Design, Deployment and Performance of an Open Source Spectrum Access System

Shem Kikamaze

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

Carl B. Dietrich, Chair

Jeffrey H. Reed

Harpreet S. Dhillon

September 26, 2018

Blacksburg, Virginia

Keywords: Spectrum Access System, Spectrum Management, Cognitive Radios, Software Defined Radios, Testbed

Copyright 2018, Shem Kikamaze

Design, Deployment and Performance of an Open Source Spectrum Access System

Shem Kikamaze

(ABSTRACT)

Spectrum sharing is possible, but lacks R&D support for practical solutions that satisfy both the incumbent and secondary or opportunistic users. The author found a lack of an openly available framework supporting experimental research on the performance of a Spectrum Access System (SAS) and propose to build an open-source Software Defined Radio (SDR) based framework. This framework will test different dynamic spectrum scenarios in a wireless testbed. This thesis presents our Spectrum Access System prototype, discusses the design choices and trade-offs and provides a proof of concept implementation. We show that an Internet-accessible CORNET testbed provides the ideal platform for developing and testing the SAS functionality and its building blocks and offerss the hardware and software as a community resource for research and education. This design provides the necessary interfaces for researchers to develop and test their SAS-related modules, waveforms and scenarios.

Design, Deployment and Performance of an Open Source Spectrum Access System

Shem Kikamaze

(GENERAL AUDIENCE ABSTRACT)

In this information age, the number of wireless devices is growing faster than the infrastructure required to make wireless communication possible. This creates a possibility of not having enough radio spectrum to keep up with this growing demand. To alleviate this issue, there is a need to research and find more ways of efficiently utilizing the current spectrum resources available. Dynamic spectrum allocation is one way forward to archiving this goal. Frequency channels are assigned to devices based on prevailing conditions like device location and availability of channels that would cause low interference to other devices. Spectrum utilization is based on time, frequency and space with devices having the ability to hop to the best channel available. In this thesis, an open source Spectrum Access System (SAS) was created as a platform through which dynamic spectrum allocation research can be done. The SAS is centralized management system that logs information about the prevailing spectrum usage, and in turn uses this information to dynamically allocate spectrum to devices and networks. This thesis shows how it was implemented, its current performance, and the steps that different researchers can take to add their own functionalities.

Dedication

…to my Maama, Nabaweesi Kizza

Acknowledgments

I would like to send my heartfelt gratitude to my advisor, Dr. Dietrich for his guidance and help through this research. Our shared vision, of creating platforms that enable further research, was a great motivation in accomplishing this thesis. I am also thankful for the any hardware equipment that I received. I had more than enough USRPs, computers, networking equipment to utilize for this project.

I would like to thank my co-advisors, Dr. Reed and Dr. Dhillon for providing me with ideas on the directions of my thesis.

I also would like to thank Wireless@VT for providing a great place through which I could learn as much as I could. There were many projects available other than my research that I enjoyed working on. Special thanks to Hilda Reynolds for perks (March Madness, food); they helped to keep the spirits up.

Randall Nealy was of huge help with everything hardware. He helped me a lot with utilizing lab equipment, and with constructing the final testbed that I used.

Special thanks to Xavier Gomez for keeping the testbed up and running, and finding new ways to make it easily accessible.

Not forgetting Dr. Vuk Marojevic, who found ways for getting new equipment for me.

I would also like to acknowledge and thank Ahmad Jauhar, who built the sensing mechanism for this project. Sensing was a core component for this project, and I would not have accomplished anything without it.

I will be forever grateful for this experience.

Contents

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 Background 1

1.2 Motivation 3

1.3 Policy Review 3

1.3.1 SAS-CBSD Communication 4

1.4 Channel Allocation 5

1.5 Localization 6

2 Spectrum Access System 9

2.1 Hardware 9

2.2 Software 11

2.2.1 Inter-Process Communication (IPC) 12

2.2.2 Sensor 13

2.2.3 Spectrum Access System 15

2.2.4 CBSD 20

2.2.5 PU 22

2.2.6 Peripheral Software 22

3 Experimental Implementation 27

3.1 Hardware Setup 27

3.1.1 Host Computing 28

3.2 RFNEST Configuration 32

3.2.1 Dependency Installation 32

3.2.2 Setting Up 34

3.2.3 Running 36

3.3 Installation 36

3.3.1 Background Services 37

3.4 Using the SAS 38

3.4.1 Logging In 38

3.4.2 Starting the SAS 38

3.5 Cognitive Engine Implementation 46

3.6 Data Logging and Visualization 49

4 Results 51

4.1 PU Detection 51

4.1.1 Localization 52

4.1.2 Detection Latency 55
4.2 SU Performance 56
4.3 Real Time Sensor Performance 57

5 Conclusion 60

6 Future Work 61

Bibliography 64

Appendices 68

Appendix A Radio Environment Map 69
A.1 Sensor Information 70
A.2 CBSD Information 70

Appendix B SAS-CBSD Interface 72

Appendix C SAS Engine 75

Appendix D Software Interconnectivity 76

List of Figures

1.1 Request Reply Response Example between SAS and CBSD 5

1.2 Trilateration 6

2.1 Physical Connection to the RFNEST 10

2.2 Modular Architecture of the whole system 12

2.3 GNURadio Flow Graph for controlling the USRP 13

2.4 GNURadio Flow Graph for Processing Sensor Information 15

2.5 SAS Server Side Processes 19

2.6 Network Stack for the SU/CBSD 22

2.7 GPS Software Diagram 24

2.8 Graphical Interface to recreate real environmental models 25

2.9 Architecture for interconnection of programs controlling the RFNEST 26

3.1 Emulated Back-haul Network Architecture Implemented 28

3.2 Emulated Virtual Network that connects to the external NIC 29

3.3 Virtual Network for USRPs 30

3.4 External Switch 31

3.5 USRP detected with internal routing 32

3.6 Created Virtual Interfaces 32

3.7 Configuring the whole experiment 39

3.8 Rudimentary PU detection with a single sensor 48

3.9 Trilateration Detection Using Three Sensors 49

4.1 Histogram Showing the Error in Possible Distance between the Sensor 1 and the PU 52

4.2 Histogram Showing the Error in Possible Distance between the Sensor 2 and the PU 53

4.3 Histogram Showing the Error in Possible Distance between the Sensor 3 and the PU 53

4.4 Histogram Showing the Error in Possible Distance after Trilateration 54

4.5 Histogram Showing the Error in Possible Distance after Trilateration after Moving Average 55

4.6 Histogram Showing Latency of Detection for a single sensor 56

4.7 Histogram Showing Latency of Detection with Localization 57

4.8 Real Time SU Performance Display 58

4.9 Real Time Sensor Display 59

6.1 Jmeter Application 62

6.2 Dual CBSD and Sensor 63

A.1 Entity Relationship Diagram of the REM 69

B.1 Request-Response Communication Sequence 73
B.2 Class Diagram for the SAS-CBSD Interface 74
C.1 Class Diagram for the SAS Engine 75
D.1 Current Software Interconnection 76

List of Tables

2.1 Interconnected Modules - Sensor 15

2.2 Interconnected Modules - Centralized Sensor Aggregator 16

2.3 Interconnected Modules - Radio Environment Map 18

2.4 Interconnected Modules - SAS-CBSD Interface 18

2.5 Interconnected Modules - SAS Cognitive Engine 20

4.1 Average Latency after 50 PU transmissions in seconds 56

Chapter 1

Introduction

Radio frequency (RF) spectrum regulation for commercial wireless communications is shifting from strictly regulated spectrum access through long-term licensing to a more dynamic spectrum use based on shorter-term spectrum availability, rules and priorities. This is because, in terms of efficiently utilizing the current frequency bands, the current static structure needs improvement because of instances when the designated incumbent user is dormant. In some instances, there are geographical areas, under the same static license, where there is poor communication quality with the incumbent. In such instances, licensed and unlicensed secondary should be able to utilize the frequency band, while creating little to no effect to the incumbent and ensure harmonious spectrum coexistence. To implement dynamic spectrum access, The United States Federal Communications Commission (FCC) proposed commercial utilization of the 3.5 GHz band, which is the initial test band for this kind of spectrum sharing [7].

