Cognitive Radio Network Testbed (CORNET):

Design, Deployment, Administration and Examples

Daniel Richard DePoy

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

In

Electrical Engineering

Tamal Bose

Carl B. Dietrich

Timothy R. Newman

May 29 2012

Blacksburg, Virginia

Keywords: Cognitive Radio, GnuRadio, Software Defined Radio, Testbeds, Dynamic Spectrum Access

Cognitive Radio Network Testbed (CORNET): Design, Deployment, Administration and Examples

Daniel Richard DePoy

ABSTRACT

Development of Cognitive Radio (CR) applications, which rely on a radio's ability to adapt intelligently to it's spectral surroundings will soon make the all important technological jump from research interest to systems integration, as demand for highly adaptive wireless applications expand.

VT-CORNET (Virginia Tech – Cognitive Radio Network Testbed) is a unique testbed concept, designed to facilitate this technology leap by offering researchers – both local and remote – the opportunity to conduct CR experiments on an installed infrastructure of highly flexible radio nodes. These nodes – 48 in total – are distributed throughout four floors of a building on the Virginia Tech campus, and provide researchers with diverse options in terms of channel conditions and deployment scenarios. The radios themselves consist of the widely used USRP2 Software

Defined Radio (SDR) platform, coupled to a centrally located cluster of rack servers – which provide a high performance GPP environment for real-time software based signal processing. VT-CORNET is specially licensed to operate our low-power nodes over a broad range of frequencies, which provides researcher the opportunity to conduct experiments on live spectrum – in the presence of real primary users. Testbeds are a widely used tool in the wireless and networking fields, and VT-CORNET expands the concept through a focus on CR research and education.

This thesis describes the construction and deployment of the CORNET testbed in detail. Specific contributions made to the testbed include the design and implementation of the management network, as well as the initial deployment of the SDR nodes in the ceiling. In addition, this thesis describes the administration and management of the CORNET GPP cluster, and provides a instructions for the basic usage of CORNET from an administrative and user perspective. Finally, this thesis describes a number of custom SDR waveforms implemented on CORNET which demonstrate the utility of the testbed for cognitive radio applications.

DEDICATION

To my dog, Socrates

ACKNOWLEDGEMENTS

I would like to thank everyone. Dr. Bose, Dr. Dietrich, Dr. Newman, and the rest of the faculty and staff with Wireless at Virginia Tech. I would like to thank my ECE advisors (Cynthia, et.al) and everyone who has given me guidance. I would also like to thank my parents and family for their support.

Table of Contents

1 Introduction.....1

2 Background.....4

2.1 Software Defined Radio.....4

2.2 Cognitive Radio.....6

2.3 Testbeds.....7

3 Cognitive Radio Network Testbed.....10

3.1 Overview.....10

3.1.1 Motivation 11

3.1.2 Intended Usage.....12

3.1.2 Hardware.....14

3.1.3 Software.....17

3.1.4 Remote Access.....18

3.2.1 User Plane.....20

3.2.2 Radio Plane.....21

3.2.3 Network Plane.....22

3.3 Administration.....22

3.3.1 Re-Imaging the Cluster.....23

3.3.2 Image Server Overview 25

3.3.3 Intelligent Platform Management Interface.....28

3.3.4 Debian Preseeding.....31

3.3.5 Breaking the PXE Boot Loop.....35

3.3.6 Local Repository Mirror.....39

3.3.7 Non-Repository Software.....40

3.3.8 LDAP and PAM User Authentication.....43

3.3.9 IP Tables and Secure Shell Tunneling.....44

3.3.10 Web Server.....46

4 CORNET DSA Example.....51

4.1 Overview.....51

4.2 GnuRadio Waveform.....52

4.2.1 Physical Layer.....52

4.2.2 Data Link Layer.....53

4.2.3 Application Layer.....55

4.3 Dynamic Spectrum Access Protocol.....55

4.3.1 Spectrum Sensing.....56

4.3.2 Peer Discovery and Coordination.....59

4.3.3 Channel Selection Behavior.....59

4.4 Webcam/GnuRadio Interface.....61

4.4.1 Gstreamer.....62

4.4.2 FIFO FIlesink....64
4.5 Summary of Operation....64
4.5.1 Synchronization....65
4.5.2 Convergence....66
4.5.3 Transmission....67
4.5.4 Channel Evacuation....68
4.5.5 Re-Convergence....69
5 Using CORNET....71
5.1 Overview....71
5.2 Logging Into CORNET....72
5.2.1 Linux/Unix Secure Shell....72
5.2.2 Windows Secure Shell....75
5.2.3 NX Remote Desktop Client....79
5.3 User Directory Operations....84
5.3.1 Directories and Permissions....85
5.3.2 File Management....86
5.4 Example Waveforms and Tools....88
5.4.1 USRP2 Diagnostics....89
5.4.2 GnuRadio Tools....92
5.4.3 Benchmark Example Waveforms....96

5.4.4 GnuRadio Companion.....99

5.4.5 Using OSSIE.....103

5.5 Installing Custom Software.....106

5.5.1 Dependencies.....107

5.5.2 Compiling Source Code.....107

5.5.2 Installing Software on Several Nodes.....108

5.6 Other CORNET Demonstrations.....109

5.6.1 Multichannel Ad-Hoc DSA Demonstration.....109

5.6.2 USRP2 Spectrum Sensing.....113

6 Conclusions and Future Work.....116

References.....118

Appendix A.....123

List of Figures

Figure 1. CORNET Node Distribution.....10

Figure 2. CORNET GPP Cluster.....15

Figure 3. USRP2 in Ceiling.....16

Figure 4. CORNET Architecture.....20

Figure 5. IPMI Web Server User Interface.....31

Figure 6. CORNET Web Page.....46

Figure 7. CORNET Admin Web Page.....47

Figure 8. Node Administration Web Page.....48

Figure 9. User Create Web Page.....48

Figure 10. DSA Application Block Diagram.....62

Figure 11. DSA Synchronization.....66

Figure 12. DSA Convergence.....67

Figure 13. Steady-State Transmission.....68

Figure 14. DSA Channel Evacuation.....69

Figure 15. DSA Re-Convergence 70

Figure 16. Accept SSH RSA Key.....73

Figure 17. Password Prompt.....73

Figure 18. PuTTY Configuration.....77

Figure 19. Accept SSH Certificate (PuTTY).....78

Figure 20. PuTTY Password Prompt....78
Figure 21. NX Client Session Configuration....80
Figure 22. NX Client Desktop Setup....81
Figure 23. NX Client Login Prompt....82
Figure 24. NX Client RSA Key Exchange....82
Figure 25. NX Remote Desktop....83
Figure 26. NX Client Session Exit Prompt....84
Figure 27. Examples and Tools....89
Figure 28. USRP2 IP Address....90
Figure 29. USRP2 Ping Response....91
Figure 30. USRP2 UHD Configuration....92
Figure 31. Signal Generator GUI....94
Figure 32. FFT Plot Tool....95
Figure 33. RX Benchmark Output....98
Figure 34. TX Benchmark Output....98
Figure 35. GnuRadio Companion....99
Figure 36. FFT Waveform....100
Figure 37. USRP Source Properties....101
Figure 38. Generate the Flow Graph....102
Figure X. Running FFT Waveform....103

Figure 39. Eclipse With OEF.....105

Figure 40. Install OEF Plugins.....106

Figure 41. Ad-Hoc DSA Waveform.....110

Figure 42. Ad-Hoc DSA Protocol.....113

Figure 43. Spectrum Sensing Flowgraph.....115

List of Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
ACK	Acknowledgment
CORNET	Cognitive Radio Network Testbed
CPU	Central Processing Unit
CR	Cognitive Radio
CSMA/CD	Carrier Sensing Multiple Access/Collision Detection
CTS	Clear To Send
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DSA	Dynamic Spectrum Access
DSP	Digital Signal Processing
FCC	Federal Communications Commission
FEC	Forward Error Correction
FFT	Fast Fourier Transform
FIFO	First In First Out
FOSS	Free Open Source Software
FPGA	Field Programmable Gate Array
FRP	Frequency Resolution Protocol
GMSK	Gaussian Minimum Shift Keying
GPP	General Purpose Processor
GRC	GnuRadio Companion
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IC	Integrated Circuit
ICTAS	Institute for Critical Technology and Applied Science
IP	Internet Protocol
IPMI	Intelligent Platform Management Interface

ISM	Industrial, Scientific, Medical
KVM	Keyboard Video and Monitor
LDAP	Lightweight Directory Access Protocol
LLC	Logical Link Control
MAC	Media Access Control
NBP	Network Bootstrap Program
NFS	Network Filesystem
NIC	Network Interface Card
N/LOS	Non/Line-of-sight
OEF	OSSIE Eclipse Feature
OFDM	Orthogonal Frequency Division Multiplexing
OSI	Open Systems Interconnection
OSSIE	Open Source SCA Implementation-Embedded
PAM	Pluggable Authentication Module
PXE	Pre-Execution Environment
RAM	Random Access Memory
RF	Radio Frequency
RTS	Ready To Send
SCA	Software Communications Architecture
SDE	Software Development Environment
SDR	Software Defined Radio
SSH	Secure Shell
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
UHD	Universal Hardware Driver
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral

1

Introduction

Through the continued support of partners in government and industry, Wireless@VT has laid the groundwork for an innovative and unique testbed concept that aims to bridge the gap between theoretical knowledge and real world applications in the field of cognitive radio and cognitive networking. The testbed, called CORNET (Cognitive Radio Networking Testbed) is unique in a number of important ways that set it apart from similar projects around the country. Specifically, the distribution of radios inside of an active research facility allows for the opportunity to prototype and test a variety of cognitive networking concepts on "live" spectrum, in a high-fading indoor scenario. For the academic researcher, CORNET is designed to provide the user access to a flat architecture between the radio and the user planes, allowing researchers to configure parts of the testbed in any way they wish to accommodate a "bottom-up" experimental design flow. To this end, CORNET provides the user with the means to run experiments at any

level of the OSI networking model, with an emphasis on physical, data link, and network level technologies. A highly flexible Software Defined Radio (SDR) API is provided by GnuRadio, which includes a variety of physical and data link layer tools that may be used to abstract certain parts of a communications system – allowing users to focus on their specific research topic without concerning themselves too much with tangential programming tasks.

Beyond simply building a novel tool for wireless networking experiments, CORNET has a strong focus on accommodating the development of technology which will foster the emergence of future cognitive radio systems. USRP2s are used as a reconfigurable, physical layer entity, and can be programmed to perform a variety of tasks – from basic spectrum sensing to emulation of a GSM base station [19]. All 48 nodes of our testbed make use of the USRP2 (with the planned addition to newer USRP platforms, such as the USRPe100) which, combined with the distribution of the nodes across 4 floors of an academic building, gives researchers an opportunity to perform cognitive networking experiments on an medium to large scale. Each USRP can be configured with a custom FPGA image or RF daughterboard to allow an even greater level of flexibility in designing and running experiments. Additionally

Wireless@VT possess one of the largest continuous experimental spectrum licenses in the country for use with CORNET [8] – which applies to any experiments carried out within the facility – giving the research community the ability to perform CR prototyping within the bounds of FCC restrictions. This experimental license represents a critical and unique aspect of CORNET, since the license permits full use of our hardware, rather than being restricted to ISM band transmissions.

CORNET provides a number of unique capabilities designed to extend the ability of the academic community to push forward with cognitive radio research. The testbed extends and augments the testbed concepts implemented at other Universities. Rather than aiming to compete with or duplicate other projects, testbeds at other Universities were studied, with the aim of emulating their strengths, while adapting their concepts towards a focus on cognitive networking capabilities. In this way, CORNETl benefits the academic community by providing a stable and flexible platform for novel research.

2

Background

2.1 Software Defined Radio

As a wireless networking testbed, CORNET is intended to provide a platform through which researchers can develop and test CR algorithms and applications on a networked cluster of Software Defined Radio (SDR) nodes [20]. SDR is a critical enabling technology for CR applications, as it moves most of the non-trivial signal processing away from application-specific Integrated Circuits (ICs) and Digital Signal Processing (DSP) chips, onto flexible General Purpose Processor (GPP) platforms. By shifting DSP away from application-specific architectures, and onto today's powerful CPU platforms, a SDR gains the ability to alter transmission parameters simply by making adjustments to software. In this way, a SDR eliminates the need for physical buttons and knobs by replacing them with variables written into the signal processing code. If this concept is taken a step further towards defining algorithms which

govern when and how a radio should adjust these variables, we arrive at the concept of cognitive radio [18].

In VT-CORNET, the deployment of SDR nodes is tailored specifically around their utility for CR applications, and researchers are invited to approach their experimental needs from the physical layer up. This concept is unique for the level of customization that it provides to the users in terms of heterogeneous scenario deployment. Each node can be configured in a virtually unlimited number of ways by either modifying example code or building entirely new waveforms from scratch. Advanced users are offered unrestricted access to individual nodes in order to develop entirely new waveforms, while less experienced users are provided with access to a catalog of example code from the GnuRadio and OSSIE libraries. VT-CORNET is capable of serving both users' needs, and this philosophy differs from other wireless testbeds, which implement specific physical and transport layer protocols, and focus primarily on networking applications [17]. In contrast, CR applications may require real-time adaptive modifications to a radio's operating parameters at all layers of the OSI model, and any experimental platform must accommodate this need. The aim of VT-CORNET is to

provide such an open environment specifically for CR research and education.

2.2 Cognitive Radio

Cognitive Radio and its related applications are currently at the forefront of wireless communications research. Such technology promises to usher in devices which not only cooperatively share spectrum, but which intelligently coordinate, respond, and adapt to dynamic environmental and network conditions. The continued development of CR technology has obvious applications to military networks, but additionally play a large role in future commercial networks as a means of sharing finite spectrum resources among an increasingly growing pool of consumer-grade wireless devices and applications [3].

A CR is generally implemented using a Software Defined Radio (SDR) platform with additional AI/machine learning capability. SDR is a broad term used to describe a radio which implements some or all of its Digital Signal Processing (DSP) functionality on a General Purpose Processor (GPP). By implementing DSP functionality on a GPP rather than application specific chipsets, the radio may be made dynamic. A CR

takes this concept one step further by adding a level of artificial intelligence to the programming. This intelligence extends beyond simple "sense to act" behavior (e.g. CSMA/CD protocols), but rather encompasses a broad range of behavior and learning algorithms used to optimally select or modify the operating parameters of a radio for some specific task.

2.3 Testbeds

Testbeds have played an important role in the development of networking and wireless technology, and will continue to do so as new applications and paradigms develop within the field. CORNET takes inspiration from a number of similar projects, and is intended to apply the proven utility of the testbed concept to Cognitive Radio research. This approach to testbed design is unique in a number of key ways, which are consistent with CR research goals. Primarily, CORNET nodes provide a high degree of flexibility in terms of radio waveform implementation and adaptive capabilities. During the design process, several existing wireless testbeds were studied in order to determine their significance relative to our needs. Most example cases make use of radio hardware tailored to a

single wireless technology – be it WiFi, bluetooth, 3GPP, or some combination of similar equipment. Since CR applications often require a radio to possess adaptive link capabilities that these protocols do not implement, this approach was seen as less suitable for CR research. Furthermore, instead of locating the radio nodes away from potential sources of interference (in a shielded or remote room), there are benefits to deploying the network in a manner which exposes the nodes to live radio spectrum. CORNET nodes are deployed indoors, over 4 floors of a campus research facility – with 12 nodes per floor. The benefit of distributing nodes over an indoor area is the variety of channel conditions such an arrangement provides. Researchers have the option to use adjacent, line-of-sight nodes in combination with nodes located in different parts of the building, or on different floors in order to emulate a number of wireless networking scenarios. This combination of live spectrum and non-ideal channel conditions represents a philosophical departure from previous testbed designs, and offers a highly relevant prototyping environment for CR experiments.

Several other University testbeds focused on wireless research implement similar prototyping architectures, many of which inspired the design of CORNET. The Wireless Networking Research Testbed at UC

Riverside [25] is similarly deployed in a live building environment and includes USRPs in addition to WiFi platforms. The original EmuLab at the University of Utah [26] and the ORBIT testbed at Rutgers [17] have also recently upgraded a number of their testbed nodes to include USRP peripherals. What sets CORNET apart from these testbeds is the use of the more capable USRP2 hardware in conjunction with the powerful GPP cluster. The USRP2 can tune a much larger instantaneous bandwidth than the USRP1 due to the Ethernet-based host connection (~20MHz compared to 8MHz), and the GPP rack hardware provides more than enough computing power to handle the extra data. Many of the other testbeds mentioned predate CORNET by several years, and were originally designed for purposes other than CR research, and software defined radio nodes were only added recently. CORNET was designed from the very beginning with a focus on CR research, and uses only SDR platforms for the radio interface. Finally, CORNET is unique in that its operation is covered by an FCC research license [8], permitting the operation of low-power transmitters between 150MHz and 3600MHz within the ICTAS building where the testbed is located.

3

Cognitive Radio Network Testbed

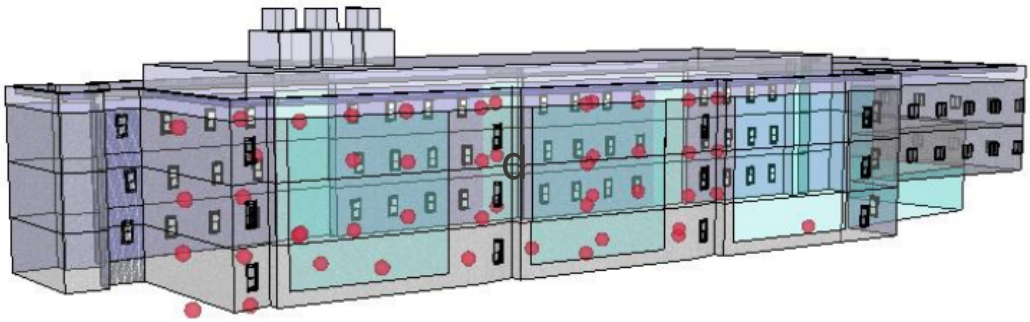

Figure 1. CORNET Node Distribution

3.1 Overview

The Cognitive Radio Network Testbed (CORNET) at Virginia Tech in a unique testbed environment, one which provides a platform for researchers and undergraduate students to perform wireless networking experiments and lessons on a network of reconfigurable radio devices.

The network is located in the Institute for Critical Technology and Applied Science (ICTAS) building 1 (fig. 1), which is located on the

Virginia Tech main campus in Blacksburg, Virginia. CORNET is made up of 48 SDR nodes, distributed in the ceiling of the building's four floors. Each node consists of a reconfigurable radio device, tied to a powerful GPP server cluster. The GPP cluster is sufficiently powerful to accommodate a variety of common SDR waveforms used in CR research, as well as maintaining sufficient overhead to accommodate the deployment of new or novel waveforms.

In addition to Virginia Tech researchers, CORNET is open to academic and government scientists and engineers, who may log in remotely to perform tests or experiments. As previously mentioned, and experimental FCC license covers nearly the entire operational range of the radio devices, allowing remote users to run waveforms with arbitrary bandwidth over a broad frequency range, without needing to seek a separate experimental spectrum license.

3.1.1 Motivation

Testbeds have played an important role in the development of networking technologies since the first computer networks were developed. With recent academic interest into CR systems and networks,

Virginia Tech researchers identified the need for a similar experimental network that could be used to develop and test wireless technologies and protocols which may be used in future CR networks.

Several other wireless testbeds which have traditionally used legacy standards like 802.11 for testing, have started to augment their single-standard radios with reconfigurable platforms, similar to those used in CORNET. This is a sign that the academic community is beginning to move towards the prototyping phase of CR product and application development. The third generation partnership (3GPP) specifications for LTE networks already implements basic cognitive radio principles [1], and the need to expand this trend is becoming apparent as spectrum resources become increasingly more scarce.

3.1.2 Intended Usage

The intended usage of CORNET informs the motivations for many of the design decisions which went into building it, though it does not limit the scope of possible experiments for which it is suitable. As mentioned in the previous sections, the goal of CORNET was to place many SDR nodes in a relevant spectrum and propagation environment in order to

better accommodate real world CR prototyping. This being the case, the nodes are set up in a minimalistic fashion, to provide a base configuration of common SDR signal processing tools that users would be free to customize as they choose.

The basic design flow that was conceived around CORNET would have researchers develop prototype waveforms independently (offline), and then log into and configure several CORNET nodes for further testing. CORNET nodes can be configured to act as part of an experimental network, or they can be used for observation/data collection, interference emulation, and a variety of other useful tasks. The reconfigurable nature of the nodes makes them very flexible in this regard. Most of these functions are most easily accomplished via the UNIX command line for experienced users, therefore an emphasis was placed on using Secure Shell (SSH) access to minimize network overhead.

In addition to research, CORNET is intended to be an education tool that students new to SDR and CR concepts could use to remotely complete lab activities and projects. SSH is less well suited for this purpose, so it was desirable to provide a remote desktop option for usage where graphical applications are needed. It should be noted that the

remote desktop capabilities were never intended to be used as a cloud based SDR development environment, though many users have adapted them to this purpose with good results. The remote desktop performance is acceptable for simple GUI interaction, but is slower than a normal desktop, and can feel sluggish compared to a local desktop.

