University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Theses, Dissertations, and Student Research from Electrical & Computer Engineering | Electrical & Computer Engineering, Department of

Summer 7-25-2017

Application-aware Cognitive Multi-hop Wireless Networking Testbed and Experiments

Trenton T. Evans
University of Nebraska-Lincoln, evanstrenton@gmail.com

Follow this and additional works at: http://digitalcommons.unl.edu/elecengtheses

Part of the Computer Engineering Commons, Other Electrical and Computer Engineering Commons, Signal Processing Commons, and the Systems and Communications Commons

Evans, Trenton T., "Application-aware Cognitive Multi-hop Wireless Networking Testbed and Experiments" (2017). *Theses, Dissertations, and Student Research from Electrical & Computer Engineering*. 89.
http://digitalcommons.unl.edu/elecengtheses/89

This Article is brought to you for free and open access by the Electrical & Computer Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Theses, Dissertations, and Student Research from Electrical & Computer Engineering by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

APPLICATION-AWARE COGNITIVE MULTI-HOP WIRELESS NETWORKING TESTBED AND EXPERIMENTS

by

Trenton T. Evans

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Telecommunications Engineering

Under the Supervision of Professor Yi Qian

Lincoln, Nebraska

May, 2017

APPLICATION-AWARE COGNITIVE MULTI-HOP WIRELESS NETWORKING TESTBED AND EXPERIMENTS

Trenton T. Evans, M.S.

University of Nebraska, 2017

Adviser: Yi Qian

In this thesis, we present a new architecture for application-aware cognitive multi-hop wireless networks (AC-MWN) with testbed implementations and experiments. Cognitive radio is a technique to adaptively use the spectrum so that the resource can be used more efficiently in a low cost way. Multihop wireless networks can be deployed quickly and flexibly without a fixed infrastructure. In presented new architecture, we study backbone routing schemes with network cognition, routing scheme with network coding and spectrum adaptation. A testbed is implemented to test the schemes for AC-MWN. In addition to basic measurements, we implement a video streaming application based on the AC-MWN architecture using cognitive radios. The Testbed consists of three cognitive radios and three Linux laptops equipped with GNU Radio and GStreamer, open source software development toolkit and multimedia framework respectively. Resulting experiments include a range from basic half duplex data to full duplex voice communications and audio/video streaming with spectrum sensing. This testbed is a foundation for a scalable multipurpose testbed that can be used to test such networks as AC-MWN, adhoc, MANET, VANET, and wireless sensor networks. Experiment results demonstrate that the AC-MWN is applicable and valuable for future low-cost and flexible communication networks.

DEDICATION

Dedicated to my father Dr. Ted Evans Jr. and to the memory of my mother Betty Evans (1944 - 2011)

ACKNOWLEDGMENTS

Foremost, I would like to thank my advisor, Dr. Yi Qian for providing his time, wisdom and support throughout my graduate school experience. You have been a great role model and mentor who always kept me motivated and taught me what it takes to be a quality researcher. I am proud to say that I was a part of your research team.

I'd like to thank my thesis committee members for your guidance and I am honored to have published an article with my name next to yours as a co-author. Being able to share ideas with you has been a great experience.

I'd like to thank Dr. Bing Chen for being a mentor and providing wisdom, support, motivation and encouraging me to do the Masters degree program.

I'd like to thank Professor Roger Sash and Professor Herbert Detloff and convey my appreciation for sharing their wisdom and honoring me with the opportunity to be their teaching assistant.

I'd like to thank the Electrical and Computer Engineering Department faculty and staff for always going above and beyond what they are expected to do. Also, for making myself and other students feel like part of one big cohesive team. The knowledge gained and lessons learned while attending school here are immeasurable and have undoubtedly made me a better person, scholar and productive citizen.

I'd like to thank the graduate students who were on Dr. Yi Qian's research team with me. The constant support and exchange of ideas between us provided a motivating environment. A special thanks to Dr. Feng Ye who not only contributed to my published article but was always there to provide feedback and guidance. Also, thank you Dr. Zhihui Shu, Shengjie Xu and Dongfeng Fang for everything you all have done for me.

I’d like to thank Kossivi Tossou for his technical contributions on the testbed and detailed documentation.

I’d like to thank my girlfriend Nicole LaPatka for her love, support, motivation and patience while I spent many hours in the lab.

I’d like to thank my Dad, Dr. Ted Evans and my step-mother Carol Lamb Evans (CJ) for always being there for me with love, support and motivation. Dad, youve always inspired me to be better and believed that I could accomplish this even when I didn’t. Words cannot express my gratitude to both of you.

Contents

Contents vi

1 Introduction 1

2 Background of Multi-hop Wireless Cognitive Radio Networks 6

2.1 MWCRN 6

2.2 Application Adaptation for AC-MWN 8

2.3 Spectrum Adaptation for AC-MWN 9

2.4 Multi-hop Wireless Network Routing Protocols 13

2.4.1 Routing protocols 13

2.4.2 Single hop 14

2.4.3 Multi-hop 15

2.4.3.1 Proactive 16

2.4.3.2 Reactive 17

2.4.3.3 Hybrid 17

2.4.4 Static networks 18

2.4.5 Dynamic networks 19

3 Experimentation of AC-MWN 20

3.1 Equipment Setup 20

3.1.1 Software Defined Radio (SDR) configuration 22

4 Basic Testing **26**

4.1 Single-hop Transmission 29

4.1.1 Half-duplex one-way transmission 29

4.1.2 Adaptive One-Way Transmission 31

4.2 Half-duplex two-way transmission 32

4.3 Full-Duplex Transmission 35

4.3.1 Adaptive Full-Duplex Transmission 38

4.4 Multi-Hop Transmission 40

4.4.1 Simple Transceiver Transmission with Two Smart Radios . . . 40

4.4.2 Two Radio Multi-Hop (Round-Trip) Transmission 41

4.4.3 Three Radio Multi-Hop (Circular) Transmission 45

5 Spectrum Sensing **53**

5.1 Sensing with No Primary User Activities 53

5.2 Sensing with Primary User Activities 54

5.3 Audio/Video Application 59

5.3.1 Webcam 59

5.3.2 User Datagram Protocol (UDP) 60

5.3.3 Video File Transfer via Virtual Channel 61

5.3.4 Live Streaming Using Two USRP2 SDRs 62

5.3.5 Audio Recording and Streaming with GStreamer 63

5.3.6 Live Video with Test Audio 69

5.3.7 Live Video and Live Audio 70

5.4 Creating GRC Block for Spectrum Sensing and Channel Allocation . 73

5.5 Live Audio, Spectrum Sensing, and Dynamic Channel Allocation . . 77

5.5.1 Python Script 77

5.5.1.1 Main 77

5.5.1.2 Get Average 78

5.5.1.3 Spectrum Sense & Channel Allocation 79
5.5.1.4 GStreamer Live Audio 79
5.5.1.5 GNU Radio Companion 81
5.5.2 Running Demo 82
5.5.3 Basic Measurements for Benchmark 83
5.5.4 Video Streaming Application 85

6 Conclusion **94**

Bibliography **97**

Chapter 1

Introduction

In multi-hop wireless networks (MWN), there are one or more intermediate nodes along the path (route) that receive and forward packets via wireless links. In cellular and wireless local area networks, wireless communications only occurs on the last link between a base station and the wireless end system. Multi-hop wireless networks have several benefits: (1) Compared with networks with a single wireless link, multi-hop wireless networks can extend the coverage of a network and improve connectivity; (2) Transmission over multiple short links might require less transmission power and energy than that required over long links; (3) Multi-hop wireless networks can be quickly deployed without the support (or with limited support) from wired infrastructure. Due to such salient features, multi-hop wireless networks are expected to play a key role in modern society, helping to improve quality of life and to solve problems related to homeland security, protection of critical infrastructure, and the diagnosis and treatment of illnesses. In the past two decades, we have witnessed a dramatic growth in wireless communications and networks, with mobile devices such as cell phones, personal digital assistants (PDAs), and laptop computers becoming essential to everyday life. Such a trend has been accelerated in the past three years, driven by

the popularity of a new generation of mobile devices like netbooks, smart phones (iPhone, Andriod Phones, etc.), and other new gadgets (Kindle reader, iPad, etc.). Our society has been rapidly evolving toward the pervasive computing age, in which the network infrastructure shall support not only traditional communication patterns (i.e., human-to-human, human-to-computer, and computer-to-computer) but also the communication needs from devices such as mobile phones, PDAs, sensors, and radio frequency identification (RFID) devices. To support the so-called Internet-of-things, it becomes a major challenge to fully explore multi-hop wireless networks. Given the recent upward trend in wireless traffic, capacity demand increases faster than spectral efficiency and availability (in particular at hot spots/areas). On the other hand, it has been well-known that in wireless communications most of the spectrum is significantly under-utilized in most of the time. This simple fact has attracted researchers from academia and industry who are interested in the next generation cognitive and radio communication systems. Despite the importance of such ongoing efforts, we have observed a huge gap between the research on cognitive radios and the network applications. For instance, most researchers on cognitive radios focus on the spectrum sensing and dynamic spectrum allocation, but few of them have considered to interact with the network layer so that the network can accommodate the requirements of applications from the upper layer. In spite of many recent research activities on the topics related to multi-hop wireless networks, including cognitive radios, capacity analysis and improvement, and wireless applications, there is still a critical gap in the knowledge base to understand the network design principles to meet the requirements of different applications for multi-hop wireless networks with the constraints of spectrum availability.

In this thesis, we propose and study a new wireless networking architecture, Application-Aware Cognitive Multi-hop Wireless Networks (AC-MWN) and imple-

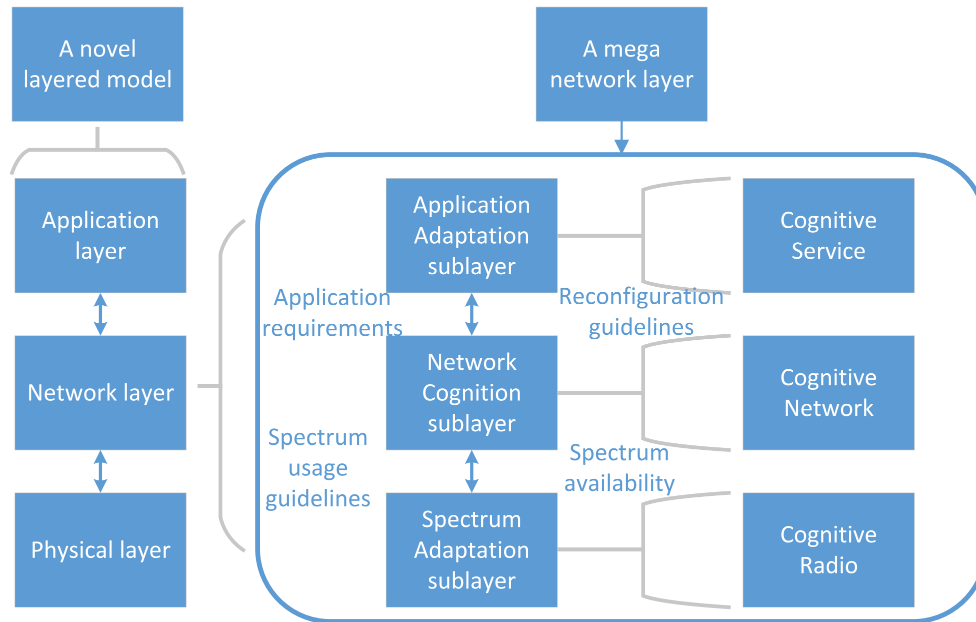

Figure 1.1: A new model and a mega network layer for the AC-MWN architecture

mentation of testbed. In AC-MWN, we apply a new layered model that has three layers (application layer, network layer and physical layer), as shown in Fig. 1.1. We propose to design a mega network layer for multi-hop wireless networks that will combine the major functionalities of medium access control layer, network layer and transport layer in the traditional layered model of wireless networks. The mega network layer will have cognition related to the spectrum availability in the network and the requirements of applications of the network. Moreover, the new design not only can efficiently utilize the spectrum resources, but also can effectively adjust the application resources, such as storage and computational capabilities, etc. Since each MWN will be deployed as an autonomous domain even if it is interconnected with the Internet, we envision that our novel layered model and the mega network layer for the AC-MWN architecture can be implemented and deployed in the MWN domain regardless of the standard 7-layer ISO/OSI model or the standard TCP/IP layer model, in a manner similar to most wireless sensor networks. AC-MWN can push the capacity limit for MWNs and at the

same time to accommodate the requirements of different applications in MWNs; thus, it achieves application-aware cognitive multi-hop wireless network design.

At the center of the AC-MWN architecture is a mega network layer model as shown in Fig. 1.1, which consists of three sublayers: application adaptation sublayer, network cognition sublayer, and spectrum adaption sublayer. The application adaptation sublayer is the interface in the network layer that interacts with the application layer; it passes the application requirements from the application layer to the network cognition sublayer, and it receives reconfiguration guidelines from the network cognition sublayer. The network cognition sublayer is the central part of the AC-MWN for cognitive networking. This layer takes the application requirements from the application adaptation sublayer, and it also receives the spectrum availability information from the spectrum adaptation sublayer. Both the application requirements and the spectrum availability information will be used to generate the best fitting routing strategies in the network layer (for the required applications with the spectrum constraints); then, the best fitting routing strategies will be mapped into spectrum usage guidelines to be passed to spectrum adaptation sublayer below. The best fitting routing strategies will also be translated into reconfiguration guidelines and then sent to the application adaptation sublayer above. The spectrum adaptation sublayer is the interface in the network layer that interacts with the physical layer for cognitive radio functions like spectrum sensing and dynamic spectrum allocations. The previously published AC-MWN architecture [1] focused on the mega network layer, which including application adaption sublayer, network cognition sublayer, and spectrum adaption sublayer, will be discussed in the three sections following.

In Chapter II., we describe the application adaptation, and the design of network cognition for AC-MWN, including backbone network construction, routing with channel bonding, and routing with network coding as proposed in [1], [2] and

[3]. Also, we elaborate the spectrum adaptation for AC-MWN, including channel-aware routing for both unicast and multicast transmissions. Next, a background in ad-hoc routing protocols is given with examples to help define the need for a more robust hybrid protocol. Finally, simulation setup and results are discussed. We set up a testbed, and show experiment results of sensing, channel switching, as well as a video streaming application based on the presented AC-MWN.

Chapter 2

Background of Multi-hop Wireless Cognitive Radio Networks

2.1 MWCRN

Multi-hop wireless cognitive radio networks (MWCRNs) combine two different technologies that increase spectrum efficiency. These technologies are multi-hop wireless networking (MWN) and cognitive radio networking (CRN). Combining these two is made possible by recent advances in hardware and routing protocols that enable multi-channel, device-to-device (D2D) and machine-to-machine (M2M) RF communications. Spectrum demand is increasing at unprecedented levels. This is largely due to factors such as urbanization, new high data applications, and the Internet of Things (IoT). The population density of our cities is increasing and therefore so is the demand for bandwidth from end users. Adding 4G LTE and 5G high speed and capacity channels with big data analytics and the low cost of sensors is resulting in government and businesses deploying massive sensor networks to gather information. Also, due to the demand of mobile multi-media services there are predictions of global wireless-traffic volume increasing by three

orders of magnitude by 2020 starting in 2010 [4].

Efficiently utilizing costly spectrum is necessary due to the limited amount available. One way to do this is by decreasing the load on the base station (BS). Multi-hop D2D and M2M communications allow information to be shared from sources other than the BS, thus decreasing its load. This benefits of using the aforementioned technologies is explained very well in [5]. These technologies allow peer-to-peer (P2P) communications which take advantage of device proximity and ubiquitous advanced mobile devices capable of handling the multiple protocols needed.

Cognitive radio networking uses spectrum sensing to detect unused portions of the radio frequency (RF) spectrum in order to diffuse traffic, alleviating congestion [6]. However, most of the RF spectrum is leased to commercial or government entities. These entities pay the Federal Communication Commission (FCC) for a license and these license holders are known as primaries in CRN research lexicon. When a channel is not occupied by a primary user (PU), a secondary user (SU) can use the channel opportunistically after sensing. The major characteristic of cognitive radios is that the activities of PUs change dynamically and channel availability changes from time to time. Thus, the SUs have to efficiently sense the activities of PUs and make decisions on which channel to use in the changing environment.

Fundamentally, the required elements of a MWCRN include spectrum sensing, fair dynamic channel allocation, adequate quality of service (QoS), and seamless handover, enabling mobility [7].

2.2 Application Adaptation for AC-MWN

In this section, we describe the design of the application adaptation sublayer for AC-MWN, and the interactions with the application layer above and the network cognition sublayer below. In this way, the application adaptation sublayer will achieve "application-aware" design in AC-MWN. In future MWNs, each type of application can be characterized by communication patterns and service requirements [8].

Communication patterns: A network application can have one of the specific communication patterns [8]: (1) one-to-one (unicast), (2) one-to-many (multicast), (3) one-to-all (broadcast), (4) many-to-one, and (5) many-to-many.

Service requirements: A network application can also be associated with a variety of service requirements, including data rate type (fixed or variable), delay type (real-time, non-real-time, and delay-tolerant), and security and reliability requirements [8].

In the application adaptation sublayer design for AC-MWN, we specifically focus on the application characteristics of communication patterns and service requirements with data rate type and delay type, which are the major characteristics of network applications that could affect the design and operation of MWNs. The work in [8] shows that WiMAX networks can be properly complemented through advance connection management and scheduling in the network layer with the consideration of application characteristics from the application layer above. In this AC-MWN we consider different combinations of communication patterns and service requirements for future applications of MWNs, and the impact of the network application characteristics on MWN network layer design including routing schemes.

We define a simple interface between the application layer above and the application adaption sublayer, so that a network application can be abstracted in

the application adaption sublayer in terms of the communication patterns and service requirements discussed early, to be used by the network cognition sublayer below. We also define a simple interface between the application adaption sublayer and the network cognition sublayer below, so that the abstracted application requirements can be passed to the network cognition sublayer for routing and scheduling, and it will also receive the reconfiguration guidelines from the routing and scheduling functions of the network cognition sublayer, so that different applications will be accommodated in MWNs.

2.3 Spectrum Adaptation for AC-MWN

In this section we present the design of the spectrum adaptation sublayer for AC-MWN, and the interactions with the network cognition sublayer above and the physical layer below. The spectrum adaptation sublayer will obtain the spectrum availability information from the spectrum sensing function in the physical layer below, and pass it to the network cognition sublayer above for the network layer function (routing) design. At the same time, it will receive the spectrum usage guidelines from the network cognition sublayer above, and the spectrum usage guidelines will be used for dynamic spectrum allocation in the physical layer below.

Here, we investigate how the spectrum availability information can be abstracted and be used by the algorithms in the network cognition sublayer. Spectrum sensing has been comprehensively studied for the past ten years. In order to achieve the cognitive network design in AC-MWN, it is necessary to implement the basic spectrum sensing schemes and provide the realistic spectrum availability information to the network layer. We consider the spectrum occupancy as random variables over frequency and time domain. Based on the stochastic properties

and asymptotic performance of eigenvalues of random matrices, we can apply these properties for the spectrum sensing in cognitive radios. We can capture the channel state information by exploring the following schemes.

Scheme 1 - Energy detection for wideband spectrum sensing. In wideband cognitive radio, wideband (i.e. from 0 to 3 GHz) spectrum sensor scanning multiple licensed bands may not be practical for all feature detection algorithms to identify the primary users (PUs) operating in the measured frequency band. In this case, it may be preferred to use energy detection. As a secondary user (SU) or cognitive user, this sensing and transmission function is performed over the wider bandwidth to give the highest probability of detecting unused spectra-opportunistic transmission. The unique sensing function requires quality hardware such that the front-ends have several GHz sampling rate with high resolution (at least 12 bits), if GHz bandwidth need to be searched. Therefore, we can use energy-based sensing which does not require a priori knowledge of the signals.

Scheme 2 - Spectrum sensing using cyclostationarity. The inherent spectral redundancy caused by the use of a cyclic prefix in orthogonal frequency division multiplexing (OFDM) signals has been exploited in several literature, e.g., [9, 10]. A unified approach to the recognition of signals belongs to the three basic air interfaces categories: single carrier TDMA, OFDM systems, and single carrier CDMA systems. It is also used in wideband CDMA. It has been used in a framework of overlay/underlay cognitive radio. This unified approach may be the most promising from the view point of stochastic performance, if there is a priori information about the communications such as modulation format [10]. Therefore higher-order statistics of the cyclostationary signals are explored for spectrum sensing.

Scheme 3 - Sensing dynamic range of front-end. As written in [11]:

> A major limitation is a radios front-end ability to detect weak signals

> is its dynamic range, which dictates the requirement for number of bits in A/D converter. Since it is difficult to design high-resolution A/D converters - the pricing will not follow Moore's law, it is highly desirable to relax the A/D requirement. In addition, the power consumption and complexity of ADC increases nearly exponentially with the resolution or the number of bits.

For example, TV broadcasters have set a stringent limit for the digital TV signals to be reliably detected (probability of detection greater than 90% and probability of false alarm less than 10%) at a signal strength of -116 dBm per IEEE 802.22 parameters [12]. This translates to roughly -21 dB of signal-to-noise ratio (SNR) based on the receiver noise figure (NF) of around 11 dB and the use of omni-directional antenna for spectrum sensing. Based on the traditional estimation and detection framework, FCC determined a detection sensitivity of -114 dBm. Measurements suggest that using this threshold will result in limited white space availability, especially in metropolitan area where spectrum demand is high. The research community has developed cooperative approaches to spectrum sensing that do not require the same fading margins because they can exploit cooperative diversity. However, these approaches are impractical because the current regulatory model is based on certification of individual devices, and there is no notion of certifying the cooperative performance of devices. Therefore, we plan to develop fast algorithms insensitive to the dynamic range.

In the design of spectrum adaptation sublayer for AC-MWN, we first study how the spectrum availability information can be abstracted and be used by the algorithms in the network cognition sublayer, i.e., how to capture the channel state information by exploring the three schemes described above, Scheme 1 - Energy detection for wideband spectrum sensing; Scheme 2 - Spectrum sensing using cyclostationarity; and Scheme 3 - Sensing dynamic range of front-end. We define a

simple interface between the network cognition sublayer above and the spectrum adaption sublayer, so that the spectrum availability information obtained above can be used by the network cognition sublayer above to make the routing and transmission decisions. We further define a simple interface between the spectrum adaption sublayer and the physical layer below, so that spectrum usage guidelines can be passed to the physical layer for dynamic spectrum access, thus a cognitive MWN.

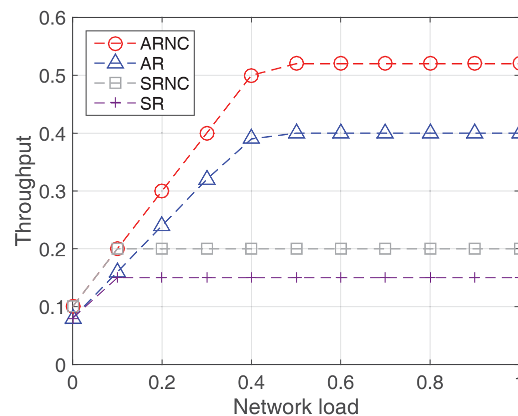

Figure 2.1: Network load v.s. throughput in uni-cast

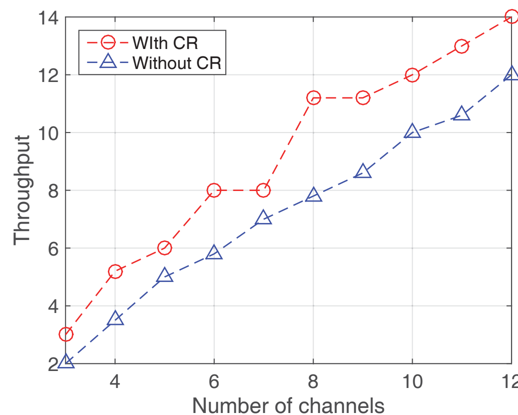

Figure 2.2: Number of channels v.s. throughput in multicast transmission

Because of the dynamic activities of PUs, SUs need to allocate their resources and route accordingly so that the multi-hop transmission can remain connected

with high throughput. In [13], we apply network coding with backpressure algorithm and dynamic channel allocation scheme into unicast routing in AC-MWN. Our objective is to maximize the aggregated throughput of all time slots while ensuring the stability of all the queues from the backpressure algorithm. As shown in Fig. 2.1, our schemes (i.e., AR for adaptive routing and ARNC for adaptive routing with network coding) achieve higher throughput than both shortest path routing (SR) and shortest path routing with network coding. Moreover, we once again prove that network coding improves network performance.

In addition to unicast, we further study channel allocation and routing in multicast transmission in AC-MWN [14]. Multicast has its advantage of saving spectrum resources by broadcasting. Due to the dynamic activities of PUs, tree-based routing schemes may not work well for multicast transmission in AC-MWN. In our schemes, we maximize the transmission rate of the network with several multicast sessions. As shown in Fig. 2.2, our scheme achieves higher throughput.

2.4 Multi-hop Wireless Network Routing Protocols

2.4.1 Routing protocols

Considering the importance of mobility, scalability, and adaptability, MWN protocols are ideal for modern wireless networks.

Two categories of MWN routing protocols include reactive and proactive [15]. Hybrid protocols combine advantageous components of both proactive and reactive protocols. Hierarchical protocols exclusively use proactive or reactive protocols depending on situation and can switch when network environment changes. A high level flow chart is shown in Figure 2.3. Each node in proactive routing maintains routing tables and often use modified versions of Dijkstra's algorithm to calculate the shortest path to every other node, especially in the simulated

environment [16]. Reactive routing is also known as on demand routing because instead of keeping periodically updated tables, each node will initiate a route discovery stage when it needs to transmit a message. Proactive routing such as Optimized Link State Routing Protocol (OLSR) has greater routing control overhead than reactive but has no route discovery delay. Also, the overhead generated is independent of the number of routes being created. However, reactive routing such as Ad hoc On-Demand Distance Vector Routing (AODV) reduces the amount of overhead but increases delay.

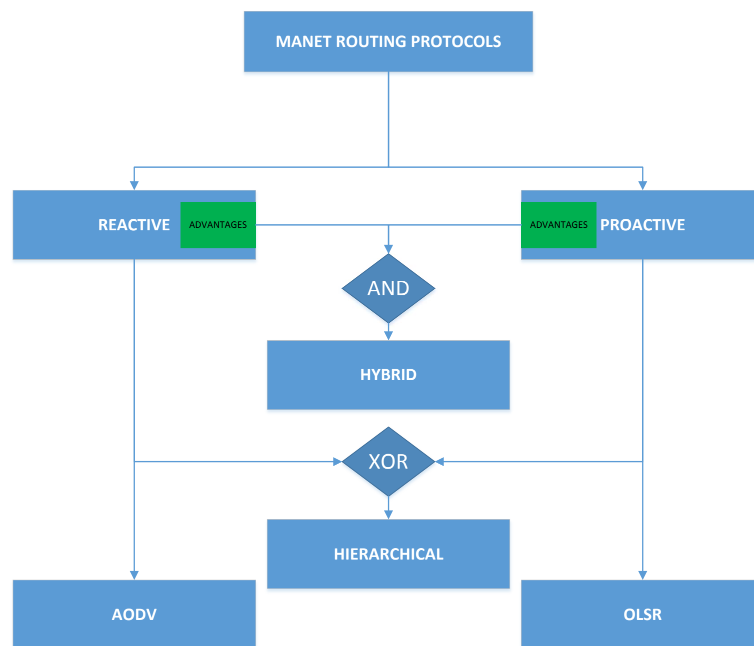

Figure 2.3: Manet Routing Schemes

2.4.2 Single hop

Classic cellular networks are an example of single-hop networks where the user equipment (UE) communicates directly with the base station (BS). This config-

Table 2.1: OLSR vs. AODV

Characteristics	**OLSR**	**AODV**
End-to-end delay	LOW	HIGH
Communication overhead	HIGH	LOW
Scalability	LOW	MEDIUM

uration is not power or spectrum efficient. Using the Friis power transmission equation shown in (2.1) you can see that radius or R in the denominator is the only non-constant (usually the change in frequency is negligible), showing that the transmission power is dependent on the distance between BS and EU. Therefore, adding hops can reduce congregate transmission power and increase energy efficiency [17].

$$\frac{P_r}{P_t} = G_r G_t \frac{(\lambda)^2}{(4\pi R)^2} \tag{2.1}$$

2.4.3 Multi-hop

Choosing the correct routing protocol depends on several factors and one of those is how fast the topology changes. These changes include number of nodes, connectivity, size and rate of change. Also, there are several different types of ad-hoc networks such as some wireless sensor networks (WSN), mobile ad-hoc networks (MANETs) and vehicular ad-hoc networks (VANETs). Even more suited for our area of research is cognitive wireless mesh networks and cognitive radio ad-hoc networks (CRAHNs) presented in [18] and [19] respectively. Each of these networks have unique engineering constraints and therefore often rely on different routing protocols.

2.4.3.1 Proactive

Proactive ad-hoc routing protocols such as Optimized Link State Routing (OLSR) reduces delay but has higher overhead. Figure 2.4 simplifies OLSR route discovery into three stages. This visual assumes that the Multipoint Relays (MPRs) are established. First, nodes periodically exchange "HELLO" messages to populate link, neighbor and two-hop neighbor sets [20]. The optimization component of OLSR comes from reducing flooding of control messages by use of centralized MPRs that have the highest vertex degree. Once MPRs have an updated list of their neighborhood they broadcast link state messages to the nodes that includes routing tables to every other node.

