Lessons Learned Constructing the NG-Mesh Wireless Test-Bed

LESSONS LEARNED CONSTRUCTING THE NG-MESH WIRELESS TEST-BED

BY

STANLEY WAI-KWONG NG, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2011) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	Lessons Learned Constructing the NG-Mesh Wireless Test-Bed
AUTHOR:	Stanley Wai-Kwong Ng B.Eng., (Electrical & Computer Engineering) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Ted H. Szymanski
NUMBER OF PAGES:	xvi, 122

Abstract

This thesis presents the lessons learned from building an IEEE 802.11 wireless mesh network (WMN) test-bed. Each network node consists of a Linux processor with multiple IEEE 802.11b/g transceivers operating in the 2.4 GHz band. Each transceiver consists of a medium access control (MAC) and base-band processor (BBP) in addition to a radio. A device driver was modified to control some of the key transceiver functions. The test-bed's Wi-Fi interfaces can be programmed to implement any mesh communication topology. All Wi-Fi interfaces use omni-directional antennas and the IEEE 802.11b operation mode.

The test-bed design is easily extendable to incorporate newer Wi-Fi technologies. Measurements of co-channel interference in each Wi-Fi channel including received signal strength (RSS) and signal-to-interference-and-noise ratio (SINR) are presented. The AutoMin algorithm was developed in order to use the captured physical layer (PHY) metrics to avoid Wi-Fi congestion during test-bed operation. A comparison of a software-based spectrum analyzer to a commercial one is described. Key Wi-Fi functions in the Ralink driver source code are explored in depth. The compliance of the Ralink chip-set to the IEEE 802.11b spectral mask was verified. The maximum driver-induced retuning rate for the popular Ralink radio was found experimentally. This data can be used to optimize the performance of IEEE 802.11 WMNs.

Acknowledgements

I would like to express my deepest gratitude to my supervising professor, Dr. Ted H. Szymanski, for providing me with the opportunity to work and learn with him for the past two years and for all of his support, understanding, and patience during this time.

I would also like to acknowledge Dr. Terence D. Todd and Dr. Dongmei Zhao for assuming the roles of examiners for my thesis defense and providing me with their invaluable feedback. I would like to thank the graduate students and staff, especially Cheryl Gies of the Faculty of Electrical and Computer Engineering at McMaster University for their assistance during my study and research.

Finally I extend my thanks to my parents Amy and Jack, my sisters Stephanie and Sarah, my girlfriend Michelle, and my friends for their love and support throughout my academic journey.

Terms & Abbreviations

ACK Acknowledgement

AMRR Adaptive Multi-Rate Retry

AODV Ad-Hoc On-Demand Distance Vector Routing

AP Access Point

BBP Base-Band Processor

BW Bandwidth

CQI Channel Quality Indicator

CTS Clear-To-Send

CLI Command Line Interface

CPU Central Processing Unit

DHCP Dynamic Host Configuration Protocol

DYMO Dynamic MANET On-Demand Routing

DSR Dynamic Source Routing

EEPROM Electrically Erasable Programmable Read-Only Memory

EOL End-Of-Life

FCC Federal Communications Commission

FPGA Field-Programmable Gate Array

HAL Hardware Abstraction Layer

HDD Hard Disk Drive

IARIA International Academy, Research, and Industry Association

IBSS Independent Basic Service Set

IC Industry Canada or Integrated Circuit (Depending on context)

ICMP Internet Control Message Protocol

IEEE Institute of Electrical and Electronic Engineers

IP Internet Protocol

ISM Industrial, Scientific, and Medical

MAC Medium Access Control

MANET Mobile Ad-Hoc Network

MIMO Multiple In Multiple Out

MPDU MAC Protocol Data Unit

MS Mobile Station

NIC Network Interface Card

NG-Mesh Next-Generation Mesh

ODMRP On-Demand Multicast Routing Protocol

OLPC One Laptop Per Child

OLSR Optimized Link State Routing Protocol

OS Operating System

PC Personal Computer

PER Packet Error Rate

PMF Probability Mass Function

PRR Packet Retry Rate

PHY Physical Layer

QoS Quality-of-Service

RAM Random Access Memory

RHEL Red Hat Enterprise Linux

RF Radio Frequency

RSS Received Signal Strength

RSSI Received Signal Strength Indicator

RTS Request-To-Send

RTT Round-Trip-Time

SINR Signal-to-Interference-and-Noise Ratio

SNR Signal-to-Noise Ratio

SSID Service Set Identifier

TCP Transmission Control Protocol

TDM Time Division Multiplexing

UDP User Datagram Protocol

USB Universal Serial Bus

VoIP Voice over Internet Protocol

VM Virtual Machine

WEP Wired Equivalent Privacy

Wi-Fi Wireless Fidelity

WLAN Wireless Local Area Network

WMN Wireless Mesh Network

WPA/WPA2 Wi-Fi Protected Access (Version 1 and 2)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Multi-hop infrastructure WMNs represent an inexpensive network access technology, which can potentially provide last-mile accessibility to much of the world. Industry estimates that by 2020 there will be several billion wireless devices providing a range of new services. WMNs represent a promising infrastructure for supporting these wireless devices as well as providing general communications infrastructure for both rural and urban settings. Capacity and scalability are key challenges for such networks. Multi-channel multi-band meshes can use multiple radio channels in multiple frequency bands to improve system capacity and throughput. Optimizing the design of such WMNs requires statistics on physical layer noise and co-channel interference. To date there have been very few publications that present detailed physical layer measurements of co-channel interference encountered in practical WMN test-beds. To address this problem, this thesis presents a small Wi-Fi test-bed composed of IEEE 802.11b nodes operating in ad-hoc mode, called the Next-Generation (NG) Mesh. Each network node contains multiple Wi-Fi transceivers controlled in a Linux environment. To study co-channel interference between Wi-Fi channels in the 2.4

GHz industrial, scientific, and medical (ISM) band, physical layer measurements of RSS and SINR are reported. The contributions of this thesis are summarized as follows.

- An in-depth analysis of the popular Ralink driver source code for controlling key network interface card (NIC) functionalities such as implementing radio transmission power control and Wi-Fi channel switching is presented.
- A comparison between a commercial and a software-based 2.4 GHz spectrum analyzer is presented.
- A network congestion avoidance algorithm called AutoMin was developed.
- An easily extendable test-bed called NG-Mesh was developed.
- The compliance of the consumer grade rt2570 Ralink chip-set to the IEEE 802.11b spectral mask while operating under practical settings was verified.
- The maximum driver-induced retuning rate of the popular Ralink radio was found experimentally.
- A detailed analysis of IEEE 802.11b adjacent channel interference is presented.

1.1 IEEE 802.11b Spectral Mask Requirements

The Institute of Electrical and Electronic Engineers (IEEE) has specified guidelines to minimize co-channel interference in the IEEE 802.11 standard [1]. Figure 1.1 illustrates the eleven North American Wi-Fi channels in the 2.4 GHz band. Each channel spreads its signal power over 22 MHz while their peaks are spaced 5 MHz

apart. Consequently, only channels that are spaced at least five Wi-Fi channels apart such as 1, 6 and 11 are logically orthogonal (i.e., their signals are non-overlapping). The IEEE 802.11b spectral mask shown in Figure 1.2 mandates a drop of at least 30 dBm at a displacement of 11 MHz (i.e., two channels) away from the peak of an active channel. An additional drop of at least 20 dBm is required at double this distance. These requirements ensure adequate attenuation between IEEE 802.11b channels in the 2.4 GHz band. However, the spectral mask requirements apply to a single device tested in isolation and may not apply to a real network deployment due to interference from multiple nearby networks and other microwave sources.

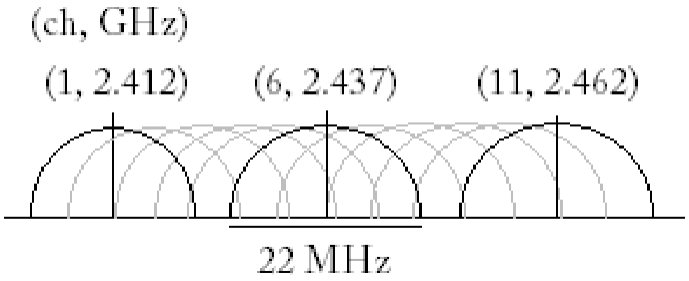


Figure 1.1: The 2.4 GHz band.

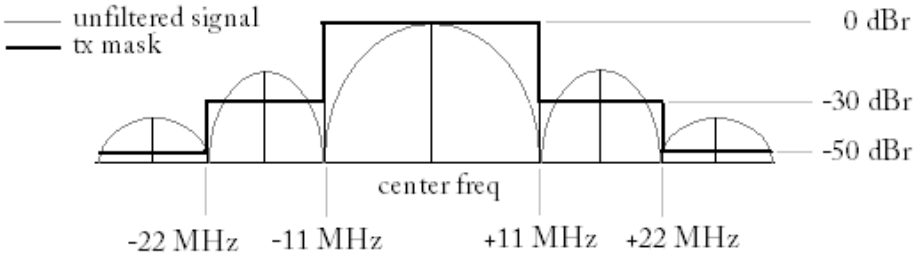


Figure 1.2: The IEEE 802.11b spectral mask.

1.2 Thesis Organization

In order to test for co-channel interference in a practical IEEE 802.11 network deployment, large file transfers are performed on each Wi-Fi channel within the NG-Mesh test-bed while the received signal strength indication (RSSI) and SINR on all Wi-Fi channels are measured. Using 15 dBm of transmission power over very short (i.e., 0.5 meter) wireless links and the maximum IEEE 802.11b data-rate of 11 Mbps, SINRs in the range of 50 to 60 dBm were consistently reported in the presence of several remote IEEE 802.11 networks. Given the lack of published data on physical layer measurements in realistic network test-beds, this data may help optimize network designs.

The remainder of this thesis is organized as follows. Chapter 2 summarizes other wireless networking test-beds that have emerged in the literature over the last decade. Chapter 3 describes the design of our NG-Mesh test-bed. Chapter 4 presents our experiments and an analysis of the results. Chapter 5 concludes this thesis with some lessons learned and suggestions for future work.

Chapter 2

Related Work

This chapter summarizes other recently proposed wireless test-bed designs that were published in the literature over the past ten years.

2.1 Routing-Centric Test-Beds

In [2], a network of *One Laptop Per Child* (OLPC) laptops [3] was used to create a test-bed. The nodes implement the IEEE 802.11s mesh networking amendment atop a Marvell IEEE 802.11b/g Wi-Fi chip-set. Multi-hop mesh topologies were configured while the probability mass function (PMF) of the round-trip-times (RTTs) from internet control message protocol (ICMP) pings was computed. The routing of traffic between network nodes was based on the ad-hoc on-demand distance vector (AODV) routing protocol.

In [4], a Wi-Fi network was installed in a natural reserve spanning several kilometers in order to collect ecological data. Each network node consists of multiple Atheros IEEE 802.11b/g Wi-Fi transceivers capable of multi-channel communication.

These interfaces depend on device driver interaction with a hardware abstraction layer (HAL) in order to access the hardware functions of the chip-set at the physical layer. The optimized link state routing (OLSR) protocol was used. All nodes were solar-powered with a mixture of directional and omni-directional antennas. Packet loss was recorded. The test-bed was used to investigate the correlation between packet error rate (PER) and RSSI in order to evaluate improvements in rate control algorithms and routing protocols.

In [5], a virtualized environment was created for multi-hop communication between network nodes. Each ad-hoc node had two IEEE 802.11a/b/g Atheros-driven Wi-Fi interfaces with omni-directional antennas. The OLSR routing protocol was used. Average throughput and packet loss was recorded.

In [6], OLSR routing of user datagram and transmission control (UDP and TCP) protocol traffic was used in a mobile ad-hoc network (MANET). The Mobile, stationary, and relay nodes used Ralink chip-sets and drivers. Throughput, RTT, and packet loss were the metrics considered.

In [7], a formula was derived for approximating the required amount of signal attenuation for a small-scale Wi-Fi network (i.e., spanning a few meters) to emulate a large-scale version accurately. The transmit power (i.e., and therefore range) and the inter-node spacing between the wireless interfaces were scaled down accordingly. Variable attenuators (i.e., Broadwave 751-002-030 devices) were used. Two Atheros IEEE 802.11b/g Wi-Fi interfaces were used at each network node. This test-bed was built to evaluate the routing of UDP packets using the OLSR routing protocol. Link quality, throughput, and data-rates were measured.

In [8], voice-over-internet protocol (VoIP), video, and text data transmission was

evaluated in a WMN test-bed. IEEE 802.11 nodes with Intersil chip-sets were used with the OLSR routing protocol. Mobility and handoff tests were also performed.

In [9], a network was used for testing layer 2 routing. IEEE 802.11b wireless local area network (WLAN) interfaces with Intersil and Lucent chip-sets were used. ICMP RTTs were captured and VoIP applications were tested in this test-bed.

In [10], a WMN was created to study various routing and channel selection protocols. The AODV routing protocol was used in multi-hop tests. Each network node consisted of two NICs. A throughput comparison between network nodes with single and dual wireless interfaces was presented.

In [11], a routing scheme for WMNs was presented for re-routing traffic to bypass non-operational nodes. Each network node had a Ubiquity IEEE 802.11g interface. Throughput was measured against node failures. The ability of the proposed algorithm to route traffic around failed nodes was also quantified.

In [12], a Wi-Fi test-bed was used to evaluate routing protocols for service discovery. The OLSR and dynamic MANET on-demand (DYMO) routing protocols were used. Each network node had two Atheros IEEE 802.11a/b/g interfaces. Delay and total number of services discovered were measured against the number of hops in the communication path.

In [13], a Wi-Fi test-bed was built for cognitive radio network protocol development. The OLSR routing protocol was used. Each network node was equipped with an Atheros IEEE 802.11a/b/g interface that was driven by MadWifi drivers. A new link metric that takes into account the link's maximum data-rate was proposed for improved route selection in multiple data-rate networks.

In [14], an ad-hoc Wi-Fi test-bed consisting of mobile terrestrial and aerial nodes

was developed. Lucent IEEE 802.11b interfaces were used at all network nodes with dynamic source routing (DSR).

In [15], a MANET was developed to carry out unicast and multicast routing. The nodes operated with Lucent wireless chip-sets. The routing protocols used were DSR and on-demand multicast routing protocol (ODMRP).

In [16], MANET design strategies were presented in support of multi-hop routing protocol development. IEEE 802.11b NICs were used. Network nodes were wired together to eliminate the interference effects of the radio environment. In a second configuration, the Wi-Fi interfaces were shielded and used with signal attenuators for close range wireless communication.

In [17], an ad-hoc network for testing routing protocols was created. Lucent wireless chip-sets were employed at each network node. Signal quality was used in routing decisions. A path loss model was used for correlating signal quality with the distance between the nodes in order to compute a virtual mobility metric. Both OLSR and AODV routing protocols were evaluated. Successful ICMP pings and network connectivity rates were measured. Packet loss versus hop count was analyzed.

In [18], a wireless ad-hoc network was introduced to study routing protocols. Mobility was emulated using signal attenuators. The DSR routing protocol was used with IEEE 802.11b Wi-Fi chip-sets at each network node. The percentage of packets received and acknowledgement (ACK) timeouts were plotted.

2.2 Performance-Centric Test-Beds

In [19], a campus-wide WMN was described to provide internet access to over one thousand students. Each network node had three omni-directional IEEE 802.11b/g

Wi-Fi interfaces with Ralink chip-sets. The OLSR routing protocol was used. Throughput and the number of faulty links were tracked over time. Due to the close proximity of the mesh nodes, a high degree of co-channel interference was present. Although physical layer metrics such as the signal-to-noise ratio (SNR) were collected, no solution was proposed to mitigate the interference.

In [20], an ad-hoc network was used for developing and testing a modified version of the IEEE 802.11 back-off algorithm. A combination of Ralink, Intersil, and Atheros IEEE 802.11b/g chip-sets were used in this test-bed. Average delay and throughput was measured and compared to traditional IEEE 802.11 retransmission algorithms.

In [21], a mesh network that was composed of multiple quickly deployable relay nodes was developed to provide emergency communication services in real-time. Each node had a Marvell IEEE 802.11b/g radio with an omni-directional antenna. The OLSR routing protocol was used. RTT for multiple hops was measured. SNR measurements were used to infer link quality to guide the deployment of additional relay nodes. A case study concluded that the SNR statistics that were reported by the commercially available radios were able to accurately indicate link reliability. In contrast to PER statistics, which report transmission errors that have already occurred, SNR measurements are able to provide reliable predictions of impending link failures before they occur.

In [22], a wireless networking test-bed stretched throughout a large urban area along bus routes. Each network node comprised of two Ubiquiti IEEE 802.11b/g and 900 MHz wireless interfaces with omni-directional antennas. The RTT was measured against distance, radio frequency, and number of hops.

In [23], a Wi-Fi network was created to evaluate VoIP performance. MadWifi

drivers [24] were used to operate the Atheros IEEE 802.11a chip-sets. The driver's turbo mode was used to deliver a data-rate of 108 Mbps in the 5 GHz frequency band. The OLSR routing protocol was used. Delay and bandwidth (BW) were measured for various numbers of hops and transmission protocols used, which were either UDP or TCP.

In [25], a wireless networking test-bed was used to measure handoff latency of high-speed vehicles that were roaming between stationary road-side access points (APs). Each AP node had two IEEE 802.11a radios. Software was developed to monitor and record handoff management and metrics. Handoff between APs occurred whenever the link between a mobile station (MS) and the currently associated AP exhibited an RSSI level that was lower than a pre-defined threshold.

In [26], a Wi-Fi test-bed was constructed to test TCP per-flow fairness and to improve multi-hop performance by varying IEEE 802.11e channel access MAC parameters including transmit opportunity duration, inter-frame space, and contention window size. Atheros IEEE 802.11a/b/g interfaces were employed while running MadWifi drivers. Throughput was measured under per-station and per-flow BW allocation schemes. The results were then compared.

In [27], a multi-hop wireless ad-hoc networking test-bed was created with nodes installed with 900 MHz radios to test real-time data transfer using applications including VoIP. The DSR routing protocol was used with roaming mobile and stationary nodes. A retransmission timeout algorithm that adapted to network congestion was proposed. Packet loss and inter-packet jitter was recorded.

In [28], virtualization was used to emulate and organize a MANET topology within the virtual space created across several host computers. Network node mobility was

tested virtually without any actual movement of the physical equipment. Software was developed and used to implement network protocols for both UDP and TCP traffic.

2.3 Thesis Motivation

Numerous Wi-Fi test-beds were developed over the last ten years using various chip-sets primarily for testing performance and routing. To the best of our knowledge, there have been few detailed accounts in the literature reporting experimental results for inter-channel interference. Thus, the work presented in the upcoming chapters can be helpful to achieving more efficient RF spectrum sharing, which is critical to the optimization of all future wireless networks as ever more devices contend for this limited natural resource. Chapter 3 introduces the NG-Mesh test-bed that was developed for this thesis.

Chapter 3

The NG-Mesh Test-Bed

This chapter describes the NG-Mesh test-bed design that was developed for this thesis.

3.1 The Wi-Fi Interfaces

The Wi-Fi NIC chosen for our test-bed is the Linksys WUSB54Gv4. This end-of-life (EOL) consumer-grade product has undergone four revisions and is virtually free of hardware and software deficiencies. It has a retractable 2 dBi antenna that is omni-directional. A product sheet and user manual for this Wi-Fi NIC are available from [29]. It's chip-set and software driver are discussed next.

3.1.1 The Wi-Fi Chip-set

The Linksys WUSB54Gv4 Wi-Fi NIC uses the Ralink rt2500usb (i.e., rt2570) chip-set corresponding to vendor ID *13b1* and product ID *000d*. This chip-set can operate in the IEEE 802.11b/g modes but does not support IEEE 802.11n. A partial rt2570

spec sheet is available from [30] while the corresponding driver source code from [31] indicates that the following were at least partially implemented.

- **IEEE 802.11a:** Specifies 5 GHz Wi-Fi channels at up to 54 Mbps.
- **IEEE 802.11d:** Specifies additional geographic region information.
- **IEEE 802.11e:** Specifies quality-of-service (QoS) enhancements.
- **IEEE 802.11i:** Specifies security enhancements.
- **IEEE 802.11h:** Specifies spectrum and transmit power extensions.

Radio frequencies in the 2.4 GHz spectrum are available to the rt2570 chip-set. The permitted range of bit-rates and Wi-Fi channels are configurable within the software device driver by specifying the operational geographic location and PHY mode. The chip-set used in NG-Mesh can be configured to operate in many countries around the world including North America, Europe, Israel, and Japan.

This chip-set supports the bit-rates for IEEE 802.11b (i.e., 1, 2, 5.5, and 11 Mbps) and IEEE 802.11g (i.e., 6, 9, 12, 18, 24, 36, 48, and 54 Mbps). Although the source code mentions bit-rates of 72 and 100 Mbps, they were not achievable during testing nor were they advertised in any of the Wi-Fi NIC's documentation. As part of rt2570's feature-set, IEEE 802.11 rate control algorithms are also supported. Thus algorithms such as adaptive multi-rate retry (AMRR) can be used. In AMRR, retransmissions occur at decreasing data rates before finally discarding frames that persistently fail to transmit [32].

The maximum transmission power output of this chip-set's radio is 100 mw or equivalently 20 dBm according to the driver source code. However the marketing

datasheet for the corresponding Linksys WUSB54Gv4 Wi-Fi NIC from [29] states that the transmit power typically falls between 13 dBm and 17 dBm. According to this datasheet, the receiver sensitivity is nominally in the range of -65 dBm to -80 dBm. Depending on the environment the device is operating in, a maximum communication distance of approximately 90 meters indoors and 455 meters outdoors can be achieved according to several product review websites such as [33].

For security, wired equivalent privacy (WEP) and Wi-Fi protected access version 1 (WPA) hardware encryption can be enabled. Although there are instances of WPA2 encryption indicated throughout the driver source code and the product's user manual, the marketing specs of the corresponding Linksys WUSB54Gv4 Wi-Fi NIC that houses the chip-set from [29] does not pitch its availability. All of our attempts to configure the devices for WPA2 encrypted connections have failed during testing.

3.1.2 The Wi-Fi Driver

Within the Linux community, a team of developers known as Serialmonkey maintains and enhances open-source device drivers for Ralink chip-sets. Their legacy drivers from the May 12, 2009 build were selected to operate the Wi-Fi interfaces of our test-bed. All Serialmonkey [31] driver code bases are written in the *C* programming language. The following are some key Wi-Fi NIC parameters that can be controlled from the rt2570 driver code. A detailed discussion of each will be presented later.

- Power control is possible by writing to specific chip-set registers.
- Packet header information can be manipulated during data transmission.
- Channel switching can be performed by writing to specific chip-set registers.

- Adjustment of transmit and receive frame sizes can be achieved.
- Allowable radio channel subsets can be changed by specifying geographic region and PHY mode.
- The bit-rate can be switched between the chip-set's supported speeds.

Other capabilities of the rt2570 driver includes the ability to put the Wi-Fi interfaces into monitor, managed, or ad-hoc modes, enabling and disabling request-to-send (RTS) and clear-to-send (CTS), and simultaneously sending and receiving data on multiple active interfaces and channels. With Linux networking tools such as *iwlist*, the interface can scan its vicinity for nearby networks. The driver also allows for the storage of up to four encryption keys. According to our own tests, a maximum radio retuning rate of one Wi-Fi channel switch every 30 ms is achievable by the driver without causing device operation to stall. This limitation serves to be the lower-bound on the time spent in computing scheduling decisions in routing protocols employed in our test-bed. To the best of our knowledge, there are no published results on maximum radio frequency (RF) channel switching rates when implemented from software drivers. Faster switching can possibly be realized in hardware using field-programmable gate arrays (FPGA) for example.

Since each Wi-Fi NIC can only be used as a single network interface in our test-bed, mesh nodes with multiple interfaces must have more than one Wi-Fi NIC connected to it. The driver used in this thesis lacks support for the master mode. Thus the Wi-Fi interfaces cannot operate as AP gateways. The repeater mode is also unsupported. Therefore, manual relaying of packets, which is implementable in socket or driver code must be developed in order to imitate this functionality. This driver does not

support secondary mode. If this functionality is desired, manual implementation is required for any network node redundancy to be established.

A general requisite of using the rt2570 driver is that it must be compiled from source code within a Linux environment that is operating with a kernel version of 2.4 or higher.

Compiling and Installing the Wi-Fi Driver Module

The *rt2570-cvs-daily.tar.gz* archive from [31] contains all rt2570 driver source code files and can be extracted into any arbitrary workspace directory with the following command under Linux: *tar xfpz rt2570-cvs-daily.tar.gz*

To compile the driver source code, the following sequence of commands should be executed from the *<root dir>/rt2570-cvs-2009041204/Module* directory.

1. **make clean:** Purges previously built object files from the build directory.
2. **make debug:** Builds the driver module with trace mode enabled so that debug messages can be seen.
3. **make install:** Installs the compiled driver module in the Linux OS.

In order to verify that the build was successful, the output of the *make debug* command should be similar to Figure 3.1. The output of a successful *make install* command should be similar to Figure 3.2. In order to streamline these commands and make frequent driver compilation and installation more convenient, the *rebuild.sh* shell script was created as shown in Figure 3.3. The next steps will outline the method to load the driver module into the Linux kernel.

1. **rmmod rt2*:** Removes all existing Ralink drivers that are already loaded into the Linux kernel to avoid driver conflicts.

2. **insmod rt2570.ko debug=31:** Inserts the rt2570 driver module into the Linux kernel while enabling full debug mode.

3. **lsmod | grep rt2:** Lists all Ralink driver modules that are currently loaded into the Linux kernel.

```
make[1]: Entering directory `/usr/src/kernels/2.6.18-164.2.1.el5-xen-x86_64'
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/rtusb_main.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/mlme.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/rtusb_bulk.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/connect.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/sync.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/rtusb_init.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/rtmp_tkip.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/wpa.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/rtmp_wep.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/rtusb_info.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/assoc.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/auth.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/auth_rsp.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/md5.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/rtusb_io.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/sanity.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/rtusb_data.o
  CC [M]  /root/802.11/rt2570-cvs-2009041204/Module/rt2x00debug.o
  LD [M]  /root/802.11/rt2570-cvs-2009041204/Module/rt2570.o
  Building modules, stage 2.
  MODPOST
  CC      /root/802.11/rt2570-cvs-2009041204/Module/rt2570.mod.o
  LD [M]  /root/802.11/rt2570-cvs-2009041204/Module/rt2570.ko
make[1]: Leaving directory `/usr/src/kernels/2.6.18-164.2.1.el5-xen-x86_64'
```

Figure 3.1: Successful compilation of the rt2570 driver code.

To verify that the driver is loaded into the Linux kernel properly, Figure 3.4 describes the expected output of the *lsmod* command. The first column denotes the driver module name, the second column denotes the size in bytes of memory used by

```
2.6 module install
make -C /lib/modules/2.6.18-164.2.1.el5xen/build SUBDIRS=/root/802.11/rt2570-cvs
-2009041204/Module  modules_install
make[1]: Entering directory `/usr/src/kernels/2.6.18-164.2.1.el5-xen-x86_64'
  INSTALL /root/802.11/rt2570-cvs-2009041204/Module/rt2570.ko
  DEPMOD  2.6.18-164.2.1.el5xen
make[1]: Leaving directory `/usr/src/kernels/2.6.18-164.2.1.el5-xen-x86_64'
/sbin/depmod -a
*** Update /etc/modprobe.d/ralink alias for rausb0
```

Figure 3.2: Successful installation of the rt2570 driver module into the Linux OS.

```
make clean
make debug
make install
```

Figure 3.3: Contents of rebuild.sh.

the module, and the third column represents the number of devices that is currently using the module.

```
rt2570                282496  0
```

Figure 3.4: Successfully loaded but unused driver module.

In order to view the diagnostic messages that are generated by the rt2570 driver during interface operation, the *dmesg* shell command can be used. All trace statements relating to the driver typically start with *rt2570* in the standard output. Figure 3.5 shows an example of the device driver reporting the transmit power and average RSSI level upon Wi-Fi interface initialization.