3.1.2 Hardware

Each CORNET node consists of a dedicated GPP device, connected to a reconfigurable SDR device. The GPP devices are SuperMicro rack servers with 12GB of RAM and Quad-Core server CPUs. 56 servers make up the CORNET cluster (fig. 2), with 48 used as dedicated GPP devices, and the remaining 8 serving various network related functions. The servers are sufficiently powerful to run complex SDR waveforms, while performing other tasks, such as video encoding, simultaneously. Furthermore, the RAM and CPU specifications make the nodes highly useful for offline data post-processing as well – using OCTAVE, Python, or similar. The powerful GPP platforms, capable of high performance multi-tasking, are important to the CORNET concept, as CR applications often span multiple OSI layers. In addition to running modern physical

layer SDR waveforms, the SDR platform must have sufficient overhead to run various cognitive engine and machine learning algorithms in parallel. For this reason, the rack hardware is of utmost importance to the CR testbed concept.

Figure 2. CORNET GPP Cluster

48 USRP2 devices [10], manufactured by Ettus research, make up the other segment of the SDR nodes (fig. 3). Each USRP2 device has a dedicated 1000baseT Ethernet connection back to the GPP cluster, which limits the sampling rate to a maximum of 25 million samples per second (25 Msps), though the USRP2 hardware itself is capable of higher rates.

The USRP2 platform was selected for CORNET for a number of reasons: it can be configured to emulate most PHY layer waveforms; Ethernet back-haul allows for much longer cabling than the USB specification; and most importantly, there exists a large open source community which supports the USRP platforms. The open source community is especially beneficial, as it allows users to become familiar with the platform before they ever log into CORNET, easing the learning curve associated with using the testbed.

Figure 3. USRP2 in Ceiling

The USPR2 motherboard itself is primarily responsible for digital sampling, quantization, and sample rate conversion operations. Each device requires a separate daughterboard to act as an RF front end. The daughterboards are swappable, and are cover different frequency bands, with slightly different feature sets. CORNET employs the WBX

daughterboards, manufactured by Ettus research, as well as a custom Radio Frequency Integrated Circuit (RFIC) daughterboard at Virginia Tech. The specifications of the WBX daughterboards are avalable at [7]. The WBX board was selected for its broad range of tunable frequencies, which covers the range from 50MHz-2200MHz. The antennas attached to the USRP2s are tri-band 144/430/1200MHz omni-directional whips.

3.1.3 Software

Nearly all of the software used in the default CORNET node image is Free Open Source Software (FOSS). As previously mentioned, a dedication to FOSS platforms allows users to become familiar with the CORNET software before using the network, saving time for the administrators, and users alike.

The default Operating System (OS) on each CORNET node is a 64 bit version of Ubuntu 10.04 LTS. Ubuntu is a popular Linux distribution, which has been adapted to a large variety of applications and which has a large open source community in its support. Ubuntu was also selected due to its well maintained software repositories, which make it easier to add or remove software from the nodes.

The primary signal processing libraries offered on the default CORNET image are GnuRadio – another well supported FOSS software project, and OSSIE – Virginia Tech's in-house Software Communications Architecture (SCA) compliant FOSS project. Between these two frameworks, users have a wide variety of options in terms of waveform prototyping and deployment. Users are also welcome to install other SDR libraries, such as Liquid DSP [16], as needed for their research.

3.1.4 Remote Access

A core aspect of the philosophy behind CORNET is the testbed's ability to provide the flexibility of numerous co-located USRP2 radio devices to a broad variety of users. By nature, the USRP2 lends itself to a diversity of different research tasks and applications – which have varying requirements in terms of bandwidth, latency and interactivity. For an experienced user, most tasks in GnuRadio and OSSIE can be accomplished via the Linux command line, and require only secure shell access.

Secure Shell, or SSH is an application layer networking protocol, standard in most Linux distributions, which allows a user to securely

access a remote machine running an SSH server application. SSH is not limited to shell access only - users may also use an SSH connection to run small graphical applications over the same encrypted connection. SSH connections are secure, and require relatively little bandwidth compared to remote desktop applications, making them the preferred choice where they will suffice.

For less experienced users, or where a greater level of interactivity is required, CORNET nodes each run a FreeNX remote desktop server in addition to an SSH server, which users may also access. FreeNX was selected as the preferred remote desktop application from many options, because it functions directly on top of the existing SSH infrastructure, and requires little configuration. Client programs which are compatible with the FreeNX implementation on CORNET are available for Windows, Linux and Mac. Alternately, a single remote desktop connection, from which several SSH connections are made to additional nodes, may provide better performance than many individual remote desktop connections.

3.2 Architecture

The CORNET architecture can be thought of as consisting of three parts, or planes: the user plane – which is the portion of the network presented to a typical user, the radio plane – which consists of the SDR hardware and software, and the network plane – which consists of the necessary infrastructure required to operate the network.

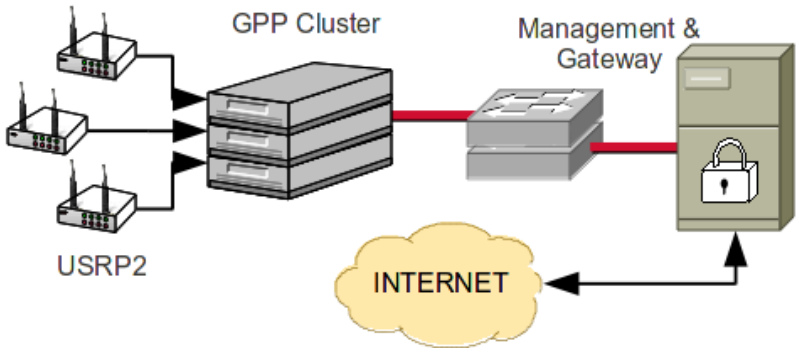

Figure 4. CORNET Architecture

3.2.1 User Plane

A typical user will log into individual nodes using a separate SSH or remote desktop connection for each node accessed at once. Once logged on, each user is presented with their home directory, which is mounted from a local Network Filesystem (NFS) server. Each user maintains and organizes their own NFS directory, and may use it to install custom software, store and access code used in experiments, or to store data for

later post-processing. Users also have limited sudo privileges on each node, permitting them a great deal of flexibility to customize the software as needed.

3.2.2 Radio Plane

The radio plane consists of the USRP2 hardware, and the connection back to their respective GPP servers. A major advantage of the USRP2 over similar USB connected SDR platforms (such as the USRP1) is the 1000base-T gigabit Ethernet connection between the GPP server and the USRP2, which allows both longer cabling runs, and higher throughput.

The Ethernet connections must be made directly, without the presence of a switch, in order to ensure a sufficiently latency-free connection between the GPP and the USRP2. The 1000base-T and USRP2 sampling specifications (16-bit ADCs, 14-bit DACs) permit the transmission of 25 million I&Q ("in-phase" and "quadrature") samples per second between the USRP2 and the host server. Since the 25Msps metric refers to a baseband signal represented by I&Q data, this translates to a maximum radio frequency bandwidth of 25 MHz after up-conversion.

3.2.3 Network Plane

The network plane provides the core networking functionality to CORNET, including user authentication, software management, and connection management. Additional network entities will be described in greater detail in section 3.3, and consist of servers dedicated to: PXE/re-Imaging, LDAP authentication, Network file storage, Apache, and the Firewall/Gateway.

3.3 Administration

Effective administration is an important part of keeping CORNET operational. The administration of such a wireless testbed is similar in may regards to the administration of any small scale local area network, though the novelty of CORNET necessitates customized solutions to many administrative tasks. The following sections will describe the specific implementation details of CORNET and how these details related to network administration in general.

3.3.1 Re-Imaging the Cluster

"The Cluster" refers specifically to the rack server nodes which make up the user plane and the signal processing side of the radio plane, and does not refer to any of the rack servers or switches which make up the network plane. "Re-Imaging the cluster" refers to the process of unattended operating system installation on some or all of the nodes at once. It is important to draw the distinction between traditional imaging operations, in which a formatted filesystem image is copied to disk byte by byte, and the concept of imaging being discussed here, where a normal operating system installation routine is automated via the use of PXE and boot scripts. The distinction is worth discussing because such a scripted installation permits more flexibility than a traditional imaging operation since there is no need to maintain a master disk image external of the network. Instead, a list of preferences, software and customizations is defined entirely within a pre-seed file, which can be used in a portable manner to install a wide variety of Debian-based distributions (of which Ubuntu is simply one option).

PXE stands for "Pre-Execution Environment" and is a widely used method for using a local area network as an installation medium, as

opposed to a DVD or USB drive. CORNET uses the PXELINUX implementation of the SYSLINUX bootloader to initialize an unattended installation of Ubuntu, as well as a variety of additional software. The basic PXE boot sequence is as follows:

1. The client(s) to install are powered on, and enter the PXE boot sequence by loading the PXE firmware from the first Network Interface Card (NIC eth0). The client begins seeking out a local DHCP server in order to obtain a network (IP) address.
2. The DHCP server contains static IP assignments for the subnet, as well as a proxy entry which instructs the client to the location of a lightweight thin-client bootloader called the Network Bootstrap Program (NBP).
3. The NBP is downloaded to the client via Trivial File Transfer Protocol (TFTP) and mounted in RAM. On CORNET, the TFTP server is located on the same machine as the DHCP sever (192.168.1.2) at /tftpboot/pxelinux.0
4. A configuration file containing a menu of pre-defined boot instructions is downloaded from the same TFTP server after the NBP has been verified. In addition to the SYSLINUX boot

parameters, the configuration file contains the locations of the Linux kernel to install, as well as a distribution-specific initial RAM disk image.

5. The NBP executes the SYSLINUX boot commands and initiates the installation process.

On CORNET, the files for installation are served via NFS from the same dedicated image server which also provides the required DHCP and TFTP services to the network. In essence, the contents of a Linux "Live CD/DVD" are extracted and placed in a local NFS directory, where it can be mounted remotely by the client, and used as a repository for the core operating system installation – similar to how a traditional CD/DVD based installation would progress. The following section details the filesystem overview of the image server.

3.3.2 Image Server Overview

In order to provide a more complete view of how PXE booting is accomplished, an overview of relevant parts of the image server's

operation are presented in detail. The important directory structures used in the imaging process are outlined below:

/tftpboot
→ pxelinux.0
→ menu.c32
/tftpboot/pxelinux.cfg
→ default
→ conf_1
→ symlinks
→ power
/tftpboot/initrd
→ initrd.gz
/tftpboot/kernels
→ linux
→ vmlinuz
/tftpboot/template
/tftpboot/template/libs
→ boot_cfg
→ first_boot
→ ldap.conf
→ nsswitch.conf

The image server is located internally at 192.168.1.2, and can only be accessed from within the network. The outline above is a description of the TFTP and NFS filesystems utilized during the boot process. As previously stated, pxelinux.0 is the first piece of the process to be downloaded once the DHCP server has assigned an address to the client. The next step in the process is for the client to download the correct configuration file, located at /tftpboot/pxelinux.cfg/. An example entry from this file is shown below. For more specifics about the meaning of

the configuration file, and boot parameters, see the SYSLINUX documentation [24].

```
Default menu.c32

TIMEOUT 100
PROMPT 0

label localboot
LOCALBOOT 0

Label CORNET_64_install
        menu default
        kernel /CORNET_64/linux
        append auto url=http://192.168.1.2/custom.seed boot=casper \
                locale=en_US console-keymaps-at/keymap=us          \
                netcfg/choose_interface=eth0 netboot=nfs           \
                nfsroot=192.168.1.2:/tftpboot/CORNET_64/           \
                initrd=CORNET_64/initrd.gz priority=critical
```

The client searches for the file sequentially within the TFTP directory, searching first for a custom file name based on it's own network address. If no custom configuration file is present, the client downloads the configuration file named "default." The files named "symlinks" and "power" within the /tftpboot/pxelinux.cfg are custom administrative programs which are used to set the configuration files correctly prior to installing, rebooting, or shutting down the system without causing conflicts, such as the PXE boot loop (described in section 3.3.5).

The directories /tftpboot/kernels and /tftpboot/initrd store the kernel and initial ram disk images referenced in the configuration file. There are additionally several template directories, /tftpboot/[templates], which have different names based on the image with which they are associated. For example, in the configuration file example above, the "CORNET_64" template is being used. These template directories replicate the contents of an installation CD/DVD that will be served via NFS (the root NFS directory is also /tftpboot on the image server) during the actual installation. Within each template directory, there is also a /libs folder, which contains additional configuration scripts that cannot easily be integrated into the imaging process. The contents of these scripts can be found in Appendix A.

3.3.3 Intelligent Platform Management Interface

The SuperMicro rack servers used in the CORNET cluster are equipped with an Intelligent Platform Management Interface (IPMI) port, which allows OS independent remote management of the individual servers. The IPMI standard, developed and maintained by Intel [14] is commonly used for a variety of remote network administration tasks

including monitoring and remote console operations. Primarily, IPMI is used on CORNET nodes for remote power control purposes; in order to initialize the imaging process; or reboot the cluster. Each node requires a separate IP address for it's individual IPMI port, starting at 192.168.1.201 for node 1, 192.168.1.202 for node 2, and so on.

There are a number of software options available which are compliant with the IPMI standard. Among the numerous options, FreeIPMI [11] is a mature open source implementation which provides a convenient command-line interface that can be easily integrated into custom tools. The command most commonly simply changes the power status of the server chassis – allowing the nodes to be turned on and off over the network:

```
ipmi-chassis -h 192.168.1.xxx -u [user] -p [password]                    \
                --chassis-control=POWER-UP
```

This command is used within the "power" script mentioned in section 3.3.2, which is used to image, shut down, or reboot several nodes at once. The "--chassis-control=" flag accepts several options apart from POWER-UP, including: POWER-DOWN, POWER-CYCLE, HARD-RESET and

SOFT-SHUTDOWN. More information about the command can be found in the documentation on the FreeIPMI website.

In addition to the command line tools, the SuperMicro IPMI implementation includes a built in web server, allowing most of the IPMI functionality to be accessed from a local browser. The web server can be accessed from the local network by entering the corresponding IP address of the IPMI port for the desired node (Fig. 5). The web interface additionally provides a java-based, virtual Keyboard/Video/Mouse (KVM) switch, accessible under the "Remote Access" tab. The virtual KVM switch supports the ability to forward the screen output from the remote server over the local network. Screen forwarding over IPMI is handled at a low level, and does not require the operating system to load in order to function (like a virtual desktop) – allowing an administrator to perform BIOS and boot customization tasks over the network.

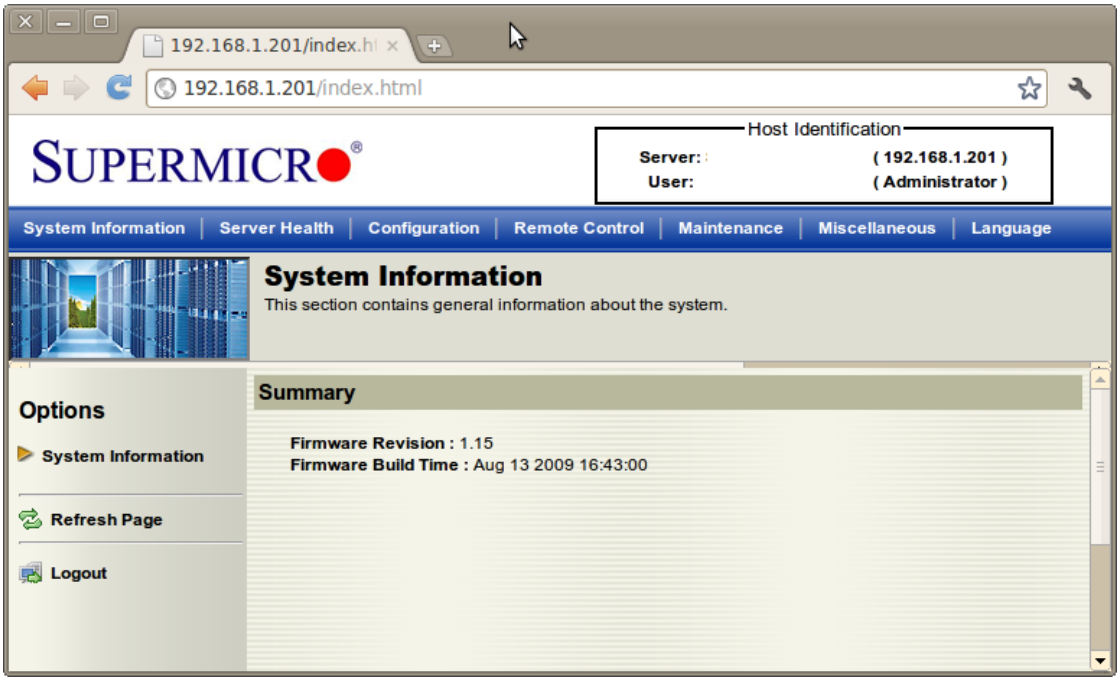

Figure 5. IPMI Web Server User Interface

3.3.4 Debian Preseeding

As a Debian-based Linux operating system, Ubuntu is installed through the use of the Debian installer system [6]. This system includes functionality which allows administrators to perform a scripted, unattended installation of Debian operating systems through the use of a preseed file. Though preseeding may be used for installing Ubuntu from any medium, it is most commonly employed for the type of PXE network installation used to re-image the CORNET cluster. The preseed file itself is primarily a formatted list of answers that the Debian installer would typically prompt the user to answer during an attended installation.

Typical scripted input routines will handle disk partitioning, keyboard/language setup, and network configuration. In addition, the Debian installer recognizes a series of APIs which provide an administrator with a high degree of control over small details of the resulting system configuration.

Re-imaging the CORNET cluster relies heavily on the Debian installer and associated preseeding routines to provide flexibility to the imaging process. One significant advantage of using preseeding, compared to a more direct method of imaging (using *dd* to clone a partition), is that it does not require a "gold" image to be maintained separate from the cluster. Instead, the entire system configuration is entirely defined by the Ubuntu software version and contents of the preseed file. The preseed file is downloaded to the client being imaged following the completion of the PXE boot routine – after the bootloader has passed control of the system over to the installer. The file is downloaded via HTTP from the image server, where it is stored at /var/www.

A repository mirror is used on the cluster to increase the speed at which nodes can download the packages required for installation. The preseed file contains entries for configuring the system to use the local

mirror under the *#Mirror/Apt* heading. The command *pkgsel/include* follows the repository mirror setup under the heading *#Packages* and includes a list of the repository software to download, using the *pkgsel/include* command. The packages listed are mostly needed as dependencies for building GnuRadio and OSSIE, and additional packages may be simply appended to the list. Next, the *LDAP* heading is followed by the routines required to configure the client node for LDAP authentication. After the LDAP routine, pressed scripting of the disk partitions, administrative user creation and network configuration, and keyboard take place – these routines are fairly straight-forward, and further explanations can be found in the Debian installer documentation.

The final command in the preseed file is the *preseed/late_command* routine, which allows for the execution of custom commands from the Debian installer thin-client. This is critically important for breaking the PXE boot loop, as will be discussed in the following section, but it is also used for other operations.

```
d-i     preseed/late_command string mkdir /target/media/mnt; chroot
        /target sudo mount -o nolock,rw 192.168.1.2:/tftpboot /media/mnt;
        rm /target/media/mnt/pxelinux.cfg/C0A8; chroot /target cp
        /media/mnt/CORNET_64/libs/first_boot_uhd  /etc/init.d;  chroot
        /target cp /media/mnt/CORNET_64/libs/boot_cfg /etc/init.d; chroot
        /target chmod +x /etc/init.d/first_boot_uhd; chroot /target chmod
        +x /etc/init.d/boot_cfg; chroot /target update-rc.d first_boot_uhd
        defaults; chroot /target mkdir /users; chroot /target mkdir /tools;
        chroot  /target  cp  /media/mnt/CORNET_64/libs/ldap.conf  /etc;
        chroot /target cp /media/mnt/CORNET_64/libs/ldap.conf /etc/ldap;
        chroot /target cp /media/mnt/CORNET_64/libs/nsswitch.conf /etc;
        chroot /target cp -r /media/mnt/gnuradio-3.3.0
        /home/cornetadmin/gnuradio-3.3.0; chroot /target cp -r
        /media/mnt/gnuradio_uhd    /home/cornetadmin/gnuradio_uhd;
        chroot /target cp -r /media/mnt/uhd /home/cornetadmin/uhd;
        chroot /target cp /media/mnt/ossie-0.8.2.tar.gz
        /home/cornetadmin/ossie-0.8.2.tar.gz; chroot /target cp -r
        /media/mnt/CORNET_64/libs/ossie-misc
        /home/cornetadmin/ossie-misc; chroot /target umount /media/mnt;
        chroot /target updatedb; chroot /target nss_updatedb ldap
```

The code snippet shown above is the complete late-command used for the cluster installation process. The script performs several tasks, in the following order:

- Modifies the image server /tftpboot/pxelinux.cfg directory to break the PXE boot loop.
- Copies the boot scripts *first_boot* and *boot_cfg* and sets them as boot-time executables.

- Copies *LDAP* configuration files to the client node which are used in conjunction with the preseed commands to set LDAP authentication.
- Copies GnuRadio, UHD and OSSIE source code to the client node, which will be compiled and installed by the *first_boot* script.

3.3.5 Breaking the PXE Boot Loop

A common difficulty in using PXE to initiate an unattended network installation process is known as the "PXE boot loop," and it is not a problem unique to CORNET. The post-BIOS behavior of the PXE NBP is dependent on the configuration file downloaded from the image server once the client has received an IP address. In order to make the installation truly unattended, the entry in the configuration file corresponding to the correct bootloader commands must be marked as the default entry. When installation is finished, the server is rebooted in order to complete the process as much of the configuration is applied only after the first reboot. However, the BIOS is still configured to boot from the network by default, and the PXE configuration file is likewise configured to initiate a new installation routine when the server is power cycled.

