode, Access Point, i.e. master mode, ad-hoc mode i.e. IBSS mode, WDS (wireless distributed system) to create large wireless network by linking with neighbor AP, monitor mode etc. made madwifi platform complex, not easily understandable and has been cumulated lots of code.

The whole madwifi code is consists of several parts. I just described here as some modules. The device driver, as kernel module named `ath_pci` consists from modules `ath`, `ath_hal`, `net80211` and `ath_rate`.

- `net80211` or `ieee80211` stack – this part is originally hacked from FreeBSD which contains generic IEEE802.11 functionality. For BSDs, this stack supports numerous WLAN devices. It, however, has been imported and customized only for Atheros wireless LAN chipsets. This module implements lost of called back which can be called by `ath_hal`, `ath` module provided that, it has to be exported by EXPORT_SYMBOL. `net80211` module also consists of WLAN authentication, cryptographical part.
- `ath` module – this module defines Atheros WLAN controller specific callbacks for net80211 module access to the hardware through HAL module. It contains time critical part of 802.11 management, e.g. beacon management, device's `ioctl`, configure and setup transmit (TX) and receive (RX) queue, PCI bus controlling connected with the CPU etc.
- `hal` module – Hardware Access Layer, `hal` module is responsible to access to hardware. This closed source component, basically maintained by the vendor, Atheros, itself, can be thought something like *firmware* of card with the only exception is, its not stored into the card, instead consider as kernel module. Commercial point is, it's required less flash memory on the board which can reduce market value of device. By definition, `hal` is not exactly firmware, since firmware is hardwared program executable on board microcontroller. According to the argument of vendor, due to chipset's versatility to tune wide range of frequencies, even in unlicensed bands (non-ISM), to enforce limit on transmit power etc and for some legal issue, Atheros keeps the code of `hal` module as closed source. Moreover, there is no documentation for hal exception a public interfaces in *`hal/ah.h`*. Soon we will see this unavailability has made our implementation so hard (see chapter 6).

- `ath_rate` module - this module selects the appropriate algorithm for the best transmission rate. Among 802.11a, b, g, multiple bit rates, this module sets the device's transmission when sending data packet. MadWifi includes three different algorithm to choose bit rate: a) onoe algorithm, b) amrr algorithm, c) SampleRate algorithm

#### 4.3.2.3 Architectural Overview of MadWifi

In the last section (4.3.2.3), we have described all modules consist of madwifi. This section will shed some light on organization and design view of madwifi code followed by some important and related data structure to our implementation.

I have customized and implemented two approaches (see section 3.1, 3.2) using MadWifi (version 0-9.1, virtual AP, `vap` branch) on Linux kernel 2.6.12. Madwifi has several modules, our customization was involved with basically *`ath`* and *`wlan`* or *`net80211`* module. *`ath`* module consists of the context of all hardware related operations, methods to access hardware (mostly in *`if_ath.c`*, C source file), PCI bus configure ( if_ath_pci.c), board configuration (*`if_ath_ahb.c`*), ioctl related definition etc. *`ath`* layer is used as *`callback`* that allows to call upper layer. i.e. 802.11 layer, *`ath_rate`* selection layer, binary module of madwifi, HAL layer.

Device's private ( *`*priv`* ) data structure *`ath_softc`* that contains pointer of other concerned structures, e.g. *`ath_hal`*, ieee80211com, net_device, beacon xmit slot (*`sc_bslot`* ) of ieee80211vap, beacon miss interrupt tasklet (sc_bmisstq) , beacon stuck interrupt tasklet (sc_bstucktq) of struct ATH_TQ_STRUCT as well as some member variable, e.g.

- HAL queue number for outgoing beacon (sc_bhalq),
- missed beacon transmission (sc_bmisscount)
- buffer for beacon frame (ath_buffhead)
- next slot for beacon xmit (sc_bnext), etc.

Hardware access layer (HAL) API defined in structure *`ath_hal`* in */hal/ah.h* . To obtain a reference of *`ath_hal`* structure driver has to call *`ath_hal_attach()`*. All hardware-related operation must call back into the HAL through this interface, `ath_hal`, i.e. this structure contains pointer of all HAL functions. Driver specific node state defined in structure *`ath_node`* in */ath/if_athvar.h* Device's buffer declared as structure *`ath_buf`* (in */ath/if_athvar.h*) that contains physical address of buffer descriptor, *`sk_buff`* pointer of *`ath_buf`* etc.

In the `net80211` module, 802.11 layer's control state is split into a common portion. one- to –one map from a physical device to one or more

virtual APs (vap) those are bound to instance of structure *ieee80211com* . But each vap has corresponding kernel device entity. All traffic and control flows, issuing *ioctls* go through each device entity. Some important and more commonly used data structure are defined in net80211 module, for example, *ieee80211vap, ieee80211com, ieee80211node* etc.

For the description of implementation and customization of madwifi driver code, we define some terminology. For example:

**HAL function** – we already know, the chipset vendor provided a public interface ( */hal/ah.h* ) of their closed source module *ath_hal* this interface file contains definition of all the functions prototype and HAL member variables. We refer these functions as HAL functions. Most of the cases, prefix of these functions *ath_hal-,* e.g. *ath_hal_beaconinit()*.

**beacon state** – most of the beacon management functionality is time critical. So device has to configure (by driver coding) and set some beacon parameters and beacon timer. We defined these set of parameters, such as beacon timer, beacon timer, nextTBTT, *beacon_miss_count*, nextDTIM etc as `beacon state`. Per station *beacon state* is defined as *struct BEACON_STATE* in *hal/ah.h*.

***Device's Interrupt Handler:***

Every network device should have an interrupt handler routine registered in *dev->irq* in *struct net_device*. *ath_intr()* routine has been registered as interrupt handler of madwifi using *request_irq()* kernel method in */ath/if_ath_pci.c* Most of actual processing are deferred from this method. We need to understand interrupt handler carefully since software beacon alert (SWBA) interrupt important for this implementation. *ath_intr()* method invokes as soon as any interrupt occurred by device. First we need to figure out the reason(s) for the interrupt by calling the HAL function *ath_hal_getisr()*. For this implementation, we are only concern with possible cause of interrupt may be beacon alert time (SWBA) and beacon miss exceed (*HAL_INT_BMISS* ). Device notifies to driver to prepare beacon frame by occurring *software beacon alert* (SWBA) interrupt sufficiently before next beacon transmission time during the next TBTT since it needs some time to prepare, generate and update the dynamic content of beacon frame followed by putting into the beacon queue. So it might be difficult to meet the timing constraint under load, if SWBA interrupt is not occurred sufficiently before time to send next beacon. Other than these two kinds of interrupt, common interrupt are, frame receive (HAL_INT_RX), receive error (HAL_INT_RXEOL)to re-read link when RXE bit set etc. but study of these is beyond of this project.

#### 4.3.2.4 Time Synchronization Functionality of MadWifi

This section will describe specific implementation description and design architecture of beacon management functionality of madwifi as master mode, i.e. AP mode. We know (see section 2.3) access point is suppose to send beacon frame at exactly every beacon time interval with time accuracy of few micro-seconds.

Missing to send few numbers of consecutive beacon frames is considered severe annoyance of the wireless network and may lead to collapse the whole network. Most of the part of this rigid time critical beacon management has been let to control by device's firmware ( e.g hostAP [8] ) or device's microcontroller to ensure this time accuracy.

Wireless LAN chipset vendor, Atheros Communications Inc. provided a public interface of their closed source Linux kernel module `ath_hal` (in *`hal/ah.h`*) without any specific documentation (very few vendors are kind enough to open source community). Most of member variables and function is the part of *`struct ath_hal`* defined in *`hal/ah.h.`* In our implementation some `hal` functions are excessively used, for example, (it is not possible to give all of them. Only important one is here):

*`ath_hal_setupxtxdesc()`*- to set transmission queue(TX) descriptor.

*`ath_hal_intrset()`*- to enable or disable interrupt firing

*`ath_hal_beaconinit()`*- to initialize beacon state

*`ath_hal_gettsf64()`*- to get current chipset TSF time in µs

*`ath_hal_puttxbuf`*- to put frame in a hardware queue

*`ath_hal_txstart`*- to enable any TX queue

etc..

Beacon management functionality of madwifi driver can be described in two folded: a) beacon state initialization b) interrupt driven transmitting.

***a) Beacon state initialization:***

Initialization of beacon state has been done *`ath_beacon_config()`* function in *`ath/if_ath.c`*. This function has been called to start or restart beacons. This function, for an AP, set up the device to notify the driver time to prepare and issue next beacon frame, according code, it is referred as *`software beacon alert`* (SWBA). Lets explain step by step, the series of jobs done by this function *`ath_beacon_config()`*. Understanding this part is important for our implementation.

- Take the beacon interval from *net80211* module. In usual case, beacon interval is given when device is up (using the wireless tools iwconfig) and suppose to be kept internally in ic_lintval variable of

ieee802com struct defined in */net80211/ieee80211_var.h* and also AND-ed with HAL_BEACON_PERIOD to ensure in correct margin value.

- Since we are only interested to AP mode, nextTBTT will be equivalent to beacon interval and AP is supposed to schedule a beacon frame as the next frame for transmission at each TBTT [21]
- Enabling beacon timer and SWBA interrupt by OR-ing interrupt mask variable *sc_imask* ( member variable of *priv data structure ath_softc in */ath/if_athvar.h* ) with HAL_INT_SWBA constant.
- One of the major jobs of beacon state initialization is configure and setup a hardware device queue for particularly only beacon frame transmission. This have been done by implementing function *ath_beaconq_config()*. The responsibility of this function to get a hardware transmit (TX) queue dedicated for beacon using a HAL function *ath_hal_gettxqueueprops()*. In our implementation, device's TX queue number 9 was *beacon queue*. The value of some TX queue parameters of 802.11 MAC, e.g. Inter Frame Space (IFS) duration, CWmin, CWmax are to be set in this function to always burst out beacon traffic.
- A HAL beacon management function, *ath_hal_beaconinit()* is being invoked and passed the OR-ed value of beacon interval with constant HAL_BEACON_RESET_TSF and HAL_BEACON_ENA successively to enable particular chipset's register followed by beacon *miss_count* set to 0. Interrupt firing disabling and enabling again has been done by invoking hal function ath_hal_intrset with the variable *sc_imask*.

So we understand how beacon state is initialized to set and configure the device to issue SWBA. Now we analyze, when this *ath_beacon_config()* function is to be invoked (see figure X). Actually to start or restart, this function needs to invoked, i.e.

- when device is up, *init_module()* of driver is run, it needs to start beacon.
- when chip needs to reset followed by some fatal hardware error or FIFI RX queue over run or any problem.
- when channel has been changed or set

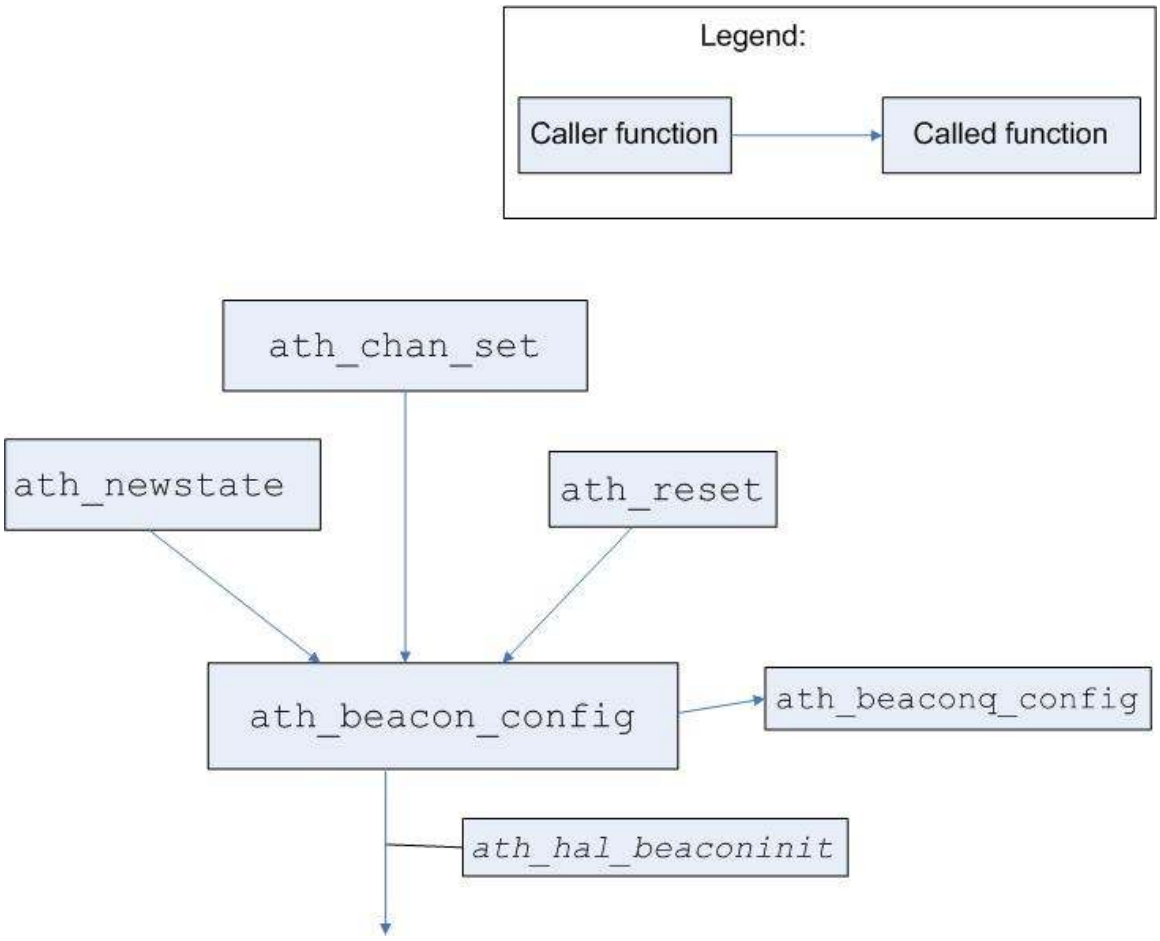



Figure 4.6: Flow of function calling to start/restart beacon

*ath_newstate()* also invokes *ath_beacon_alloc()* function to allocate and setup beacon frame content. If any previous allocation for beacon exists, release the associated *skb*. The beacon frame buffer must be 32-bit aligned. *mbuff()* routine is supposed to return something with this alignment from *ieee80211_beacon_alloc()* method of net80211 module. *ath_beacon_alloc()* basically allocates socket buffer (*sk_buff*) for beacon frame.