1.1 Background

The FCC regulations require the use of geographically distributed central management systems called Spectrum Access Systems (SAS) that allocate the frequency channels between 3550 MHz and 3700 MHz to non-incumbent users. The SAS acts as a spectrum guardian and allocates channels to radios that do not own a long-term exclusive license. The SAS is a

term that was introduced to manage the newly-available 3.5 GHz US spectrum, but can be applied to other shared bands. When the incumbent is not transmitting in a given geographical area, secondary users can fill the spectrum hole. The commercial use of the US 3.5 GHz band, which was traditionally used by government radios, mostly Naval radars, is termed as the Citizens Broadband Radio Service and the radios are called Citizens Broadband Radio Service Devices (CBSDs). In the US, CBSDs are further grouped into two: those that have Priority Access Licenses (PALs) to access the channels and those that have General Authorized Access (GAA) rights. PALs are provided by the FCC through competitive bidding. This is a three-tier spectrum sharing system with the incumbent users having the highest priority, followed by the PAL users.

The 3.5 GHz SAS's main job is to allocate spectrum dynamically to the CBSDs while enforcing a strict protection of the incumbent from interference by other users. In locations, parts of the band, and time intervals that are not used by or reserved for use by an incumbent, the PAL users will be protected from the GAA users. To achieve this, the SAS continuously monitors the entire band. All information about the spectrum can be stored in a database that builds a Radio Environment Map (REM) [29]. The spectrum information stored in the REM is based on the location and time at which it was obtained, as well as the radio access technologies and policies of the radios. The REM can also store information about CBSDs and how often each of the channels is being requested and utilized by the different CBSDs. The REM database can also contain public information about the incumbents. This information can be obtained from available public federal information and support effective spectrum allocation to secondary users. Some of the information in the REM database is shared from one SAS to another through inter-SAS communication. This adds extra redundancy that would improve on the aggregate interference protection of the primary users. This information may include PAL area information, exclusion zones, and so forth.

1.2 Motivation

Due to the lack of an openly available framework for experimental research on the performance of a SAS, we propose to build an open-source SDR-based framework that can be used to test different dynamic spectrum allocation scenarios, algorithms, and parameter choices. With these tests, we would be able to derive different ways to improve on the dynamic spectrum allocation in this frequency band. Our SAS framework uses the wireless testbed, CORNET[24]. Existing SAS testbeds include one implemented by Bell Labs [14], which uses LTE base stations (eNodeBs) and WIFI Access Points with already established hardware-specific MAC protocols. Another testbed was built to evaluate the use of the 3.5 GHz band for next generation LTE-based public safety systems [23]. Our testbed significantly differs from these approaches in several ways, including: (1) exclusive use of software-defined radio in the implementation, (2) use of open-source software across the entire protocol stack, and (3) remote accessibility. The Wireless Innovation Forum is developing a communication protocol between the SAS and the CBSDs[27]. This is the communication protocol that we use as the basis of our framework and testing.

1.3 Policy Review

This thesis introduces the reader to the current status of the Open-Source SAS that has been created by the author, and the design choices made to accomplish the task. Each component of the SAS is explained in great detail to provide the reader with an insight into how the component interacts with other components to make the whole system work. This is important because of unavailability of any other Open-Source SAS. The reader is able to critique the design choices made by the author, and because of the Open-Source nature of

the project, the reader has the option to make changes to improve on the SAS, or gain an insight on adding more components to augment the existing SAS.
As mentioned in chapter 1.1, the FCC was able to approve Spectrum Access Administrators[16] who formed a Spectrum Sharing Committee at WinnForum to provide a basis of the architecture of the SAS, CBSD and Environment Sensing Capability (ESC). The committee is also responsible for the communication protocol between the different entities mentioned.
As standardization of the policy is still in progress, the SAS implementation, for this thesis, tries to follow the current documents of the policy. This section is going to summarize the current policy that is being implemented.

1.3.1 SAS-CBSD Communication

The SAS-CBSD communication is implemented in the application layer (OSI model) of the both the SAS and CBSD. Such communication implies that the current policy does not intend it to be faster than current link layer protocols, but rather a well defined specification that uses underlying lower layer protocols.
The current communication policy relies on the well-established Hypertext Transfer Protocol(HTTP)[9] as the application layer protocol with a presumption of a reliable TCP transport layer. HTTP maintains a persistent connection between the CBSD and SAS to prevent any overhead of re-establishing a connection for every request that is being sent.
With HTTP, encapsulated messages between the SAS and the CBSDs are formatted as human-readable text in form of JavaScript Object Notation (JSON)[6].

With the established HTTP, the WinnForum added a Transport Layer Security for a secure HTTPS connection. Each CBSD and SAS will have a security certificate which authorizes communication between known CBSDs and the SAS. For linux machines, this is easily

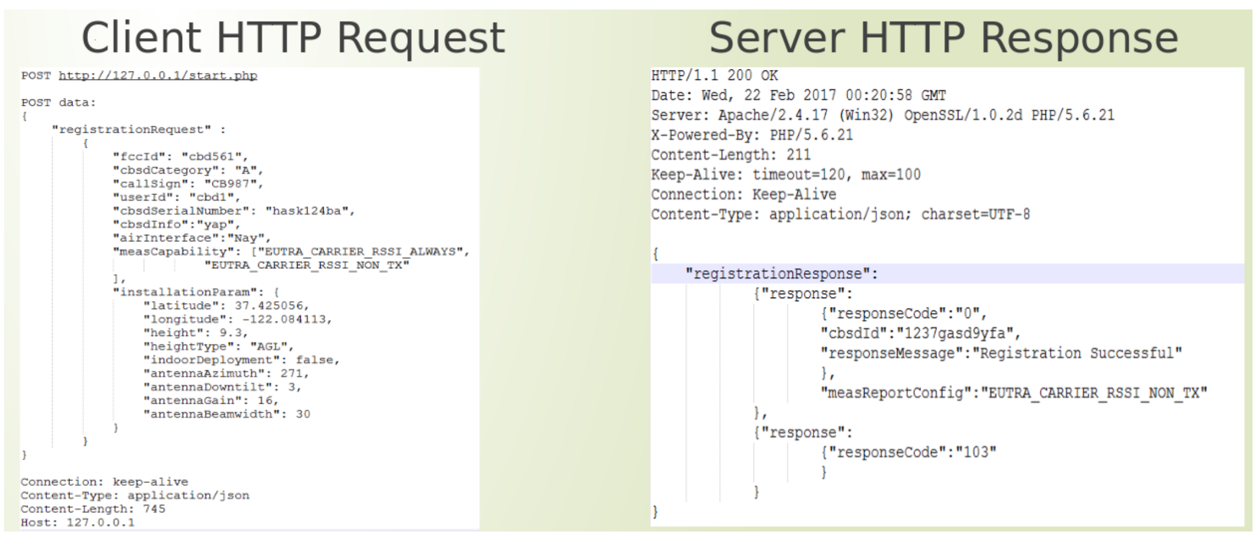


Figure 1.1: Request Reply Response Example between SAS and CBSD

implemented with the OpenSSL[26], which is an free for commercial and non-commercial usage.

1.4 Channel Allocation

The SAS does not provide channel allocation for the incumbent or PU, but rather the rest of the CBSDs in the tier system. The main motivation behind this thesis, was not to create algorithms for channel allocation, but to provide a platform for testing the various theoretical algorithms. There are a variety of theories behind channel allocation and most of them have been simulated. However, few have been able to implement these algorithms on practical dynamic spectrum allocation systems [19].

Sashika [20] utilizes multi-linear programming to find an optimized solution with channel allocation based on spatial distances between CBSDs. This solution is useful in situations where multiple CBSDs are provided with the same channel but with distances far enough to reduce interference. An initial phase of this algorithm is implemented, without taking into

account spatial distances between CBSDs, but rather spatial distances between CBSDs and the PU.

No multi-linear programming is implemented yet, but rather a reactive based approach since the location and transmission time of the PU is unknown to the SAS.