Therefore, without any additional intervention, the installation process will occur repeatedly in a loop, as the servers are rebooted at the end of the installation routine. This repeated installation sequence is commonly referred to as the PXE boot loop.

In order to interrupt the boot loop, the PXE configuration file on the image server must contain two bootloader directives – one to initialize the installation, and one to boot the server from the local drive after installation. Three custom scripts are employed for this purpose – the first two scripts are used initialize the installation routine from the image server, and the third script initializes the local boot routine from the client server automatically following a successful installation. The initialization scripts, called *power* and *symlinks,* are located on the image server in the /tftpboot/pxelinux.cfg directory. In order to start the installation, the following command is issued from an image server shell:

```
./power [command] [node]
```

The *[command]* parameter can be *shut* to power down a node, *install* or *installuhd* to image a node, or *boot* to power on a node without re-imaging. The *[node]* parameter is the list of the nodes to which the

[command] will be applied, and corresponds to the last octets of the nodes' IPMI address. For example, to re-image nodes 1-1 and 1-2, the administrator would log into the image server, navigate the shell to /tftpboot/pxelinux.cfg and issue the command:

```
./power install 201 202
```

When the *install* command is issued, the *power* script calls the *symlinks*, or *symlinks_uhd* script, which creates a symbolic link to the correct PXE configuration file inside the same directory. The PXE configuration file associated with the installation process is called *cog1_conf*, but it will not be recognized as a valid PXE configuration file since the filename is not formatted to be recognized by the NBP. When *symlinks* is executed, a symbolic link with a properly formatted filename is created, and is linked to the *cog1_conf* configuration file. The symbolic link filename is *C0A8* – which corresponds to the first two octets of the subnet in hexadecimal notation (*C0* = 192, *A8* = 168), which is a format that the NBP recognizes. If either the *boot* or *shut* commands are issued instead of *install*, the *power* script ensures the symbolic link is not present by attempting to delete it. In this scenario, the NBP will select the configuration file titled

default which contains bootloader commands to boot locally from the internal drive.

Since the installation is unattended and non-interactive, an additional process is required to automatically delete the *C0A8* file following a completed installation, after which the nodes will boot into the newly installed operating system according to the *default* PXE configuration file. To accomplish this task, a series of commands is scripted into the Debian preseed file under the *late-command* directive. When the installation process is initialized, the Debian installer thin-client creates a /target directory on the node, at the root of its temporary filesystem, and performs the installation within this directory. Upon completion, the installer changes the root filesystem location to /target and hands control of the system over to the default user (*cornetadmin* in this case). The *late-command* is applied upon successful completion of the installation routine, but before the installer performs the *chroot* into /target. This way, commands can be issued while the installation media is still mounted. The relevant parts of the *late-command* are shown below:

```
d-i     preseed/late_command string mkdir /target/media/mnt;        \
            chroot /target sudo mount -o nolock,rw                  \
            192.168.1.2:/tftpboot/media/mnt; rm                     \
            /target/media/mnt/pxelinux.cfg/C0A8;
```

First, the PXE configuration directory on the image server is mounted to /target on the node(s) being installed. This requires manually changing root (*chroot*) to /target in order to use the *mount* command, which is not present in the Debian installer thin-client. Once the remote directory is mounted, the *C0A8* configuration file is removed, and the system can be power cycled. On the next boot, the NBP will use the *default* configuration file to boot locally – preventing an PXE boot loop. It is also important to note that the *C0A8* file will exist on the image server until installation is completed, so any nodes which are power cycled while this file still exists will also begin the installation routine. As a result, the *power* script should always be used to power cycle the nodes to ensure that the PXE directory is properly configured.

3.3.6 Local Repository Mirror

The installation procedure is a networked and automated version of a typical CD based installer, so only the core Ubuntu packages are available

from the installation medium. Other packages must be downloaded from an external code repository. However, when imaging a large number of nodes at once, it is inefficient to have each node download these packages from the external Ubuntu repositories individually. Instead, a mirror of the Ubuntu repositories is maintained on the local network, which is connected with 1000Base-T gigabit Ethernet links. The repository mirror is located on the internal network at 192.168.1.102.

3.3.7 Non-Repository Software

Most of the software installed by default on CORNET nodes can be found in the Ubuntu repositories maintained by Canonical. However, certain pieces of software, specifically OSSIE and the most recent versions of GnuRadio/UHD, are not available from the standard repositories. As such, these software packages must be compiled from source code – a process which requires additional customization of the unattended installation routine via the Debian preseed file, as well as the execution of custom scripts which are run once during the first reboot of the nodes.

The first step in the process is to configure and build the source code on a freshly imaged template node, keeping track of any additional software dependencies which must be met. Additional dependencies that are available via standard repositories are then added to the list of packages to install inside the preseed file. When the template build is complete, the compiled code is copied to the /tftpboot directory on the image server. The directories called *uhd*, *gnuradio_uhd,* and *OSSIE_0.8.2* are all examples of pre-compiled code within this directory. When the Debian installer reaches the *late-command* routine, these directories are copied over to the nodes using the following (example) command:

```
d-i     preseed/late_command string [...] chroot /target cp -r          \
        /media/mnt/gnuradio_uhd  /home/cornetadmin/gnuradio_uhd;
```

This process is repeated within the *late-command* routine for each software package installed in this way. Additionally, two custom boot-time scripts, called *first_boot* and *boot_cfg,* which reside inside the /tftpboot/[template]/libs directory on the image server, are copied to the /etc/init.d directory on the target node. These files are set as executable boot-time scripts, to be run during the OS initialization routine:

```
d-i     preseed/late_command string […]
        chroot /target cp /media/mnt/CORNET_64/libs/first_boot      \
        /etc/init.d; chroot /target chmod +x /etc/init.d/boot_cfg;  \
        chroot /target update-rc.d first_boot_uhd defaults;         \
```

The *first_boot* script contains instructions for installing and configuring the compiled source code for the GnuRadio, UHD and OSSIE software packages using the following scripted commands:

```
cd /home/cornetadmin/gnuradio-3.3.0/
./configure
make
sudo make install
sudo ldconfig
sudo updatedb
```

The *first_boot* script performs a number of other operations, which will be discussed in subsequent sections, but it only needs to be executed once during the initial OS configuration. On subsequent reboots, the *first_boot* script is replaced by the *boot_cfg* script, which does not contain any of the one-time configuration instructions.

This basic procedure may also be useful to users who with to install non-repository software on several nodes at once without the need to build the source on each node. Source code may be compiled once inside

a user's NFS directory, and installed as needed by issuing *configure* and *make install* commands while logged into individual nodes.

3.3.8 LDAP and PAM User Authentication

User authentication on CORNET nodes is handled by the Lightweight Directory Access Protocol (LDAP), in conjunction with Pluggable Authentication Modules (PAM). LDAP and PAM are frequently used in conjunction for user management on Unix networks because they provide a great deal of flexibility in terms of managing access to individual network services in a manner which is agnostic to the specific network architecture and usage requirements. In essence, the LDAP/PAM combination allows a single permissions policy to be implemented across an entire network, in much the same way that a security policy would be implemented on an individual Unix machine. User profiles are stored centrally on and LDAP server, which may contain information about individual users and to which policy groups they belong. In general, an individual network device contains a series of PAMs which govern group or user access to various services and applications unique to each node.

On CORNET nodes, LDAP is only used for SSH authentication, so only a single PAM file is required. Once an SSH session has been authenticated, the default group permissions are enforced as if the user was logged in locally. LDAP configuration is part of the default installation procedure, and is scripted in the Debian preseed file:

```
ldap-auth-config    ldap-auth-config/dbrootlogin boolean true
ldap-auth-config    ldap-auth-config/override boolean false
ldap-auth-config    ldap-auth-config/ldapns/ldap-server string      \
                     ldap://192.168.1.101
ldap-auth-config    ldap-auth-config/binddn string                  \
                     dc=wireless,dc=vt,dc=edu
ldap-auth-config    ldap-auth-config/bindpw string doesnotexist
ldap-auth-config    ldap-auth-config/rootbinddn string              \
                     dc=wireless,dc=vt,dc=edu
ldap-auth-config    ldap-auth-config/rootbindpw string doesnotexist
ldap-auth-config    ldap-auth-config/ldapns/ldap_version select  3
ldap-auth-config    ldap-auth-config/ldapns/base-dn string          \
                     dc=wireless,dc=vt,dc=edu
```

CORNET nodes uses the default OpenLDAP packages from the Ubuntu repositories. More information about configuring LDAP and PAM in general can be found on the Ubuntu help page [5].

3.3.9 IP Tables and Secure Shell Tunneling

The CORNET gateway and firewall are responsible for tunneling external SSH connections to the correct nodes based on the port used by

the incoming connection. The tunneling is accomplished through the use of IP tables [4] on the gateway, which has an internal address of 192.168.1.100 and an external address of 128.173.221.40, and which resolves to cornet.wireless.vt.edu via DNS. The process to establish an IP table forwarding rule for an SSH connection is achieved by issuing two commands on the gateway server:

```
sudo iptables -A PREROUTING -t nat -i eth1 -p tcp --dport [port] -j DNAT
        --to [Internal IP]:22

sudo iptables -A FORWARD -p tcp -d [Internal IP] --dport 22 -j ACCEPT
```

The first command creates a network address translation rule for the gateway, indicating that *eth1* is the internal NIC. The rule only applies to TCP connections, and forwards all incoming traffic on port *[port]* to the destination *[Internal IP]* on port 22. The second command sets up a firewall rule allowing the internal NIC (*eth0*) to forward the connection. In this way, the first command creates the route, and the second command enables it. Additional port forwarding rules may be configured in this way for HTTP traffic, or for other applications which must operate across the gateway.

3.3.10 Web Server

The administrative web page, along with the official CORNET web page, are hosted internally on a local web server located at 192.168.1.103. The web pages includes several tools which are useful for performing basic administrative tasks as well as providing a reference for users. The official web page and wiki can be accessed via a browser at *cornet.wireless.vt.edu* (fig. 6) and the administrative web page can be accessed from *cornet.wireless.vt.edu/admin/* (fig. 7).

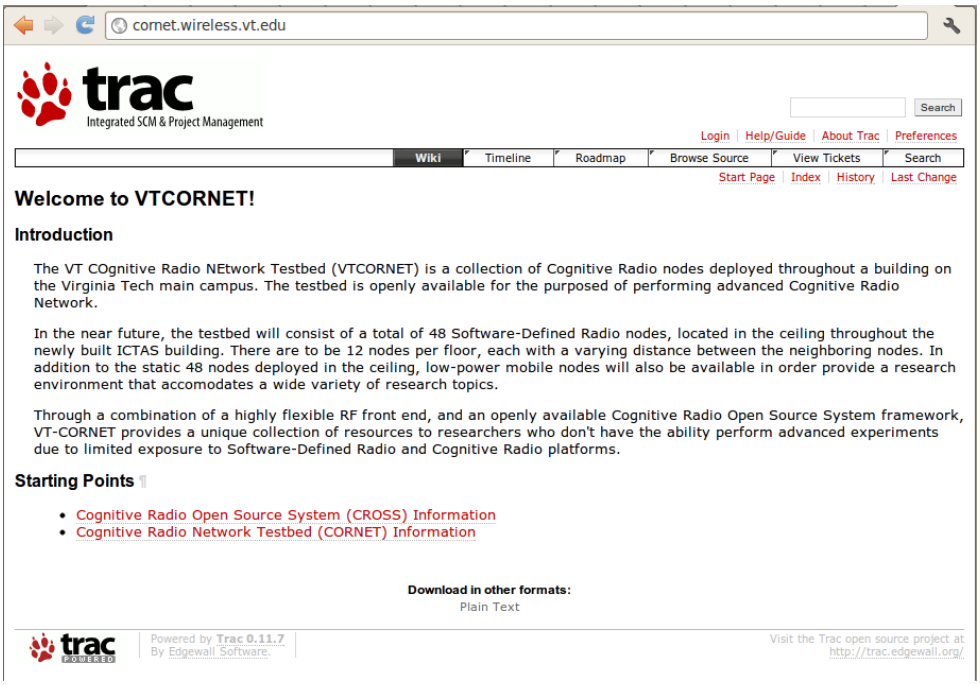

Figure 6. CORNET Web Page

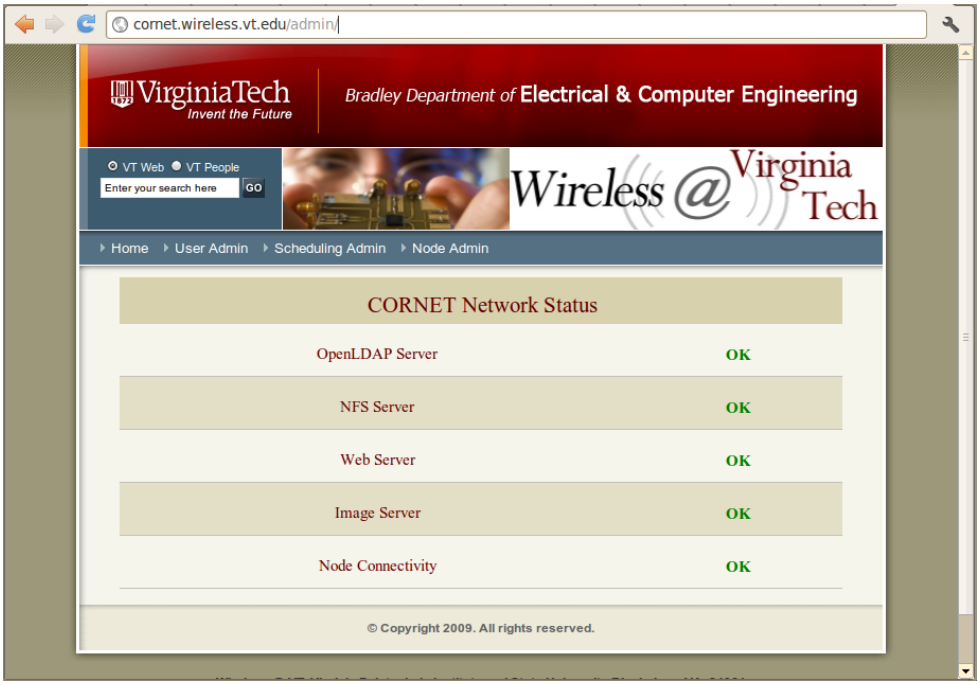

Figure 7. CORNET Admin Web Page

The administrative web page contains four links near the top of the page to *Home, User Admin, Scheduling Admin,* and *Node Admin.* New users can be created from the *User Admin* page, and nodes can be imaged, shut down, or re-booted from the *Node Admin* page by selecting the appropriate action from the drop down menus next to each node (fig. 8) and then pressing *Submit* at the bottom of the page. It is important to remember that a node must be shut down prior to being re-installed, or the installation command will have no effect. New users are created using the *User Admin* page, as shown in figure 9.

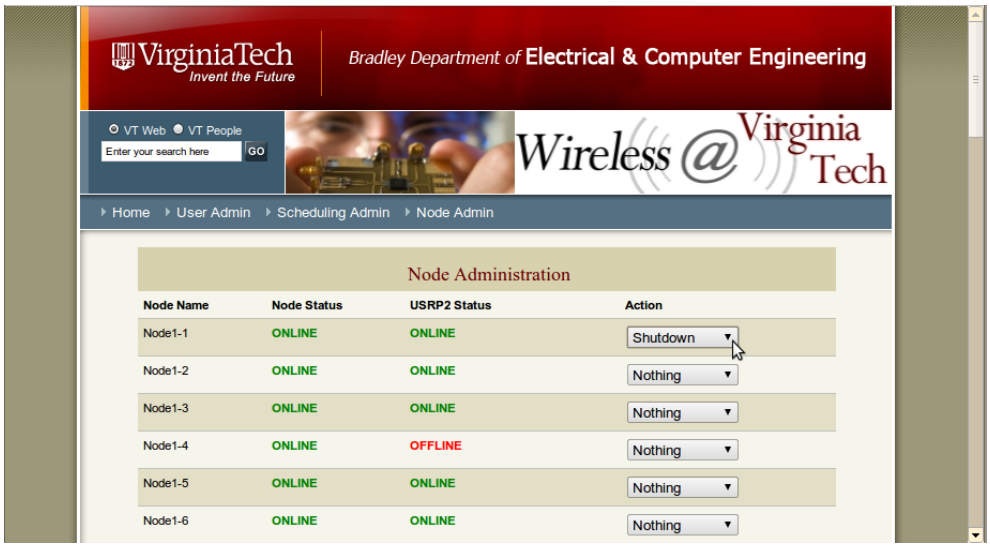

Figure 8. Node Administration Web Page

Figure 9. User Create Web Page

The user management web-application creates the LDAP user, but does not create an NFS directory for them. In order to create the NFS directory, an administrator must log into to one of the nodes and manually create the account and set the ownership:

```
cd /users
mkdir /[user name]
chown -R [user name].[user name] [user name]
```

The PHP and HTML code which define the web pages are stored at /var/www/, and are hosted by an Apache server application.

One important script which is used by both the administrative and user web pages is the *check_usrp.php* script. This PHP script polls the nodes to determine whether they are currently online and active by attempting to execute remote SSH commands, and parsing their output. The script is stored at /usr/local/bin/ and is run once per hour as a *cron* job. The output of the *check_usrp* script is a text file called *node_status.txt* which is referenced by the web sites when the status of the nodes is displayed. The process of querying the nodes is as follows – first, an attempt is made to establish an SSH connection with each node. If a connection is established, the node is powered on. Next, the script attempts to execute the *find_usrps* GnuRadio tool to determine if a USRP2 with legacy drivers is connected and available:

```
$stream = ssh2_exec($con, "sudo find_usrps -e eth1");
stream_set_blocking($stream, true);
$output = stream_get_contents($stream);
$pos = strpos($output, "00:50");
if($pos === false) {
        $usrp_status[$usrp] = 0;
} else {
        $usrp_status[$usrp] = 1;
}
```

If the *find_usrps* check fails, the script checks for a UHD enabled USRP in a similar manner:

```
if($usrp_status[$usrp] == 0) {
        $stream = ssh2_exec($con, "uhd_find_devices");
        stream_set_blocking($stream, true);
        $output2 = stream_get_contents($stream);
        $uhdstatus = preg_match("/serial..[0-9][0-9][0-9]...");
        if($uhdstatus == 1) {
                $usrp_status[$usrp] = 2;
        }
}
```

4

CORNET DSA Example

4.1 Overview

This section details the operation of a GnuRadio Dynamic Spectrum Access waveform which runs on CORNET. The example touches on numerous topics which are relevant to CORNET, and specific to implementing Cognitive Radio applications in a testbed environment. These topics include the implementation of a two-way, half-duplex physical layer waveform in GnuRadio, as well as the implementation of signaling and transmission protocols for a master/client data link. In addition, spectrum sensing and peer coordination is discussed in the context of the example waveform. Overall, the example demonstrates a two way data link, and which highlights the utility of CORNET nodes for Cognitive Radio application prototyping.

4.2 GnuRadio Waveform

The SDR waveform which defines the DSA application spans multiple levels of the OSI model. The operation is defined by a GnuRadio physical layer waveform and packet parsing classes, in addition to custom Python routines which define the network layer and DSA behavior. The waveform itself is based off of the *benchmark_tx.py* and *benchmark_rx.py* example waveforms, which use classes from *transmit_path.py,* and *receive_path.py* to abstract a large part of the waveform's implementation. Packet and sequencing, framing, modulation and pulse shaping are handled by the *mod_pkt* and *demod_pkt* classes defined in the *pkt.py* class.

4.2.1 Physical Layer

The physical layer portion of the DSA application is a basic single-carrier, GMSK-based, packet radio data-link which has a variable center frequency in the range of 422.56 MHz to 457.56 MHz divided into 1.0 MHz wide channels. On the forward (video) link, the Gaussian pulse shaping filter uses a bandwidth-time product of 0.35 and a bit rate of 1.5 Mb/s, which produces a -3 dB filter bandwidth of 0.525 MHz. On the

return (ACK) link, the data rate is 100 Kb/s, which produces a -3 dB filter bandwidth of 35.0 KHz.

4.2.2 Data Link Layer

The DSA waveform is an example of a simple point-to-point wireless link, so framing, addressing and media access are all handled at the data-link layer of the OSI model. The protocols and behaviors which define the DSA functionality of the waveform can also be through of as data-link entities, since peer coordination, signaling and channel selection all fall under this description as well.

The media access protocol is a modified version of 802.11 CSMA/CA [2] with a simplified RTS/CTS signaling structure. Since the data link is persistent and highly asymmetric in a single direction, an acknowledgment based routine is used following a synchronization handshake. The master node synchronization and transmission routines for the ACK behavior are shown below. Synchronization is accomplished by sending SYN beacons, and dwelling on random channels until a peer is acknowledged. Once synchronized, the master node begins transmitting the video feed to the client 10 frames at a time. After each 10 burst, the

master node pauses to senses the channel for non-peer interference, and then waits to receive an ACK from the client. If an ACK is received, the master's timeout counter is reset and another 10 frames are transmitted. If no ACK is received, and the timeout counter expires, the synchronization routine is re-initiated. The timeout counter is set to be approximately the amount of time it takes to send 25 frames, so it requires multiple dropped/missed acknowledgments to initiate a channel evacuation.