***b) Interrupt driven transmission of beacon frame:***

From the previous section, *interrupt handler*, we already know, hardware issues time critical interrupt beacon alert time (SWBA) interrupt at sufficiently before the next TBTT. Interrupt occur before the nextTBTT since we need some time to prepare, generate, update some dynamic

content of beacon frame taking account into current state and finally after completion of generation of beacon frame we need to put into the beacon queue. We will see here, bit details of this! The series of tasks have to be done to prepare beacon frame and post generation task and a bit description of methods involve with this phase. It is important to note that, most of execution and processing of this section of code is common for both approaches of this project implementation.

When madwifi interrupt handler, *ath_intr()* realized the SWBA interrupt has occurred, so it is time to prepare beacon frame and this has been done by invoking *ath_beacon_send()* method. Figure X shows the flow function calling until completion of beacon generation process. First task should be a routine check whether the previous beacon has released or not. If it is pending, so we miss to send one beacon frame which obviously is not good. If we miss consecutive a constant number of beacon so a situation reffered as `beacon stuck` is occurred. We called this constant number of missed beacon as `BEACON MISS THRESHOLD` (say we set this value 5) If beacon miss count is crossed this threshold we are in beacon stuck situation so it needs to reset the device otherwise, it will reduce performance of wireless network dramatically and associated stations will not understand the existence of network.

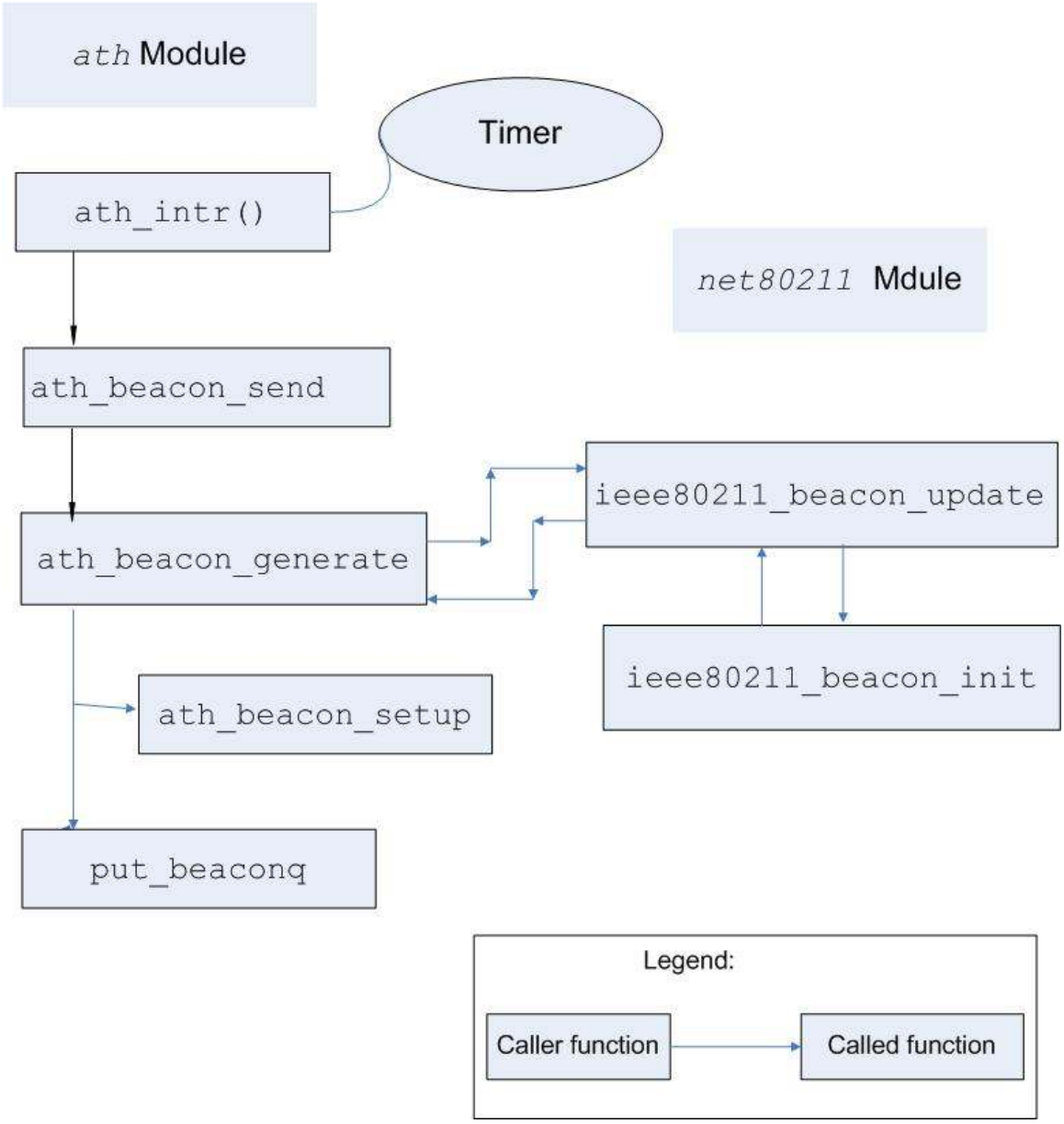



Figure 4.7: execution flow of interrupt driven beacon transmission

If we don't find any pending beacon, we should move on. So it's time to fill up the content of beacon frame for each virtual AP (*vap*, if more than one). This has been done by invoking function *ath_beacon_generate()* and return the pointer of complete beacon frame as ath_buf. sk_buff has been used to link up with beacon buffer. However, the important job, updating dynamic content of beacon frame based on current state is being performed by callback function *ieee80211_beacon_update()* of 802.11 layer in *net80211* module ( */net80211/ieee80211_beacon.c*

). This method first checks whether channel changed by calling function *ieee80211_doth_findchan()* and set the value. Most of the field's value of beacon frame (see section 4.3.2.4.b ) is set by the function called *ieee80211_beacon_init()*. First field of beacon frame format, timestamp (64-bit long) is suppose to set by the hardware. The value of timestamp field will be the time (e.g. in milliseconds) when exactly first bit of beacon frame gets into the air [21]. Next field, beacon interval (2 octet) suppose to store in *ni_intval* of *struct ieee80211_node*, third field capability information, then SSID followed by supported rate returned from the function *ieee80211_add_rates()*. CF parameter set information element, WME, WPA parameter and Traffic Information message (TIM) bitmap generated by AP and vendor specification. This project work is not concerned with these former set of parameters.

```
beacon_send()
{

      if( check previous beacon is pending )
      {
            beacon_miss_count++ ;
            if( consecutive beacon_miss_count is greater than
                BEACON_MISS_THRESHOLD )
            {
                  chipset_reset();
                  return from this method;
            }
      }
      // if comes here beacon_miss_count should resume again
       generate_beacon_frame();
      if( beacon generation and update is successfull )
      {
            a. stop current DMA to beacon_queue;
            b. insert the pointer of beacon frame into beacon queue;
            c. enable beacon queue;
            d. increase global beacon_xmit varible;
      }

}

N.B. In this pseudocode, it is not been used real name of functions
      to make it reader friendly
```

Figure 4.8: Pseudo code of interrupt driven beacon frame transmission

Allocation of beacon frame, set the frame length, and set appropriate bit is done by *ieee80211_beacon_alloc()* function. However, after filling up and update the values of field of beacon frame, execution should back from 802.11 layer (net80211 module) to ath module. Now its time to construct the descriptor of transmission (TX) queue by invoking ath_beacon_setup() and a hal function

*ath_hal_setuptxdesc()* by passing frame length, 802.11 layer header length, Atheros packet type (beacon), rate, ACK flags, RTS/CTS duration etc. it also needs to (un)map from PCI bus to CPU and vice versa.

So now we have successfully generated beacon frame. Therefore only task has left to put it into beacon frame. Before putting, we need to stop current DMA to beacon queue to be safe side. A hal function *ath_hal_stoptxdma()* is used then put the new beacon frame (frame address) into beacon queue (indicated by *sc_bhalq*) and enable to hardware queue. To do these tasks, two hal function is used, *ath_hal_puttxbuf(), ath_hal_txstart()* accordingly.

***Timing constraint and responsibility to start beacon transmission:***

Time synchronization of IEEE802.11 standard (see section 2.3) AP shall set the value of timestamp field of beacon so that it will be equal to the value of TSF timer of AP at the first bit of timestamp is transmitted to the physical layer of the device plus AP's transmitting delay from MAC-Physical interface to its interface with wireless medium [21]. So it is unlikely possible to follow this timing constraint to put timestamp by software code. Moreover, this timing constraint has to maintain the synchronization of TSF timer within 4 microsecond plus propagation delay of Physical to Physical layer [21]. Therefore, most of WLAN chipset, like Atheros, takes care of setting timestamp. We have seen in section 4.3.2.4.a, beacon state initialization, before starting to transmit beacon by AP periodically, device needs to initialize beacon state. Device has to know two time, first is when SWBA interrupt has to issue, second beacon interval and nextTBTT. One of the major hal function, *ath_hal_beaconinit()*, is used to do this by setting up chipset special register with two beacon parameter, beacon interval and nextTBTT. Moreover, it is very important to know that, according to the HAL specification, this hal function is suppose to use for following two tasks when chip set will act as AP:

- set the hardware with new beacon interval, nextTBTT
- start chipset TSF timer to increment (in microsecond) from zero.

So clearly if we do initialize beacon state, consequently, TSF chipset timer is being reset and start counting from zero. We will see next section, this is one of draw back we can not avoid for implementation for close loop approach. After beacon initialization, device already knows two important times, when to issue SWBA interrupt and time to send generated and queued beacon frame. SWBA interrupt occurs sufficiently before time to send next beacon. After this interrupt occur, execution goes on according to the description of section 4.3.2.4.b, `interrupt driven beacon transmission`, So beacon frame will be gated into the beacon queue and hardware responsibility to start sending the frame if medium is free. In case of, however, busy medium, beacon will be sending immediately after the current frame transmission. But this time shall not be accumulated with the `nextTBTT` or beacon interval.

#### 4.3.2.5 Specific Description of OLA Implementation

We already know from the section 3.3, the main idea of OLA, open loop approach is solely replacement of existing TSF functionality of AP to send beacon frame. Ideal case will be, AP does not know any timing information, get delimiter packet from TDP send instantly beacon frame and ensure the timing constraint of TSF synchronization between AP and station.

If we analyze the open loop approach, two major steps have to be done to achieve the goal. One, disable the existing TSF functionality of AP, second; make the transmission of beacon driven by delimiter by sidestepping the hardware responsibility to start sending beacon frame. We have already discussed how `delimiter` traversed inside Linux kernel, then ultimately comes to AP interface (PCMCIA interface) buffer that enable to madwifi driver to have delimiter packet like other packet destined to AP interface or any station associated with AP. This design, however, has one drawback to particularly this project work, described in chapter 6.

We have implemented *`ath_xmit()`* method pointed to device's (*`net_device`*) `hard_start_xmit` function pointer. Therefore, when kernel routing decision is over, particularly when netfilter and queuing discipline (see section 4.3.1 ) release delimiter or any other packet destined to AP interface or its associated station, *`ath_xmit()`* method will be invoked. No outgoing packet will be transmitting until this function execution. Now we need to be confirmed whether the packet is delimiter or other non-delimiter. In the former case, we invoke *`ieee80211_hardstart()`* for default `xmit` function for the device for non-delimiter packet which will be processed and classified in `net80211` module by calling *`ieee80211_hardstart()`* and *`ieee80211_classified()`* in */net80211/ieee80211_output.c*.

From the specification of TDP router implementation [23], we already know that, delimiter is special UDP packet which Type of Service (TOS) or `Differentiated Services Field Codepoints` DSCP [24] of IP header portion should contain specific pattern to identify delimiter. We implement to perform this identification procedure using a method called *`if_delimiter()`*. This method return true value the packet is delimiter UDP type packet, otherwise return false. Simplest way to do this, extract IP header of the packet and using *`tos`* member variable of *`struct iphdr`* defined in */source/include/net/ip.h* . After confirming the delimiter we invoke *`handler_tdp()`* this function actually consider as process context initiator for all TDP AP implementation.