1.5 Localization

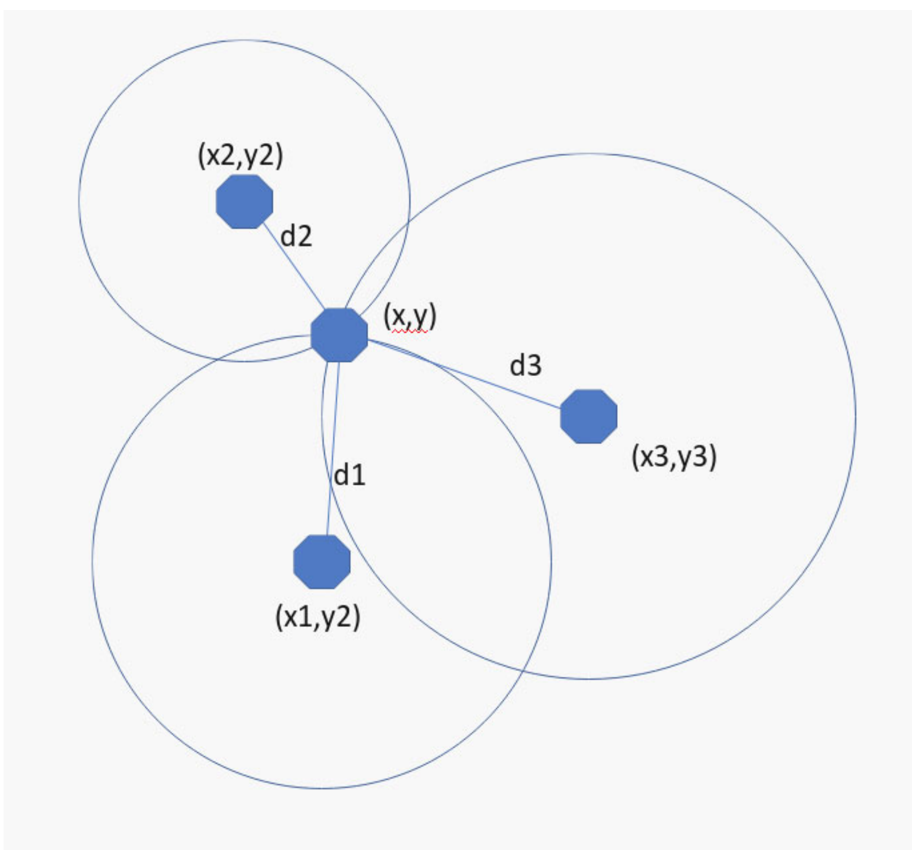


Figure 1.2: Trilateration

To be able to utilize spatial distance for channel allocation, each node in the whole environment has to provide its GPS location. Unfortunately, the PU may not provide its location to the SAS, and therefore, sensors are used to locate the distance between the PU and the

sensor. Since the distance is estimated, the GPS coordinates of the PU can be calculated from:

$$(x_1 - x)^2 + (y_1 - y)^2 = d_1 \tag{1.1}$$

This is expanded upon to obtain a matrix for all sensors involved:

$$\begin{bmatrix} (x_1 - x)^2 & (y_1 - y)^2 \\ (x_2 - x)^2 & (y_2 - y)^2 \\ (x_3 - x)^2 & (y_3 - y)^2 \end{bmatrix} = \begin{bmatrix} d_1^2 \\ d_2^2 \\ d_3^2 \end{bmatrix} \tag{1.2}$$

This is then represented in the form $Ax = B$:

$$A = \begin{bmatrix} 2(x_3 - x_1) & 2(y_3 - y_1) \\ 2(x_3 - x_2) & 2(y_3 - y_2) \end{bmatrix} \text{ and } B = \begin{bmatrix} d_1^2 - d_3^2 - x_1^2 - y_1^2 + x_3^2 + y_3^2 \\ d_2^2 - d_3^2 - x_2^2 - y_2^2 + x_3^2 + y_3^2 \end{bmatrix} \tag{1.3}$$

Furthermore, the x and y coordinates might not fall on the intersection of three circles, and the matrix equation might not have any solution. The position of the PU is then estimated using $(A^T A)^{-1} A^T B$ Least Squares Method.

Equations 1.1 utilize Cartesian coordinates. GPS coordinates are converted to Cartesian using the following equations:

$$x = R \cdot \cos(latitude) \cdot \cos(longitude)$$

$$y = R \cdot \cos(latitude) \cdot \sin(longitude)$$

$$z = R \cdot \sin(latitude)$$

$$\text{where } R = 6{:}3781 \times 10^6 m \text{ which is the Radius of theEarth}$$

This converted coordinates are utilized by the matrix equations and the final estimated

cartesian coordinates are converted back using:

$$\begin{aligned} \text{longitude} &= \text{atan2}(y; x) \\ \text{latitude} &= \text{acos}(x; R \,\square\, \cos(\text{longitude})) \end{aligned}$$

Chapter 2

Spectrum Access System

2.1 Hardware

Virginia Tech has a testbed of 48 host computers with Universal Software Radio Peripherals (USRP) called Cognitive Radio Network [13]. This networked 48 SDRs/ nodes (host computers and USRPs) are the hardware platform on which the SAS was built for initial testing and verification. Furthermore, the flexibility of waveforms created by the SDRs and the open-source nature of these Linux host machines has been provided a good software platform for this research. All software needed to run the experiments can be installed on all the nodes, and due to their inherent flexibility, each node can designated as either the SAS, PU or SU. This will be further explained in the software section 2.2.An added advantage of utilizing the CORNET testbed is the reliable IP connection between the nodes. This ensures that all SUs have constant and reliable communication with the SAS. Initially the experimentation was done with the CORNET testbed and it was confirmed that the SAS was working [13]. However,to archive more consistent and reliable results, the CORNET testbed was enhanced by adding on additional hardware to accompany the 48 initial nodes. As mentioned in Chapter 1, the SAS has to dynamically allocate spectrum based on space, time and frequency. The testbed posed a challenge with regards to space, since nodes were enclosed in a single building.Furthermore, each part of the building had a completely random environment, which made results intractable.

To enhance the testbed, a channel emulator called RFNEST[28] was added to the testbed. The channel emulator creates a controllable and repeatable radio environment with knobs to add various channel scenarios including but not limited to Doppler spread, Delay spread, antenna orientation and antenna gain. Furthermore, different radio propagation channel models can used to emulate urban and suburban environments. For experimentation, the Free-space Path Loss model was utilized.

The RFNEST A210 has a total of 8 TX/RX ports to which USRPs are connected. Because of the software advantage of adding antenna gain in the Digital Signal Processing of the RFNEST, connected USRPs power is highly attenuated at the ports to prevent over-the-air leakage across ports, but then digitally boosted by increasing the software antenna gain of the RFNEST.

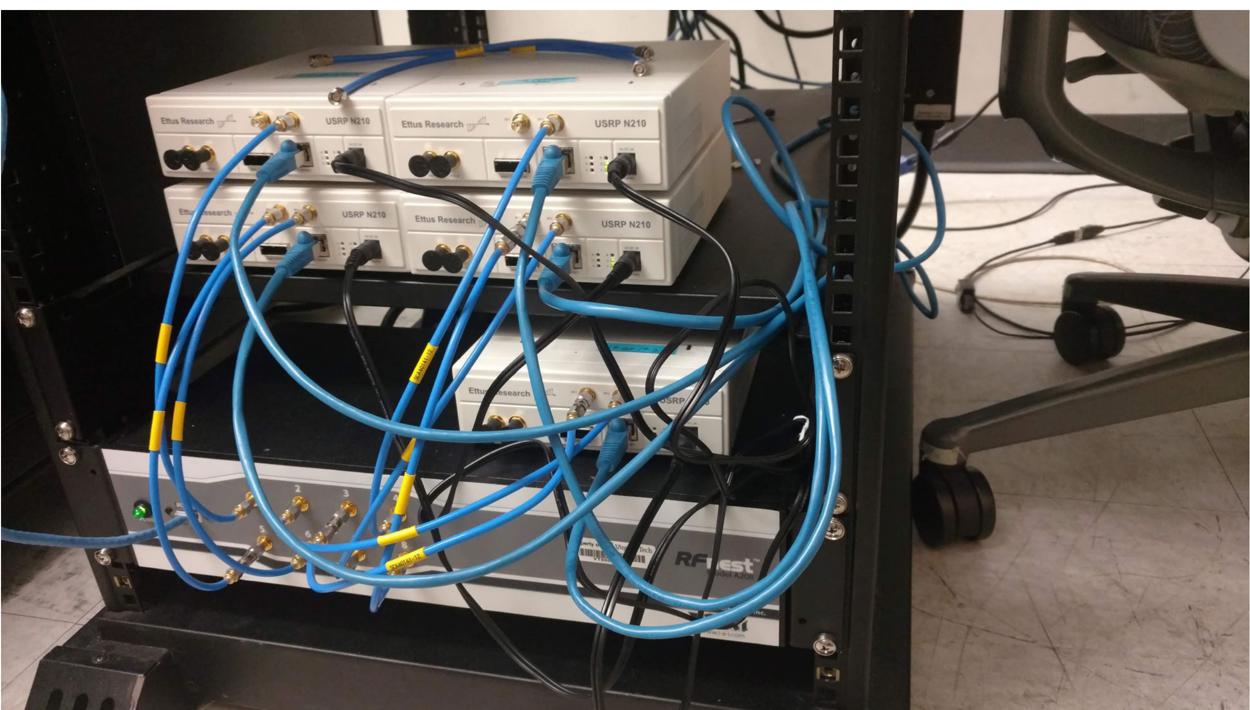


Figure 2.1: Physical Connection to the RFNEST

Preliminary measurement of output power from the USRPs by a Spectrum Analyzer provided

a power output of around -10 dBm/2MHz with no gain. This is within the range specified for the RFNEST input of < 10 dBm [12], since the widest bandwidth to be used is 10 MHz. But in case of experiments where a user accidentally increases the output power of USRPs by 30 dB, the 20 dB attenuator attached to the RFNEST ports would reduce the power to an acceptable range.