Synchronize:
Select channel
Send Master Beacon
    Client beacon found?
        No → Select new channel
Send SYN packet
    Got SYN-ACK?
        No → Select new channel
Send ACK

Send Data:
Send 10 frames, pause
    Got ACK?
        No → Check for timeout
    Got timeout?
        Yes → Re-synchronize

Acknowledgments are only sent in one direction, from the client node to the master node. Since a synchronized client expects continuous data to be sent from the master, its timeout routine simply measures the time between receiving CRC-passed frames from the master. If interference

occurs on the link which prevents the client node from receiving packets, or which causes the master node to miss several expected ACK packets, a channel evacuation will occur.

4.2.3 Application Layer

The application layer is defined by the high level functionality of the waveform presented to the end user. For this example, the application is intended to be a one-way, real-time video link which can avoid primary users and malicious interference. In addition to the video display, the user is presented with a simple GUI which shows the synchronization status and current channel of the data link. Finally, Gstreamer – an open source multimedia signal processing framework [12] – handles encoding, multiplexing and video streaming at the client's application layer.

4.3 Dynamic Spectrum Access Protocol

The implementation of the DSA protocol is designed to operate over an asymmetric point-to-point data link, where video is streamed in the forward direction (from master to client) and acknowledgments are transmitted in the reverse direction (from client to master). The data link

is frequency agile, in that participant nodes will coordinate to select a valid (unoccupied) communications channel, and will seek out a new channel if interference is detected on the current one. A number of steps are involved in the process of peer discovery, coordination, data transmission and channel evaluation, which define the operation of the DSA protocol. In general, the DSA functionality is heuristic, and does not rely on explicit signaling to initiate a channel evacuation operation, or to coordinate the next channel to select. Rather, participant nodes decide whether a given channel is valid through independent spectrum measurements, as well as observation of peer behavior.

4.3.1 Spectrum Sensing

The ability for the nodes to sense the spectrum, and to determine the fitness of a given channel in real-time based on these measurements is of utmost importance to the successful operation of the data link. There are a number of different algorithms and techniques defined in the literature for performing efficient spectrum sensing in various scenarios. The most basic of these techniques is energy detection [28], whereby interference

and primary user transmissions are detected by measuring the amount of energy at the output of the receiver's channel filters.

Sensing occurs at two different stages of the waveform operation – during the synchronization phase as well as during steady-state transmission. In both cases, the sensing operation attempts to detect the presence of unknown transmissions, which may originate from a primary user or malicious interferer. During the synchronization phase, energy detection is employed in order to find unoccupied channels, and during steady state operation, the sensing attempts to determine if the channel is still clear, or if a new channel must be selected. In order to use an energy detector during normal transmission, both the master and client nodes must momentarily stop transmitting at the same instant while the sensing routine runs. This dwell period ensures that neither node will mistake a peer transmission for interference and trigger a false alarm channel evacuation event.

As previously stated, the operation of the DSA sensing routines, and the coordination of sensing activity between nodes is done in such a way as to limit the amount of meta-information which must be exchanged between nodes. The sensing is cooperative in the sense that both the master or client node can infer the results of the other's sensing operation

based entirely on the behavior of peer nodes. This principle will be described in greater detail in the following sections.

The sensing routines themselves are implemented in GnuRadio using a probe component attached to the incoming channel filter. The probe routine computes a windowed summation of the filter output, and returns the current energy value when called. The value returned is a derived energy measurement, which is a function of the data type, USRP2 gain settings, and antenna parameters. The energy measurement is compared to a threshold value to determine if other transmissions are present in the frequency band of interest. This threshold can be altered depending on the waveform requirements and deployment scenario. For this demonstration waveform, a threshold value of 1.0x10^5 was informally determined to provide a low probability of false alarm and missed detection events. Since CORNET nodes operate on live spectrum, it is difficult to empirically determine the false alarm and missed detection rates for the waveform, due to the presence of interfering transmitters, of which the user has no knowledge or control.

4.3.2 Peer Discovery and Coordination

Peer discovery is achieved through a beacon transmission and detection routine, during which the master node asserts itself on random vacant channels until it is located by a client node. During the discovery phase, both nodes broadcast unique beacons on random channels, but only the master beacon can initiate the synchronization routine – If the master detects a client beacon, it responds with a beacon of its own. Once the client has spotted the master beacon, it responds with a synchronization (SYN) frame to initiate a three-way handshake, after sensing the channel to ensure it agrees with the assessment that the channel is vacant. When the three-way handshake is completed successfully, the channel is said to be synchronized, and both nodes begin their steady-state transmission routines.

4.3.3 Channel Selection Behavior

Once the transmission operation has reached steady-state, the master begins sending 10-frame bursts of encoded H.264 video data, and the client responds with an ACK frame for every 10 error-free frames it receives. The continued exchange of data and ACK frames serves to keep

the wireless link active, which in turn accommodates the passive coordinated sensing protocol implemented in the example. The sensing coordination between the master and client node is said to be passive because there is not direct exchange of sensing information between nodes, yet the nodes can coordinate their sensing operations by observing the behavior of their counterpart. That is, if the master detects interference and must evacuate the channel, the client will time-out due to the lack of new packets being received and begin its own synchronization routine. Likewise, if the client detects interference that is not empirically measured by the master, the lack of returning acknowledgments will inform the master node that something has gone wrong on the data link.

In this way, sensing for interference or primary users is carried out across several OSI layers. At the physical layer, energy detection is performed independently by each node, the results of which provide an empirical basis for determining the status of a given channel. At the data-link layer, the quality of the wireless link is characterized at the client node by watching for out-of-sequence data frames, and at the master node by keeping track of the time between receiving ACK frames from the client. This provides a means of inferring the channel occupancy status which is independent from the energy detector. If an interfering signal is

too weak to trigger an energy detection evacuation, but strong enough to cause dropped packets, the link quality will inform the network to search for a better channel. Inferring channel quality in this way is slower than using the energy detector, since it requires packet data to transverse the channel, but it is more robust in terms of accommodating interference scenarios where the energy detector is likely to fail.

4.4 Webcam/GnuRadio Interface

The process of moving video data between a USB webcam and the GnuRadio waveform involves the coordination of several different applications and processes. In general, a Linux signal processing pipeline converts raw video data to a streamable format, and a First In First Out (FIFO) buffer manages the flow of data between the pipeline and the GnuRadio waveform. A high level overview of the application interface between GnuRadio and the USB webcam is shown in fig. 10.

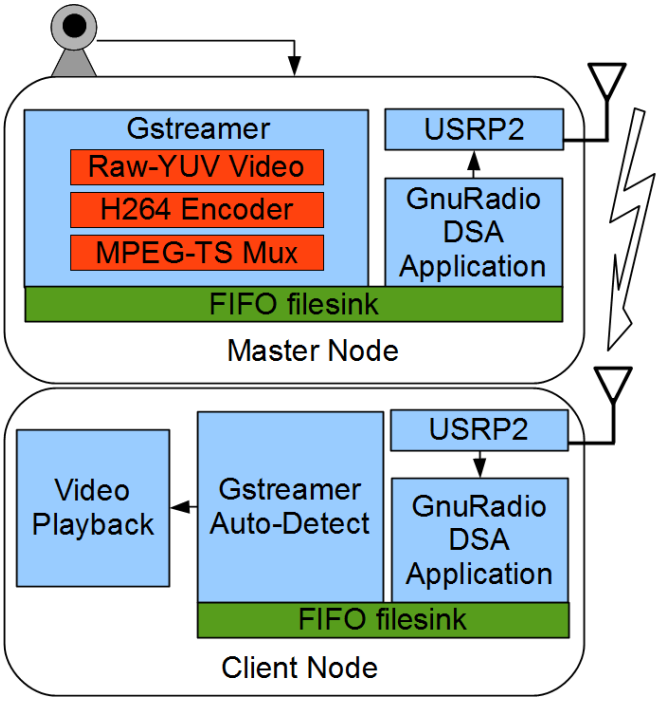

Figure 10. DSA Application Block Diagram

4.4.1 Gstreamer

Gstreamer is an open source multimedia framework which provides a command-line interface for constructing video and audio encoding pipelines. In the example waveform, Gstreamer is used to convert the raw video output of the USB webcam into an H.264 compatible video stream, and to multiplex the encoded video into an MPEG transport stream. The Gstreamer command used for this purpose is as follows:

```
gst-launch -v v4l2src device="/dev/video0" ! video/x-raw-yuv, width=320,
        height=240, framerate=15/1 ! clockoverlay ! x264enc, fps=15,
        sliced-threads=on, input-res=320x240, vbv-bufsize=10, vbv-
        maxrate=300 ! mpegtsmux ! filesink
        location=/home/cornetadmin/myfifo.raw
```

The command is entered as a single line with a single return at the end. The *gst-launch* command is used to initialize and construct the encoding pipeline. Since only video is being encoded, the *-v* flag is used, followed by the Unix device address of the USB webcam (/dev/video0). The exclamation point character is used to indicate the discrete pipeline stages, which are as follows:

- video/x-raw-yuv: Indicates that the input to the pipeline will be YUV formatted, 15 frames per second video with a resolution of 320x240 pixels.
- clockoverlay: Applies a digital time reference to the video based on the time of the local operating system clock.
- X264enc: Open Source H.264 encoder [13] with thread-slicing and variable bit rate encoding turned on. The encoding is set to be computationally efficient and low quality.
- mpegtsmux: An open source MPEG transport stream generator used with default settings.

The MPEG transport stream is an important part of the waveform, as it provides error correction and transport-layer connectivity to the video stream without the need to implement FEC in GnuRadio [15]. The output of the MPEG multiplexer produces a bit stream which must be buffered and used as a data source for the GnuRadio waveform.

4.4.2 FIFO File sink

A FIFO file is a standard file type that can be created in Linux, using the command *mkfifo*. Once it has been created, it may be accessed by one or more programs as an input or output file, but needs to be opened simultaneously on both ends before it becomes writable or readable. This has the effect of creating a universal, high level file buffer which may be used between the video encoding operation and the GnuRadio waveform.

4.5 Summary of Operation

The DSA application is started from within a script, which is used to initialize the Gstreamer pipeline and FIFO buffer. In general, opening the FIFO for reading before it is open for writing results in a blocking procedure call, so it is preferred to open the FIFO for writing first by

initializing the video pipeline, followed by the GnuRadio waveform. The *run_cam* scripts used to start the master node and client node are as follows:

Master:

```
#!/bin/bash
sudo gst-launch -v v4l2src device="/dev/video0" ! video/x-raw-yuv,
        width=320, height=240, framerate=15/1 ! clockoverlay ! x264enc,
        fps=15,  sliced-threads=on,  input-res=320x240,  vbv-bufsize=10,
        vbv-maxrate=300  ! mpegtsmux ! filesink
        location=/home/cornetadmin/myfifo.raw & sudo python
        /home/cornetadmin/src/gnuradio/gnuradio-
        examples/python/digital/IMMasterSlaventia.py -e eth1 --master &
```

Client:

```
#!/bin/bash
sudo gst-launch playbin framerate=15/1,
        uri=file:///home/cornetadmin/myfifo2.raw & sudo python
        /home/cornetadmin/src/gnuradio/gnuradio-
        examples/python/digital/IMMasterSlaventia_slv.py -e eth1
```

4.5.1 Synchronization

The waveform operation begins when both nodes are brought online, and begin to search for a peer node. The master node asserts itself randomly in channels deemed unoccupied via energy detection by broadcasting a synchronization beacon. Similarly, the client node senses for free channels and attempts to hail the master node by transmitting a beacon of it's own.

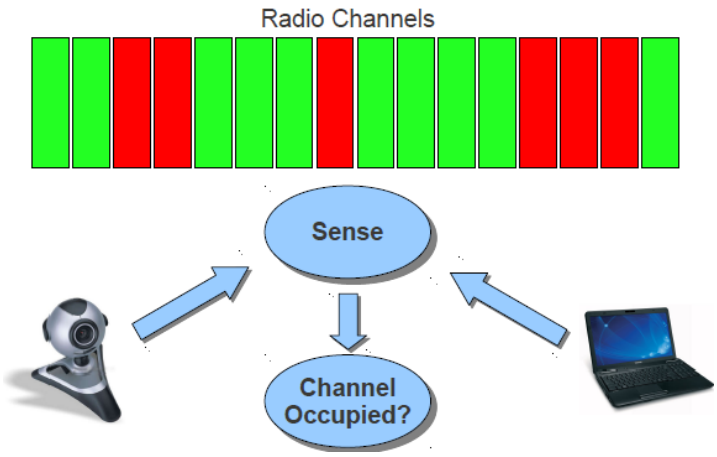

Figure 11. DSA Synchronization

4.5.2 Convergence

Convergence is achieved in one of two ways, and occurs only when both the master and client node are tuned to the same channel. If the client sees a valid master beacon, it initiates a three way handshake by responding with a SYN frame. If the master views the client beacon, it asserts itself on the channel using its own beacon, and waits for the client to respond with a SYN frame. Once the three way handshake has been successfully completed, the nodes move into steady-state transmission.

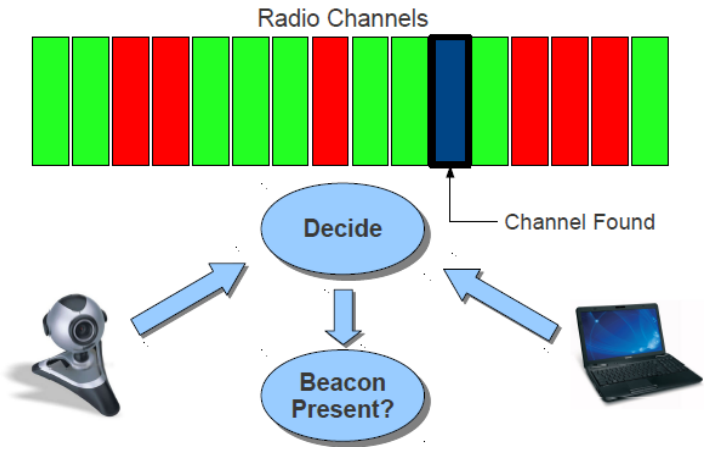

Figure 12. DSA Convergence

4.5.3 Transmission

Once synchronized, the transmission routine requires the master node to transmit 10 frame bursts of video data. The client node responds with an ACK frame for every 10 frames of video data received. The data link remains active as long as neither the master or client timeout counters reach zero. The timeout length for the master is approximately the time required to send 25 frames, and is reset upon each ACK received from the client. The timeout length for the client is approximately equal to 10 frame lengths, and is reset each time a frame is received successfully. Following each successful ACK exchange, both nodes enter a sensing routine for approximately one frame length in order to sense the channel for interference.

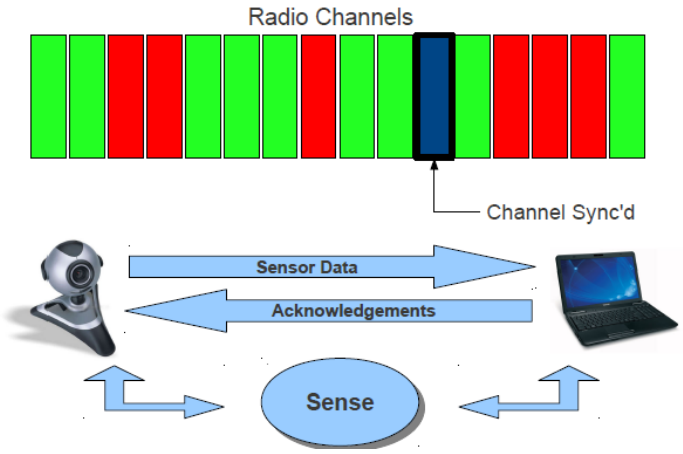

Figure 13. Steady-State Transmission

4.5.4 Channel Evacuation

A channel evacuation event will occur when there is interference present on the synchronized channel. The waveform uses two methods for detecting interference – both nodes perform energy detection on the channel following a successful ACK exchange, and additionally keep track of the length between ACK frames (for the master) and the length between video frames (for the client). If the energy detection operation returns a value greater than a threshold value, the node(s) evacuate the channel immediately. Once one node evacuates the channel, its peer will likewise timeout and enter the re-synchronization routine, even if it does not detect energy on the channel.

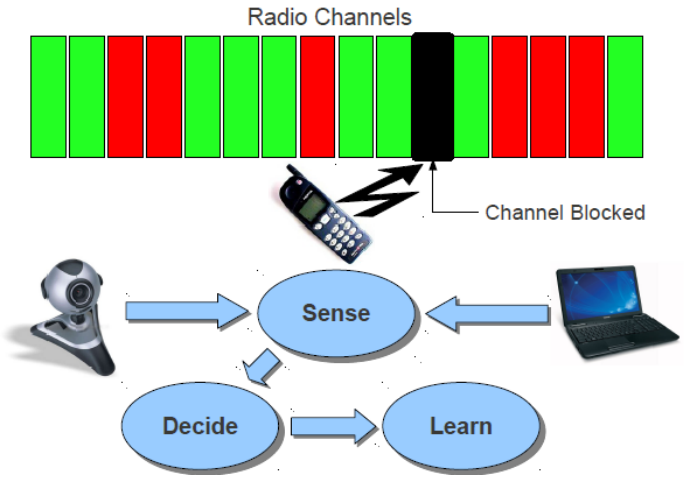

Figure 14. DSA Channel Evacuation

4.5.5 Re-Convergence

Following a channel evacuation event, the node enter a re-synchronization routine which is similar to the original synchronization routine. The nodes learn from the evacuation event, and mark the previous channel as invalid until a new sensing routine determines otherwise. After a new channel is found, the nodes once again enter the steady-state transmission routine until interference is detected.

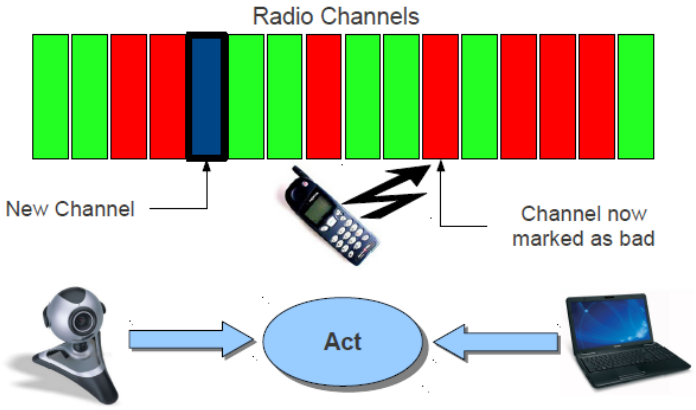

Figure 15. DSA Re-Convergence

5

Using CORNET

5.1 Overview

This section is presented as a reference manual for CORNET users. The manual is intended to provide new users with a step-by-step guide to performing certain fundamental operations on CORNET nodes, such as logging into the nodes, managing their user directory, running waveforms and installing custom software.

Much of the learning curve associated with CORNET operation stems from inexperience working in remote Unix environments. Users who wish to run experiments on CORNET should be familiar with the USRP2 platform, and GnuRadio in general, prior to requesting a password. Users will not find a large library of ready-to-run example waveforms, but rather tools and a 'blank-slate' SDR platform they may utilize to design, prototype or deploy waveforms of their own construction. This section is

intended to aid users to this end by familiarizing them with the basic mechanics of CORNET operation and usage.

5.2 Logging Into CORNET

Depending on their OS choice, users may log into CORNET nodes by using either a SSH or an NX remote desktop client. Each node is accessed individually by establishing an SSH connection with the CORNET gateway on the TCP port corresponding to that node. The gateway is configured to forward SSH traffic based on these port assignments – node 1-1 corresponds to port 7001, node 1-2 corresponds to port 7002, and so on.

5.2.1 Linux/Unix Secure Shell

From Linux or Unix-based Apple operating systems, open a new terminal and log into node 1-1 using the following command:

```
ssh [user]@cornet.wireless.vt.edu -p [port]
```

Where *[user]* is the assigned user name and *[port]* is 7001 for node 1-1. The first time a user logs into a node from an unrecognized machine, they

will need to accept the SSH key from the host node by typing *yes* when prompted (Fig. 16). Once they key exchange has been approved, the user will be prompted to enter their assigned password (Fig. 17).