Here it is important to know that, TDP router implementation explains that, the delimiter sent by TDP router is more frequent, the accuracy of TDP timing aligned with UTC more precise. For instance, sending delimiter every 1 ms (UTC time) will be more accurate than sending every 100 ms (UTC time). We already define the delimiter which would be used to drive beacon frame, i.e. would be consider as an indication of sending beacon is called `tick`. So if TDP router is configured to send delimiter at 1 ms, certainly every 100$^{th}$ delimiter will be a tick if beacon interval of TDP AP is suppose to 100 ms (UTC time). We define this required number of delimiter as `tick interval` for the implementation. We have option, for the experimental purpose, to the implementation to set the `tick interval` constant according to the configuration of TDP router and beacon interval of TDP AP. For example,

*`Case A:`* Delimiter interval is equal to beacon interval of TDP AP `tick interval` will be 1

*`Case B:`* Delimiter interval is 1 ms (UTC), beacon interval of TDP AP 100 ms (UTC), certainly `tick interval` will be 100

*`Case C:`* Delimiter interval is 50 ms (UTC), beacon interval of TDP AP 100 ms (UTC), therefore tick interval is 2

```
OLA(sk_buff_for_packet, net_device_for_AP_interface )
{
   //get the delimiter or non-delimiter packet
   //for AP interface from kernel

      //extract IP header and get the TOS/DS filed
      if( if_delimiter())
      {
            received_delimiter++;
            if(received_delimiter is equal to TICK_INTERVAL )
            {
                  //time to send beacon frame

                  tdp_driven_beacon_send();
                  //similar procedure with Figure X
                  //except used h/w queue is data queue
                  //or best effort queue
                  free_from_kernel(delimiter);
                  received_delimiter = 0;
            }

      }else{
       //default xmit function for the device
       // trafic classification followed by priority setup
       kernel_dev_queue_xmit(delimiter);
       return;
      }

}

N.B. In this pseudocode, it is not been used real name of functions
      to make it reader friendly
```

Figure 4.9: Pseudo code of open loop approach

When we determine received delimiter as `tick` that will be similar to SWBA interrupt except to give to the hardware responsibility to send. From the description of section 4.3.2.4.a, beacon state initialization and beacon management functionality of madwifi, we already know that, it is unlikely possible to maintain timing constraint without giving responsibility to hardware to send beacon frame. However, one of the major goals of OLA is replacement of existing TSF functionality AP. So hardware is not responsible to start sending beacon rather a tick would be consider as indicator to start sending within zero time, ideally. It may not possible to achieve this goal without having much control to the hardware. We will discuss this issue more in section 4.3.2.5. Rather, we defined implementation of OLA as `feasibility study` of OLA. Without giving hardware to start to send beacon frame, the only way has left to work with

existing chipset, to use different hardware queue to transmit beacon frame as normal data frame. Certainly this will not ensure the timing constraint and degrade performance seriously while network is busy. We invoke to generate and transmit beacon we invoke function *tdp_beacon_send().* This is mostly customize version of *ath_beacon_send()* and execution of program flow is also very similar to the section 4.3.2.4.b. that's why we leave here more detail description. However, after generation of beacon frame, we put it into the a hardware data queue by using function *ath_hal_puttxbuf()* passed into hardware queue number . Since this implementation is considered as feasibility study, we experimented by using different queue to transmit beacon frame, e.g. best-effort queue, a data queue basically used WME (wireless multimedia extension, 802.11e ) QoS. It is important to note here, after gated beacon frame into the queue, we have nothing to do other than safely anticipation that; hardware will start to send it immediately.

#### 4.3.2.6 Specific Description of CLA Implementation

We have studied in the section 3.3.2, goal of close loop solution is to make alignment of existing timing functionality of AP and make it time driven with TDP. Since delimiter packet doesn't contain any timing information itself, arriving time of it should be considered as TDP timer in local system. Some simple steps have been followed to implement this approach.

- preserve delimiter arriving time as TDP timer
- measure the drift between TDP timer and chipset time
- if drift is crossed to a predefined threshold value, align the beacon frame transmitting time with TDP time, otherwise continue

In this implementation, we reuse few methods from open loop implementation, e.g. *ath_xmit()* for receiving all packets for AP interface , if_delimiter(), for determining delimiter packet, handler_tdp() for start TDP processing etc., all of these function has been described details previous section.

```
CLA(sk_buff_for_packet, net_device_for_AP_interface )
{
   //get the delimiter or non-delimiter packet
   //for AP interface from kernel

      //extract IP header and get the TOS/DS filed
      if( if_delimiter())
      {
            received_delimiter++;
            increase TDP_timer according to TICK_INTERVAL;

            if(received_delimiter is equal to TICK_INTERVAL )
            {
                  //measuring drift
                  TSF_time = get_chipset_time();
                  drift = differnce(TSF_time, TDP_time);
                  if( drift is greater than threshold value )
                  {
                     //so it needs to allign beacon transmission
                     //so it requires beacon   initialization, enable
                     //beacon timer, enable SWBA interrupt again etc...
                        tdp_beacon_init();

                     //TDP timer should start counting from current TSF
                     //time again
                        TDP_timer = get_chipset_time();

                  }

                  free_from_kernel(delimiter);
                  received_delimiter = 0;
            }

      }else{
       //default xmit function for the device
       // trafic classification followed by priority setup
       kernel_dev_queue_xmit(delimiter);
       return;
      }

}


      //tdp_driven_beacon_send();
      //similar procedure with Figure X. no need to sketch again...

N.B. In this pseudocode, real name of functions is nor used
      to make it reader friendly
```

Figure 4.10: Pseudo code of close loop approach

One of the major goal of timing synchronization of CLA implementation is to measure drift between two time line, one chipset's own timer, two, TDP timer (UTC time). But we don't have TDP timer in the system. We have maintained
TDP timer using a counter variable, *tdp_timer* in *ath/if_ath.c* that should initialize with TSF timer of chipset two times, when AP is turned from normal master(AP) mode to TDP operation mode and when any beacon initialization for TDP occurs. tsf_timer is another counter that hold the current TSF time, chipset time returned by method `ath_hal_gettsf64()` which should be the value of 1 MHz chipset clock increasing one every microsecond.

To measure drift between two timelines, one timer suppose to be reference time, drift will be with reference to that timer. It is certainly not possible to set the chipset timer to any adjusted time. This idea leads to make us TSF timer as reference time. So difference between *tdp_timer* and *tsf_timer* will be the amount of drift. According to the TDP router implementation explains that, the delimiter sent by TDP router is more frequent, the accuracy of TDP timing aligned with UTC more precise. For instance, sending delimiter every 1 ms (UTC time) will be more accurate than sending every 100 ms (UTC time). To keep option for further improvement and increase TDP AP system performance and scalability, we introduced the idea of `tick interval` (see previous section). By setting the appropriate value we can configure the beacon interval parameter of TDP AP whether TDP router is sending delimiter any lower interval. This idea also defines how tdp_timer will be increasing. For example if TDP router is configured to send delimiter at 1 ms (UTC), *tdp_timer* will be increasing by 1000 microsecond and the same way.

We have checked whether the drift has been crossed to a threshold value referred as `drift threshold` which is a configurable parameter of TDP AP at every tick interval, i.e. at beacon interval. For our experiment we set drift threshold value is 1000 microsecond. If this drift is crossed the `drift threshold` that means each beacon frame is being sent 1000 microsecond later or before with respect to TDP timer. So we need to synchronize TDP AP with TDP timer. We called this process as `alignment`. So we need to initialize beacon state as described section 4.3.2.4(a), Although we don't need to follow all steps, but we need re-enable beacon timer, re-start SWBA interrupt firing etc. These series of task has been in *tdp_beacon_init()* For AP operation nextTBTT and beacon interval parameter will be same, but according to Atheros chipset specification and `hal` function interval will OR-ed with

HAL_BEACON_RESET_TSF, HAL_BEACON_ENA and eventually initialize the chipset registers by invoking hal function *ath_hal_beaconinit()*.

We already know, it is serious performance degration if we miss any consecutive delimiter or delayed to arrive delimiter packet since TDP router is being act as timing master of TDP AP. We devised a simple algroithm to handle this like, we stored two time values *timenow* which contain current kernel time (in microsecond) returned by the method *get_kerneltime_us()* that actually used kernel API *current_kernel_time()* defined in */source/kernel/time.c* and *timeprev* which contains previous time of arriving of immediate last delimiter packet. We tried to find out whether we miss a delimiter by checking the difference between these two time values is more larger (defined in constant *DELIMITER_MISS_THRESH* ) than delimiter interval of TDP router. The number of consecutive missing (*delimiter_misscount*) delimiter is crossed a predefined constant value, we should reset the chipset or try to find out the reason. In case of missing less than *DELIMITER_MISS_THRESH* or delayed we should increase *tdp_timer* according to last recorded delimiter interval.

Beacon state initialization has to invoke a closed source hal function *ath_hal_beaconinit()* which leaves a compensation to CLA implementation, i.e. this method reset the TSF timer means TSF timer start counting from zero at every alignment. Timestamp value of beacon frame will be changed accordingly, so station associated with AP should not be affected. Once alignment is completed, then its hardware responsibility to fire SWBA interrupt and eventually to start transmit beacon frame at exact time at predefined beacon interval as described in section 4.3.2

We got an unavoidable difficulty to set beacon interval of TDP AP in implementation of CLA. The existing WLAN chipset certainly designed by following IEEE802.11 standard that specify the beacon interval will be in Time Unit (TU), a measurement of time equal to 1024 μs, i.e. 100 TU = 102.4 ms. Hardware allows to set beacon interval as an integer value, for example 100 TU. In this case, AP suppose to send beacon frame at 102.4 ms interval.

On the other hand, in the CLA, beacon interval of TDP AP will be according to delimiter interval of TDP router, equal to delimiter sending interval or an integer (say 100) multiplication of delimiter sending interval. According to TDP router implementation, the configurable granularity of setting delimiter sending interval is 1 ms now. It may possible to change. But for our experimental purpose, we set an appropriate tick interval

so that there will be a corresponding integer `time unit` value of beacon interval of AP. For instance, we did experiment

125 TU = 128 ms or

167 TU = `171.008` ms (with 8 μs error)

This constraint can be eliminated by allowing TDP router to configure for sending delimiter at μs (UTC) unit.

*Chapter Five:*

# System Evaluation

The previous chapter describes of implementation of open loop approach and close loop approach. We have evaluated the system considering some issues, e.g. stability, scalability, and performance analysis of the prototype in different scenarios.

## 5.1 Evaluation of Open Loop Approach:

We have studied in section 4.3.2.5 that, it is reasonably impossible to achieve the goal of the OLA without full control of hardware and associated driver code. We defined this implementation as feasibility study of OLA, since we considered beacon frame as a ordinary management frame of 802.11 management instead of giving hardware responsibility to start send it and certainly after putting it into the queue, driver code has nothing do except anticipate it will send as soon as gated into the queue. Therefore, it there is no other traffic for AP to send out, beacon frame is supposed to send immediately.

During evaluation of performance of feasibility of OLA, we have mainly concentrated to determine:

- the tick (TDP) driven beacon interval. and
- how promptly beacon frame is driven by a tick (delimiter).

We believe that, by measuring elapsed time between two times at which two consecutive beacon frames captured by a reliable and powerful network analyzer, Fluek Network, we will get tick driven beacon interval of TDP Access Point. Since, in our experimental setup there was no other significant traffic, we can anticipate that when we put beacon frame into the queue, device will start to send it. We have measured this time as beacon arrival time at network analyzer in different scenarios. For example:

Here we should mention, in all of the experiment, we have used the testing tools, such as Fluek network analyzer to capture beacon frame using its wireless interface, as well as its Ethernet interface to capture wired packet. The arrival time of frame shows at system time up to ‘µs’ unit. We also use kernel time of TDP AP (from kernel syslog) that represents

current kernel time of the system, showable up to ns. It is important to notice, both values are coming from an ordinary PC architecture which may have some error. However, it's supposed to be very negligible. Moreover, the TDP router is now also implemented on ordinary PC, since, according to its specification, it also has some negligible error in sending delimiter aligned with UTC time. We should consider, the following result is with this error (if any).

***Scenario-1:*** TDP router configured as delimiter interval 200 ms (UTC)

So, ideally, beacon interval of TDP AP is 200 ms(UTC).

In the figure 5.1(a) (b) showed snapshot of the measurement. We have realized considerably constant beacon arrival interval (in ms, system time of network analyzer) with some (e.g. 7 microsecond) plus or minus. However, this more or less value is not constant any more, for instance, we found, interval is about or more than 50 microseconds.