A dedicated host machine is designated to control the RFNEST with the connection between the RFNEST and the machine being an 100BASE-T Ethernet. Any other Ethernet type does not work due to the inherent I/O configuration of the RFNEST Ethernet port.

2.2 Software

A modular approach is utilized to design the software behind the functionality of the whole system. In this modular design, a software module is designed to execute a few manageable functions with various interprocess communication mechanisms utilized to share data from one module to another. Breaking down the whole system into small manageable features enabled the use of different programming languages for each module. This ensured that the programmer used a programming language that was more efficient for a specific functionality. For example, all functionalities that deal with the physical layer are written in C++, which is more efficient with memory management, while Python is used in application OSI layers for most of the modules due to its readability and its extensible collection of linear algebra modules. Furthermore, the modular approach provides an easy way of isolating an issue that might arise from the system due to corner and edge cases [32].

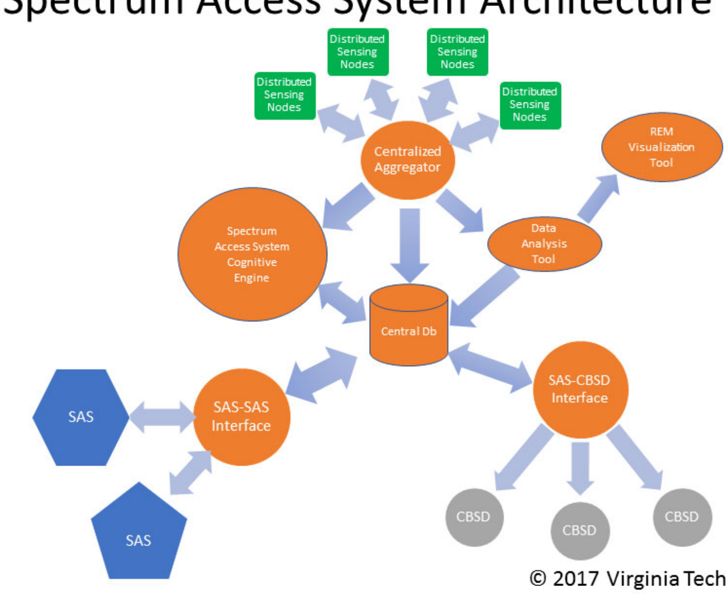


Figure 2.2: Modular Architecture of the whole system

2.2.1 Inter-Process Communication (IPC)

Various IPC tools are utilized to share data between the different modules (processes and threads). For modules that are in the same host, shared memory or message queues [5] are used, while processes that share data between different host machines utilize network sockets. For instances in which different modules are written with different programming languages, sockets are used as the IPC due to the well established socket library in most programming languages, albeit that the socket IPC is slower than other IPCs[5]. Network sockets also provide bidirectional communication, which is important for processes that provide feedback. For every module that is mentioned the various IPCs used will be noted in a table.

2.2.2 Sensor

Current Sensor Software is written in Python and C++ for GNURadio [21]. Sensor software is split into two separate processes that share data through a shared memory buffer. One process is used to fetch In-phase and Quadrature (IQ) signal data from the USRP while the other process does the actual processing of the IQ data.

- UHD Control Flow Graph: This is the flow graph that interfaces with the USRP. Using the GNURadio inbuilt UHD module, IQ data from a specific center frequency and bandwidth is read from the USRP. These data are transferred via 1 Gbps Ethernet from the USRP to the Host machine with GNURadio. Variables needed to control the USRP are its sampling rate, center frequency and the control of the Receiver Gain. To

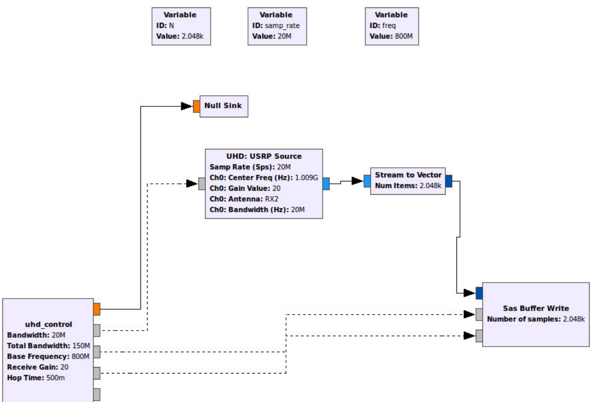


Figure 2.3: GNURadio Flow Graph for controlling the USRP

  provide an option for frequency hopping, such that sensors can cover a wider frequency range, the uhd_ control block was created. The total bandwidth to be covered through frequency hopping is specified plus the amount of time per hop. In fig 2.3, the Total Bandwidth is specified as 150MHz plus a hopping time of 500ms.

  Narrow Band Frequency hopping was employed rather than a fixed full bandwidth approach, to provide an option for (1) SDRs whose hardware is limited to Narrow

bandwidth, (2) Low bitrate Ethernet or USB connections between the SDR and host machine, (3) Lower speed computing resources needed to process the incoming signal.

- Signal Processing Flow Graph: This is the flow graph that processes the IQ data that has been inserted into the shared memory buffer. The PSD block calculates power in each frequency bin based on the given FFT length . For the experiment, an FFT length of 4096 was used. The power in each frequency bin is then transferred to the Energy Detector block for channelization. With channelization, the average power is calculated for a given number of bins. For example:

FFT Length = 4096

Number of subchannels = 10

Number of bins whose power is to be averaged = 4096/10 = 409

The prevailing noise floor is then subtracted from the average power in a specific sub-channel to create a metric termed as the occupancy metric. In this case, the occupancy metric is percentage of power above the prevailing noise floor. This is then stored in the local PSQL database which every sensor has.Additional information is then sent to the Centralized Sensor Aggregator for further processing and storage in the Centralized REM.

$$occ = (AveragePower - NoiseFloor)/NoiseFloor \tag{2.1}$$

As can be shown in Fig. 2.4, the sensor also has a block for reading gps data from a gps device that is connected to that sensor. Sensors periodically send gps information to the SAS.

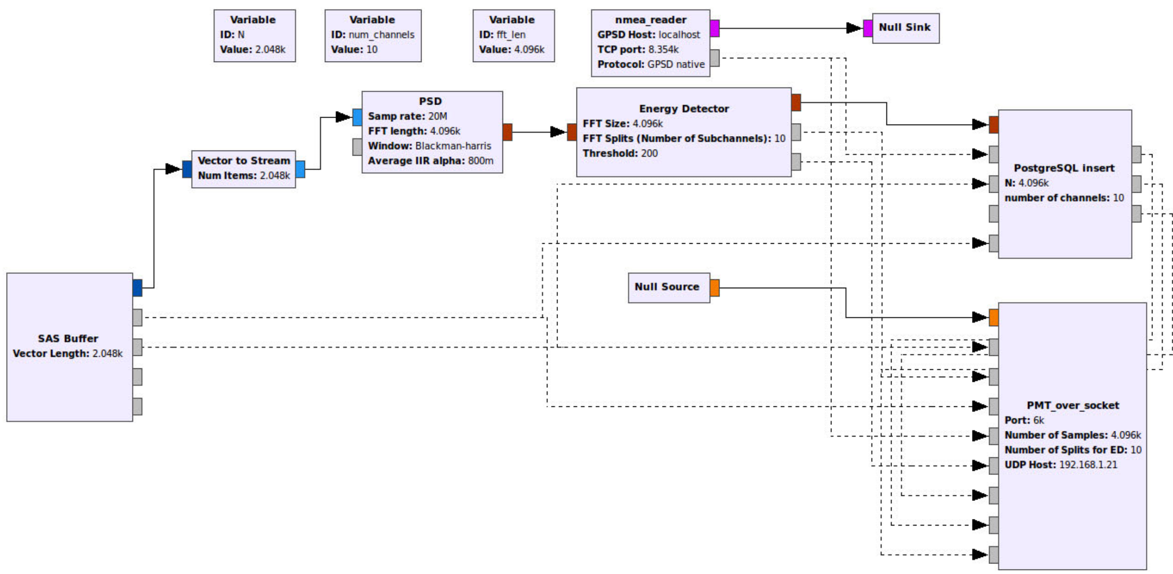


Figure 2.4: GNURadio Flow Graph for Processing Sensor Information

Table 2.1: Interconnected Modules - Sensor

Module	IPC	Additional Information
Centralised Sensor Aggregator	Network Sockets	UDP information received. Information serially packaged in GNURadio PMT(Polymorphic Data Type) [21]

2.2.3 Spectrum Access System

The SAS, whose main purpose is to dynamically allocate spectrum to SUs, is comprised of various modules which run as individual processes on a single host computer. In reality, the modules are designed to run on different computers in instances in which one computer does not have enough computing resources to efficiently process incoming information from many sensors and SUs. Greater detail in the functionality of the modules is written in the subsequent subsections.