Once the newly compiled Ralink driver module has been loaded into the Linux kernel successfully, configuring and powering on the network interfaces are the final steps required to activate them.

```
rt2570: <-- AsicAdjustTxPower: TxPwr = 15, AvgRssi = -121
rt2570: <-- AsicAdjustTxPower: TxPwr = 15, AvgRssi = -121
rt2570: <-- AsicAdjustTxPower: TxPwr = 15, AvgRssi = -121
rt2570: <-- AsicAdjustTxPower: TxPwr = 15, AvgRssi = -121
rt2570: <-- AsicAdjustTxPower: TxPwr = 15, AvgRssi = -121
rt2570: <-- AsicAdjustTxPower: TxPwr = 15, AvgRssi = -121
```

Figure 3.5: Diagnostic messages in Linux.

Key Driver Files and Functions

This subsection points out the specific locations within the driver source code that are relevant to this thesis.

Before any changes to the driver source code can take effect, recompilation and reinstallation of the driver module are required.

The Wi-Fi drivers used in our test-bed rely on a periodically executed 500 line function called *PeriodicExec* residing in the driver source code file *mlme.c* near line 910. The key duties of this function are listed as follows.

- A call to the *AsicAdjustTxPower* function is made to update the radio's transmission power output.
- The chip-set's internal counters keep track of the RSSI, packet retry rate (PRR), PER, and channel quality indicator (CQI) metrics.
- Conditional statements are processed for automatically changing the data-rate based on PRR and PER metrics. The bit-rate is decreased if the PRR passes a threshold. If not enough data is available to compute the PRR, then the bit-rate is decreased if any PER is experienced. The bit-rate is increased if there is no PER and the PRR is less than 10%.

Channel quality is calculated at lines 1120 to 1122 as shown in Figure 3.6. While the maximum CQI value is ultimately capped at 100 (i.e., lines 1123 and 1124), this metric is captured in its raw form in this thesis as computed from the Ralink equation, which can be greater than 100, in order to capture fluctuations.

```
1118            // ChannelQuality = W1*RSSI + W2*TxPRR + W3*RxPER
1119                //   (RSSI 0..100), (TxPER 100..0), (RxPER 100..0)
1120            pAd->Mlme.ChannelQuality = (RSSI_WEIGHTING * pAd->PortCfg.LastRssi +
1121                                        TX_WEIGHTING * (100 - TxPRR) +
1122                                        RX_WEIGHTING* (100 - RxPER)) / 100;
1123            if (pAd->Mlme.ChannelQuality >= 100)
1124                pAd->Mlme.ChannelQuality = 100;
1125        }
1126
1127    DBGPRINT(RT_DEBUG_INFO, "MMCHK - CQI= %d,
1128                (Tx Fail=%d/Retry=%d/Total=%d, Rx Fail=%d/Total=%d, RSSI=%d dbm)\n",
1129        pAd->Mlme.ChannelQuality, TxFailCount, TxRetryOkCount, TxTotalCnt, RxFailCnt, RxCnt,
1130                pAd->PortCfg.LastRssi - pAd->BBPTuningParameters.RSSIToDbmOffset);
```

Figure 3.6: CQI calculation in the driver code.

The meanings of the variables that are used in the *CQI* equation is as follows.

- **RSSI_WEIGHTING:** Has a value of 40% and is defined as a constant in *mlme.h* near line 89.
- **RX_WEIGHTING:** Has a value of 20% and is defined as a constant in *mlme.h* near line 91.
- **TX_WEIGHTING:** Has a value of 40% and is defined as a constant in *mlme.h* near line 90.
- **pAd->PortCfg.LastRssi:** The most recent RSSI value as detected by the interface. An offset of 121 is used to convert this value to be in units of dBm.
- **TxPRR:** The packet retransmission rate of the transmitter. It is computed near line 1015 in *mlme.c* within the *PeriodicExec* function as shown in Figure 3.7.

- **RxPER:** The packet error rate experienced by the receiver. It is computed near line 1035 in *mlme.c* within the *PeriodicExec* function as shown in Figure 3.7.

```
1003    if (TxTotalCnt < 5) // if too few TX samples, skip TX related statistics
1004    {
1005        TxPER = 0;  // don't take TxPER into CQI consideration if too few sample
1006        TxPRR = 0;
1007    }
1008    else
1009    {
1010                if (TxFailCount < TxTotalCnt)
1011                {
1012                        USHORT temp = TxOneRetryCnt + TxMRetryCnt + TxFailCount;
1013                        TxPER = (TxFailCount * 100) / TxTotalCnt;
1014                        if (temp < TxTotalCnt)
1015                                TxPRR = (temp * 100) / TxTotalCnt;
1016                        else
1017                                TxPRR = 100;
1018                }
1019                else
1020                        TxPER = 100;
1021    }
1022
1023    //
1024    // calculate RX PER
1025    //
1026        RxFailCnt = pAd->MACCounters[0];
1027    RxOkCnt   = pAd->WlanCounters.ReceivedFragmentCount.vv.LowPart -
1028                pAd->Mlme.PrevWlanCounters.ReceivedFragmentCount.vv.LowPart;
1029    RxCnt = RxOkCnt + RxFailCnt;
1030
1031
1032    if (RxCnt < 5)
1033        RxPER = 0;  // don't take RxPER into ChannelQuality consideration if too few sample
1034    else
1035        RxPER = (RxFailCnt * 100) / RxCnt;
1036        DBGPRINT(RT DEBUG TRACE, "RxPER: %d\n", RxPER);
```

Figure 3.7: TxPRR and RxPER calculations in the driver code.

As shown in Figure 3.7, if the transmitted or received packet count is less than five, then *TxPRR* and *RxPER* values are not factored into the *CQI* computation. The *TxOneRetryCnt*, *TxMRetryCnt*, *TxFailCnt*, and *RxFailCnt* variables representing packet retransmission and fail count values are read in from internal chip-set counters in the fashion shown at line 1026.

The *PeriodicExec* function is actually called by the 200 line *MlmePeriodicExec* function in the same source code file. It is situated near line 547 and manages the

connection between the rt2570 interface and the AP if it is configured in infrastructure mode. In ad-hoc (i.e., IBSS) mode, this function manages the links between the Wi-Fi interface and all of its peers.

The remainder of this subsection discusses various driver code modifications to control key aspects of the Wi-Fi interfaces.

Power Control Transmission power control is implemented in the *AsicAdjustTxPower* function of the driver source code file *mlme.c* as shown in Figure 3.8 to Figure 3.11.

As shown in the source code comments of Figure 3.8, the transmit power of the radio is initially read from the electrically erasable programmable read-only memory (EEPROM) of the chip-set. After initialization, it is automatically adjusted depending on the distance between the AP and the Wi-Fi interface. This perceived distance is determined by the RSSI metric.

The *R3* variable is measured in units of dBm and represents the transmission power emitted from the Ralink radio. Allowable power settings must fall within the range of 1 mw to 100 mw (i.e., 0 dBm to 20 dBm). As shown at line 3631 in Figure 3.11, *R3 = 15* is inserted into the driver code in order to set the transmission power to 15 dBm as an example of how to specify the transmission power manually. In this way, the periodic auto-adjustment of the power levels in the transmitter is nullified. The conditional statement at line 3632 numerically compares the value of the variables *CurrTxPwr* and *R3*. If and only if they are not equal then lines 3633 to 3641 will be activated and will set the value of *CurrTxPwr* to the value of *R3*. The value of *R3* in this example was chosen arbitrarily. Code lines 3633 to 3641 sets the transmit power level of the Ralink radio by calling the function *RTUSBWriteRFRegister* to

initiate the writing of the new power setting to the appropriate chip-set register via the USB controller to which the Wi-Fi interface is connected. Here, *pAd* is a pointer directed at the structures identifying the particular USB Wi-Fi NIC of interest. Additionally, the value of *R3* is converted into a form that the chip-set can understand and a diagnostic message is generated that is accessible in debug mode to indicate the status of the transmit power setting change operation.

```
3505 VOID AsicAdjustTxPower(
3506         IN PRT2570ADAPTER pAd)
3507 {
3508         ULONG R3, Channel, CurrTxPwr;
3509
3510         if ((pAd->PortCfg.Channel >= 1) && (pAd->PortCfg.Channel <= 14))
3511                 Channel = pAd->PortCfg.Channel;
3512         else
3513                 Channel = 1;
3514                  // don't have calibration info for 11A, temporarily use Channel 1
3515
3516         // get TX Power base from E2PROM
3517         R3 = pAd->PortCfg.ChannelTxPower[Channel - 1];
3518         if (R3 > 31)  R3 = 31;
3519
3520         // E2PROM setting is calibrated for maximum TX power (i.e. 100%)
3521         // We lower TX power here according to the percentage specified from UI
3522         if (pAd->PortCfg.TxPowerPercentage == 0xffffffff)       // AUTO TX POWER control
3523         {
3524                 // only INFRASTRUCTURE mode and 100% TX power need furthur calibration
3525                 if (pAd->MediaState == NdisMediaStateConnected)
3526                 {
3527                         // low TX power upon very-short distance to AP to solve some
3528                         // vendor's AP RX problem in this case, no TSSI compensation is required.
3529                         if ((pAd->DrsCounters.fNoisyEnvironment == FALSE) &&
3530                             (pAd->PortCfg.AvgRssi > (pAd->BBPTuningParameters.RSSIToDbmOffset
3531                                 - RSSI_FOR_LOWEST_TX_POWER)))
3532                                 R3 -= LOWEST_TX_POWER_DELTA;
3533                         else if ((pAd->DrsCounters.fNoisyEnvironment == FALSE) &&
3534                             (pAd->PortCfg.AvgRssi > (pAd->BBPTuningParameters.RSSIToDbmOffset
3535                                 - RSSI_FOR_LOW_TX_POWER)))
3536                                 R3 -= LOW_TX_POWER_DELTA;
3537                         // 2004-03-16 give OFDM rates lower than 48 mbps 2 more DB
3538                         else if ((pAd->PortCfg.TxRate <= RATE_36) && (pAd->PortCfg.TxRate > RATE_11))
3539                         {
3540                             R3 +=2;
3541                             if (R3 > 31) R3 = 31;
3542                         }
3543
3544
3545                         // 2 exclusive rules applied on CCK rates only -
3546                         //        1. always plus 2 db for CCK
3547                         //        2. adjust TX Power based on TSSI
```

Figure 3.8: The AsicAdjustTxPower function (1 of 4).

```
3548                        else
3549                        if (pAd->PortCfg.TxRate <= RATE_11)
3550                        {
3551                                // if "auto calibration based on TSSI" is not required, then
3552                                // always give CCK 2 more db
3553                                if (pAd->PortCfg.bAutoTxAgc == FALSE)
3554                                {
3555                                        // do not give RATE_11 2 more db to avoid the case that
3556                                        // the outgoing NULL frame (for rate upgrade evaluation)
3557                                        // using 12 Mbps OFDM rate is over-powered
3558                                        if (pAd->PortCfg.TxRate <= RATE_5_5)
3559                                        {
3560                                                R3 += 2;  // plus 2 db
3561                                                if (R3 > 31) R3 = 31;
3562                                        }
3563                                }
3564
3565                                // Auto calibrate Tx AGC if bAutoTxAgc is TRUE and TX rate is CCK,
3566                                // because E2PROM's TSSI reference is valid only in CCK range.
3567                                else
3568                                {
3569                                        UCHAR   R1,TxPowerRef, TssiRef;
3570
3571                                        R3 = (pAd->PortCfg.LatchRfRegs.R3 >> 9) & 0x0000001f;
3572                                        if (pAd->Mlme.PeriodicRound % 4 == 0) // every 4 second
3573                                        {
3574                                        TxPowerRef = pAd->PortCfg.ChannelTxPower[Channel - 1];
3575                                        TssiRef    = pAd->PortCfg.ChannelTssiRef[Channel - 1];
3576                                        RTUSBReadBBPRegister(pAd, BBP_Tx_Tssi,  &R1);
3577                                        if ((TssiRef >= (R1 + pAd->PortCfg.ChannelTssiDelta)) ||
3578                                                (TssiRef <= (R1 - pAd->PortCfg.ChannelTssiDelta)))
3579                                        {
3580                                                // Need R3 adjustment. However, we have to make sure
3581                                                // there is only plus / minus 5 variation allowed
3582                                                if (TssiRef > R1)
3583                                                {
3584                                                        R3 = (R3 < (ULONG) (TxPowerRef + 5))
3585                                                        ? (R3 + 1) : R3;
3586                                                        if (R3 > 31)
3587                                                                R3 = 31;
3588                                                                DBGPRINT(RT_DEBUG_TRACE,"TSSI(R1)=%d,
3589                                                                ++TxPwr=%d\n", R1, R3);
3590                                                        }
```

Figure 3.9: The AsicAdjustTxPower function (2 of 4).

```
3591                                                    else
3592                                                    {
3593                                                            R3 = (R3 > (ULONG) (TxPowerRef - 5))
3594                                                            ? (R3 - 1) : R3;
3595                                                            DBGPRINT(RT_DEBUG_TRACE,"TSSI(R1)=%d,
3596                                                            --TxPwr=%d\n", R1, R3);
3597                                                    }
3598                                                }
3599                                            }
3600                                        }
3601                                    }
3602                            }
3603                    }
3604                    else // fixed AUTO TX power
3605                    {
3606                            if (pAd->PortCfg.TxPowerPercentage > 90)  // 91 ~ 100%, treat as 100% in terms of mW
3607                                ;
3608                            else if (pAd->PortCfg.TxPowerPercentage > 60)  // 61 ~ 90%, treat as 75% in terms of mW
3609                                R3 -= 1;
3610                            else if (pAd->PortCfg.TxPowerPercentage > 30)  // 31 ~ 60%, treat as 50% in terms of mW
3611                                R3 -= 3;
3612                            else if (pAd->PortCfg.TxPowerPercentage > 15)  // 16 ~ 30%, treat as 25% in terms of mW
3613                                R3 -= 6;
3614                            else if (pAd->PortCfg.TxPowerPercentage > 9)   // 10 ~ 15%, treat as 12.5% in terms of mW
3615                                R3 -= 9;
3616                            else                                           // 0 ~ 9 %, treat as MIN(~3%) in terms of mW
3617                                R3 -= 12;
3618                            if (R3 > 31)  R3 = 0;   // negative value, set as minimum 0
3619
3620                            // 2004-03-16 give TX rates <= 36 mbps 2 more DB
3621                            if (pAd->PortCfg.TxRate <= RATE_36)
3622                            {
3623                                R3 +=2;
3624                                if (R3 > 31) R3 = 31;
3625                            }
3626                    }
```

Figure 3.10: The AsicAdjustTxPower function (3 of 4).

```
3627
3628            // compare the desired R3.TxPwr value with current R3, if not equal
3629            // set new R3.TxPwr
3630            CurrTxPwr = (pAd->PortCfg.LatchRfRegs.R3 >> 9) & 0x0000001f;
3631            R3=15;
3632            if (CurrTxPwr != R3)
3633            {
3634                    CurrTxPwr = R3;
3635                    R3 = (pAd->PortCfg.LatchRfRegs.R3 & 0xfffffc1ff) | (R3 << 9);
3636                    DBGPRINT(RT_DEBUG_INFO, "--- %s: R3=0x%08x hi=0x%04x lo=0x%04x\n",
3637                                            __FUNCTION__, R3,
3638                                            (USHORT)(R3 >> 16), (USHORT)(R3 & 0x0000ffff));
3639                    RTUSBWriteRFRegister(pAd, R3);
3640                    pAd->PortCfg.LatchRfRegs.R3 = R3;
3641            }
3642        DBGPRINT(RT_DEBUG_TRACE, "<-- %s: TxPwr = %d, AvgRssi = %d\n",
3643                                __FUNCTION__, CurrTxPwr,
3644                                pAd->PortCfg.AvgRssi - pAd->BBPTuningParameters.RSSIToDbmOffset);
3645
3646 }
```

Figure 3.11: The AsicAdjustTxPower function (4 of 4).

Packet Header Manipulation Initiation of hardware encryption and transmission as well as all packet preprocessing is handled in the *rtusb_data.c* source code file. This includes the source and destination addresses defined in the variables *pHeader->Controlhead.Addr2* and *Header_802_11.Controlhead.Addr1* respectively. Examples of each address type can be seen near lines 2001 and 923 correspondingly in the code file. Modification of the MAC addresses contained in these variables is possible through the manipulation of their values. As their containing function *RTUSBHardEncrypt* exceeds 900 lines of code, only the relevant code portion is presented in Figure 3.12 to show how destination addresses are stored at the transmitting node.

```
907         memset(&Header_802_11, 0, sizeof(HEADER_802_11));
908         //
909         // Start making 802.11 frame header
910         //
911         if (INFRA_ON(pAdapter))
912         {
913                 // In BSS mode, AP's address(BSSID) is the destination address of all outgoing packets
914                 // Address 1 - BSSID
915                 memcpy(&Header_802_11.Controlhead.Addr1, &pAdapter->PortCfg.Bssid, ETH_LENGTH_OF_ADDRESS);
916                 // Address 3 - DA
917                 memcpy(&Header_802_11.Addr3, (PUCHAR) pVirtualAddress, ETH_LENGTH_OF_ADDRESS);
918                 Header_802_11.Controlhead.Frame.ToDs = 1;
919         }
920         else
921         {
922                 // Address 1 - DA
923                 memcpy(&Header_802_11.Controlhead.Addr1, (PUCHAR) pVirtualAddress, ETH_LENGTH_OF_ADDRESS);
924                 // Address 3 - BSSID
925                 memcpy(&Header_802_11.Addr3, &pAdapter->PortCfg.Bssid, ETH_LENGTH_OF_ADDRESS);
926         }
927         // Address 2 - SA in both infrastructure & ad-hoc modes
928         memcpy(&Header_802_11.Controlhead.Addr2, pAdapter->CurrentAddress, ETH_LENGTH_OF_ADDRESS);
```

Figure 3.12: Header address processing in the RTUSBHardEncrypt function.

An example of packet header manipulation can be demonstrated by inserting the following code above line 923 in Figure 3.12 to change the destination address in the IEEE 802.11 header to *00:00:00:00:00:00* in an ad-hoc packet being transmitted.

for (i=0; i<6; i++){pVirtualAddress[i]="00";}

Similarly, the source address can also be changed in the same way. Since the containing function *RTUSBRxPacket* is more than 900 lines in length, only the relevant

code portion is presented in Figure 3.13 to show how source addresses are stored at the receiving node.

```
1988        DBGPRINT(RT_DEBUG_INFO,"-  %s: Receiving unicast frame\n", __FUNCTION__);
1989        //
1990        // Begin frame processing
1991        //
1992        // DA is always address 1
1993        pDestMac = (PUCHAR)     &(pHeader->Controlhead.Addr1);
1994        // Seclect SA by different mode
1995        if (INFRA_ON(pAdapter))         // For infrastructure, SA is address 3
1996        {
1997                pSrcMac = (PUCHAR) &(pHeader->Addr3);
1998        }
1999        else                            // For IBSS     mode, SA is     address 2
2000        {
2001                pSrcMac = (PUCHAR) &(pHeader->Controlhead.Addr2);
2002        }
```

Figure 3.13: Header address processing in the RTUSBRxPacket function.

Retuning the Transmitter The *AsicSwitchChannel* function near line 3259 of the *mlme.c* source code file initiates the switching of the Ralink radio between Wi-Fi communication channels. Changing to Wi-Fi channel 11 for example can easily be done by calling *AsicSwitchChannel(pAd, 11)* from the driver code. This function adjusts the transmitter to a suitable power level upon switching to a new Wi-Fi channel as shown in Figure 3.14. Figure 3.15 continues to list the available RF integrated circuit (IC) configurations. In our test-bed, the rt2570 chip-set uses the RT2525E transceiver chip (i.e., the radio) corresponding to line 3375 in Figure 3.16, which is confirmed with the trace output generated by line 3421 of Figure 3.17. The test-bed interfaces therefore respond to modifications to this portion of code including any changes to the transmit power by adjusting *R3*'s value. This part of the code also tunes the transmitter to the desired communication frequency by making function

calls to perform the appropriate transmitter configurations. All transmitter configurations are performed through function calls within the *rtusb_io.c* source code file. The first function called is *RTUSBWriteRFRegister*. This function, in turn, executes calls to the function *RTUSBWriteMACRegister*. The chain of function calls continues to *RTUSB_VendorRequest* and then finally to *usb_control_msg* where the final instructions are packaged and sent to the physical USB connected Wi-Fi NIC. It is here that register writing operations are performed to configure the transmitter. With each subsequent function call, the values of the *RF2525eRegTable[index]* variables are further parsed, processed, and readied for low-level transport to the Ralink chip-set.

```
3259 VOID AsicSwitchChannel(
3260         IN PRT2570ADAPTER pAd,
3261         IN UCHAR Channel)
3262 {
3263         ULONG R3;
3264         UCHAR index;
3265
3266         // TODO: need to update E2PROM format to add 802.11a channel's TX power calibration values
3267         if (Channel <= 14)
3268         R3 = pAd->PortCfg.ChannelTxPower[Channel - 1];
3269         else
3270                 R3 = pAd->PortCfg.ChannelTxPower[0];
3271
3272         if (R3 > 31)  R3 = 31;
3273
3274         // E2PROM setting is calibrated for maximum TX power (i.e. 100%)
3275         // We lower TX power here according to the percentage specified from UI
3276         if (pAd->PortCfg.TxPowerPercentage > 90)           // 91 ~ 100%, treat as 100% in terms of mW
3277                 ;
3278         else if (pAd->PortCfg.TxPowerPercentage > 60)  // 61 ~ 90%, treat as 75% in terms of mW
3279                 R3 -= 1;
3280         else if (pAd->PortCfg.TxPowerPercentage > 30)  // 31 ~ 60%, treat as 50% in terms of mW
3281                 R3 -= 3;
3282         else if (pAd->PortCfg.TxPowerPercentage > 15)  // 16 ~ 30%, treat as 25% in terms of mW
3283                 R3 -= 6;
3284         else if (pAd->PortCfg.TxPowerPercentage > 9)   // 10 ~ 15%, treat as 12.5% in terms of mW
3285                 R3 -= 9;
3286         else                                          // 0 ~ 9 %, treat as 6.25% in terms of mW
3287                 R3 -= 12;
3288
3289         R3 = R3 << 9; // shift TX power control to correct RF R3 bit position
3290         switch (pAd->PortCfg.RfType)
3291         {
3292                 case RFIC_2522:
3293                         for (index = 0; index < NUM_OF_2522_CHNL; index++)
3294                         {
3295                                 if (Channel == RF2522RegTable[index].Channel)
3296                                 {
3297                                         R3 = R3 | RF2522RegTable[index].R3; // set TX power
```

Figure 3.14: The AsicSwitchChannel function (1 of 4).

```
3298                                    RTUSBWriteRFRegister(pAd, RF2522RegTable[index].R1);
3299                                    RTUSBWriteRFRegister(pAd, RF2522RegTable[index].R2);
3300                                    RTUSBWriteRFRegister(pAd, R3);
3301                                    pAd->PortCfg.LatchRfRegs.Channel = Channel;
3302                                    pAd->PortCfg.LatchRfRegs.R1 = RF2522RegTable[index].R1;
3303                                    pAd->PortCfg.LatchRfRegs.R2 = RF2522RegTable[index].R2;
3304                                    pAd->PortCfg.LatchRfRegs.R3 = R3;
3305                                    pAd->PortCfg.LatchRfRegs.R4 = RF2522RegTable[index].R4;
3306                                    break;
3307                            }
3308                    }
3309                    break;
3310
3311            case RFIC_2523:
3312                    for (index = 0; index < NUM_OF_2523_CHNL; index++)
3313                    {
3314                            if (Channel == RF2523RegTable[index].Channel)
3315                            {
3316                                    R3 = R3 | RF2523RegTable[index].R3; // set TX power
3317                                    RTUSBWriteRFRegister(pAd, RF2523RegTable[index].R1);
3318                                    RTUSBWriteRFRegister(pAd, RF2523RegTable[index].R2);
3319                                    RTUSBWriteRFRegister(pAd, R3);
3320                                    RTUSBWriteRFRegister(pAd, RF2523RegTable[index].R4);
3321                                    pAd->PortCfg.LatchRfRegs.Channel = Channel;
3322                                    pAd->PortCfg.LatchRfRegs.R1 = RF2523RegTable[index].R1;
3323                                    pAd->PortCfg.LatchRfRegs.R2 = RF2523RegTable[index].R2;
3324                                    pAd->PortCfg.LatchRfRegs.R3 = R3;
3325                                    pAd->PortCfg.LatchRfRegs.R4 = RF2523RegTable[index].R4;
3326                                    break;
3327                            }
3328                    }
3329                    break;
3330
3331            case RFIC_2524:
3332                    for (index = 0; index < NUM_OF_2524_CHNL; index++)
3333                    {
3334                            if (Channel == RF2524RegTable[index].Channel)
3335                            {
3336                                    R3 = R3 | RF2524RegTable[index].R3; // set TX power
3337                                    RTUSBWriteRFRegister(pAd, RF2524RegTable[index].R1);
3338                                    RTUSBWriteRFRegister(pAd, RF2524RegTable[index].R2);
3339                                    RTUSBWriteRFRegister(pAd, R3);
3340                                    RTUSBWriteRFRegister(pAd, RF2524RegTable[index].R4);
3341                                    pAd->PortCfg.LatchRfRegs.Channel = Channel;
3342                                    pAd->PortCfg.LatchRfRegs.R1 = RF2524RegTable[index].R1;
3343                                    pAd->PortCfg.LatchRfRegs.R2 = RF2524RegTable[index].R2;
3344                                    pAd->PortCfg.LatchRfRegs.R3 = R3;
3345                                    pAd->PortCfg.LatchRfRegs.R4 = RF2524RegTable[index].R4;
3346                                    break;
3347                            }
3348                    }
```

Figure 3.15: The AsicSwitchChannel function (2 of 4).

```
3349                        break;
3350
3351                case RFIC_2525:
3352                        for (index = 0; index < NUM_OF_2525_CHNL; index++)
3353                        {
3354                                if (Channel == RF2525RegTable[index].Channel)
3355                                {
3356                                        R3 = R3 | RF2525RegTable[index].R3; // set TX power
3357                                        RTUSBWriteRFRegister(pAd, RF2525RegTable[index].R1);
3358
3359                                        RTUSBWriteRFRegister(pAd, RF2525RegTable[index].R2);
3360
3361                                        RTUSBWriteRFRegister(pAd, R3);
3362
3363                                        RTUSBWriteRFRegister(pAd, RF2525RegTable[index].R4);
3364
3365                                        pAd->PortCfg.LatchRfRegs.Channel = Channel;
3366                                        pAd->PortCfg.LatchRfRegs.R1 = RF2525RegTable[index].R1;
3367                                        pAd->PortCfg.LatchRfRegs.R2 = RF2525RegTable[index].R2;
3368                                        pAd->PortCfg.LatchRfRegs.R3 = R3;
3369                                        pAd->PortCfg.LatchRfRegs.R4 = RF2525RegTable[index].R4;
3370                                        break;
3371                                }
3372                        }
3373                        break;
3374
3375                case RFIC_2525E:
3376                        for (index = 0; index < NUM_OF_2525E_CHNL; index++)
3377                        {
3378                                if (Channel == RF2525eRegTable[index].Channel)
3379                                {
3380                                        RTUSBWriteRFRegister(pAd, RF2525eRegTable[index].TempR2);
3381                                        RTUSBWriteRFRegister(pAd, RF2525eRegTable[index].R4);
3382                                        R3 = R3 | RF2525eRegTable[index].R3; // set TX power
3383                                        RTUSBWriteRFRegister(pAd, RF2525eRegTable[index].R1);
3384                                        RTUSBWriteRFRegister(pAd, RF2525eRegTable[index].R2);
3385                                        RTUSBWriteRFRegister(pAd, R3);
3386                                        RTUSBWriteRFRegister(pAd, RF2525eRegTable[index].R4);
3387                                        pAd->PortCfg.LatchRfRegs.Channel = Channel;
3388                                        pAd->PortCfg.LatchRfRegs.R1 = RF2525eRegTable[index].R1;
3389                                        pAd->PortCfg.LatchRfRegs.R2 = RF2525eRegTable[index].R2;
3390                                        pAd->PortCfg.LatchRfRegs.R3 = R3;
3391                                        pAd->PortCfg.LatchRfRegs.R4 = RF2525eRegTable[index].R4;
3392                                        break;
3393                                }
```

Figure 3.16: The AsicSwitchChannel function (3 of 4).

```
3394                        }
3395                        break;
3396
3397                case RFIC_5222:
3398                        for (index = 0; index < NUM_OF_5222_CHNL; index++)
3399                        {
3400                                if (Channel == RF5222RegTable[index].Channel)
3401                                {
3402                                        R3 = R3 | RF5222RegTable[index].R3; // set TX power
3403                                        RTUSBWriteRFRegister(pAd, RF5222RegTable[index].R1);
3404                                        RTUSBWriteRFRegister(pAd, RF5222RegTable[index].R2);
3405                                        RTUSBWriteRFRegister(pAd, R3);
3406                                        RTUSBWriteRFRegister(pAd, RF5222RegTable[index].R4);
3407                                        pAd->PortCfg.LatchRfRegs.Channel = Channel;
3408                                        pAd->PortCfg.LatchRfRegs.R1 = RF5222RegTable[index].R1;
3409                                        pAd->PortCfg.LatchRfRegs.R2 = RF5222RegTable[index].R2;
3410                                        pAd->PortCfg.LatchRfRegs.R3 = R3;
3411                                        pAd->PortCfg.LatchRfRegs.R4 = RF5222RegTable[index].R4;
3412                                        break;
3413                                }
3414                        }
3415                        break;
3416
3417                default:
3418                        break;
3419        }
3420
3421        DBGPRINT(RT_DEBUG_TRACE, "--- AsicSwitchChannel(RF=%d) to #%d,"
3422                        " TXPwr pct = %d = %dmW,"
3423                        " R1=0x%08x, R2=0x%08x, R3=0x%08x, R4=0x%08x\n",
3424                pAd->PortCfg.RfType,
3425                pAd->PortCfg.LatchRfRegs.Channel,
3426                pAd->PortCfg.TxPowerPercentage,
3427                pAd->PortCfg.ChannelTxPower[Channel - 1],
3428                pAd->PortCfg.LatchRfRegs.R1,
3429                pAd->PortCfg.LatchRfRegs.R2,
3430                pAd->PortCfg.LatchRfRegs.R3,
3431                pAd->PortCfg.LatchRfRegs.R4);
3432
3433 }
```

Figure 3.17: The AsicSwitchChannel function (4 of 4).