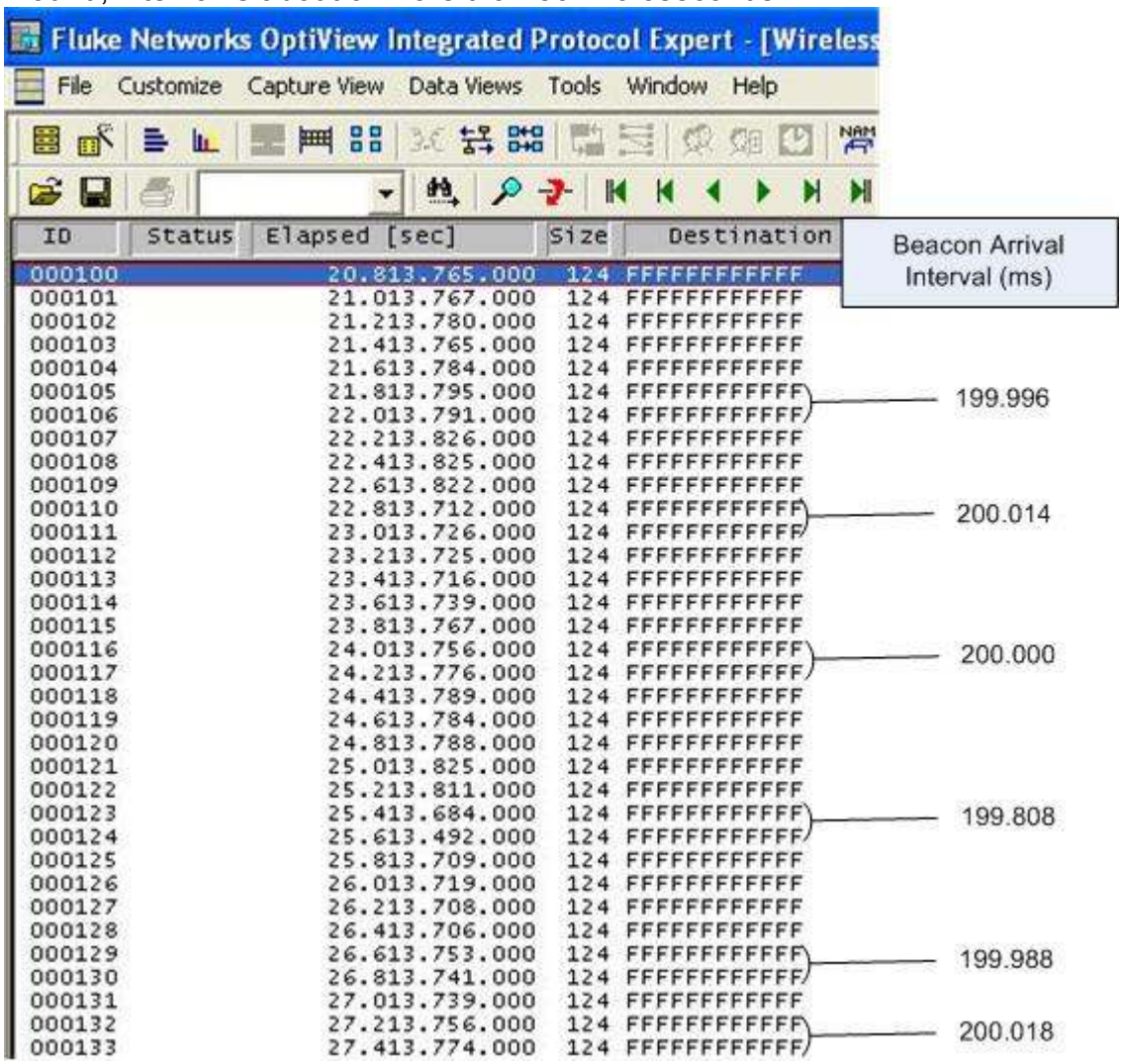
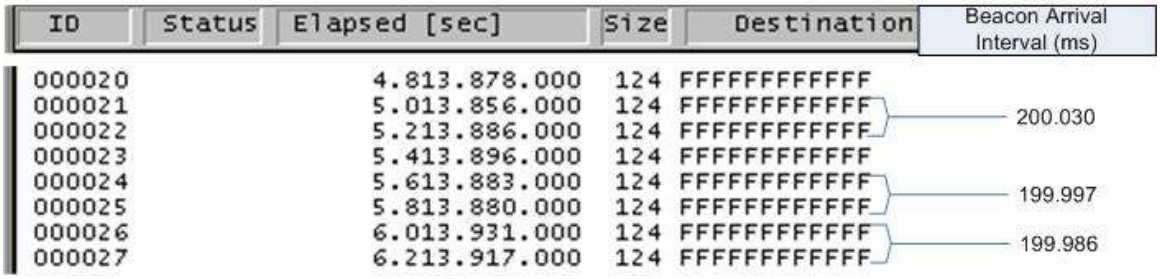


Figure 5.1(a): snapshot of beacon arrival interval from TDP AP in OLA



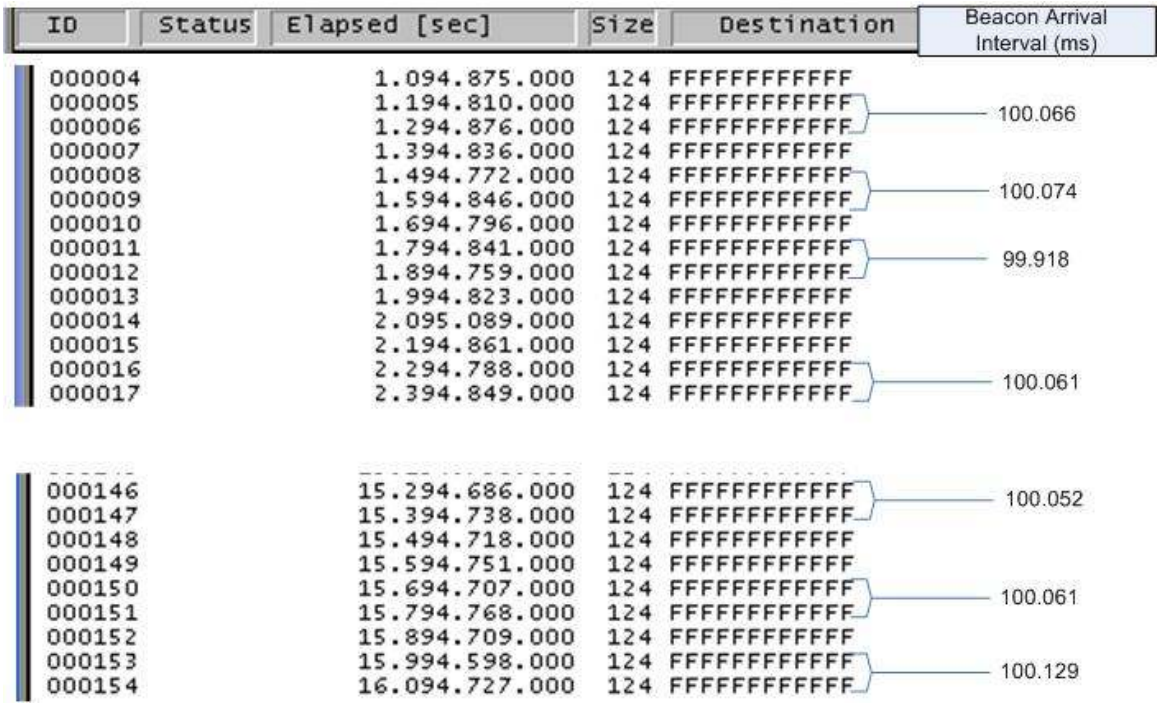
| ID | Status | Elapsed [sec] | Size | Destination | Beacon Arrival Interval (ms) |
|---|---|---|---|---|---|
| 000020 | | 4.813.878.000 | 124 | FFFFFFFFFFFF | |
| 000021 | | 5.013.856.000 | 124 | FFFFFFFFFFFF | 200.030 |
| 000022 | | 5.213.886.000 | 124 | FFFFFFFFFFFF | |
| 000023 | | 5.413.896.000 | 124 | FFFFFFFFFFFF | |
| 000024 | | 5.613.883.000 | 124 | FFFFFFFFFFFF | 199.997 |
| 000025 | | 5.813.880.000 | 124 | FFFFFFFFFFFF | |
| 000026 | | 6.013.931.000 | 124 | FFFFFFFFFFFF | 199.986 |
| 000027 | | 6.213.917.000 | 124 | FFFFFFFFFFFF | |

Figure 5.1(b): snapshot of beacon arrival interval from TDP AP in OLA

***Scenario-2:*** TDP router configured as delimiter interval 100 ms (UTC)

So, ideally, beacon interval of TDP AP will be 100 ms(UTC).

In the figure 5.2 showed snapshot of the measurement. We have realized almost similar value with scenario-1, i.e. considerably constant beacon arrival interval (in ms, system time of network analyzer) with some (e.g. 7 microsecond) plus or minus. However, this more or less value is not constant any more, for instance, interval is about or more than 50 or even 70 µs.



| ID | Status | Elapsed [sec] | Size | Destination | Beacon Arrival Interval (ms) |
|---|---|---|---|---|---|
| 000004 | | 1.094.875.000 | 124 | FFFFFFFFFFFF | |
| 000005 | | 1.194.810.000 | 124 | FFFFFFFFFFFF | 100.066 |
| 000006 | | 1.294.876.000 | 124 | FFFFFFFFFFFF | |
| 000007 | | 1.394.836.000 | 124 | FFFFFFFFFFFF | |
| 000008 | | 1.494.772.000 | 124 | FFFFFFFFFFFF | 100.074 |
| 000009 | | 1.594.846.000 | 124 | FFFFFFFFFFFF | |
| 000010 | | 1.694.796.000 | 124 | FFFFFFFFFFFF | |
| 000011 | | 1.794.841.000 | 124 | FFFFFFFFFFFF | 99.918 |
| 000012 | | 1.894.759.000 | 124 | FFFFFFFFFFFF | |
| 000013 | | 1.994.823.000 | 124 | FFFFFFFFFFFF | |
| 000014 | | 2.095.089.000 | 124 | FFFFFFFFFFFF | |
| 000015 | | 2.194.861.000 | 124 | FFFFFFFFFFFF | |
| 000016 | | 2.294.788.000 | 124 | FFFFFFFFFFFF | 100.061 |
| 000017 | | 2.394.849.000 | 124 | FFFFFFFFFFFF | |
| 000146 | | 15.294.686.000 | 124 | FFFFFFFFFFFF | 100.052 |
| 000147 | | 15.394.738.000 | 124 | FFFFFFFFFFFF | |
| 000148 | | 15.494.718.000 | 124 | FFFFFFFFFFFF | |
| 000149 | | 15.594.751.000 | 124 | FFFFFFFFFFFF | |
| 000150 | | 15.694.707.000 | 124 | FFFFFFFFFFFF | 100.061 |
| 000151 | | 15.794.768.000 | 124 | FFFFFFFFFFFF | |
| 000152 | | 15.894.709.000 | 124 | FFFFFFFFFFFF | |
| 000153 | | 15.994.598.000 | 124 | FFFFFFFFFFFF | 100.129 |
| 000154 | | 16.094.727.000 | 124 | FFFFFFFFFFFF | |

Figure 5.2: snapshot of beacon arrival interval from TDP AP in OLA, scenario 2.

***Scenario-3:*** TDP router configured as delimiter interval 200 ms (UTC)

So, ideally, beacon interval of TDP AP will be 200 ms(UTC).

In this experiment, we use best effort hardware queue to transmit beacon.

In the figure 5.3 showed snapshot of the measurement. We have realized almost similar value with scenario-1, i.e. considerably constant beacon arrival interval (in ms, system time of network analyzer) with some (e.g. 7 microsecond) plus or minus.

| ID | Status | Elapsed [sec] | Size | Destination | Beacon Arrival Interval (ms) |
|---|---|---|---|---|---|
| 000009 | | 2.563.717.000 | 124 | FFFFFFFFFFFF | |
| 000010 | | 2.763.750.000 | 124 | FFFFFFFFFFFF | 199.989 |
| 000011 | | 2.963.739.000 | 124 | FFFFFFFFFFFF | |
| 000012 | | 3.163.739.000 | 124 | FFFFFFFFFFFF | |
| 000013 | | 3.363.769.000 | 124 | FFFFFFFFFFFF | |
| 000014 | | 3.563.765.000 | 124 | FFFFFFFFFFFF | 200.000 |
| 000015 | | 3.763.765.000 | 124 | FFFFFFFFFFFF | |
| 000016 | | 3.963.793.000 | 124 | FFFFFFFFFFFF | |
| 000017 | | 4.163.781.000 | 124 | FFFFFFFFFFFF | |
| 000018 | | 4.363.657.000 | 124 | FFFFFFFFFFFF | |
| 000019 | | 4.563.681.000 | 124 | FFFFFFFFFFFF | 200.133 |
| 000020 | | 4.763.714.000 | 124 | FFFFFFFFFFFF | |
| 000108 | | 22.363.668.000 | 124 | FFFFFFFFFFFF | 199.980 |
| 000109 | | 22.563.648.000 | 124 | FFFFFFFFFFFF | |
| 000110 | | 22.763.671.000 | 124 | FFFFFFFFFFFF | |
| 000111 | | 22.963.660.000 | 124 | FFFFFFFFFFFF | |
| 000112 | | 23.163.683.000 | 124 | FFFFFFFFFFFF | 199.884 |
| 000113 | | 23.363.567.000 | 124 | FFFFFFFFFFFF | |
| 000114 | | 23.563.556.000 | 124 | FFFFFFFFFFFF | |
| 000115 | | 23.763.567.000 | 124 | FFFFFFFFFFFF | |
| 000116 | | 23.963.597.000 | 124 | FFFFFFFFFFFF | 199.987 |
| 000117 | | 24.163.574.000 | 124 | FFFFFFFFFFFF | |
| 000118 | | 24.363.584.000 | 124 | FFFFFFFFFFFF | |
| 000119 | | 24.563.618.000 | 124 | FFFFFFFFFFFF | |
| 000120 | | 24.763.600.000 | 124 | FFFFFFFFFFFF | 200.008 |
| 000121 | | 24.963.608.000 | 124 | FFFFFFFFFFFF | |

Figure 5.3: snapshot of beacon arrival interval from TDP AP in OLA, scenario 3 (best effort hardware queue).