Centralized Sensor Aggregator (CSA)

The main function of the CSA is collect data from all the sensors that are connected to the SAS. Sensors with knowledge of the IP address of the CSA constantly feed information through UDP sockets to latter. The CSA then filters out undesirable information from the sensor. For example, instances when the sensor hops from one channel to another, and sends mismatched frequency information; such data are deleted. The CSA also makes the decision on whether a sensor is in failure state. Failure state happens when the USRP stops working and sensors resend the same data to the CSA twice. Due to the random nature of the channel, sensors cannot send the same information twice. The CSA feeds the filtered information into the Centralized Radio Environment Management (REM) database. For further analysis, some of the sensor information is also diverted to the experimental tool for visualization and further analysis if needed.

Table 2.2: Interconnected Modules - Centralized Sensor Aggregator

Module	IPC	Additional Information
Sensor	Network Sockets	UDP information received. Information serially packaged in GNURadio PMT [21]
REM	Network Sockets	TCP Connection to the DB. Insertion and Update instructions sent to DB
Data Analysis	Network Sockets	GNURadio PMT information sent to another data analysis tool

Radio Environment Map (REM)

The REM is a PSQL database in which all data about the SAS, sensors and SUs is stored. Each sensor and SU has its own table, which can be referenced by the SAS in decision making.

- Sensor Data: Information stored by the sensor includes its GPS location, IP address and its last active time. Other information stored is what is termed as the occupancy metric(occ) of a given frequency band. The occ is a generic term that is used for the measure of the current state of a frequency band. In its current state, the occ is being used as a measure of the power in a given frequency band, but different researchers can use it to hold other values, for example, the probability that a PU is transmitting with a particular frequency through cyclostationary feature detection [30].
  Other than the REM DB that functions with the SAS, each sensor had its own internal DB that contains historical measurement data.

- SU Data: The same location information about the SU is stored in the DB. The REM also stores the current transmission frequency and the grant time provided by the SAS to a particular SU. Change in Information about the SU is updated every heartbeat interval [27]. SUs are required to send heartbeat status information to the SAS every designated heartbeat interval. This information can include confirmation of the change in the provided frequency or a change in location for a moving SU.

- SAS Data: Every decision made by the SAS about the Radio Environment is stored in the REM. These decisions include the grants given out by the SAS, the detection of the PU, and the perceived location of the PU when trilateration is used. The REM also contains the calculated distance of each sensor from a PU.

- PU Data: Since the PU is not part of the SAS, no information about the PU is currently stored in the REM. The SAS has to utilize the available and historical sensor information to determine the presence of a PU in the Radio Environment.

Table 2.3: Interconnected Modules - Radio Environment Map

Module	IPC	Additional Information
CSA	Network Sockets	TCP Connection. Insertion and Update instructions for sensor information
SAS Cognitive Engine	Network Sockets	TCP Connection. Read, Insert and Update information on SAS decisions
SAS-CBSD Interface	Network Sockets	TCP Connection. Update information about SUs

SAS-CBSD Interface

The SAS-CBSD interface interconnects the SAS with the SUs/ CBSDs. It is an Apache Server that listens to incoming HTTP connections from the SUs. The Apache Server was chosen due to its free software Apache License [1] and its easy integration with the PHP Server-Side Scripting language with its open-source license [2]. PHP already has an inbuilt module that interfaces with the Apache Server. Extra PHP-PSQL modules are added to interface with the REM Database(DB). When a CBSD creates a HTTP connection with the Apache Server, a new PHP background instance is created to respond to the HTTP request sent. Therefore, the new PHP instance has no memory of previous HTTP connections with a particular CBSD. Because of this, a DB is needed to store information about particular CBSD, such that for each HTTP connection, the PHP queries the DB for additional information on the connecting CBSD. If the CBSD is unknown, the server will reject the request with an appropriate response.

Table 2.4: Interconnected Modules - SAS-CBSD Interface

Module	IPC	Additional Information
CBSD	Network Sockets	TCP Connection. HTTP request and response from CBSD
REM	Network Sockets	TCP Connection. Read, Insert and Update information on SAS decisions

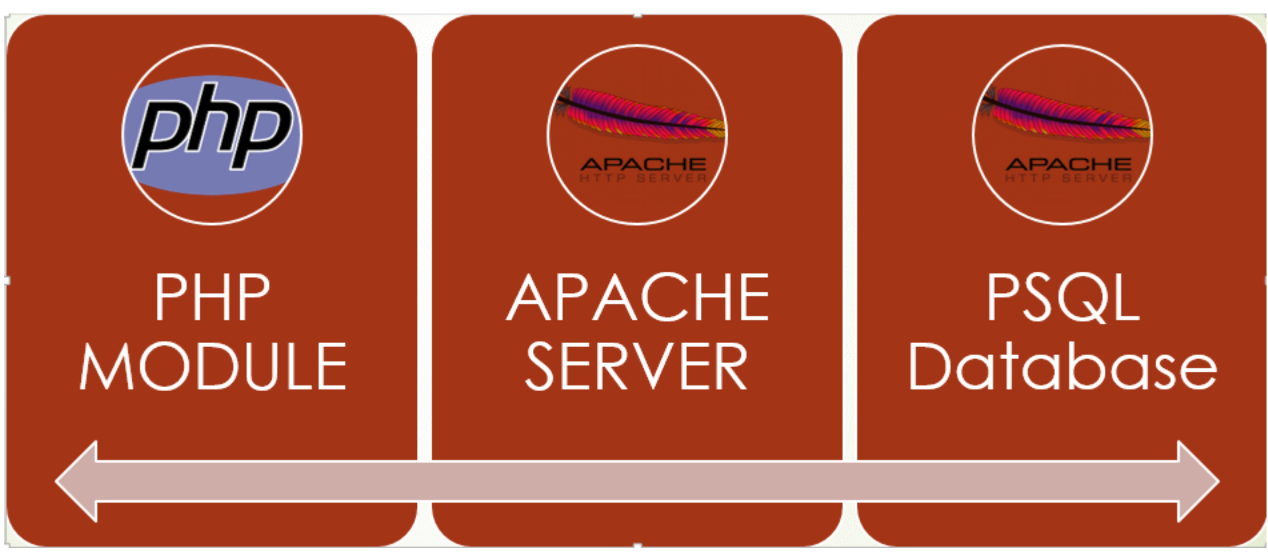


Figure 2.5: SAS Server Side Processes

SAS Cognitive Engine (CE)

This is the module of the SAS that performs the decision making on the Radio Environment managed by the SAS. The program is written in Python due to the plethora of data analysis tools and libraries available in the programming language. Libraries employed include NumPy, which is a computational library useful with linear algebra and random signal analysis[18]. The author is also heavily utilizing the Pandas package for data analysis. Pandas Data Structures [17] are structured in both Matrix and Tabular form which makes transfer of data from the REM seamless, and easier to use. The basic Engine.py employs these basic libraries, and any Python script that is derived from the basic Engine.py builds on top of a good functional Python script. The basic functions of Engine.py are:

- Fetch sensor information from the REM Database and store it in tabular structures called DataFrames with a time-stamp. New sensor data are appended to existing DataFrames to create a structure with historical information for analysis. Therefore, the Centralized REM does not store historical information in the DB, but rather in

the memory of the CE. This is useful because the DB stores information on the Hard Drive whose read/write speeds are much slower than actual RAM, and it would be at a disadvantage to read large historical information from the DB/Hard Drive.