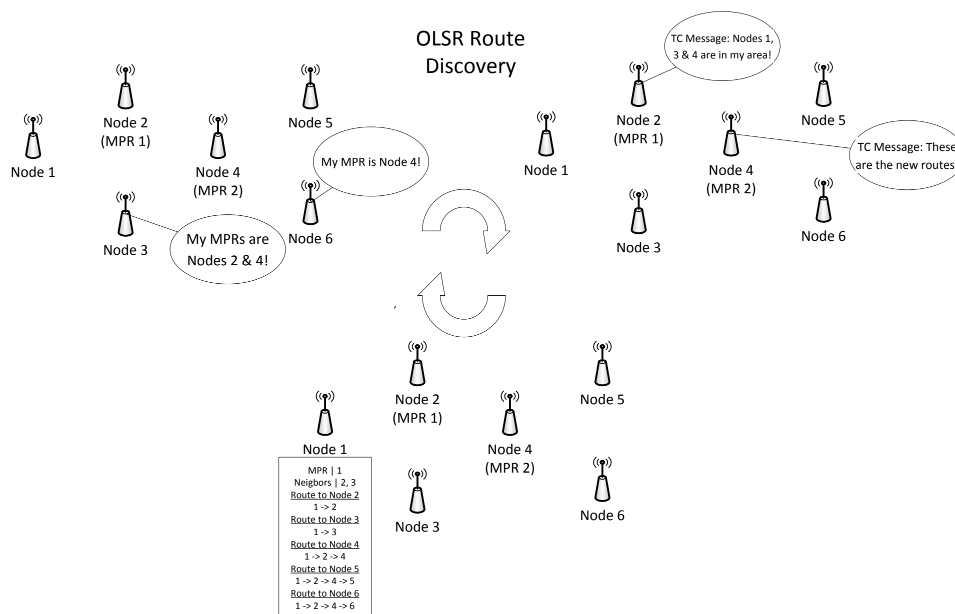

Figure 2.4: OLSR Route Discovery

2.4.3.2 Reactive

Reactive ad-hoc routing protocols minimize broadcast flooding by limiting route discovery requests to an as needed basis. Ad-hoc On-demand Distance Vector (AODV) is a common reactive routing protocol. Figure 2.5 shows a source node initiating a route request (RREQ), to communicate with the destination node. Nodes 2 and 3 are intermediate nodes and upon receiving the RREQ it creates a temporary return route table and then broadcasts the RREQ to any node within range. This re-broadcast happens until the destination node is found and a route reply (RREP) message is sent back to the source node. However, if the destination node is no longer in the network, the RREQ will expire after a period of time. The RREP is sent along the path and each intermediate node uses its return route table to pass it along. Intermediate nodes often move or drop out of the network during communication sessions which creates a link failure error. Link failure notifications are sent to neighbors within range and this is repeated until the source receives it and begins the route discovery procedure again.

2.4.3.3 Hybrid

In [21], an on demand spectrum-tree based routing protocol is presented which is a hybrid protocol that uses proactive to establish and maintain adjacent channel or subnet connections and reactive for intranet communications. This method establishes a root upon initialization that periodically sends an in-channel (spectrum band) announcement message telling other nodes within the same channel to connect. After children connect with the root they update tables such as a cost metric determined by the quality of service (QoS). This same metric determines if it is worth another node to connect directly to the root or to connect through an intermediate node. Every time a node joins the network the announcement message is changed to let the rest of the network know what the configuration is

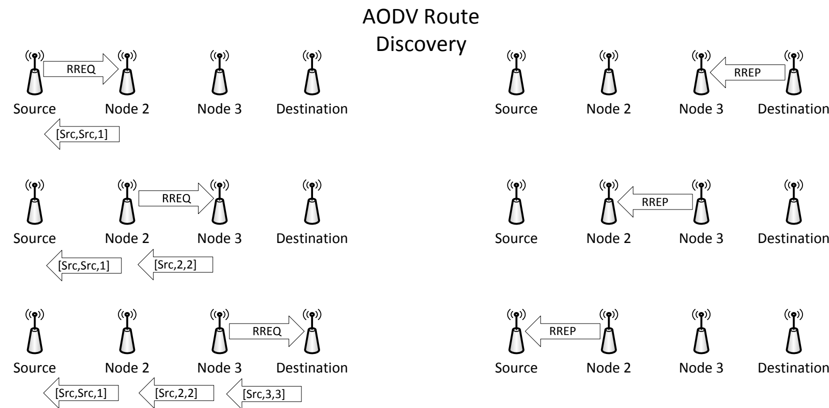

Figure 2.5: AODV Route Discovery

so that each of them can update the cost metric. If the PU changes status to ″on″, the root sends a message telling the others what channel to change to. Also, when a node needs to communicate with a node in an adjacent channel it sends a request to send and the intermediate nodes calculate the cheapest path to a gateway node that sends it to the gateway node in the adjacent channel. A simple example of a tree based network formation is shown in figure 2.6 where the roots are chosen by highest degree and channel availability.

2.4.4 Static networks

WSNs are often static. Proactive routing protocols work well with these because routing table and topology updates happen less frequent which keeps overhead low. Proactive routing has many benefits such as low latency but overhead is a

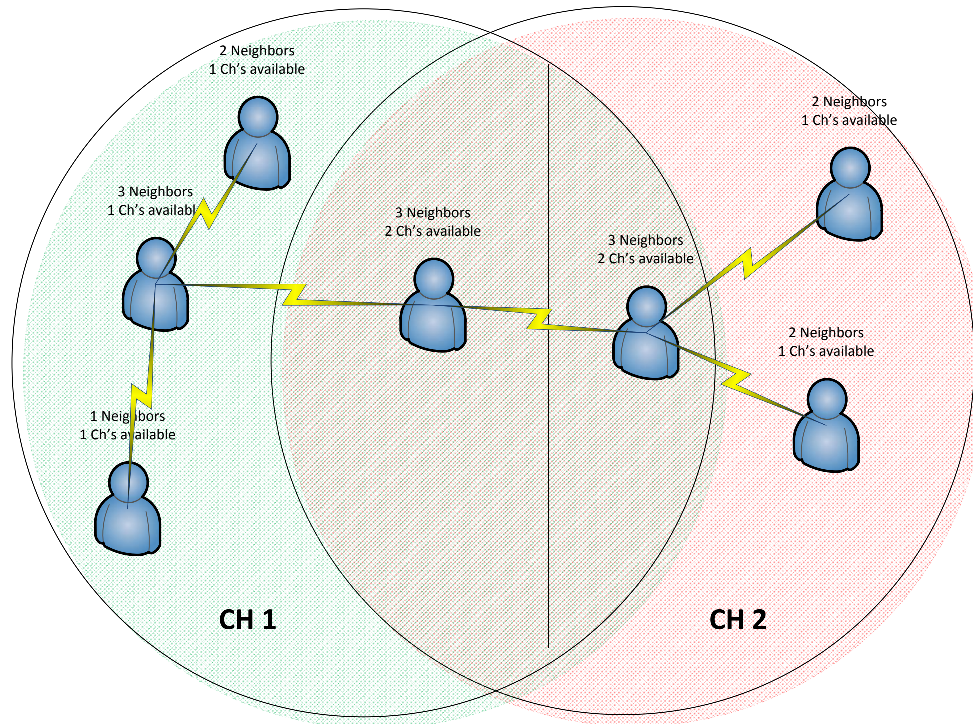

Figure 2.6: Tree Formation

limiting factor as stated in [22]. Not all WSNs are ad-hoc networks but some are and they commonly are static. Often these networks only change when a new node in installed or another drops out for maintenance issues such as battery replacement.

2.4.5 Dynamic networks

Typical networks that are commonly dynamic are VANETs and MANETs. Complete path routing tables are not always practical when the topology is constantly changing. Therefore, proactive routing is not always the optimal choice but reactive is instead. One highly dynamic example would be a high speed VANET such as discussed in [23] where communication links are of very short duration and network density varies greatly.

Chapter 3

Experimentation of AC-MWN

3.1 Equipment Setup

The testbed consists of three laptops, two USRP-N210 SDRs, and one USRP2 SDR with USRP being an abbreviation for Universal Software Radio Peripheral. Each laptop is running Ubuntu version 13.10 64-bit operating system with GNU Radio software package. The radios were equipped with firmware and WBX-120 RF daughterboards like the ones shown in Figure 3.1.

The USRP itself handles the digital signal processing (DSP) such as analog to digital conversion (ADC) and digital to analog conversion (DAC) using a field programmable grid array (FPGA). Also, the USRP has physical interfaces such as Ethernet, multiple-in multiple-out (MIMO), and RF Daughterboard like the WBX. Enabling RF capabilities the end user must buy and install a RF Daughterboard onto the USRP main board chassis and connect it via MCX-Bulkhead cables (MCX-M to SMA-F connectors) as shown in Figures 3.2 and 3.3.

Antennas were chosen after selecting a USRP and RF Daughterboard to meet the needs of project. Just like choosing the RF Daughterboard the end user must know the frequency range that will be used. The VERT900 Antenna shown in

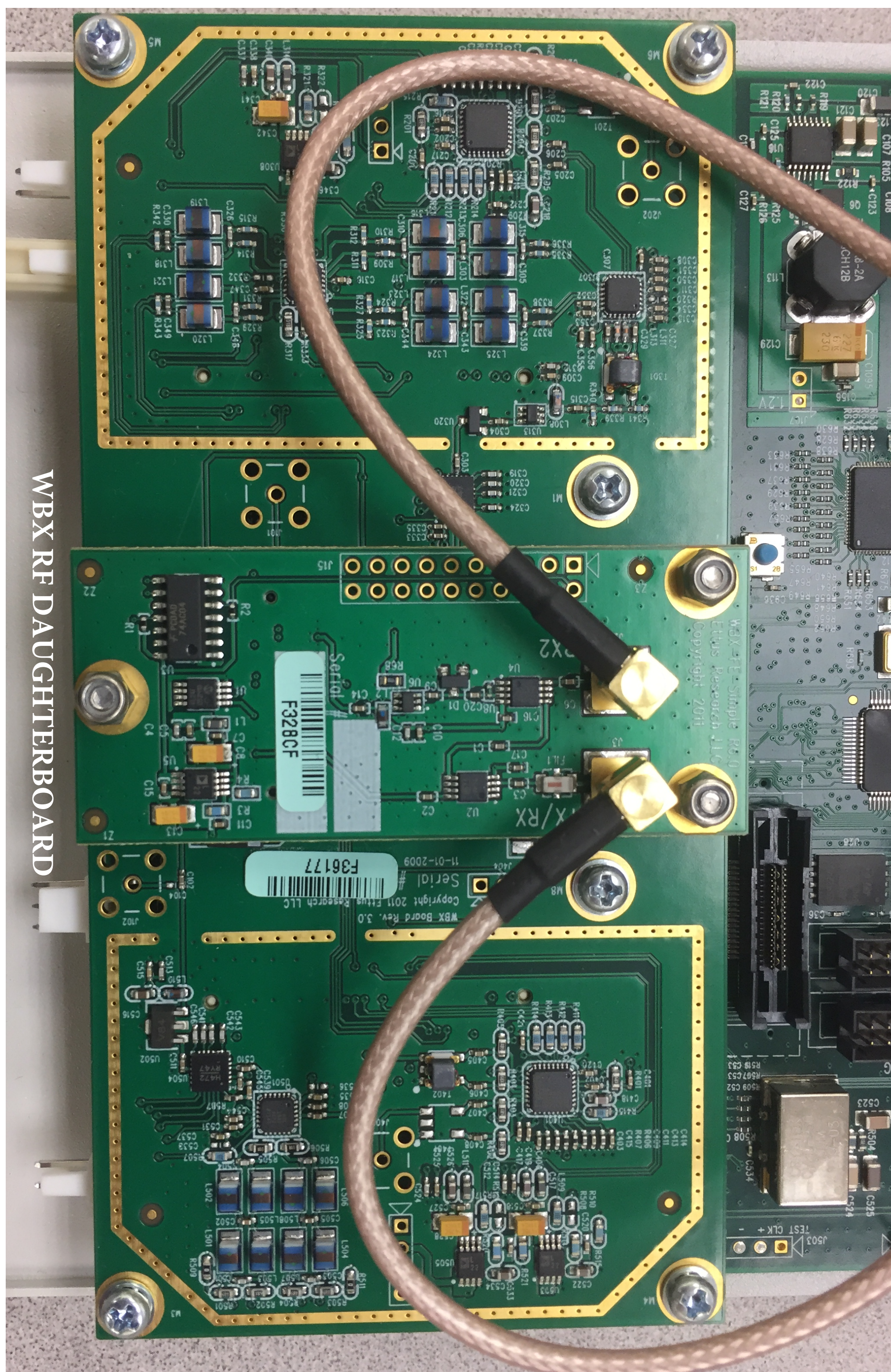

Figure 3.1: WBX Daughterboard

Figure 3.4 has a low band from 824 to 960 MHz which covers the high end of IEEE 802.22a-2014 Regional Area Network (RAN) standard which is 54 to 862 MHz [24]. Likewise, the WBX RF Daughterboard has a range of 50 MHz - 2.2 GHz. Another consideration was the channel bandwidth (BW) needs which is a minimum of 6 MHz for television broadcast. The WBX has a maximum BW of 40 MHz which exceeds our needs. Finally, Figure 3.5 shows the hardware after the WBX is installed, antennas screwed on and enclosure secured to the USRP N210.

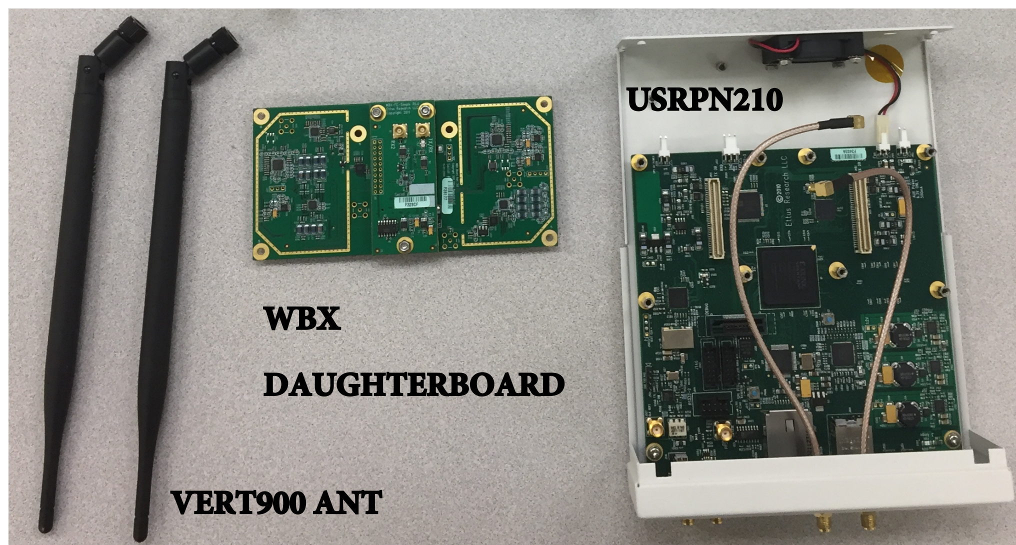

Figure 3.2: WBX Daughterboard, USRP N210 and VERT900 Antennas

3.1.1 Software Defined Radio (SDR) configuration

There are three main things that must be done to setup the system so that the USRP can be interfaced with GNU Radio and GNU Radio Companion (GRC). These include:

- Download GNU Radio software and packages, once installation is complete use gnuradio-companion to run the graphical programming interface of GNU Radio.
- Download required packages for building the USRP Hardware Driver (UHD).

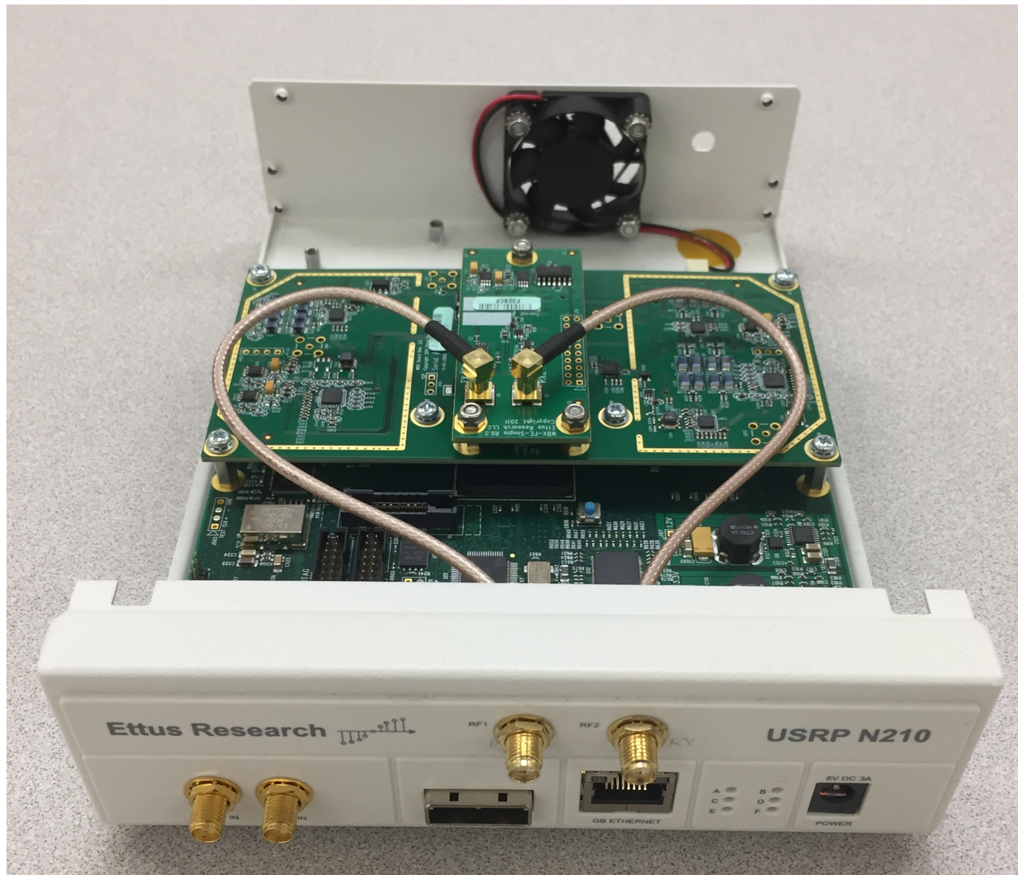

Figure 3.3: USRP N210 with installed WBX RF Daughterboard

- Configure the network interface by assigning a static IP address to the host computer Ethernet interface card connected to the USRP2 with $ifconfig\ eth0$ 192.168.10.1 and finally to test connection with $ping$ 192.168.10.2.

Figure 3.4: VERT900 Antennas

Figure 3.5: USRP N210 with two VERT900 atnennas

Chapter 4

Basic Testing

Global population is increasing and many rural communities populations are dramatically decreasing because of migration to urban areas. An estimated 66 percent of the population will live in urban areas by mid-century [25]. Population density of our cities will continue to rise which increases the demand for bandwidth. Considering our limited radio frequency (RF) spectrum, researchers have been looking toward cognitive radio networking (CRN) as a temporary solution to the overcrowding. CRN utilizes idle RF channels for limited amounts of time to relieve congestion on other channels. These channels are owned by the license holder which is known as the primary user (PU). An example of a PU would be a broadcast television station that airs sub channels in different languages, but only has enough programming to broadcast on that channel for twelve hours a day or only during peak hours. These researchers need testbeds to test and analyze their algorithms; and software defined radios (SDR) are the tool of choice because of the low cost and flexibility. Universal Software Radio Peripheral (USRP) radios are interoperable with GNU Radio programming software which includes hardware drivers specifically for USRPs.

Cognitive radio networking (CRN) is a promising technology based on spec-

trum sensing and opportunistic spectrum access which can improve the spectrum efficiency of wireless networks [26]. The secondary users would sense the activities of primary users periodically according to Federal Communications Commission (FCC) [27]. When a channel is not occupied by a primary user, a secondary user can use the channel opportunistically after sensing. The major characteristic of cognitive radios is that the activities of primary users change dynamically and channel availability changes from time to time. Thus, the secondary users have to efficiently sense the activities of the primary users and make decisions on which channel to use in the dynamic environment. In order to study the performance of the cognitive radios, we investigate how to develop a testbed and implement algorithms on it.

Multi-hop cognitive radio networks bring many challenges in testbed design. First, transmissions on different hops might interfere with each other. Therefore, we investigate how to implement multi-hop transmission on different channels using USRP2. Second, secondary users have to sense the activities of the primary users instantly to get the availability information of the channels. Thus, we need to develop a practical sensing scheme to find whether the primary users are working or not. After one available channel is found, the secondary users have to switch to this channel as soon as possible. Third, the routing and channel allocation schemes have to be embedded on testbed. We need to implement our routing and allocation schemes on testbed to show the effectiveness of our algorithms.

To prove the theoretical performance in multi-hop cognitive radio networks, we develop a testbed in this project. First, we set up the system so that the USRP can be interfaced with GNU Radio. Second, we study several different transmission schemes including single-hop and multi-hop transmissions based on USRP and smart radios. Then, we investigate how to perform spectrum sensing using cognitive radios. Finally, we implement our routing and channel allocation

schemes on our testbed. Our testbed is developed as the following.

In this chapter, we set up a testbed for the AC-MWN to implement the basic functions of cognitive radio and the schemes. Moreover, we implement a video stream application in AC-MWN based on the testbed setup. The testbed consists of three laptops, two Universal Software Radio Peripheral (USRP) USRP2-N210 software defined radios (SDR) [28], and one USRP2 SDR. The SDRs are capable of transmitting in both 824-960 MHz and 1710-1990 MHz spectrum. Most of the work done was in the lower band. Each laptop operates as signal generator and processor. The laptops are identically configured, running Ubuntu [29] version 13.10 64-bit operating system with GNU Radio and GStreamer software packages.

The main tasks of this REU project include: Computer programming and network configuration using Universal Software Radio Peripheral (USRP) hardware; and performance evaluation through simulations and measurements. Using this technology we will complete the followings:

- Basic functions

  Naive one-way transmission

  Naive half-duplex transmission

  Multi-channel one-way transmission

  Naive full-duplex transmission

  Adaptive one-way transmission

  Adaptive half-duplex transmission

- Sensing

  Sensing if primary user has activity at a given channel

- Cognitive radio technique application

  Adaptive channel switching

- Multi-hop applications

  File transferring, voice chat, and video conference applications using cognitive radio technique

4.1 Single-hop Transmission

4.1.1 Half-duplex one-way transmission

In this part, we are going to set up one radio as the transmitter and one for receiving, and successfully send and receive verifiable data. The data is verified at the receiving end when benchmark_rx.py prints the message and how many successful packets were received.

Two smart radios, each equips one antenna. One smart radio is set up for transmitting, the other for receiving, single dedicated channel (e.g., FM radio).

We successfully achieved this milestone by transmitting and receiving based on frequencies, by packages transfer and data transfer. We used benchmark code under our gnuradio folder to accomplish these tasks.

- First we composed a .odb file with the sentence "It is a beautiful day!!". The .odb file extension is associated with OpenOffice in Ubuntu, which we saved in the same folder of our benchmark code ($usr/local/share/gnuradio/examples/digital/ofdm$).

- Then we modified the benchmark_tx.py and benchmark_rx.py code to conform as transmitting and receiving a data file.

- And finally we executed the scripts below from the transmission terminal and reception terminal:
  $./benchmark_tx.py-f900M--from-file=nice.odb$
  $./benchmark_rx.py-f900M$

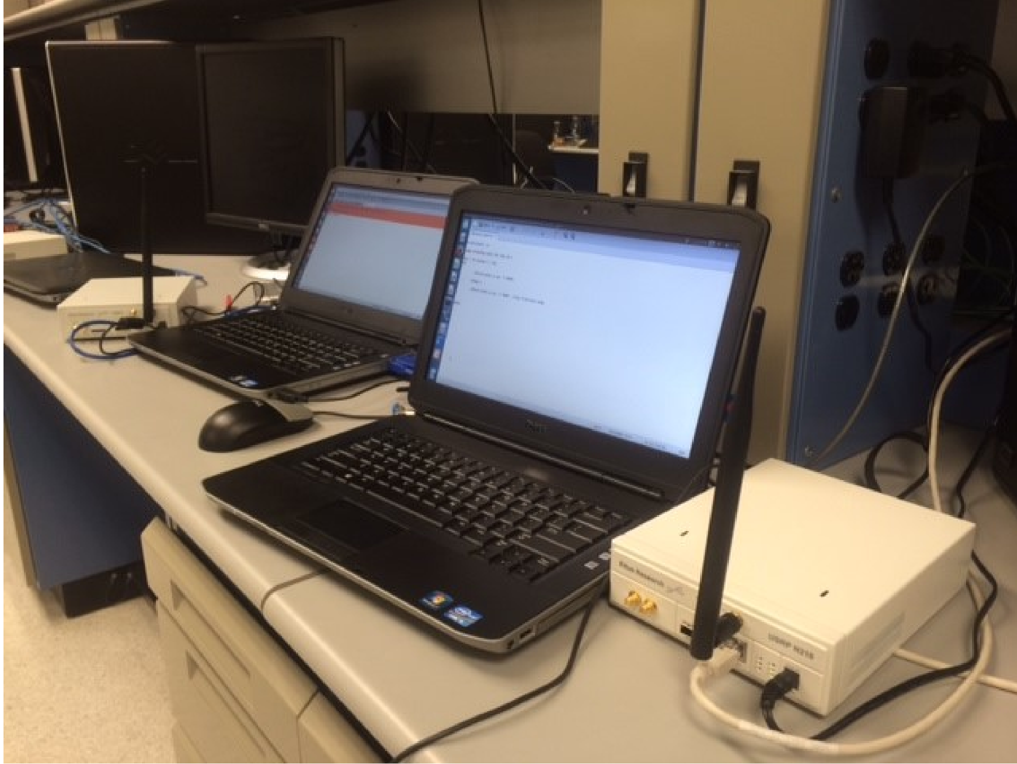

Figure 4.1: Two USRPs with One Antenna

```
user@PKI346-LT03: ~        user@PKI346-LT03: /usr/local/share/gnuradio/examples/digital/ofdm_mod
    Unable to set the thread priority. Performance may be negatively affected.
    Please see the general application notes in the manual for instructions.
    EnvironmentError: OSError: error in pthread_setschedparam

No gain specified.
Setting gain to 15.500000 (from [0.000000, 31.000000])
-- Loaded /home/user/.uhd/cal/tx_iq_cal_v0.2_F36177.csv
Using Volk machine: avx_64_mmx_orc
Warning: failed to enable realtime scheduling
.........U......................................................................................................................................
................................................................................................................................................
................................................................................................................................................
................................................................................................................................................
................................................................................................................................................
................................................................................................................................................
................................................................................................................................................
................................................................................................................................................
.......................................................................................................................^Z
[1]+  Stopped                 ./benchmark_tx.py -f 900M
user@PKI346-LT03:/usr/local/share/gnuradio/examples/digital/ofdm_mod$ ./benchmark_tx.py -f 900M
linux; GNU C++ version 4.8.1; Boost_105300; UHD_003.005.003-87-g8f4000ff

-- Opening a USRP2/N-Series device...
-- Current recv frame size: 1472 bytes
-- Current send frame size: 1472 bytes

UHD Warning:
    Unable to set the thread priority. Performance may be negatively affected.
    Please see the general application notes in the manual for instructions.
    EnvironmentError: OSError: error in pthread_setschedparam

No gain specified.
Setting gain to 15.500000 (from [0.000000, 31.000000])
-- Loaded /home/user/.uhd/cal/tx_iq_cal_v0.2_F36177.csv
Using Volk machine: avx_64_mmx_orc
Warning: failed to enable realtime scheduling
.........U.....................................................................................................................................^Z
[2]+  Stopped                 ./benchmark_tx.py -f 900M
user@PKI346-LT03:/usr/local/share/gnuradio/examples/digital/ofdm_mod$
```

Figure 4.2: benchmark_tx_py Terminal Initialization

```
user@PKI346-LT02: ~                    user@PKI346-LT02: /usr/local/share/gnuradio/examples/digital/ofdm
ok: True        pktno: 56       n_rcvd: 57      n_right: 56
ok: True        pktno: 57       n_rcvd: 58      n_right: 57
ok: True        pktno: 58       n_rcvd: 59      n_right: 58
ok: True        pktno: 59       n_rcvd: 60      n_right: 59
ok: True        pktno: 60       n_rcvd: 61      n_right: 60
ok: True        pktno: 61       n_rcvd: 62      n_right: 61
ok: True        pktno: 62       n_rcvd: 63      n_right: 62
ok: True        pktno: 63       n_rcvd: 64      n_right: 63
ok: True        pktno: 64       n_rcvd: 65      n_right: 64
ok: True        pktno: 65       n_rcvd: 66      n_right: 65
ok: True        pktno: 66       n_rcvd: 67      n_right: 66
ok: True        pktno: 67       n_rcvd: 68      n_right: 67
ok: True        pktno: 68       n_rcvd: 69      n_right: 68
ok: True        pktno: 69       n_rcvd: 70      n_right: 69
ok: True        pktno: 70       n_rcvd: 71      n_right: 70
ok: True        pktno: 71       n_rcvd: 72      n_right: 71
ok: True        pktno: 72       n_rcvd: 73      n_right: 72
ok: True        pktno: 73       n_rcvd: 74      n_right: 73
ok: True        pktno: 74       n_rcvd: 75      n_right: 74
ok: True        pktno: 75       n_rcvd: 76      n_right: 75
ok: True        pktno: 76       n_rcvd: 77      n_right: 76
ok: True        pktno: 77       n_rcvd: 78      n_right: 77
ok: True        pktno: 78       n_rcvd: 79      n_right: 78
ok: True        pktno: 79       n_rcvd: 80      n_right: 79
ok: True        pktno: 80       n_rcvd: 81      n_right: 80
ok: True        pktno: 81       n_rcvd: 82      n_right: 81
ok: True        pktno: 82       n_rcvd: 83      n_right: 82
ok: True        pktno: 83       n_rcvd: 84      n_right: 83
ok: True        pktno: 84       n_rcvd: 85      n_right: 84
ok: True        pktno: 85       n_rcvd: 86      n_right: 85
ok: True        pktno: 86       n_rcvd: 87      n_right: 86
ok: True        pktno: 87       n_rcvd: 88      n_right: 87
ok: True        pktno: 88       n_rcvd: 89      n_right: 88
ok: True        pktno: 89       n_rcvd: 90      n_right: 89
ok: True        pktno: 90       n_rcvd: 91      n_right: 90
ok: True        pktno: 91       n_rcvd: 92      n_right: 91
ok: True        pktno: 92       n_rcvd: 93      n_right: 92
ok: True        pktno: 93       n_rcvd: 94      n_right: 93
user@PKI346-LT02:/usr/local/share/gnuradio/examples/digital/ofdm$
```