To find out how promptly the beacon frame is drived by `tick`, we setup experimental testbed as shown in Figure 5.4. We capture two different frames into Ethernet interface and wireless interface by network analyzer. Ethernet interface, connected with Gigabit hub, captures delimiter

(tick), on the other hand, wireless interface of network analyzer captures that particular delimiter (tick) driven beacon frame and we saved time at which delimiter as well as beacon captured. As nature of hub, Gigabit hub broadcast delimiter packet to

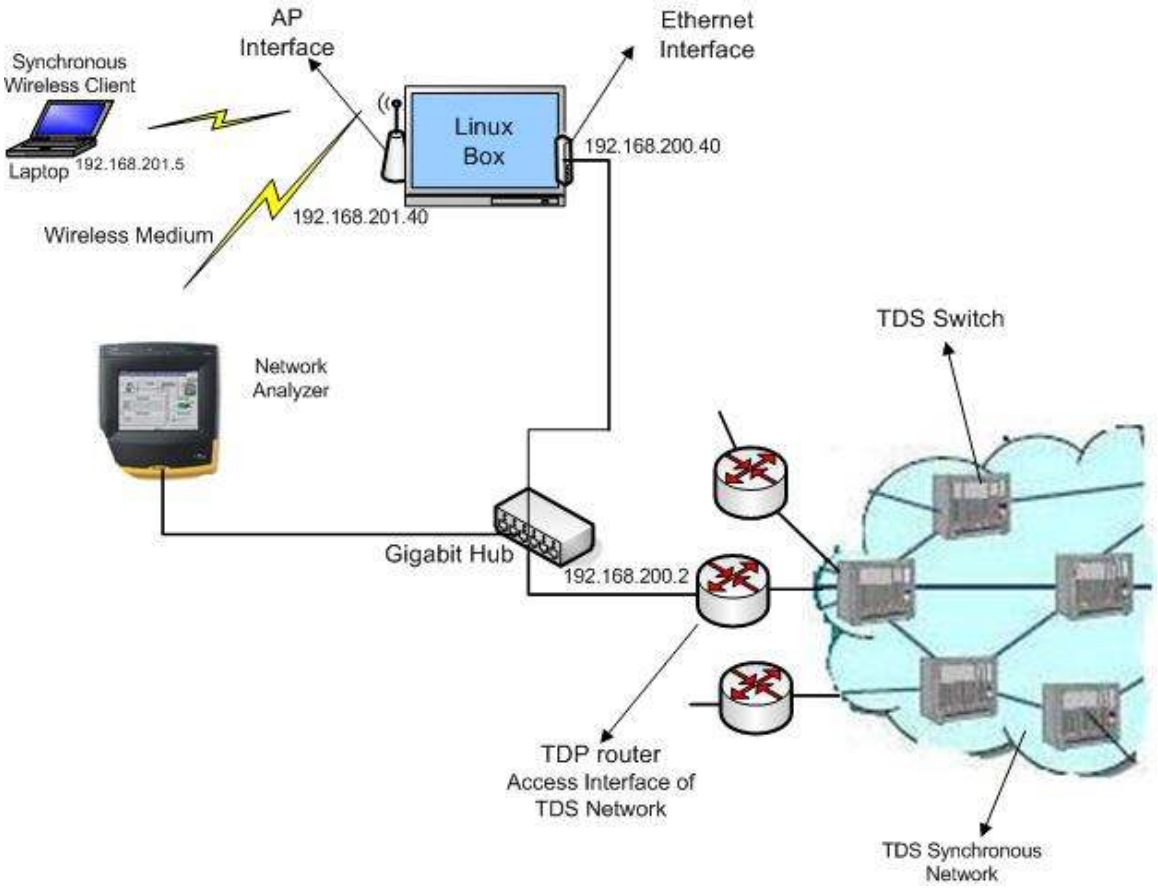



Figure 5.4: Experimental setup for OLA

Ethernet interface of network analyzer as well as Ethernet interface of Linux box which has wireless interface that would be act as AP. Delimiter will be traversed and routed inside linux kernel get into AP interface and will be turned to SWBA interrupt to start sending beacon frame. This beacon frame will be driven by the delimiter. From section 4.3.1, we already know that, it takes significant time to be traversed and routed of delimiter packet inside linux kernel and then generate and update beacon frame. So we can safely assume that, time, when beacon frame captured will be later than

time, when delimiter captured by network analyzer. Therefore, difference between these two times will indicate the promptness of beacon frame that is driven by a tick (delimiter), since delimiter packet is suppose to be gated into the Ethernet interfaces of both network analyzer and Linux box at the same time. We did experiment same scenarios like previous paragraph.

In all scenarios, although, beacon frame is captured in expected and considerably constant interval (with some microsecond +/- ), but delimiter driven beacon frames are captured delayed with around 50 ms. We referred this delay as phase shift. See figure 5.5 and 5.6

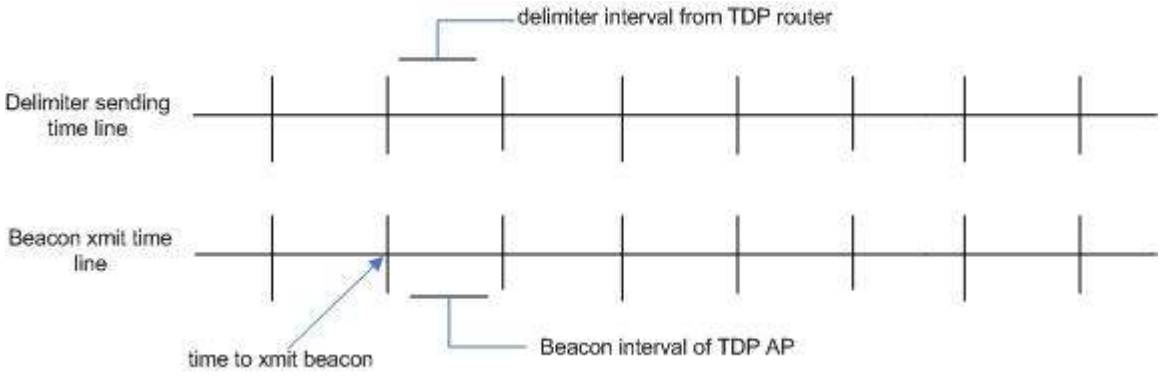



Figure 5.5: Ideal timeline of delimiter and beacon frame transmission in OLA

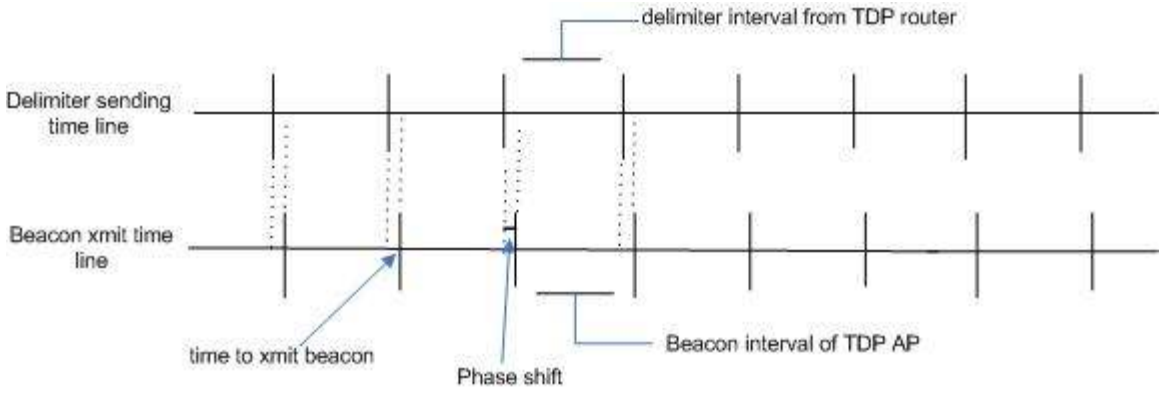



Figure 5.6: Experimental result of time line of delimiter and beacon frame transmission in OLA

If we analyze the cause of phase shift of beacon xmit in our experimental setup, it will be the added value of time of the following steps:

a) uncertain and unknown time for delimiter to make routing decision by linux iptables, IP layer of Linux kernel, to put pcmcia (AP) interface kernel buffer while jumping between two interface. See section 4.3.1
b) After confirming delimiter as tick, SWBA interrupt, time to send beacon now. Invoking some methods to generate beacon frame followed by update some dynamic filed according to current state.
c) unpredictable time to stay into the hardware data queue, see section 4.3.2.5.
d) time to transmit by TDP AP and receive by network analyzer the whole beacon packet ( around 1200 μs) at 1 Mb/s data rate (practically ) of device ( beacon frame size is 124 * 8 bit )

We measured amount of this delay time, phase shift, in few scenarios (see previous section).

***Scenario-1:*** TDP router configured with delimiter interval 200 ms (UTC)
So, ideally, beacon interval of TDP AP is 200 ms(UTC).
And it has been used data queue.

| Packet Number | Delimiter arrival time | Beacon arrival time | Phase Shift(ms) |
|---|---|---|---|
| 9 | 20:18:18.232138 | 20:18:18.294918 | 62.780 |
| 10 | 20:18:18.432146 | 20:18:18.494927 | 62.781 |
| 11 | 20:18:18.632155 | 20:18:18.694962 | 62.807 |
| 12 | 20:18:18.832163 | 20:18:18.894961 | 62.798 |
| 13 | 20:18:19.032172 | 20:18:19.094956 | 62.784 |
| . . | | | |
| 63 | 20:18:29.032095 | 20:18:29.094855 | 62.760 |
| 64 | 20:18:29.232103 | 20:18:29.294897 | 62.794 |
| 65 | 20:18:29.432113 | 20:18:29.494893 | 62.780 |
| 66 | 20:18:29.632120 | 20:18:29.694913 | 62.793 |
| 67 | 20:18:29.832128 | 20:18:29.894913 | 62.785 |
| 68 | 20:18:30.032139 | 20:18:30.094913 | 62.774 |
| 69 | 20:18:30.232146 | 20:18:30.294937 | 62.791 |

Table 5.1: Phase shift of OLA in experiment scenario1

***Scenario-2:*** TDP router configured with delimiter interval 100 ms (UTC). So, ideally, beacon interval of TDP AP will be 100 ms (UTC). In this experiment, we use data queue to transmit beacon. Table 5.2 shows the result.

| Packet Number | Delimiter arrival time | Beacon arrival time | Phase Shift(ms) |
|---|---|---|---|
| 69 | 19:15:12.927679 | 19:15:12.994834 | 67.155 |
| 70 | 19:15:13.027745 | 19:15:13.094762 | 67.017 |
| 71 | 19:15:13.127687 | 19:15:13.194732 | 67.045 |
| 72 | 19:15:13.227754 | 19:15:13.294594 | 66.840 |
| . . | | | |
| 100 | 19:15:16.027746 | 19:15:16.094774 | 67.028 |
| 101 | 19:15:16.127688 | 19:15:16.194760 | 67.072 |
| 102 | 19:15:16.227630 | 19:15:16.294781 | 67.151 |
| 103 | 19:15:16.327698 | 19:15:16.394709 | 67.011 |
| 104 | 19:15:16.427640 | 19:15:16.494774 | 67.134 |
| 105 | 19:15:16.527706 | 19:15:16.594532 | 66.826 |
| 106 | 19:15:16.599732 | 19:15:16.694784 | 95.052 |

Table 5.2: phase shift (delay) of above setup in OLA

***Scenario-3:*** TDP router configured with delimiter interval 200 ms (UTC)
So, ideally, beacon interval of TDP AP will be 200 ms(UTC). In this experiment, we use best effort hardware queue to transmit beacon.

| Packet Number | Delimiter arrival time | Beacon arrival time | Phase Shift(ms) |
|---|---|---|---|
| 6 | 20:35:44.276598 | 20:35:44.363769 | 87.171 |
| 7 | 20:35:44.476605 | 20:35:44.563765 | 87.160 |
| 8 | 20:35:44.676613 | 20:35:44.763765 | 87.152 |
| 9 | 20:35:44.876622 | 20:35:44.963793 | 87.171 |
| . . | . | | |
| 107 | 20:36:04.476575 | 20:36:04.563556 | 86.981 |
| 108 | 20:36:04.676585 | 20:36:04.763567 | 86.982 |
| 109 | 20:36:04.876593 | 20:36:04.963597 | 87.004 |
| 110 | 20:36:05.076601 | 20:36:05.163574 | 86.973 |
| 111 | 20:36:05.276611 | 20:36:05.363584 | 86.973 |
| . . | | | |
| 145 | 20:36:12.076523 | 20:36:12.163614 | 87.091 |
| 146 | 20:36:12.276531 | 20:36:12.363507 | 86.976 |
| 147 | 20:36:12.476538 | 20:36:12.563504 | 86.966 |

Table 5.3: phase shift of scenario-3 in OLA

## 5.2 Evaluation of Close Loop Approach:

We studied the implementation of close loop approach, the objective this solution is to make alignment of existing timing functionality of AP and make it time driven with TDP router. We should evaluate rigorously how much drift between TDP timer and chipset time is happening at every tick interval, moreover, in due course, how many alignment (if any) is needed for a particular time range. Secondly check out how precisely beacon frame transmission is time driven with TDP router. We should explain here the methodology of experiment to evaluate CLA solution how we prepared the result of experiment. We have used very common kernel debugging system *printk()* and redirect syslog ( */var/log/messages* ) daemon. Details of this description should be beyond here. We found the required data in a text file followed by parsing this text file writing some simple shell script. Graphical representation has been created using spreadsheet software (MS Excel).

***Experimental Configuration-1:***

TDP router is configured to send delimiter at 1 ms(UTC) interval and tick interval is 100 ms. So TDP AP is supposed to take every 100$^{th}$ delimiter as tick and measure the drift followed by an alignment, if any.