- Analyze data in search of the presence of the PU across the spectrum at each sensor. The analysis is done in a Python class that is derived from the base Engine.py. A derived class is used such that different classes can be used while maintaining the base class.

Table 2.5: Interconnected Modules - SAS Cognitive Engine

Module	IPC	Additional Information
REM	Network Sockets	TCP Connection. Read, Insert and Update information on SAS decisions

2.2.4 CBSD

The CBSD/SU is written in both Python and C++. It follows closely to the OSI model, in such a way that the application layer, which does the HTTP request/response activity, is written in Python while all underlying layers are written in C++.

- Application Layer: The application layer is controlled by radio.py with other peripheral module. The main purpose of the application layer is to create an HTTP connection with the SAS. Once a connection is established, the application layer registers with the SAS based on its id. For each registration, the application layer is assigned a session id called the cbsd_id. Extra functions of the application layer are to randomly request and to negotiate with the SAS for frequency channels and grant time. The application layer also has a socket connection with a GPS device to obtain its current location.

This location is always provided to the SAS in an event when the SU changes location. The application layer also sends and receive information from the lower layers that control the radio.

- Network Layer: The network layer is comprised of either a TUN or TAP interface, which are virtual network interfaces. The TUN interface is used in instances when there is no need for a encapsulation of a packet in a Layer 2 MAC Ethernet header, while the TAP interface is used if Ethernet frames are to be used [15]. The main purpose of the TUN/TAP interface is to create a network interface whose lower layer can be read by other software. All other network interfaces are read by the Kernel which redirects frames to an actual hardware.
  In our configuration, due to the use of software-defined radios, the lower MAC and PHY layer are entirely in software and the TUN/TAP is a bridge between the application layer software and the MAC/PHY layer software.

- Link and Physical Layer: The PHY and Link layer are taken care of by CRTS[24]. For flexibility and generality, CRTS uses the extensible cognitive radio (ECR) waveform, which has an OFDM-based physical layer implemented using the liquid-dsp signal processing library [10]. This open-source library is distributed as with CRTS, as the underlying PHY software to control the SDRs at each node. CRTS is a highly controllable radio experimentation framework that measures performance, such as throughput and packet-error rate in real time, for each radio link of interest during each transmission. All these performance data are sent to the application layer for further distribution to other analysis tools.

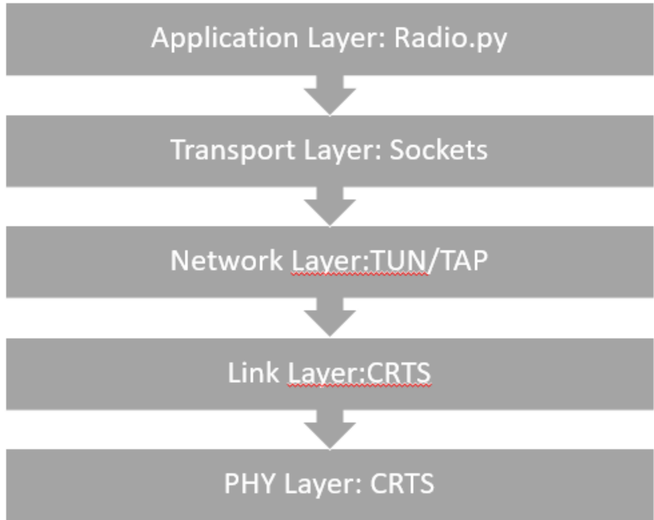


Figure 2.6: Network Stack for the SU/ CBSD

2.2.5 PU

The PU used in the experiment is a derivative of the CBSD implementation. Since CRTS has the ability to create an interferer, the implementation is the same as the CBSD, but with no connection to the SAS. Since the PU has a license to the channel, it can transmit whenever it wants. It is the work of the SAS to coordinate with the SUs to prevent any interference with the PU. The PU sends information to a data analysis tool which is not part of the SAS, but keeps track of the attributes of the PU for performance analysis.

2.2.6 Peripheral Software

Other software was implemented to create a cohesive system between the different modules. GPS information is needed across the entire system. A controller is needed to control any of the modules and check their status. A centralized system is also needed to collect all performance data from all the modules for post-processing and potential visualization.

GPS

To translate GPS information from GPS receiver, the whole system is utilizing a GPS service daemon called the GPSD [22]. GPSD gets data from either USB or serially connected GPS devices. The background service is started on startup for all the host computers in the network. GPSD creates a TCP server connection at port 2947 to which all programs that need GPS information can connect.

In instances when the host does not have a GPS device attached, a researcher has the ability to manually set the GPS location of each device. This is through an extra program called GPSFake. GPSFake reads a configuration file holding the manual GPS coordinates, and serially transfers them to GPSD service. Therefore, in instances when a GPS device is connected, GPSFake should be shutdown for GPSD to switch to the GPS device.

In instances when a researcher wants to get GPS information from another program such as the Centralized RFNEST Software that does not adhere to the GPS protocol, a third program called the GPSProxy was written. Peripheral programs send serialized JSON objects to the GPSProxy with the intended GPS information for that node. GPSProxy then translates the received coordinates to the NMEA data [4] understood by the rest of the programs.

RFNEST

Control of the RFNEST is achieved by two main programs which are the RFView and Channel Emulator Control(CEC). CEC is the program that handles communication between the RFNEST hardware and the host machine. CEC utilizes the Extendable Mobile Ad-hoc Network Emulator (EMANE) which emulates MAC and PHY of simulated networks[3], but in this case, the emulation of the PHY is not handled in software but rather by the RFNEST hardware. The RFView on the other hand is a JAVA-based program with a GUI meant for

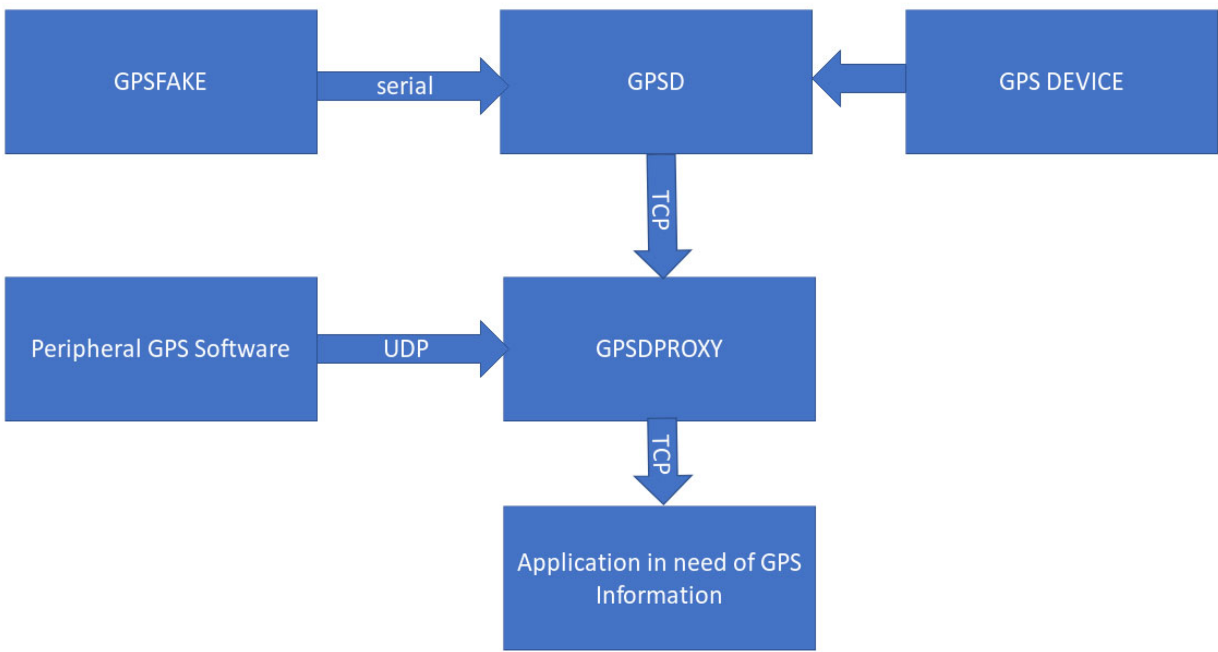


Figure 2.7: GPS Software Diagram

users to create various network scenarios. The graphical scenarios are stored in XML files to be loaded by other RFView software.