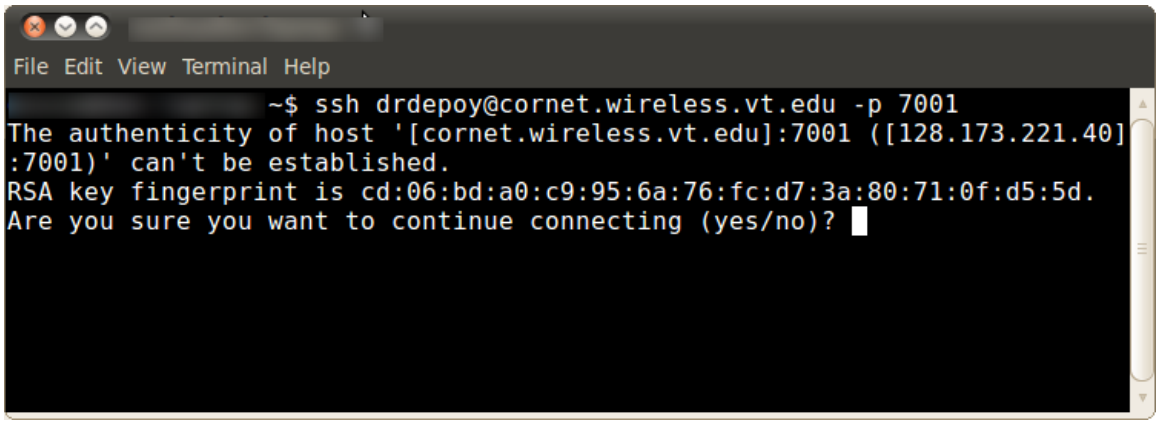

Figure 16. Accept SSH RSA Key

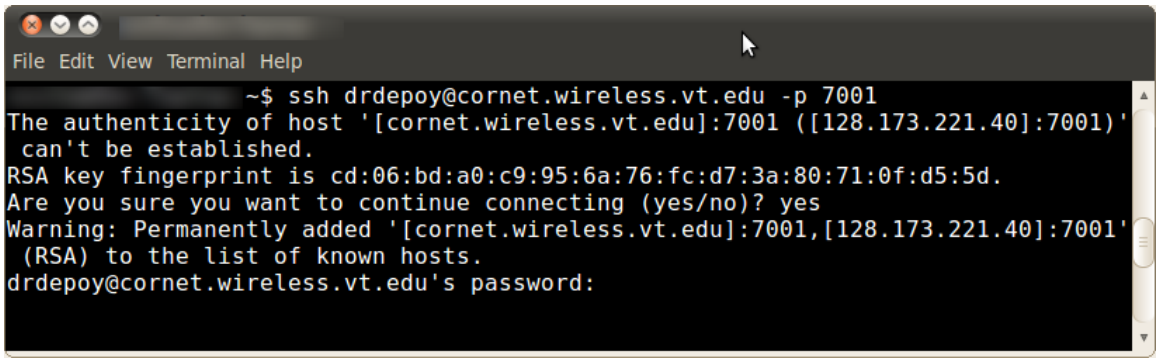

Figure 17. Password Prompt

The SSH key pairs associated with each individual node are re-generated when the nodes are imaged, so users may occasionally receive an error about an incorrect key pair, which prevents them from logging in. In this case, the simplest solution is to clear the local SSH host list using

the following command, in order to initiate a new RSA key exchange on the next login attempt:

```
rm ~/.ssh/known_hosts
```

In some cases, users may not wish to remove all their SSH keys. In this case, the following command should be used instead:

```
ssh-keygen -R [cornet.wireless.vt.edu]:7001
```

In the previous command, *[cornet.wireless.vt.edu]* is not a bracketed parameter, and should be typed verbatim, brackets included. The port number following the colon (7001 in this example) is the port number of the node which requires updated RSA key authentication.

In addition to running command line applications on the remote CORNET shell, it is also possible to forward simple X.11 GUI applications over the standard SSH connection. In order to do this, the *-X* flag should be added to the SSH command. In addition, the *-C* flag will enable link compression, and often improves GUI performance when combined with the *-X* flag:

```
ssh -XC [user]@cornet.wireless.vt.edu -p [port]
```

It is important that users remember to end their SSH sessions cleanly, ensuring that no software is left running under their user. To close the session cleanly, simply type *exit* in the terminal. Occasionally a dropped wireless connection or carelessness results in a user remaining logged into the node once a new session has been established. To force-close all processes started under a previous session and end the sessions, use the following command when logged into a remote node:

```
sudo pkill -u [user]
```

Where *[user]* is the user being logged out – this command will end the current SSH session, as well as any previously established sessions, and the user will have to login to the node again.

5.2.2 Windows Secure Shell

SSH is not included in Windows based operating systems by default, but there are several free compatible applications from which to choose.

One popular option is PuTTY [23], which is free, cross platform, and open source. Instructions will be given with PuTTY screen shots, but the configuration will be similar for most SSH clients.

Fig. 18 shows the initial configuration of a CORNET SSH connection using PuTTY. The host name dialogue should read *cornet.wireless.vt.edu* and the port dialogue should contain the port corresponding to the desired node, as shown below. All other settings may be left as defaults, or adjusted based on user preference. Pressing the *Open* button will initialize the SSH connection, and a user may be prompted to approve the RSA key exchange (Fig. 19) if it is the first time they are connecting to a given node. After approving the key exchange, the user is prompted for both their login and password from within the PuTTY shell (Fig. 20).

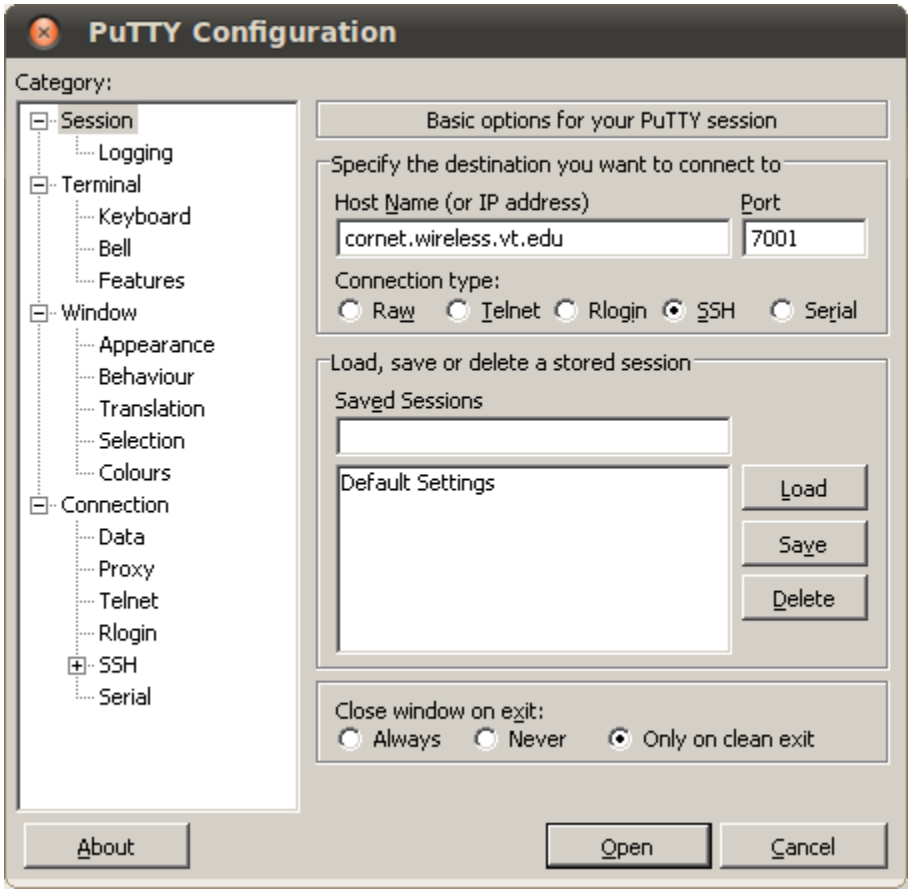

Figure 18. PuTTY Configuration

Figure 19. Accept SSH Certificate (PuTTY)

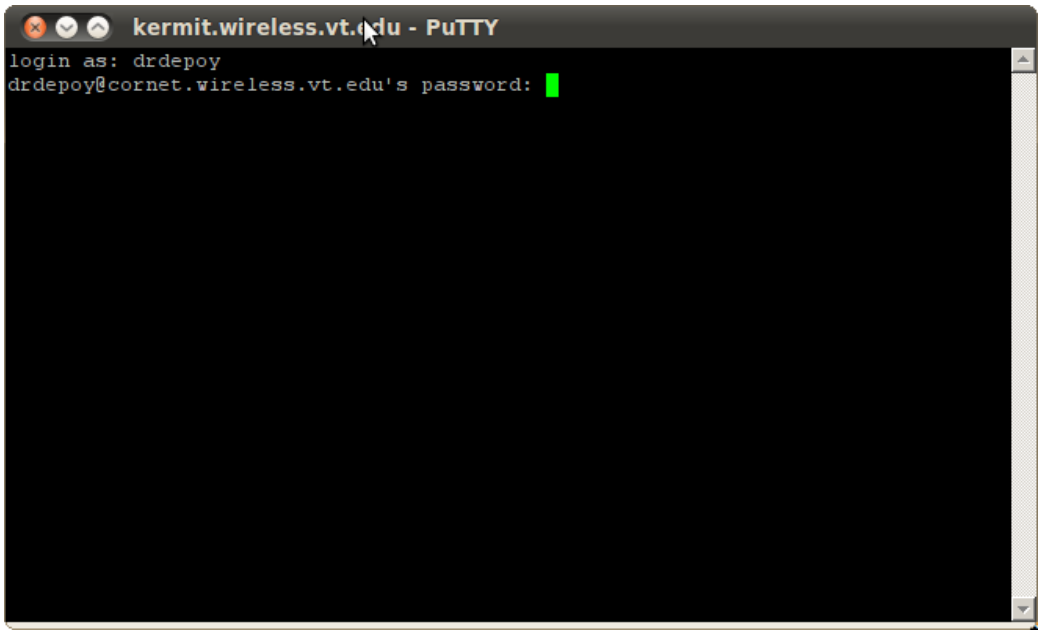

Figure 20. PuTTY Password Prompt

Unfortunately, configuring X11 forwarding from a PuTTY shell in Windows is not a trivial process, since Windows cannot render X11 without modifications which are beyond the scope of this document. Windows users who require remote GUI interaction should use the NX remote desktop client.

5.2.3 NX Remote Desktop Client

NX Technology is a cross platform remote desktop standard which operates over a modified SSH session. The standard is maintained by the company NoMachine [21] but versions prior to release 4.0 are open source and licensed under the Gnu Public License. There are a number of free and open source remote desktop clients which are compatible with NX Technology, and NoMachine provides their official client software for free [22]. The following instructions include screen shots from the official NoMachine client for Linux, and the configuration should be similar for different clients and operating systems.

Open the *NX Connection Wizard* and click *Next* on the first screen to bring up the session information prompt (Fig. 21). Select a nickname and type it into the S*ession* prompt. Use *cornet.wireless.vt.edu* as the host

address, and 7001 (depending on the node to be accessed) as the port. The connection type slider controls the level of compression and frame rate used to rendering the remote desktop, and should be set to *ADSL* or lower in most circumstances unless a high quality display is required. Continue by clicking *Next.*

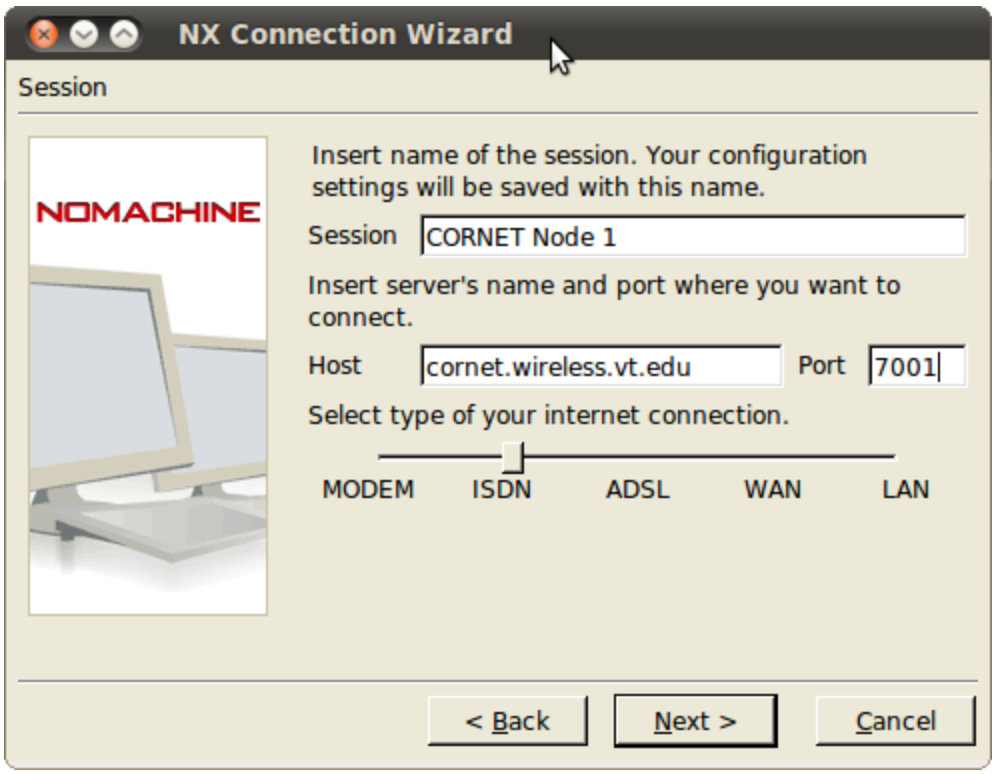

Figure 21. NX Client Session Configuration

On the following screen (Fig. 22), select *Unix* and *Gnome* as the desktop type to use, and leave the display size set to *Available Area,* unless otherwise desired. Leave *Disable encryption of all traffic* left

unchecked, and click *Next* to continue to the next screen, and select *Finish* to complete the connection setup. Repeat this process to configure a session for each node to be accessed.

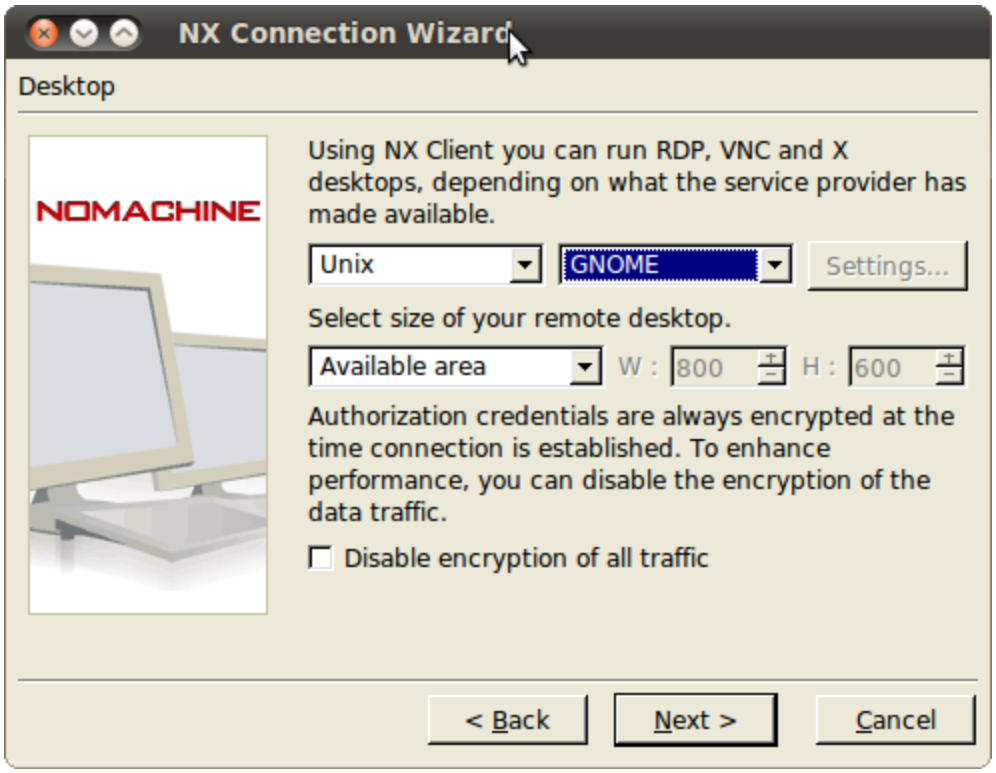

Figure 22. NX Client Desktop Setup

When configuration is complete, the login prompt will be displayed (Fig. 23). Type the correct user name and password into the dialogue and press *Login* to initialize the session. Users may have to approve the SSH RSA key exchange if they have not logged into the node since it was last imaged (Fig. 24). Upon successfully logging into a node, the user will be

presented with a remote desktop window contained within their local desktop environment (Fig. 25). The desktop displayed is the contents of the *Desktop* folder within the user's NFS home directory. If no *Desktop* folder is present, an empty one will be created following the first successful login.

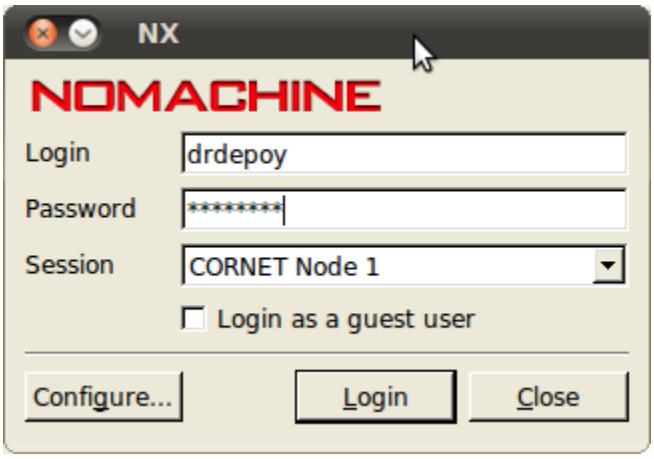

Figure 23. NX Client Login Prompt

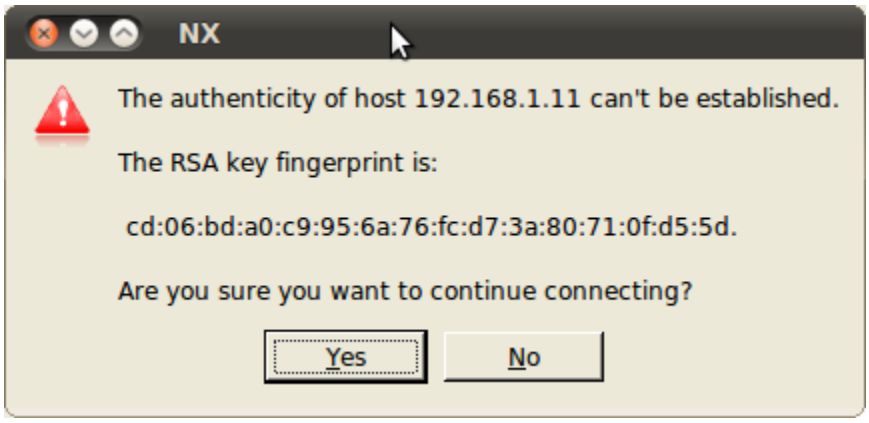

Figure 24. NX Client RSA Key Exchange

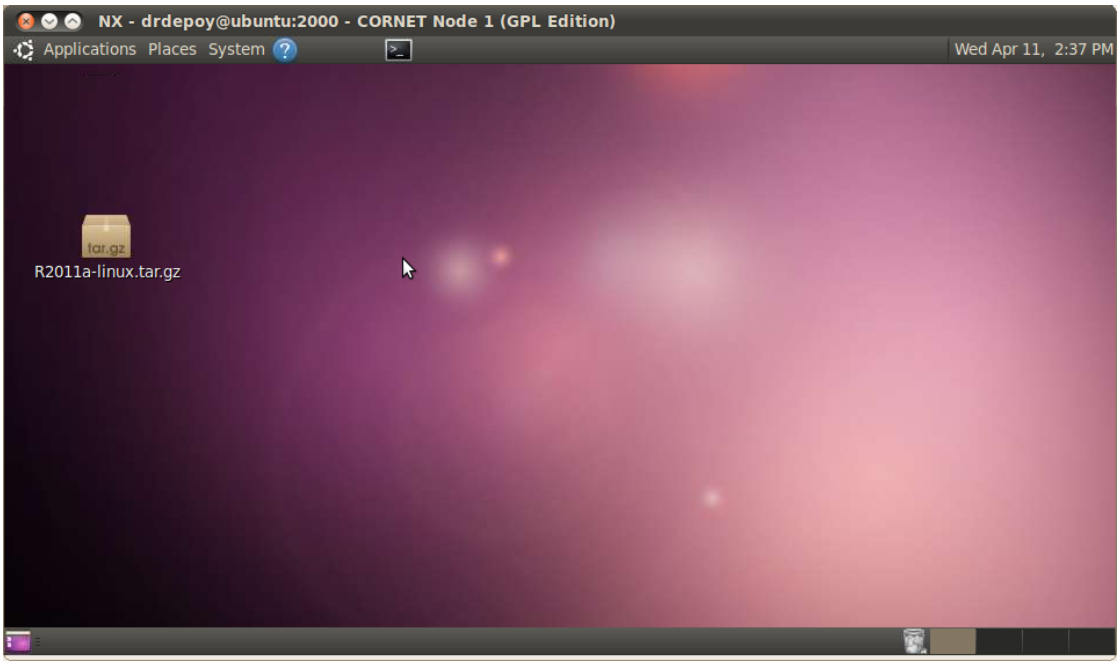

Figure 25. NX Remote Desktop

To stop the NX session, and log out of the node, simply click on the default "close window" button for the OS in use – shown as the orange *X* in figure 25. The exit dialogue (fig. 26) will appear, and users should select *End* to terminate the session and any processes started by the user. Alternately, a user can choose *Disconnect* to save the state of the session for the next login. In most cases, selecting *Disconnect* is not recommended except for brief downtime, since there is no guarantee that the session will still exist when the user returns if the node needs to be power cycled, or imaged for another user.

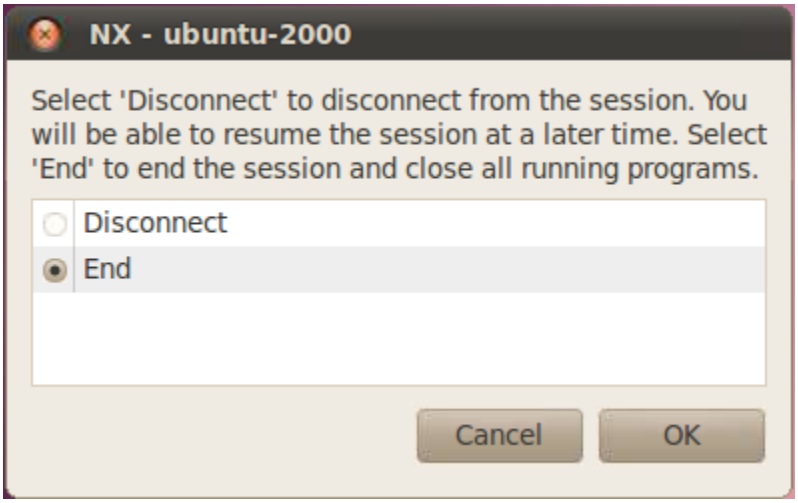

Figure 26. NX Client Session Exit Prompt

5.3 User Directory Operations

Each user is provided with a private NFS directory in which they may develop custom waveforms, compile code or store experimental data. The user directories are the default (home) folder the user is presented upon logging into the network, and are configured to be persistent across all nodes, and are not affected when individual nodes are re-imaged. For this reason, users are strongly encouraged to backup any important files, configurations, scripts or other work in their personal directories to protect against accidental loss or corruption.