We have measured how much time is drifted with reference of TSF time per tick basis for few hundred alignments in experimental configuration- 1. The value of drift is not constant any more but it increased almost constantly. Table 5.4 shows some snaps of continuous tick number and drift. For instance in one alignment it was needed 1927 ticks to reach drift of 1009 µs. So, averagely, about 0.524 µs drift has been reached in 1 tick (100 ms)

| Continuous serial number of Tick | Tick Number | Drift (microsecond) with reference of TSF time |
|---|---|---|
| . . | . . | . . |
| 6264 | 1951 | -892 |
| 6265 | 1952 | -952 |
| 6266 | 1953 | -1010 **(alignment)** |
| 6267 | 1 | 66 |
| 6268 | 2 | 9 |
| 6269 | 3 | 75 |
| . . | . . | . . |

| | | |
|---|---|---|
| 6962 | 696 | -259 |
| 6963 | 697 | -318 |
| 6964 | 698 | -252 |
| . . | . . | . . |
| 7709 | 1443 | -749 |
| 7710 | 1444 | -683 |
| 7711 | 1445 | -741 |
| . . | . . | . . |
| 8192 | 1926 | -951 |
| 8193 | 1927 | -1009 **(alignment)** |
| 8194 | 1 | 66 |
| 8195 | 2 | 9 |
| . . | . . | . . |

Table 5.4 : drift per tick and showed the alignment

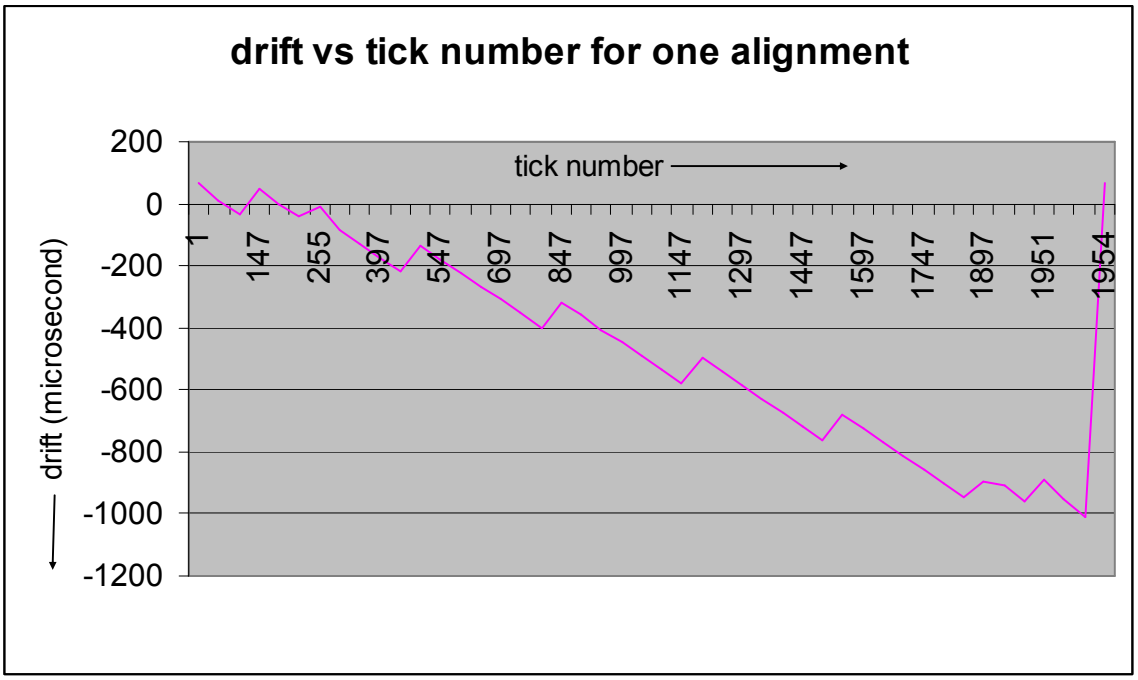



Figure 5.7: drift per tick for one arbitrary alignment

Figure 5.7 shows how drift is increased per tick. We choose arbitrary 3 alignments and for each alignment how drift gradually increase. For simplicity, we plot this graph by value of drift every 50$^{th}$ number tick,

otherwise graph won't be readable. Number of required ticks for one alignment is not constant but averagely almost equal.

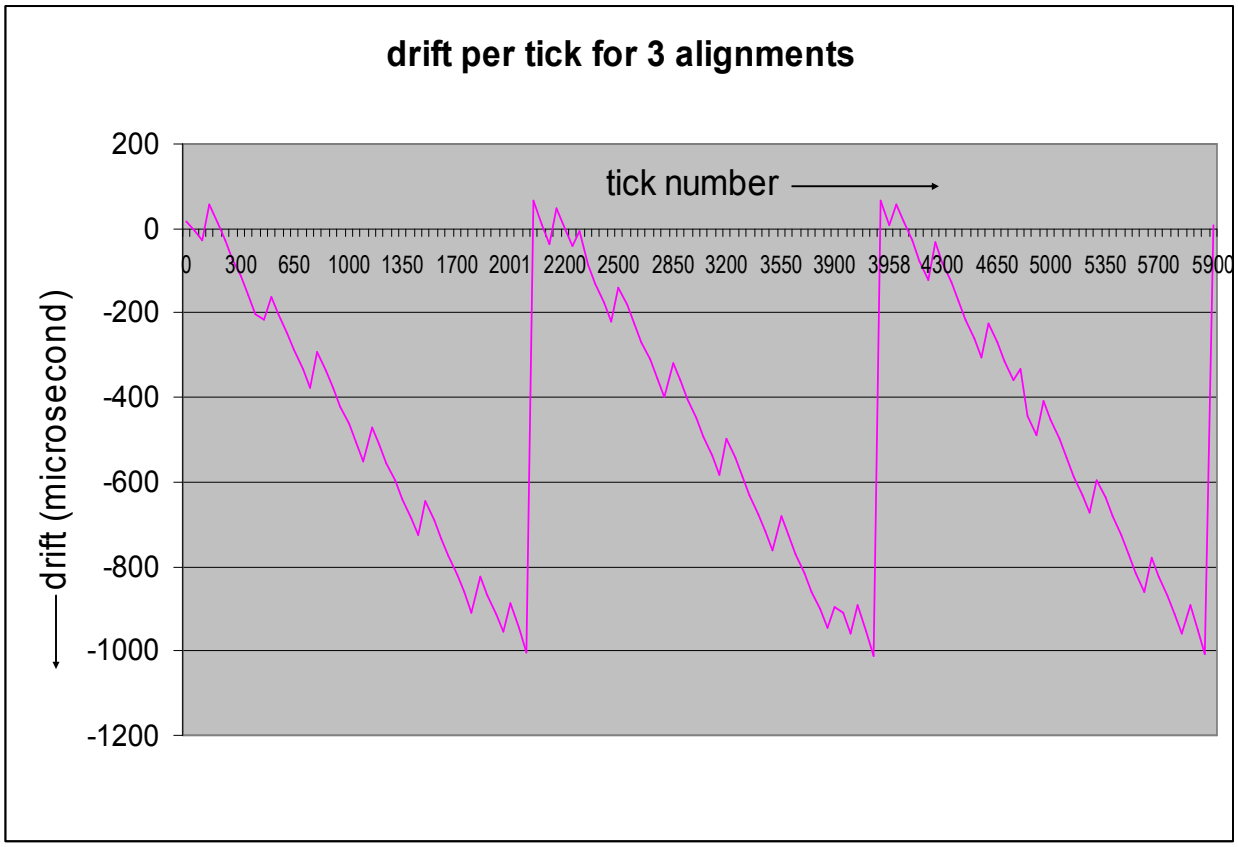



Figure 5.8: drift per tick for three consecutive arbitrary alignment

We evaluate the stability of TDP AP in respect with the required number of alignments in particular time duration. We configured the system as experimental configuration-1 and analyze the system performance two long time period, one is from 4:11:48 to 7:06:42 (system time) another one is from 9:18:43 to 15:23:20. we kept record the kernel time of system when an alignment has occurred, see Table 5.5 and Table 5.6 and how many alignment has occurred from start of these experiments.

| Tick Number | System Time (upto second) | Number of Alignments happened so far |
|---|---|---|
| 3864 | 4:11:48 | -- |

| | | |
|---|---|---|
| 11187 | 4:24:01 | 7 |
| 20306 | 4:39:13 | 12 |
| 29496 | 4:54:32 | 17 |
| 38615 | 5:09:44 | 22 |
| 47790 | 5:25:01 | 27 |
| 57019 | 5:40:24 | 32 |
| 66233 | 5:55:45 | 37 |
| 75438 | 6:11:06 | 42 |
| 90193 | 6:35:41 | 50 |
| 99486 | 6:51:11 | 55 |
| 108794 | 7:06:42 | 60 |
| 118089 | 7:22:11 | 65 |
| 127386 | 7:37:41 | 70 |
| 136752 | 7:53:17 | 75 |

Table 5.5: Number of alignment happened from system time from 4:11:48 to 7:06:42

If is important to know, how much time needs for one alignment (from Table 5.6):

| Time | No of Alignment |
|---|---|
| 15:23:20 | 116 |
| 9:34:33 | 6 |

This table shows 110 alignments occur in 5:48:47 hour, i.e.
( 5*60*60 + 48*60 + 47 ) = 20927 second .
So averagely, one alignment has occurred at every ( 20927 ÷ 110 ) = 190.2455 seconds

| System time (up to second) | Number of Alignments happened so far |
|---|---|
| 9:18:43 | -- |
| 9:34:33 | 6 |
| 9:50:22 | 11 |
| 10:06:10 | 16 |
| 10:21:57 | 21 |

| | |
|---|---|
| 10:37:47 | 26 |
| 10:53:47 | 31 |
| 11:09:55 | 36 |
| 11:25:55 | 41 |
| 11:41:46 | 46 |
| 11:57:34 | 51 |
| 12:13:22 | 56 |
| 12:32:19 | 62 |
| 12:48:07 | 67 |
| 13:03:56 | 72 |
| 13:19:46 | 77 |
| 13:35:38 | 82 |
| 13:51:28 | 87 |
| 14:04:06 | 91 |
| 14:19:55 | 96 |
| 14:35:47 | 101 |
| 14:51:38 | 106 |
| 15:07:30 | 111 |
| 15:23:20 | 116 |

Table 5.6: Number of alignment happened from system time 9:18:43 to 15:23:20

We have showed another important data, average number of tick needed to occur per alignment with respect to time. We have organized record in almost 6 hours, from 9:18:43 to 15:23:20 calculated average number of tick needed per alignment

| System time (upto second) | Elapsed Time | No. of Alignments in elapsed time | Average Number of Tick Needed per alignment |
|---|---|---|---|
| 9:18:43 | -- | -- | -- |
| 9:34:33 | 0:15:50 | 5 | 1899 |
| 9:50:22 | 0:15:49 | 5 | 1896 |
| 10:06:10 | 0:15:48 | 5 | 1894 |
| 10:21:57 | 0:15:47 | 5 | 1899 |

| | | | |
|---|---|---|---|
| 10:37:47 | 0:15:50 | 5 | 1921 |
| 10:53:47 | 0:16:00 | 5 | 1935 |
| 11:09:55 | 0:16:08 | 5 | 1920 |
| 11:25:55 | 0:16:00 | 5 | 1902 |
| 11:41:46 | 0:15:51 | 5 | 1896 |
| 11:57:34 | 0:15:48 | 5 | 1896 |
| 12:13:22 | 0:15:48 | 5 | 2275 |
| 12:32:19 | 0:18:57 | 6 | 1580 |
| 12:48:07 | 0:15:48 | 5 | 1897 |
| 13:03:56 | 0:15:49 | 5 | 1899 |
| 13:19:46 | 0:15:50 | 5 | 1905 |
| 13:35:38 | 0:15:52 | 5 | 1899 |
| 13:51:28 | 0:15:50 | 5 | 1516 |
| 14:04:06 | 0:12:38 | 4 | 2374 |
| 14:19:55 | 0:15:49 | 5 | 1902 |
| 14:35:47 | 0:15:52 | 5 | 1903 |
| 14:51:38 | 0:15:51 | 5 | 1904 |
| 15:07:30 | 0:15:52 | 5 | 1900 |
| 15:23:20 | 0:15:50 | 5 | -- |

Table 5.7: Average number of tick per alignment from system time 9:18:43 to 15:23:20

We found stability of alignment occurring is considerably constant (but not exactly constant, see figure 5.10 and 5.11), for example, averagely, one alignment took around 1909.636 tick during time period from 9:34:33 to 15:23:20, on the other way, in this experiment tick interval was 100 ms (UTC) so $(1909.636 \times 100)$ at every 190963.6 ms (UTC) one alignment has occurred.

We already know, alignment is initializing the beacon state with re-enable beacon timer, restart SWBA interrupt and as a side effect of it restart counting TSF timer from zero. However, it is remain absolutely hardware responsibility to start sending beacon frame which supposed to be gated into the beacon hardware queue before time to send beacon as next frame. Now we should think about the experiment how beacon frame is transmitting while alignment is occurring at averagely 190.2455 seconds interval. We did this experiment in following configuration:

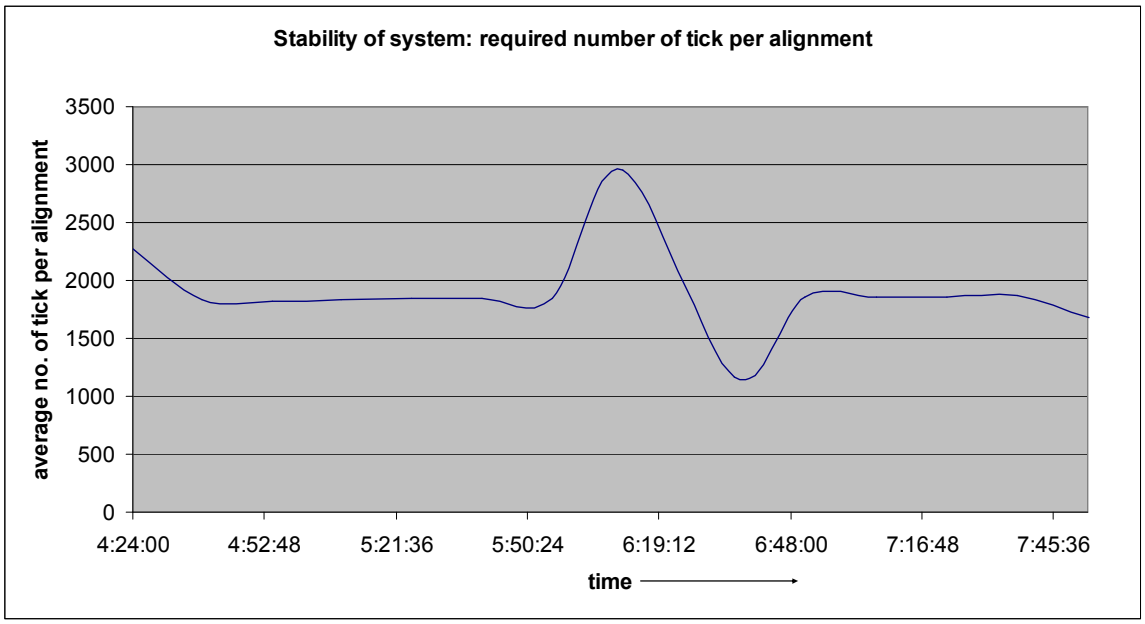



Figure 5.10: linear graphical presentation of average number of required tick per alignment

Figure 5.11: linear graphical presentation of average number of required tick per alignment from system time 9:18:43 to 15:23:20

***Experimental Configuration-2:***

TDP router is configured to send delimiter at 1 ms(UTC) interval and tick interval is 128. So TDP AP is supposed to take every 128$^{th}$ delimiter as tick and measure the drift followed by an alignment, if any. Beacon interval of TDP AP is 125 TU (i.e. 128 ms) and drift threshold is set to 1 ms (1000 μs ).