Figure 4.3: benchmark_tx_py Terminal Running Output

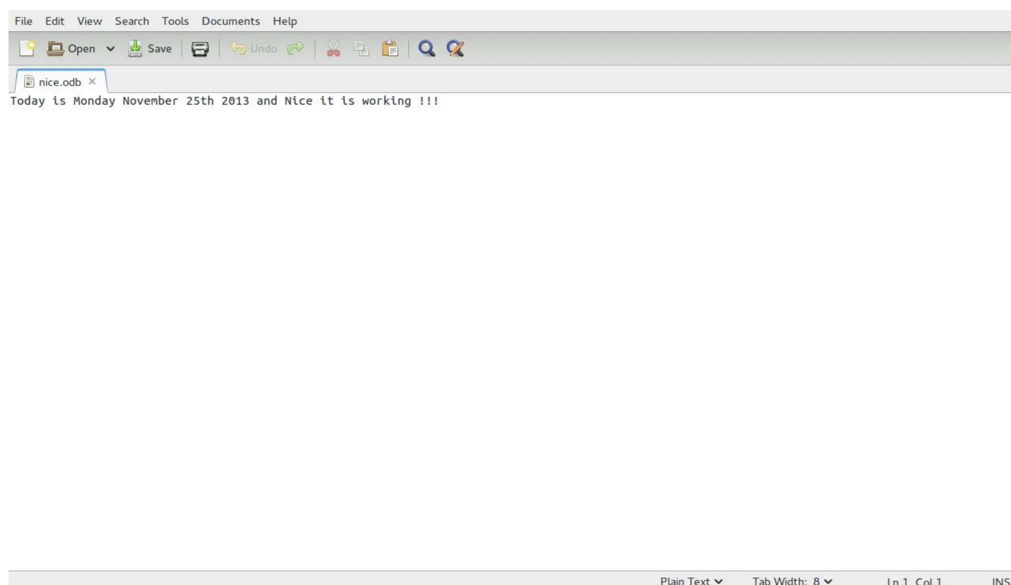

Figure 4.4: benchmark_tx_py Output to Document

4.1.2 Adaptive One-Way Transmission

In this part, we are going to have two smart radios, each equipped with one antenna. First, one smart radio is set up for transmitting and the other for receiving. Then a single dedicated channel that can be switched according to an

```
user@PKI346-LT03: ~          user@PKI346-LT03: /usr/local/share/gnuradio/examples/digital/ofdm_mod
No gain specified.
Setting gain to 15.500000 (from [0.000000, 31.000000])
Warning: failed to enable realtime scheduling
.U-- Opening a USRP2/N-Series device...
-- Current recv frame size: 1472 bytes
-- Current send frame size: 1472 bytes

UHD Warning:
    Unable to set the thread priority. Performance may be negatively affected.
    Please see the general application notes in the manual for instructions.
    EnvironmentError: OSError: error in pthread_setschedparam

No gain specified.
Setting gain to 15.500000 (from [0.000000, 31.000000])
Warning: failed to enable realtime scheduling
.Uuser@PKI346-LT03:/usr/local/share/gnuradio/examples/digital/ofdm_mod$ ./benchmark_tx.py -f 900M --from-file=nice.odb
linux; GNU C++ version 4.8.1; Boost_105300; UHD_003.005.003-87-g8f4000ff

-- Opening a USRP2/N-Series device...
-- Current recv frame size: 1472 bytes
-- Current send frame size: 1472 bytes

UHD Warning:
    Unable to set the thread priority. Performance may be negatively affected.
    Please see the general application notes in the manual for instructions.
    EnvironmentError: OSError: error in pthread_setschedparam

No gain specified.
Setting gain to 15.500000 (from [0.000000, 31.000000])
-- Loaded /home/user/.uhd/cal/tx_iq_cal_v0.2_F36177.csv
Using Volk machine: avx_64_mmx_orc
Warning: failed to enable realtime scheduling
.U-- Opening a USRP2/N-Series device...
-- Current recv frame size: 1472 bytes
-- Current send frame size: 1472 bytes

UHD Warning:
    Unable to set the thread priority. Performance may be negatively affected.
    Please see the general application notes in the manual for instructions.
```

Figure 4.5: benchmark_tx_py Terminal Initialization with File Sink to Document

```
Today is Monday November 25th 2013 and Nice it is working !!!

ok: True        pktno: 0        n_rcvd: 1        n_right: 1
```

Figure 4.6: benchmark_rx_py Received Text Saved to Document

algorithm automatically (e.g., FM radio, but will change channel once every 10 seconds). Settings and results are below in Figures 4.8 and 4.9. Setting: two radios with two antennas

4.2 Half-duplex two-way transmission

In this part, we are going to have one radio as transmitter and one as receiver. Radio 1 starts as the transmitter running benchmark_tx.py and will send data for five seconds while Radio 2 is receiving running benchmark_rx.py during that time. After five seconds the bash script will stop the running python file and start the other switching the role of the radio from transmitter to receiver or vice versa.

Figure 4.7: Two USRPs with Two Antennas Each

Two smart radios, each equips one antenna. Each radio takes turn to be transmitter or receiver (e.g., 5 seconds as transmitter, 5 seconds as receiver, but acting automatically)

The settings and results are shown below: We accomplished this by creating a bash script with our benchmark_tx.py and benchmark_rx.py codes:

From the 1st radio (will receive first and then will send data)

#!/*bin*/*bash*

*sudo ifconfig eth*0 192.168.10.1

for i in(seq 1 2 20) – code to be repeatedly executed

do

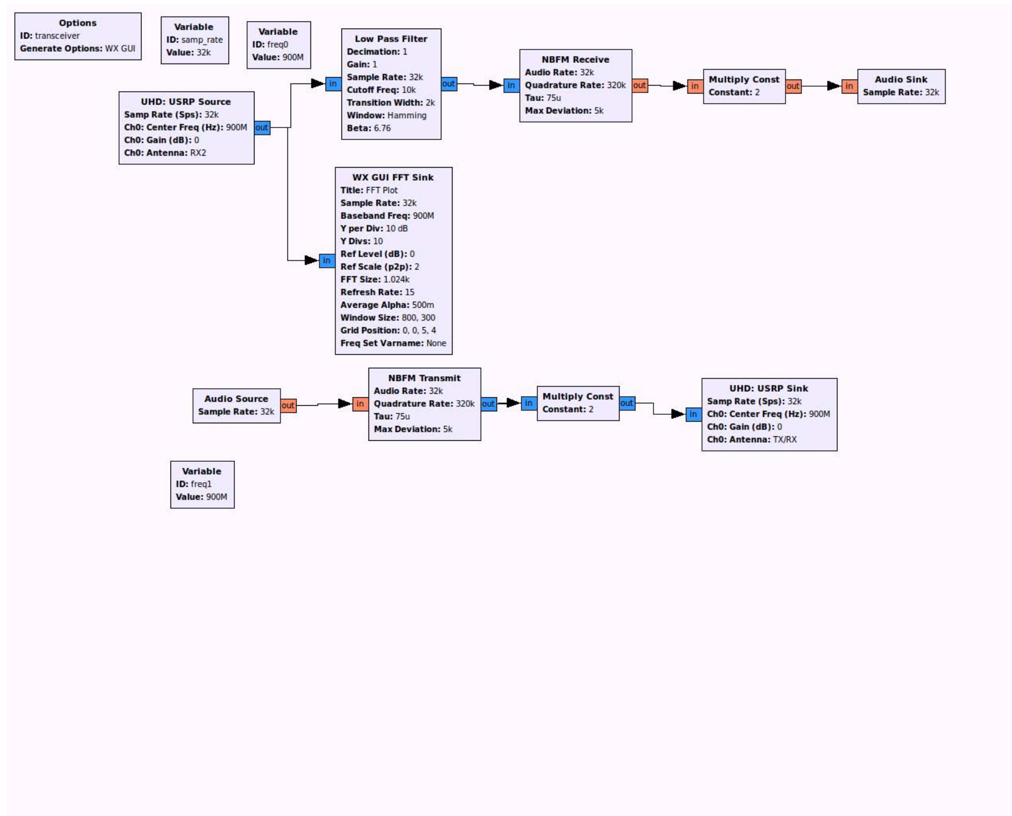

Figure 4.8: Adaptive One-way Transmit GRC Blocks

$./benchmark_rx.py\ f\ 900M$ – Receiving

$sleep$ 5; – Delay for a specified time

$killall\ benchmark_rx.py$; – Stop receiving

$./benchmark_tx.py\ f\ 900M\ from - file = nice.odb$; – Transmitting

$done$

From the 2nd radio (will transmit first then will receive a data)

#!$/bin/bash$

$sudo\ ifconfig\ eth$0 192.168.10.1

$for\ i\ in$(seq 1 2 20) – code to be repeatedly executed

do

$./benchmark_tx.py\ f\ 900M\ -from - file = nice.odb$; – Transmitting

$./benchmark_rx.py\ f\ 900M$ – Receiving

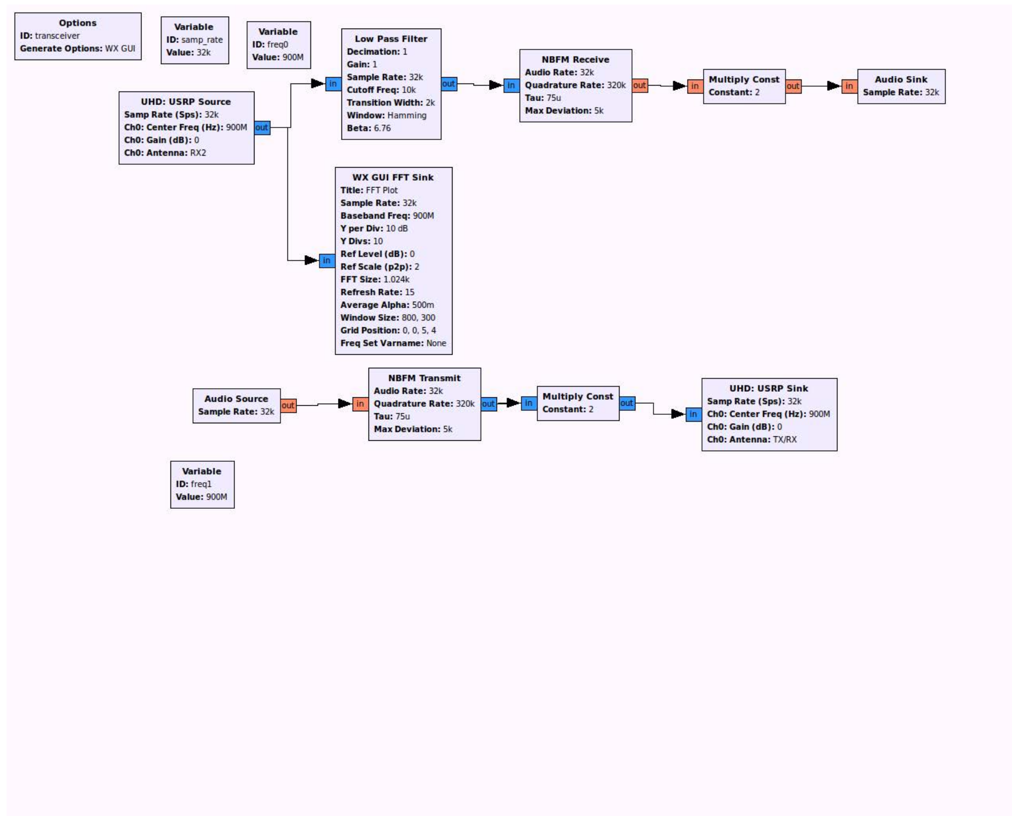

Figure 4.9: Adaptive One-way Receive GRC Blocks

sleep 5; – Delay for a specified time

killall benchmark_rx.py; – Stop receiving

done

We executed the bash script and we can see that each radio alternated transmitting and receiveing automatically. Bash script at the transmitter is shown if Figures 4.10, 4.11 and 4.12

4.3 Full-Duplex Transmission

In this part, we are going to have two radios with two antennas each. Radio 1 will transmit using the TX/RX antenna while Radio 2 will receive on the same channel using the RX2 antenna. Next, Radio 2 will be set up to transmit on a

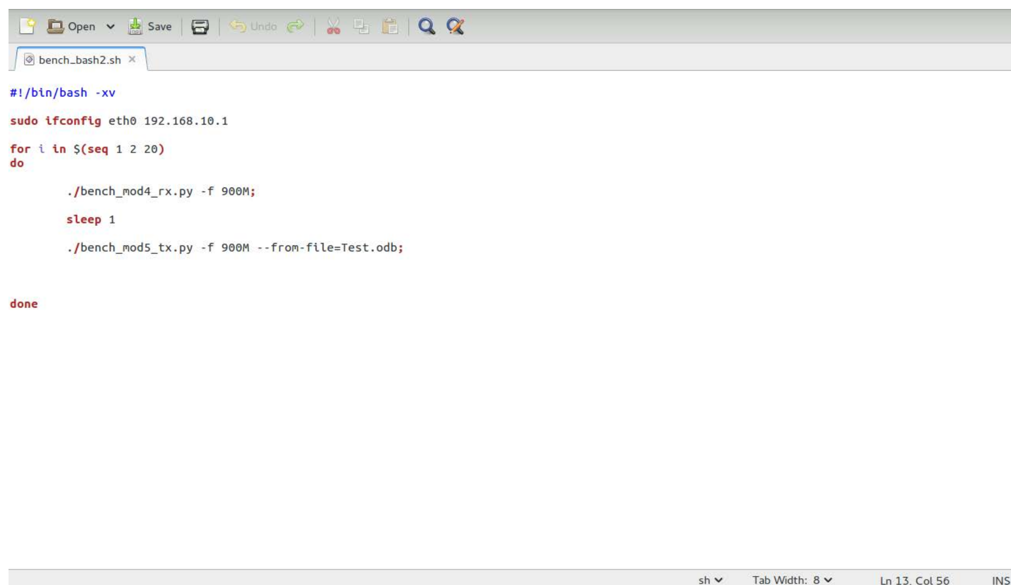

Figure 4.10: Half Duplex Two-way Transmit Bash Script

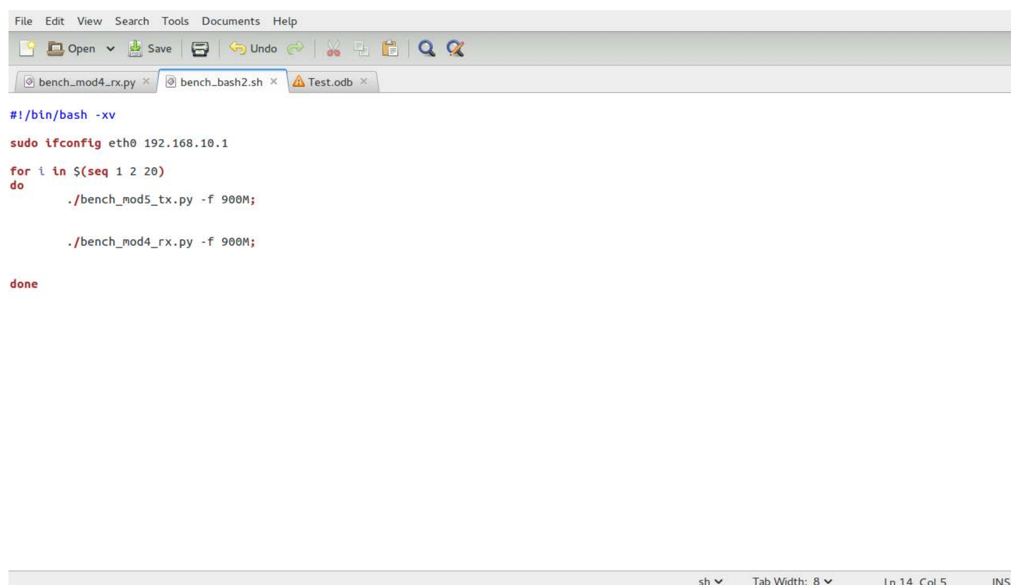

Figure 4.11: Half Duplex Two-way Receive Bash Script

different channel using its TX/RX antenna while Radio 1 receives on the same channel using the RX2 antenna.

First, one-way multi-channel testing. Two smart radios, each equipped with two antennas. One smart radio is set up for transmitting and the other for receiving. Transmission is set up using two channels through two antennas simultaneously.

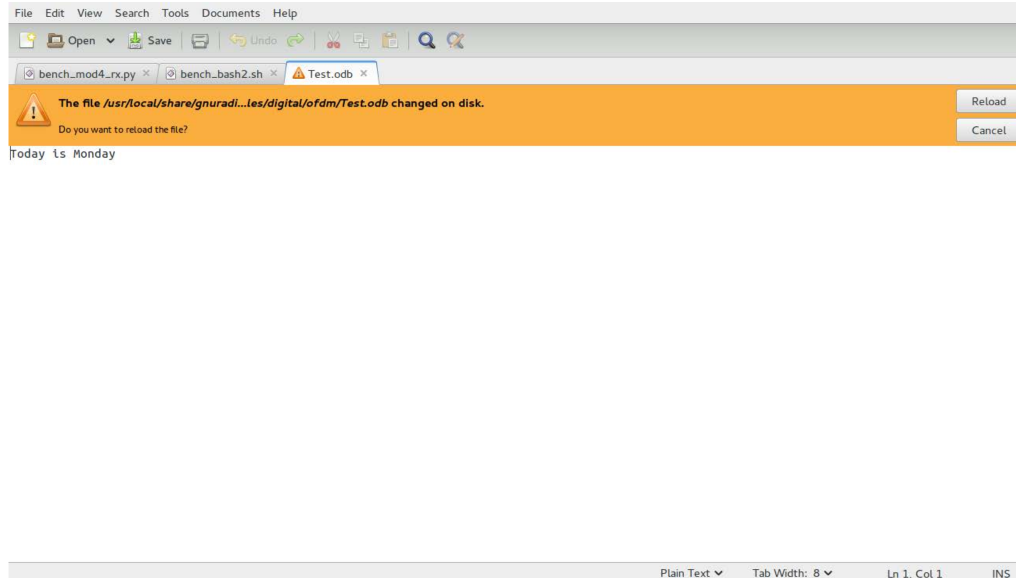

Figure 4.12: Half Duplex Two-way Receive Output to Document

We repeated the same design and procedure with the Naive one-way transmission but with two antennas.

Full-duplex communications achieved with having two smart radios, each equipped with two antennas. One antenna of a smart radio is set up for transmitting, while the other is set up for receiving. Two-way transmission is achieved.

We also worked on IO file management between the transmitting and receiving radios, and then added an acknowledgment time stamp when either one received or transmitted in the proper folder designed in advance by using the benchmark scripts. The objective was to save the data in text file after receiving in from the transmitting radio; then to resend the same data in the text file to the first radio with acknowledgment time and date of reception. Settings and results are shown below:

Transmission script: Needed to be added to the benchmark_tx script

$def\ read_dir():$

$If\ os.path.exist('''')$

$file = open('','')$ - Opens a file on the specified path with read/write access.

$ts = str(datetime.datetime.now())$ - Print the actual time.

$file.write(ts)$

$global\ ack$

$file.close()$

Reception script: Needed to be added in the benchmark_rx script.

$def\ read_dir():$

$If\ not\ os.path.exist("")$

$Os.mkdir("")$

$file = open(",")$ - Opens a file on the specified path with read/write access.

$global\ x$

$x = payload$

$file.write(x)$

$file.close()$

Setup: two radios with two antennas each shown in Figure 4.13

Odb file on transmitter side shown in Figure 4.14

4.3.1 Adaptive Full-Duplex Transmission

In this part, we are going to set up hardware as shown in Figure 4.13, but we use GRC flow chart software to utilize the built in threading capabilities. The scripts are same as Full-Duplex shown above but the flow charts shown in Figures 4.8 & 4.9 are combined to make one flow chart to create a transceiver.

Figure 4.13: Two USRPs with Two Antennas Each

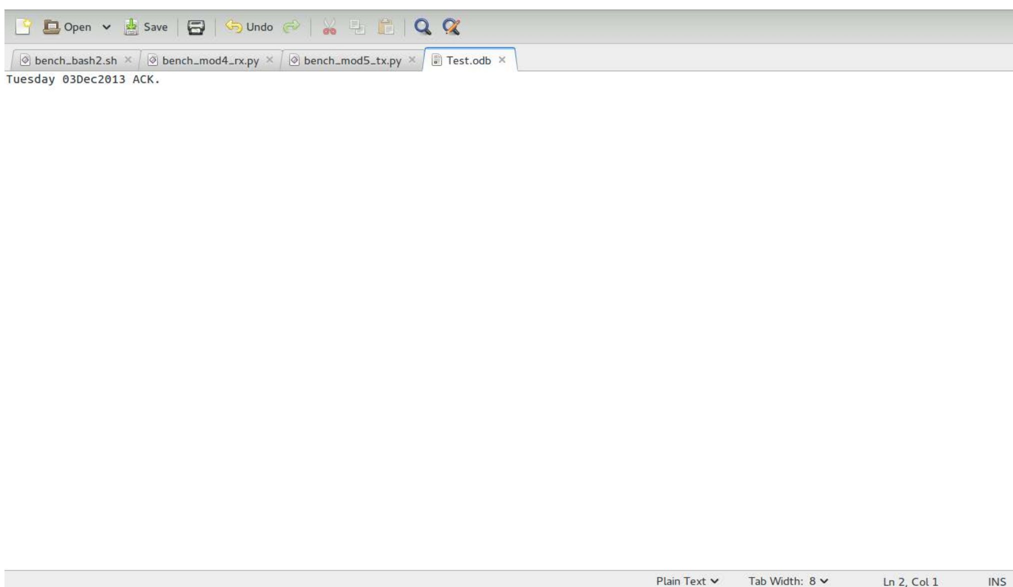

Figure 4.14: Transmit Side Document

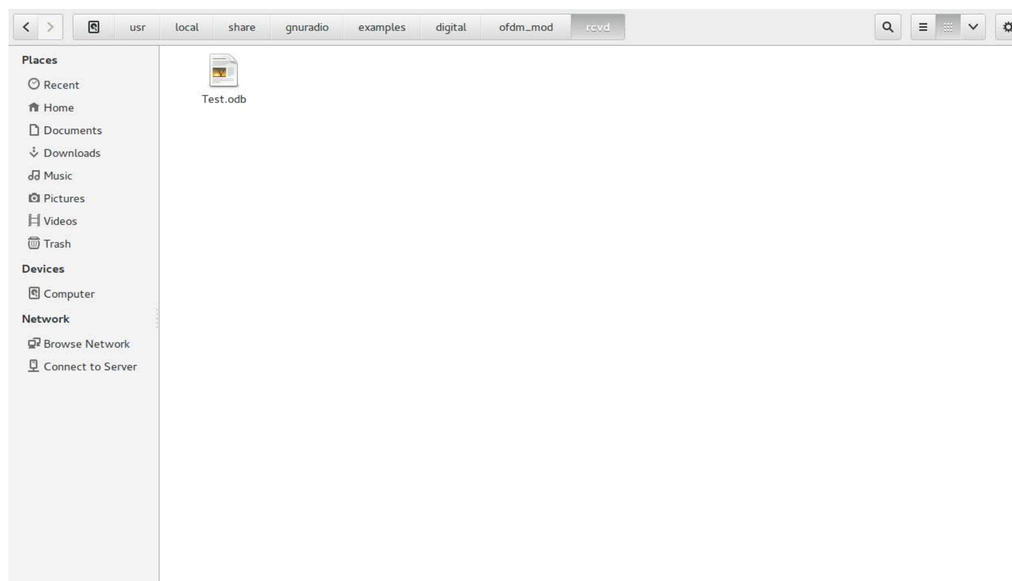

Figure 4.15: Receive Side Document Created

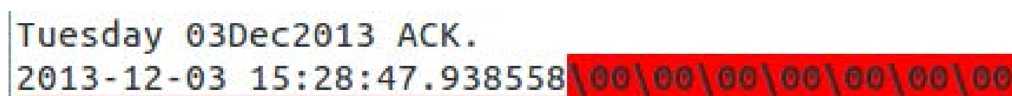

Figure 4.16: Receive Side Document Received with Time Stamp

```
^Cuser@PKI346-LT03:/usr/local/share/gnuradio/examples/digital/ofdm_mod$ ./bench_mod4_rx.py -f 900M
linux; GNU C++ version 4.8.1; Boost_105300; UHD_003.005.003-87-g8f4000ff

-- Opening a USRP2/N-Series device...
-- Current recv frame size: 1472 bytes
-- Current send frame size: 1472 bytes

UHD Warning:
    Unable to set the thread priority. Performance may be negatively affected.
    Please see the general application notes in the manual for instructions.
    EnvironmentError: OSError: error in pthread_setschedparam

No gain specified.
Setting gain to 19.000000 (from [0.000000, 38.000000])
Using Volk machine: avx_64_mmx_orc
>>> gr_fir_ccf: using SSE
>>> gr_fir_fff: using SSE
Warning: failed to enable realtime scheduling

ok: True         pktno: 0        n_rcvd: 1       n_right: 100
```

Figure 4.17: Receive Side Terminal Output with Verification

4.4 Multi-Hop Transmission

4.4.1 Simple Transceiver Transmission with Two Smart Radios

In this part, we are going to use two radios setup with the GRC file shown in Figure 4.21 and both channels are observed with the FFT GUI as shown in Figure

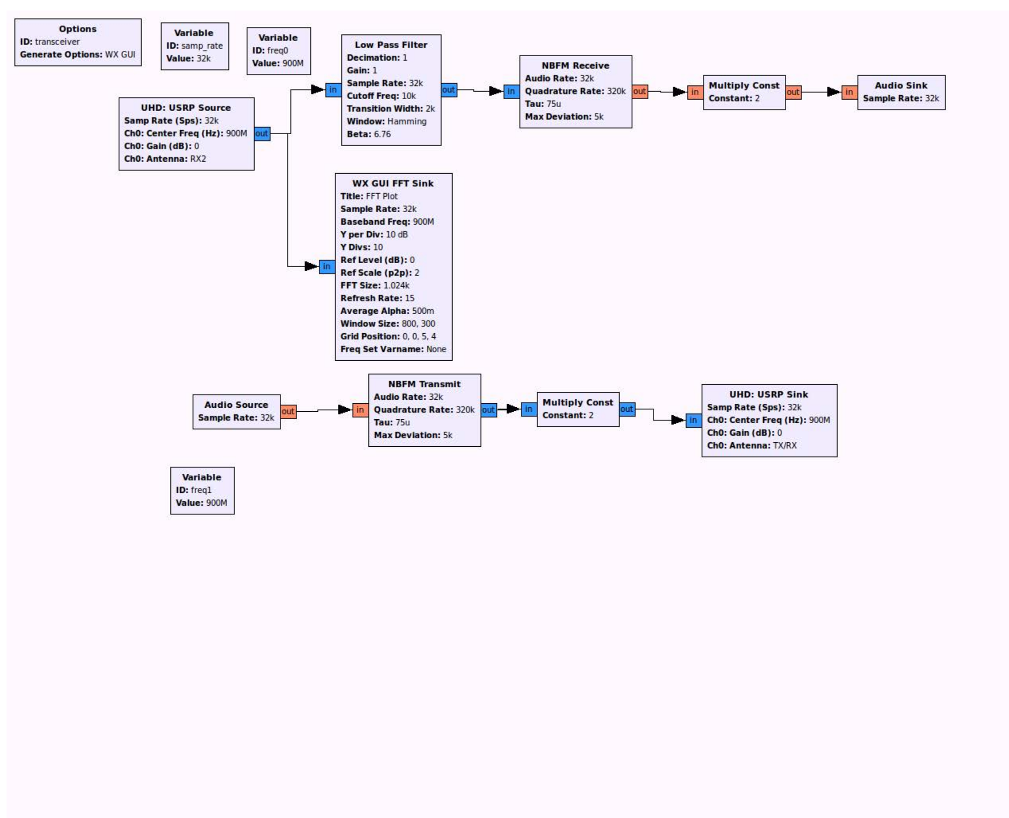

Figure 4.18: Adaptive Full Duplex GRC Configuration

4.22.

The settings are shown below: The block below was designed in GRC. It consisted of two separated blocks, one for the transmission and the other for the receiving. GNU Radio Companion components used for this design are:

Equipment configuration: One radio with one laptop as shown in Figure 4.20.