At line 146 of the *mlme.c* source code file, a set of arrays of type *RTMP_RF_REGS* define chip-set register values that correspond to each Wi-Fi channel for the 2525E transceiver. As described earlier, *AsicSwitchChannel* makes calls to a chain of other functions in order to write these values into the appropriate hardware registers through a USB controller. This map of register values define the Ralink radio's range of permitted transmit power and center frequency values and is shown in Figure 3.18.

```
146 RTMP_RF_REGS_1 RF2525eRegTable[] = {
147 // using 10 Mhz reference clock
148 //      ch   TempR2              R1          R2          R3(TX0~4=0) R4
149         {1,  0x940008aa, 0x94022010, 0x9400089a, 0x94060111, 0x94000e1b},
150         {2,  0x940008ae, 0x94022010, 0x9400089e, 0x94060111, 0x94000e07},
151         {3,  0x940008ae, 0x94022010, 0x9400089e, 0x94060111, 0x94000e1b},
152         {4,  0x940008b2, 0x94022010, 0x940008a2, 0x94060111, 0x94000e07},
153         {5,  0x940008b2, 0x94022010, 0x940008a2, 0x94060111, 0x94000e1b},
154         {6,  0x940008b6, 0x94022010, 0x940008a6, 0x94060111, 0x94000e07},
155         {7,  0x940008b6, 0x94022010, 0x940008a6, 0x94060111, 0x94000e1b},
156         {8,  0x940008ba, 0x94022010, 0x940008aa, 0x94060111, 0x94000e07},
157         {9,  0x940008ba, 0x94022010, 0x940008aa, 0x94060111, 0x94000e1b},
158         {10, 0x940008be, 0x94022010, 0x940008ae, 0x94060111, 0x94000e07},
159         {11, 0x940008b7, 0x94022010, 0x940008ae, 0x94060111, 0x94000e1b},
160         {12, 0x94000902, 0x94022010, 0x940008b2, 0x94060111, 0x94000e07},
161         {13, 0x94000902, 0x94022010, 0x940008b2, 0x94060111, 0x94000e1b},
162         {14, 0x94000906, 0x94022010, 0x940008b6, 0x94060111, 0x94000e23}
163 };
```

Figure 3.18: Transmitter tuning rt2570 register values.

The PHY mode is also considered by the driver when defining the allowable communication channel set. For example, IEEE 802.11a can support 5 GHz Wi-Fi channels whereas IEEE 802.11b/g only supports the ones in the 2.4 GHz band. These channel set definitions are hard-coded at the start of the *sync.c* source code file as shown in Figure 3.19, which correspond to the frequencies defined near the start of the *rtusb_info.c* source code file as shown in Figure 3.20. When no Wi-Fi channel is specified, the transmitter defaults to 2.412 GHz which corresponds to channel 1.

The geographic region in which the Wi-Fi interface is operating also defines the allowable register values to be written. In North America, for example, only W-Fi channels 1 to 11 are used in accordance to the regulations set out by the Federal Communications Commission (FCC) in the USA and Industry Canada (IC) in Canada. These channels correspond to the radio frequencies between 2.412 GHz and 2.462 GHz. The code implementing this control can be found in the driver source code file

sync.c around line 1484 within the *BuildChannelList* function as shown in Figure 3.21.

The comments in Figure 3.22 shows that support for 5 GHz Wi-Fi channels has not been finalized in this version of the rt2570 driver. As a result of the unfinished channel map definition near line 1532, the radio can always be configured to operate at the frequencies between 5.180 GHz to 5.805 GHz regardless of the geographic region as long as the device can be configured into the IEEE 802.11a PHY mode.

```
37 // 2.4 Ghz channel plan
38 UCHAR Ra24Ghz_FCC[] = {1,2,3,4,5,6,7,8,9,10,11};
39 UCHAR Ra24Ghz_IC[]  = {1,2,3,4,5,6,7,8,9,10,11};
40 UCHAR Ra24Ghz_ESTI[]= {1,2,3,4,5,6,7,8,9,10,11,12,13};
41 UCHAR Ra24Ghz_SPAIN[] = {10,11};
42 UCHAR Ra24Ghz_FRANCE[] = {10,11,12,13};
43 UCHAR Ra24Ghz_MKK[] =  {14};
44 UCHAR Ra24Ghz_MKK1[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14};
45 UCHAR Ra24Ghz_ISRAEL[]  = {3,4,5,6,7,8,9};
46
47 // 5 Ghz channel plan
48 UCHAR Ra5Ghz_UNII[] = {36,40,44,48,52,56,60,64,  149,153,157,161};
49 UCHAR Ra5Ghz_MMAC[] = {34,38,42,46};
50 UCHAR Ra5Ghz_HyperLAN2[] = {36,40,44,48,52,56,60,64,  100,104,108,112,116,120,124,128,132,136,140};
```

Figure 3.19: Channel sets based on governing bodies and PHY modes.

```
61 #define     MAP_CHANNEL_ID_TO_KHZ(ch, khz)  {                \
62                 switch (ch)                                  \
63                 {                                            \
64                     case 1:     khz = 2412000;   break;      \
65                     case 2:     khz = 2417000;   break;      \
66                     case 3:     khz = 2422000;   break;      \
67                     case 4:     khz = 2427000;   break;      \
68                     case 5:     khz = 2432000;   break;      \
69                     case 6:     khz = 2437000;   break;      \
70                     case 7:     khz = 2442000;   break;      \
71                     case 8:     khz = 2447000;   break;      \
72                     case 9:     khz = 2452000;   break;      \
73                     case 10:    khz = 2457000;   break;      \
74                     case 11:    khz = 2462000;   break;      \
75                     case 12:    khz = 2467000;   break;      \
76                     case 13:    khz = 2472000;   break;      \
77                     case 14:    khz = 2484000;   break;      \
78                     case 36:  /* UNII */  khz = 5180000;   break;     \
79                     case 40:  /* UNII */  khz = 5200000;   break;     \
80                     case 44:  /* UNII */  khz = 5220000;   break;     \
81                     case 48:  /* UNII */  khz = 5240000;   break;     \
82                     case 52:  /* UNII */  khz = 5260000;   break;     \
83                     case 56:  /* UNII */  khz = 5280000;   break;     \
84                     case 60:  /* UNII */  khz = 5300000;   break;     \
85                     case 64:  /* UNII */  khz = 5320000;   break;     \
86                     case 149: /* UNII */  khz = 5745000;   break;     \
87                     case 153: /* UNII */  khz = 5765000;   break;     \
88                     case 157: /* UNII */  khz = 5785000;   break;     \
89                     case 161: /* UNII */  khz = 5805000;   break;     \
90                     case 100: /* HiperLAN2 */  khz = 5500000;   break;     \
91                     case 104: /* HiperLAN2 */  khz = 5520000;   break;     \
92                     case 108: /* HiperLAN2 */  khz = 5540000;   break;     \
93                     case 112: /* HiperLAN2 */  khz = 5560000;   break;     \
94                     case 116: /* HiperLAN2 */  khz = 5580000;   break;     \
95                     case 120: /* HiperLAN2 */  khz = 5600000;   break;     \
96                     case 124: /* HiperLAN2 */  khz = 5620000;   break;     \
97                     case 128: /* HiperLAN2 */  khz = 5640000;   break;     \
98                     case 132: /* HiperLAN2 */  khz = 5660000;   break;     \
99                     case 136: /* HiperLAN2 */  khz = 5680000;   break;     \
100                    case 140: /* HiperLAN2 */  khz = 5700000;   break;     \
101                    case 34:  /* Japan MMAC */   khz = 5170000;   break;   \
102                    case 38:  /* Japan MMAC */   khz = 5190000;   break;   \
103                    case 42:  /* Japan MMAC */   khz = 5210000;   break;   \
104                    case 46:  /* Japan MMAC */   khz = 5230000;   break;   \
105                    default:    khz = 2412000;   break;      \
106                }                                            \
107            }
```

Figure 3.20: The channel to frequency map.

```
1476 /*
1477     ========================================================================
1478     Description:
1479         Update PortCfg->ChannelList[] according to 1) Country Region 2) RF IC type,
1480         and 3) PHY-mode user selected.
1481         The outcome is used by driver when doing site survey.
1482     ========================================================================
1483  */
1484 VOID BuildChannelList(
1485     IN PRT2570ADAPTER pAd)
1486 {
1487     UCHAR i,  index = 0;
1488     memset(pAd->PortCfg.ChannelList, 0, MAX_LEN_OF_CHANNELS);
1489
1490     // if not 11a-only mode, channel list starts from 2.4Ghz band
1491     if (pAd->PortCfg.PhyMode != PHY_11A)
1492     {
1493         switch (pAd->PortCfg.CountryRegion)
1494         {
1495             case REGION_FCC:    // 1 - 11
1496                 memcpy(&pAd->PortCfg.ChannelList[index], Ra24Ghz_FCC, sizeof(Ra24Ghz_FCC));
1497                 index += sizeof(Ra24Ghz_FCC);
1498                 break;
1499             case REGION_IC:     // 1 -11
1500                 memcpy(&pAd->PortCfg.ChannelList[index], Ra24Ghz_IC, sizeof(Ra24Ghz_IC));
1501                 index += sizeof(Ra24Ghz_IC);
1502                 break;
1503             case REGION_ISRAEL:  // 3 - 9
1504                 memcpy(&pAd->PortCfg.ChannelList[index], Ra24Ghz_ISRAEL, sizeof(Ra24Ghz_ISRAEL));
1505                 index += sizeof(Ra24Ghz_ISRAEL);
1506                 break;
1507             case REGION_ETSI:   // 1 - 13
1508                 memcpy(&pAd->PortCfg.ChannelList[index], Ra24Ghz_ESTI, sizeof(Ra24Ghz_ESTI));
1509                 index += sizeof(Ra24Ghz_ESTI);
1510                 break;
1511             case REGION_SPAIN:  // 10 - 11
1512                 memcpy(&pAd->PortCfg.ChannelList[index], Ra24Ghz_SPAIN, sizeof(Ra24Ghz_SPAIN));
1513                 index += sizeof(Ra24Ghz_SPAIN);
1514                 break;
1515             case REGION_FRANCE: // 10 -13
1516                 memcpy(&pAd->PortCfg.ChannelList[index], Ra24Ghz_FRANCE, sizeof(Ra24Ghz_FRANCE));
1517                 index += sizeof(Ra24Ghz_FRANCE);
1518                 break;
1519             case REGION_MKK:    // 14
1520                 memcpy(&pAd->PortCfg.ChannelList[index], Ra24Ghz_MKK, sizeof(Ra24Ghz_MKK));
```

Figure 3.21: The BuildChannelList function (1 of 2).

```
1521                index += sizeof(Ra24Ghz_MKK);
1522                break;
1523            case REGION_MKK1:   // 1 - 14
1524                memcpy(&pAd->PortCfg.ChannelList[index], Ra24Ghz_MKK1, sizeof(Ra24Ghz_MKK1));
1525                index += sizeof(Ra24Ghz_MKK1);
1526                break;
1527            default:            // Error. should never happen
1528                break;
1529        }
1530    }
1531
1532    if ((pAd->PortCfg.PhyMode == PHY_11A) || (pAd->PortCfg.PhyMode == PHY_11ABG_MIXED))
1533    {
1534        // 2003-10-05 john - use UNII temoparaily for all regulation domains for easy
1535        // test until RF guys confirm the supported channel plans
1536        memcpy(&pAd->PortCfg.ChannelList[index], Ra5Ghz_UNII, sizeof(Ra5Ghz_UNII));
1537        index += sizeof(Ra5Ghz_UNII);
1538    }
1539
1540    pAd->PortCfg.ChannelListNum = index;
1541
1542    DBGPRINT(RT_DEBUG_TRACE,"country code=%d, RFIC=%d, PHY mode=%d, support %d channels\n",
1543        pAd->PortCfg.CountryRegion, pAd->PortCfg.RfType, pAd->PortCfg.PhyMode,
1544        pAd->PortCfg.ChannelListNum);
1545    for (i=0;i<index;i++)
1546    {
1547        DBGPRINT(RT_DEBUG_INFO,"channel #%d\n", pAd->PortCfg.ChannelList[i]);
1548    }
1549 }
```

Figure 3.22: The BuildChannelList function (2 of 2).

Changing the Transmitted Fragment Size Recall that all packet processing is handled by the driver source code file *rtusb_data.c*. Each packet to be transmitted is assigned a priority as shown in Figure 3.23. The transmitted fragment size can be changed in the driver source code and can potentially affect the overall data throughput. To do this, the *AllowFragSize* variable can be modified. Changing this variable will affect the number of fragments each packet is divided into as shown near line 185 within the *RTUSBSendPacket* function in Figure 3.24. No packet fragmentation is allowed in multicast and broadcast traffic. The number of unicast packet fragments is determined by the packet size divided by the value of the *AllowFragSize* variable. Changing the fragment size past the threshold of 2346 bytes will cause RTS packets

to be sent when packets are divided into more than one fragment to be transmitted. This maximum MAC protocol data unit (MPDU) frame size can be changed in the driver source code by modifying the *MAX_FRAME_SIZE* constant near line 99 in the *rtmp_def.h* header file. All fragments are eventually placed in a transmission queue as shown in Figure 3.25. Packet size modification must be performed on both the transmitting and the receiving interfaces if the constant *MAX_FRAME_SIZE* is changed. Otherwise, the packets may be dropped at the receiving node as shown near line 2482 in the *RTUSBRxPacket* function of Figure 3.26. Alternatively, this frame size check can be removed from the source code to avoid the dropping of large packets.

In order to view the contents of packets that were being sent from one Wi-Fi interface to another within our test-bed, the Wireshark network protocol analyzer [34] was used to capture the data packets. TCP packets were found to have a constant size of 1514 bytes, of which, 1448 bytes made up the payload or data while the remaining 66 bytes was header information. This is the result observed from our socket-level client application sending 1024 byte chunks of a file to our server node. Whenever a file was sent using the best-effort UDP protocol instead under the same testing conditions, the packets captured with Wireshark had a maximum length of 1066 bytes, of which, exactly 1024 bytes was data and the remaining 42 bytes was header information. UDP communication is much more light-weight than TCP.

```
112 NDIS_STATUS     RTUSBSendPacket(
113         IN      PRT2570ADAPTER  pAdapter,
114         IN  struct sk_buff  *skb)
115 {
116
117
118         PVOID                   pVirtualAddress;
119         struct sk_buff_head     *pTxQueue;
120         ULONG                                   Priority;
121         UCHAR                   NumberOfFrag;
122         UCHAR                   RTSRequired;
123         UINT                    AllowFragSize;
124         UCHAR                   AccessCategory;
125         NDIS_STATUS             Status = NDIS_STATUS_FAILURE;
126         UCHAR                   PsMode;
127
128         // Init priority value
129         Priority = 0;
130         AccessCategory = 0;
131
132         if (skb && pAdapter->PortCfg.BssType == BSS_MONITOR &&
133                    pAdapter->PortCfg.MallowRFMONTx == TRUE)
134         {
135                 skb_queue_tail(&pAdapter->SendTxWaitQueue, skb);
136                 return (NDIS_STATUS_SUCCESS);
137         }
138
139         if (skb)
140         {
141                 Priority = skb->priority;
142                 // 802.11e/d4.4 June, 2003
143                 if (Priority <=2)
144                     AccessCategory = 0;
145                 else if (Priority == 3)
146                     AccessCategory = 1;
147                 else if (Priority <= 5)
148                     AccessCategory = 2;
149                 else
150                     AccessCategory = 3;
151                 DBGPRINT(RT_DEBUG_INFO, "-   %s: Priority = %d, AC = %d\n",
152                                 __FUNCTION__, Priority, AccessCategory);
153         }
154         // For TKIP, MIC value is treated as payload, it might be fragmented through
155         // different MPDUs.
156         if (pAdapter->PortCfg.WepStatus == Ndis802_11Encryption2Enabled)
```

Figure 3.23: The RTUSBSendPacket function (1 of 3).

```
157         {
158                 skb->data_len += 8;
159         }
160
161         pVirtualAddress = (PVOID)skb->data;
162         // Check for virtual address allocation, it might fail !!!
163         if (pVirtualAddress == NULL)
164         {
165                 DBGPRINT(RT_DEBUG_TRACE,"<---RTUSBSendPacket NULL pVirtualAddress\n");
166                 // Resourece is low, system did not allocation virtual address
167                 // return NDIS_STATUS_FAILURE directly to upper layer
168                 return (Status);
169         }
170
171         //
172         // Check for multicast or broadcast (First byte of DA)
173         //
174         if ((*((PUCHAR) pVirtualAddress) & 0x01) != 0)
175         {
176                 // For multicast & broadcast, there is no fragment allowed
177                 NumberOfFrag = 1;
178         }
179         else
180         {
181                 // Check for payload allowed for each fragment
182                 AllowFragSize = (pAdapter->PortCfg.FragmentThreshold) - LENGTH_802_11
183                 - LENGTH_CRC;
184
185                 // Calculate fragments required
186                 NumberOfFrag = ((skb->data_len - LENGTH_802_3 + LENGTH_802_1_H)
187                 / AllowFragSize) + 1;
188                 // Minus 1 if the size just match to allowable fragment size
189                 if (((skb->data_len - LENGTH_802_3 + LENGTH_802_1_H) % AllowFragSize)
190                 == 0)
191                 {
192                         NumberOfFrag--;
193                 }
194         }
195
196         // Check for requirement of RTS
197         if (NumberOfFrag > 1)
198         {
199                 // If multiple fragment required, RTS is required only for the first
200                 // fragment if the fragment size large than RTS threshold
201                 RTSRequired = (pAdapter->PortCfg.FragmentThreshold >
202                 pAdapter->PortCfg.RtsThreshold) ? 1 : 0;
203         }
204         else
205         {
```

Figure 3.24: The RTUSBSendPacket function (2 of 3).

```
206                 RTSRequired = (skb->data_len > pAdapter->PortCfg.RtsThreshold) ? 1 : 0;
207         }
208
209         // RTS/CTS may also be required in order to protect OFDM frame
210         if ((pAdapter->PortCfg.TxRate >= RATE_FIRST_OFDM_RATE) &&
211                 pAdapter->PortCfg.BGProtectionInUsed)
212                 RTSRequired = 1;
213         //DBGPRINT(RT_DEBUG_TRACE, "Number of fragments :%d , include RTS :%d\n",
214         //NumberOfFrag, NumberOfFrag + RTSRequired);
215
216         // Save framnet number to Ndis packet reserved field
217         RTMP_SET_PACKET_FRAGMENTS(skb, NumberOfFrag);
218
219         // Save RTS requirement to Ndis packet reserved field
220         RTMP_SET_PACKET_RTS(skb, RTSRequired);
221
222         pTxQueue = &pAdapter->SendTxWaitQueue;
223         if (INFRA_ON(pAdapter))
224         {
225                 // In infrastructure mode, simply enqueue the packet into Tx waiting
226                 // queue.
227
228                 // Enqueue Ndis packet to end of Tx wait queue
229                 skb_queue_tail(pTxQueue, skb);
230                 Status = NDIS_STATUS_SUCCESS;
231         }
232         else
233         {
234                 // In IBSS mode, power state of destination should be considered.
235                 PsMode = PWR_ACTIVE;            // Faked
236                 if (PsMode == PWR_ACTIVE)
237                 {
238                         // Enqueue Ndis packet to end of Tx wait queue
239                         skb_queue_tail(pTxQueue, skb);
240                         Status = NDIS_STATUS_SUCCESS;
241                 }
242         }
243         return Status;
244 }
```

Figure 3.25: The RTUSBSendPacket function (3 of 3).

```
2482         else if ((pAdapter->FragFrame.RxSize + PacketSize) > MAX_FRAME_SIZE)
2483         {
2484                 // Fragment frame is too large, it exeeds the maximum frame size.
2485                 // We have to drop it.
2486                 // Clear Fragment frame contents
2487                 memset(&pAdapter->FragFrame, 0, sizeof(FRAGMENT_FRAME));
2488                 Status = NDIS_STATUS_FAILURE;
2489                 break;
2490         }
```

Figure 3.26: Frame size check in the RTUSBRxPacket function.

Bit-Rate Selection The PHY rates available to the rt2570 chip-set are 1, 2, 5.5, and 11 Mbps for IEEE 802.11b and 6, 9, 12, 18, 24, 36, 48, and 54 Mbps for IEEE 802.11g. These bit-rates are defined in *rtmp_def.h* near line 524 as shown in Figure 3.27. The rates of 72 and 100 Mbps are present in the source code but were not accessible during experiments.

```
524 #define RATE_1                              0
525 #define RATE_2                              1
526 #define RATE_5_5                            2
527 #define RATE_11                             3
528 #define RATE_6                  4   // OFDM
529 #define RATE_9                  5   // OFDM
530 #define RATE_12                 6   // OFDM
531 #define RATE_18                 7   // OFDM
532 #define RATE_24                 8   // OFDM
533 #define RATE_36                 9   // OFDM
534 #define RATE_48                 10  // OFDM
535 #define RATE_54                 11  // OFDM
536 #define RATE_72                 12
537 #define RATE_100                13
538 #define RATE_FIRST_OFDM_RATE    RATE_6
539 #define RATE_AUTO_SWITCH        255 // for PortCfg.FixedTxRate only
```

Figure 3.27: Bit-rates available to the rt2570 chip-set.

To set the desired bit-rate manually from within the driver source code to 5.5 Mbps assuming IEEE 802.11b mode is already set, *pAdapter->PortCfg.TxRate = RATE_5_5* can be inserted in the code file *rtusb_main.c* near line 451 within the *CMDHandler* function. This will have the effect of continuously resetting the transmission rate to 5.5 Mbps even if TCP rate control is active due to *CMDHandler* calling the periodically executed *MlmePeriodicExec* function.

Implementing the Software-Based Spectrum Analyzer A software-based 2.4 GHz spectrum analyzer was developed for this thesis through modifying the rt2570

driver source code near line 451 of *rtusb_main.c* within the *CMDHandler* function as shown in Figure 3.28. With the per-second execution of the instructions within this case statement, the added code (i.e., lines 454 to 460) prints out the Wi-Fi channel that the transmitter is currently tuned to and displays the value of the corresponding RSSI register. An offset of -121 is applied to convert the RSSI readings to be in units of dBm. A counter is used to ensure that the Ralink radio frequency is cyclically changed to the next Wi-Fi channel every five seconds. The periodicity of this channel switching was chosen arbitrarily. It was found through our experiments that this latency is required for the RSSI registers to be updated with a stable value after each channel switch. The printed trace statements containing these RSSI values are then processed on the OS level to produce the per-channel average RSSI values. The described case statement is the starting point of the device driver's periodic task execution. This periodicity can be redefined to be more or less frequent by adjusting the value of *MLME_TASK_EXEC_INTV* (i.e., in units of milliseconds) near line 45 in *mlme.h*. The maximum Wi-Fi channel switching rate was found through experimentation to be approximately thirty-three switches per second. In this case, the *MLME_TASK_EXEC_INTV* variable would be modified to have a periodicity of 30 ms in *mlme.h* while a call to the *AsicSwitchChannel* function is placed in the case statement as shown in Figure 3.28.

```
452 case    RT_OID_PERIODIC_EXECUT:
453         MlmePeriodicExec(pAdapter);
454         DBGPRINT(RT_DEBUG_TRACE, "CH: %d, RSSI: %d\n", ch, pAdapter->PortCfg.LastRssi
455         - pAdapter->BBPTuningParameters.RSSIToDbmOffset);
456         count++;
457         if(count % 5 == 0){
458                 ch++; ch = ch % 12; if(ch == 0){ch++;}
459                 AsicSwitchChannel(pAdapter, ch);
460         }
```

Figure 3.28: The software-based spectrum analyzer implementation.

3.2 NG-Mesh Test-Bed Setup

The subsections to follow describe the setup of the NG-Mesh test-bed and all of its components.

3.2.1 The Platform

Table 3.1 lists the main components making up the central platform of the NG-Mesh that was developed for this thesis.

Component	**Specifications**
PC	Quad-core Q9300 2.53GHz CPU, 4GB RAM, 100 GB HDD space
VM	VMware Workstation 7, 512 MB RAM, 1 CPU, 8 GB HDD space
OS	64-bit RHEL 5.4, kernel version 2.6.18

Table 3.1: The NG-Mesh components.

Two pairs of Linksys WUSB54Gv4 Wi-Fi interfaces making up two network nodes connect to the central PC via a non-attenuating four-port USB hub. Each node is controlled by a virtualized Linux environment that configures one dedicated receiving and one dedicated transmitting Wi-Fi interface. The resulting table-top test-bed is shown in Figure 3.29. All Wi-Fi interfaces are configured to operate in ad-hoc mode within the same Wi-Fi cell so that no association with any AP is required. In this way, the nodes can communicate directly with one another. The receiver of each node operates on a pre-defined Wi-Fi channel and is assigned a unique IP address while the transmitter of each is tuned to another radio frequency and is also given an IP address of its own.

Figure 3.29: The NG-Mesh test-bed.

The *iptables* software firewall in Linux is disabled to avoid the need to configure specific routing rules for each network interface. Within a single Linux OS, it is not easy to configure communication between multiple locally connected Wi-Fi interfaces. All traffic between them will not be routed properly through the intended communication endpoints. Instead traffic between all local network interfaces will automatically be routed through the virtual loop-back interface internal to the Linux OS. In order to avoid this unwanted behavior, all locally connected Wi-Fi interfaces that are to communicate with each other must be partitioned into independent virtual instances of the Linux OS to emulate physically separate network nodes. Hence the use of virtualized environments in our test-bed. Since the network topology of the NG-Mesh is defined completely in the virtual space of a single PC, configuration of all network interfaces in our test-bed is performed via this central console. All network interface configurations take place via command-line interface (CLI) within each Linux virtual machine (VM) (i.e., the network nodes). Shell scripts were written

for this thesis to streamline repetitive commands in order to configure and monitor each of these interfaces in our Wi-Fi network proficiently.