However, it is important to realize that, some time hardware stop sending only one 'queued' beacon if any alignment is happened exactly drift threshold, $d_t$, *before* the next beacon transmission time, see figure 5.13(a) and 5.13(b). On the other hand when an alignment happened exactly drift

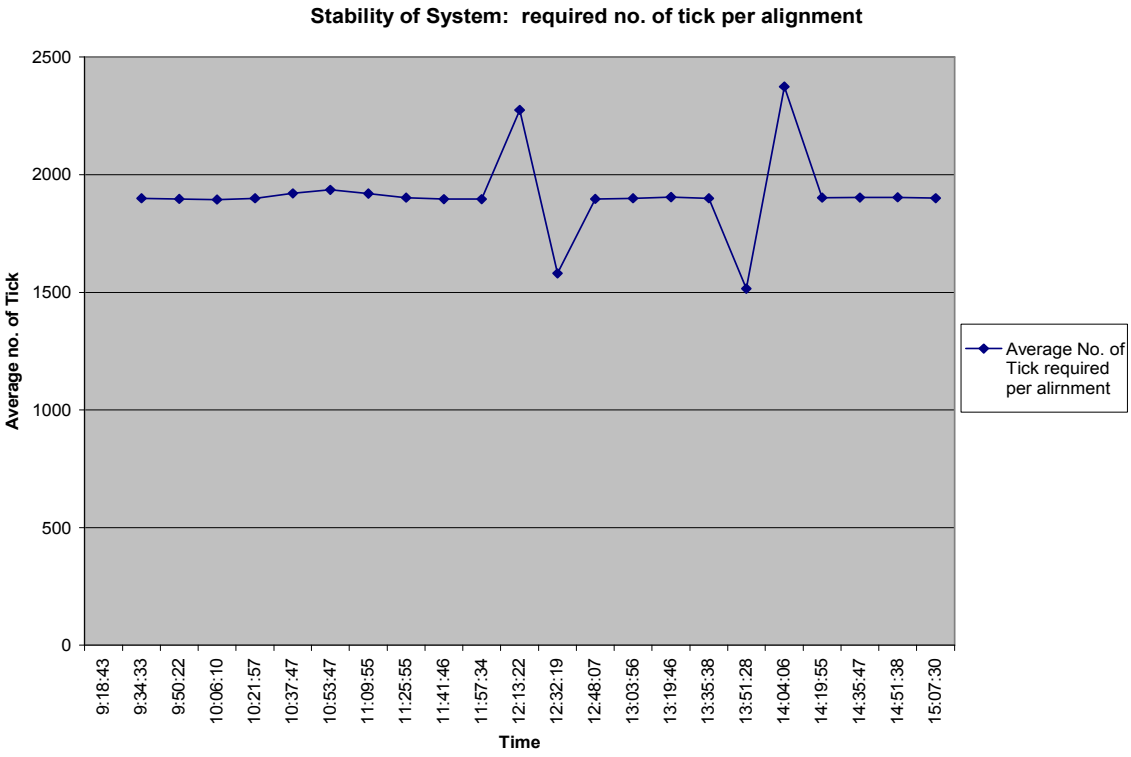
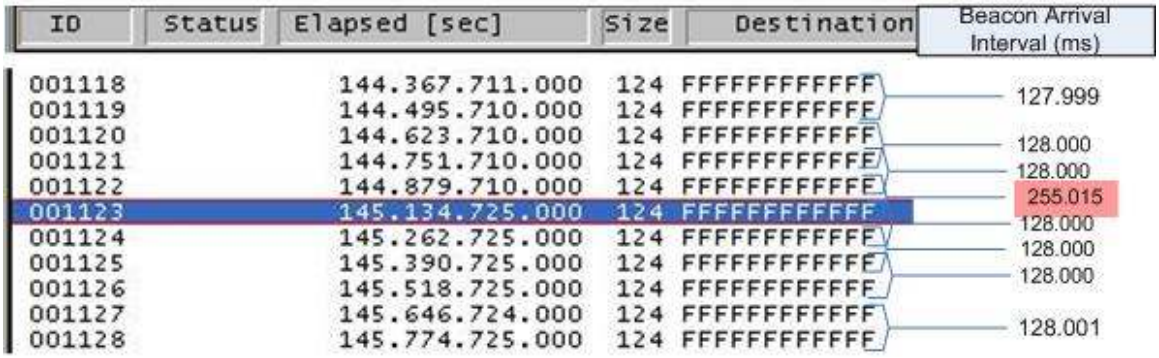



Figure 5.12 shows a snapshot of capturing beacon frame arrival time at network analyzer.

| ID | Status | Elapsed [sec] | Size | Destination | Beacon Arrival Interval (ms) |
|---|---|---|---|---|---|
| 001104 | | 142.575.713.000 | 124 | FFFFFFFFFFFF | |
| 001105 | | 142.703.713.000 | 124 | FFFFFFFFFFFF | 127.999 |
| 001106 | | 142.831.712.000 | 124 | FFFFFFFFFFFF | |
| 001107 | | 142.959.712.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 001108 | | 143.087.712.000 | 124 | FFFFFFFFFFFF | |
| 001109 | | 143.215.712.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 001110 | | 143.343.712.000 | 124 | FFFFFFFFFFFF | |
| 001111 | | 143.471.712.000 | 124 | FFFFFFFFFFFF | 127.999 |
| 001112 | | 143.599.711.000 | 124 | FFFFFFFFFFFF | |

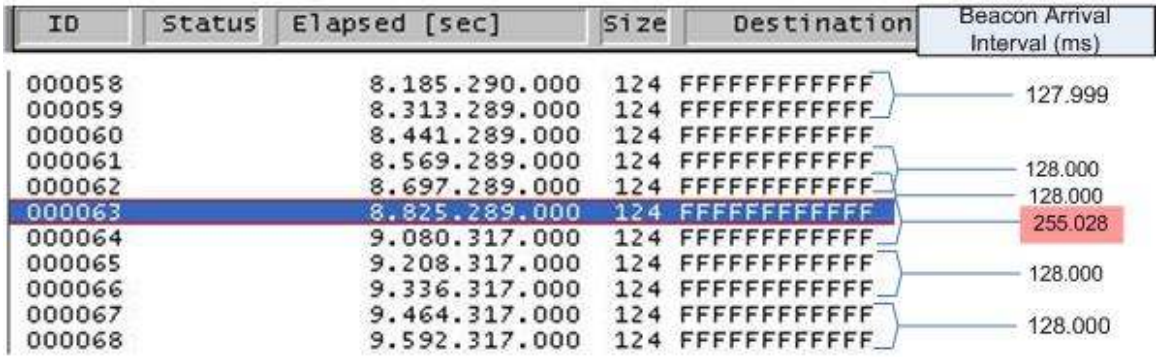
Figure 5.12: A snapshot of capturing beacon frame arrival time

threshold *'after'* the next beacon transmission time, within that drift threshold time, it should not be any 'queued' beacon frame. Since TSF timer start counting again, beacon arrival interval will be beacon interval time plus $d_t$ drift threshold.



| ID | Status | Elapsed [sec] | Size | Destination | Beacon Arrival Interval (ms) |
|---|---|---|---|---|---|
| 001118 | | 144.367.711.000 | 124 | FFFFFFFFFFFF | |
| 001119 | | 144.495.710.000 | 124 | FFFFFFFFFFFF | 127.999 |
| 001120 | | 144.623.710.000 | 124 | FFFFFFFFFFFF | |
| 001121 | | 144.751.710.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 001122 | | 144.879.710.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 001123 | | 145.134.725.000 | 124 | FFFFFFFFFFFF | 255.015 |
| 001124 | | 145.262.725.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 001125 | | 145.390.725.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 001126 | | 145.518.725.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 001127 | | 145.646.724.000 | 124 | FFFFFFFFFFFF | |
| 001128 | | 145.774.725.000 | 124 | FFFFFFFFFFFF | 128.001 |

Figure 5.13 (a): case 1, beacon frame arrival time while alignment occur



| ID | Status | Elapsed [sec] | Size | Destination | Beacon Arrival Interval (ms) |
|---|---|---|---|---|---|
| 000058 | | 8.185.290.000 | 124 | FFFFFFFFFFFF | |
| 000059 | | 8.313.289.000 | 124 | FFFFFFFFFFFF | 127.999 |
| 000060 | | 8.441.289.000 | 124 | FFFFFFFFFFFF | |
| 000061 | | 8.569.289.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 000062 | | 8.697.289.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 000063 | | 8.825.289.000 | 124 | FFFFFFFFFFFF | 255.028 |
| 000064 | | 9.080.317.000 | 124 | FFFFFFFFFFFF | |
| 000065 | | 9.208.317.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 000066 | | 9.336.317.000 | 124 | FFFFFFFFFFFF | |
| 000067 | | 9.464.317.000 | 124 | FFFFFFFFFFFF | 128.000 |
| 000068 | | 9.592.317.000 | 124 | FFFFFFFFFFFF | |

Figure 5.13 (b): case 2 of beacon frame arrival time while alignment occur

Figure 5.13(a), (b) shows our result of Experimental configuration-2 can consider as former case. When alignment happened, beacon arrival interval is 255 ms (128ms + 128ms – 1 ms) Since beacon interval 128 ms and alignment occur exactly before drift threshold (1000 µs).

We also got the result for later case, while alignment needs $d_t$ time after beacon transmission time (when TDP timer is ahead of TSF timer). Beacon arrival interval was 129.018 ms (128 ms + 1000 µs) with few µs error.

In the first case, a client associated with TDP AP is getting beacon frame after (about) two beacon interval time while an alignment occur. On the other hand, second case, client will be getting beacon frame drift threshold time before the beacon interval.

This incongruity, an irregularity of beacon frame transmission timing is only happen after (around) every 2000 beacon frames. So certainly it is not a serious problem for a station.

## *Chapter Six:*

# Discussion and Future Work

This chapter discusses some difficulty, we have faced, to implement the proposed solution on the existing hardware, associated with closed source code, known limitation of the implemented solutions with some experimental suggestion to extend this project work.

TDP router is considered as timing master of a particular time-driven wireless network. One of the major responsibilities of TDP router is to send a special UDP packet, delimiter to TDP AP at predefined time interval to reveal its existence and provide timing indication to time-driven wireless network.
TDP AP has to assume safely following things:

- There is no propagation delay between a delimiter is sending by TDP router until AP interface receiving that. So time between those two event suppose to virtually zero.
- TDP router is sending delimiter in correct time.
- Any delimiter should be not missing.

As describe above, timing indication for both solutions, CLA and OLA is being taken from confirming of arrival of delimiter packet. Delimiter frame itself, doesn't contain any timing information from TDP network. In the current implementation, delimiter packet has been let to be routed via IP layer (netfilter) of kernel, as described in section 4.3.1, that takes uncertain, unpredictable and non-constant time. We got the experimental result of OLA, section 5.1, phase shift of about 50 ms. One of the major cause of this delay is time to traverse delimiter packet. More over, from the evaluation of CLA, figure 5.7, amount of drift for each tick is not uniformly equal. Since in the current TDP router implementation, the delimiter packet doesn't have any timing value of current time of TDP router, so clearly it is NOT expected to let delimiter be routed inside kernel iptables. This work was outside of scope of this project, can be considered as one of the major future work to make wireless network time-driven.

Recent Linux kernel has rich 'modularity' feature that leads inter-module communication is more dynamic and easier [18]. As a kernel module, wireless interface driver can get indication of arriving delimiter packet directly from Ethernet device driver module. There is several ways to implement this idea [18], which certainly enable to avoid IP layer routing

time of delimiter packet. This could be the starting point of future work of this project.

One of major drawback of current implementation of TDP router, working as timing master of TDP AP, is sending delimiter without putting any timing value in the frame. That makes complicated the implementation. We believe modification to TDP router should be part of future work that should facilitate CLA implementation easier.

Although, the GPS card can support an interrupt granularity (e.g.1024 μs), but the current implementation of TDP router specify delimiter sending granularity (e.g. TF duration 125μs for slow link) has to be submultiple of the second (because the TDP supercycle duration is 1 UTC second). So, currently, it is possible to configure delimiter granularity (for example) 1000 μs, 2000 μs, 500 μs, 250 μs etc. On the other hand, since the design of existing WLAN chipset is certainly followed IEEE802.11 standard that specify the beacon interval will be in Time Unit (TU), a measurement of time equal to 1024 μs. Therefore, we got an unavoidable difficulty while aligning beacon interval of TDP AP with delimiter interval since WLAN chipset allows only TU (say 100 TU) which is 100 multiple of 1024 μs. This constraint limits the freedom to set beacon interval of TDP AP with possibly two values, practically:

125 TU = 128 ms or

167 TU = 171.008 ms (with 8 μs error)

Theoretically, closed loop solution is not ideally time-driven with TDP timer. Once an alignment happened TSF timer start counting again, sending beacon frame 'aligned' with TDP timer with some μs before or after because of the drift. However, this difference will never cross drift threshold. This phenomenon has described in figure 6.1 and 6.2.