4.4.2 Two Radio Multi-Hop (Round-Trip) Transmission

In this part, we are going to use two radios, one with our regular transceiver GRC file and another with a relay file. The relay is set up to receive on one frequency and immediately transmit the data on another frequency. Therefore, the data is transmitted from Radio 1 and channel A, received by Radio 2 on channel A then transmitted by Radio 2 on channel B, and finally, received by Radio 1 on channel B.

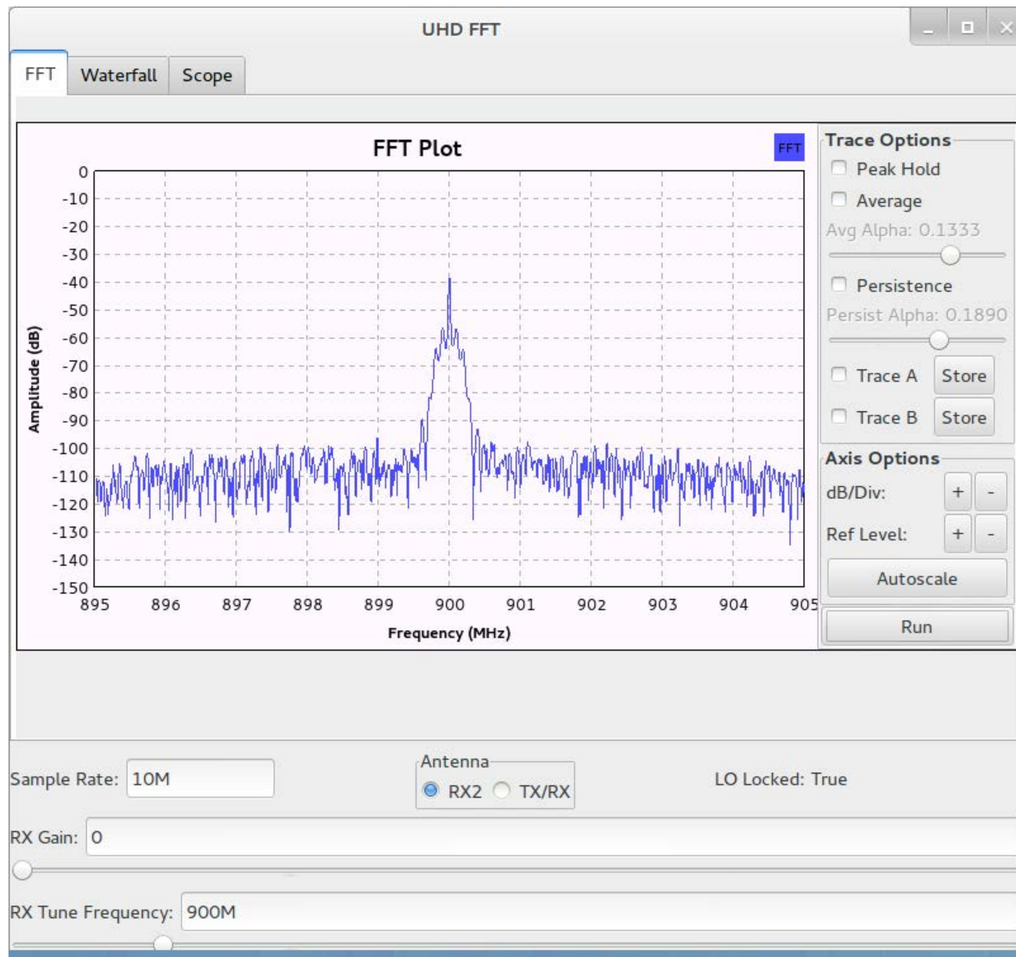

Figure 4.19: Adaptive Full Duplex FFT Plot

In this section we designed a full duplex transmission with data exchange between two radios. We used specific blocks as OFDM, File Source and File Sink. The full duplex was tested and it worked perfectly by exchanging data from the receiving radio to the transmission radio in a folder we pre-created for the circumstance. Settings and results are shown below:

Equipment configuration: two radios with two antennas and two laptops as shown in Figure 4.25

The first radio sent data information with two sets of blocks in GRC companion (a receiver and transmitter blocs) as shown in Figure 4.26. Data received from the second system is shown in Figure 4.27. The second system received the data,

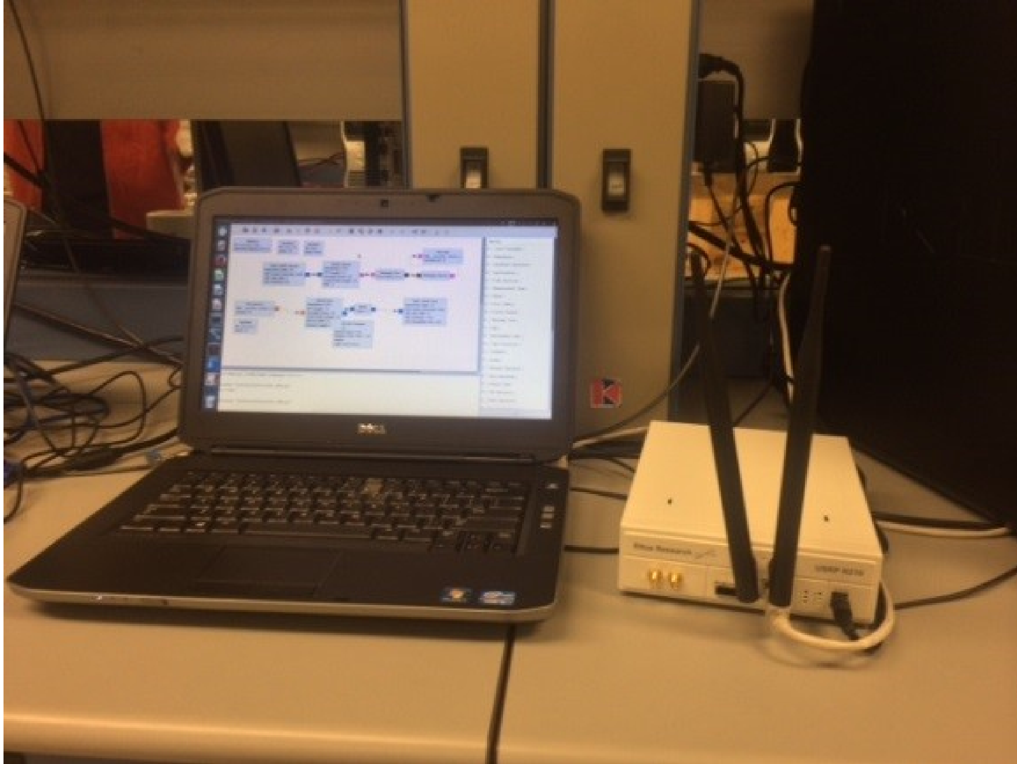

Figure 4.20: One USRP with Two Antennas and One Laptop

saved and resent it to the first system creating the loop. Figure 4.28 shows the GRC configuration for the second system, and Figure 4.29 shows the output.

Radio 1:

Transmission frequency: 950 MHz

Receiving frequency: 900 MHz

File source:

We created an ofdm_tx folder with the sentence PKI OPEN HOUSE on directory /home/user/ofdm_tx

File sink (where the file is received after the second radio resent what it received.):

/home/user/ofdm_rx

Radio 2:

Transmission frequency: 900 MHz

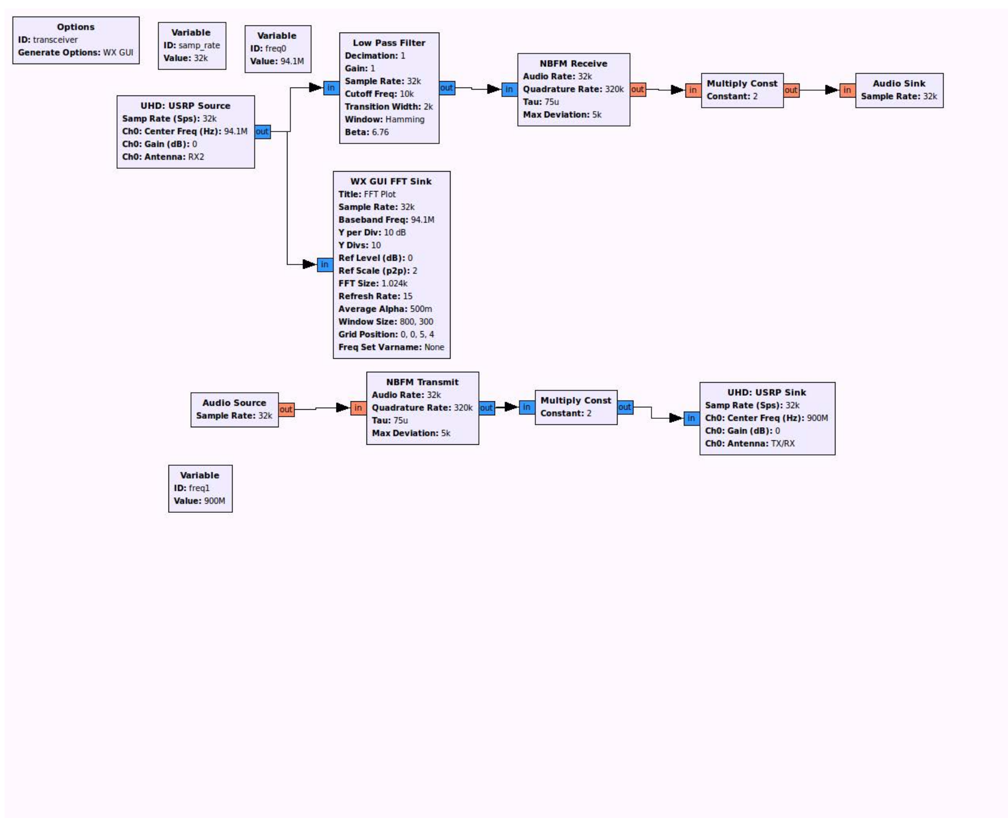

Figure 4.21: Multi-hop Transceiver GRC Configuration

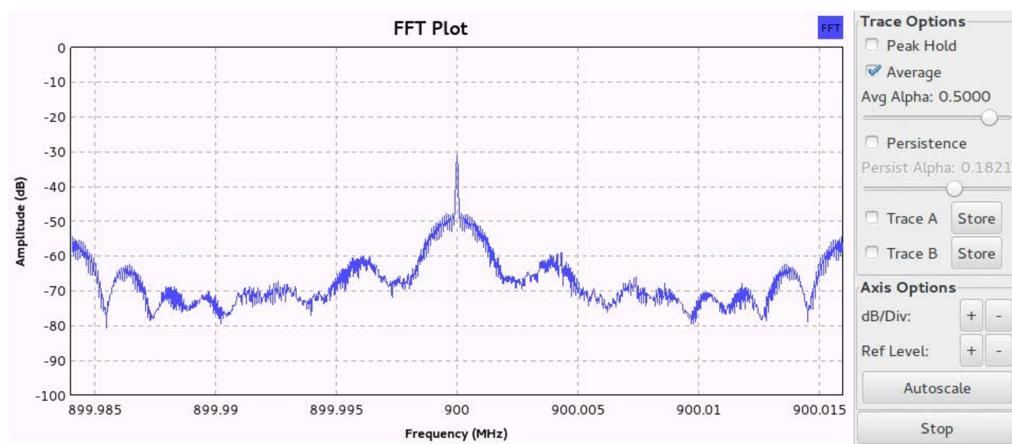

Figure 4.22: FFT plot of Transceiver block design with 1 MHz BW

Receiving frequency: 950MHz

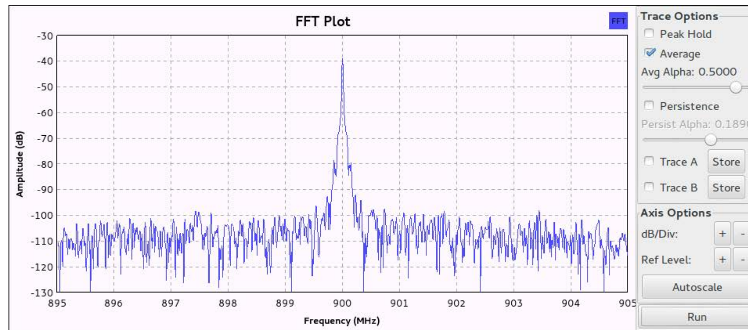

Figure 4.23: FFT plot of Transceiver block design with 10 Mhz BW at 900 MHz

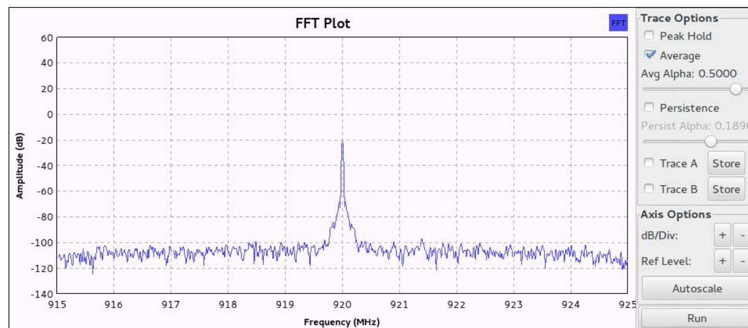

Figure 4.24: FFT plot of Transceiver block design with 10 Mhz BW at 920 MHz

4.4.3 Three Radio Multi-Hop (Circular) Transmission

In this part, we are going to use the same design for the simple loop transmission between 2 radios. The first radio was set on the frequency of 950 MHz to transmit and 900 MHz to receive. The second radio receives at 950 MHz but retransmits the

Figure 4.25: Two USRP radio with two antennas and two laptops each

same information at 830 MHz and the third radio receives at the frequency of 830 MHz but retransmits the same data to the first radio on the frequency of 900MHz. The design was tested and it worked successfully. Settings and results are shown below:

Setting: Three radios, each with one laptop as shown in 4.30

Radio 1:

Transmission frequency: 950MHz

Receiving frequency: 900MHz

Radio 2:

Transmission frequency: 830MHz

Receiving frequency: 950MHz with FFT shown in Figure 4.34

Figure 4.26: Multi-hop Transceiver GRC Configuration - Endpoint 1

Figure 4.27: Multi-hop Data Saved to Document Received by Relay

Figure 4.28: Multi-hop Transceiver GRC Configuration - Relay

Radio3:

Transmission frequency: 900 MHz

Receiving frequency: 830MHz with FFT shown in Figure 4.33

Radio 3: Design Template and output shown in Figures 4.31 & 4.32

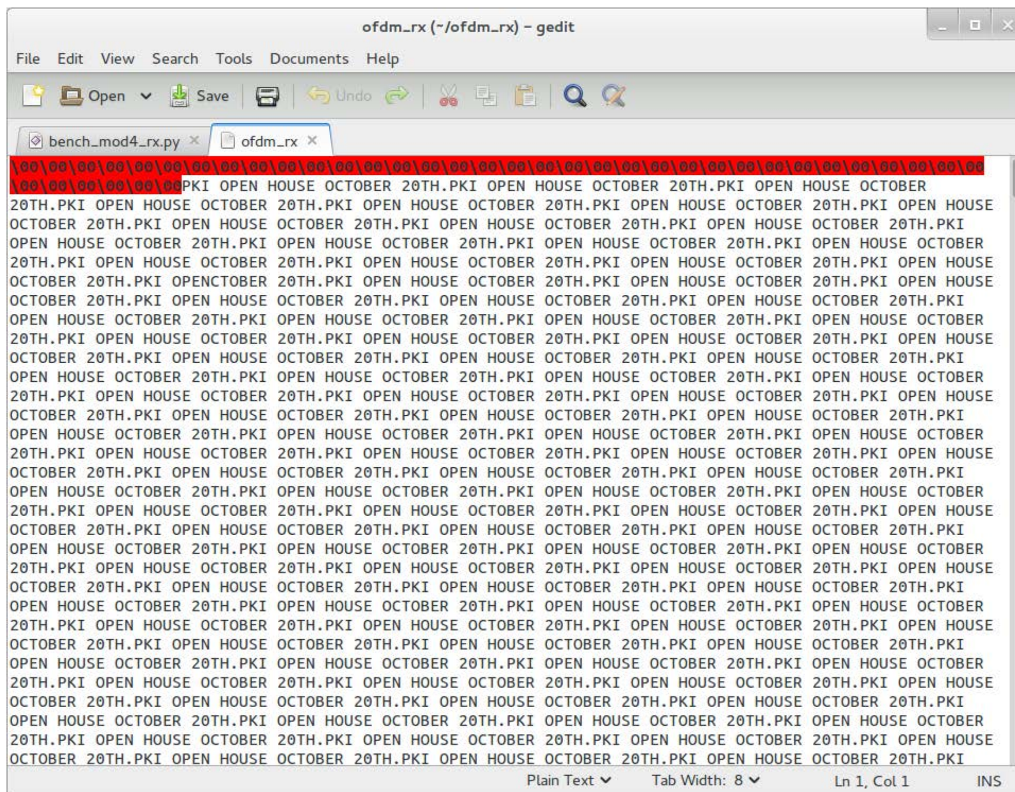

Figure 4.29: Multi-hop Data Saved to Document Received by Endpoint 2

Figure 4.30: Three USRPs with Two Antennas and Two Laptops Each

Figure 4.31: Circular transmission, Radio 3, GRC configuration

Figure 4.32: Circular transmission, Radio 3, output

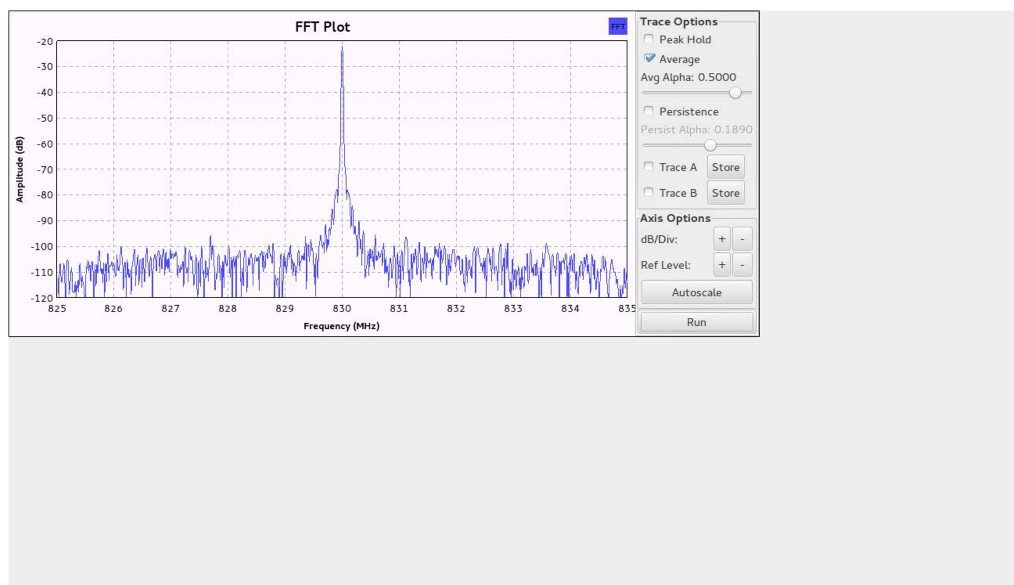

Figure 4.33: Circular transmission, Radio 3, receive frequency FFT plot 830 MHz

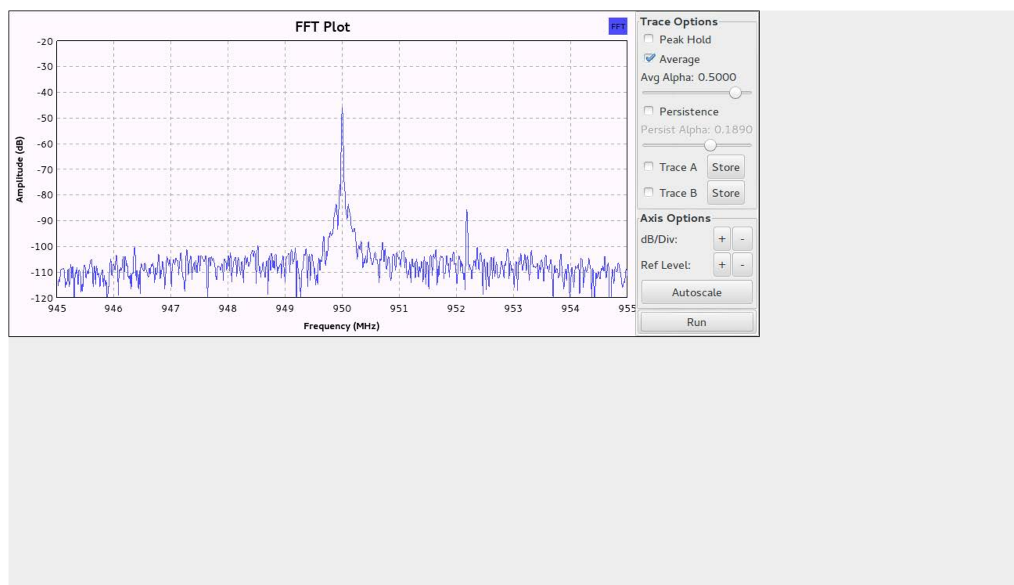

Figure 4.34: Circular transmission, Radio 2, receive frequency FFT plot 950 MHz

Chapter 5

Spectrum Sensing

5.1 Sensing with No Primary User Activities

In this part, we are going to use programs provided with the GNU Radio package such as uhd_fft.grc, and usrp_spectrum_sense.py to observe spectrum activity.

Redesign of a configuration file in GRC to output amplitude and frequency (uhd_fft.grc). The block is shown below with the GRC graph shown in Figure 5.1. On the FFT Plot generated by the GRC graph shown in Figure 5.2 we could see the average noise amplitude was around -120dB and close to 900MHz it was little bit over -100dB.

We also used usrp_spectrum_sense.py script to validate our design. Under the directory /usr/local/share/gnuradio/examples/uhd we ran the script ./usrp_spectrum_sense.py 895M 905M The output showed and confirmed that our noise_floor_db average was -120dB as shown in Figure 5.3

Setting: One radio with two antennas as shown in Figure 4.20

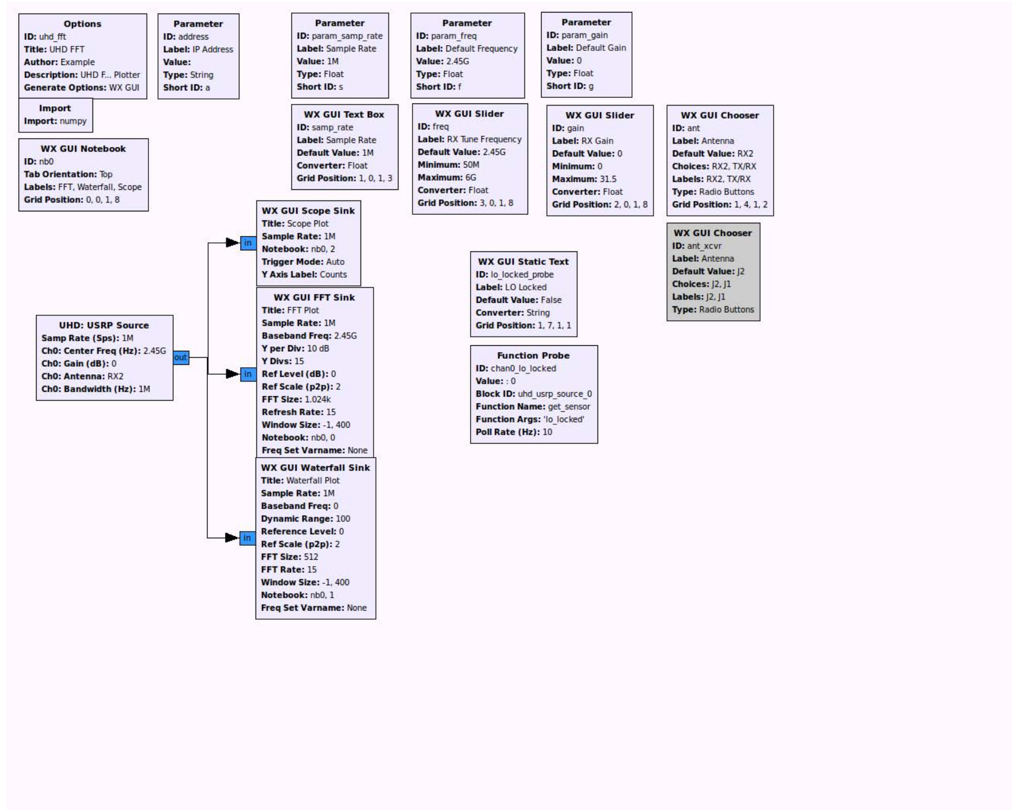

Figure 5.1: GRC configuration with built-in sensing blocks

5.2 Sensing with Primary User Activities

In this part, we included another radio to transmit at different frequencies to observe changes in noise floor power received.

Work continued on the frequency sensing milestone but this time we used a transmission block design to send a signal at 900MHz. Figures 5.2 and 5.3 is when no signal was sent to the uhd_fft design on the radio 1. Figures 5.4 and 5.5 are when the transmission signal is run on the second radio at 900M. The amplitude when no signal is available was at -93.75 dBm, and it was -67 dBm when a signal was detected with -34 dBm as peak amplitude.

usrp_spectrum_sense.py script method

We also used usrp_spectrum_sense.py script when the transmission signal was available. We set it to sense between 895MHz to 905MHz. Under the directory

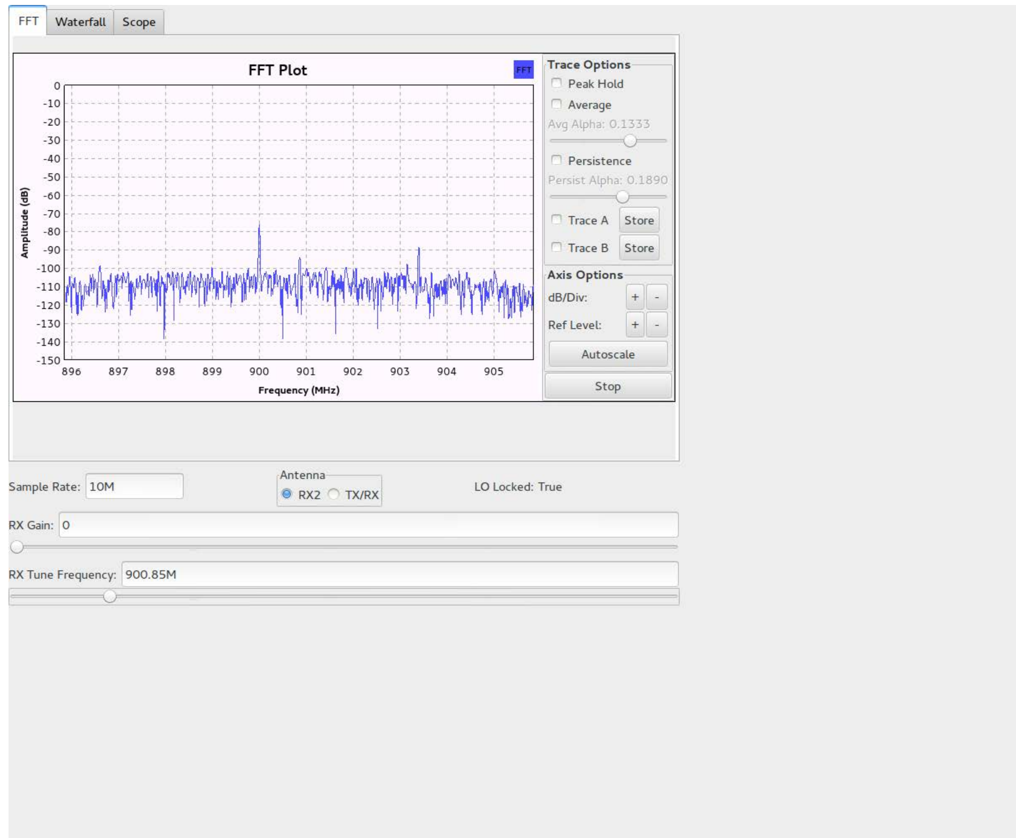

Figure 5.2: Sensing noise floor around -120 dB with FFT plot

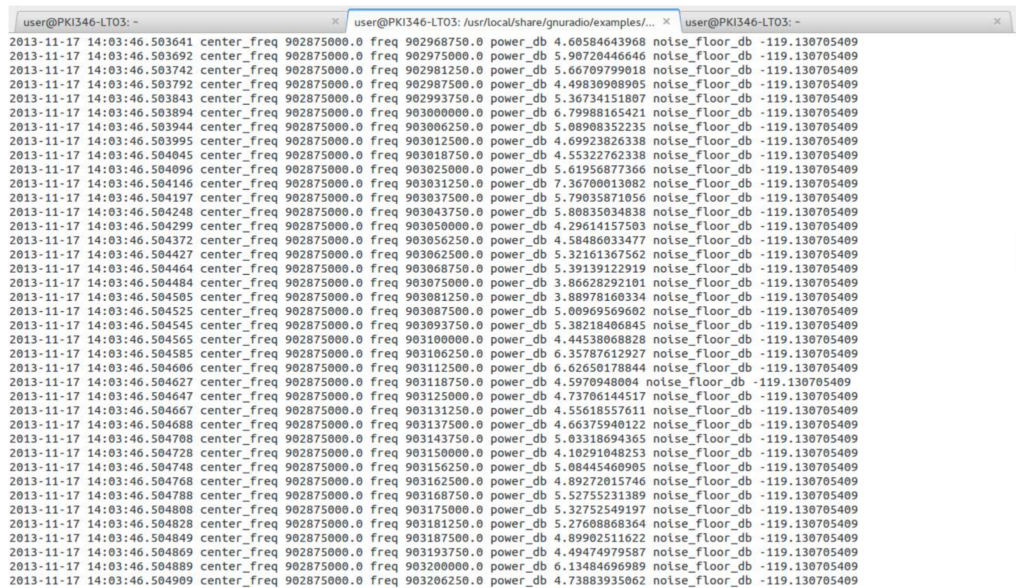

Figure 5.3: Output of usrp_spectrum_sense.py confirming noise floor

/usr/local/share/gnuradio/examples/uhd we ran the script with the following command: ./ursp_spectrum_sense.py 895MHz 905MHz
with the output shown in Figure 5.5.