The remaining subsections describe the Linux tools used, the TCP and UDP socket applications, as well as the shell scripts that were developed during this thesis for the configuration and monitoring of the Wi-Fi interfaces within the NG-Mesh.

3.2.2 Interface Configuration in Ad-Hoc Mode

By default, the Linksys WUSB54Gv4 Wi-Fi interfaces operate in the IEEE 802.11b PHY mode. Configuring any number of these interfaces into an active ad-hoc network involves running the *rausb%d.adhoc.debug.sh* shell script that was written for this thesis. Figure 3.30 presents the scripts' contents. The first line ensures that all Ralink driver modules that are currently loaded in the kernel are removed. The second line loads the rt2570 driver module into the kernel while enabling full debug mode. Next, each of the network interfaces is configured to join the same ad-hoc cell. This is implemented in a *for loop* as *ifconfig* is queried to obtain a complete list of all currently connected Ralink adapters. For each of the interfaces detected in the list, the following sequence of configurations is performed.

1. **iwconfig <interface> mode Ad-Hoc:** Puts the interface into ad-hoc mode.
2. **iwconfig <interface> enc off:** Disables encryption.
3. **iwconfig <interface> essid test:** Sets the SSID to *test*.
4. **iwconfig <interface> channel 11:** Tunes the transmitter to channel 11.
5. **ifconfig <interface> <IP address>:** Powers on the interface and assigns a

network layer address starting from 192.168.81.138 while incrementing by one in the last octet with each additional interface configured.

6. **iwconfig <interface> ap <cell ID>:** The final conditional statement is used for all interfaces other than the first one configured (i.e., *rausb0*) to join the ad-hoc cell with the cell ID associated with *rausb0*.

```
# Ensure Correct Driver is Loaded
for i in rt2570 rt2500usb rt2x00usb rt2x00lib; do rmmod $i; done
insmod ../rt2570-cvs-2009041204/Module/rt2570.ko debug=31

# Configure Ad-Hoc Network
for i in `ifconfig -a | grep rausb | awk '{print $1}'`; do iwconfig $i mode Ad-H
oc; iwconfig $i enc off; iwconfig $i essid test; iwconfig $i channel 11; ifconfi
g $i 192.168.81.`expr 138 + $(echo $i | sed -e "s/^.*\(.\)$/\1/")` netmask 255.2
55.255.0 up; if [ $(echo $i | sed -e "s/^.*\(.\)$/\1/") -ne 0 ]; then iwconfig $
i ap `iwconfig rausb0 | grep Cell | awk '{print $5}'`; fi; done

# Initiate Ad-Hoc Network
for i in `ifconfig -a | grep rausb | awk '{print $1}'`; do iwlist $i scan; done
```

Figure 3.30: Contents of rausb%d.adhoc.debug.sh.

The interfaces were configured with the bit-rate of 11 Mbps during testing with the *iwconfig <interface> rate 11* command. The last line in the shell script instructs each newly configured interface to perform a scan of its surroundings to detect nearby networks. This sanity check is performed to ensure that all newly configured interfaces are operational. The following command ensures that each interface was configured correctly and has successfully joined a common ad-hoc network. If all interfaces were configured correctly, then all lines of output (i.e., ad-hoc Cell ID) from this command will be identical.

for i in 'ifconfig -a | grep rausb | awk 'print $1'; do iwconfig $i | grep Cell | awk 'print $5'; done

The number of Wi-Fi interfaces actively using the rt2570 kernel driver module at each node can also be verified. For example, if there is only one Wi-Fi interface configured on the node, then the output from the *lsmod* command should be similar to Figure 3.31.

```
rt2570                282496  1
```

Figure 3.31: The rt2570 driver in use by one Ralink interface.

3.2.3 Interface Configuration in Infrastructure Mode

The *rausb0.ap.sh* shell script was developed for this thesis to configure the Wi-Fi interfaces into the managed mode to associate with an AP. This script was only tested with APs protected with WEP encryption. For rt2570 chip-sets to interface with APs running WPA encryption schemes, further configuration is required. Figure 3.32 displays the contents of the script. The managed mode interface configuration commands are described next.

- **modprobe rausb0 ifname=rausb%d:** Loads the Wi-Fi driver into the Linux kernel without enabling debug mode and sets the interface naming convention to increment by one starting from *rausb0* for each additional interface connected to the system.
- **iwconfig rausb0 key <KEY>:** This is the same as using *enc* to set the encryption key.
- **iwconfig rausb0 ap <MAC Address of AP>:** Specifies the AP MAC address to connect to.

- **dhclient -l -d rausb0:** Sends a dynamic host configuration protocol (DHCP) request to the AP asking for a network layer address assignment lease and waits for a response.

- **ifconfig eth0 down:** Powers off the wired interface on the system that may be currently connected to the local AP at the test site (i.e., in our case, this interface was named *eth0*).

```
for i in rt2570 rt2500usb rt2x00usb rt2x00lib; do rmmod $i; done
modprobe rausb0 ifname=rausb%d
ifconfig rausb0 192.168.81.138 netmask 255.255.255.0 up
iwconfig rausb0 essid ng-link
iwconfig rausb0 key 4941055139653420548671826 5
iwconfig rausb0 channel 3
iwconfig rausb0 ap 34:ef:44:ac:eb:c9
dhclient -1 -d rausb0
ifconfig eth0 down
```

Figure 3.32: The contents of rausb0.ap.sh.

To check for NG-Mesh connectivity to the AP, an internet control message protocol (ICMP) ping request from the configured interface can be sent. Since the AP in our tests provides internet connectivity, the ping test was sent to *www.google.com* using the command *ping -I rausb0 www.google.com*. If connected properly, the output should be similar to Figure 3.33.

```
PING www.l.google.com (66.249.81.104) from 192.168.81.138 rausb0: 56(84) bytes o
f data.
64 bytes from lga15s01-in-f104.1e100.net (66.249.81.104): icmp_seq=1 ttl=128 tim
e=33.1 ms
```

Figure 3.33: Successfully pinging www.google.com.

3.3 The Socket Programs

TCP and UDP socket applications were written in the *C* programming language for this thesis to send and receive data between the NG-Mesh Wi-Fi interfaces. The receiver interface of each network node is operated by a server socket application that writes all received data coming from the client node to disk. Conversely, the transmitter of each node is operated by a client socket application that reads keyboard text or a file from disk as input and sends it wirelessly to the server node in the network. In our implementation, the server's role is to continuously listen on a hard-coded port for data to receive from a transmitting client. Each Wi-Fi interface in the test-bed is assigned and bound to a unique IP address and port. The remaining subsections contain the socket programming source code, some implementation details, and a short guide for compiling and running the software.

3.3.1 The Source Code Files

UDP communication was implemented in the *udpclient.c* and *udpserver.c* code files while TCP communication was implemented in the *tcpclient.c* and *tcpserver.c* code files. These applications were developed with reference to socket programming guidelines from [35], [36], and [37].

3.3.2 The UDP Client

The complete source code for the UDP client program is shown in Figure 3.34 to Figure 3.37. In order to control the routing of packets within the NG-Mesh, special

measures were taken to ensure that specific network node interfaces were in communication with each other. For instance, the call to *SO_BINDTODEVICE* forces all data packets to exit the client node only through the hard-coded interface *rausb1* as shown in Figure 3.35. Additionally, binding the socket to a hard-coded IP address and port ensures that all packets originating from the client interface will have the correct source information in its headers.

When the file transfer option in the code is uncommented and thus enabled, file descriptors are set up to read from the file handle opened to access the specified file located on the local disk drive. Chunks of the file are then sent through the bound socket to the server's listening interface. The starting and ending transmission times are captured during the code's execution which is accurate to the nanosecond.

The flow of items sent from the client to the server occurs in the following sequence. First the file size is measured and sent to the server in order for the server-side application to know how much data to expect to receive for the complete file. The filename is sent next in order to notify the server which file is being received. Finally the file data is sent.

```
/*******************************************************************************
McMaster University
EE701 – Stanley Ng (0050336)
UDP Client Main Program
Sends user input or file to UDP server

Reference:
     Michael J. Donahoo, Kenneth L. Calvert, "TCP/IP Sockets In C," 2001
     Brian W. Kernighan, Dennis M. Ritchie, "The C Programming Language," Second Edition, 1998
     Brian Hall, “Beej’s Guide to Network Programming Using Internet Sockets,” 2009
*******************************************************************************/

// Include header files
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

// Start of main program
int main(){

    // Variable declarations
    int sock, rc;
    struct sockaddr_in server_addr, loc_addr;
    char send_data[1024];
    FILE *fp;
    char filename [25] = "space.jpg";
    char buffer [1024];
    int total_bytes=0, bytes_sent = 0;

    // Create socket for UDP communication
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1){
        perror("Error Opening Socket\n");
        exit(1);
    }
```

Figure 3.34: The UDP client source code (1 of 4).

```
43
44
45      // Specify packet egress interface and bind to socket
46      if (setsockopt(sock, SOL_SOCKET, SO_BINDTODEVICE, "rausb1", 6) == -1) {
47          perror("Error Binding to Interface\n");
48          exit(1);
49      }
50
51      // Specify server network layer address and port
52      server_addr.sin_family = AF_INET;
53      server_addr.sin_port = htons(5000);
54      server_addr.sin_addr.s_addr = inet_addr("192.168.81.138");
55      bzero(&(server_addr.sin_zero),8);
56
57      // Specify client network layer address and port
58      loc_addr.sin_family = AF_INET;
59      loc_addr.sin_port = htons(5000);
60      loc_addr.sin_addr.s_addr = inet_addr("192.168.81.139");
61
62      // Bind client interface to socket
63      if (bind(sock, (struct sockaddr *)&loc_addr, sizeof(struct sockaddr)) == -1) {
64          perror("Error Binding to IP and Port\n");
65          exit(1);
66      }
67
68
69      /*****************************************************************************
    **
70      Option 1 - Send text to server
71      *****************************************************************************
    */
72      /*****************************************************************************
    **
73      while (1){
74
75          // Get user input
76          printf("Send a text message to the server: ");
77          gets(send_data);
78
79          // Send user input to server
80          sendto(sock, send_data, strlen(send_data), 0, (struct sockaddr *)&server_addr,
     sizeof(struct sockaddr));
81      }
```

Figure 3.35: The UDP client source code (2 of 4).

```
82      ********************************************************************************
    */
83
84      /********************************************************************************
    **
85      Option 2 - Send file to server
86      ********************************************************************************
    */
87      /********************************************************************************
    **
88      // Setup file pointer
89      fd_set wfds;
90      fp = fopen("/root/802.11/C/OUT/space.jpg", "rb");
91
92      // Meaure file size
93      struct stat fileinfo;
94
95      // Specify file to send to server
96      stat("/root/802.11/C/OUT/space.jpg", &fileinfo);
97      total_bytes = fileinfo.st_size;
98
99      while (fp) {
100         FD_ZERO(&wfds);
101         FD_SET(sock, &wfds);
102         if (FD_ISSET(sock, &wfds)) {
103             if (!feof(fp)) {
104
105  // Send file size to server
106                 if (total_bytes) {
107
108      // Capture start data transmission timestamp
109                 system("echo TX Start Time: `date +%H:%M:%S.%N` >> /root/802.11/C/vm1.tx.
    log");
110  rc = sendto(sock, &total_bytes, sizeof(total_bytes), 0, (struct sockaddr *)&server_ad
    dr, sizeof(struct
111  sockaddr));
112                     printf("total_bytes: %d\n", total_bytes);
113                     total_bytes = 0;
114                 }
115
116   // Send filename to server
117                 else if (strcmp(filename,"NULL") != 0) {
```

Figure 3.36: The UDP client source code (3 of 4).

```
118                     rc = sendto(sock, filename, sizeof(filename), 0, (struct sockaddr
    *)&server_addr, sizeof(struct
119           sockaddr));
120                     printf("filename: %s\n", filename);
121                     strcpy(filename, "NULL");
122                 }
123
124   // Send file data to server
125                 else {
126                     rc = fread(buffer, 1, sizeof(buffer), fp);
127                     rc = sendto(sock, buffer, rc, 0, (struct sockaddr *)&server_addr,
    sizeof(struct sockaddr));
128                     if (rc < 0) {
129                         printf("Transmission Error!\n");
130                         break;
131                     }
132                     bytes_sent+=rc;
133                     printf("tx: %d\n",bytes_sent);
134                     //sleep(1);
135                 }
136             }
137             else break;
138         }
139     }
140     // Capture end data transmission timestamp
141     system("echo TX End Time: `date +%H:%M:%S.%N` >> /root/802.11/C/vm1.tx.log");
142     printf("File Transfer Complete\n");
143
144     // Release File Pointer
145     fclose(fp);
146     ********************************************************************************
    */
147
148     // Release Socket
149     close(sock);
150 }
151 // End of main program
```

Figure 3.37: The UDP client source code (4 of 4).

3.3.3 The UDP Server

The complete source code for the UDP server program is shown in Figure 3.38 to Figure 3.41. On the server-side, the routing of packets is configured to force all received data to enter only through the hard-coded Wi-Fi interface with the familiar call to the *SO_BINDTODEVICE* socket option. In contrast to the client-side application, the server binds the server IP address and port to the socket in order to instruct the server interface, which is also bound to the socket, to accept only the packets that have a destination address and port that matches the ones specified and bound to that interface.

Being at the receiving end of the communication stream, the file descriptor settings mirror the ones found at the client-side of the socket. Thus, on the server-side, data is read from the socket and written to the specified location on the local hard disk. Similar to the capturing of starting and ending time-stamps during transmission on the client endpoint, the starting and ending time-stamps for data reception is also captured on the server-side.

In order to compliment the order of items sent by the client application, the server must listen for the client-sent items in the client-sent order. Namely, file size comes first, followed by filename, and finally the file data is expected last.

```
/*******************************************************************************
McMaster University
EE701 – Stanley Ng (0050336)
UDP Server Main Program
Listens for user input or file from UDP client

Reference:
     Michael J. Donahoo, Kenneth L. Calvert, "TCP/IP Sockets In C," 2001
     Brian W. Kernighan, Dennis M. Ritchie, "The C Programming Language," Second Editi
on, 1998
     Brian Hall, "Beej's Guide to Network Programming Using Internet Sockets," 2009
*******************************************************************************/

// Include header files
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

// Start of main program
int main(){

    // Variable declarations
    int addr_len, bytes_read, sock, rc;
    char recv_data[1024];
    struct sockaddr_in server_addr , client_addr;
    FILE *fp;
    char path [100] = "";
    char filename [25] = "NULL";
    char buffer [1024] = { 0 };
    int total_bytes = 0;
    int bytes_received = 0;

    // Open socket for UDP communication
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1){
        perror("Error Opening Socket\n");
        exit(1);
    }
```

Figure 3.38: The UDP server source code (1 of 4).

```
43
44      // Set packet ingress interface and bind to socket
45      if (setsockopt(sock, SOL_SOCKET, SO_BINDTODEVICE, "rausb0", 6) == -1){
46          perror("Error Binding to Interface\n");
47          exit(1);
48      }
49
50      // Specify server network layer address and port
51      server_addr.sin_family = AF_INET;
52      server_addr.sin_port = htons(5000);
53      server_addr.sin_addr.s_addr = inet_addr("192.168.81.138");
54      bzero(&(server_addr.sin_zero),8);
55
56      // Bind server interface to socket
57      if (bind(sock,(struct sockaddr *)&server_addr, sizeof(struct sockaddr)) == -1){
58          perror("Error Binding to IP and Port");
59          exit(1);
60      }
61
62      addr_len = sizeof(struct sockaddr);
63      printf("\nUDP Server Listening for Incoming Connection on Port 5000\n");
64      fflush(stdout);
65
66      /*************************************************************************
    **
67      Option 1 - Receive text from client
68      *************************************************************************
    */
69      /*************************************************************************
    **
70      while (1){
71          bytes_read = recvfrom(sock,recv_data,1024,0, (struct sockaddr *)&client_addr,
    &addr_len);
72          recv_data[bytes_read] = '\0';
73          printf("Message from %s:%d - %s\n", inet_ntoa(client_addr.sin_addr), ntohs(cli
    ent_addr.sin_port), recv_data);
74          fflush(stdout);
75      }
76      *************************************************************************
    */
77
78      /*************************************************************************
    **
```

Figure 3.39: The UDP server source code (2 of 4).

```
79      Option 2 - Receive file from client
80      ********************************************************************************
    */
81      /********************************************************************************
    **
82
83      // Setup file pointer
84      fd_set rfds;
85
86      while(1){
87          FD_ZERO(&rfds);
88          FD_SET(sock, &rfds);
89          if (FD_ISSET(sock, &rfds)){
90
91              // Receive file size
92              if (!total_bytes){
93
94            // Capture start data reception timestamp
95            system("echo RX Start Time: `date +%H:%M:%S.%N` >> /root/802.11/C/vm1.rx.log
    ");
96
97            memset(&total_bytes, 0, sizeof(total_bytes));
98    rc = recvfrom(sock,&total_bytes,sizeof(total_bytes),0, (struct sockaddr *)&client_ad
    dr, &addr_len);
99                  printf("total_bytes: %d\n",total_bytes);
100             }
101
102             // Receive filename
103             else if (strcmp(filename,"NULL") == 0){
104                 rc = recvfrom(sock,filename,sizeof(filename),0, (struct sockaddr *)&cl
    ient_addr, &addr_len);
105
106                 // Specify location to store received file data
107                 strcat(path,"/root/802.11/C/IN/");
108                 strcat(path,filename);
109                 fp = fopen(path, "wb");
110                 printf("path: %s\n",path);
111             }
112
113             // Receive file data
114             else {
115                 memset(buffer, 0, sizeof(buffer));
```

Figure 3.40: The UDP server source code (3 of 4).

```
116                 rc = recvfrom(sock,buffer,sizeof(buffer),0, (struct sockaddr *)&client
    _addr, &addr_len);
117                 if (rc < 0) {
118                     printf("Transmission Error!\n");
119                break;
120          }
121          bytes_received+=rc;
122          printf("rx: %d\n",bytes_received);
123          rc = fwrite(buffer, 1, rc, fp);
124             }
125         }
126
127         // Ensure file transfer is complete
128         if (bytes_received == total_bytes){
129
130             // Capture end data reception timestamp
131             system("echo RX End Time: `date +%H:%M:%S.%N` >> /root/802.11/C/vm1.rx.log
    ");
132             printf("File Transfer Complete\n");
133             break;
134         }
135         else{
136             printf("Incomplete file received total size %d, received %d\n", total_byte
    s, bytes_received);
137         }
138     }
139
140     // Release file pointer
141     fclose(fp);
142     ********************************************************************************
    */
143
144     // Release socket
145     close(sock);
146     return 0;
147 }
148 // End of main program
```

Figure 3.41: The UDP server source code (4 of 4).

3.3.4 The TCP Client

The complete source code for the TCP client program is shown in Figure 3.42 to Figure 3.45. Unique to the TCP client is the *connect* function that is used to establish a connection between the local client and remote server sockets.

```
1  /*****************************************************************************
2  McMaster University
3  EE701 – Stanley Ng (0050336)
4  TCP Client Main Program
5  Sends user input or file to TCP server
6
7  Reference:
8       Michael J. Donahoo, Kenneth L. Calvert, "TCP/IP Sockets In C," 2001
9       Brian W. Kernighan, Dennis M. Ritchie, "The C Programming Language," Second Editi
   on, 1998
10      Brian Hall, “Beej’s Guide to Network Programming Using Internet Sockets,” 2009
11 *****************************************************************************/
12
13 // Include header files
14 #include <sys/stat.h>
15 #include <sys/socket.h>
16 #include <sys/types.h>
17 #include <netinet/in.h>
18 #include <netdb.h>
19 #include <stdio.h>
20 #include <string.h>
21 #include <stdlib.h>
22 #include <unistd.h>
23 #include <errno.h>
24
25 // Start of main program
26 int main(){
27
28      // Variable declarations
29     int sock, rc;
30     struct sockaddr_in server_addr, loc_addr;
31     FILE *fp;
32     char filename [25] = "space.jpg";
33     char buffer [1024];
34     int total_bytes=0, bytes_sent = 0;
35
36     // Create socket for TCP communication
37     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1){
38         perror("Error Opening Socket\n");
39         exit(1);
40     }
41
42     // Specify packet egress interface and bind to socket
```

Figure 3.42: The TCP client source code (1 of 4).

```
43      if (setsockopt(sock, SOL_SOCKET, SO_BINDTODEVICE, "rausb1", 6) == -1){
44          perror("Error Binding to Interface\n");
45          exit(1);
46      }
47
48      // Specify server network layer address and port
49      server_addr.sin_family = AF_INET;
50      server_addr.sin_port = htons(5000);
51      inet_aton("192.168.81.138", &server_addr.sin_addr);
52      memset(&(server_addr.sin_zero),0,8);
53
54      // Specify client network layer address and port
55      loc_addr.sin_family = AF_INET;
56      loc_addr.sin_port = htons(5000);
57      inet_aton("192.168.81.139", &loc_addr.sin_addr);
58      memset(&(loc_addr.sin_zero),0,8);
59
60      // Bind client interface to socket
61      if (bind(sock, (struct sockaddr *)&loc_addr, sizeof(struct sockaddr)) == -1){
62          perror("Error Binding to IP and Port");
63          exit(1);
64      }
65
66      // Establish connection to server
67      if (connect(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) == -1)
    {
68          perror("Error Connecting to Server\n");
69          exit(1);
70      }
71
72      /*****************************************************************************
    **
73      Option 1 - Send text to server
74      *****************************************************************************
    */
75      /*****************************************************************************
    **
76      write(sock, "hello!", 6);
77      *****************************************************************************
    */
78
79      /*****************************************************************************
    **
```

Figure 3.43: The TCP client source code (2 of 4).

```
 80     Option 2 - Send file to server
 81     ********************************************************************************
    */
 82     /********************************************************************************
    **
 83
 84     // Setup file pointer
 85     fd_set wfds;
 86     fp = fopen("/root/802.11/C/OUT/space.jpg", "rb");
 87
 88     // Meaure file size
 89     struct stat fileinfo;
 90
 91     // Specify file to send to server
 92     stat("/root/802.11/C/OUT/space.jpg", &fileinfo);
 93     total_bytes = fileinfo.st_size;
 94
 95     while (fp) {
 96         FD_ZERO(&wfds);
 97         FD_SET(sock, &wfds);
 98         if (FD_ISSET(sock, &wfds)) {
 99             if (!feof(fp)) {
100
101                 // Send file size to server
102          if (total_bytes) {
103
104             // Capture start data transmission timestamp
105             system("echo TX Start Time: `date +%H:%M:%S.%N` >> /root/802.11/C/vm1.tx.l
    og");
106
107             rc = send(sock, &total_bytes, sizeof(total_bytes), 0);
108             printf("total_bytes: %d\n", total_bytes);
109             total_bytes = 0;
110         }
111
112         // Send filename to server
113         else if (strcmp(filename,"NULL") != 0) {
114             rc = send(sock, filename, sizeof(filename),0);
115             printf("filename: %s\n", filename);
116             strcpy(filename, "NULL");
117         }
118
119         // Send file data to server
```

Figure 3.44: The TCP client source code (3 of 4).

```
120         else {
121             rc = fread(buffer, 1, sizeof(buffer), fp);
122             rc = send(sock, buffer, rc , 0);
123             if (rc < 0) {
124                 printf("Transmission Error!\n");
125                 break;
126             }
127             bytes_sent+=rc;
128             printf("tx: %d\n",bytes_sent);
129             //sleep(1);
130         }
131             }
132             else break;
133         }
134     }
135     // Capture end data transmission timestamp
136     system("echo TX End Time: `date +%H:%M:%S.%N` >> /root/802.11/C/vm1.tx.log");
137     printf("File Transfer Complete\n");
138
139     // Release file pointer
140     fclose(fp);
141     ********************************************************************************
    */
142
143     // Release socket
144     close(sock);
145     return 0;
146 }
147 // End of main program
```

Figure 3.45: The TCP client source code (4 of 4).

3.3.5 The TCP Server

The complete source code for the TCP server program is shown in Figure 3.46 to Figure 3.49. Specific to the TCP server are the *listen* and *accept* functions that are used to put the server socket into listening mode to wait for an incoming client connection request and then accept it.

```
/*******************************************************************************
McMaster University
EE701 – Stanley Ng (0050336)
TCP Server Main Program
Listens for user input or file from TCP client

Reference:
     Michael J. Donahoo, Kenneth L. Calvert, "TCP/IP Sockets In C," 2001
     Brian W. Kernighan, Dennis M. Ritchie, "The C Programming Language," Second Edition, 1998
     Brian Hall, “Beej’s Guide to Network Programming Using Internet Sockets,” 2009
*******************************************************************************/

// Include header files
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

// Start of main program
int main(){

    // Variable declarations
    int sock, connected, bytes_read, rc, sin_size;
    char buf[1024] = { 0 };
    FILE *fp;
    char path [100] = "";
    char filename [25] = "NULL";
    char buffer [1024] = { 0 };
    int total_bytes=0;
    int bytes_received = 0;
    struct sockaddr_in server_addr,client_addr;

    // Open socket for TCP communication
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1){
        perror("Error Opening Socket\n");
        exit(1);
    }
```

Figure 3.46: The TCP server source code (1 of 4).

```
43
44      // Specify packet ingress interface and bind to socket
45      if (setsockopt(sock, SOL_SOCKET, SO_BINDTODEVICE, "rausb0", 6) == -1){
46          perror("Error Binding to Interface\n");
47          exit(1);
48      }
49
50      // Specify server network layer address and port
51      server_addr.sin_family = AF_INET;
52      server_addr.sin_port = htons(5000);
53      inet_aton("192.168.81.138", &server_addr.sin_addr);
54      memset(&(server_addr.sin_zero),0,8);
55
56      // Bind server interface to socket
57      if (bind(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) == -1){
58          perror("Error Binding to IP and Port\n");
59          exit(1);
60      }
61
62      // Setup socket to listen for incoming client connections
63      if (listen(sock, 3) == -1){
64          perror("Error Setting Socket Into Listen Mode\n");
65          exit(1);
66      }
67
68      printf("\nTCP Server Listening for Incoming Connection on Port 5000\n");
69      fflush(stdout);
70      sin_size = sizeof(struct sockaddr_in);
71
72      // Accept incoming client connections
73      connected = accept(sock, (struct sockaddr *)&client_addr,&sin_size);
74
75      /*****************************************************************************
    **
76      Option 1 - Receive text from client
77      *****************************************************************************
    */
78      /*****************************************************************************
    **
79      printf("\n Incoming connection from %s:%d received\n",
80      inet_ntoa(client_addr.sin_addr),ntohs(client_addr.sin_port));
81
82      bytes_read = read(connected, buf, sizeof(buf));
```

Figure 3.47: The TCP server source code (2 of 4).

```
 83     if( bytes_read > 0 ) {
 84         printf("Received text message: %s\n", buf);
 85     }
 86     ********************************************************************************
    */
 87
 88     /********************************************************************************
    **
 89     Option 2 - Receive file from client
 90     ********************************************************************************
    */
 91     /********************************************************************************
    **
 92
 93     // Setup file pointer
 94     fd_set rfds;
 95
 96     while (1) {
 97         FD_ZERO(&rfds);
 98         FD_SET(connected, &rfds);
 99         if (FD_ISSET(connected, &rfds)) {
100
101             // Receive file size from client
102             if (!total_bytes) {
103         memset(&total_bytes, 0, sizeof(total_bytes));
104         rc = recv(connected, &total_bytes, sizeof(total_bytes), 0);
105
106         // Capture start data reception timestamp
107         system("echo RX Start Time: `date +%H:%M:%S.%N` >> /root/802.11/C/vm1.rx.log")
    ;
108         printf("total_bytes: %d\n",total_bytes);
109             }
110
111             // Receive filename from client
112             else if (strcmp(filename,"NULL") == 0) {
113                 rc = recv(connected, filename, sizeof(filename),0);
114
115         // Specify location to store received data
116         strcat(path,"/root/802.11/C/IN/");
117         strcat(path,filename);
118         fp = fopen(path, "wb");
119         printf("path: %s\n",path);
120             }
```

Figure 3.48: The TCP server source code (3 of 4).

```
121
122             // Receive file data from client
123             else {
124                 memset(buffer, 0, sizeof(buffer));
125                 rc = recv(connected, buffer, sizeof(buffer), 0);
126                 if (rc < 0) {
127             printf("Transmission Error!\n");
128             break;
129                 }
130                 bytes_received+=rc;
131                 printf("rx: %d\n",bytes_received);
132                 rc = fwrite(buffer, 1, rc, fp);
133             }
134         }
135
136         // Ensure file transfer is complete
137         if (bytes_received == total_bytes) {
138
139             // Capture end data reception timestamp
140             system("echo RX End Time: `date +%H:%M:%S.%N` >> /root/802.11/C/vm1.rx.log
    ");
141             printf("File Transfer Complete\n");
142             break;
143         }
144         else {
145             printf("Incomplete file received total size %d, received %d\n", total_byte
    s, bytes_received);
146         }
147     }
148
149     // Release file pointer
150     fclose(fp);
151     ********************************************************************************
    */
152
153     // Release socket
154     close(sock);
155     return 0;
156 }
157 // End of main program
```

Figure 3.49: The TCP server source code (4 of 4).