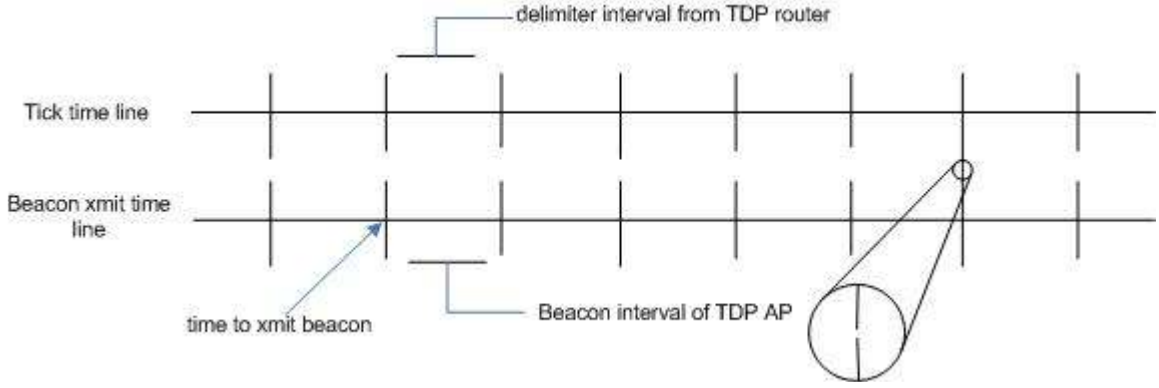



Figure 6.1: Ideally tick time and beacon frame xmit time line

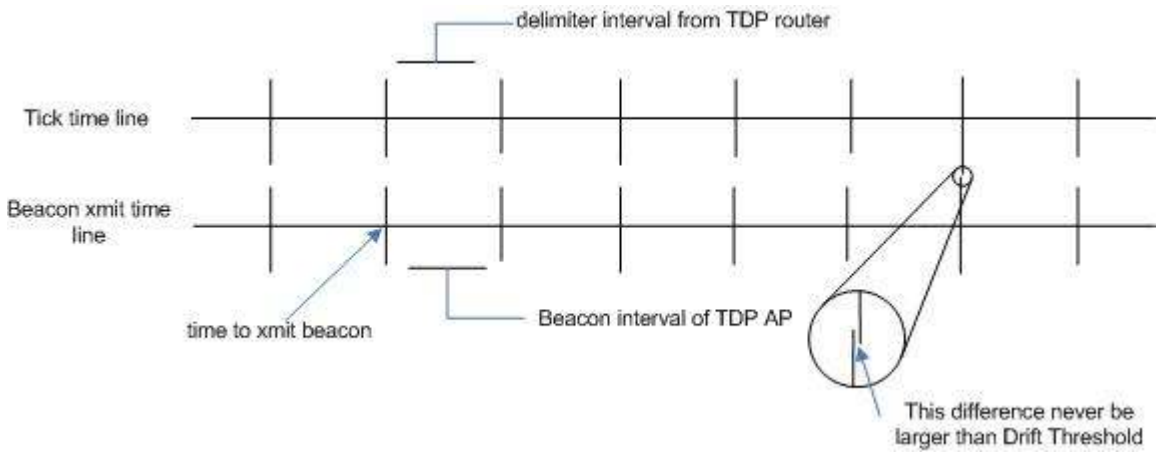



Figure 6.2: tick time and beacon xmit time line with drift in CLA

What is the compensation, we have to pay for an alignment. During alignment we had to invoke an HAL routine (closed source) that (re)start counting TSF timer. But with the current hardware, this is not happened for normal operation except when device experienced a fatal error needs to reset chip or switch to new channel. Since we don't have code of this routine or even any official specification from vendor, we can not try to avoid this limitation of the current implementation.

More over, we got the result, see section 5.2. b) Experimental Configuration-1, in the first case, when alignment occurred `drift threshold`, $d_t$, time before the nextTBTT, a client associated with TDP AP is getting beacon frame after (about) two beacon interval time. On the other hand, second case, when alignment occurred $d_t$ time just after the last beacon transmission time, client will be getting beacon frame $d_t$ time after the beacon interval.

From the description of CLA, we are informed that, beacon state initialization needs to invoke a hal function (*ath_hal_beaconinit*) which is a closed sourced function provided by chipset vendor. Unfortunately, like other vendors, Atheros did not provide any kind of specification or documentation of binary module (HAL) for open source community. We have been verified from open source community, the major task of this function is to set chipset with two major beacon parameters, nextTBTT, beacon interval and (re)start counting TSF timer from zero and stop sending already queued beacon since this function also needs to enable beacon timer of chipset and another logic is what will be timestamp value of queued beacon frame. It should mention here again timestamp value will be TSF time when first bit of beacon frame goes to physical layer of the

device. So timestamp value of that beacon will be $d_t$ µs Unfortunately, we don't have any official reference of this specification from chipset vendor.

The above incongruity, an irregularity of beacon frame transmission timing is only happen for one beacon after (around) every 2000 beacon frames. So certainly it is not a serious problem for a station.

Other than above known limitation, we have seen the CLA implementation has achieved up to the mark. We believe, these limitations can be worked out by choosing appropriate platform that is solely open source. For example compare with Madwifi with Atheros provided closed HAL; bcm43xx, as described in section 4.1.c, should be more useful platform to implement solutions for this project [11]. We would have chosen this platform; however, unfortunately, bcm43xx was not stable that time. But at the time writing this report, bcm43xx, which has no binary crap, is stable and should be considered as the platform of future work.

We have realized that, it may not possible to achieve goal of OLA completely without having microcontroller of WLAN chipset or full code of lower level MAC. That's why we called this implementation as feasibility study or emulation of OLA. Evaluation of current OLA implementation (section 5.1) on current hardware, we have found significant good result in the case of consecutive beacon interval. However, phase shift between TDP timer and TSF timer is significantly more, that makes this implementation complicated.

As part of future work, it can be possible to use the current implementation of CLA with some added functionality as describe above to port into embedded Linux kernel with different CPU architecture for a commercial access point. This work needs to do following steps, e.g. some modification of current implementation of CLA, cross compilation for target architecture in consider with presence of Linux kernel in the target AP. Vendor provided binary part, HAL, of this driver specify that, it supports couple of architecture e.g. x86, ARM, MIPS, CRIS etc. A good example can be found [24] [25].

## Conclusion

This work has provided a kernel-based prototypal solution for wireless extension of time-driven switching network on the existing hardware for 802.11 protocol stack. The Implementation envisaged the emulation of UTC-synchronized beacon frame generation of an access point that helped to design of a synchronous packet scheduler. The implementation has been done directly in kernel space of Linux operating system that manages network layer and partially MAC layer. Implementation of proposed approaches was difficult since most of the time-critical activities of management frame controlled by firmware of chipset and vendor specified closed source code. However, the goal of the close-loop has been achieved with a known limitation. Experimental result shows that, it is non-trivial to implement open-loop approach on existing hardware and standard.

# Reference:


[1] Y. Ofek, D. Agrawal, M. Baldi, G. Marchetto, V. T. Nguyen, D. Severina, "A Novel Approach for Supporting Streaming Media: Design, Implementation and Experiments", *ACM Multimedia 2006*, Santa Barbara, CA USA, October 22-28, 2006

[2] Y. Ofek, M. Baldi, "Multi-Terabit/s IP Switching with Guaranteed Service for Streaming Traffic," *IEEE INFOCOM 2006 High-Speed Networking Workshop*, Barcelona (Spain), Apr. 2006.

[3] M. Baldi, Y. Ofek, “End-to-end Delay Analysis of Videoconferencing over Packet Switched Networks”, in *IEEE INFOCOM April 1998*.

[4] M. Baldi, Y. Ofek, “Common Time Reference for Interactive Multimedia Applications”, *IEEE International Conference on Multimedia & Expo (ICME2000)*, New York, NY, USA, July-Aug. 2000, pp. 1679-1682

[5] Yoram Ofek, IP-Flow project description, http://ip-flow.dit.unitn.it/

[6] Y. Ofek, M. Baldi, “Blocking Probability with Time-driven Priority Scheduling”, *SCS Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2000)*, Vancouver, BC, Canada, July, 2000.

[ 7 ] The official website of OpenWrt , http://openwrt.org/ last accessed October, 2006

[8] Jouni K. Malinen, Host AP driver: official site and mailing list, September, 2006 http://hostap.epitest.fi/

[9] Intersil Corporation, the provider of PRISM Wireless LAN chipset, now maintained by Conexant Systems, Inc., www.conexant.com/ , http://www.intersil.com/ , September, 2006

[10] Open Source Project for Linux kernel driver for Broadcom bcm43xx wireless chipset, http://bcm43xx.berlios.de/

[11] IEEE 802.11e task group E, specification of 802.11e standard.
[12] Paul Gortmaker, "Linux Ethernet-Howto", v2.8, Oct 29, 2000

[13] V. Gunffens, " Path of a Packet in the Linux Kernel", Apritl 2003.

[14] Rusty Russell, Harald Welte, " Linux netfilter Hacking HOWTO ", July, 2002
[15] F. Baker. *RFC1812 - Requirements for IP Version 4 Routers*, June 1995.

[16] David A. Rusling. *The Linux Kernel*. 1996, Chapter 10.

[17] Stephen Hines, Bartow Wyatt, and J. Morris Chang, "Increasing Timing Resolution for Processes and Threads in Linux ".
[18] Jonathan Corbet, A. Rubini, G. Kroah-Hartman Linux Device Drivers, Third Edition, Chapter 17: Network Drivers, O'Reilly.
[19] MadWifi project at sourceforge site http://sourceforge.net/projects/madwifi/
[20] The official webpage of Atheros Communications, www.atheros.com

[21] MAC Sub Layer Management Entity, Specification of IEEE 802.11 standard, section 11.1, 1999
[22] Atheros Communication Inc. Wireless product factsheet, www.atheros.com

[23] Guido Marchetto, Masters Thesis, "Prototypal Implementation Of A Time-Driven Priority Router", January, 2005

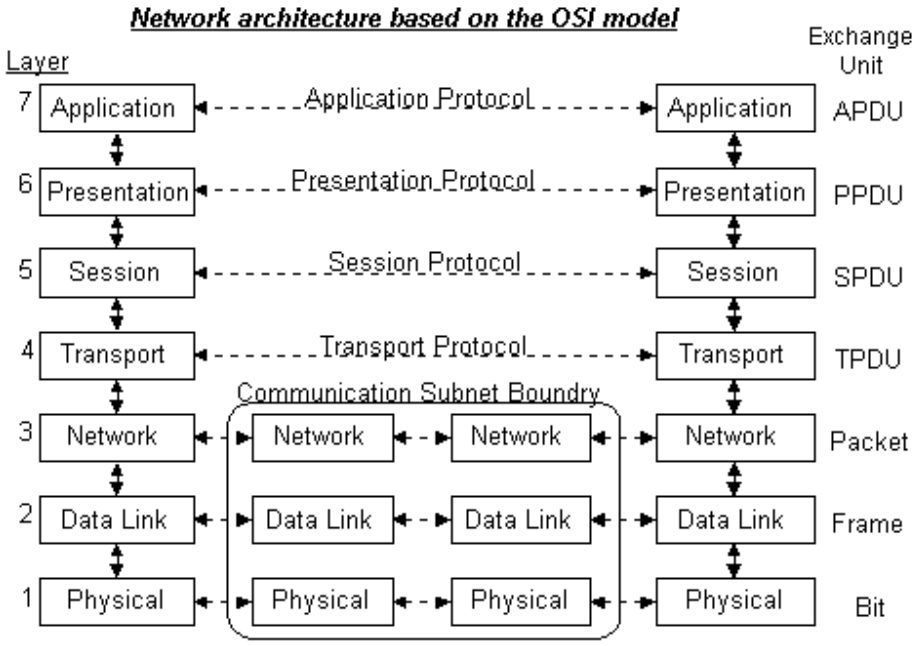
[24] OpenWRT forum, http://forum.openwrt.org/viewtopic.php?id=5585 last access November 03, 2006
[25] Cross-compilation toolkit, http://www.scratchbox.org/ last access November 03, 2006

[26] Open source project of shorewall, netfilter configuration tool, http://www.shorewall.net/ last access October, 2006

[27] Nichols, K., Blake, S., Baker, F. and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", *RFC 2474*, December 1998.

# Appendix

### Appendix A.1: The OSI Layered Model




### Appendix A.2:

The configuration file set (valid for Ethernet –Wireless interconnection) for shorewall, used for enable IP routing inside Linux kernel iptables, is written below:

*Zones file:*

```
ZONE   TYPE   OPTIONS      IN                    OUT
                           OPTIONS               OPTIONS
```

```
-----------------------------------------------
fw    firewall
wifi  ipv4
cable ipv4
```

*Interfaces file:*

```
ZONE    INTERFACE          BROADCAST      OPTIONS
-----------------------------------------------
wifi        ath1             detect
cable       eth0             detect
```

*Policy file:*

```
SOURCE      DEST        POLICY
-----------------------------------------------
cable       wifi        ACCEPT
wifi        cable       ACCEPT
all         $FW         ACCEPT
all         all         REJECT
```