As can be shown by Fig. 2.8, the GUI helps users to create their own models which are then translated by the CEC into usable data for controlling the RFNEST hardware. The lines shown in the map indicate the RF channel between the nodes. As shown in the map, Node_ 8 and Node_ 11 have two RF channels to indicate bi-directional communication between the two nodes.

Due to Intellectual Property considerations, no code or internal structure of the RFView or the CEC was changed. Any additional software was meant to record any useful output information from the CEC. The CEC communicates to only the RFView and the Hardware through Broadcast messages or through an output log on to the terminal. To circumnavigate subscribing to the same UDP ports for the incoming multi cast packet so as to read the broadcast messages, the author chose to get information from the output log because it

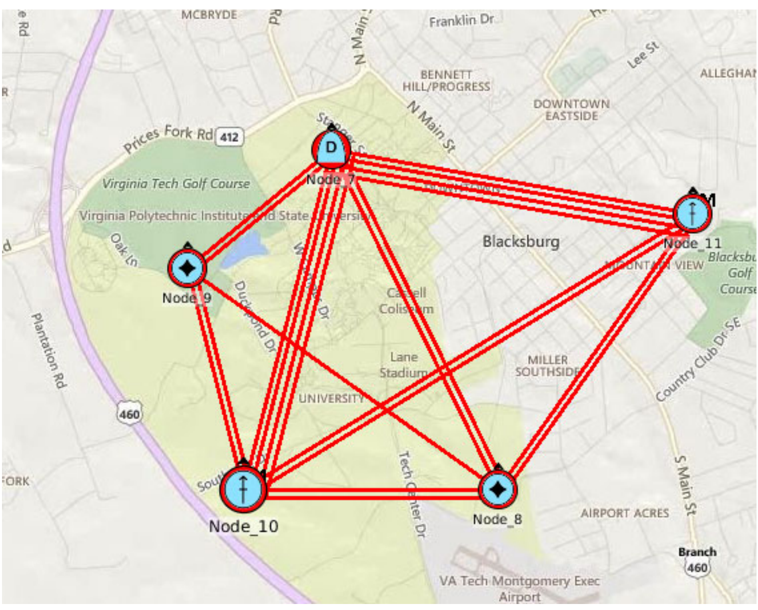


Figure 2.8: Graphical Interface to recreate real environmental models

was easier to program. Terminal output is redirected to a UDP port whose client is able to decipher the information to obtain GPS coordinates of the individual nodes of the RFNEST.

```
./run_debug.sh > /dev/udp/127.0.0.1/9867
```

The coordinates from CEC are packaged as JSON objects are redirected to their respective host computers by rfnest_info.py client process. This is the way in which the RFNEST provides GPS coordinates to the rest of the system.

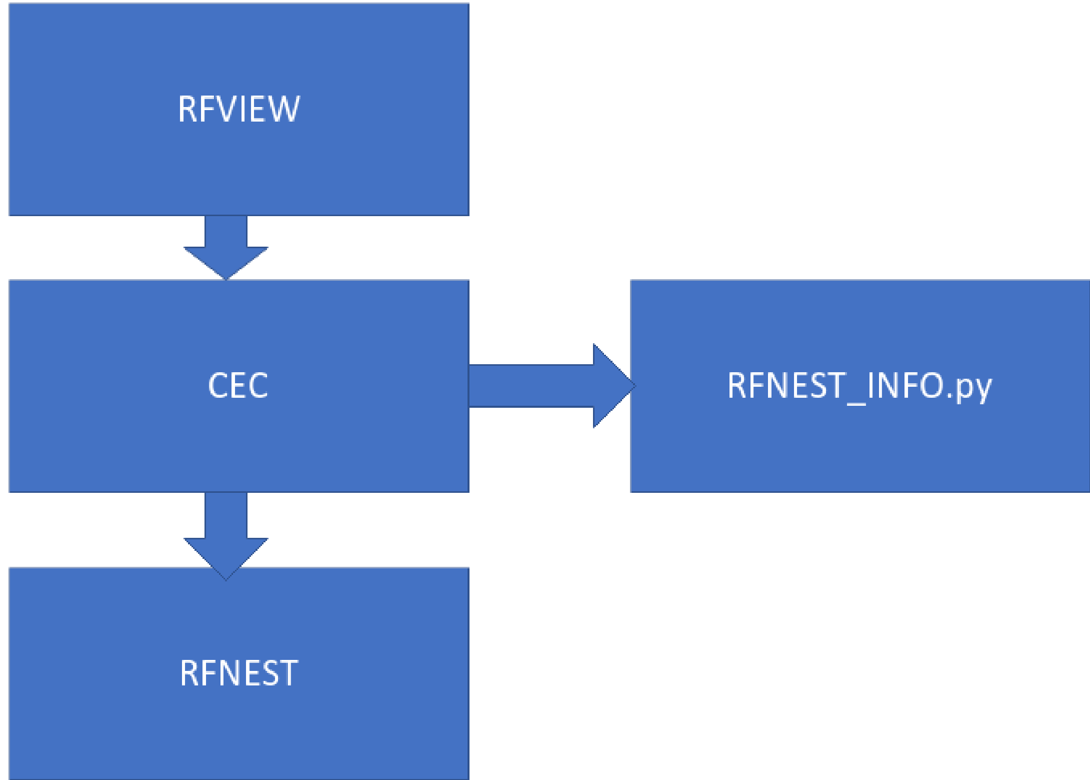


Figure 2.9: Architecture for interconnection of programs controlling the RFNEST

Chapter 3

Experimental Implementation

Chapter 2 provided a general description of the different hardware and software used for the whole system to work. This chapter goes into greater detail of how the system is implemented on the CORNET testbed, and how each module is installed and run. It also goes further into the background services that are setup to coordinate the functionality of the modules.

3.1 Hardware Setup

As previously noted, the SAS is setup on the current CORNET testbed. Due to the flexibility of the system, another experimental CORNET testbed replica is implemented in a single Xeon computer. The new replica was setup to:

1. Prove that as long as nodes are interconnected through a network, the software can be implemented on any available hardware

2. For future experiments, nodes can be virtualized to provide a basis for experimentation using more compact hardware

3.1.1 Host Computing

A dual 3.0 GHz 128GB 12-core Xeon processor(Intel Xeon E5-2697 v2) is used in the final experimentation. The computer has the latest VMWare ESXI, which is a hypervisor that is utilized to manage and run virtual machines in one computer. The hypervisor manages the memory, performance and hardware allocation for each virtual machine that is implemented in the Xeon computer that was used. The computer also has 4 external 10Gbps Ethernet cards with the ability of addition of 4 other Ethernet cards. For our experimentation, two additional external cards were added. One for external access by individual who want to experiment, and the other for direct management of the physical computer.
For the experiment, 5 virtual machines, running Ubuntu 14.04 with 8GB of RAM, were setup. Each computer was interconnected through a virtual network to emulate network-connected system. Furthermore, additional images that are utilized with CORNET were also added for easy access which the CORNET system offered [8].

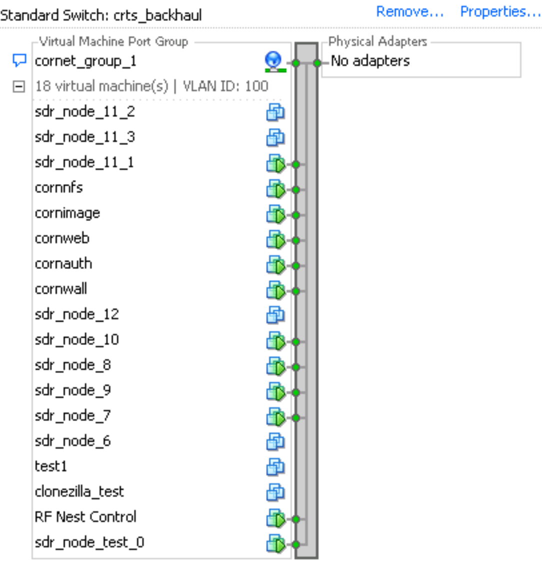


Figure 3.1: Emulated Back-haul Network Architecture Implemented

Each of the nodes on the back-haul network is provided with an IP address based on the node

number. For example, Node 1 would have an IP address of 192.168.1.11 while Node 7 will have 192.168.1.17. The networking convention behind these virtual computers is the same as that provided by CORNET. The network is a virtualization of the Physical CORNET network that is explained in [8]. A virtual machine that controls the RFNEST is also added to the list, with its own unique IP address.