5.3.1 Directories and Permissions

When a user logs into a CORNET node, their personal user directory, which is located at /users/[user name], is used as the home directory for the remote SSH or NX session. Each user is the owner of their NFS directory, though the permissions allow other users to read, write and execute within these directories by default. Users may choose to make their entire directory private, or to create private folders within their home directory. To create a new folder, use the following shell command:

```
mkdir /users/[user name]/folder
```

Users may set permissions on any folder or file which they own by using the *chmod* command. To set a folder so that it can only be accessed by the owner, the following command is used:

```
chmod -R 700 /users/[user name]/folder
```

The previous command will prevent any other users from viewing the contents of /folder. For less stringent permissions, the following command will allow other users to read and execute code within a folder, but not to modify the contents:

```
chmod -R 755 /users/[user name]/folder
```

Non-administrator users are limited to executing the *chmod* and *chown* commands as a non-root user, meaning they cannot use *sudo* to alter directory permissions. This is done in order to prevent users from altering the permissions set on folders and files owned by other users. In addition, the *visudo* command, which is used to manage *sudo* access, is also restricted for non-administrators. For this reason, it is important that users create and modify files and folders within their user directory as themselves, without prepending the *sudo* command. For more information about changing permissions with the *chmod* command, see the documentation [9].

5.3.2 File Management

In order to develop waveforms, or perform experiments using CORNET nodes, it is common for users to move files between their local computing platform, and their CORNET user directory. In addition, users may wish to modify example code from GnuRadio or OSSIE for their experiments.

To copy code from a local machine to a remote CORNET node, the *scp* command is issued in a local terminal (or from PuTTY in Windows). For example, to move the local file /home/foo.dat to /users/ [user_name]/folder on CORNET node 1-1, the following command is issued locally:

```
scp -P 7001 /home/foo.dat [user_name]@cornet.wireless.vt          \
        .edu:/users/[user_name]/folder
```

To move the remote file located in /users/[user_name]/folder/foo.dat back into the local directory /home/, the following command is issued locally:

```
scp -P 7001 [user_name]@cornet.wireless.vt.edu:/users/            \
        [user_name]/folder/foo.dat /home/
```

In general, the syntax of the command is

```
scp -P [port] [source] [destination]
```

Where *[source]* and *[destination]* may be a local or remote file or directory.

Users who wish to modify example waveforms from GnuRadio or OSSIE are strongly encouraged to make copies of the code they wish to

modify within their home folder, in order to preserve the examples in their original form. Many of the example waveforms are split into multiple files to make them easier to parse. As such, it is recommended that users clone the whole directory for the examples they with to modify. For example, the *benchmark_tx.py* waveform depends on *transmit_path.py* and *usrp_transmit_path.py.* A user who wishes to modify this example should use the following commands to create a new folder within their home directory and copy the example waveform to that directory:

```
mkdir /users/[user_name]/new_folder

cp -R /home/cornetadmin/gnuradio_uhd/gnuradio-                    \
        examples/python/digital/ /users/[user_name]/new_folder
```

5.4 Example Waveforms and Tools

The standard OS image on CORNET includes a number of GnuRadio and OSSIE waveforms which users may find useful as general tools, or as examples from which to build new applications depending on their ability.

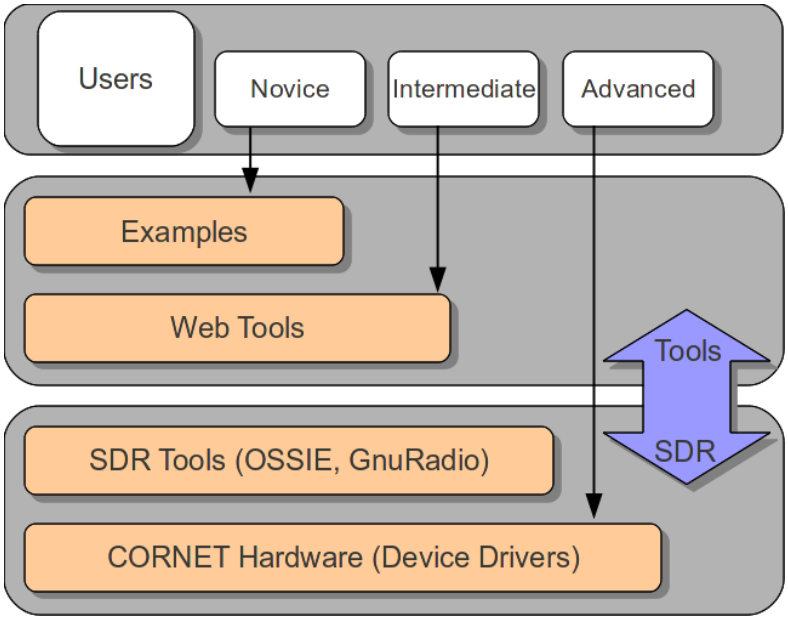

Figure 27. Examples and Tools

5.4.1 USRP2 Diagnostics

When troubleshooting custom waveforms, it is often necessary to troubleshoot the interface between the USRP2 and the corresponding host server. CORNET nodes must be configured in a specific way in order to use the USRP2 peripheral, and there are several tools available to check for connectivity. Instructions within this section assume the USRP2 and host server are both configured with UHD firmware.

The first step to check for USRP2 connectivity is to determine the IP address assigned to the Ethernet port *eth1* by using the command *ifconfig*

eth1 from a terminal. The IP address should be 192.168.10.1 as shown in figure 28.

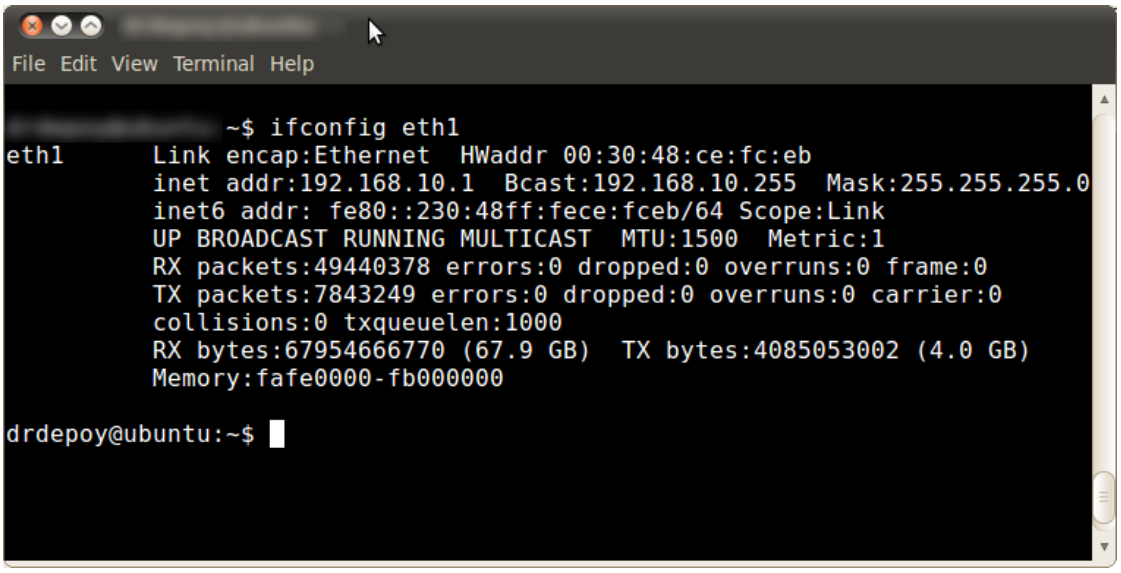

Figure 28. USRP2 IP Address

If *eth1* is configured with the correct IP address, the next step is to determine whether the device can be pinged, using the following command:

```
ping 192.168.10.2
```

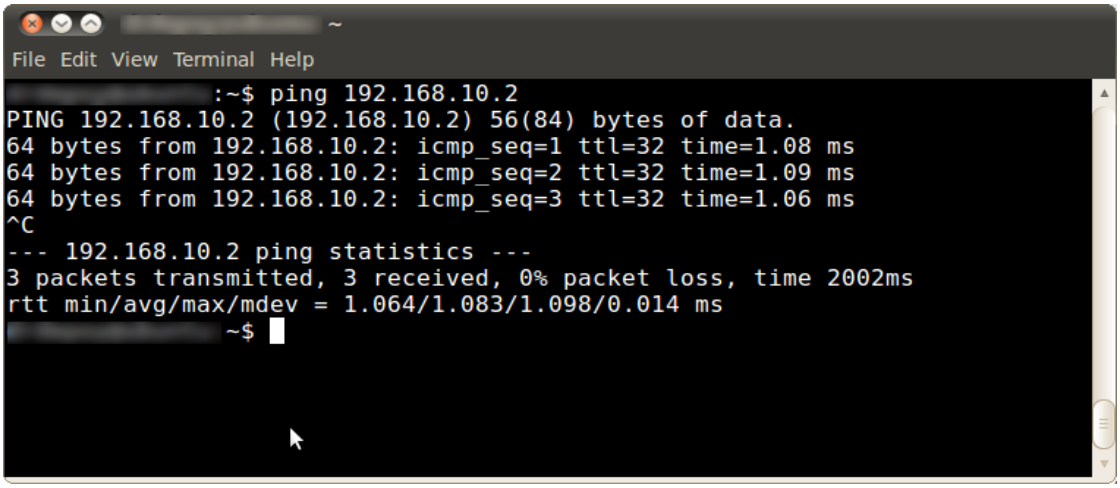

Figure 29. USRP2 Ping Response

The output from the ping operation should appear similar to fig. 29 if the USRP2 is connected correctly. Finally, the last diagnostic tool available to troubleshoot USRP2 connectivity is the *uhd_find_devices* command, which will produce the output shown in fig. 30 if the node is fully operational and connected. The *ifconfig* and *ping* commands are useful for determining whether the USRP2 and host node are configured with the correct networking settings, and the *uhd_find_devices* command will return true if the network is configured correctly and the correct UHD host drivers are installed on the node.

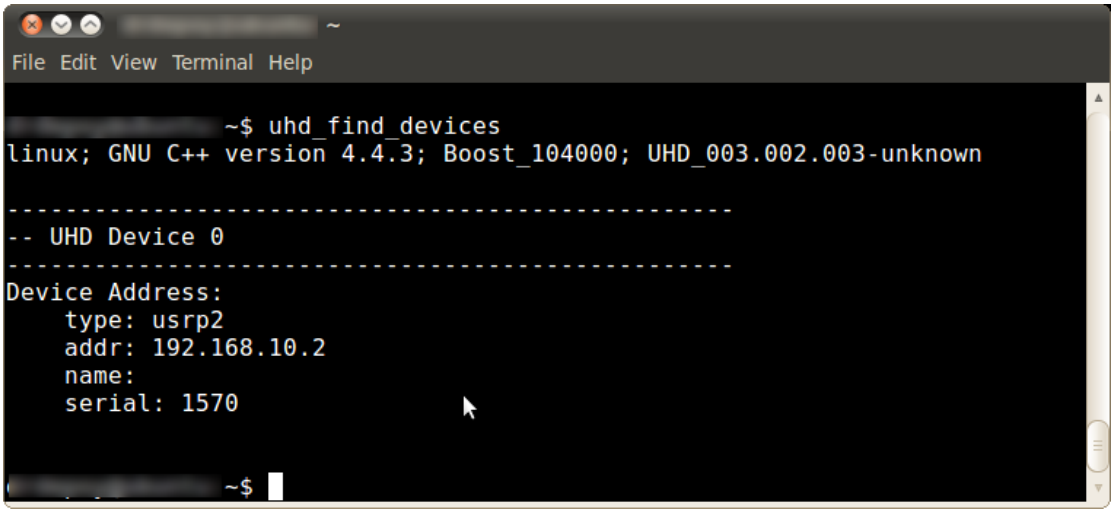

Figure 30. USRP2 UHD Configuration

5.4.2 GnuRadio Tools

GnuRadio includes several built-in SDR tools which are useful for a variety of purposes. Among these, the built in signal generator and FFT tools are especially valuable to users running CR experiments, or designing prototype waveforms to this end. The signal generator waveforms may be used to insert several different types of narrow-band or wide-band signals into a given frequency band, while the FFT plotting tools are useful for demonstration and debugging purposes in applications where spectrum visualization is desired. These GnuRadio tools are installed system-wide, and can be accessed by all users from any directory.

There are two signal generator options, depending on whether a GUI or command-line interface is desired. To generate and broadcast a signal using the command-line, the following template command should be filled in as needed:

```
uhd_siggen.py -f [freq] -g [gain] --amplitude=[amplitude]  --[type]
```

Where *[freq]* is the RF center frequency of the generated signal, *[gain]* is the transmit gain, from 0 to 25, *[amplitude]* is the signal amplitude from 0 to 1, and *[type]* is the type of waveform to transmit. Additional waveform options can be displayed by using the --*help* flag in place of command flags shown above.

```
uhd_siggen.py --help
```

An interactive signal generator tool (fig. 31) can be accessed using the *uhd_siggen_gui.py* command from the command line. The GUI based waveform can be configured in real time for the same functionality as the command-line-only version.

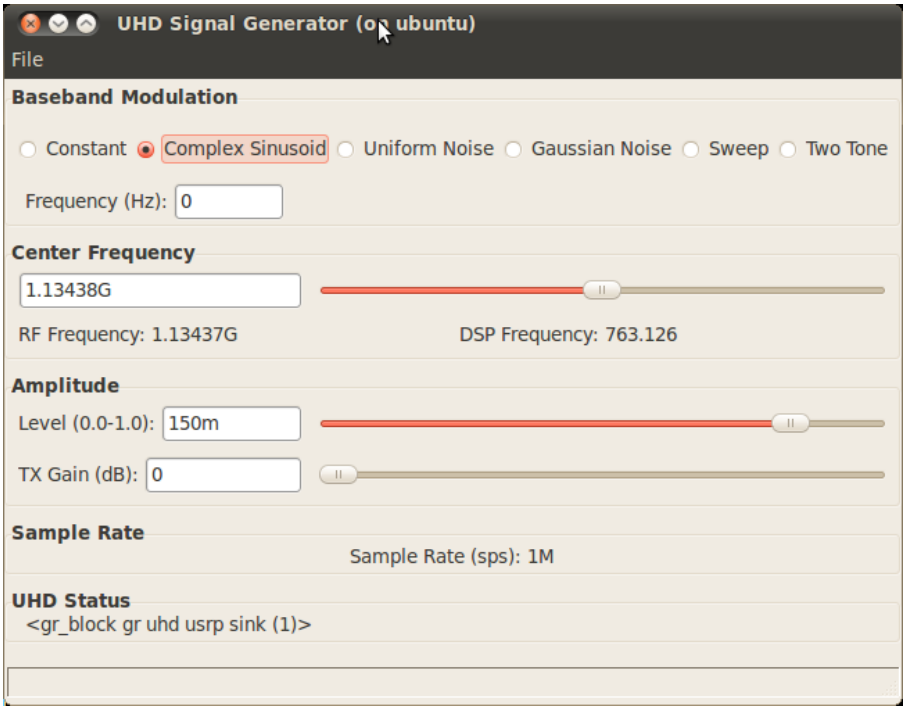

Figure 31. Signal Generator GUI

In addition to the built-in signal generator tools, the built-in FFT display can provide a useful means of visualizing the transmission properties of a given waveform, or for demonstration purposes. The FFT plot tools are accessed using the following command template:

```
uhd_fft.py -f [freq] -s [sample rate]
```

Where *[freq]* is the receiver frequency, and *[sample rate]* sets the bandwidth of the displayed FFT plot. Fig. 32 shows an example FFT plot

of a node receiving band-limited white noise which is being generated from a neighboring node, using the signal generator tools outlined above.

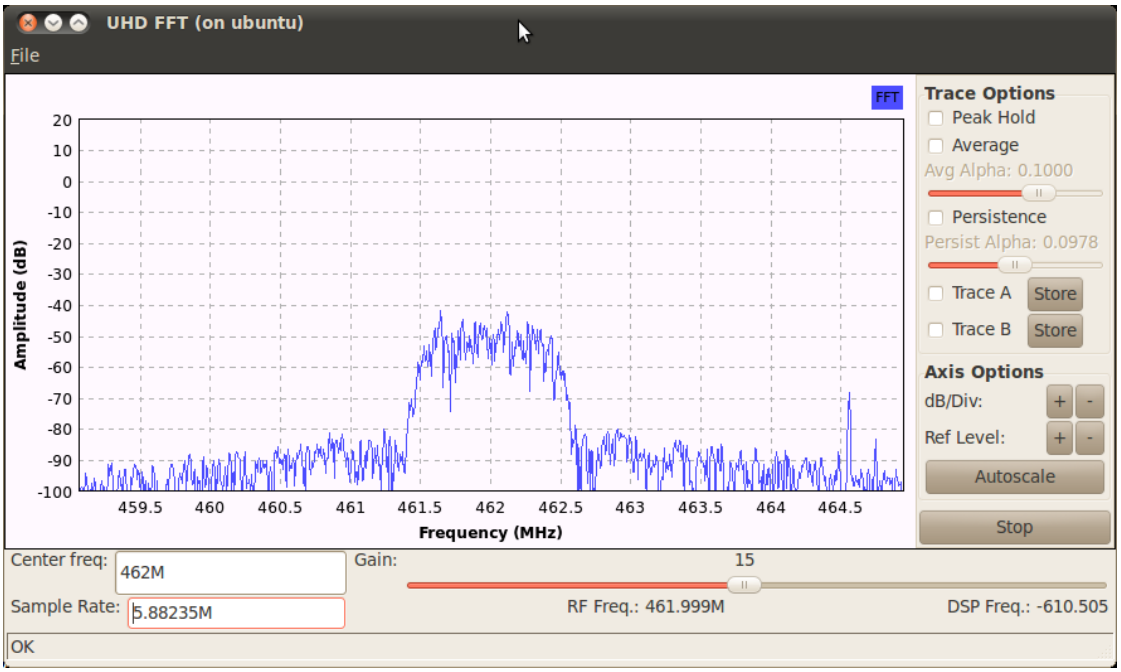

Figure 32. FFT Plot Tool

It should be noted that in order to use any GUI applications from an SSH connection, the X11 forwarding flag must be set when the SSH connection is initialized. Additionally, the kernel level Ethernet buffers may need to be re-sized to prevent over-runs and under-runs from occurring when using high sampling rates. The following commands will set the Ethernet kernel buffers to the correct length:

```
sudo sysctl -w net.core.rmem_max=50000000
sudo sysctl -w net.core.wmem_max=1048576
```

5.4.3 Benchmark Example Waveforms

Benchmark waveforms provide a good way for users to get acquainted with basic CORNET node operation, as well as the structure of GnuRadio Python code. The waveforms are designed as a simple and quick way to establish a one-way digital data link between a transmitter node, running a *benchmark_tx* waveform, and a receiver node, running a *benchmark_rx* waveform. This data link is useful for inferring the channel quality between two fixed nodes by altering various transmission parameters such as gain, symbol rate, and bandwidth and observing the dropped-packet rate that results. In addition, the benchmark waveforms provide a useful physical layer template which may be modified and used as the basis for a CR waveform.