The output shown in Figure 5.5 shows that our noise_floor_db average was -102 dBm around 900MHz the center frequency at which the transmission radio is sending signal. At 899MHz the average noise_floor was -112 dBm and at 901MHz the noise_floor was at -114 dBm. Settings and results are shown below:

Conclusion: Our radio senses at -102 dBm Setting: Set two radios with GRC companion, one to transmit at 900M the other to run the uhd_fft.grc to sense as shown in Figure 5.4.

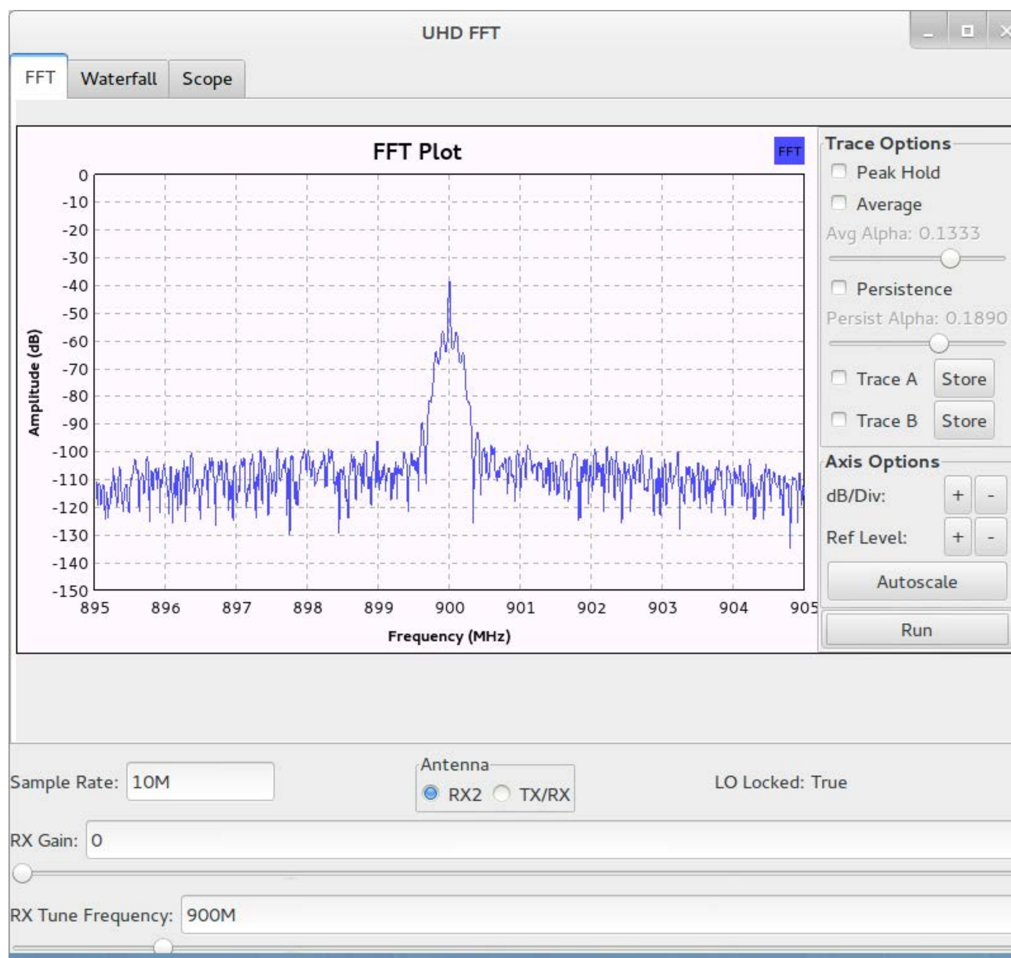

Figure 5.4: FFT plot sensing primary at 900 MHz

```
user@PKI346-LT03: ~    user@PKI346-LT03: /usr/local/share/gnuradio/exam...    user@PKI346-LT03: ~
2013-11-07 10:48:09.442385 center_freq 900125000.0 freq 899906250.0 power_db 55.4475254288 noise_floor_db -102.905683348
2013-11-07 10:48:09.442486 center_freq 900125000.0 freq 899912500.0 power_db 51.6952971936 noise_floor_db -102.905683348
2013-11-07 10:48:09.442602 center_freq 900125000.0 freq 899918750.0 power_db 49.1569733988 noise_floor_db -102.905683348
2013-11-07 10:48:09.442707 center_freq 900125000.0 freq 899925000.0 power_db 47.9869748658 noise_floor_db -102.905683348
2013-11-07 10:48:09.442809 center_freq 900125000.0 freq 899931250.0 power_db 47.8567526805 noise_floor_db -102.905683348
2013-11-07 10:48:09.442928 center_freq 900125000.0 freq 899937500.0 power_db 48.7839485543 noise_floor_db -102.905683348
2013-11-07 10:48:09.443028 center_freq 900125000.0 freq 899943750.0 power_db 50.4569752889 noise_floor_db -102.905683348
2013-11-07 10:48:09.443130 center_freq 900125000.0 freq 899950000.0 power_db 51.5205970115 noise_floor_db -102.905683348
2013-11-07 10:48:09.443232 center_freq 900125000.0 freq 899956250.0 power_db 52.2362070988 noise_floor_db -102.905683348
2013-11-07 10:48:09.443347 center_freq 900125000.0 freq 899962500.0 power_db 53.4690297913 noise_floor_db -102.905683348
2013-11-07 10:48:09.443432 center_freq 900125000.0 freq 899968750.0 power_db 55.2873822282 noise_floor_db -102.905683348
2013-11-07 10:48:09.443491 center_freq 900125000.0 freq 899975000.0 power_db 57.6247547733 noise_floor_db -102.905683348
2013-11-07 10:48:09.443547 center_freq 900125000.0 freq 899981250.0 power_db 58.2075603027 noise_floor_db -102.905683348
2013-11-07 10:48:09.443607 center_freq 900125000.0 freq 899987500.0 power_db 62.0832216242 noise_floor_db -102.905683348
2013-11-07 10:48:09.443676 center_freq 900125000.0 freq 899993750.0 power_db 67.3961647284 noise_floor_db -102.905683348
2013-11-07 10:48:09.443740 center_freq 900125000.0 freq 900000000.0 power_db 67.3153057211 noise_floor_db -102.905683348
2013-11-07 10:48:09.443795 center_freq 900125000.0 freq 900006250.0 power_db 61.8400957284 noise_floor_db -102.905683348
2013-11-07 10:48:09.443855 center_freq 900125000.0 freq 900012500.0 power_db 58.5621927642 noise_floor_db -102.905683348
2013-11-07 10:48:09.443912 center_freq 900125000.0 freq 900018750.0 power_db 57.3521156233 noise_floor_db -102.905683348
2013-11-07 10:48:09.443965 center_freq 900125000.0 freq 900025000.0 power_db 55.2847640216 noise_floor_db -102.905683348
2013-11-07 10:48:09.444053 center_freq 900125000.0 freq 900031250.0 power_db 53.2494721701 noise_floor_db -102.905683348
2013-11-07 10:48:09.444109 center_freq 900125000.0 freq 900037500.0 power_db 52.545366421 noise_floor_db -102.905683348
2013-11-07 10:48:09.444163 center_freq 900125000.0 freq 900043750.0 power_db 51.9001049266 noise_floor_db -102.905683348
2013-11-07 10:48:09.444217 center_freq 900125000.0 freq 900050000.0 power_db 50.4026827065 noise_floor_db -102.905683348
2013-11-07 10:48:09.444270 center_freq 900125000.0 freq 900056250.0 power_db 48.7239946747 noise_floor_db -102.905683348
2013-11-07 10:48:09.444324 center_freq 900125000.0 freq 900062500.0 power_db 47.9891807807 noise_floor_db -102.905683348
2013-11-07 10:48:09.444390 center_freq 900125000.0 freq 900068750.0 power_db 46.8651378129 noise_floor_db -102.905683348
2013-11-07 10:48:09.444446 center_freq 900125000.0 freq 900075000.0 power_db 47.5185196466 noise_floor_db -102.905683348
2013-11-07 10:48:09.444500 center_freq 900125000.0 freq 900081250.0 power_db 51.0510775501 noise_floor_db -102.905683348
2013-11-07 10:48:09.444553 center_freq 900125000.0 freq 900087500.0 power_db 55.6713096692 noise_floor_db -102.905683348
2013-11-07 10:48:09.444606 center_freq 900125000.0 freq 900093750.0 power_db 58.7223890776 noise_floor_db -102.905683348
2013-11-07 10:48:09.444659 center_freq 900125000.0 freq 900100000.0 power_db 56.1745733024 noise_floor_db -102.905683348
2013-11-07 10:48:09.444724 center_freq 900125000.0 freq 900106250.0 power_db 52.8777144993 noise_floor_db -102.905683348
2013-11-07 10:48:09.444781 center_freq 900125000.0 freq 900112500.0 power_db 49.9958817175 noise_floor_db -102.905683348
2013-11-07 10:48:09.444835 center_freq 900125000.0 freq 900118750.0 power_db 48.1252843085 noise_floor_db -102.905683348
2013-11-07 10:48:09.444889 center_freq 900125000.0 freq 900125000.0 power_db 46.350717357 noise_floor_db -102.905683348
2013-11-07 10:48:09.444943 center_freq 900125000.0 freq 900131250.0 power_db 45.8224894236 noise_floor_db -102.905683348
2013-11-07 10:48:09.444997 center_freq 900125000.0 freq 900137500.0 power_db 45.0045728974 noise_floor_db -102.905683348
2013-11-07 10:48:09.445050 center_freq 900125000.0 freq 900143750.0 power_db 44.2982721442 noise_floor_db -102.905683348
2013-11-07 10:48:09.445117 center_freq 900125000.0 freq 900150000.0 power_db 44.4029204878 noise_floor_db -102.905683348
2013-11-07 10:48:09.445173 center_freq 900125000.0 freq 900156250.0 power_db 44.0187959081 noise_floor_db -102.905683348
2013-11-07 10:48:09.445227 center_freq 900125000.0 freq 900162500.0 power_db 43.1198574517 noise_floor_db -102.905683348
2013-11-07 10:48:09.445281 center_freq 900125000.0 freq 900168750.0 power_db 43.9842003505 noise_floor_db -102.905683348
2013-11-07 10:48:09.445334 center_freq 900125000.0 freq 900175000.0 power_db 44.7347575272 noise_floor_db -102.905683348
2013-11-07 10:48:09.445386 center_freq 900125000.0 freq 900181250.0 power_db 46.9525591743 noise_floor_db -102.905683348
2013-11-07 10:48:09.445450 center_freq 900125000.0 freq 900187500.0 power_db 49.2262961053 noise_floor_db -102.905683348
2013-11-07 10:48:09.445505 center_freq 900125000.0 freq 900193750.0 power_db 50.3158782387 noise_floor_db -102.905683348
2013-11-07 10:48:09.445558 center_freq 900125000.0 freq 900200000.0 power_db 46.4834837789 noise_floor_db -102.905683348
2013-11-07 10:48:09.445611 center_freq 900125000.0 freq 900206250.0 power_db 42.2551472051 noise_floor_db -102.905683348
2013-11-07 10:48:09.445664 center_freq 900125000.0 freq 900212500.0 power_db 38.5584752188 noise_floor_db -102.905683348
2013-11-07 10:48:09.445717 center_freq 900125000.0 freq 900218750.0 power_db 35.0354506812 noise_floor_db -102.905683348
2013-11-07 10:48:09.445770 center_freq 900125000.0 freq 900225000.0 power_db 33.1813308817 noise_floor_db -102.905683348
2013-11-07 10:48:09.445834 center_freq 900125000.0 freq 900231250.0 power_db 32.4359117966 noise_floor_db -102.905683348
```

Figure 5.5: Output of usrp_spectrum_sense.py showing sensing power and noise floor with primary on

2013-11-07 10:48:08.946518 center_freq 899375000.0 freq 899718750.0 power_db 36.3289261445 noise_floor_db -112.140832222

2013-11-07 10:48:08.946571 center_freq 899375000.0 freq 899725000.0 power_db 33.9021367626 noise_floor_db -112.140832222

2013-11-07 10:48:08.946624 center_freq 899375000.0 freq 899731250.0 power_db 32.7616186608 noise_floor_db -112.140832222

2013-11-07 10:48:08.946677 center_freq 899375000.0 freq 899737500.0 power_db 34.906888034 noise_floor_db -112.140832222

2013-11-07 10:48:08.946745 center_freq 899375000.0 freq 899743750.0 power_db 36.5907079158 noise_floor_db -112.140832222

2013-11-07 10:48:09.439582 center_freq 900125000.0 freq 899750000.0 power_db

27.3321624648 noise_floor_db -102.905683348

2013-11-07 10:48:09.439726 center_freq 900125000.0 freq 899756250.0 power_db 30.0780642102 noise_floor_db -102.905683348

2013-11-07 10:48:09.439858 center_freq 900125000.0 freq 899762500.0 power_db 32.2757709265 noise_floor_db -102.905683348

2013-11-07 10:48:09.439962 center_freq 900125000.0 freq 899768750.0 power_db 33.0531365666 noise_floor_db -102.905683348

2013-11-07 10:48:09.440124 center_freq 900125000.0 freq 899775000.0 power_db 35.5477540436 noise_floor_db -102.905683348

2013-11-07 10:48:09.940027 center_freq 900875000.0 freq 900500000.0 power_db 16.1476730148 noise_floor_db -114.387927013

2013-11-07 10:48:09.940143 center_freq 900875000.0 freq 900506250.0 power_db 12.7616848263 noise_floor_db -114.387927013

2013-11-07 10:48:09.940205 center_freq 900875000.0 freq 900512500.0 power_db 9.832082081 noise_floor_db -114.387927013

2013-11-07 10:48:09.940282 center_freq 900875000.0 freq 900518750.0 power_db 10.62603825 noise_floor_db -114.387927013

2013-11-07 10:48:09.940372 center_freq 900875000.0 freq 900525000.0 power_db 11.8954172239 noise_floor_db -114.387927013

In this part, we are observing change in received signal power. Sensing with signal power:

Sensing with no signal transmission

Settings and results are shown below: We ran usrp_spectrum_sense.py script when any transmission signal was not available between 895MHz to 905MHz and observe the power of the signal received at the output.

We can observe that the power_db of the signal the receiver is sensing around 4 to 5 dB when no signal is present as shown in Figures 5.6 & 5.7. Figures 5.8 & 5.9

show that power_db received is 35 to 50 dB when PU is transmitting.

```
user@PKI346-LT03: /usr/local/share/gn...  ×  user@PKI346-LT03: /usr/local/share/gn...  ×  user@PKI346-LT03: /usr/local/share/gn...  ×  user@PKI346-LT03: /usr/local/share/gnu...  ×
2013-12-04 17:37:32.624847 center_freq 900625000.0 freq 900281250.0 power_db 4.27438383236 noise_floor_db -118.252610973
2013-12-04 17:37:32.624900 center_freq 900625000.0 freq 900287500.0 power_db 3.13691176279 noise_floor_db -118.252610973
2013-12-04 17:37:32.624957 center_freq 900625000.0 freq 900293750.0 power_db 4.01045771315 noise_floor_db -118.252610973
2013-12-04 17:37:32.625013 center_freq 900625000.0 freq 900300000.0 power_db 5.10912109752 noise_floor_db -118.252610973
2013-12-04 17:37:32.625090 center_freq 900625000.0 freq 900306250.0 power_db 4.7258529307 noise_floor_db -118.252610973
2013-12-04 17:37:32.625172 center_freq 900625000.0 freq 900312500.0 power_db 2.36676371559 noise_floor_db -118.252610973
2013-12-04 17:37:32.625231 center_freq 900625000.0 freq 900318750.0 power_db 3.00449012256 noise_floor_db -118.252610973
2013-12-04 17:37:32.625287 center_freq 900625000.0 freq 900325000.0 power_db 4.13499057243 noise_floor_db -118.252610973
2013-12-04 17:37:32.625342 center_freq 900625000.0 freq 900331250.0 power_db 4.07166237601 noise_floor_db -118.252610973
2013-12-04 17:37:32.625394 center_freq 900625000.0 freq 900337500.0 power_db 3.67965398525 noise_floor_db -118.252610973
2013-12-04 17:37:32.625482 center_freq 900625000.0 freq 900343750.0 power_db 4.09046745688 noise_floor_db -118.252610973
2013-12-04 17:37:32.625543 center_freq 900625000.0 freq 900350000.0 power_db 6.15964047541 noise_floor_db -118.252610973
2013-12-04 17:37:32.625602 center_freq 900625000.0 freq 900356250.0 power_db 6.13899925656 noise_floor_db -118.252610973
2013-12-04 17:37:32.625655 center_freq 900625000.0 freq 900362500.0 power_db 3.36073109052 noise_floor_db -118.252610973
2013-12-04 17:37:32.625713 center_freq 900625000.0 freq 900368750.0 power_db 3.44938141238 noise_floor_db -118.252610973
2013-12-04 17:37:32.625773 center_freq 900625000.0 freq 900375000.0 power_db 3.68114525512 noise_floor_db -118.252610973
2013-12-04 17:37:32.625864 center_freq 900625000.0 freq 900381250.0 power_db 3.84520110398 noise_floor_db -118.252610973
2013-12-04 17:37:32.625927 center_freq 900625000.0 freq 900387500.0 power_db 3.48332118781 noise_floor_db -118.252610973
2013-12-04 17:37:32.625982 center_freq 900625000.0 freq 900393750.0 power_db 4.38438446862 noise_floor_db -118.252610973
2013-12-04 17:37:32.626038 center_freq 900625000.0 freq 900400000.0 power_db 4.17331319478 noise_floor_db -118.252610973
2013-12-04 17:37:32.626095 center_freq 900625000.0 freq 900406250.0 power_db 3.56907615728 noise_floor_db -118.252610973
2013-12-04 17:37:32.626167 center_freq 900625000.0 freq 900412500.0 power_db 4.54674170169 noise_floor_db -118.252610973
2013-12-04 17:37:32.626247 center_freq 900625000.0 freq 900418750.0 power_db 5.05339033354 noise_floor_db -118.252610973
2013-12-04 17:37:32.626306 center_freq 900625000.0 freq 900425000.0 power_db 4.67884254606 noise_floor_db -118.252610973
2013-12-04 17:37:32.626361 center_freq 900625000.0 freq 900431250.0 power_db 5.68337447612 noise_floor_db -118.252610973
2013-12-04 17:37:32.626416 center_freq 900625000.0 freq 900437500.0 power_db 4.76325333485 noise_floor_db -118.252610973
2013-12-04 17:37:32.626473 center_freq 900625000.0 freq 900443750.0 power_db 4.34321621247 noise_floor_db -118.252610973
2013-12-04 17:37:32.626563 center_freq 900625000.0 freq 900450000.0 power_db 6.12130924818 noise_floor_db -118.252610973
2013-12-04 17:37:32.626627 center_freq 900625000.0 freq 900456250.0 power_db 6.23850754527 noise_floor_db -118.252610973
2013-12-04 17:37:32.626680 center_freq 900625000.0 freq 900462500.0 power_db 5.30309686864 noise_floor_db -118.252610973
2013-12-04 17:37:32.626732 center_freq 900625000.0 freq 900468750.0 power_db 4.19646240558 noise_floor_db -118.252610973
2013-12-04 17:37:32.626783 center_freq 900625000.0 freq 900475000.0 power_db 4.52078952762 noise_floor_db -118.252610973
2013-12-04 17:37:32.626834 center_freq 900625000.0 freq 900481250.0 power_db 4.20222822106 noise_floor_db -118.252610973
2013-12-04 17:37:32.626916 center_freq 900625000.0 freq 900487500.0 power_db 4.90476562711 noise_floor_db -118.252610973
2013-12-04 17:37:32.626977 center_freq 900625000.0 freq 900493750.0 power_db 5.34156349197 noise_floor_db -118.252610973
2013-12-04 17:37:32.627029 center_freq 900625000.0 freq 900500000.0 power_db 5.05281808905 noise_floor_db -118.252610973
2013-12-04 17:37:32.627081 center_freq 900625000.0 freq 900506250.0 power_db 3.86770589482 noise_floor_db -118.252610973
2013-12-04 17:37:32.627132 center_freq 900625000.0 freq 900512500.0 power_db 4.57265999464 noise_floor_db -118.252610973
2013-12-04 17:37:32.627183 center_freq 900625000.0 freq 900518750.0 power_db 4.20552276302 noise_floor_db -118.252610973
```

Figure 5.6: Output of usrp_spectrum_sense.py showing sensing power and noise floor with primary off

5.3 Audio/Video Application

Audio/Video (AV) applications are great for demos, and GStreamer is a well-documented application programming interface (API) that was used to capture, encode and pipe AV to GNU Radio.

5.3.1 Webcam

The first GStreamer application written was to capture live video from the built in webcam on the laptop and display it on the screen. Figure 5.10 shows the successful script and results.

```
2013-12-04 17:37:32.624847 center_freq 900625000.0 freq 900281250.0 power_db 4.27438383236
2013-12-04 17:37:32.624900 center_freq 900625000.0 freq 900287500.0 power_db 3.13691176279
2013-12-04 17:37:32.624957 center_freq 900625000.0 freq 900293750.0 power_db 4.01045771315
2013-12-04 17:37:32.625013 center_freq 900625000.0 freq 900300000.0 power_db 5.10912109752
2013-12-04 17:37:32.625090 center_freq 900625000.0 freq 900306250.0 power_db 4.7258529307 n
2013-12-04 17:37:32.625172 center_freq 900625000.0 freq 900312500.0 power_db 2.36676371559
2013-12-04 17:37:32.625231 center_freq 900625000.0 freq 900318750.0 power_db 3.00449012256
2013-12-04 17:37:32.625287 center_freq 900625000.0 freq 900325000.0 power_db 4.13499057243
2013-12-04 17:37:32.625342 center_freq 900625000.0 freq 900331250.0 power_db 4.07166237601
2013-12-04 17:37:32.625394 center_freq 900625000.0 freq 900337500.0 power_db 3.67965398525
2013-12-04 17:37:32.625482 center_freq 900625000.0 freq 900343750.0 power_db 4.09046745688
2013-12-04 17:37:32.625543 center_freq 900625000.0 freq 900350000.0 power_db 6.15964047541
2013-12-04 17:37:32.625602 center_freq 900625000.0 freq 900356250.0 power_db 6.13899925656
2013-12-04 17:37:32.625655 center_freq 900625000.0 freq 900362500.0 power_db 3.36073109052
2013-12-04 17:37:32.625713 center_freq 900625000.0 freq 900368750.0 power_db 3.44938141238
2013-12-04 17:37:32.625773 center_freq 900625000.0 freq 900375000.0 power_db 3.68114525512
2013-12-04 17:37:32.625864 center_freq 900625000.0 freq 900381250.0 power_db 3.84520110398
2013-12-04 17:37:32.625927 center_freq 900625000.0 freq 900387500.0 power_db 3.48332118781
2013-12-04 17:37:32.625982 center_freq 900625000.0 freq 900393750.0 power_db 4.38438446862
2013-12-04 17:37:32.626038 center_freq 900625000.0 freq 900400000.0 power_db 4.17331319478
2013-12-04 17:37:32.626095 center_freq 900625000.0 freq 900406250.0 power_db 3.56907615728
2013-12-04 17:37:32.626167 center_freq 900625000.0 freq 900412500.0 power_db 4.54674170169
2013-12-04 17:37:32.626247 center_freq 900625000.0 freq 900418750.0 power_db 5.05339033354
2013-12-04 17:37:32.626306 center_freq 900625000.0 freq 900425000.0 power_db 4.67884254606
2013-12-04 17:37:32.626361 center_freq 900625000.0 freq 900431250.0 power_db 5.68337447612
2013-12-04 17:37:32.626416 center_freq 900625000.0 freq 900437500.0 power_db 4.76325333485
2013-12-04 17:37:32.626473 center_freq 900625000.0 freq 900443750.0 power_db 4.34321621247
2013-12-04 17:37:32.626563 center_freq 900625000.0 freq 900450000.0 power_db 6.12130924818
2013-12-04 17:37:32.626627 center_freq 900625000.0 freq 900456250.0 power_db 6.23850754527
2013-12-04 17:37:32.626680 center_freq 900625000.0 freq 900462500.0 power_db 5.30309686864
2013-12-04 17:37:32.626732 center_freq 900625000.0 freq 900468750.0 power_db 4.19646240558
2013-12-04 17:37:32.626783 center_freq 900625000.0 freq 900475000.0 power_db 4.52078952762
2013-12-04 17:37:32.626834 center_freq 900625000.0 freq 900481250.0 power_db 4.20222822106
2013-12-04 17:37:32.626916 center_freq 900625000.0 freq 900487500.0 power_db 4.90476562711
2013-12-04 17:37:32.626977 center_freq 900625000.0 freq 900493750.0 power_db 5.34156349197
2013-12-04 17:37:32.627029 center_freq 900625000.0 freq 900500000.0 power_db 5.05281808905
2013-12-04 17:37:32.627081 center_freq 900625000.0 freq 900506250.0 power_db 3.86770589482
2013-12-04 17:37:32.627132 center_freq 900625000.0 freq 900512500.0 power_db 4.57265999464
2013-12-04 17:37:32.627183 center_freq 900625000.0 freq 900518750.0 power_db 4.20552276302
```

Figure 5.7: Output of usrp_spectrum_sense.py showing sensing power with primary off

5.3.2 User Datagram Protocol (UDP)

Initially, research showed other people using UDP to send data from GStreamer to GNU Radio, however, those people were using wired media between SDRs. Figures 5.11 and 5.12 show the script and GRC graph created to test communications using this protocol. Another reason UDP was looked was because we did not know how to create a pipe between GStreamer and GNU Radio. Our first idea was to use sockets or write to a port with one application while reading from the same port with another. Please notice that the below test was done using a channel model (virtual channel).