3.3.6 Compiling and Running the Socket Applications

The following is a list of configurable parameters in the socket source code that is presented in this section.

- path, filename

- buf, buffer, send_data, received_data
- Client and server IP addresses, port, and interface name

filename specifies the name of the file to be sent from the client application to the server. *path* represents the absolute location of the file on disk to be sent from the client and stored on the server. The buffer variables determine the fragment size of the file to be read from the client-side local hard drive. The fragments are then sent through the socket from the client endpoint and received on the server-side of the socket to be written to disk. The client and server IPs, ports, and interfaces are also key parameters that should be considered for successful operation. Careful configuration of these variables is fundamental to proper client and server interaction. Due to the static tests that were performed repeatedly during this thesis work, the parameters listed in this subsection were always configured with the same values.

Prior to compiling any socket code, the user must decide whether the transfer of text strings or a file is desired of the applications. If the former functionality is chosen, then the code segment labeled *Option 1* in both client and server source code files must be uncommented. If the latter functionality is elected, then the code segment labeled *Option 2* in all relevant source code files must be uncommented instead. Compiling the socket code can be accomplished with any *C* or *C++* compiler. Under Linux, this can be done using the following shell command.

gcc -o <resulting executable filename> <filename of the source code file to be compiled>

The server application on the server node must be started first. Once the server application is ready to process client requests, the client application can then be initiated on the client node.

3.4 The Metrics Collector

A combination of using the Wireshark network protocol analyzer, shell scripting, rt2570 driver diagnostic messages, and socket code debug statements makes it possible for many statistics to be recorded in real-time during test-bed network operations on a per Wi-Fi interface basis. Key metrics captured include RSSI, PER, PRR, CQI, file transfer duration, SINR, and effective data-rate.

The *monitor.sh* shell script was written for this thesis to capture Wi-Fi driver diagnostic messages and is shown in Figure 3.50. This script implements a forever loop that puts all OS-level diagnostic messages into a log file that can be analyzed at a later time. As an example, if a trace statement containing the W-Fi channel number and corresponding RSSI reading is inserted into the driver source code to implement a software-based spectrum analyzer, then this information will be placed in the *rt2570.debug.log* file when the *monitor.sh* script is run. Specific data can be extracted from this file and statistics can then be calculated from the extracted data.

```
while [ 1 ]
do
        dmesg -c >> rt2570.debug.log
done
```

Figure 3.50: The general contents of monitor.sh.

The *collect.sh* script was written for this thesis to extract per channel stats from the metrics gathered by the *monitor.sh* script. This script calls upon yet another script called *process.sh* in order to compute average RSSI values that were reported by the rt2570 driver. The pseudo-code of the *collect.sh* script is shown in Figure 3.51. The *for* statement loops through each of the eleven North American Wi-Fi channels and extracts per-channel RSSI values from the log file on disk. Once all statistics are

gathered, the *process.sh* script as shown in Figure 3.52 is executed. This script first initializes the variables a and b to zero before computing the RSSI averages for each of the eleven channels.

```
for i in 1 2 3 4 5 6 7 8 9 10 11; do `grep Channel rt2570.debug.log | grep RSSI
| awk '{print Value}' > tmp`; `./process.sh`;  done
```

Figure 3.51: The general contents of collect.sh.

```
a=`expr 0`
b=`expr 0`
for i in `cat tmp`; do a=`expr $a + $i`; done
b=`wc -l tmp`
a=$(echo "scale=2; $a/$b" | bc)
echo $a
```

Figure 3.52: The general contents of process.sh.

3.5 The Wi-Spy 2.4 GHz Spectrum Analyzer

The Wi-Spy 2.4i entry-level ISM spectrum analyzer was used in the NG-Mesh. Accompanying software is available for Microsoft and Apple OS's [38], or Linux [39], which can be used to plot real-time graphical representations of the RSSI levels in the 2.4 GHz band. The software can also compute the average and peak RSSI levels over time on a per frequency basis and detect nearby networks. The Chanalyzer Lite software for Microsoft and Apple systems can monitor the frequency range between 2.4 GHz and 2.492 GHz in 375 kHz increments and can detect RSSI levels in the range between -102 dBm and -6.5 dBm in 0.5 dBm increments. The plots generated are updated approximately once per second. Under Linux, the Kismet Spectrum-Tools software enables Wi-Spy interfaces to monitor the frequency range between 2.4 GHz

and 2.483 GHz in 199 kHz steps and report 419 samples approximately once every 300 ms. Once plugged into the USB port, the Wi-Spy spectrum analyzer was recognized as a mouse or keyboard. This suggests that the chip-set employed in this product is a Wi-Fi mouse or keyboard radio receiver. A comparison between the Kismet and Chanalizer software packages reveal that the latter offers more features and views.

3.5.1 Using the Spectrum Analyzer Software Under Linux

The Kismet Spectrum-Tools software can be downloaded from [39]. After uncompressing the package, the following sequence of commands was executed to activate the spectrum analyzer's Wi-Fi interface.

1. **Change directory to the uncompressed Spectrum-Tools folder.**
2. **./configure:** Audits the system against a checklist to ensure that all software dependency requirements for using the spectrum analyzer software are met.
3. **make:** Compiles the spectrum analyzer software source code.
4. **Execute any of the following four applications that will result from successful compilation.**
    - **spectool_gtk:** A GUI-based application that shows spectral data in the following views.
        - **Planar View:** Plots the average, peak, and current RSSI levels in the 2.4 GHz band.
        - **Spectral View:** Plots the spectrum-wide intensity of RSSI levels in real-time.

  - **Topographic View:** Plots the signal peaks over time across the 2.4 GHz spectrum.

- **spectool_raw:** A CLI-based program that outputs textual spectral data readings in real-time approximately once every 300 ms.
- **spectool_net:** A network server for remote sensor configuration.
- **spectool_curses:** Graphs spectral data in the shell in a simplistic form.

Only the *spectool_gtk* and *spectool_raw utilities* were used in the experiments conducted for this thesis.

The spectrum analyzer was used to implement a congestion avoidance algorithm that periodically retunes our test-bed's Wi-Fi interfaces to the quietest communication channel and is presented next.

3.5.2 The AutoMin Shell Script

A shell script was written for this thesis that implements the AutoMin algorithm to automatically switch the active communication channel in the NG-Mesh to the least congested one within the 2.4 GHz band. This script implements an OS-level process that constantly monitors the 2.4 GHz spectrum for average RSSI values in each Wi-Fi channel using the retail spectrum analyzer described earlier. Once it finds the channel with the minimum RSSI level, it tunes the Ralink radio to it. If the currently used Wi-Fi channel becomes shared with other devices, its RSSI will start to rise and the Automin algorithm will retune the Ralink radio to the next least congested Wi-Fi channel according to the spectrum analyzer.

The algorithm starts off with an initialization section that computes the spectral

data collection end time using the system clock. In the experiments, the data collection time-span was set to one minute arbitrarily. The Kismet *spectool_raw* executable was used to capture real-time spectral data across the 2.4 GHz band in textual form. Since each sample was reported 199 KHz apart starting from 2.4 GHz, the exact positions corresponding to the Wi-Fi channel center frequencies had to be pre-computed as shown in Table 3.2. The formula used to compute each of these positions is *Position=(Target Channel Frequency-2.4 GHz)/(199 KHz)+1*. The RSSI data for the eleven North American Wi-Fi channels is extracted from the spectrum analyzer output using these pre-computed positions and is used to calculate an average that is stored in a data file. A sample from the data file is shown in Figure 3.53. All RSSI data are in units of dBm. As more data is collected in real-time, the cumulative averages are updated immediately.

Ch.#	**Pos.**	**GHz**
1	61	2.412
2	86	2.417
3	112	2.422
4	137	2.427
5	162	2.432
6	187	2.437
7	212	2.442
8	237	2.447
9	262	2.452
10	287	2.457
11	313	2.462

Table 3.2: The positions of interest in the spectral data output.

```
-102 -104 -105 -103 -102 -103 -102 -100 -103 -103 -103
-103 -103 -105.5 -104 -102 -103 -102.5 -101 -104 -102 -101.5
-103.333 -103 -105.333 -104.333 -103 -103.333 -103.667 -102 -103.333 -103 -102.667
-103 -103 -105 -104.5 -103.5 -103.25 -104 -103 -103 -103.5 -102.5
-102.8 -103.4 -104.4 -104.6 -103.2 -103.2 -103.8 -102.6 -103 -103.6 -102.2
-103.167 -103.667 -104.333 -104.333 -103.167 -103.167 -103.5 -102.833 -103 -103.667 -101.5
-103.143 -103.857 -104.286 -104 -103.143 -103.286 -103.429 -102.714 -103.286 -103.857 -101.714
-103 -103.75 -104.25 -104 -103.125 -103.5 -103.375 -102.875 -103.5 -103.75 -101.75
-103 -103.667 -104.222 -103.778 -103.222 -103.556 -103.333 -103 -103.556 -103.778 -102
-103.1 -103.8 -104.1 -103.6 -103.2 -103.4 -103.3 -103 -100 -101.7 -102.2
-103.273 -103.909 -104.182 -103.455 -103.364 -103.455 -103.364 -103 -100.273 -101.909 -102.364
-103.417 -103.917 -104.167 -103.5 -103.25 -103.5 -103.25 -102.917 -100.667 -102.167 -102.5
```

Figure 3.53: Sample content in the RSSI data file.

The AutoMin script takes average RSSI data as input to perform congestion control in the test-bed periodically and is shown in Figure 3.54 to Figure 3.56.

Figure 3.54 shows that after initialization, spectral data collection is performed by the spectrum analyzer continuously except when a timer periodically expires. When data collection stops, the average RSSI data file is analyzed. The minimum average RSSI is identified along with the currently active communication channel.

In Figure 3.55, the difference between the previous minimum RSSI found (i.e., the currently active channel's noise floor) and the newly identified minimum RSSI is computed and stored in *deltaRSSI*. This value will later be used to determine if there is some other channel in the 2.4 GHz spectrum with a lower noise floor when compared with the currently active communication channel. Next, the channel corresponding to the newly found minimum RSSI is identified along with the average RSSI for the currently active communication channel. If the Wi-Fi channel was switched recently, the RSSI level of the currently active channel is updated to the most recent average RSSI level as reported by the spectrum analyzer. This is done so that the new communication channel's recently increased RSSI level (i.e., due to the channel now being occupied by the NG-Mesh's Wi-Fi interfaces) won't cause an

immediate channel switch to occur erroneously. The difference between the previously reported RSSI level and the one captured in the current data collection period for the communication channel in use is taken next and stored in *threshRSSI*. This value will later be used to predict if the active communication channel will become congested.

Figure 3.56 shows that a channel switch is performed only if the Wi-Fi channel in use is not the same as the one with the newly detected minimum RSSI and if either *deltaRSSI* or *threshRSSI* is greater than the value of the pre-set threshold. In our implementation, this threshold value was set to 5 dBm arbitrarily. This means that if there exists a channel that is different from the currently active one that also has an average RSSI level of more than 5 dBm lower than the noise floor of the current channel or if the current channel's average RSSI has risen by more than 5 dBm, which signifies the arrival of new users who are sharing this part of the spectrum, then a channel switch to the one with the most recently detected lowest average RSSI will occur. The final step in the algorithm is to reset the timer and restart the data collection process for the next iteration of the Automin algorithm.

```
#!/bin/bash

# Init
floorRSSI=999
prevRSSI=999
justSwitched=0
startTimer=`date +%M`
resetTimer=`expr $startTimer + 1`
/root/spectools-2010-04-R1/spectool_raw | grep -v device | grep -v MHz | sed 's/:/\n/g
' | grep -v USB | awk '{s61 += $61} {s86 += $86} {s112 += $112} {s137 += $137} {s162 +
= $162} {s187 += $187} {s212 += $212} {s237 += $237} {s262 += $262} {s287 += $287} {s3
13 += $313} {print s61/NR " " s86/NR " " s112/NR " " s137/NR " " s162/NR " " s187/NR "
 " s212/NR " " s237/NR " " s262/NR " " s287/NR " " s313/NR}' > data &

while [ 1 ]
do
        # Init
        curTime=`date +%M`
        ch=0
        threshRSSI=0

        if [ $curTime -eq $resetTimer ]
        then
                # stop collecting spectral data
                kill -9 `ps -ef | grep spectool_raw | grep -v grep | awk '{print $2}'`

                # Find min RSSI
                minRSSI=`tail -1 data | sed 's/ /\n/g' | sort -n | head -1`

                # Find current channel
                curCh=`iwconfig rausb0 | grep Freq | sed 's/=/ /g' | sed 's/\./ /g' |
awk '{print $4}'`
                for i in 412 417 422 427 432 437 442 447 452 457 462
                do
                        ch=`expr $ch + 1`
                        if [ $curCh -eq $i ]
                        then
                                curCh=$ch
                                echo "prevCh: $curCh"
                        fi
                done
                ch=0

```

Figure 3.54: The AutoMin shell script (1 of 3).

```
39                # Find difference between last min RSSI and current min RSSI
40                deltaRSSI=$(echo "($floorRSSI - $minRSSI)/1" | bc | sed 's/\./ /g' | a
   wk '{print $1}')
41
42                echo "minRSSI: $minRSSI"
43                echo "deltaRSSI: $deltaRSSI"
44
45                for i in `tail -1 data`
46                do
47                        # Find channel corresponding to min RSSI
48                        ch=`expr $ch + 1`
49                        if [ $minRSSI = $i ]
50                        then
51                                minCh=$ch
52                        fi
53
54                        # Find RSSI corresponding to current channel
55                        if [ $curCh -eq $ch ]
56                        then
57                                curRSSI=$i
58                        fi
59                done
60
61                echo "avg spectral data: `tail -1 data`"
62                echo "minCh: $minCh"
63                echo "RSSI of current channel: $curRSSI"
64
65                if [ $justSwitched -eq 1 ]
66                then
67                        prevRSSI=$curRSSI
68                        justSwitched=0
69                fi
70
71                echo "prevRSSI: $prevRSSI"
72
73                # Find difference between current RSSI and last RSSI on current channe
   l
74                tmpRSSI=$(echo "($prevRSSI)/1" | bc | sed 's/\./ /g' | awk '{print $1}
   ')
75                if [ $tmpRSSI -ne 999 ]
76                then
77                        threshRSSI=$(echo "($curRSSI - $prevRSSI)/1" | bc | sed 's/\./
    /g' | awk '{print $1}')
```

Figure 3.55: The AutoMin shell script (2 of 3).

```
78                  fi
79                  echo "threshRSSI: $threshRSSI"
80
81                  # switch channel if required
82                  if [ $curCh -ne $minCh -a \( $deltaRSSI -ge 5 -o $threshRSSI -ge 5 \)
   ]
83                  then
84                          floorRSSI=$minRSSI
85                          iwconfig rausb0 channel $minCh
86                          echo "newCh: $minCh"
87                          iwconfig rausb0
88                          justSwitched=1
89                  else
90                          echo "no channel switch required."
91                  fi
92                  echo "just switched to new channel? $justSwitched"
93                  echo "noise floor of current channel: $floorRSSI"
94
95                  # restart spectral data collection for the next session
96                  resetTimer=`expr $curTime + 1`
97                  /root/spectools-2010-04-R1/spectool_raw | grep -v device | grep -v MHz
    | sed 's/:/\n/g' | grep -v USB | awk '{s61 += $61} {s86 += $86} {s112 += $112} {s137
   += $137} {s162 += $162} {s187 += $187} {s212 += $212} {s237 += $237} {s262 += $262} {s
   287 += $287} {s313 += $313} {print s61/NR " " s86/NR " " s112/NR " " s137/NR " " s162/
   NR " " s187/NR " " s212/NR " " s237/NR " " s262/NR " " s287/NR " " s313/NR}' > data &
98          fi
99  done
```

Figure 3.56: The AutoMin shell script (3 of 3).

3.6 Test-bed Summary

The NG-Mesh is composed of the following components.

- The Ralink rt2570 chip-set.
- A centralized PC with multiple virtualized Linux OSs.
- Configuration scripts that were written for this thesis.
- The SerialMonkey device driver source code.
- TCP and UDP socket code that was developed for this thesis.

- Metrics collection scripts that were written for this thesis.
- An entry-level spectrum analyzer.
- The AutoMin shell script implementing a congestion avoidance algorithm.

Our test-bed setup enables physical layer measurements such as RSSI and SINR statistics to be readily captured during file transmission between the network nodes on each of the eleven North American Wi-Fi channels. Testing results are reported next.

Chapter 4

Experimental Results

This chapter describes the tests that were carried out with the NG-Mesh test-bed and their experimental results. All experiments took place in a room measuring sixteen square meters within a residential neighborhood near McMaster University.

4.1 Overview: The Basic Unit of NG-Mesh

The tests conducted for this thesis involved the recording of several metrics during successive UDP file transfers of a 100 MB file between a pair of client and server network interfaces on each of the eleven North American Wi-fi channels. The goal was to determine the effect of radio activity occurring at particular center frequencies on the rest of the 2.4 GHz spectrum. Background noise level was measured using the readings from one of the rt2570 interfaces operating in monitor mode and a commercial spectrum analyzer. The average interference and noise level at the test site was found to be -99 dBm throughout the 2.4 GHz band. Since this reading is near the lower sensitivity limit of the transceivers used to detect it, we believe that

the actual value may be even lower (i.e., the hardware used is not sensitive enough to detect the true noise floor). The test site had a local AP operating on Wi-Fi channel 1 that was unplugged during later experiments.

Sensitive timing functions such as the handling of retransmission timeouts are stored in the inaccessible firmware of the Ralink interfaces. Thus the exact implementation of the rt2570's multi-rate retry, back-off, and rate control algorithms are unknown. It is for this reason that the TCP protocol was not used in our file transfer tests.

Figure 4.1 shows the general network setup used during our tests. The communication link was established through pairing the client node's transmitter with the receiver of the server node while a monitor node, which was either a software-based or retail spectrum analyzer, was configured to capture statistics.

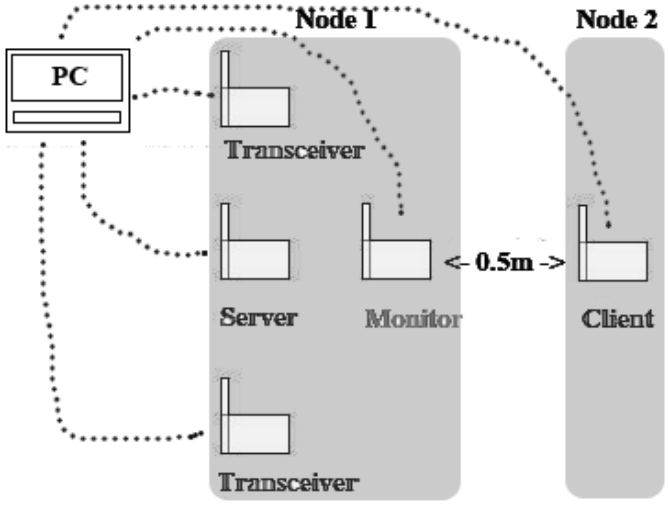


Figure 4.1: NG-Mesh client to server communication monitored with a spectrum analyzer.

4.2 The 2.4 GHz Band Prior to File Transmission: Establishing the Baseline

RSSI readings throughout the 2.4 GHz band were measured to determine the baseline levels present at the test site in the absence of local transmission activities. All file transfer tests were performed after establishing this baseline during a training period. The retail spectral analyzer that was introduced in Chapter 3 was used to verify the interference and noise floor of -99 dBm. The Chanalyzer Lite software for Microsoft Windows was used to generate baseline graphs to illustrate the RSSI levels detected across the spectrum.

In Figure 4.2, the density view of the spectrum baseline RSSI levels is shown. This graph represents one hour of RSSI data collected by the retail spectrum analyzer prior to any file transmission at the test site. This view indicates the peak, average, and instantaneous RSSI levels that are updated several times per second. The bottom of the plot depicts the most frequently occurring RSSI readings in the spectrum and represents the average RSSI levels detected at the test site. Nearby networks are indicated by captions with the one emitting the strongest signal operating on Wi-Fi channel 1 followed by weaker signals detected from more remote networks operating on channels 6 and 11. The existence of nearby APs operating on channels 1, 6, and 11 is confirmed in Figure 4.3. At the time of capture, there were six networks within radio range of the test site. Their RSSI readings are updated once every three seconds. The remote APs residing on channels 6 and 11 exhibited negligible RSSI levels. The average interference and noise floor was computed to be around -99 dBm across the 2.4 GHz spectrum. This suggests that the network activity on all nearby APs was

quite infrequent at the time of testing. The strongest signal came from the test site's own *Link1* AP operating on Wi-Fi channel 1. This AP was later unplugged to avoid potential interference during file transmission tests.

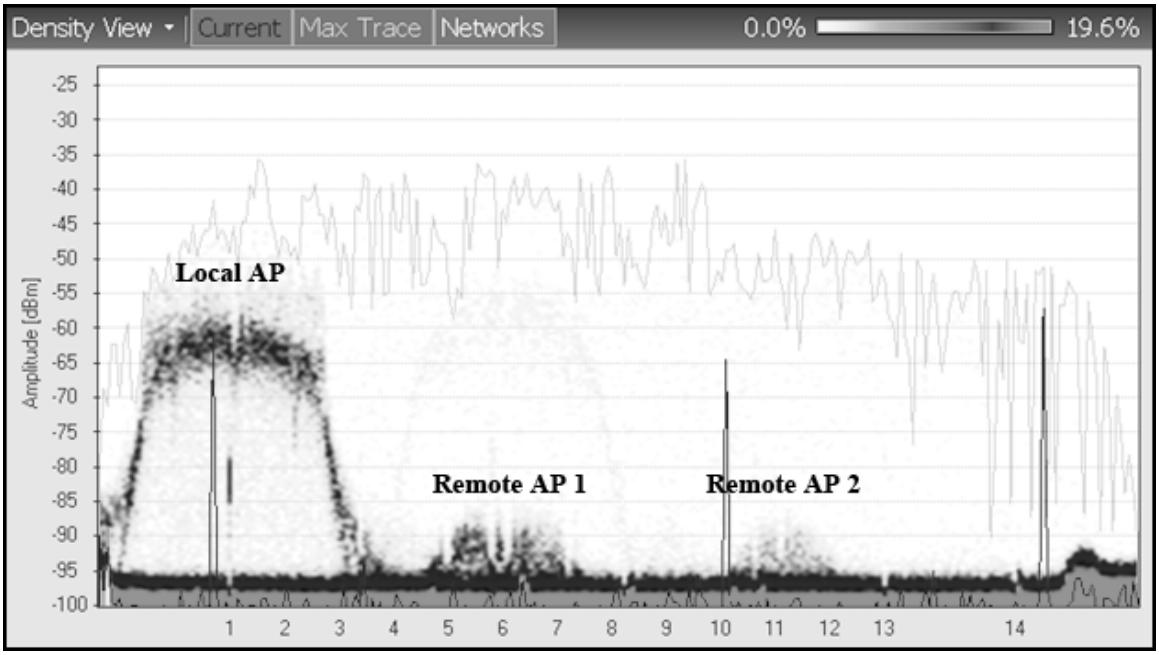


Figure 4.2: The baseline RSSI level in the 2.4 GHz spectrum (density view).

Wi-Fi Networks

Graph	SSID	Channe	RSSI
☐	Link1	1	-50
☐		6	-92
☐	frasers-network	11	-98
☐	frasers-network	11	-97
☐	Isma'ilis	6	-84
☐	BELL904	1	-92

Figure 4.3: Nearby Wi-Fi networks at the test site.

The 2D waterfall view of Figure 4.4 and its 3D counterpart shown in Figure 4.5 conveys the intensity of the RSSI levels in the Wi-Fi channels captured over time.

The plots slowly shift upwards as they are updated with newer readings that are reported by the spectrum analyzer. The plots show the strong -50 dBm signal of the local AP occupying the center frequency of Wi-Fi channel 1, followed by the weaker signals of the remote APs at channels 6 and 11.

The current, average, and peak RSSI levels are shown in the bar-chart view of Figure 4.6. Figure 4.7 contains the corresponding data used by the spectrum analyzer software to produce these bar-charts. This data was captured after powering off the local AP that was operating on Wi-Fi channel 1. The bar-charts show that the overall average RSSI levels in the 2.4 GHz band at the test site was approximately -99 dBm even though there were occasional occurrences of higher RSSI readings centering on channels 1, 6, and 11 that were caused by distant APs.

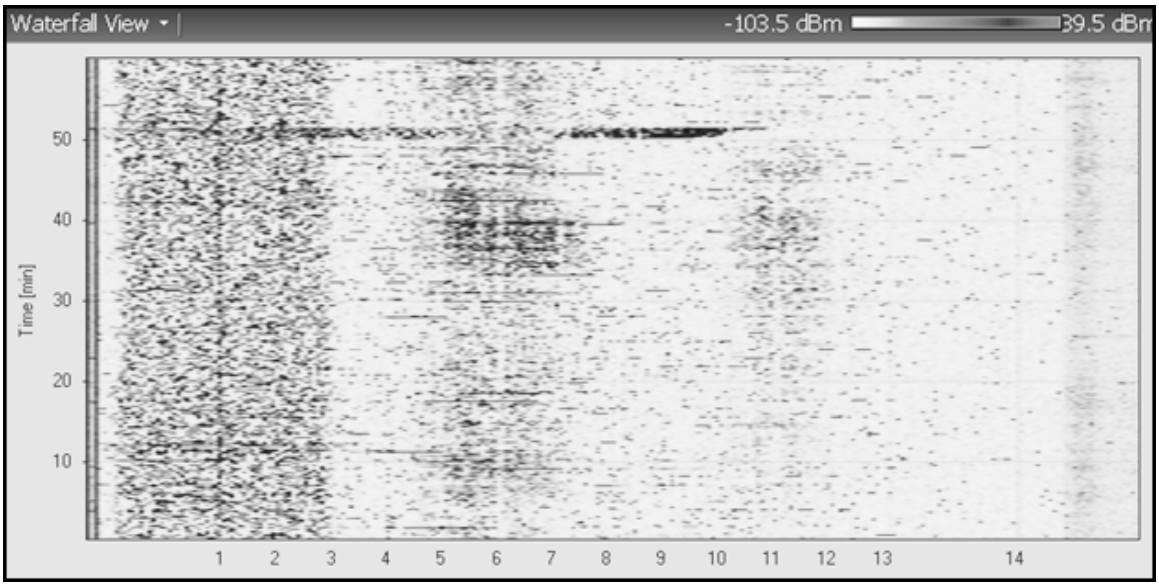


Figure 4.4: The baseline RSSI level in the 2.4 GHz spectrum (waterfall view).