The benchmark waveform code is located on each node at /home/cornetadmin/gnuradio_uhd/gr-digital/examples/ and contains example code to run both narrow-band and OFDM waveforms. To run the example code, a user must be logged onto two nodes which are

located reasonably close together. On the first node, the receiver waveform is started:

```
cd /home/cornetadmin/gnuradio_uhd/gr-digital/examples/narrowband
python benchmark_rx.py -f [freq]
```

On the second node, the transmitter waveform is started:

```
cd /home/cornetadmin/gnuradio_uhd/gr-digital/examples/narrowband
python benchmark_tx.py -f [freq]
```

If successful, the output on the first (receiver) node should be similar to figure 33, and the the output on the second (transmitter) node, should be similar to figure 34. The data links can be configured in a variety of ways in order to make them faster or more robust. A complete list of options can be viewed by appending the --*help* flag to the benchmark commands:

```
python benchmark_tx.py --help
```

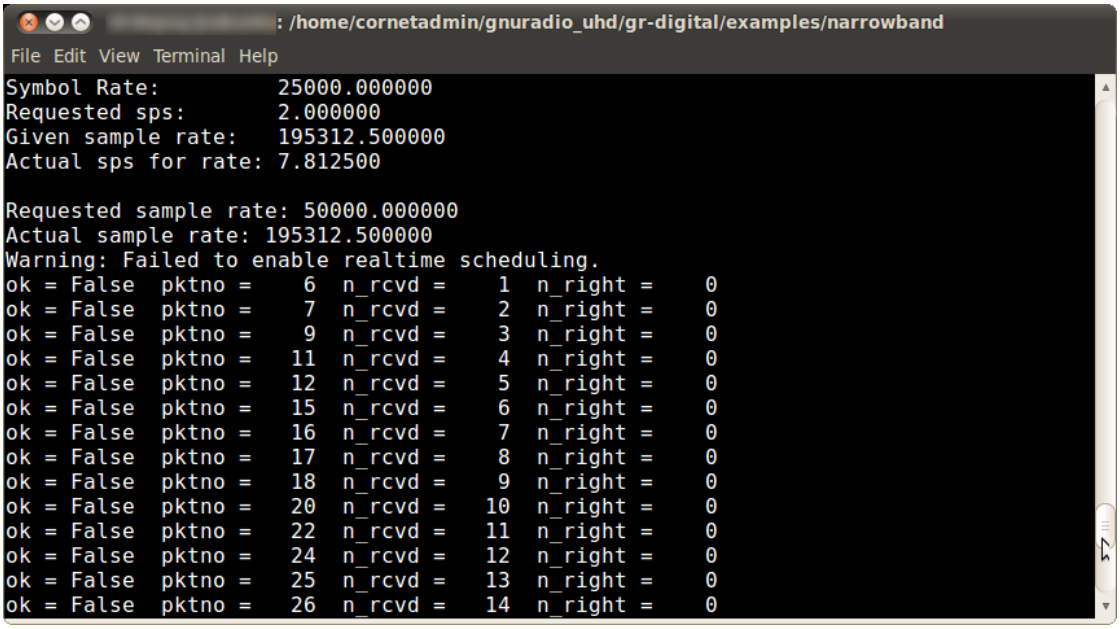

Figure 33. RX Benchmark Output

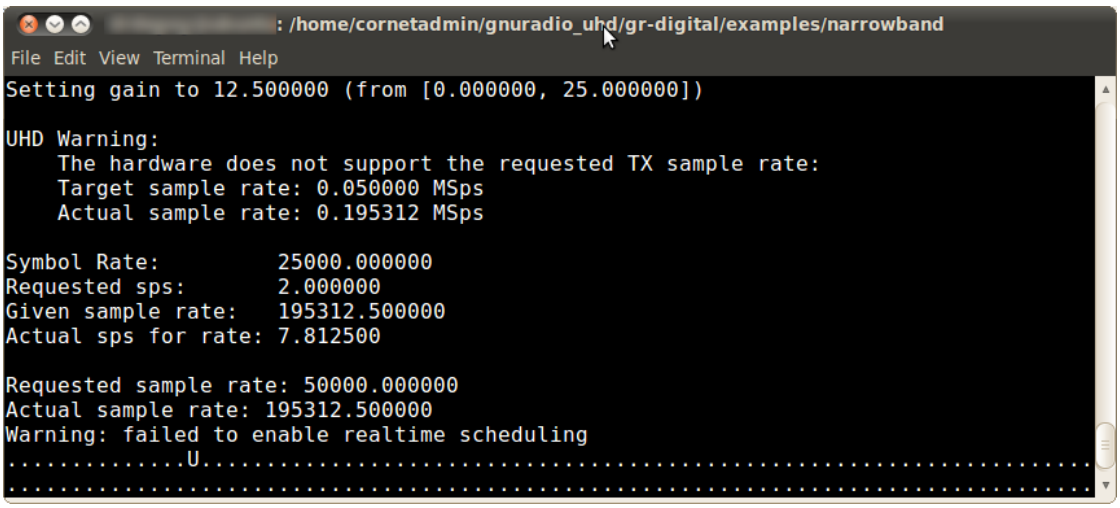

Figure 34. TX Benchmark Output

5.4.4 GnuRadio Companion

GnuRadio Companion is a drag-and-drop waveform development GUI environment for GnuRadio, which is useful for rapid prototyping and testing. In addition, it may be used as a starting point for developing the physical layer operation of a CR application, from which python code can be automatically generated and integrated into a larger protocol stack implementation. A short example is given here in order to demonstrate the basic design flow associated with using GnuRadio Companion.

GRC can be successfully run from an SSH session with X11 forwarding enabled, but it runs better from within an NX remote desktop session.

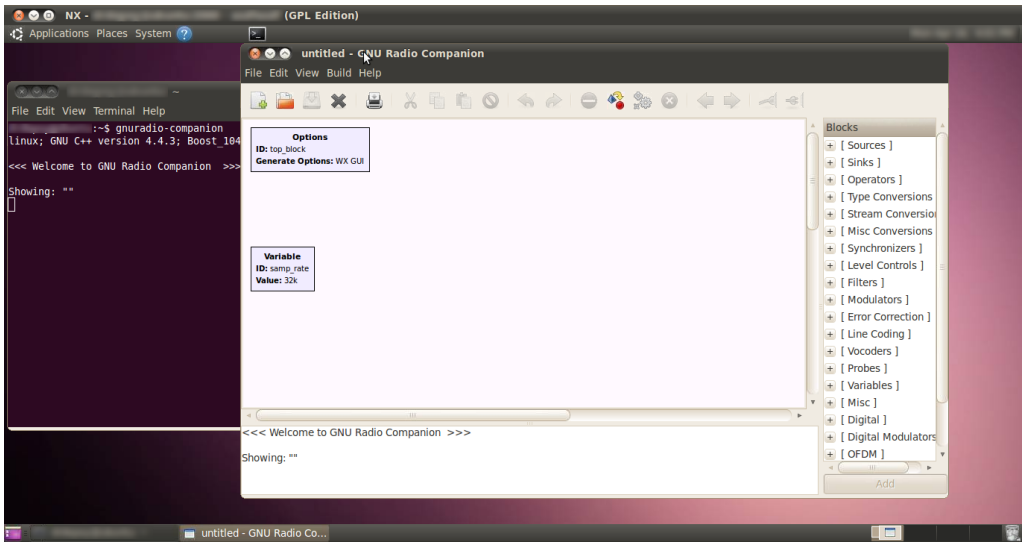

Figure 35. GnuRadio Companion

Once logged into a node, GRC may be started by opening a new terminal and typing *gnuradio-companion*, which should open a window similar to that shown in fig. 35. To make a waveform to display an FFT plot, drag the components *UHD:USRP Source* and *WX GUI FFT Sink* from the panel on the right, into the main work area on the left (fig. 36). Connect the components by clicking one blue port, followed by the other. A black arrow is displayed between connected ports.

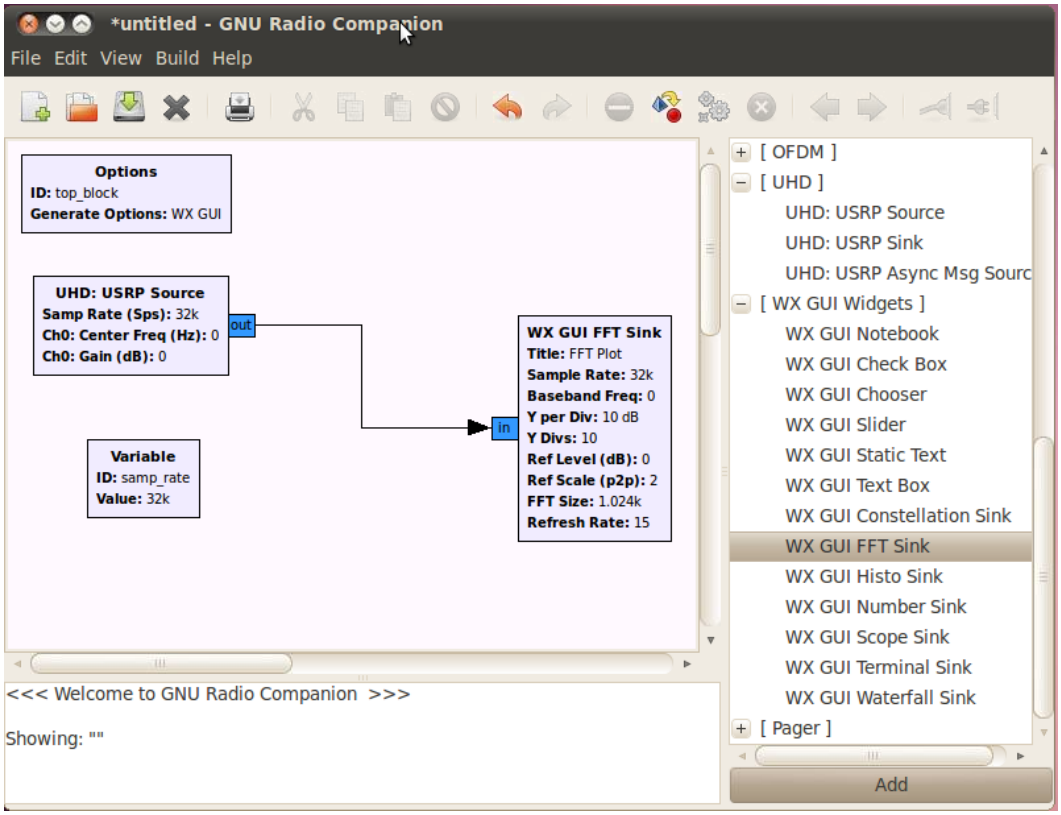

Figure 36. FFT Waveform

Double-click on the *USRP Source* component in order to display its configuration parmeters (fig. 37), and set the frequency to 750MHz. Press *OK* to save the changes.

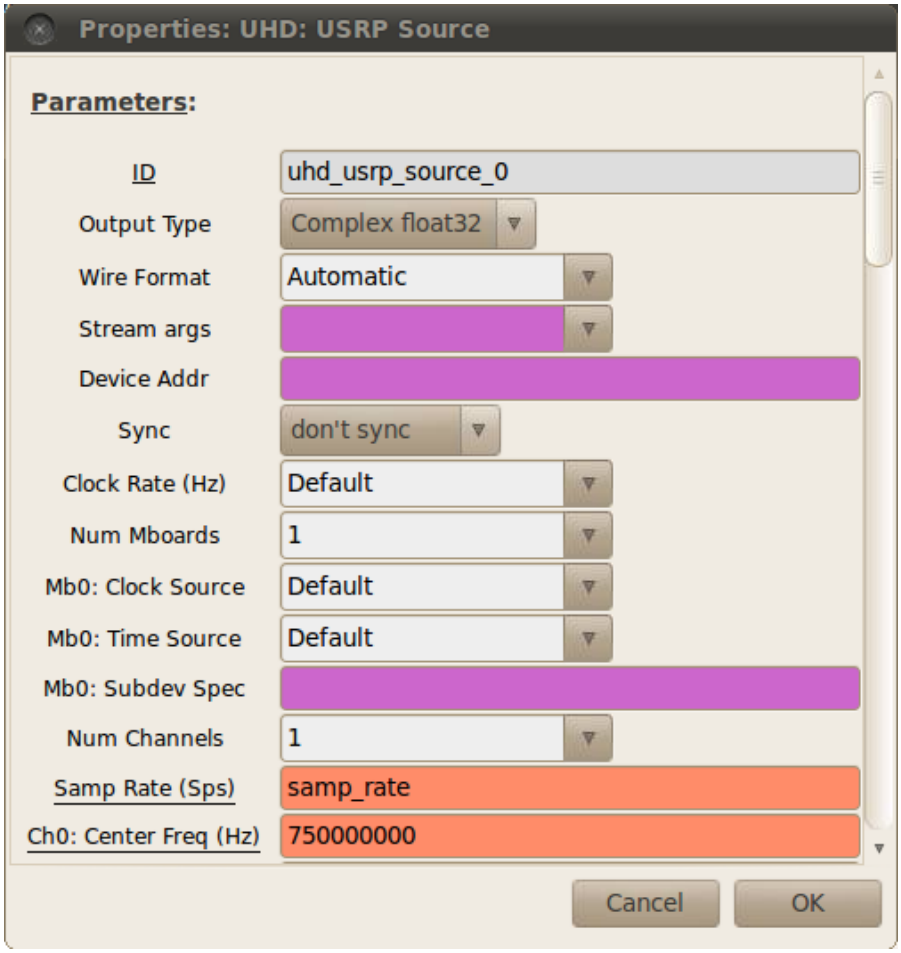

Figure 37. USRP Source Properties

Change the sampling frequency of the waveform by similarly double-clicking on the *samp_rate* component and setting the value to *12M*. Save the waveform and generate the flow graph and Python code by clicking on the button shown in fig. 38. If there are errors during the flow graph

generation, they will be reported in the terminal in the lower left hand corner of the GUI display.

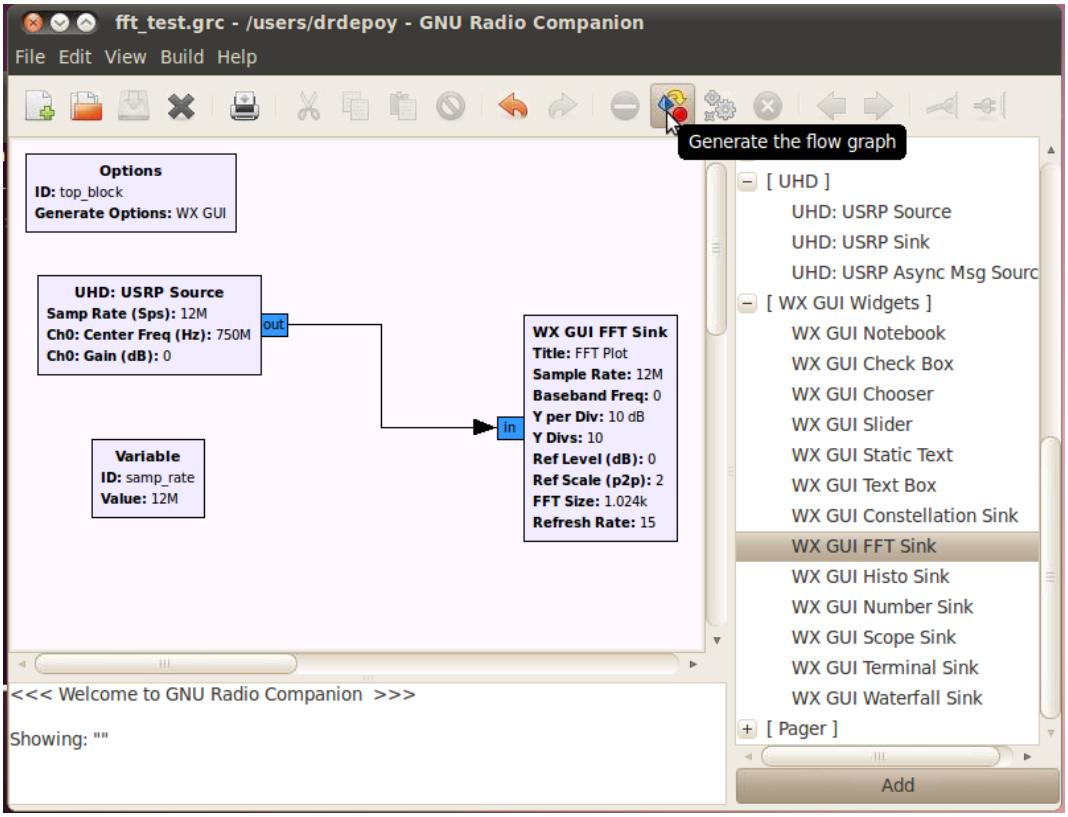

Figure 38. Generate the Flow Graph

If there are no errors, start the waveform by pressing the cog-shaped button next to the *Generate Flow Graph* button. The waveform should start, and an FFT plot should be displayed. The FFT display properties on the left panel can be further adjusted in order to achieve the desired spectrum view.

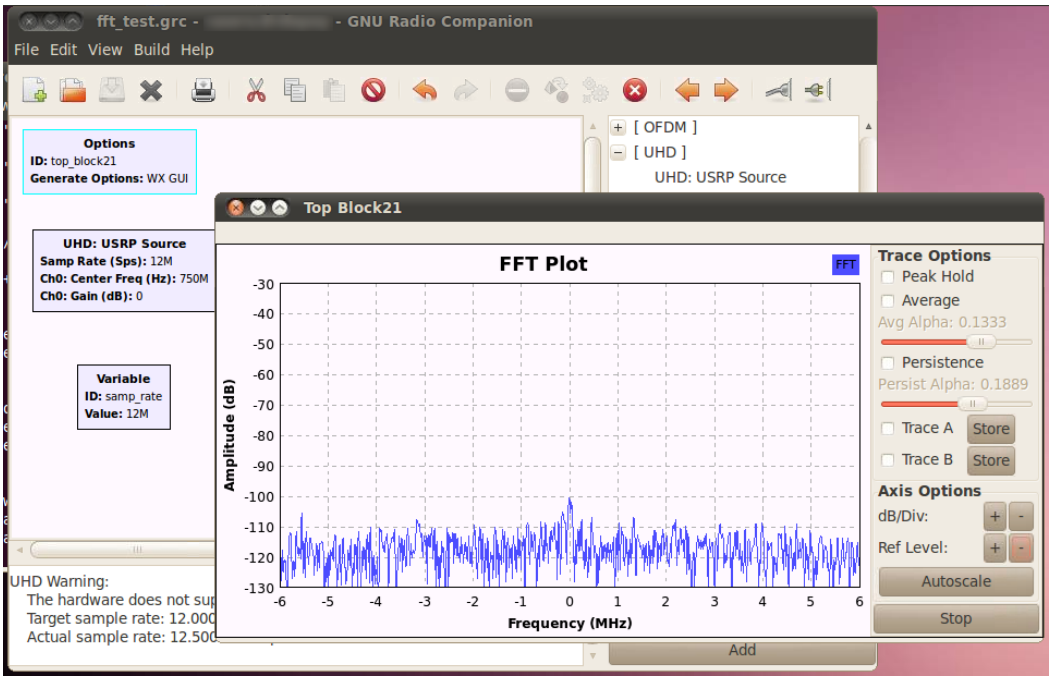

Figure X. Running FFT Waveform

5.4.5 Using OSSIE

Detailed tutorials and examples for using OSSIE as a SDR framework can be found on the official website [27]. CORNET nodes which are enabled to run the UHD firmware on the USRP2 do not have OSSIE enabled by default, due to the fact that the OSSIE source configure scripts require non-UHD drivers to be present on the system before installation can be completed. Users who wish to use OSSIE may request an older node OS image be installed which uses the legacy GnuRadio USRP drivers by default, or they may enable OSSIE manually using the procedure in this section. Currently, there is no UHD device included in

the OSSIE 0.8.2 release used on CORNET, so users who require use of a USRP2 with the UHD firmware installed will need to create their own device, or locate one in the OSSIE repositories.

To enable OSSIE on individual nodes, first log into the node via SSH or NX remote desktop. Navigate to the /tools/ directory and execute the *OSSIEenable* script using *sudo:*

```
cd /tools
sudo ./OSSIEenable
```

The OSSIE enable script will install the older GnuRadio USRP drive libraries required as well as compile and install the OSSIE framework. The installation process requires a significant amount of code to compile and install, and takes about 35 minutes to complete. About halfway through the installation, the script will requires the user to approve the installation of a repository package by answering *Yes* when prompted.

Development and prototyping of OSSIE waveforms is integrated into the Eclipse Software Development Environment (SDE) through the OSSIE Eclipse Feature (OEF). In order to check if the plugins are enabled, start Eclipse from a remote shell with X11 forwarding enabled,

or from an NX session, by typing *eclipse* into a terminal. If the *OSSIE* menu appears at the top of the window, then OEF is already installed and enabled, as in fig. 39.

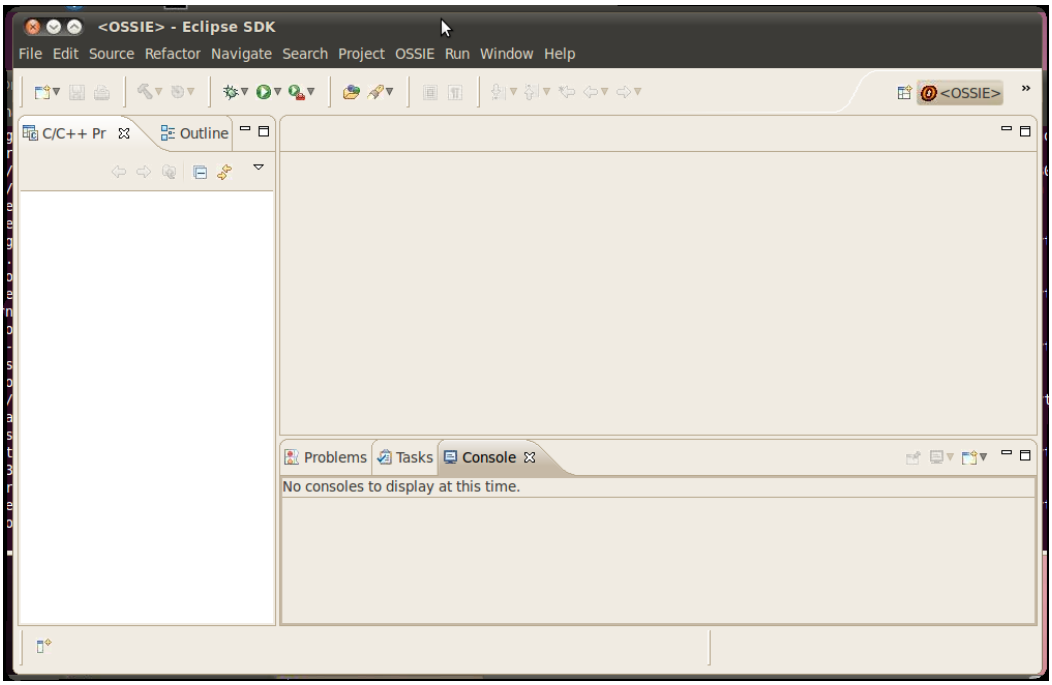

Figure 39. Eclipse With OEF

If the *OSSIE* menu is not present, it can be easily enabled by selecting *Help → Install New Software...* and selecting the *eclipse internal* site from the drop down menu (fig. 40).