A Wireshark filter to sniff UDP packets at port 1234 and the USRP Ethernet

```
user@PKI346-LT03: /usr/local/share/gn...   user@PKI346-LT03: /usr/local/share/gn...   user@PKI346-LT03: /usr/local/share/gn...   user@PKI346-LT03: /usr/local/share/gnu...
2013-12-04 17:39:52.980039 center_freq 899875000.0 freq 900056250.0 power_db 48.6323563254 noise_floor_db -108.607500438
2013-12-04 17:39:52.980107 center_freq 899875000.0 freq 900062500.0 power_db 48.1930762365 noise_floor_db -108.607500438
2013-12-04 17:39:52.980184 center_freq 899875000.0 freq 900068750.0 power_db 47.8402007092 noise_floor_db -108.607500438
2013-12-04 17:39:52.980241 center_freq 899875000.0 freq 900075000.0 power_db 49.3893048187 noise_floor_db -108.607500438
2013-12-04 17:39:52.980305 center_freq 899875000.0 freq 900081250.0 power_db 52.9724292254 noise_floor_db -108.607500438
2013-12-04 17:39:52.980362 center_freq 899875000.0 freq 900087500.0 power_db 58.0426770761 noise_floor_db -108.607500438
2013-12-04 17:39:52.980416 center_freq 899875000.0 freq 900093750.0 power_db 60.6882986731 noise_floor_db -108.607500438
2013-12-04 17:39:52.980490 center_freq 899875000.0 freq 900100000.0 power_db 58.1125881794 noise_floor_db -108.607500438
2013-12-04 17:39:52.980561 center_freq 899875000.0 freq 900106250.0 power_db 54.6785110443 noise_floor_db -108.607500438
2013-12-04 17:39:52.980626 center_freq 899875000.0 freq 900112500.0 power_db 51.9217820534 noise_floor_db -108.607500438
2013-12-04 17:39:52.980685 center_freq 899875000.0 freq 900118750.0 power_db 50.4805776433 noise_floor_db -108.607500438
2013-12-04 17:39:52.980739 center_freq 899875000.0 freq 900125000.0 power_db 48.5043834428 noise_floor_db -108.607500438
2013-12-04 17:39:52.980792 center_freq 899875000.0 freq 900131250.0 power_db 46.9465235772 noise_floor_db -108.607500438
2013-12-04 17:39:52.980878 center_freq 899875000.0 freq 900137500.0 power_db 45.7239226291 noise_floor_db -108.607500438
2013-12-04 17:39:52.980941 center_freq 899875000.0 freq 900143750.0 power_db 43.1259312948 noise_floor_db -108.607500438
2013-12-04 17:39:52.980995 center_freq 899875000.0 freq 900150000.0 power_db 44.8320617799 noise_floor_db -108.607500438
2013-12-04 17:39:52.981048 center_freq 899875000.0 freq 900156250.0 power_db 44.710410082 noise_floor_db -108.607500438
2013-12-04 17:39:52.981102 center_freq 899875000.0 freq 900162500.0 power_db 45.738357524 noise_floor_db -108.607500438
2013-12-04 17:39:52.981155 center_freq 899875000.0 freq 900168750.0 power_db 46.5221750159 noise_floor_db -108.607500438
2013-12-04 17:39:52.981242 center_freq 899875000.0 freq 900175000.0 power_db 46.2229440141 noise_floor_db -108.607500438
2013-12-04 17:39:52.981305 center_freq 899875000.0 freq 900181250.0 power_db 48.8144222536 noise_floor_db -108.607500438
2013-12-04 17:39:52.981359 center_freq 899875000.0 freq 900187500.0 power_db 51.183284819 noise_floor_db -108.607500438
2013-12-04 17:39:52.981414 center_freq 899875000.0 freq 900193750.0 power_db 51.9649365261 noise_floor_db -108.607500438
2013-12-04 17:39:52.981468 center_freq 899875000.0 freq 900200000.0 power_db 47.9796661379 noise_floor_db -108.607500438
2013-12-04 17:39:52.981521 center_freq 899875000.0 freq 900206250.0 power_db 43.8347139975 noise_floor_db -108.607500438
2013-12-04 17:39:52.981606 center_freq 899875000.0 freq 900212500.0 power_db 40.36161627 noise_floor_db -108.607500438
2013-12-04 17:39:52.981669 center_freq 899875000.0 freq 900218750.0 power_db 37.2069131044 noise_floor_db -108.607500438
2013-12-04 17:39:52.981723 center_freq 899875000.0 freq 900225000.0 power_db 34.565962271 noise_floor_db -108.607500438
2013-12-04 17:39:52.981776 center_freq 899875000.0 freq 900231250.0 power_db 33.434033033 noise_floor_db -108.607500438
2013-12-04 17:39:52.981830 center_freq 899875000.0 freq 900237500.0 power_db 29.9127903296 noise_floor_db -108.607500438
2013-12-04 17:39:52.981883 center_freq 899875000.0 freq 900243750.0 power_db 27.0177478054 noise_floor_db -108.607500438
2013-12-04 17:39:53.474705 center_freq 900625000.0 freq 900250000.0 power_db 35.0656158241 noise_floor_db -114.069464118
2013-12-04 17:39:53.474837 center_freq 900625000.0 freq 900256250.0 power_db 32.905772356 noise_floor_db -114.069464118
2013-12-04 17:39:53.474947 center_freq 900625000.0 freq 900262500.0 power_db 30.2099825081 noise_floor_db -114.069464118
2013-12-04 17:39:53.475048 center_freq 900625000.0 freq 900268750.0 power_db 31.9707673886 noise_floor_db -114.069464118
2013-12-04 17:39:53.475157 center_freq 900625000.0 freq 900275000.0 power_db 34.5851684468 noise_floor_db -114.069464118
2013-12-04 17:39:53.475266 center_freq 900625000.0 freq 900281250.0 power_db 35.9007956834 noise_floor_db -114.069464118
2013-12-04 17:39:53.475372 center_freq 900625000.0 freq 900287500.0 power_db 36.3975398507 noise_floor_db -114.069464118
2013-12-04 17:39:53.475480 center_freq 900625000.0 freq 900293750.0 power_db 34.4317957906 noise_floor_db -114.069464118
```

Figure 5.8: Output of usrp_spectrum_sense.py showing sensing power and noise floor with primary on

port (eth2) was used to verify packets at port. Packets were observed at both IPs and stopped when transmission stopped.

5.3.3 Video File Transfer via Virtual Channel

Upon further research pipelines were created using First In, First Out (FIFO) files which are created with the following command: sudo mkfifo filename. Figure 5.13 shows GRC flow graph that takes a video file and encodes, modulates, transfers through virtual radio channel, demodulates, decodes, and writes to the FIFO file rxvid3.ts. The virtual channel was used to ensure that all steps excluding the radio link worked properly.

Figure 5.14 is the GStreamer script used to read from the FIFO file and save it as a different file. The Linux command line shown in Figure 5.15 shows the shell file running. Finally, Figure 5.16 shows the saved file

```
2013-12-04 17:39:52.980039 center_freq 899875000.0 freq 900056250.0 power_db 48.6323563254
2013-12-04 17:39:52.980107 center_freq 899875000.0 freq 900062500.0 power_db 48.1930762365
2013-12-04 17:39:52.980184 center_freq 899875000.0 freq 900068750.0 power_db 47.8402007092
2013-12-04 17:39:52.980241 center_freq 899875000.0 freq 900075000.0 power_db 49.3893048187
2013-12-04 17:39:52.980305 center_freq 899875000.0 freq 900081250.0 power_db 52.9724292254
2013-12-04 17:39:52.980362 center_freq 899875000.0 freq 900087500.0 power_db 58.0426770761
2013-12-04 17:39:52.980416 center_freq 899875000.0 freq 900093750.0 power_db 60.6882986731
2013-12-04 17:39:52.980490 center_freq 899875000.0 freq 900100000.0 power_db 58.1125881794
2013-12-04 17:39:52.980561 center_freq 899875000.0 freq 900106250.0 power_db 54.6785110443
2013-12-04 17:39:52.980626 center_freq 899875000.0 freq 900112500.0 power_db 51.9217820534
2013-12-04 17:39:52.980685 center_freq 899875000.0 freq 900118750.0 power_db 50.4805776433
2013-12-04 17:39:52.980739 center_freq 899875000.0 freq 900125000.0 power_db 48.5043834428
2013-12-04 17:39:52.980792 center_freq 899875000.0 freq 900131250.0 power_db 46.9465235772
2013-12-04 17:39:52.980878 center_freq 899875000.0 freq 900137500.0 power_db 45.7239226291
2013-12-04 17:39:52.980941 center_freq 899875000.0 freq 900143750.0 power_db 43.1259312948
2013-12-04 17:39:52.980995 center_freq 899875000.0 freq 900150000.0 power_db 44.8320617799
2013-12-04 17:39:52.981048 center_freq 899875000.0 freq 900156250.0 power_db 44.710410082 n
2013-12-04 17:39:52.981102 center_freq 899875000.0 freq 900162500.0 power_db 45.738357524 n
2013-12-04 17:39:52.981155 center_freq 899875000.0 freq 900168750.0 power_db 46.5221750159
2013-12-04 17:39:52.981242 center_freq 899875000.0 freq 900175000.0 power_db 46.2229440141
2013-12-04 17:39:52.981305 center_freq 899875000.0 freq 900181250.0 power_db 48.8144222536
2013-12-04 17:39:52.981359 center_freq 899875000.0 freq 900187500.0 power_db 51.183284819 n
2013-12-04 17:39:52.981414 center_freq 899875000.0 freq 900193750.0 power_db 51.9649365261
2013-12-04 17:39:52.981468 center_freq 899875000.0 freq 900200000.0 power_db 47.9796661379
2013-12-04 17:39:52.981521 center_freq 899875000.0 freq 900206250.0 power_db 43.8347139975
2013-12-04 17:39:52.981606 center_freq 899875000.0 freq 900212500.0 power_db 40.36161627 no
2013-12-04 17:39:52.981669 center_freq 899875000.0 freq 900218750.0 power_db 37.2069131044
2013-12-04 17:39:52.981723 center_freq 899875000.0 freq 900225000.0 power_db 34.565962271 n
2013-12-04 17:39:52.981776 center_freq 899875000.0 freq 900231250.0 power_db 33.434033033 n
2013-12-04 17:39:52.981830 center_freq 899875000.0 freq 900237500.0 power_db 29.9127903296
2013-12-04 17:39:52.981883 center_freq 899875000.0 freq 900243750.0 power_db 27.0177478054
2013-12-04 17:39:53.474705 center_freq 900625000.0 freq 900250000.0 power_db 35.0656158241
2013-12-04 17:39:53.474837 center_freq 900625000.0 freq 900256250.0 power_db 32.905772356 n
2013-12-04 17:39:53.474947 center_freq 900625000.0 freq 900262500.0 power_db 30.2099825081
2013-12-04 17:39:53.475048 center_freq 900625000.0 freq 900268750.0 power_db 31.9707673886
2013-12-04 17:39:53.475157 center_freq 900625000.0 freq 900275000.0 power_db 34.5851684468
2013-12-04 17:39:53.475266 center_freq 900625000.0 freq 900281250.0 power_db 35.9007956834
2013-12-04 17:39:53.475372 center_freq 900625000.0 freq 900287500.0 power_db 36.3975398507
2013-12-04 17:39:53.475480 center_freq 900625000.0 freq 900293750.0 power_db 34.4317957906
```

Figure 5.9: Output of usrp_spectrum_sense.py showing sensing power with primary on

5.3.4 Live Streaming Using Two USRP2 SDRs

The next step was to successfully transmit and receive live video from the webcam on terminal one, and display the video on terminal two. Figure 5.17 shows the GStreamer script for the transmit side while Figure 5.18 shows the receive side. Figure 5.19 shows the GRC flow graph for the transmit side while Figure 5.20 shows the receive side. Figure 5.21 shows a screen shot of the live video from the receive side. Please click on the hyperlinks below to view the setup and successful execution of this test.

Link 1

Link 2

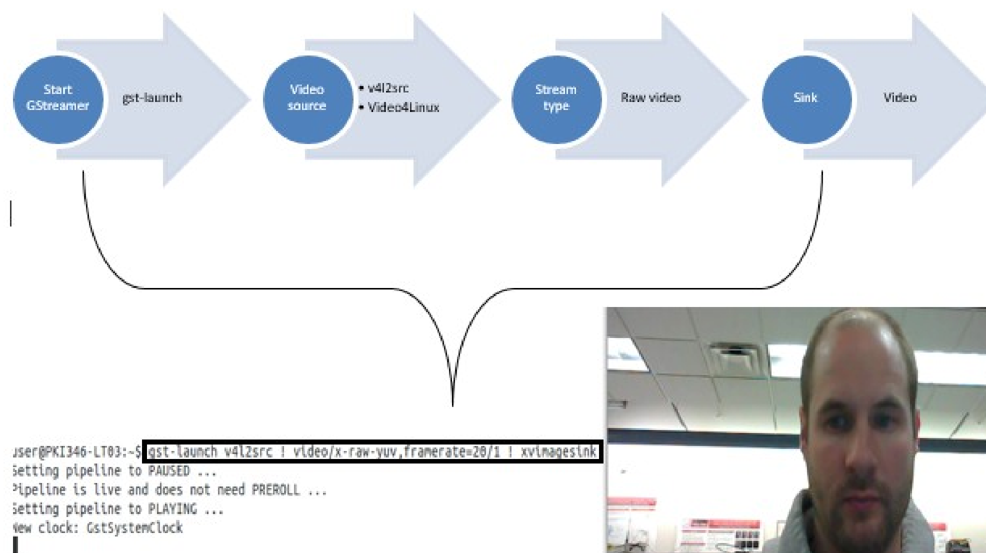

Figure 5.10: GStreamer terminal command for video transmit to GRC via UDP

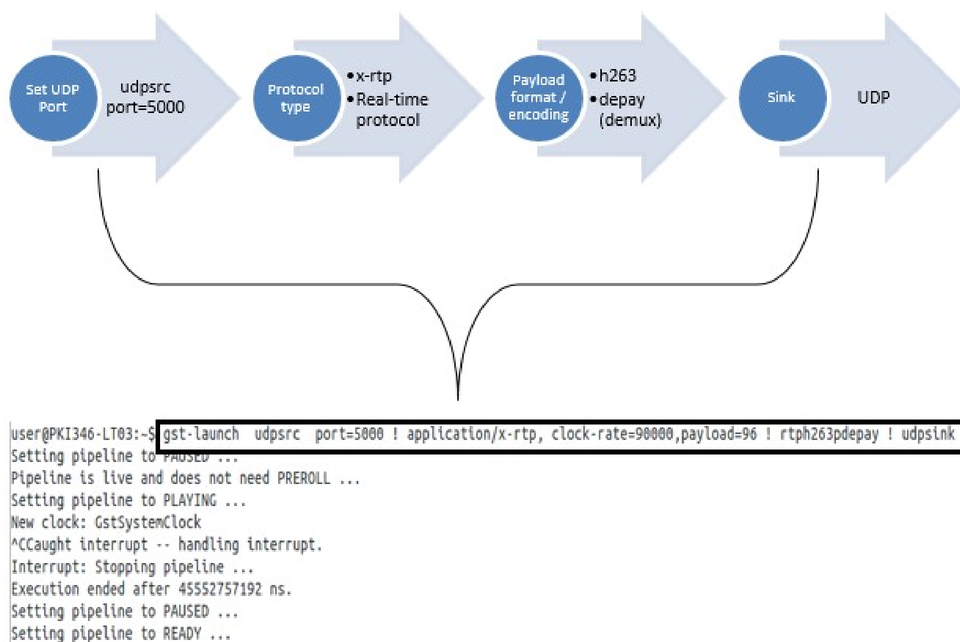

Figure 5.11: GStreamer terminal command for video receive from GRC via UDP

5.3.5 Audio Recording and Streaming with GStreamer

Audio streaming is a great application for demos and a great way to learn GStreamer. Learning the audio libraries in GStreamer was done in multiple steps which include recording live audio from microphone to file, and streaming audio between

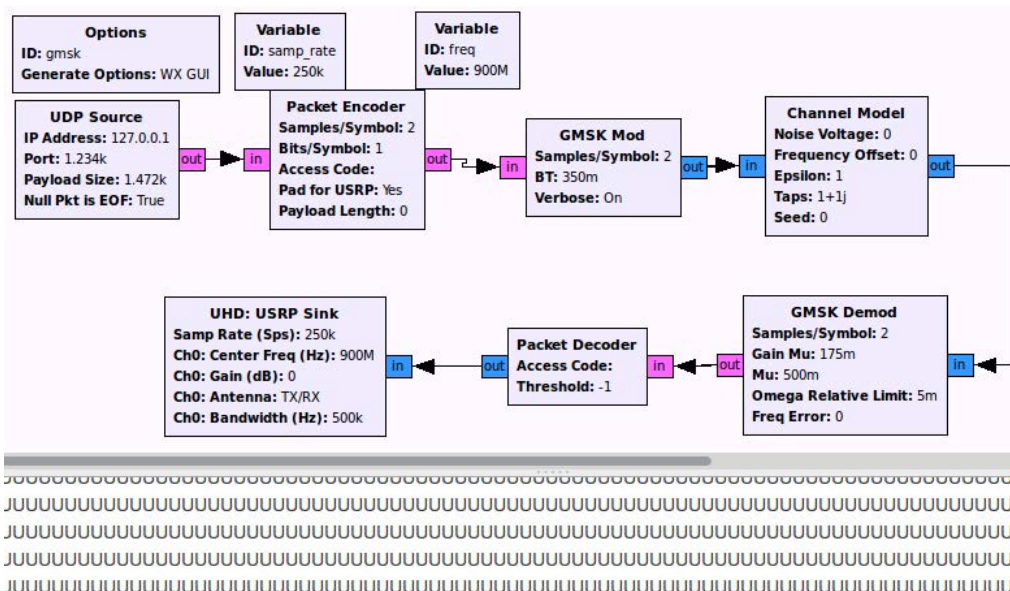

Figure 5.12: GRC flow graph to receive video from GStreamer via UDP and tx/rx using a virtual channel

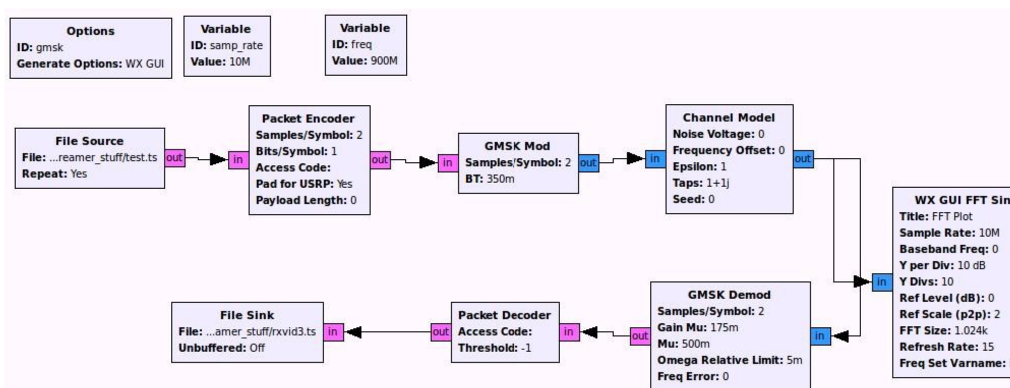

Figure 5.13: GRC flow graph to retrieve video from file, tx/rx through virtual channel and send to GStreamer using Linux FIFO pipe file

```
gst-launch -e -v filesrc location=rxvid3.ts ! queue ! filesink location=rxvidfile.ts
```

Figure 5.14: GStreamer BASH script to receive video file from GRC and save it to another file

two stations.

1. Successfully saved audio to file using GStreamer with the script shown in

```
user@PKI346-LT03:~/GStreamer_stuff$ ./gst_udptx_port.sh
Setting pipeline to PAUSED ...
Pipeline is PREROLLING ...
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstSystemClock
Got EOS from element "pipeline0".
Execution ended after 6676676855 ns.
Setting pipeline to PAUSED ...
Setting pipeline to READY ...
Setting pipeline to NULL ...
Freeing pipeline ...
user@PKI346-LT03:~/GStreamer_stuff$
```

Figure 5.15: GStreamer shell file executed and running

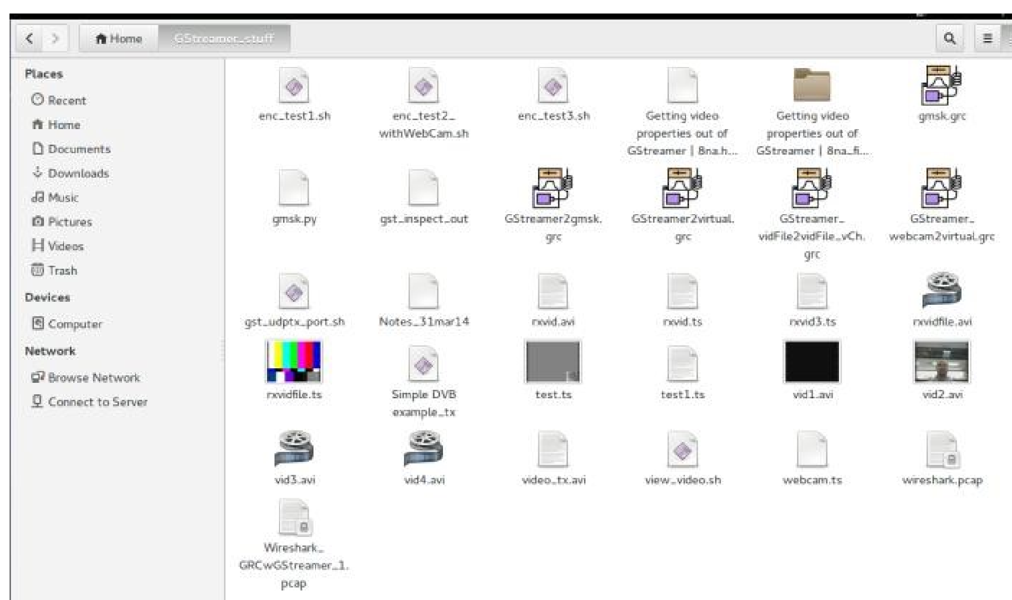

Figure 5.16: Directory with new file name rxvidfile.ts created by GStreamer

```
gst-launch -e -v v4l2src ! tee name="splitter" ! \
queue leaky=1 ! video/x-raw-yuv, framerate=10/1, wideth=320,
height=240 ! ffmpegcolorspace ! xvimagesink sync=false \
splitter. ! queue ! x264enc ! mpegtsmux ! filesink
location=test_live.ts
```

Figure 5.17: GStreamer shell script for live video streaming - transmit side

Figure 5.22

```
gst-launch-1.0 -e -v filesrc location=rxvid.ts ! queue max-size-bytes=512000 ! decodebin ! xvimagesink
```

Figure 5.18: GStreamer shell script for live video streaming - receive side

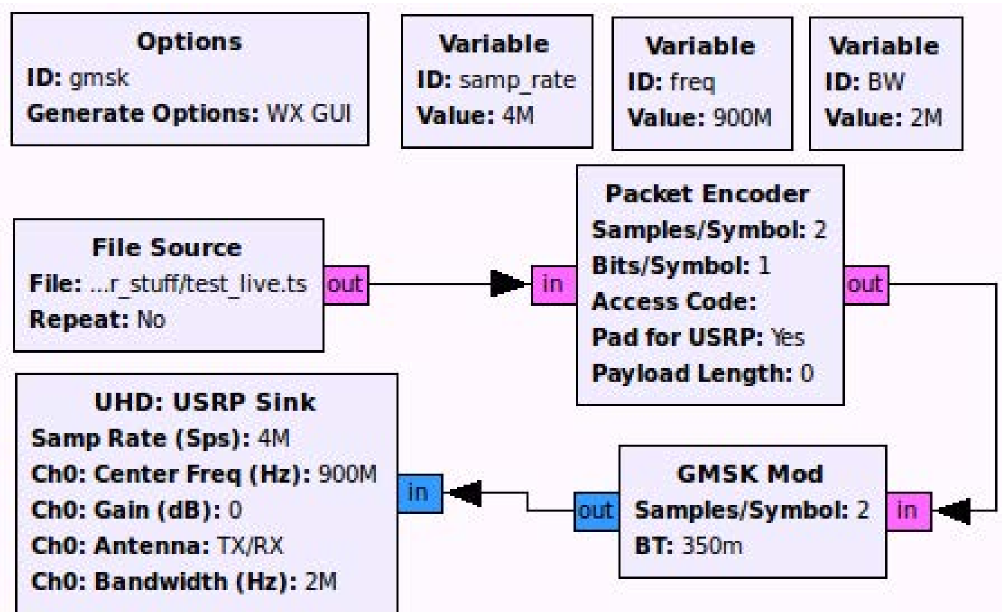

Figure 5.19: GRC flow graph for live video streaming - transmit side

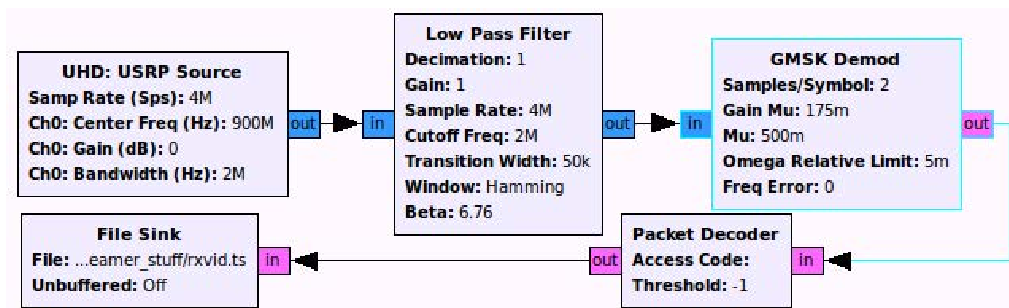

Figure 5.20: GRC flow graph for live video streaming - receive side

2. Successfully saved audio to file using GStreamer piped to GNU Radio

3. Successfully transmitted audio file from node_1 to node_2 with playback ability. Also, no error in playback due to use of throttle block in GRC

4. Successfully recorded live voice to file using built in microphone and GStream-

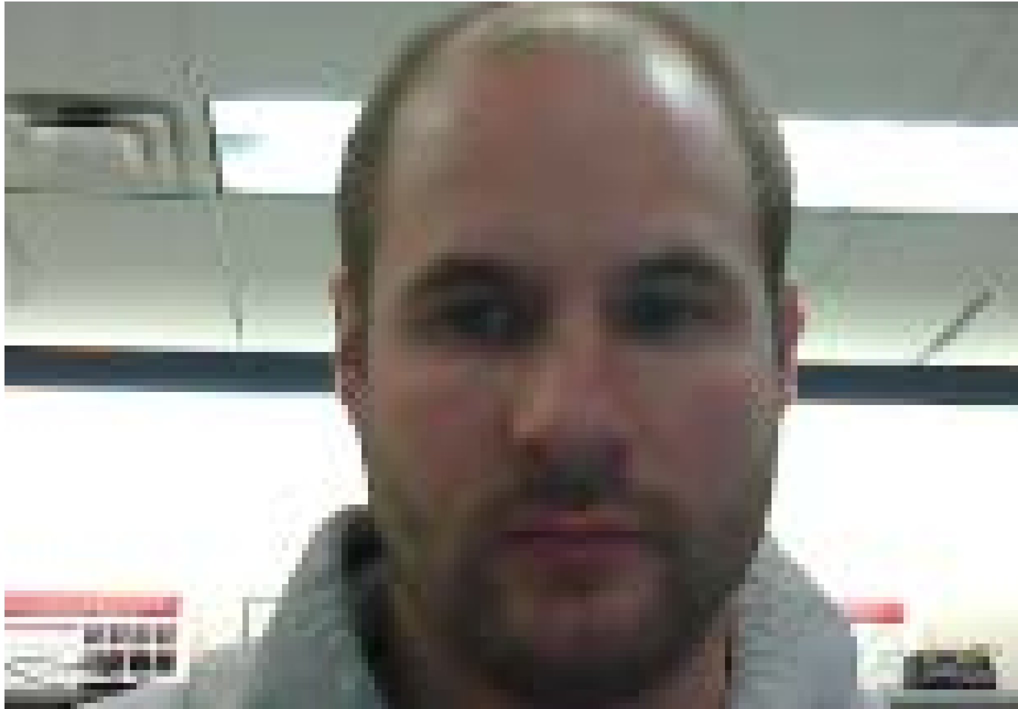

Figure 5.21: Received streamed video produced by GStreamer sink

er with the script shown in Figure 5.23

5. Successfully transmitted live voice stream from node_1 to node_2; however, noise made it difficult to hear voice

6. Cleaned up signal with use of Chebyshev bandpass filter in GStreamer; however, an echo is present and amplifies as time progresses. This echo includes a feedback that after a period of time is so loud that the receiving person is unable to hear voice

```
#Test source to file without gnuradio
#gst-launch-1.0 audiotestsrc num-buffers=440 ! audioconvert ! audio/x-raw,rate=44100,channels=2 ! queue ! mux. avimux name=mux !\
#filesink location=testAudio1.avi
```

Figure 5.22: GStreamer shell script to test creating audio tone and saving to file

To stream live audio the GStreamer script shown in Figure 5.24 was used along with the GRC flow graph shown in Figure 5.25 completes the transmit side.

```
#Live audio from mic to fifo file piped to gnuradio
#gst-launch-1.0 autoaudiosrc ! audiowsincband mode=band-pass lower-frequency=500 upper-frequency=2500 ! audioconvert ! mux. avimux name=mux !\
#filesink location=fifoAudio1.avi
```

Figure 5.23: GStreamer shell script to test creating live audio and saving to file

GStreamer makes it easy to enable the built in microphone by using autoaudiosrc. To ensure format compatibility, audioconvert is used. This GRC flow graph uses a throttle block which ended up being removed in later revisions. The throttle block literally throttles down the cpu which is not wanted. Figures 5.26 and 5.27 show the receive side GStreamer script and GRC flow graph. Once again GStreamer makes it easier on the user by providing the decodebin command which detects the incoming streams format and selects the correct decoder. Lastly, due to the noise, a band-pass filter was implemented which helped with the noise but not the echo present. To correct the echo issue, a keyboard shortcut was created to enable the user to key the microphone when needed and turn it off when idle.

```
gst-launch-1.0 autoaudiosrc ! audioconvert ! lamemp3enc target=1 bitrate=64 cbr=true ! filesink location=audioFifo.mp3
```

Figure 5.24: GStreamer shell script creating live audio and sending to GRC via FIFO pipe file

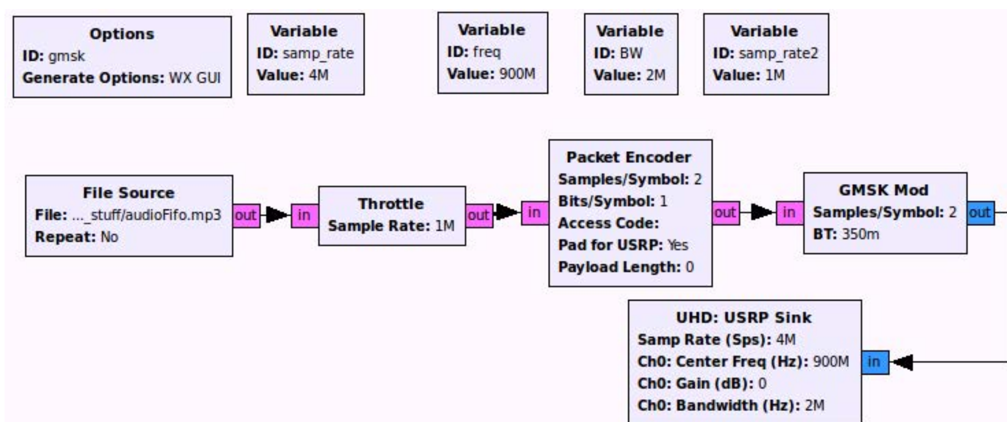

Figure 5.25: GRC flow graph receiving live audio from GStreamer via FIFO pipe file and transmitting

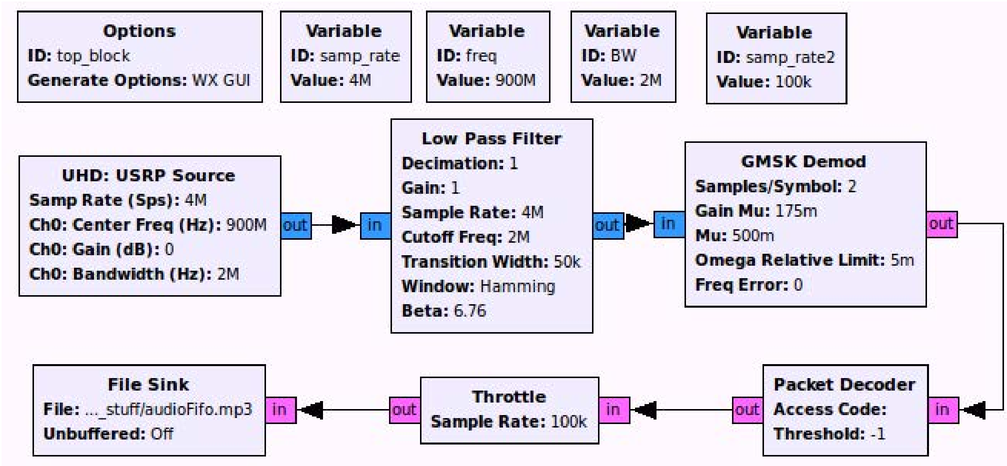

Figure 5.26: GRC flow graph receiving live audio from SDR and sending to GStreamer via FIFO pipe file

```
gst-launch-1.0 -e -v filesrc location=audioFifo.mp3 ! decodebin ! audioconvert !\
audiochebband mode=band-pass lower-frequency=500 upper-frequency=2000 poles=4 ! audioconvert ! autoaudiosink
```

Figure 5.27: GStreamer shell script to receive live audio from GRC and sending to laptop speaker hardware driver

5.3.6 Live Video with Test Audio

After succeeding in transmitting audio and video separately, the next goal was to combine the two using multiplexing and demultiplexing. Unfortunately, there were obstacles such as the program stopping on the receive side because of a premature end of stream (EOS) message. If GStreamer did not receive anything from the pipe it would assume EOS and quit. case GST_MESSAGE_EOS:

http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/chapter-bus.html

/* end-of-stream */

g_main_loop_quit (loop);

break;

Considering the problems found when trying to do both live audio and live video streaming together, live video with test audio was accomplished first. The

troubleshooting approach taken was to find the correct sample rate. The sample rate setting was the independent variable that was modified several times with different results. The best results were between 2.5M-5MHz but would achieve no longer than 13 seconds before error. USRP2 and USRP N210 both have 100MHz clocks. However, the first transmitting radio was the USRP2 so it was switched with another N210 to eliminate any interoperability issues. Next, since GStreamer is piped to the radio via GRC the timing synchronization was researched. The USRP Sink block in GNU Radio Companion (GRC) has an option to Sinc with the computer. After changing to this setting from the default of None, 49 seconds of streaming was achieved.