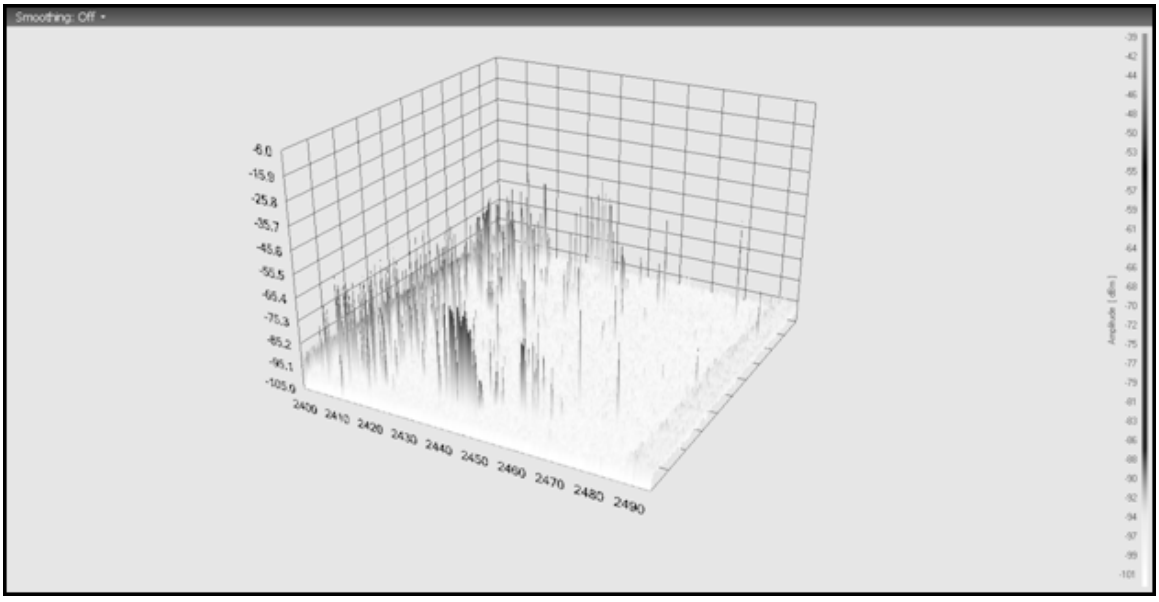


Figure 4.5: The baseline RSSI level in the 2.4 GHz spectrum (3D view).

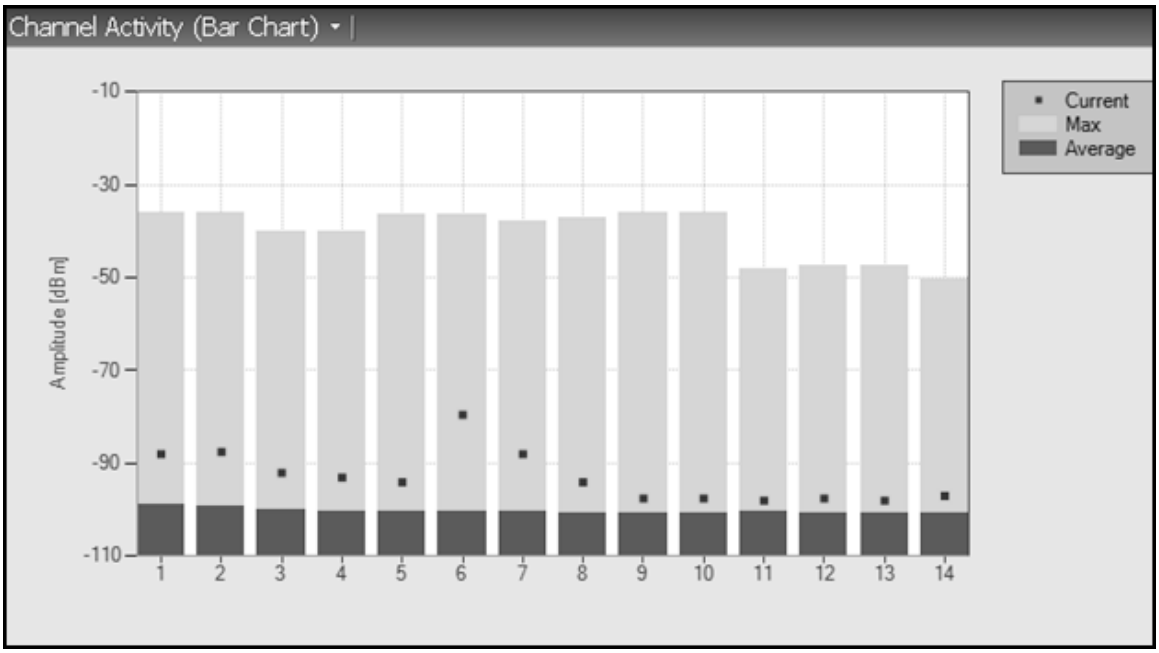
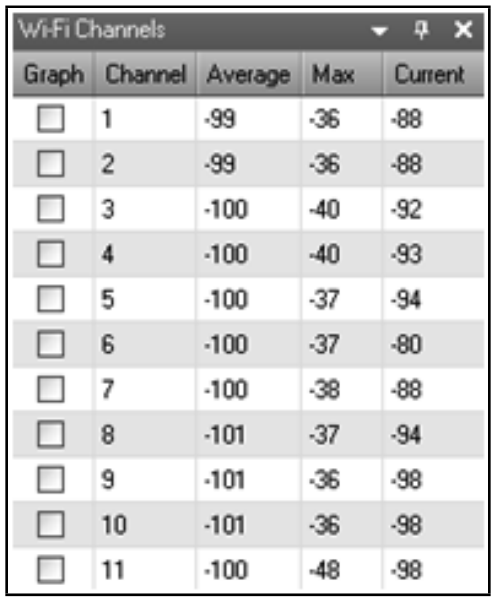


Figure 4.6: The baseline RSSI level in the 2.4 GHz spectrum (bar-chart view).



Graph	Channel	Average	Max	Current
☐	1	-99	-36	-88
☐	2	-99	-36	-88
☐	3	-100	-40	-92
☐	4	-100	-40	-93
☐	5	-100	-37	-94
☐	6	-100	-37	-80
☐	7	-100	-38	-88
☐	8	-101	-37	-94
☐	9	-101	-36	-98
☐	10	-101	-36	-98
☐	11	-100	-48	-98

Figure 4.7: The baseline RSSI level in the 2.4 GHz spectrum (Wi-Fi channels view).

4.3 The 2.4 GHz Band During File Transfer: Capturing the Spectral Change

Client to server file transfer tests were performed while the following Wi-Fi statistics were captured.

- **RSSI:** Indicates the signal strength in dBm that was received at the monitor node's listening interface.
- **RX PER:** The average percentage of reception error that was experienced at the server's listening interface.
- **TX PRR:** The average percentage of TCP retransmissions that has occurred at the client's transmitting interface.

- **CQI:** The channel quality computed as a percentage.
- **Duration:** The elapsed time of file transfer that was approximated to the nearest second at the server endpoint.
- **SINR:** The relative signal strength in dBm that was observed between the average RSSI level and the interference and noise level of -99 dBm during file transmission at the monitor node. The formula used to calculate this metric was $SINR_{dBm} = Signal_{dBm} - (Noise + Interference)_{dBm}$ and was derived from [40].
- **Bit-Rate:** The effective bit-rate in Mbps that was calculated by dividing the file size by the duration of file transfer observed at the server endpoint.

The transmit power of both the client and server Wi-Fi interfaces were fixed at 15 dBm with their bit-rates set to 11 Mbps (i.e., the maximum rate under IEEE 802.11b). During file transfer on each Wi-Fi channel, the monitor node sampled the RSSI levels on all Wi-Fi channels to capture the magnitude of impact of the transmission activity on the entire 2.4 GHz spectrum. The monitor node was the software-based spectrum analyzer that was introduced in Chapter 3. Recall that developing this device involved changing the driver source code of an rt2570 Wi-Fi interface to cycle through the eleven North American Wi-Fi channels within the 2.4 GHz band to display the RSSI register values as the radio tuned to each of the channels. The receiver stayed synchronized to each frequency for five seconds while capturing per-second RSSI readings before switching to the next channel. This process of data collection resulted in an average RSSI value that was computed for each channel. These per-second readings represented semi real-time values. When

using this method of metrics collection, the RSSI values collected occasionally did not update immediately upon re-tuning the transmitter to a new Wi-Fi channel. The results reported by the software-based spectrum analyzer will later be compared with the ones reported by the commercial spectrum analyzer.

Figure 4.8 demonstrates some sample output from the *iwconfig* Linux utility as the software-based spectrum analyzer interface detects the signal (i.e., RSSI) level for Wi-Fi channel 1 (i.e., 2.412 GHz). The interference and noise level was also captured as the modified NIC driver cycles through each Wi-Fi channel. The ad-hoc cell ID is set to *test1* in our tests.

```
rausb0    RT2500USB WLAN  ESSID:"test1"
          Mode:Ad-Hoc  Frequency=2.412 GHz  Cell: EA:3F:A5:6E:89:28
          Bit Rate=11 Mb/s
          RTS thr:off   Fragment thr:off
          Encryption key:off
          Link Quality=88/100  Signal level:-43 dBm  Noise level:-92 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0
```

Figure 4.8: Sample output from the iwconfig Linux utility.

Table 4.1 to Table 4.11 presents the file transmission metrics collected for each of the eleven North American Wi-Fi channels. The boldfaced column in each table represents the Wi-Fi channel used for file transfer between the client and server interfaces. The metrics captured in the remaining columns represent the effect of the file transfer on the rest of the 2.4 GHz spectrum.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	**-7**	-5	-8	-9	-36	-38	-39	-42	-38	-43	-44
RX PER (%)	**1**	-	-	-	-	-	-	-	-	-	-
TX PRR (%)	**0**	-	-	-	-	-	-	-	-	-	-
CQI (%)	**106**	-	-	-	-	-	-	-	-	-	-
Duration (sec)	**195**	-	-	-	-	-	-	-	-	-	-
SINR (dBm)	**92**	94	91	90	63	61	60	57	61	56	55
Bit-Rate (Mbps)	**4**	-	-	-	-	-	-	-	-	-	-

Table 4.1: The effect of file transfer occurring on Wi-Fi channel 1 as reported by the software-based spectrum analyzer.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	-42	**-8**	-7	-10	-15	-36	-38	-41	-42	-38	-40
RX PER (%)	-	**3**	-	-	-	-	-	-	-	-	-
TX PRR (%)	-	**0**	-	-	-	-	-	-	-	-	-
CQI (%)	-	**105**	-	-	-	-	-	-	-	-	-
Duration (sec)	-	**198**	-	-	-	-	-	-	-	-	-
SINR (dBm)	57	**91**	92	89	84	63	61	58	57	61	59
Bit-Rate (Mbps)	-	**4**	-	-	-	-	-	-	-	-	-

Table 4.2: The effect of file transfer occurring on Wi-Fi channel 2 as reported by the software-based spectrum analyzer.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	-42	-38	**-7**	-7	-8	-9	-38	-40	-42	-43	-41
RX PER (%)	-	-	**6**	-	-	-	-	-	-	-	-
TX PRR (%)	-	-	**0**	-	-	-	-	-	-	-	-
CQI (%)	-	-	**104**	-	-	-	-	-	-	-	-
Duration (sec)	-	-	**199**	-	-	-	-	-	-	-	-
SINR (dBm)	57	61	**92**	92	91	90	61	59	57	56	58
Bit-Rate (Mbps)	-	-	**4**	-	-	-	-	-	-	-	-

Table 4.3: The effect of file transfer occurring on Wi-Fi channel 3 as reported by the software-based spectrum analyzer.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	-42	-38	-25	**-6**	-7	-8	-16	-36	-40	-43	-45
RX PER (%)	-	-	-	**1**	-	-	-	-	-	-	-
TX PRR (%)	-	-	-	**0**	-	-	-	-	-	-	-
CQI (%)	-	-	-	**106**	-	-	-	-	-	-	-
Duration (sec)	-	-	-	**198**	-	-	-	-	-	-	-
SINR (dBm)	57	61	74	**93**	92	91	83	63	59	56	54
Bit-Rate (Mbps)	-	-	-	**4**	-	-	-	-	-	-	-

Table 4.4: The effect of file transfer occurring on Wi-Fi channel 4 as reported by the software-based spectrum analyzer.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	-37	-38	-38	-7	**-5**	-17	-18	-19	-45	-48	-55
RX PER (%)	-	-	-	-	**1**	-	-	-	-	-	-
TX PRR (%)	-	-	-	-	**0**	-	-	-	-	-	-
CQI (%)	-	-	-	-	**106**	-	-	-	-	-	-
Duration (sec)	-	-	-	-	**199**	-	-	-	-	-	-
SINR (dBm)	62	62	61	92	**94**	82	81	81	54	51	44
Bit-Rate (Mbps)	-	-	-	-	**4**	-	-	-	-	-	-

Table 4.5: The effect of file transfer occurring on Wi-Fi channel 5 as reported by the software-based spectrum analyzer.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	-40	-36	-38	-39	-11	**-6**	-18	-20	-20	-46	-48
RX PER (%)	-	-	-	-	-	**0**	-	-	-	-	-
TX PRR (%)	-	-	-	-	-	**0**	-	-	-	-	-
CQI (%)	-	-	-	-	-	**106**	-	-	-	-	-
Duration (sec)	-	-	-	-	-	**209**	-	-	-	-	-
SINR (dBm)	59	63	61	60	88	**93**	81	79	79	53	51
Bit-Rate (Mbps)	-	-	-	-	-	**4**	-	-	-	-	-

Table 4.6: The effect of file transfer occurring on Wi-Fi channel 6 as reported by the software-based spectrum analyzer.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	-43	-40	-36	-39	-39	-7	**-5**	-18	-21	-21	-47
RX PER (%)	-	-	-	-	-	-	**0**	-	-	-	-
TX PRR (%)	-	-	-	-	-	-	**0**	-	-	-	-
CQI (%)	-	-	-	-	-	-	**106**	-	-	-	-
Duration (sec)	-	-	-	-	-	-	**200**	-	-	-	-
SINR (dBm)	56	59	63	60	60	92	**94**	81	78	78	52
Bit-Rate (Mbps)	-	-	-	-	-	-	**4**	-	-	-	-

Table 4.7: The effect of file transfer occurring on Wi-Fi channel 7 as reported by the software-based spectrum analyzer.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	-22	-43	-40	-36	-37	-38	-7	**-5**	-17	-19	-21
RX PER (%)	-	-	-	-	-	-	-	**0**	-	-	-
TX PRR (%)	-	-	-	-	-	-	-	**0**	-	-	-
CQI (%)	-	-	-	-	-	-	-	**106**	-	-	-
Duration (sec)	-	-	-	-	-	-	-	**201**	-	-	-
SINR (dBm)	77	56	59	63	62	61	92	**94**	82	80	78
Bit-Rate (Mbps)	-	-	-	-	-	-	-	**4**	-	-	-

Table 4.8: The effect of file transfer occurring on Wi-Fi channel 8 as reported by the software-based spectrum analyzer.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	-47	-46	-43	-41	-36	-37	-40	-8	**-5**	-15	-19
RX PER (%)	-	-	-	-	-	-	-	-	**0**	-	-
TX PRR (%)	-	-	-	-	-	-	-	-	**0**	-	-
CQI (%)	-	-	-	-	-	-	-	-	**106**	-	-
Duration (sec)	-	-	-	-	-	-	-	-	**202**	-	-
SINR (dBm)	52	53	56	58	63	62	59	91	**94**	84	80
Bit-Rate (Mbps)	-	-	-	-	-	-	-	-	**4**	-	-

Table 4.9: The effect of file transfer occurring on Wi-Fi channel 9 as reported by the software-based spectrum analyzer.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	-47	-49	-46	-44	-41	-37	-38	-39	-8	**-5**	-14
RX PER (%)	-	-	-	-	-	-	-	-	-	**0**	-
TX PRR (%)	-	-	-	-	-	-	-	-	-	**0**	-
CQI (%)	-	-	-	-	-	-	-	-	-	**106**	-
Duration (sec)	-	-	-	-	-	-	-	-	-	**211**	-
SINR (dBm)	52	50	53	55	58	62	61	60	92	**94**	85
Bit-Rate (Mbps)	-	-	-	-	-	-	-	-	-	**4**	-

Table 4.10: The effect of file transfer occurring on Wi-Fi channel 10 as reported by the software-based spectrum analyzer.

Metric \ Channel	1	2	3	4	5	6	7	8	9	10	11
RSSI (dBm)	-53	-51	-49	-46	-44	-41	-37	-39	-39	-6	**-5**
RX PER (%)	-	-	-	-	-	-	-	-	-	-	**0**
TX PRR (%)	-	-	-	-	-	-	-	-	-	-	**0**
CQI (%)	-	-	-	-	-	-	-	-	-	-	**106**
Duration (sec)	-	-	-	-	-	-	-	-	-	-	**193**
SINR (dBm)	46	48	50	53	55	58	62	60	60	93	**94**
Bit-Rate (Mbps)	-	-	-	-	-	-	-	-	-	-	**4**

Table 4.11: The effect of file transfer occurring on Wi-Fi channel 11 as reported by the software-based spectrum analyzer.

Table 4.12 and Table 4.13 presents the RSSI and SINR metrics that were captured by the retail spectrum analyzer during the file transmission tests on each of the Wi-Fi channels. The per-channel average RSSI data was captured using the metrics collection shell scripts that were introduced in Chapter 3. The values in the SINR matrix were computed from the RSSI readings. The boldfaced main diagonal in the tables represent the center channels of communication between the client and server interfaces while the statistics in the rest of the tables indicate the level of interference

that was experienced as a result of the file transfer tests.

Although a PHY rate of 11 Mbps was configured at both the client and server interfaces, the actual bit-rate that was observed during the file transfer tests was calculated to be 3.96 Mbps on average. This is only 36.01% of the configured speed. Since the best-effort UDP protocol was used in this experiment, no retransmission mechanisms were in place and thus collision-based back-off algorithms were not a contributing factor to the drop in throughput. The cause of the discrepancy is believed to stem from MAC and application layer overhead during file transfer.

Client \ Server	1	2	3	4	5	6	7	8	9	10	11
1	**-52**	-57	-73	-68	-80	-74	-83	-85	-88	-91	-89
2	-56	**-51**	-54	-69	-68	-76	-74	-79	-84	-87	-91
3	-64	-57	**-50**	-56	-71	-68	-79	-75	-82	-85	-89
4	-59	-73	-56	**-48**	-56	-70	-69	-79	-75	-84	-86
5	-71	-68	-71	-56	**-49**	-57	-74	-67	-78	-75	-82
6	-69	-78	-68	-70	-56	**-51**	-57	-74	-69	-80	-75
7	-75	-74	-77	-66	-70	-54	**-52**	-57	-72	-68	-81
8	-75	-82	-75	-78	-68	-69	-56	**-48**	-60	-73	-72
9	-79	-82	-81	-74	-78	-67	-69	-59	**-52**	-59	-72
10	-87	-85	-83	-82	-75	-73	-69	-72	-63	**-49**	-60
11	-84	-89	-86	-82	-81	-74	-78	-68	-72	-59	**-50**

Table 4.12: The effect of file transfer occurring on each Wi-Fi channel on spectrum-wide RSSI level as reported by the commercial spectrum analyzer.

Client \ Server	1	2	3	4	5	6	7	8	9	10	11
1	**47**	42	26	31	19	25	16	14	11	8	10
2	43	**48**	45	30	31	23	25	20	15	12	8
3	35	42	**49**	43	28	31	20	24	17	14	10
4	40	26	43	**51**	43	29	30	20	24	15	13
5	28	31	28	43	**50**	42	25	32	21	24	17
6	30	21	31	29	43	**48**	42	25	30	19	24
7	24	25	22	33	29	45	**47**	42	27	31	18
8	24	17	24	21	31	30	43	**51**	39	26	27
9	20	17	18	25	21	32	30	40	**47**	40	27
10	12	14	16	17	24	26	30	27	36	**50**	39
11	15	10	13	17	18	25	21	31	27	40	**49**

Table 4.13: The effect of file transfer occurring on each Wi-Fi channel on spectrum-wide SINR level as reported by the commercial spectrum analyzer.

The analysis of the experimental results are discussed next.

4.4 Analyzing the Experimental Results

The effects of per-channel file transfers on the 2.4 GHz band are presented in this section. First the results as reported by the software-based spectrum analyzer are presented to show the maximum extent of co-channel interference between IEEE 802.11b signals residing on non-orthogonal channels. Next, the results as reported by the retail spectrum analyzer are shown in order to contrast the differences in accuracy and responsiveness between the two devices when reporting current RSSI activity across the spectrum.

4.4.1 The Effect of File Transfer on RSSI According to the Software-Based Spectrum Analyzer

Figure 4.9 to Figure 4.12 illustrates the effect of the successive file transfer tests occurring on each of the eleven North American Wi-Fi channels in the 2.4 GHz spectrum. The RSSI data used for these plots were captured with the software-based spectrum analyzer and were presented previously. In the graphs, each curve represents a file transfer conducted at a particular radio frequency with the shaded regions emphasizing adjacent channel interference. As expected, the peak RSSI level always occurs at the channel of file transfer. Moving two channels away (i.e., 11 MHz) to the left of the channel of communication leads to a signal attenuation of about 35 dBm, which is compliant with the IEEE 802.11b spectral mask requirement. However, moving in the opposite direction results in an attenuation of only about 5 to 15 dBm, which violates the IEEE 802.11b standard and distorts the expected shape of the spectral mask. This violation is believed to be caused by the modified Ralink driver not being able to update the rt2570 chip-set registers fast enough in the software-based spectrum analyzer as the transceiver of the network interface is tuned to incrementally higher frequencies. Thus the software-based spectrum analyzer is not accurate enough to show the correct shape of the IEEE 802.11b spectral mask.

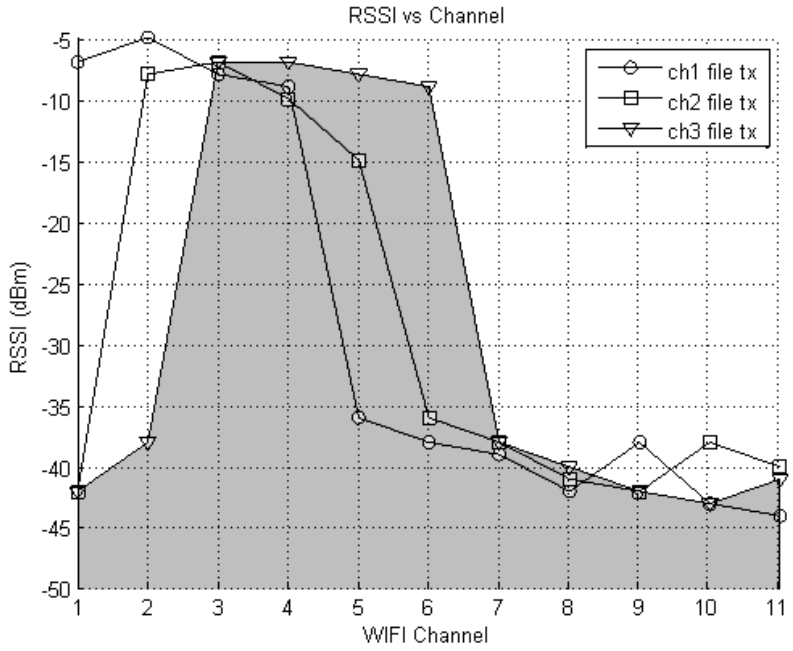


Figure 4.9: RSSI distribution due to file transmission (1 of 4).

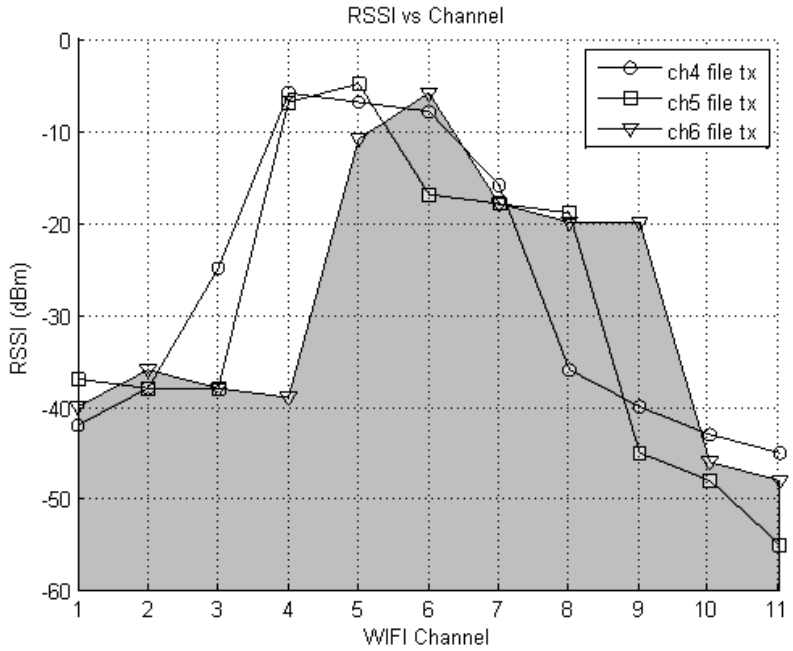


Figure 4.10: RSSI distribution due to file transmission (2 of 4).

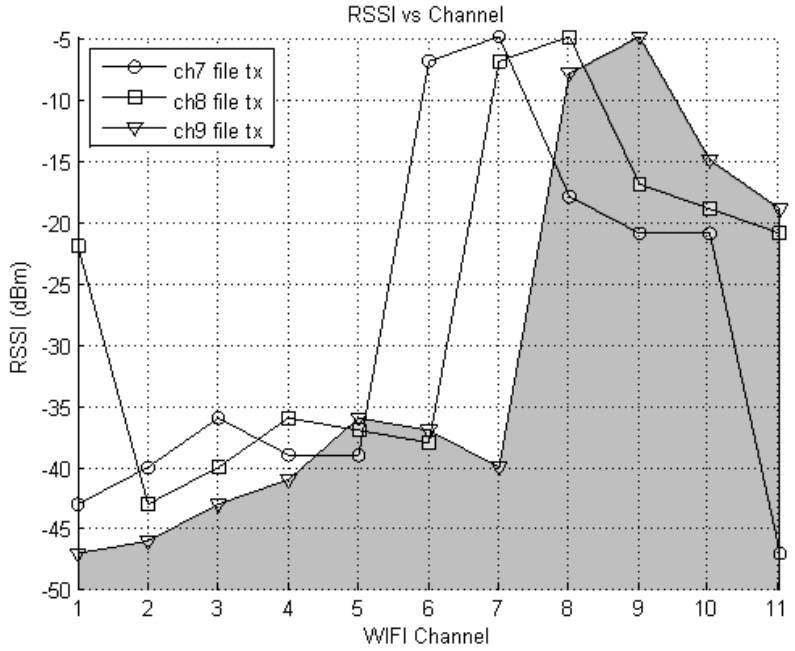


Figure 4.11: RSSI distribution due to file transmission (3 of 4).

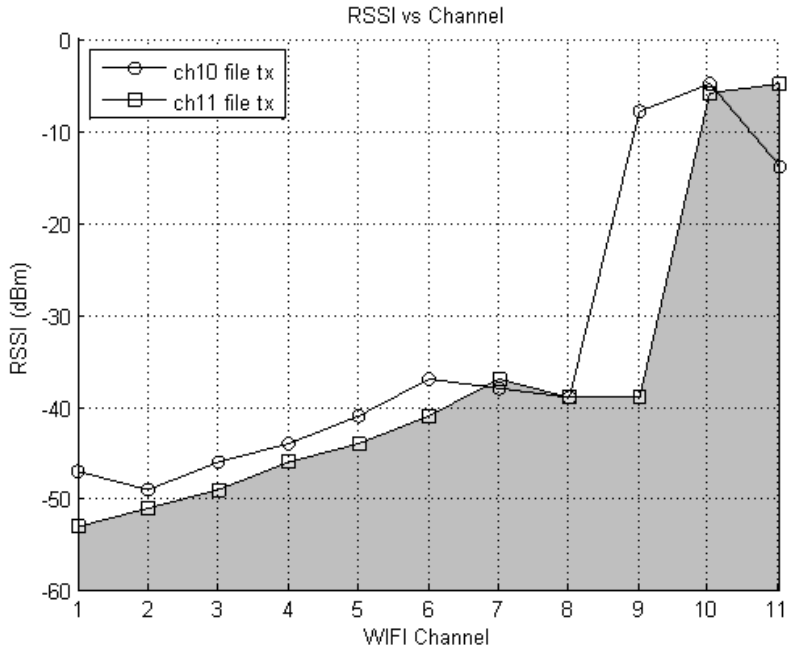


Figure 4.12: RSSI distribution due to file transmission (4 of 4).