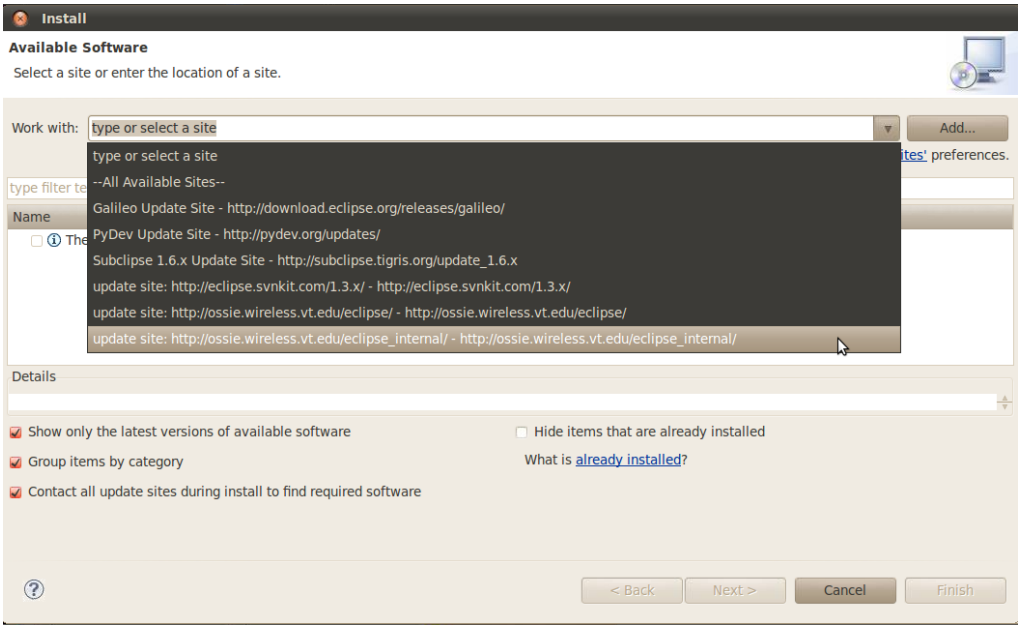

Figure 40. Install OEF Plugins

In the *Name* window, select the *OSSIE* group by checking the selection box, and pressing *Next.* Accept the license agreement, and select *Finish* to complete the installation. Eclipse is now configured for use with OSSIE.

5.5 Installing Custom Software

Users may wish to install custom software of their own creation, or from third parties. The procedures for installing such software on CORNET nodes is the same as it would be locally. As an example, the

following sections provide a short guide for installing LiquidDSP [16] on CORNET nodes.

5.5.1 Dependencies

Most software dependencies can be found in the standard Ubuntu repositories, which can be accessed using *apt-get*. For dependencies which are not present in the repositories, the source code must be built manually. For LiquidDSP, the only dependency is *FFTW* FFT libraries, which can be installed from the repository:

```
sudo apt-get install libfftw3-dev
```

5.5.2 Compiling Source Code

Generally, source code is downloaded from a third party source repository using either *GIT* or *SVN*. Users should create a new folder in their home directory, and clone the repository in this location:

```
cd /users/[user name]
mkdir src
cd /users/[user name]/src
git clone https://github.com/jgaeddert/liquid-dsp.git
```

Most source code includes a *README* or *INSTALL* text file in the root directory which provides instructions to compile and install the software. In this case, the following commands issued from /users/[user name]/liquid-dsp/ are used for installation:

```
./reconf
./configure
make
sudo make install
```

5.5.3 Installing Software on Several Nodes

Users are encouraged to download and build custom software within their home directory, as this makes it easy to install software across multiple nodes without the need to compile the code several times. Users' home directories are persistent across the entire network, so once code has been built, it can be installed on different nodes by ensuring all dependencies are met, and simply issuing the *sudo make install* command from the appropriate source directory, after logging into different nodes. The process can be streamlined even further by issuing *make install* commands directly via SSH, without the need to log into nodes individually:

```
ssh -t [user name]@cornet.wireless.vt.edu -p [port] 'cd                    \
        /users/[user name]/src/liquid-dsp && sudo make install'
```

The *-t* flag creates a virtual TTY interface with the remote machine in order to allow remotely executed commands to prompt the user for input – in this case the user will be prompted for their sudo password. The command to be executed is enclosed by the single apostrophe characters.

5.6 Other CORNET Demonstrations

There are several other example and demonstration waveforms in addition to those presented in the previous sections. These applications are located in the home directories of the users who implemented them, and may provide additional insight into the application development process for CORNET. Two such examples will be briefly described in the following sections.

5.6.1 Multichannel Ad-Hoc DSA Demonstration

In addition to the DSA waveform described in section four, a non-centralized, self-organizing, ad-hoc networking demonstration waveform

was developed as a proof of concept exercise. The waveform is based on the *tunnel.py* GnuRadio example waveform, which binds the USRP2 device to a Linux Ethernet interface, allowing it to be used as a physical layer device with access to the Linux TCP/IP protocol stack (fig. 41). Such functionality is highly useful, as it accommodates the easy creation of local wireless networks between multiple USRP2 nodes.

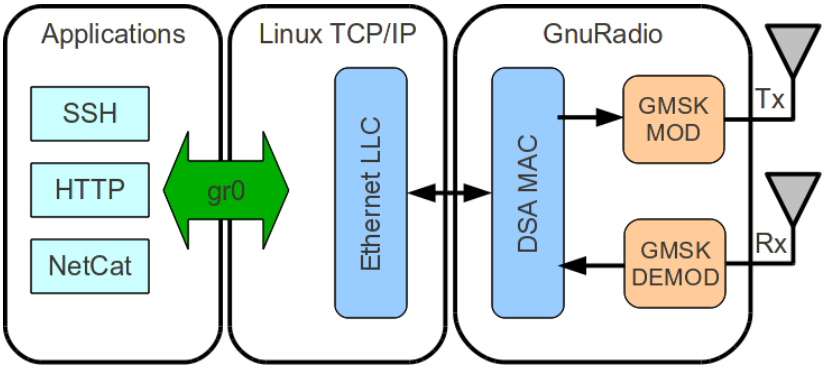

Figure 41. Ad-Hoc DSA Waveform

The demonstration waveform borrows from the *tunnel.py* functionality and modifies it in order to accommodate the creation of a self-organizing, multi-channel, wireless ad-hoc network. The ad-hoc network makes use of cooperative spectrum sensing and frequency agility in order to avoid interference from primary users, and to locate vacant spectrum for transmission. The network attempts to maintain a dedicated control channel which is reserved for node coordination and signaling operations,

and seeks out other vacant channels as needed for peer-to-peer communications. That is, nodes organize themselves on a single control channel, and exchange packets on different frequencies – returning to the control channel when their Ethernet buffers are emptied. The spectrum sensing is cooperative in that peer nodes agree on what channels to use for transmission, and additionally agree on a "next-hop" frequency for the control channel if an evacuation event should occur.

Initial network discovery and coordination is based on the Ethernet LLC and the exchange of ARP messages which maps the Ethernet interface MAC address of peer nodes to the interface IP address in an ARP table. Packet exchange is accomplished through an ARP modification which defines the current occupied channel of peer nodes, which will be called Frequency Resolution Protocol (FRP) here. When a node has data to send to the network, it first consults its ARP table to construct an Ethernet frame header based on the IP address of the destination. The node then checks its FRP table to determine if the destination node is currently listening on the primary control channel, or if it is engaged in peer-to-peer transmission on a different channel. If the destination node is determined to be listening on the control channel, the source node transmits a "Request To Send" (RTS) packet which contains

the frequency of the desired transmission channel that the source node has determined to be clear. If the destination node agrees with the transmission channel, it replies with an acknowledgment. If the destination node disagrees with the transmission channel, it replies with a conditional ACK, and suggests a different channel to be used instead. This exchange continues until the peer nodes agree on a transmission channel, at which point they broadcast FRP update packets on the control channel to alert other nodes that they are leaving. Both nodes switch to the transmission channel, and the destination node transmits a "Clear to Send" (CTS) packet to alert the source node that it has successfully changed channels. The peers then exchange packets until the connection idles long enough for one of the nodes to empty its Ethernet buffer, at which point it transmits a "done" frame on the transmission channel. If the peer node is also finished, it responds with a "done ACK" frame, and both nodes return to the control channel and broadcast FRP presence beacons to inform the network that they have returned. Figure 42 outlines the operation of the DSA MAC and signaling protocols.

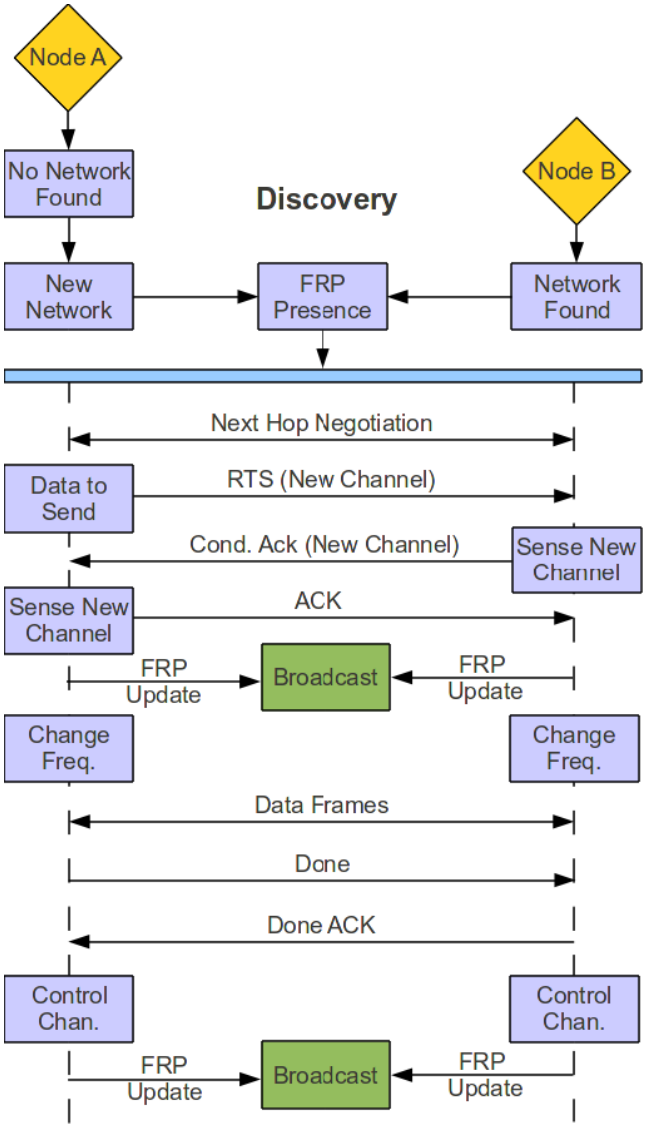

Figure 42. Ad-Hoc DSA Protocol

5.6.2 USRP2 Spectrum Sensing

Another useful example waveform which lends itself to modification is the *usrp_spectrum_sense.py* waveform. One thing that sets this waveform apart from the previous examples is that it does not use the

usrp_transmit_path class instantiation file to configure the GnuRadio flowgraph, and therefore provides a useful example for how to create flow custom flow graphs from discrete components without using GnuRadio Companion. The waveform also demonstrates the utility of the GnuRadio message sink for passing meta-information back into the waveform (i.e. - spectrum measurements.) It should be noted that the waveform included with GnuRadio is written for the USRP1 peripheral, which operates over a USB connection. Modifying the flowgraph to run on a USRP2 is fairly straight forward, and an already modified version (with some additional functionality) can be found at /users/drdepoy/usrp2_spectrum_sense.py. The modified version is intended to work with the legacy GnuRadio drivers, though once again, a port that will work with UHD drivers should be easily realizable by importing the UHD libraries and replacing the USRP2 source with a UHD version.

The basic functionality of the waveform is to perform energy detection across a wider bandwidth than the USRP2 can tune by performing a spectrum sweep, and combining energy information from discrete channels. The flowgraph and message queue is shown in figure 43.

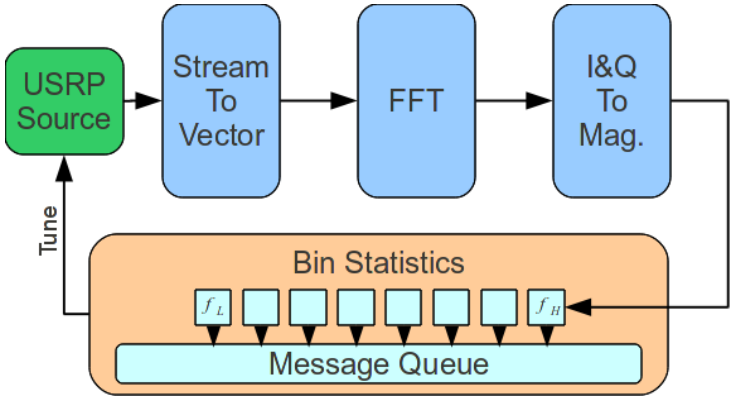

Figure 43. Spectrum Sensing Flowgraph

The message queue provides an interface between the flowgraph and the main loop of the Python code by acting as a data FIFO for the FFT energy measurements. The *bin_statistics* GnuRadio component keeps records (maximum, average or snapshot) of the FFT energy for each channel, and automatically tunes the USRP to the next frequency after each measurement. The main routine of the application can then access the message queue by calling the *parse_msg* function, which places the data vector into a local variable.

6

Conclusions and Future Work

Cognitive Radio promises to usher in a new era of possibilities for wireless communications, and will revolutionize the way we share spectrum in the very near future. VT-CORNET is a unique testbed concept which has been tailored to accommodate research and education related to CR. Our state-of-the-art facilities include highly flexible SDR radio interfaces, coupled to a powerful cluster of GPP devices and management network which provides remote access capabilities. Our radio nodes are distributed throughout a campus building, allowing researchers diverse channel conditions, deployment options and access to live spectrum with real primary users. VT-CORNET is specially licensed by the FCC to operate over a broad slice of spectrum, from 138Mhz to 3.6GHz. Our network additionally provides users with several remote access methods, as well as software tools, examples and tutorials intended to accommodate a wide array of user abilities and tasks.

Future additions to VT-CORNET will include updated RF daughterboards (using the Motorola RFIC4 chip), mobile and outdoor nodes, as well as newer USRP platforms (i.e., USRP Embedded line). Additional testbeds in the same vein as VT-CORNET are being planned for new buildings at the Blacksburg campus, and for our Capital Region campus as well. Between the current testbed and our planned expansions, VT-CORNET provides researchers with a novel platform for advancing research of CR networks.

References

[1] Alyaoui, N.; Kachouri, A.; Samet, M.; , "The fourth generation 3GPP LTE identification for cognitive radio," *Microelectronics (ICM), 2011 International Conference on* , vol., no., pp.1-5, 19-22 Dec. 2011

[2] Bianchi, G.; Fratta, L.; Oliveri, M.; , "Performance evaluation and enhancement of the CSMA/CA MAC protocol for 802.11 wireless LANs," *Personal, Indoor and Mobile Radio Communications, 1996. PIMRC'96., Seventh IEEE International Symposium on* , vol.2, no.,pp.392-396 vol.2, 15-18 Oct 1996

[3] Borth, D.; Ekl, R.; Oberlies, B.; Overby, S.; , "Considerations for Successful Cognitive Radio Systems in US TV White Space," *New Frontiers in Dynamic Spectrum Access Networks, 2008. DySPAN 2008. 3rd IEEE Symposium on*, vol., no., pp.1-5, 14-17 Oct. 2008

[4] Canonical, “IPTables Howto” (https://help.ubuntu.com/community/IptablesHowTo)

[5] Canonical, “OpenLDAP Server” (https://help.ubuntu.com/11.04/serverguide/C/openldap-server.html)

[6] Debian.org, “Installing with the Debian Installer” (http://www.debian.org/devel/debian-installer/)

[7] Ettus Research “WBX 50-2200 MHz Rx/Tx” https://www.ettus.com/product/details/WBX

[8] FCC License #: 0085-EX-PL-2011

[9] FreeBSD.org, “chmod” (http://www.freebsd.org/cgi/man.cgi?query=chmod&sektion=1)

[10] GnuRadio.org, “USRP2 FAQ” http://gnuradio.org/redmine/projects/gnuradio/wiki/USRP2

[11] GNU.org, “Free IPMI” (http://www.gnu.org/software/freeipmi/)

[12] Gstreamer, “Gstreamer Documentation” (http://gstreamer.freedesktop.org/documentation/)

[13] Gstreamer, “Gstreamer Ugly Plugins Reference Manual” (http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-ugly-plugins/html/gst-plugins-ugly-plugins-x264enc.html)

[14] Intel, “Intelligent Platform Management Interface” (http://www.intel.com/design/servers/ipmi/)

[15] ISO/IEC 13818-1:2000 - “Generic coding of moving pictures and associated audio information: Systems”

[16] Joe Gaeddert, “Liquid DSP” (https://github.com/jgaeddert/liquid-dsp)

[17] Maximilian Ott, Ivan Seskar, Robert Siracusa, Manpreet Singh, "ORBIT Testbed Software Architecture: Supporting Experiments as a Service", *Proceedings of IEEE Tridentcom 2005*, Trento, Italy, Feb 2005

[18] Mitola, J., III; Maguire, G.Q., Jr.; , "Cognitive radio: making software radios more personal," *Personal Communications, IEEE* , vol.6, no.4, pp.13-18, Aug 1999

[19] Natalizio, E.; Loscrí, V.; Aloi, G.; Paolí, N.; Barbaro, N.; , "The practical experience of implementing a GSM BTS through open software/hardware," *Applied Sciences in Biomedical and Communication Technologies (ISABEL), 2010 3rd International Symposium on* , vol., no., pp.1-5, 7-10 Nov. 2010

[20] Newman, T.R.; Hasan, S.M.S.; Depoy, D.; Bose, T.; Reed, J.H.; , "Designing and deploying a building-wide cognitive radio network testbed," *Communications Magazine, IEEE* , vol.48, no.9, pp.106-112, Sept. 2010

[21] NoMachine, (http://www.nomachine.com/)

[22] NoMachine, "Downloads" (http://www.nomachine.com/download.php)

[23] PuTTY, "PuTTY: A Free Telnet/SSH Client" (http://www.chiark.greenend.org.uk/~sgtatham/putty/)

[24] SYSLINUX, "The SYSLINUX Project" (http://www.syslinux.org/wiki/index.php/The_Syslinux_Project)

[25] UC Riverside, "UC Riverside Wireless Networking Research Testbed" (http://networks.cs.ucr.edu/testbed/index.htm)

[26] University of Utah Flux Research Group, "Emulab: The Utah Network Testbed," (http://www.emulab.net/)

[27] Virginia Tech, "Open Source SCA Implementation: Embedded" (http://ossie.wireless.vt.edu/)

[28] Wei Zhang; Mallik, R.; Letaief, K.; , "Optimization of cooperative spectrum sensing with energy detection in cognitive radio networks," *Wireless Communications, IEEE Transactions on* , vol.8, no.12, pp.5761-5766, December 2009

Appendix A

IP and Gateway Port Reference

Node #	Floor_Position	Internal IP	External Port
1	1_1	192.168.1.11	7001
2	1_2	192.168.1.12	7002
3	1_3	192.168.1.13	7003
4	1_4	192.168.1.14	7004
5	1_5	192.168.1.15	7005
6	1_6	192.168.1.16	7006
7	1_7	192.168.1.17	7007
8	1_8	192.168.1.18	7008
9	1_9	192.168.1.19	7009
10	1_10	192.168.1.20	7010
11	1_11	192.168.1.21	7011
12	1_12	192.168.1.22	7012
13	2_1	192.168.1.23	7013
14	2_2	192.168.1.24	7014
15	2_3	192.168.1.25	7015
16	2_4	192.168.1.26	7016
17	2_5	192.168.1.27	7017
18	2_6	192.168.1.28	7018
19	2_7	192.168.1.29	7019
20	2_8	192.168.1.30	7020
21	2_9	192.168.1.31	7021
22	2_10	192.168.1.32	7022
23	2_11	192.168.1.33	7023

24	2_12	192.168.1.34	7024
25	3_1	192.168.1.35	7025
26	3_2	192.168.1.36	7026
27	3_3	192.168.1.37	7027
28	3_4	192.168.1.38	7028
29	3_5	192.168.1.39	7029
30	3_6	192.168.1.40	7030
31	3_7	192.168.1.41	7031
32	3_8	192.168.1.42	7032
33	3_9	192.168.1.43	7033
34	3_10	192.168.1.44	7034
35	3_11	192.168.1.45	7035
36	3_12	192.168.1.46	7036
37	4_1	192.168.1.47	7037
38	4_2	192.168.1.48	7038
39	4_3	192.168.1.49	7039
40	4_4	192.168.1.50	7040
41	4_5	192.168.1.51	7041
42	4_6	192.168.1.52	7042
43	4_7	192.168.1.53	7043
44	4_8	192.168.1.54	7044
45	4_9	192.168.1.55	7045
46	4_10	192.168.1.56	7046
47	4_11	192.168.1.57	7047
48	4_12	192.168.1.58	7048
Image/PXE	NA	192.168.1.2	NA
Gateway	NA	192.168.1.100	NA
LDAP	NA	192.168.1.101	NA
NFS	NA	192.168.1.102	NA

Web/Apache	NA	192.168.1.103	NA