To be able to view when the error occurred, the benchmark_rx.py program was modified to kill the process upon the first error and display the time duration. The time until first error correlated to the time until EOS previously found. Using the benchmark made troubleshooting easier and helped find the efficient sample rate and bandwidth settings. These settings were changed on the GRC flow graph on the transmit side and over two minutes without error was achieved. Finally, after inputting these settings on the receive side GRC flow graph, over two minutes of continuous live video and test audio (tone) was achieved. Other settings besides sample rate, bandwidth, and sinc that were changed were the transmit and receive antenna gains and the windowing on the LPF. Antenna gains went from 0 to 0.05. Windowing went from Hamming to Rectangular. Please see Figures 5.28, 5.29, 5.30, 5.31 and 5.32 for GStreamer scripts, the terminal view of the receive script running, and GRC flow graph.

5.3.7 Live Video and Live Audio

After successfully completing the live video with test audio, the next step was to complete both live video and live audio together. Figure 5.33 is the first GStreamer

```
benchmark_rx.py    GStreamer2GMSK_30s.sh
#!/bin/sh

#This works!
gst-launch-1.0 avimux name=mux ! filesink location=pipe2 mux. v4l2src ! video/x-raw,width=640,height=480 ! \
queue2 ! videoconvert ! x264enc bitrate=498 ! mux. audiotestsrc ! queue2 !  mux.
```

Figure 5.28: GStreamer shell script creating live video & test audio and sending to GRC via FIFO pipe file

```
bench_mod5_tx.py    AVstreamRX_49s.sh
#!/bin/sh

#This works
gst-launch-1.0 filesrc location=pipe ! avidemux name=demux
demux.video_0 ! queue2 ! \
decodebin ! videoconvert ! queue2 ! autovideosink sync=false \
demux.audio_0 ! queue2 max-size-buffers=1024 ! audioconvert ! \
audioresample ! autoaudiosink sync=false
```

Figure 5.29: GStreamer shell script to receive live video & test audio from GRC and sending to video player and laptop speaker hardware driver

```
user@PKI346-LT03:~/GStreamer_stuff/17JUN14/49se
cs$ ./AVstreamRX_49s.sh
Setting pipeline to PAUSED ...
Pipeline is PREROLLING ...
Redistribute latency...
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstPulseSinkClock
Got EOS from element "pipeline0".
Execution ended after 0:02:53.696335664
Setting pipeline to PAUSED ...
Setting pipeline to READY ...
Setting pipeline to NULL ...
Freeing pipeline ...
user@PKI346-LT03:~/GStreamer_stuff/17JUN14/49secs$
```

Figure 5.30: Terminal execution of GStreamer receive shell script for live video and test audio

transmitter script that succeeded but with audio lagging video by approximately one second.

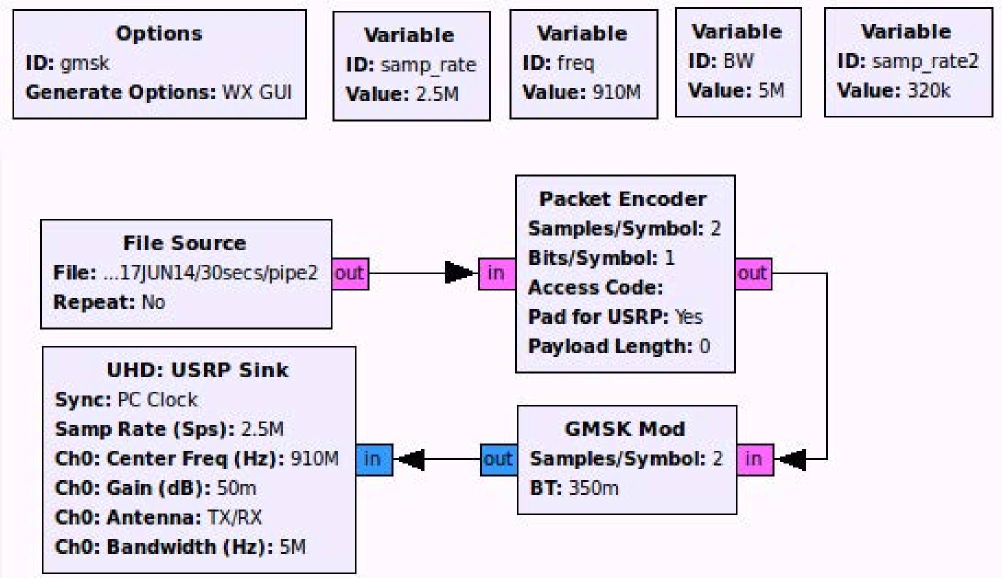

Figure 5.31: GRC flow graph receiving live video & test audio from GStreamer via FIFO pipe file and transmitting

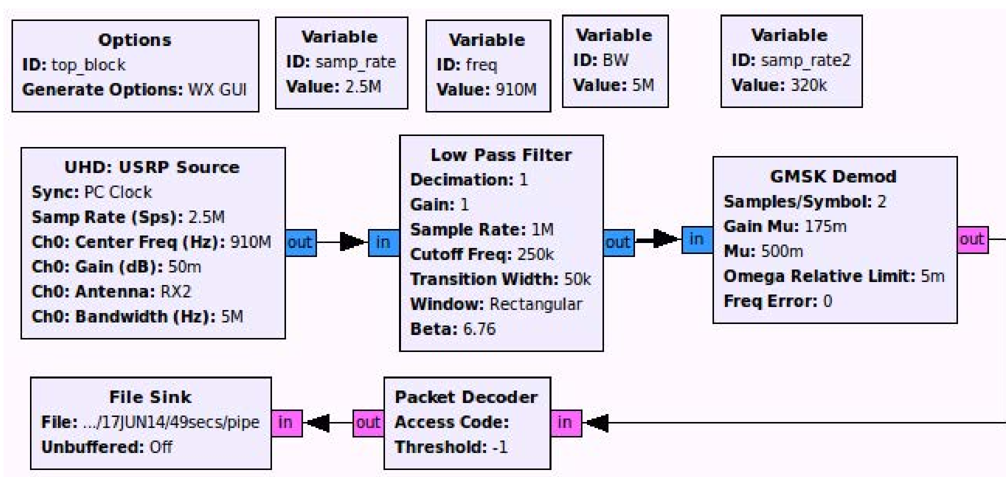

Figure 5.32: GRC flow graph receiving live video & test audio from SDR and sending to GStreamer via FIFO pipe file

```
#!/bin/sh
gst-launch-1.0 avimux name=mux ! filesink location=pipe2 mux. v4l2src ! \
video/x-raw,width=640,height=480 ! queue2 ! videoconvert ! x264enc bitrate=498 ! \
mux. autoaudiosrc ! queue2 ! audiowsincband mode=band-pass lower-frequency=500 upper-frequency=2500 ! \
audioconvert ! 'audio/x-raw,rate=44100,channels=1' ! mux.
```

Figure 5.33: GStreamer shell script creating live video & audio and sending to GRC via FIFO pipe file

5.4 Creating GRC Block for Spectrum Sensing and Channel Allocation

In order to take advantage of GRCs ability to use threading and parallel computing users have the choice of creating their own blocks. Blocks can be made using C++ or Python. GNU Radio makes this task easier by using the gr_modtool, which, creates the necessary directories and files needed. Also, GNU Radio provides a tutorial [30] that is very useful. When using Python it is useful to read the C++ functions that you are calling to understand how they work, especially previously made blocks. A good source to learn or refresh C++ is "A C++ Primer For Engineers" [31]. Files such as spectrum_sense.py and digital_bert.py are useful to understand since many of the same functions and methods will be used. Also, reading research papers such as [32] and [33] allows you to see how others have already done what you are trying to do.

After following the instructions in [30] to create the directories and files we edited the C++ or Python block file to do the intended function. Next, we edited the test file to enable testing the function. After editing test file, the XML file, which is created by gr_modtool, needs to be edited. The XML file creates the actual block you see in GRC. Once these steps are completed the user will go to the Build directory, created while following the steps in [30], and, do cmake, ldconfig, and make install. This will update the make file and add your block to GRC. While editing the C++ or Python file you can create as many inputs and outputs as needed. There are different types of blocks which include: synchronous, decimator, interpolator, hierarchical, and basic. A synchronous block was created because it will consume and produce an equal number of items per port, and may have any number of inputs or outputs. Defining the number and type of ports is done in the __init__ function as shown in Figure 5.34 below.

```
class sqpy(gr.sync_block):
    """
    docstring for block sqpy
    """
    def __init__(self):
        gr.sync_block.__init__(self,
            name="sqpy",
            in_sig=[numpy.float32],
            out_sig=[numpy.float32, numpy.int])
```

[output_port(0)_type(float), output_port(1)_type(int)]

Figure 5.34: Python snippet defining data and port type

Please notice the different types of ports defined in the example shown in Figure 5.34. This was only done because when writing the test module it will not accept more than one of the exact same block. The blocks are imported from the blocks module that comes with GNU Radio. In this case the output ports are connected to blocks.vector_sink_f and blocks.vector_sink_i, where _f is float and _i is integer. The function of the example given in [30] is a squaring function that takes the input, square it, and sends to a vector sink. This was the module that was modified to build the block needed. After changing the number of output ports the work function was changed to include a threshold that checks a variable before either squaring the inputs or adding them. This was meant to emulate a signal threshold from spectrum sensing, while the if-else statement is the channel allocation. The modified function is shown below in Figure 5.35.

```
def work(self, input_items, output_items):
    if input_items[:] >= 10:
            output_items[0][:] = input_items[0] * input_items[0]
            output_items[1][:] = input_items[0] * input_items[0]
            return len(output_items[0])
    else:
            output_items[0][:] = input_items[0] + input_items[0]
            output_items[1][:] = input_items[0] + input_items[0]
            return len(output_items[0])
```

Figure 5.35: Python function implementing threshold and channel allocation

While writing the QA test code it is helpful to know what you want the GRC

flow graph to be because it builds the flow graph using Python instead of GRC. The flow graph that was built before creating the QA test code is shown in Figure 5.36 below.

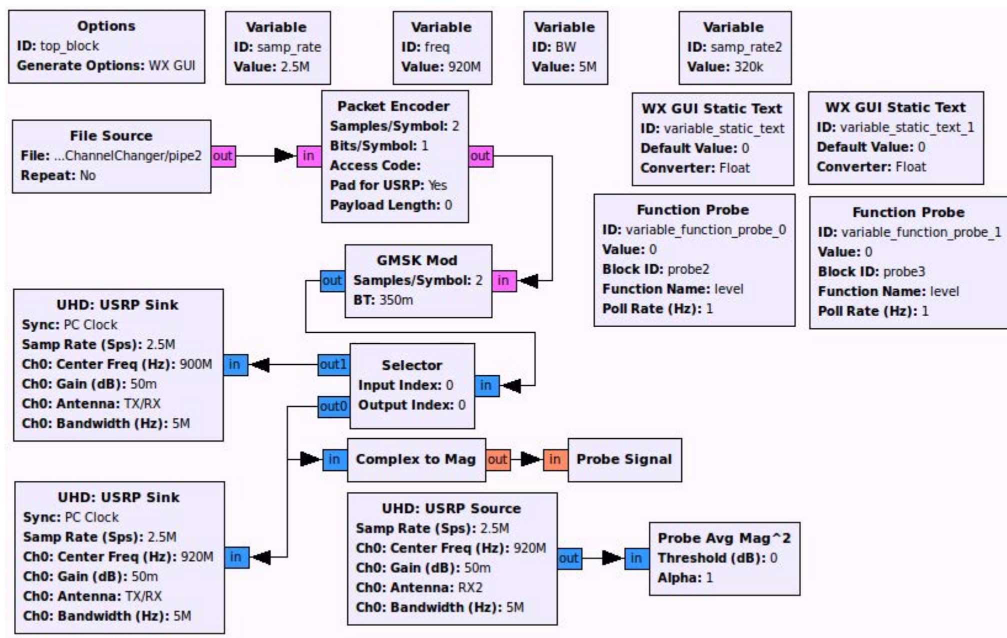

Figure 5.36: QA test GRC flow graph for visual aid

While connecting blocks in the QA test, all blocks connected need to have matching types and size of ports. Figure 5.37 shows the QA test work method with an example of how to modify the size. GNU Radio syntax states that if a module has different data types it will be stated as the suffix to the name of the block such as the vector sink examples shown in Figure 5.34 with declaring data types numpy.float32 and numpy.int. Numeric Python, known as NumPy, supplements basic Python by supporting linear algebra, Fourier transforms, random number capabilities and tools for integrating with C/C++ languages [34].

To run test the user can either go to the /build directory and either run:

```
$ make test
```

or the QA Python file from terminal using:

```
def test_001_oneXtwo (self):
    src_data = (10, 12)
    expected_result = (100, 144)
    src = blocks.vector_source_f(src_data)
    x = sqpy()
    dst = blocks.vector_sink_f()
    dstb = blocks.vector_sink_i(vlen=2)
    self.tb.connect(src,(x,0))
    self.tb.connect((x,0),dst)
    self.tb.connect((x,1),dstb)
    self.tb.run()
    result_data = dst.data()
    self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)
```

Figure 5.37: QA Python test module

```
$ sudo python foo.py
```

If the former is used then there is an output such as the one shown in Figure 5.38 & 5.39 below. Both successful and non-successful examples are shown. Notice how it states that the output received does not equal the expected output upon failure in Figure 5.39.

```
user@PKI346-LT03:~/gr-chSel/build$ sudo ctest -v -R qa --output-on-failure
Test project /home/user/gr-chSel/build
    Start 2: qa_sq
1/1 Test #2: qa_sq ............................   Passed    0.71 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.71 sec
```

Figure 5.38: Terminal execution of QA Python test - Success

```
user@PKI346-LT03:~/gr-chSel/build$ sudo ctest -v -R qa --output-on-failure
Test project /home/user/gr-chSel/build
    Start 2: qa_sq
1/1 Test #2: qa_sq ...........................***Failed    0.79 sec
F
======================================================================
FAIL: test_001_oneXtwo (__main__.qa_sq)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/user/gr-chSel/python/qa_sq.py", line 71, in test_001_oneXtwo
    self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)
  File "/usr/local/lib/python2.7/dist-packages/gnuradio/gr_unittest.py", line 87, in assertFloatTuplesAlmostEqual
    self.assertAlmostEqual (a[i], b[i], places, msg)
AssertionError: 100 != 20.0 within 6 places
```

Figure 5.39: Terminal execution of QA Python test - Failure

5.5 Live Audio, Spectrum Sensing, and Dynamic Channel Allocation

The final demo marks the culmination of learning Linux, GNU Radio, GNU Radio Companion, Python and C++ programming languages, and GStreamer multimedia framework. In this chapter we step through the main Python class, associated modules, GSstreamer script, and GRC flow graph. In the next section, the process of running the demo is described.

5.5.1 Python Script

5.5.1.1 Main

After defining variables the specsense module is called after the getavg module has calculated the average value of the input vector from the software defined radio (SDR) and converted to decibel. The specsense module takes 10,000 samples from getavg for each of three channels. After collecting the samples they are temporarily stored to three vectors, the averages of the vectors are then compared and the channel correlated to the lowest average is chosen. The above steps occur on the first iteration of the program and when the decibel threshold is exceeded. Next, it goes through a loop of switching between transmitting and receiving. This is done because when transmitting and receiving simultaneously the data from the receiver is erroneous. Next, the average value is compared to the previous value because a significant change in value usually indicates that the primary is transmitting again. Finally, if there is a significant change in value, and the decibel threshold is exceeded, while in the receive only loop, and not in the spectrum sensing mode, the spectrum sensing mode is initiated and the channel is changed. However, if those requirements are not met then the main loop continues. Below is a high level flow chart of the main class and the code can be found in the Appendix.

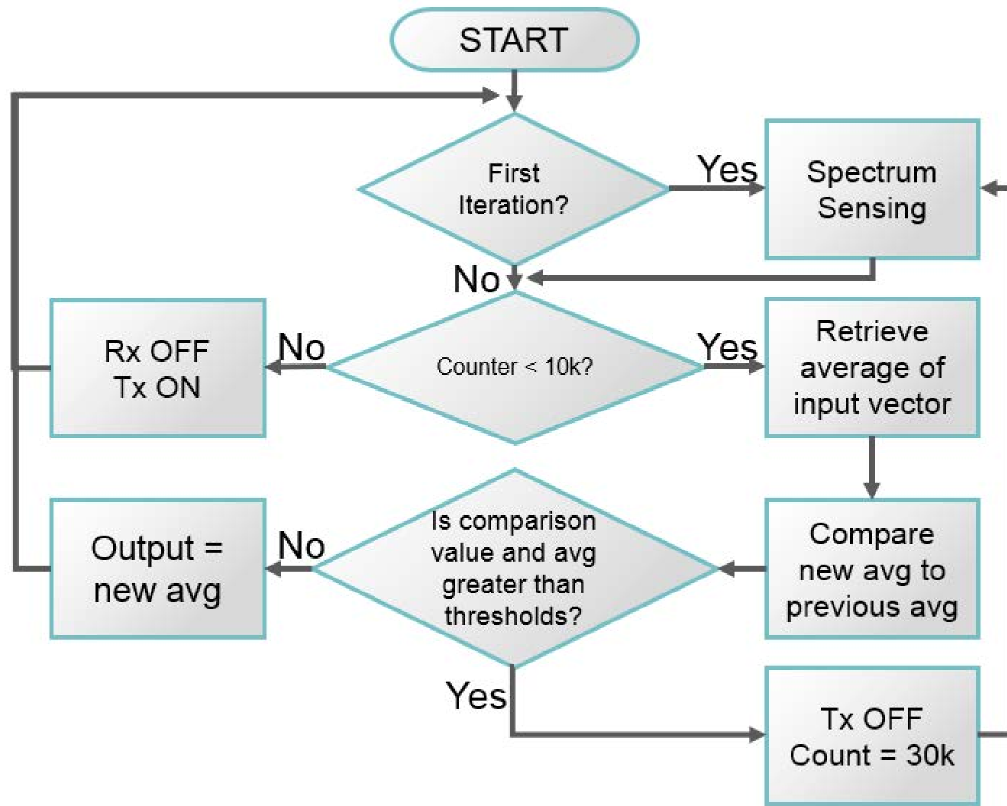

Figure 5.40: Main class flow chart

5.5.1.2 Get Average

The getavg module takes the input from the USRP which is the magnitude squared of the fft output [35]. Next, it calculates the average and converts it to decibel by using the formula shown in equation 5.1 where the average fft_output is divided by the usrp rate.

Also, noise floor calculation is shown in 5.2. Figure 5.41 is the flow chart for the getavg module.

$$10 * \log(\frac{avg_fft_output_magnitude_squared}{usrp_rate}) - noisefloor \tag{5.1}$$

$$10 * \log(\frac{min_fft_output_magnitude_squared}{usrp_rate}) \tag{5.2}$$

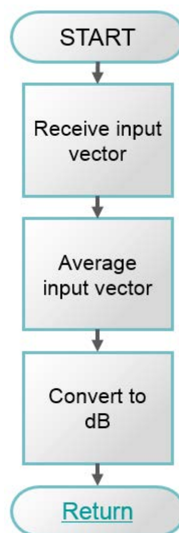

Figure 5.41: Get Average module flow chart

5.5.1.3 Spectrum Sense & Channel Allocation

The specsense module tunes the receiver to one of the three channels, depending on the counter, appends 10,000 values, received from getavg, to a temporary vector, and repeats for the remaining channels. Finally, the averages of the three vectors are compared, and the channel correlated to the lowest value is selected. The flow graph for specsense is shown in Figure 5.42 below.

5.5.1.4 GStreamer Live Audio

GStreamer is the multimedia framework used to process the raw audio input from the computer hardware, or encoded audio from the SDR, encode it or decode it respectively, and send to or receive from pipe connected to GRC. The following GStreamer script shown in Figure 5.43 is used to process and send a live audio stream.

1. Initialize application: **gst-launch-1.0**
2. Use internal microphone: **autoaudiosrc**

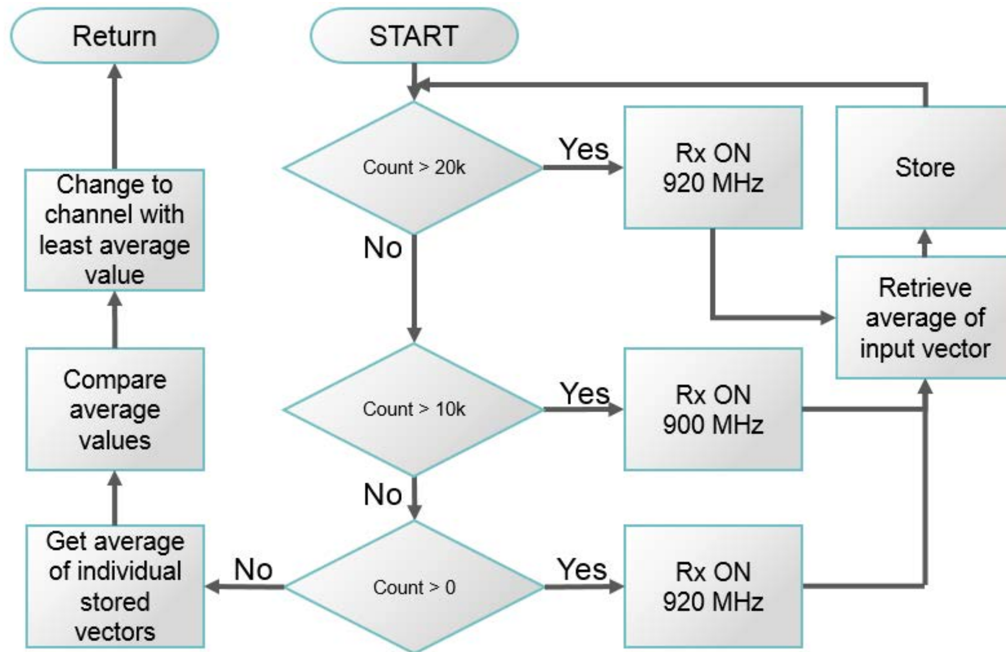

Figure 5.42: Spectrum Sensing module flow chart

```
gst-launch-1.0 autoaudiosrc ! audioconvert ! \
lamemp3enc target=1 bitrate=64 cbr=true ! \
filesink location=audioFifo.mp3
```

Figure 5.43: GStreamer shell script for live audio transmitted to GRC via FIFO pipe file

3. Prepare raw audio for conversion to different format: **audioconvert**

4. Convert to mp3: **lamemp3enc**

   a) Optimize for quality or bitrate: **target=1 (bitrate)**

   b) Set bitrate: **bitrate=64**

   c) Enforce constant bitrate encoding (CBR): **cbr=true**

5. Send to pipe connecting to GRC: **filesink location=audioFifo.mp3**

The following GStreamer script is used to process a live audio stream on the receive side.

```
gst-launch-1.0 filesrc location=audioFifo.mp3 ! decodebin ! \
queue2 max-size-buffers=1024 ! audioconvert ! \
audioresample ! autoaudiosink sync=false
```

Figure 5.44: GStreamer shell script to receive live audio from GRC and send it to the laptops speaker hardware driver

1. Initialize application: **gst-launch-1.0**
2. Receive from pipe connecting to GRC: **filesrc location=audioFifo.mp3**
3. Decode: **decodebin**
4. Use buffer because computer is slow: **queue2**
   a) Set a maximum size of buffer that allows correct playback with minimal delay: **max-size-buffers=1024**
5. Buffer raw audio for resampling and any data type conversion that are needed: **audioconvert**
6. Interpolate signal to fill in any missing information: **audioresample**
7. Play audio through built in speakers: **autoaudiosink**
   a) Ignore buffer time stamp, and clock, and play frame upon arrival (This helps to avoid frame dropping errors): **sync=false**

5.5.1.5 GNU Radio Companion

The GRC flow graph in Figure 5.45 has two parts to it, a transmit side, and a receive side. The transmit side starts with receiving the encoded stream from

GStreamer via the Linux pipe. Next, the data is packetized, modulated, and finally, transmitted. The receive side first receives data stream from SDR, and then converts stream to vector format. Next, a Fast Fourier Transform is done on the data and the magnitude squared value is sent on to the created block which decides whether or not to change the channel and if so, then which channel. In this configuration the Vector Sink block has no real task except to provide the output of the proprietary block a sink.

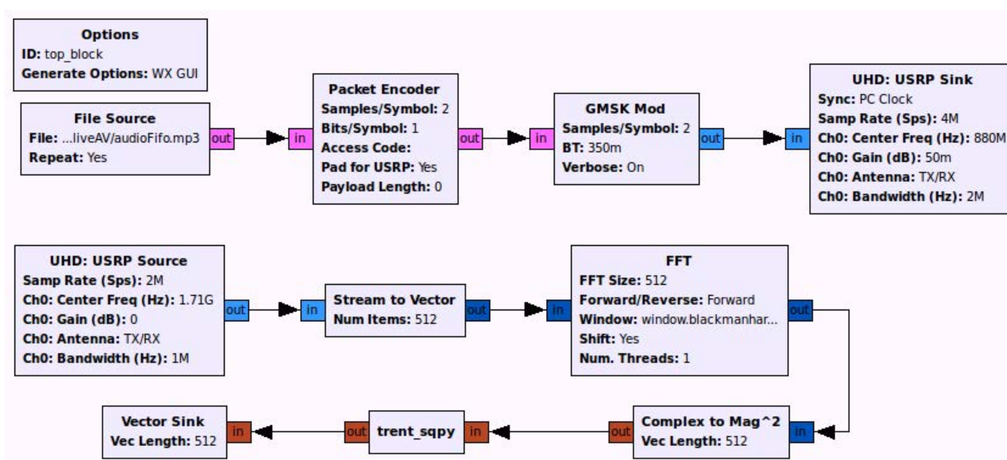

Figure 5.45: GRC flow graph with receive and transmit including custom made block

5.5.2 Running Demo

To run the demo the user needs a minimum of three nodes. The first node is the cognitive radio, the second is the live audio streaming receiver, and any other nodes are primaries. For the first and second nodes the user needs to be in the proper directories to run the GStreamer scripts and have GRC open. First, the user needs to run the GStreamer script and then the GRC flow graph, in order of first and then second node. Once live streaming is accomplished the user can run the benchmark_tx program on another node to act as a primary coming on. Upon transmitting from primary, the first node should detect, stop streaming,

scan spectrum, and change channel. Finally, the user needs to stop the GRC and GStreamer on second node and restart them with the same frequency that node one is using. The Python script for custom block is shown in Figures 5.46, 5.47, 5.48, 5.49, 5.50, & 5.51. Figure 5.52 shows the XML file to create the actual block that can be used in GRC.

5.5.3 Basic Measurements for Benchmark

When a channel is busy, SUs should sense a much higher power compared with an idle channel. In order to establish a benchmark for further implementations, we first set one SDR as a PU, and set the other two as SUs. When the PU is transmitting on a specific channel, we tune the SUs to measure the receiving power of that channel. The measurement is done for the spectrum from 824 MHz to 960 MHz with/without PU activity for 1,000 times. The average results are shown in Fig. 5.53. As we can see, receiving power is much higher with PU activity compared with the one without PU activity. The average values are used as benchmark for channel switching.