The rt2570 chip-sets were found to be compliant with the IEEE 802.11b spectral mask requirement as reported next by the commercial spectrum analyzer.

4.4.2 The Effect of File Transfer on RSSI According to the Retail Spectrum Analyzer

Figure 4.13 shows a file transfer occurring on Wi-Fi channel 3 and was produced by the retail spectrum analyzer software. The *Max* and *Average* plots are not representative of the RSSI levels strictly during the file transmission as the spectrum activity that occurred prior to the tests is factor in. The constant activity of the dynamic *Current* plot is updated in real-time during file transmission and accurately shows the shape of the IEEE 802.11b spectral mask. At the channel of activity (i.e., channel 3), a RSSI level of about -40 dBm is observed. Moving two channels away on either side results in a drop to around -65 dBm to -70 dBm. Moving yet another 11 MHz away from this point results in a further drop to about -80 dBm. Although these results are not exactly compliant with the IEEE 802.11b spectral mask requirement, some additional trials of this experiment did not produce any such violations. Hence the discrepancies seen in the plot are believed to be due to intermittent errors produced by the inexpensive consumer-grade rt2570 chip-sets or the entry-level spectrum analyzer. An alternate explanation could plausibly be intermittent network activity at the nearby APs residing on Wi-Fi channel 6 during the file transfer test.

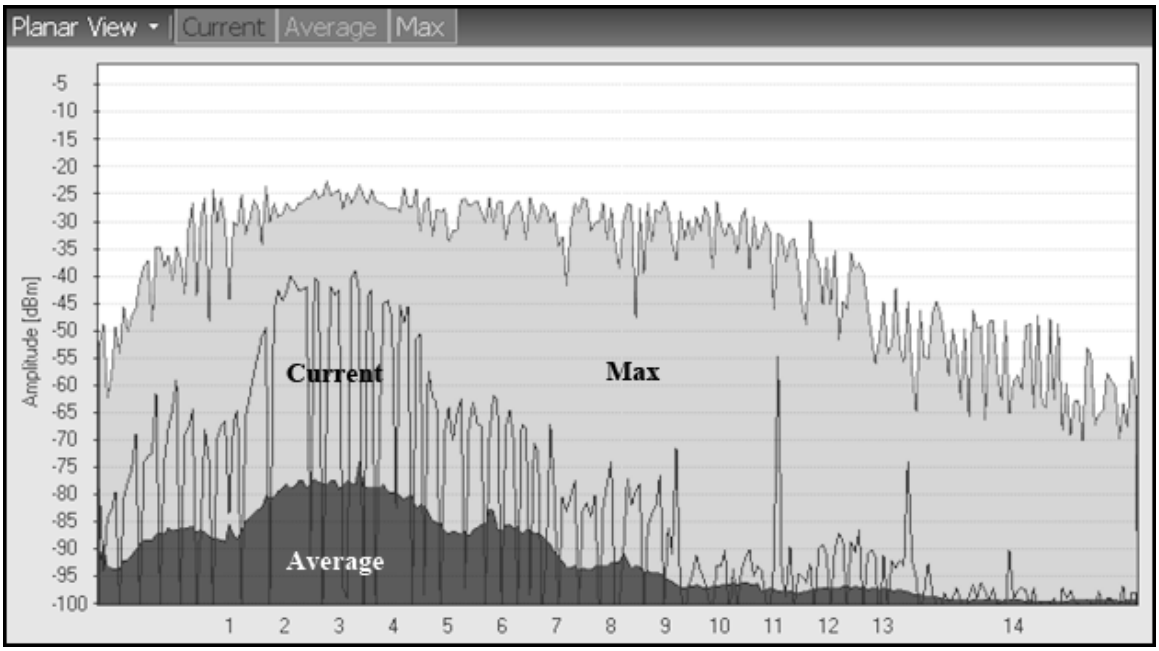


Figure 4.13: RSSI distribution due to 2.422 GHz file transmission (planar view).

A side-by-side comparison of the RSSI levels reported for a file transfer occurring on Wi-Fi channel 6 by the software-based and retail spectrum analyzers is shown in Figure 4.14. The software-based spectrum analyzer reports significantly higher RSSI values than the retail version and a more distorted shape of the IEEE 802.11b spectral mask for the same file transfer test. The first discrepancy could be attributed to differences between their receiver sensitivities and the way in which the RSSI levels are measured, computed, or reported. Thus the software-based implementation is only suitable for accurately reporting the center frequencies of file transfer.

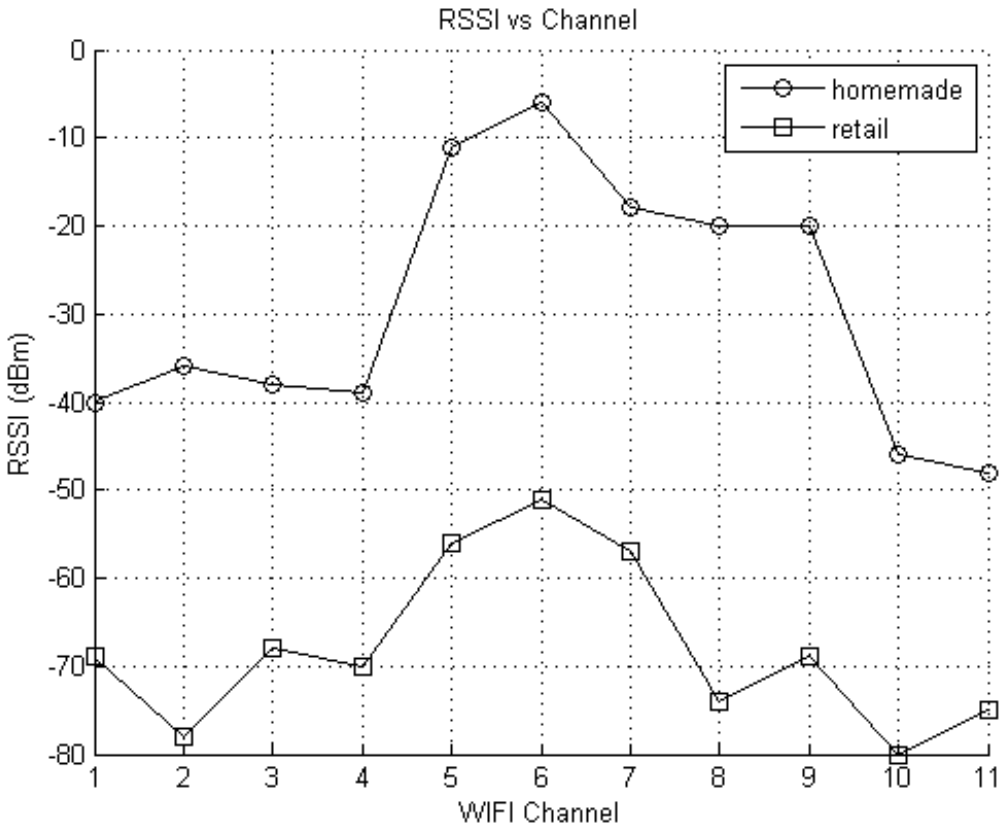


Figure 4.14: A comparison of the RSSI distributions detected by the software-based and retail spectrum analyzers.

Figure 4.15 provides an accurate representation of the average RSSI values in the 2.4 GHz band strictly during file transmission. This plot shows the frequency of RSSI levels in the spectrum over time. The solid regions represent the most frequently occurring RSSI readings during a channel 3 file transfer test. At the peak, the most frequent RSSI level is around -40 dBm. The most frequent reading at 2 and 4 channels away is around -70 dBm and -87 dBm respectively, which approximately satisfies the IEEE 802.11b spectral mask requirement. The less frequent *Beacons* scatter plot with slightly higher RSSI levels is a consequence of placing the retail spectrum analyzer

right next to the server Wi-Fi interface, which periodically emitted IEEE 802.11b transmissions within the ad-hoc cell.

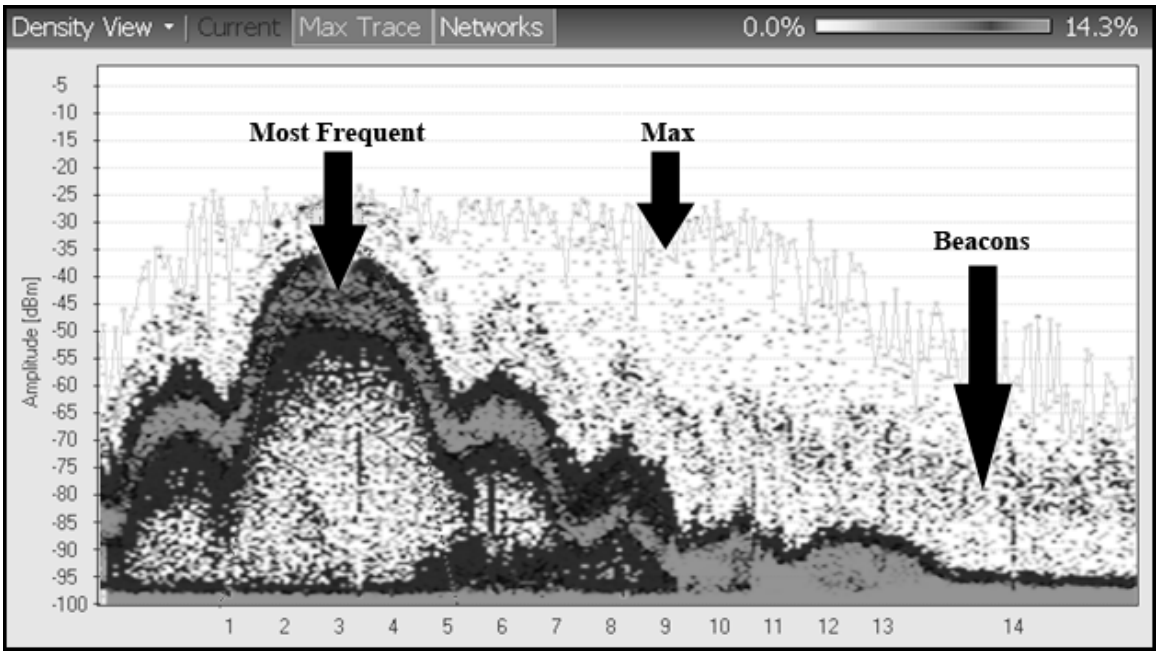


Figure 4.15: RSSI distribution due to 2.422 GHz file transmission (density view).

Figure 4.16 shows a 3D RSSI plot of a channel 3 file transfer test. This view shows the 3D representation of the IEEE 802.11b spectral mask.

Figure 4.17 captures the intensity of a partial channel 3 file transfer across the 2.4 GHz spectrum. As expected, the highest RSSI level is detected at the active communication channel while progressively lower readings were detected as displacement from this point increases on either side. The RSSI readings instantaneously increase upon file transmission and abruptly decrease as soon as transmission stops.

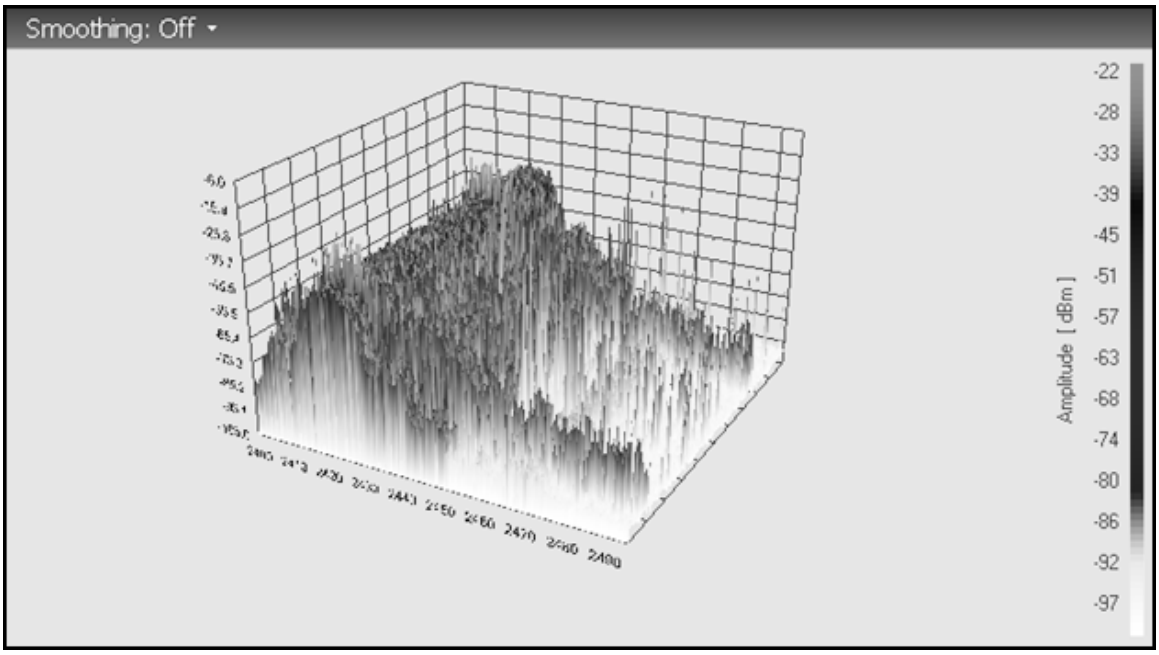


Figure 4.16: RSSI distribution due to 2.422 GHz file transmission (3D view).

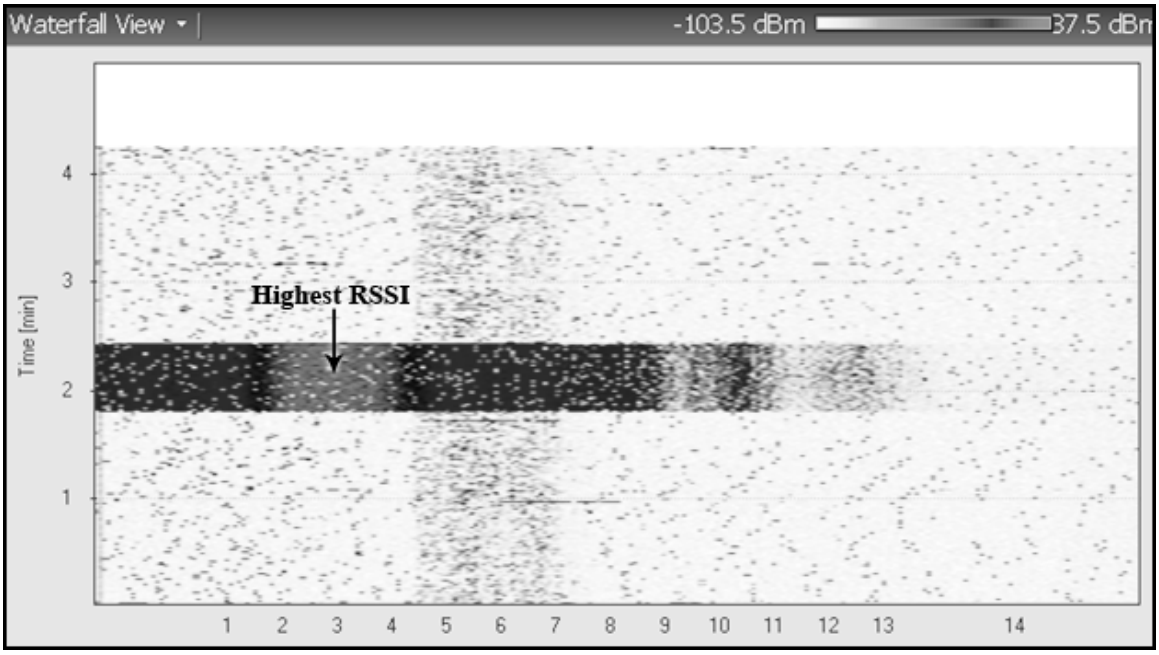


Figure 4.17: RSSI distribution due to 2.422 GHz file transmission (waterfall view).

The SINR metric presents the captured RSSI data in an alternate form. This data is discussed next.

4.4.3 The Effect of File Transfer on SINR According to the Software-Based Spectrum Analyzer

Figure 4.18 to Figure 4.21 graphs the software-based spectrum analyzer SINR data that was presented earlier. The plots have the same patterns as their RSSI counterparts and emphasize the overlapping regions that indicate neighboring channel interference, which is a result of the file transfer tests occurring on each Wi-Fi channel.

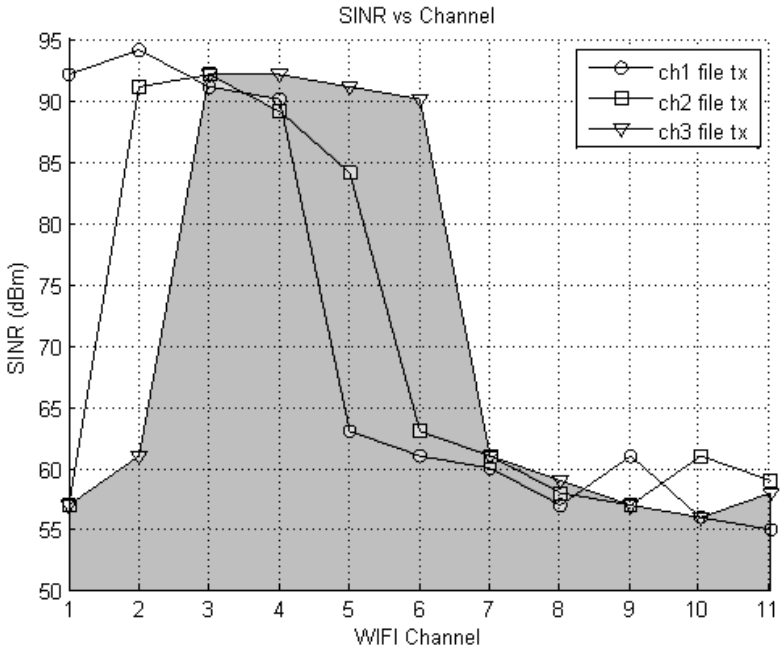


Figure 4.18: SINR distribution due to file transmission (1 of 4).

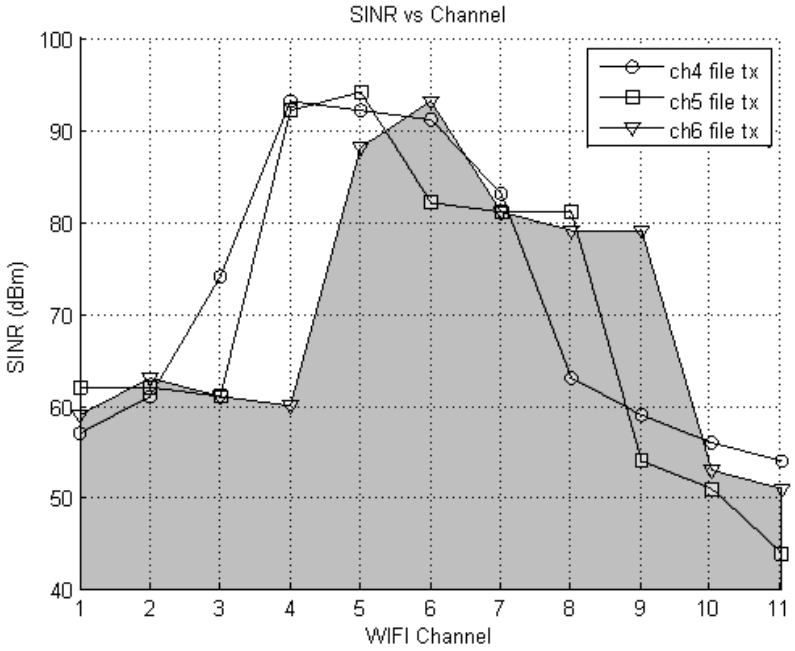


Figure 4.19: SINR distribution due to file transmission (2 of 4).

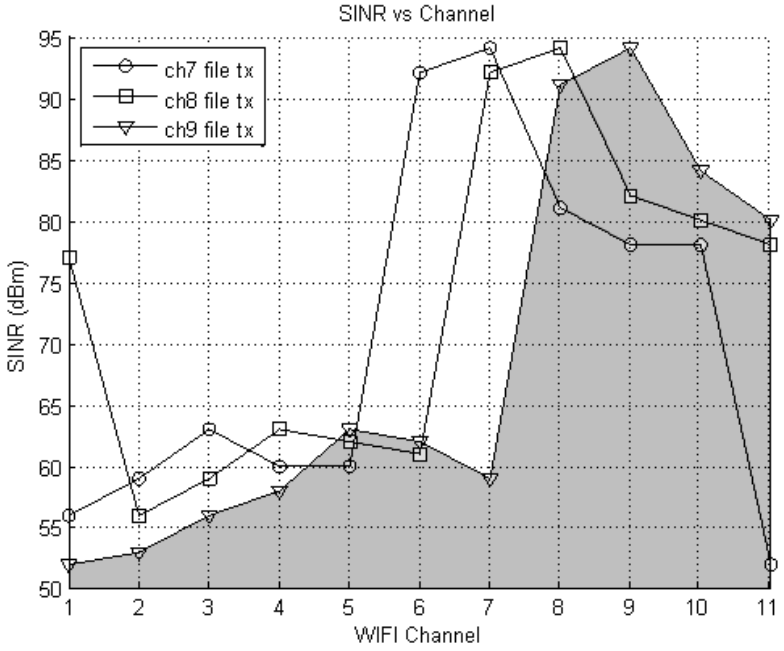


Figure 4.20: SINR distribution due to file transmission (3 of 4).

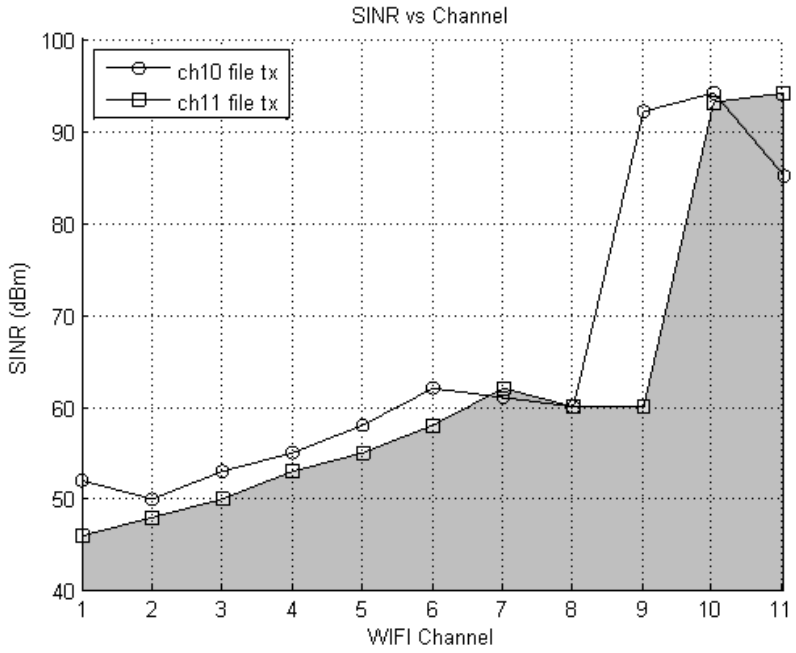


Figure 4.21: SINR distribution due to file transmission (4 of 4).

4.4.4 The Effect of File Transfer on SINR According to the Retail Spectrum Analyzer

Figure 4.22 contrasts the SINR data derived from the output of the software-based and retail spectrum analyzers against each other for a channel 6 file transfer test. Similar to the RSSI comparison presented earlier, the same spectral mask distortions and higher amplitude are present in the software-based spectrum analyzer results.

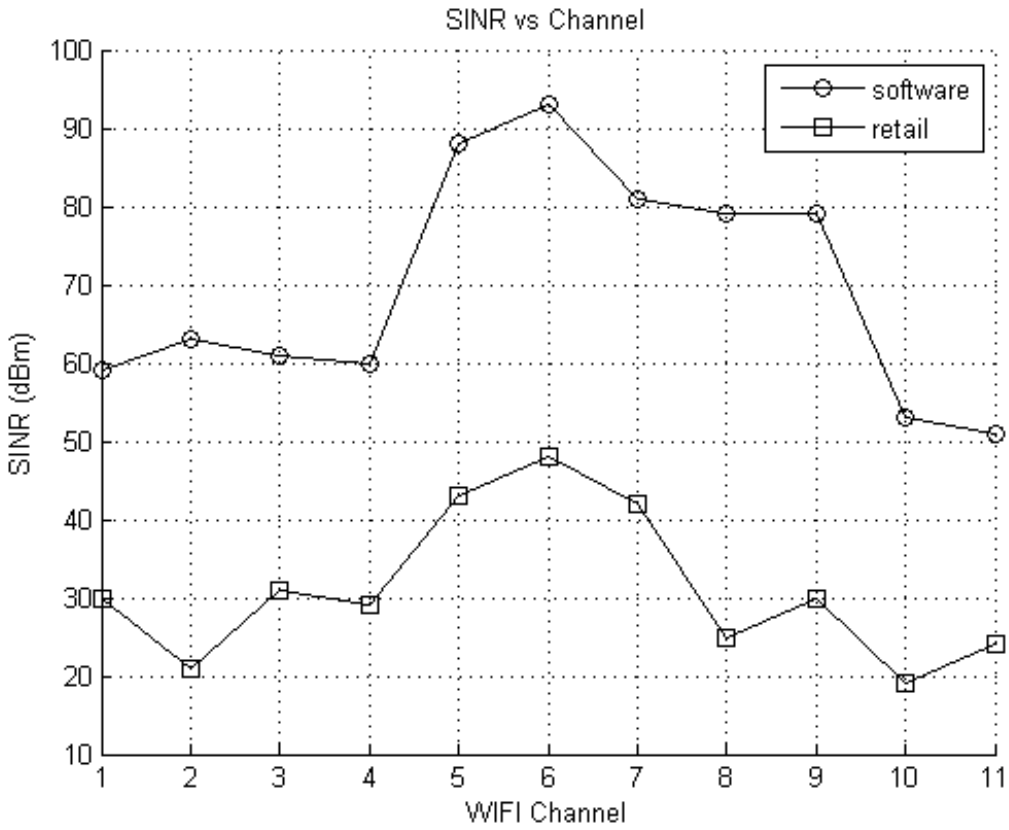


Figure 4.22: A comparison of the SINR distributions detected by the software-based and retail spectrum analyzers.

4.4.5 Statistics of the Communication Channels

Figure 4.23 aggregates together the communication channel statistics that were captured with the software-based spectrum analyzer. These metrics show that the file transfer performance was almost perfectly constant across all eleven Wi-Fi channels. This is a consequence of the -99 dBm interference and noise floor that was found on average throughout the 2.4 GHz band. If this baseline measurement was varied from channel to channel, then more fluctuations would be found in the plot.