In the second step, we measure the receiving power on the adjacent channel of a channel with PU activity. The channels chosen for sensing started at 820 MHz and finished at 960 MHz incremented by 35 MHz. The channels used for primaries were 830 MHz, 900 MHz, and 960 MHz. The results in Fig. 5.54 are the average values of 1000 samples of each aforementioned channel. As we can see, adjacent channels within 25 MHz of 900 MHz are significantly affected. However, it is not the case even within 10 MHz when the PU transmits on 830 MHz. This information can be used to establish a minimum band gap between the current channel and the new channel chosen. More channel efficiency can be achieved if adaptive channel gap is adopted based on real-time channel quality.

```
#!/usr/bin/env python

import numpy
from gnuradio import gr, analog, blocks, digital, fft, uhd
import sys, time, math
from grc_gnuradio import blks2
import scipy
from getavg import getavg
import setup_usrp
from specense import specense

class sqpy(gr.sync_block):
    global x,y,r,rxfrequency,txfrequency,ctr,freq
    x=y=r=rxfrequency=txfrequency=ctr=freq=0
    def __init__(self):
        gr.sync_block.__init__(self,
            name="sqpy",
            in_sig=[(numpy.float32,512)],
            out_sig=[(numpy.float32,512)])
    def work(self, input_items, output_items):
    global x,r,rxfrequency,txfrequency,ctr,freq,y
    if x > 0 or y == 0:
        if y == 0:
            x = 30000
        avg = getavg(input_items)
        freq = specense(avg,x)
        x-=1
        r=512
        y+=1
        ctr=0
        return len(output_items[0])
    if r > 0:
        avg_in = getavg(input_items)
        q3 = avg_in
        r-=1
        return len(output_items[0])
```

Figure 5.46: Custom block Python script - 1

```
#Setup usrp TX-----------------------------------------------
if ctr == 5000:
    addr = uhd.device_addr()
    addr["name"] = "" #leave blank to use default
    src1 = uhd.usrp_source(addr,uhd.io_type.COMPLEX_FLOAT32,
                num_channels=1)
    src1.set_center_freq(200000000)
if ctr == 5050:
    addr = uhd.device_addr()
    addr["name"] = "" #leave blank to use default
    tx1 = uhd.usrp_sink(addr,uhd.io_type.COMPLEX_FLOAT32,
                num_channels=1)
    tx1.set_center_freq(freq)
    print "Transmit Freq: ", tx1.get_center_freq()
#Setup usrp RX -----------------------------------------------
if ctr == 10000:
    addr = uhd.device_addr()
    addr["name"] = "" #leave blank to use default
    tx1 = uhd.usrp_sink(addr,uhd.io_type.COMPLEX_FLOAT32,
                num_channels=1)
    txfrequency=tx1.get_center_freq()
    tx1.set_center_freq(1700000000)
    time.sleep(0.05)
if ctr == 10100:
    addr = uhd.device_addr()
    addr["name"] = "" #leave blank to use default
    src1 = uhd.usrp_source(addr,uhd.io_type.COMPLEX_FLOAT32,
                num_channels=1)
    src1.set_center_freq(freq)
    print "rx freq is: ",src1.get_center_freq()
    ctr=0
ctr+=1
```

Figure 5.47: Custom block Python script - 2

5.5.4 Video Streaming Application

In this preliminary implementation, a one-way video streaming service between two SUs is provided based on the AC-MWN architecture using 900 MHz spectrum. The third SDR functions as a PU with arbitrary activities. For better service, channel sensing/switching need to be transparent to users. Therefore, a maximum of approximately 100 *ms* delay ensures transparency for most users [36]. However,

```python
#-------------------------------------------------------------
avg_in = getavg(input_items)
q3 = avg_in
comp = numpy.average(output_items[0][:])
comparing = numpy.abs(comp-q3)
y+=1
if  comparing > 15 and x==0 and y > 1000 and q3 > 50 and ctr in range(100,5000):
    out = q3
    print "         CHANGING CHANNEL"
    print ""
    output_items[0][:] = q3
    x=30000
    #------turn tx off------------------------------------
    addr = uhd.device_addr()
    addr["name"] = "" #leave blank to use default
    tx1 = uhd.usrp_sink(addr,uhd.io_type.COMPLEX_FLOAT32,
                num_channels=1)
    tx1.set_center_freq(1710000000)
    print "Transmit Freq: ", tx1.get_center_freq()
    print ""
    time.sleep(0.25)
    freq=specense(q3,x)
    return len(output_items[0])

else:
    output_items[0][:] = q3
    return len(output_items[0])
```

Figure 5.48: Custom block Python script - 3

```python
import numpy, scipy
from numpy.fft import fft
from numpy import array
from cmath import exp, pi

def getavg(input_items):
    m = []
    m = input_items[0][0]
    avg_m = numpy.average(m)
    q3 = 10*numpy.log10(avg_m/195312) + 120
    return q3
```

Figure 5.49: Custom block Python script - 4

neither the transmitter nor the receiver notices any interruption due to background channel sensing/switching. In the application, we adopt GStreamer as the applica-

```
#!/usr/bin/env python

from gnuradio import gr, analog, blocks, digital, fft, uhd
import numpy, scipy
from numpy.fft import fft
from numpy import array
from cmath import exp, pi
import sys, time

global count,l2,l1,l0,L2,L1,L0,rxFreq2,rxFreq1,rxFreq0
l2 = l1 = l0 = L2 = L1 = L0 = []
rxFreq2 = rxFreq1 = rxFreq0 = 0

def spectrumsense(avg,x):
    global count,l2,l1,l0,L2,L1,L0,rxFreq2,rxFreq1,rxFreq0
    count = x-1
    if count < 0:
        count = 30000
        return
    if count == 30000:
        # setup usrp ----------------------------------------------
        #..RX..
        addr = uhd.device_addr()
        addr["name"] = "" #leave blank to use default
        src1 = uhd.usrp_source(addr,uhd.io_type.COMPLEX_FLOAT32,
                    num_channels=1)
        src1.set_center_freq(920000000)
        rxFreq0 = src1.get_center_freq()
        count-=1
        return count
    elif count >= 20000 and count < 30000 :
        q3 = avg
        s0 = q3
        l0.append(s0)
        if count == 20000:
            L0=numpy.mean(l0)
            # setup usrp ------------------------------------------
            #..RX..
            addr = uhd.device_addr()
            addr["name"] = "" #leave blank to use default
            src1 = uhd.usrp_source(addr,uhd.io_type.COMPLEX_FLOAT32,
                    num_channels=1)
            src1.set_center_freq(900000000)
            rxFreq1 = src1.get_center_freq()
        count-=1
        return count
```

Figure 5.50: Custom block Python script - 5

tion programming interface (API) to capture, encode and pipe Audio/Video (AV) to GRC. Fig. 5.55 shows the video stream application. The computer on the left captures live video through webcam. Signal is transmitted from SU_1 to SU_2. Live video streaming is shown on the right-hand-side computer.

```
elif count >= 10000 and count < 20000:
    q3 = avg
    s1 = q3
    l1.append(s1)
    if count == 10000:
        L1=numpy.mean(l1)
        # setup usrp ---------------------------------------------------
        #..RX..
        addr = uhd.device_addr()
        addr["name"] = "" #leave blank to use default
        src1 = uhd.usrp_source(addr,uhd.io_type.COMPLEX_FLOAT32,
                num_channels=1)
        src1.set_center_freq(880000000)
        rxFreq2 = src1.get_center_freq()
    count-=1
    return count
elif count >= 0 and count < 10000:
    q3 = avg
    s2 = q3
    l2.append(s2)
    if count == 0:
        L2=numpy.mean(l2)
        print ""
        print "L2,L1,L0: ",L2,L1,L0
        print ""
        if L2 < L1 and L2 < L0:
            # setup usrp -----------------------------------------------
            #..RX..
            addr = uhd.device_addr()
            addr["name"] = "" #leave blank to use default
            src1 = uhd.usrp_source(addr,uhd.io_type.COMPLEX_FLOAT32,
                    num_channels=1)
            src1.set_center_freq(rxFreq2)
        elif L1 < L2 and L1 < L0:
            # setup usrp -----------------------------------------------
            #..RX..
            addr = uhd.device_addr()
            addr["name"] = "" #leave blank to use default
            src1 = uhd.usrp_source(addr,uhd.io_type.COMPLEX_FLOAT32,
                    num_channels=1)
            src1.set_center_freq(rxFreq1)
        elif L0 < L2 and L0 < L1:
            # setup usrp -----------------------------------------------
            #..RX..
            addr = uhd.device_addr()
            addr["name"] = "" #leave blank to use default
            src1 = uhd.usrp_source(addr,uhd.io_type.COMPLEX_FLOAT32,
                    num_channels=1)
            src1.set_center_freq(rxFreq0)
        print ""
        print "freq is now:  ",src1.get_center_freq()
        print ""
        time.sleep(10)
    count-=1
    return count
```

Figure 5.51: Custom block Python script - 6

```xml
<?xml version="1.0"?>
<block>
  <name>trent_sqpy</name>
  <key>trent_sqpy</key>
  <category>trent</category>
  <import>import trent</import>
  <make>trent.sqpy()</make>

  <sink>
    <name>in</name>
    <type>float</type>
    <vlen>512</vlen>
  </sink>

  <source>
    <name>out</name>
    <type>float</type>
    <vlen>512</vlen>
  </source>
</block>
```

Figure 5.52: Custom block XML code

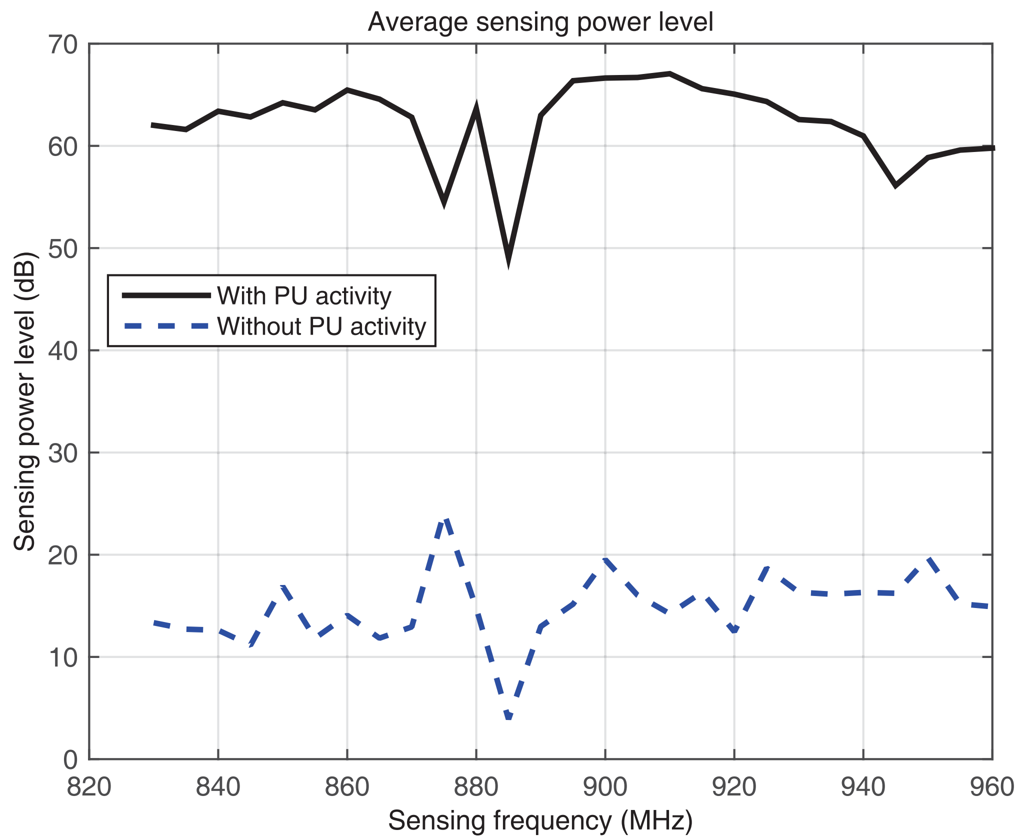

Figure 5.53: Threshold detection sensing plot

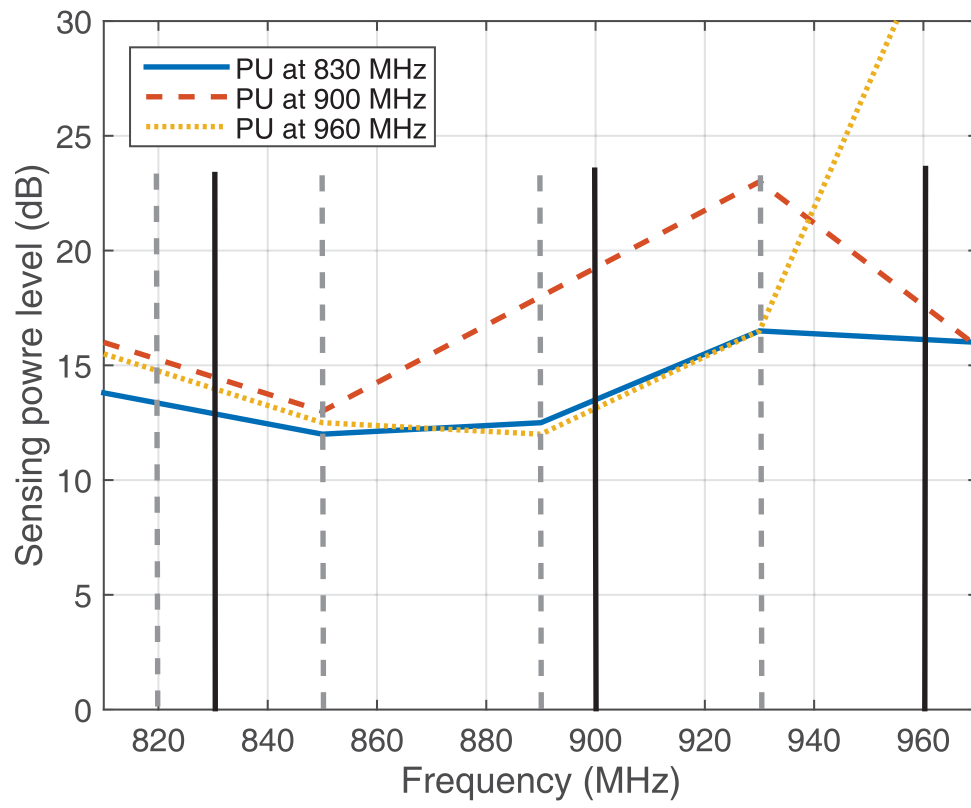

Figure 5.54: Adjacent channel sensing plot

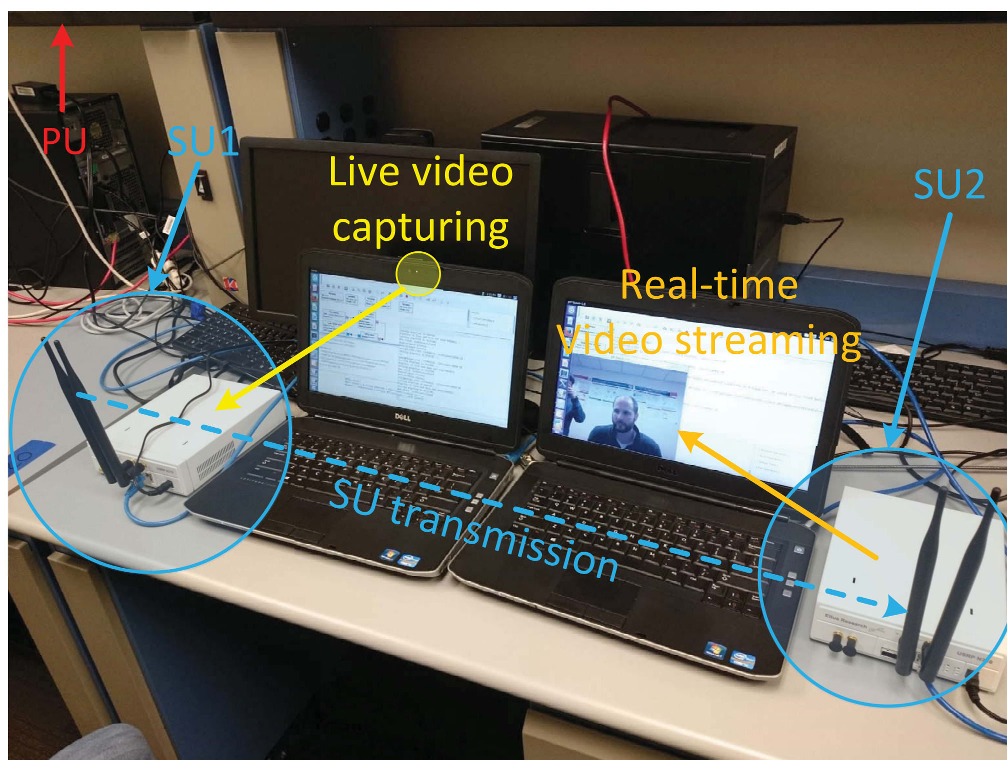

Figure 5.55: Live video streaming hardware setup and screen shot

Chapter 6

Conclusion

Multi-hop wireless cognitive radio networking (MWCRN) is a useful tool that will help alleviate congested channels. This congestion is caused by limited spectrum along with increasing demand and node density. However, researchers need testbeds to compare theory and reality, this thesis is an accumulation of work done in order to create a multi-hop wireless cognitive radio network testbed. Accomplishing this goal required a myriad of software tools including Ubuntu (Linux), GNU Radio, GStreamer, MATLAB integrated development environment (IDE) and BASH. Programming languages included Python, C++, XML, GStreamer BASH Scripting, Linux BASH Scripting, MATLAB and GNU Radio Companion guided user interface (GUI) IDE. We also used the USRP2 and USRPN210 software defined radios equipped with WBX RF Daughterboards and VERT900 antennas which are all Ettus Research products. Overall, considerable experience and technical proficiency has been gained and documented to give researchers the basic tools needed to execute testing, evaluation and observation steps of their empirical studies.

GNU Radio Companion is a great graphical programming tool that makes it easy to work with software defined radios, but it is the ability to multithread and

use multiple cores that is most useful when trying to simultaneously transmit and receive. This is accomplished by its thread-per-block scheduler. Most of the early work was done by modifying benchmark programs using Python but in order to utilize the scheduler it was necessary to build customized blocks to use with GRC. The final demo was done this way and was run along with GStreamer at the physical layer handling the audio-video processing.

GStreamer is a low-level multimedia development framework written in C that facilitates programmers with adding physical layer functionality to their applications such as providing encoding, decoding, containerization, sources, sinks and filters. Encoding and decoding formats include mp3, mpeg4, H.264 and others, while types of containers include avi, mkv, ogg, mov and others. Sources and sinks enable I/O functionality with the hardware drivers such as the microphone and webcam. Common filtering objects include low pass, high pass, Chebyshev, finite impulse response and multiplexing. GStreamer communicates with GNU Radio through a Linux First In, First Out (FIFO) pipe file.

Ettus Research's USRP software defined radios have a field programmable gate array (FPGA) that enable the user to quickly flash different configurations using a program such as GNU Radio. Other notable components embedded on the motherboard are the analog-to-digital converter (ADC) and digital-to-analog converter (DAC) and clock. The USRP has a modular analog component called a RF Daughterboard. Each daughterboard handles a certain range of spectrum. Particularly, the WBX-120 RF Daughterboard is capable of full duplex communications between 50 - 2200 MHz with 120 MHz of bandwidth. Finally, the omni-directional VERT900 antennas are compatible with the WBX-120 with ranges of 824 - 960 MHz and 1710 - 1990 MHz.

Experiments completed, starting with running examples provided, one-way and half-duplex transmissions, then culminated with live video streaming with

channel sensing and dynamic allocation. As shown in Figures 5.53 and 5.54, physical testing produces results that deviate from ideal, giving more accurate local environmental parameters than virtual testing. When looking at Figure 5.53 it is obvious that not every channel will have the same threshold value for calculating when to change to another channel. Also, considering some multi-hop wireless cognitive networking radios will be mobile, researchers will need to either run several tests or create algorithms that implement discrete sampling to frequently update the threshold values.

Figure 5.54 shows that some adjacent channels are affected more than others. Also, it shows that adjacent channels can receive a stronger signal than the center frequency channel. These observations indicate that supplementary routines may need to be created to avoid false positive primary detection.

Many failed or inefficient attempts forced us to refine settings and research different tools such as GStreamer. A good example was using the Throttle block in GRC because without it the experiment would not work but later we found that changing the sample rate and bandwidth made the block extraneous. Creating blocks expands the GRC library and allows for more complicated configurations. However, we suggest that the blocks be simple, modular and with many comments in consideration of other researchers who might be able to use them in their configuration.

This thesis has shown the progress and process of creating a testbed for Multi-hop cognitive radio networks. With more time the final demo would be improved upon by synchronizing the transmitter and receiver so that the receiver would automatically change channels along with the transmitter. Also, eliminating delays, and audio-video synchronization would be done. Many of the blocks in GRC require more knowledge in digital signal processing in order to use them.

Bibliography

[1] T. Evans, K. Tossou, F. Ye, Z. Shu, Y. Qian, Y. Yang, and H. Sharif. A new architecture for application-aware cognitive multihop wireless networks. *IEEE Wireless Communications*, 23(1):120–127, February 2016. 1

[2] Feng Ye, Jiazhen Zhou, Yaoqing Yang, H. Sharif, and Y. Qian. Constructing backbone of a multi-hop cognitive radio network with channel bonding. In *2012 IEEE Global Communications Conference (GLOBECOM)*, pages 5596–5601, Dec 2012. 1

[3] F. Ye, J. Zhou, Y. Qian, and R. Q. Hu. Application-aware routing for multi-hop cognitive radio networks with channel bonding. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 4372–4377, Dec 2013. 1

[4] A. Osseiran, V. Braun, T. Hidekazu, P. Marsch, H. Schotten, H. Tullberg, M. A. Uusitalo, and M. Schellman. The foundation of the mobile and wireless communications system for 2020 and beyond: Challenges, enablers and technology solutions. In *2013 IEEE 77th Vehicular Technology Conference (VTC Spring)*, pages 1–5, June 2013. 2.1

[5] Lili Wei, Rose Qingyang Hu, Yi Qian, and Geng Wu. Energy efficiency and spectrum efficiency of multihop device-to-device communications underlaying cellular networks. *IEEE Transactions on Vehicular Technology*, 65(1):367–380, 2016. 2.1

[6] B. Benmammar and A. Amraoui. *Radio Resouce Allocation and Dynamic Spectrum Access*. Wiley, Hoboken, NJ. 2.1

[7] Ian F. Akyildiz, Won-Yeol Lee, Mehmet C. Vuran, and Shantidev Mohanty. Next generation/dynamic spectrum access/cognitive radio wireless networks: A survey. *Comput. Netw.*, 50(13):2127–2159, September 2006. 2.1

[8] Kejie Lu, Yi Qian, Hsiao-Hwa Chen, and Shengli Fu. Wimax networks: from access to service platform. *IEEE network*, 22(3), 2008. 2.2

[9] Zhi Tian and Georgios B Giannakis. A wavelet approach to wideband spectrum sensing for cognitive radios. In *Cognitive Radio Oriented Wireless Networks and Communications, 2006. 1st International Conference on*, pages 1–5. IEEE, 2006. 2.3

[10] J. A. Bazerque and G. B. Giannakis. Distributed spectrum sensing for cognitive radio networks by exploiting sparsity. *IEEE Transactions on Signal Processing*, 58(3):1847–1862, March 2010. 2.3

[11] Robert Qiu, Nan Guo, Husheng Li, Zhiqiang Wu, Vasu Chakravarthy, Yu Song, Zhen Hu, Peng Zhang, and Zhe Chen. A unified framework for cognitive radio, cognitive radar, and electronic warfaretutorial, theory, and multi-ghz wideband testbed. *Sensors*, 9(8):6530, 2009. 2.3

[12] C. R. Stevenson, G. Chouinard, Z. Lei, W. Hu, S. J. Shellhammer, and W. Caldwell. Ieee 802.22: The first cognitive radio wireless regional area network standard. *IEEE Communications Magazine*, 47(1):130–138, January 2009. 2.3

[13] Z. Shu, J. Zhou, Y. Qian, and R. Q. Hu. Adaptive channel allocation and routing in cognitive radio networks. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 4542–4547, Dec 2013. 2.3

[14] Zhihui Shu, Yi Qian, Yaoqing Yang, and H. Sharif. Channel allocation and multicast routing in cognitive radio networks. In *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1703–1708, April 2013. 2.3

[15] S. Singla and S. Jain. Comparison of routing protocols of manet in real world scenario using ns3. In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pages 543–549, July 2014. 2.4.1

[16] F. Bertocchi, P. Bergamo, G. Mazzini, and M. Zorzi. Performance comparison of routing protocols for ad hoc networks. In *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, volume 2, pages 1033–1037 Vol.2, Dec 2003. 2.4.1

[17] Moshaddique Al Ameen, S. M. Riazul Islam, and Kyungsup Kwak. Energy saving mechanisms for mac protocols in wireless sensor networks. *International Journal of Distributed Sensor Networks*, 6(1):163413, 2010. 2.4.2

[18] K. R. Chowdhury and I. F. Akyildiz. Cognitive wireless mesh networks with dynamic spectrum access. *IEEE Journal on Selected Areas in Communications*, 26(1):168–181, Jan 2008. 2.4.3

[19] Ian F. Akyildiz, Won-Yeol Lee, and Kaushik R. Chowdhury. Crahns: Cognitive radio ad hoc networks. *Ad Hoc Netw.*, 7(5):810–836, July 2009. 2.4.3

[20] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. Optimized link state routing protocol for ad hoc networks. In *Proceedings. IEEE International Multi Topic Conference, 2001. IEEE INMIC 2001. Technology for the 21st Century.*, pages 62–68, 2001. 2.4.3.1

[21] G. M. Zhu, I. F. Akyildiz, and G. S. Kuo. Stod-rp: A spectrum-tree based on-demand routing protocol for multi-hop cognitive radio networks. In *IEEE GLOBECOM 2008 - 2008 IEEE Global Telecommunications Conference*, pages 1–5, Nov 2008. 2.4.3.3

[22] V.P Patil. Efficient aodv routing protocol for manet with enhanced packet delivery ratio and minimized end to end delay. *International Journal of Scientific and Research Publications*, 2(8):197–201, 2012. 2.4.4

[23] Kamal Deep Singh, Priyanka Rawat, and Jean-Marie Bonnin. Cognitive radio for vehicular ad hoc networks (cr-vanets): approaches and challenges. *EURASIP Journal on Wireless Communications and Networking*, 2014(1):49, 2014. 2.4.5

[24] Ieee standard for information technology–telecommunications and information exchange between systems wireless regional area networks (wran)–specific requirements - part 22: Cognitive wireless ran medium access control (mac) and physical layer (phy) specifications: Policies and procedures foroperation in the tv bands amendment 1: Management and control plane interfaces and procedures and enhancement to the management information base (mib). *IEEE Std 802.22a-2014 (Amendment to IEEE Std 802.22-2011)*, pages 1–519, May 2014. 3.1

[25] R. Otieno U. Mbeche-Smith A. Klen-Amin M. Kamiya R. Stren P. McCarney G. Tipple S. Balakrishnan V. Castan-Broto E. Pieterse B. Stiftel S. McCord-Smith B. Roberts T. Kanaley M. Cohen E. Moreno, B. Arimah. 2016. 4

[26] J. Mitola. Cognitive radio for flexible mobile multimedia communications. In *Mobile Multimedia Communications, 1999. (MoMuC '99) 1999 IEEE International Workshop on*, pages 3–10, 1999. 4

[27] Martin Powell, Copps and Adelstein. Docket no. 03-222 notice of proposed rule making and order. 2003. 4

[28] www.ettus.com. https://www.ettus.com/. 4

[29] www.ubuntu.com. https://www.ubuntu.com/. 4

[30] Gnuradio out-of-tree modules. http://gnuradio.org/redmine/projects/gnuradio/wiki/OutOfTreeModules. 5.4, 5.4

[31] K. Ponnambalam and T. Alguindigue. *A C++ Primer for Engineers: An Object-oriented Approach*. Number v. 1. McGraw-Hill Company, 1997. 5.4

[32] Z. Yan, Z. Ma, H. Cao, G. Li, and W. Wang. Spectrum sensing, access and coexistence testbed for cognitive radio using usrp. In *2008 4th IEEE International Conference on Circuits and Systems for Communications*, pages 270–274, May 2008. 5.4

[33] R. A. Rashid, M. A. Sarijari, N. Fisal, S. K. S. Yusof, and S. H. S. Ariffin. Enabling dynamic spectrum access for cognitive radio using software defined radio platform. In *2011 IEEE Symposium on Wireless Technology and Applications (ISWTA)*, pages 180–185, Sept 2011. 5.4

[34] www.numpy.org. http://www.numpy.org/. 5.4

[35] B. Reynwar T. Rondeau, C. Kuethe. usrp_spectrum_sense.py. https://github.com/gnuradio/gnuradio/blob/master/gr-uhd/examples/python/usrp_spectrum_sense.py, 2015. 5.5.1.2

[36] M. Baldi and Y. Ofek. End-to-end delay analysis of videoconferencing over packet-switched networks. *IEEE/ACM Transactions on Networking*, 8(4):479–492, Aug 2000. 5.5.4