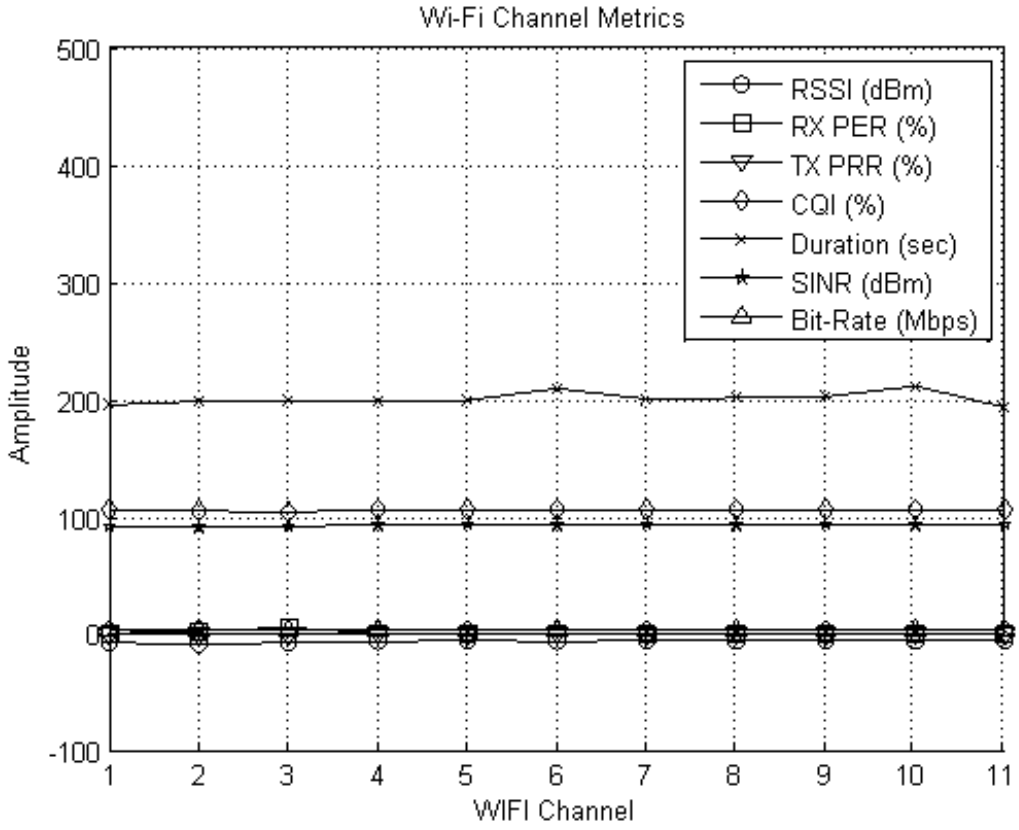


Figure 4.23: Communication channel statistics.

4.5 An Application Using Spectral Data: The AutoMin Algorithm

An application using spectral data is described. The retail spectrum analyzer was used to monitor the 2.4 GHz band. The AutoMin algorithm that was introduced in Chapter 3 uses the collected spectral information to identify the Wi-Fi channel with the lowest average RSSI and periodically adjusts the NG-Mesh network interfaces to it in order to minimize PER and maximize throughput.

Figure 4.24 presents a flowchart describing the AutoMin algorithm. The enclosed

part of the chart illustrates a handshaking process amongst the test-bed interfaces so that a consensus can be reached as to which new Wi-Fi channel they should all switch to prior to actually retuning their radios. This functionality has not been implemented in this thesis since there was only one retail spectrum analyzer available. In order to fully test this scheme, each test-bed interface would need its own spectrum analyzer. The current implementation of the NG-Mesh fits on a table top so the spectral data at each Wi-Fi interface would be identical anyway. Thus multiple spectrum analyzers are unnecessary for this setup. The enclosed part of the diagram is included for the case of extending the NG-Mesh beyond a single table top.

The *Floor Difference* is a metric that is computed in the AutoMin algorithm. It is defined as *Floor Difference=(Current Channel RSSI)-Min(RSSI of all Channels)*. *Current Channel RSSI* is the interference and noise floor that is reported by the spectrum analyzer for the current channel of communication in the test-bed. *Min(RSSI of all Channels)* is the minimum RSSI level detected by the spectrum analyzer after checking every Wi-Fi channel in the 2.4 GHz band.

The *Channel Difference* is another metric that is computed in the AutoMin algorithm. It is defined as *Channel Difference=(Current RSSI)-(Last RSSI)*. *Current RSSI* is the RSSI reading reported by the spectrum analyzer during the current period of spectral data collection for the current communication channel in the test-bed. *Last RSSI* is the RSSI reading reported by the spectrum analyzer during the last period of spectral data collection for the current communication channel in the test-bed.

Recall that the AutoMin algorithm starts by initiating spectral (i.e., RSSI) data collection with the retail spectrum analyzer. Data collection continues until the timer expires, which is set to one minute in this thesis. The *Floor* and *Channel* differences

are computed upon timer expiration and are used to determine if there exists another Wi-Fi channel with a lower RSSI level according to the currently available spectral data. If either of these differences are found to be greater than a threshold (i.e., 5 dBm in this implementation), then all test-bed interfaces are retuned to operate on the new Wi-Fi channel with the minimum average RSSI level except if this channel is the same as the currently active one. If a channel switch had occurred in the previous period of data collection, then the current communication channel is updated with the most current RSSI reading from the spectrum analyzer. This accounts for the increase in RSSI level in the newly occupied communication channel since the last period of data collection. At the end of each iteration of the algorithm, the timer is reset before spectral data collection restarts again.

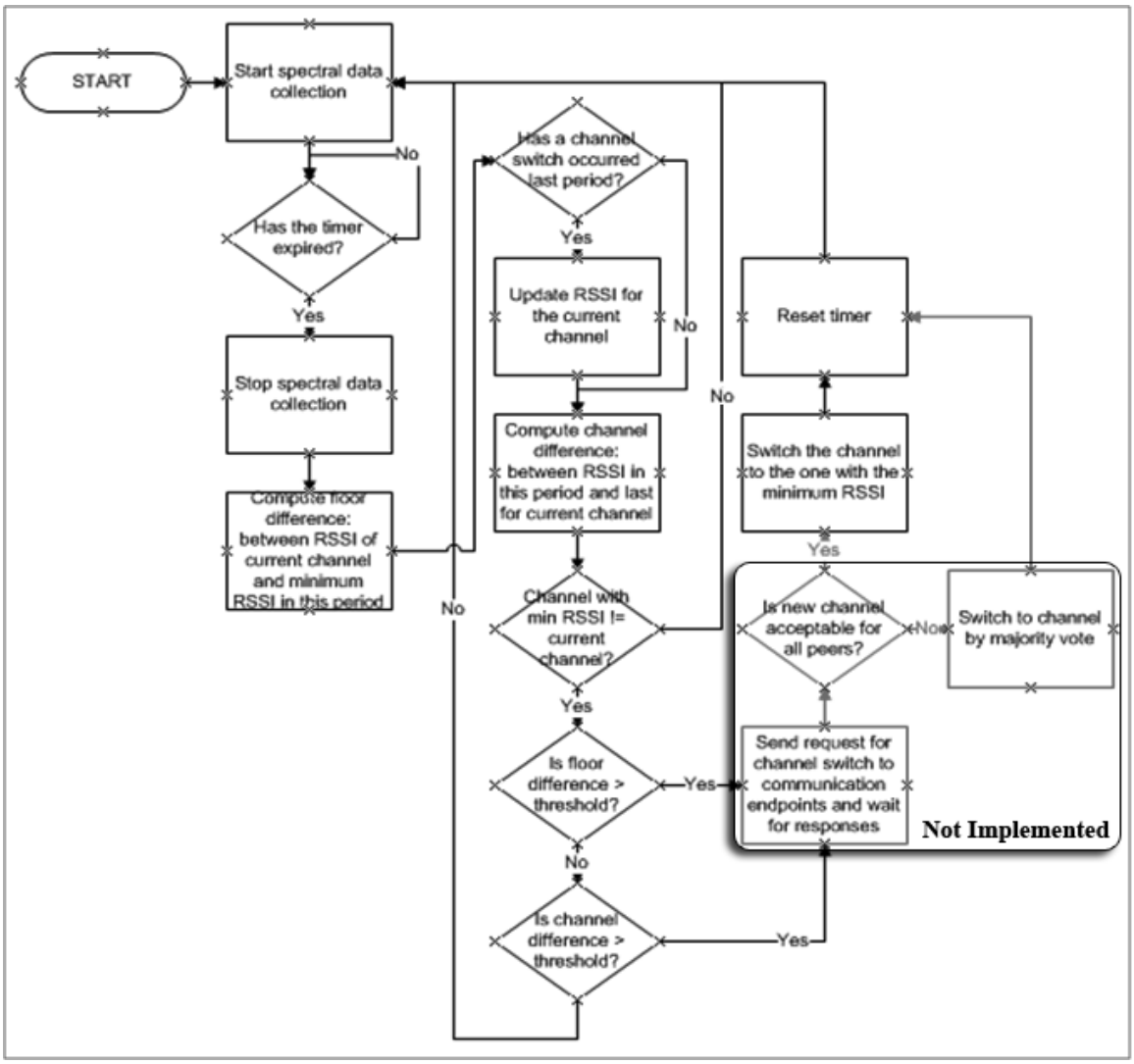


Figure 4.24: The AutoMin algorithm.

Figure 4.25 was generated using the Kismet *spectool_gtk* spectrum analyzer software and shows the effect of activating a microwave oven that is situated approximately 4 meters away from the NG-Mesh. The test-bed and microwave oven was separated by standard residential drywall measuring 12 cm thick. The microwave oven occupied Wi-Fi channel 9 as shown in the graphs. During this test, the local

AP was powered on at Wi-Fi channel 1. Other sources of interference included the remote APs that were previously detected at channels 1, 6, and 11. All interference sources are visible in the *Planar View* of the plots.

Figure 4.26 shows some of the output from the AutoMin algorithm. In the first period of spectral data collection (i.e., top third of the figure), a test-bed interface was put into infrastructure mode and although it was not actually associated with any AP, this setup was sufficient to test automatic channel switches. Prior to activating the microwave oven, this network node was setup to run the AutoMin algorithm, which tuned its radio to Wi-Fi channel 9. The microwave oven was powered on during the second period of spectral data collection (i.e., middle third of the figure), triggering an immediate switch to Wi-Fi channel 2 (i.e., the channel with the lowest average RSSI according to the current period's spectral data) as soon as the data collection timer expired. This event occurred since the average RSSI level in channel 9 had suddenly increased from -103.937 dBm in the last data collection period to -91.3725 dBm in this period due to the activation of the nearby microwave oven. This is a difference of 12.5645 dBm and is greater than the 5 dBm threshold. A switch to channel 2 was unexpected as the channel 1 local AP was powered on during this test. However, this switch is explainable since the local AP wasn't being used at the time of testing. In the third data collection period (i.e., bottom third of the figure), no channel switch occurred even though channel 4 was identified to have the lowest average RSSI level since the threshold criteria was not met. This test demonstrates the ability of the AutoMin algorithm to effectively avoid the busiest parts of the 2.4 GHz band and ensure smooth test-bed operation.

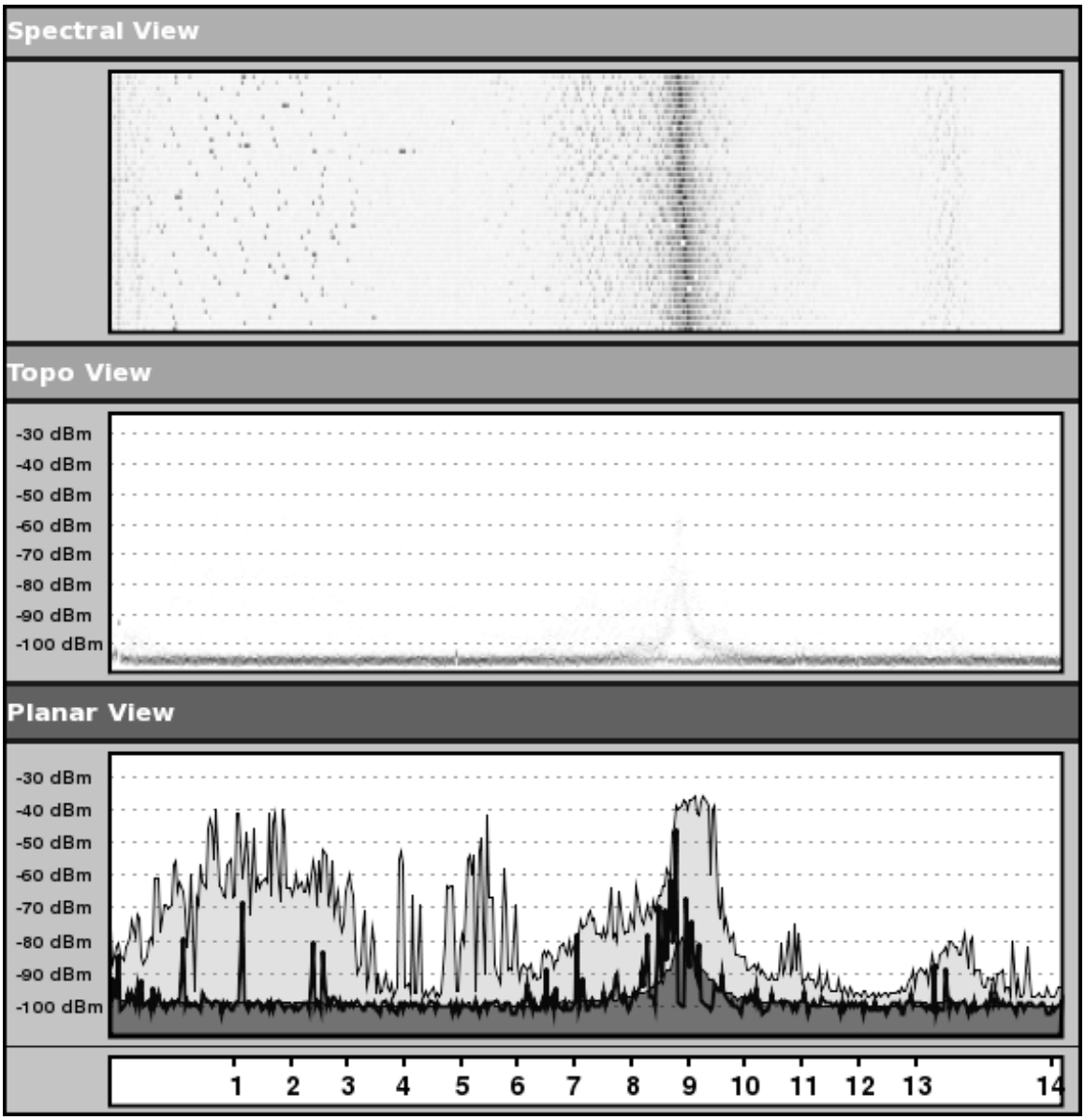


Figure 4.25: The effect of microwave oven operation on the 2.4 GHz band.

```
prevCh: 1
prevRSSI: 999
minRSSI: -103.937
deltaRSSI: 1102
avg spectral data: -102.684 -103.674 -103.779 -103.474 -103.474 -103.653 -103.579 -103.684 -103.937 -103.674 -102.579
minCh: 9
threshRSSI: 0
newCh: 9
rausb0    RT2500USB WLAN  ESSID:""
          Mode:Managed  Frequency=2.452 GHz  Bit Rate=11 Mb/s
          RTS thr:off   Fragment thr:off
          Encryption key:off
          Link Quality=0/100  Signal level:-120 dBm  Noise level:-82 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0

prevCh: 9
prevRSSI: -103.937
minRSSI: -103.647
deltaRSSI: 0
avg spectral data: -102.637 -103.647 -103.451 -103.059 -103.48 -103.52 -102.775 -102.049 -91.3725 -101.431 -102.284
minCh: 2
threshRSSI: 12
newCh: 2
rausb0    RT2500USB WLAN  ESSID:""
          Mode:Managed  Frequency=2.417 GHz  Bit Rate=11 Mb/s
          RTS thr:off   Fragment thr:off
          Encryption key:off
          Link Quality=0/100  Signal level:-120 dBm  Noise level:-82 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0

prevCh: 2
prevRSSI: -103.647
minRSSI: -103.68
deltaRSSI: 0
avg spectral data: -102.85 -103.09 -103.53 -103.68 -103.29 -103.25 -101.61 -99.57 -98.54 -103.22 -103.25
minCh: 4
threshRSSI: 0
no channel switch required.
```

Figure 4.26: A channel switching event in response to interference from a microwave oven.

The thesis implementation of the AutoMin algorithm applies to any number of network interfaces that are situated in a common wireless cell. To function beyond a single cell, extending the algorithm with a time division multiplexing (TDM) mechanism is required. The TDM mechanism would ensure that only one cell changes its frequency per time interval to avoid oscillations and instability.

4.6 Discussion

Consumer-grade Wi-Fi interfaces such as the popular Ralink chip-sets are capable of capturing RSSI metrics across the ISM band. From the driver source code, it is possible to periodically retune the Ralink radio to each of the eleven North American Wi-Fi channels and read their RSSI levels. Logic tells us that if a file transfer was occurring in isolation on Wi-Fi channel 1, then the highest RSSI reading will be detected there. And moving away to incrementally higher channels will lead to the detection of ever lower RSSI levels as the distance from the frequency of transmission activity increases. Additionally, this rate of decrease in RSSI levels should be proportional to the IEEE 802.11b spectral mask requirement. Based on this logic, it should be possible to construct a software-based spectrum analyzer from a rt2570 Wi-Fi transceiver. It was found through the experiments of this thesis that the software-based spectrum analyzer developed provided unsatisfactory results due to the following reasons.

- The RSSI readings were much higher than the actual levels in the 2.4 GHz spectrum as verified by the commercial spectrum analyzer.
- There is a noticeable latency in the updates to the RSSI registers whenever the radio is retuned to a new frequency.

For these reasons, the software-based spectrum analyzer was only able to identify the communication channel adequately.

Capturing accurate RSSI data in real-time was possible using the retail spectrum analyzer and allowed us to analyze co-channel interference patterns from file transfers occurring on each Wi-Fi channel. It also permitted us to develop the AutoMin algorithm to implement automatic congestion avoidance for our test-bed.

Chapter 5

Conclusion and Future Work

An IEEE 802.11b Wi-Fi test-bed was developed using commercially available software and transceivers (i.e., the popular Ralink rt2570 chip-set), which can be used for the rapid prototyping of wireless networks at a very low cost prior to actual deployment in the field. Each network node consists of multiple transceivers which can be individually configured using a Linux device driver to emulate any network topology of any size. The optimization of a practical network requires the ability to control key NIC functionalities such as transmission power and radio frequency. Realistic spectrum-wide physical layer statistics such as RSSI and SINR data is needed as well. Our measurements indicate that the operation of nearby devices sharing the 2.4 GHz band including Wi-Fi APs and microwave ovens can potentially affect a local network deployment. A network congestion avoidance mechanism like our AutoMin algorithm can benefit such a deployment in a densely populated setting.

A software-based spectrum analyzer was developed and its performance was compared to that of a retail one. It was found that the software-based version was not accurate enough to verify the compliance of the Ralink chip-sets against the IEEE

802.11b spectral mask requirement. Instead, the retail version confirmed this compliancy and was used to develop a detailed analysis of inter-channel interference using a sequence of file transfer tests on each of the eleven North American Wi-Fi channels.

A maximum radio retuning rate of one channel switch every 30 ms was found through driver code modification and is the lower-bound on the time that can be spent in computing scheduling decisions in routing protocols employed in the NG-Mesh. Faster rates may be possible in hardware implementations.

This work can be improved in several ways. Our test-bed uses IEEE 802.11b transceivers with omni-directional antennas, which can be replaced with IEEE 802.11n transceivers with multiple-in and multiple-out (MIMO) directional antennas. It is expected that the newer hardware will offer a significant increase in throughput and enhancements to alleviate interference from nearby ISM activities. The experiments of this thesis can also be repeated with IEEE 802.11a/g transceivers. The results from each signal type can then be compared and analyzed to infer co-channel interference characteristics. Other popular Wi-Fi chip-sets such as those made by Atheros [41] can be tested as well and compared against Ralink chip-set performance. The entry-level retail spectrum analyzer can be replaced with a state-of-the-art spectrum analyzer to capture more accurate physical layer measurements. The AutoMin algorithm can be integrated into the driver or socket code in order to gain more control over radio retuning and packet transmission operations. This algorithm can also be extended to accommodate a larger test-bed setup if multiple spectrum analyzers are available.

This thesis work was extended to the publication of an International Academy, Research, and Industry Association (IARIA) conference paper [42]. A best paper award and an invitation to publish a journal version were received.

Bibliography

[1] "IEEE 802.11 Standard," *Second Edition*, pp. 345–346, 2005.

[2] Y. O. Yazir, K. Jahanbakhsh, S. Ganti, G. C. Shoja, and Y. Coady, "A Low-cost Realistic Testbed for Mobile Ad Hoc Networks," *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 671–676, 2009.

[3] OLPC, "One Laptop Per Child," *http://www.laptop.org/en/*, 2011.

[4] D. Wu and P. Mohapatra, "QuRiNet: A Wide-Area Wireless Mesh Testbed for Research and Experimental Evaluations," *IEEE International Conference on Communication Systems and Networks (COMSNETS)*, pp. 1–10, 2010.

[5] A. Zimmermann, D. Schaffrath, M. Wenig, A. Hannemann, M. Gunes, and S. A. Makram, "Performance Evaluation of a Hybrid Testbed for Wireless Mesh Networks," *IEEE International Conference on Mobile Adhoc and Sensor Systems*, pp. 1–10, 2007.

[6] L. Barolli, M. Ikeda, F. Xhafa, and A. Duresi, "A Testbed for MANETs: Implementation, Experiences and Learned Lessons," *IEEE Systems Journal*, pp. 243–252, 2010.

[7] S. M. E. Rakabawy, S. Frohn, and C. Lindemann, "ScaleMesh: A Scalable Dual-Radio Wireless Mesh Testbed," *IEEE Annual Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks Workshops*, pp. 1–6, 2008.

[8] H. Song, C. K. Bong, J. Y. Lee, and H. S. Lee, "IEEE 802.11-based Wireless Mesh Network Testbed," *IEEE Mobile and Wireless Communications Summit*, pp. 1–5, 2007.

[9] Y. Xu and Z. Liu, "Design and Implementation of Wireless Mesh Network Testbed Based on Layer 2 Routing," *IEEE International Conference on Wireless Communications, Networking and Mobile Computing*, pp. 1–4, 2008.

[10] M. Ye, W. Wu, and X. Xu, "MR Mesh: A Multi-Radio Wireless Mesh Network Testbed," *IEEE International Conference on Wireless Communications, Networking and Mobile Computing*, pp. 1–4, 2008.

[11] D. Heah and Z. R. Zaidi, "A Testbed Implementation of Monitoring Assisted Local Route Recovery Scheme for Wireless Mesh Networks," *IEEE International Conference on Advances in Mesh Networks*, pp. 120–125, 2009.

[12] M. Krebs, S. Widyadharma, and K. H. Krempels, "Testbed Performance Analysis of Cross-Layer Based Service Discovery in Wireless Mesh Networks," *IEEE Wireless Days*, pp. 1–5, 2008.

[13] C. Edmonds, D. Joiner, S. Springer, K. Stephen, and B. Hamdaoui, "Cognitive Wireless Mesh Network Testbed," *IEEE Wireless Communications and Mobile Computing Conference*, pp. 373–376, 2008.

[14] S. Jadhav, T. Brown, S. Doshi, D. Henkel, and R. Thekkekunnel, "Lessons Learned Constructing a Wireless Ad Hoc Network Test Bed," *Wireless Network Measurement Workshop*, 2005.

[15] Z. Jin, B. Liang, Y. Shu, and O. W. Yang, "Designing and Implementing a Mobile Ad Hoc Network Testbed," *IEEE Canadian Conference on Electrical and Computer Engineering*, pp. 1559–1564, 2002.

[16] J. T. Kaba and D. R. Raichle, "Testbed on a Desktop: Strategies and Techniques to Support Multi-hop MANET Routing," *ACM MobiHoc Protocol Development*, pp. 164–172, 2001.

[17] H. Lundgren, D. Lundberg, J. Nielsen, E. Nordstrom, and C. Tschudin, "A Large-Scale Testbed for Reproducible Ad Hoc Protocol Evaluations," *IEEE Wireless Communications and Networking Conference*, pp. 412–418, 2002.

[18] S. Sanghani, T. X. Brown, S. Bhandare, and S. Doshi, "EWANT: The Emulated Wireless Ad Hoc Network Testbed," *IEEE Wireless Communications and Networking*, pp. 1844–1849, 2003.

[19] S. L. Shrestha, J. Lee, A. Lee, K. Lee, and S. Chong, "An Open Wireless Mesh Testbed Architecture with Data Collection and Software Distribution Platform," *IEEE International Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities*, pp. 1–10, 2007.

[20] Q. Nasir and M. Al-Balt, "Linux-Based Testbed for an Improved IEEE 802.11 Backoff Algorithm," *IEEE International Conference on Communications and Mobile Computing (CMC)*, pp. 237–242, 2010.

[21] M. R. Souryal, A. Wapf, and N. Moayeri, "Rapidly-Deployable Mesh Network Testbed," *IEEE Global Telecommunications Conference*, pp. 1–6, 2009.

[22] K. Lan, Z. Wang, R. Berriman, T. Moors, M. Hassan, L. Libman, M. Ott, B. Landfeldt, Z. Zaidit, and A. Senevirante, "Implementation of a Wireless Mesh Network Testbed for Traffic Control," *IEEE International Conference on Computer Communications and Networks*, pp. 1022–1027, 2007.

[23] N. Bayer, D. Hock, A. Roos, M. Siebert, B. Xu, V. Rakocevic, and J. Habermann, "VoIP Performance in MeshBed - A Wireless Mesh Networks Testbed," *IEEE Vehicular Technology Conference*, pp. 2218–2222, 2008.

[24] MadWiFi, "The MadWiFi Project," *http://madwifi-project.org/*, 2011.

[25] Y. Tarte, A. Amanna, and C. Okwudiafor, "Experimental Testbed for Investigating IEEE 802.11 Handoff in Vehicular Environment," *IEEE SoutheastCon*, pp. 222–225, 2010.

[26] V. Badarla, D. Malone, and D. J. Leith, "Implementing TCP Flow-Level Fairness Using 802.11e in a Multi-Radio Mesh Testbed," *IEEE Communications Letters*, pp. 262–264, 2008.

[27] D. A. Maltz, J. Broch, and D. B. Johnson, "Quantitative Lessons From a Full-Scale Multi-hop Wireless Ad Hoc Network Testbed," *IEEE Wireless Communications and Networking Conference*, pp. 992–997, 2000.

[28] Y. Zhang and W. Li, "An Integrated Environment for Testing Mobile Ad-Hoc Networks," *ACM MobiHoc*, pp. 104–111, 2002.

[29] Linksys, "WUSB54Gv4 Datasheet," *http://homesupport.cisco.com/en-us/wireless/lbc/WUSB54G/download*, 2010.

[30] LinuxWireless, "RT2571 EEPROM Channels," *http://wireless.kernel.org/en/developers/Documentation/specs*, 2010.

[31] SerialMonkey, "Welcome to the rt2x00 Project," *http://rt2x00.serialmonkey.com/wiki/index.php/Main_Page*, 2011.

[32] E. Ancillotti, R. Bruno, and M. Conti, "Experimentation and Performance Evaluation of Rate Adaptation Algorithms in Wireless Mesh Networks," *ACM symposium on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*, pp. 7–14, 2008.

[33] wize, "Linksys Wireless-G WUSB54G 802.11g/b Reviews: Description & Specs," *http://wize.com/wireless-adapters/p17329-linksys-wireless-g-wusb54g-80211g-b*, 2010.

[34] WiresharkFoundation, "Wireshark," *http://www.wireshark.org/*, 2011.

[35] M. J. Donahoo and K. L. Calvert, "TCP/IP Sockets In C," *First Edition, Morgan Kaufmann Publishers, San Francisco, California*, 2001.

[36] B. W. Kernighan and D. M. Ritchie, "The C Programming Language," *Second Edition, Prentice Hall, Upper Saddle River, New Jersey*, 1998.

[37] B. Hall, "Beej's Guide to Network Programming Using Internet Sockets," *http://beej.us/guide/bgnet/output/print/bgnet_USLetter.pdf*, 2011.

[38] MetaGeek, "Wi-Spy 2.4i," *http://www.wi-spy.ca/wispy24i.php*, 2010.

[39] Kismet, "Spectrum-Tools," *http://www.kismetwireless.net/spectools/*, 2010.

[40] A. L. Garcia and I. Widjaja, "Communication Networks: Fundamental Concepts and Key Architectures," *Second Edition, Mcgraw-Hill Companies, Columnbus, Ohio*, p. 113, 2004.

[41] Qualcomm, "Qualcomm Atheros," *http://www.qca.qualcomm.com/*, 2011.

[42] S. W. K. Ng and T. H. Szymanski, "Physical Layer Measurements for an 802.11 Wireless Mesh Network Testbed," *IARIA Fifth International Conference on Digital Society*, 2011.