External Networking

The experimental testbed is setup with an external IP address of 128.173.221.41. And the external IP address is one of the available IP addresses that can be connected to, by anyone in the world. For future work, a permanent IP address and host-name will chosen to suit the needs of the people experimenting with the testbed. An external 100 Mbps Ethernet is inserted into the computer to provide the external global IP address. Virtually, the external NIC is only connected to one virtual machine which acts as the firewall between the external world and the rest of the virtual machines. As explained in [8], this provides Secure Shell Tunneling through its pre-configured ip-tables.

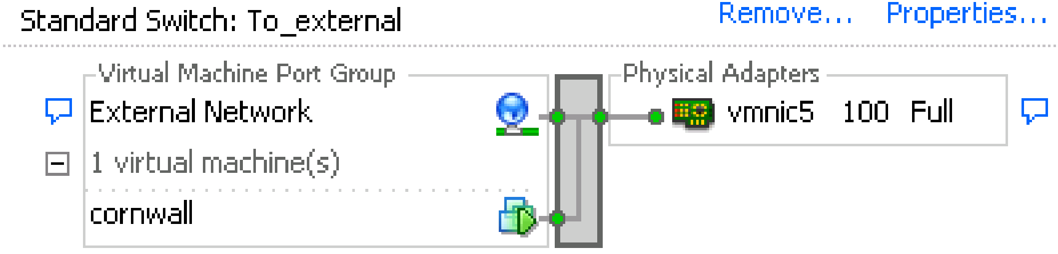


Figure 3.2: Emulated Virtual Network that connects to the external NIC

The third networking done is the connection between the virtual nodes and the USRPs. To accomplish this, all virtual machines get an extra virtual interface which connects to one virtual network. The virtual network is then connected to one 10 Gbps physical NIC. Due to

the small number of virtual machines only one external NIC was needed. There is an option of adding on more physical NICs for increased load. Requesting 20 Mega samples per second from the USRP would translate to $20 \times 16 \times 2 = 640$Mbps for 16bit IQ data. Therefore, each node for a 20MHz bandwidth would require below 1Gbps of data. The 10Gbps physical NIC would handle upto 10 hosts, but in this case, only 5 virtual hosts are used.

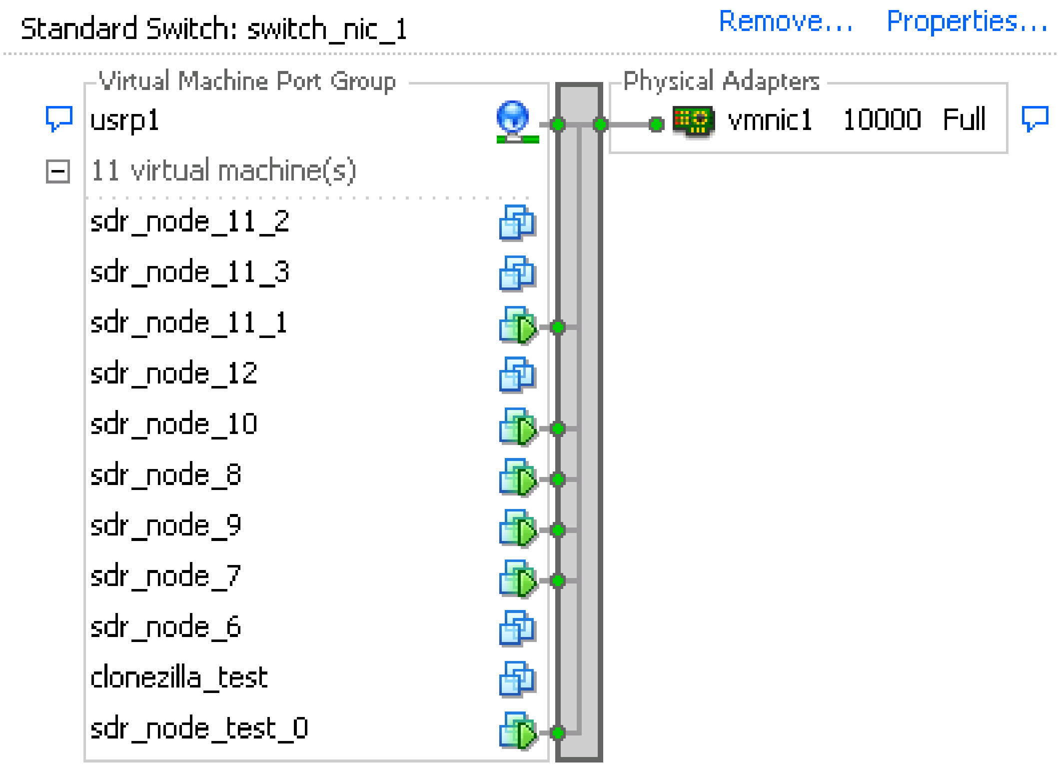


Figure 3.3: Virtual Network for USRPs

The 10 Gbps NIC is connected to an external Juniper switch to which all USRPs are connected. Utilizing the basic ARP scheme, various hosts can be able to ping various USRPs. The interconnection between the USRP and its intended virtual host is purely created through the internal networking of the VMWare machine. The virtual NIC of the virtual host is provided with an IP address and subnet masking which will only accept information from a specific usrp. An example is shown in figures 3.5 and 3.6. In these figures, node

10 with eth1 is given an IP address of 192.168.20.1 with a subnet mask of 255.255.255.0. A USRP connected is also given an IP address of 192.168.20.2. When UHD software is searching for available USRPs, it will probe only USRPs with 192.168.20.xx, which ensures that the above USRP will specifically communicate with the only Node 10.

Figure 3.4: External Switch

With the above configuration, each virtual machine is connected to a USRP through a virtual network. Furthermore, each virtual machine can also communicate with other virtual machines through a different virtual network.

The fourth network need for the experiment testbed is the connection between the virtual machine that should control the RFNEST. Since the RFNEST is required to have an 100 Mbps Ethernet connection, and no 100 Mbps is NIC present, a USB-100Mbps Ethernet is used to connect the RFNEST to the Xeon computer. The USB port is then assigned to the virtual machine that should control the RFNEST.

```
shemk@CORNET-Node-1-10-Port-7010:~$ uhd_find_devices
linux; GNU C++ version 4.8.4; Boost_105400; UHD_003.009.005-0-g32951af2

--------------------------------------------------
-- UHD Device 0
--------------------------------------------------
Device Address:
    type: usrp2
    addr: 192.168.20.2
    name:
    serial: F3ECE0


shemk@CORNET-Node-1-10-Port-7010:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.100   0.0.0.0         UG    0      0        0 eth0
10.0.0.0        0.0.0.0         255.255.0.0     U     0      0        0 mac_interface
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 eth0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 bat0
192.168.20.0    0.0.0.0         255.255.255.0   U     0      0        0 eth1
```

Figure 3.5: USRP detected with internal routing

```
eth0      Link encap:Ethernet  HWaddr 00:30:48:cd:ce:a6
          inet addr:192.168.1.20  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::230:48ff:fecd:cea6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:38101008 errors:0 dropped:22 overruns:0 frame:0
          TX packets:44084595 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:18779973836 (18.7 GB)  TX bytes:58569034783 (58.5 GB)

eth1      Link encap:Ethernet  HWaddr 00:0c:29:6c:82:a2
          inet addr:192.168.20.1  Bcast:192.168.20.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe6c:82a2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:45003020716 errors:0 dropped:770 overruns:0 frame:0
          TX packets:1454005342 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:68034049185249 (68.0 TB)  TX bytes:351327680527 (351.3 GB)
```

Figure 3.6: Created Virtual Interfaces

3.2 RFNEST Configuration

To log into the RFNEST, it is advisable to use x2go client for the RFView GUI or using ssh with x11 forwarding.

```
ssh -XC rfnest@128.173.221.41 -p 7777
```

3.2.1 Dependency Installation

In an instance when a new virtual machine is needed the following procedures are needed to install dependencies for the RFNEST software.

```
wget https://downloads.pf.itd.nrl.navy.mil/emane/archive/0.8.1-r1/emane
    -0.8.1-release-1.ubuntu-12_10.amd64.tgz
tar xzvf emane-0.8.1-release-1.ubuntu-12_10.amd64.tgz
cd emane-0.8.1-release-1/deb/ubuntu-12_10/amd64/
sudo dpkg -i *.deb
sudo apt-get install -f
```

For the CEC, a background service was created to handle both the cec and the communication of gps location to other host machines. To install the background services, the github repository must be downloaded.

```
git clone https://github.com/ShemK/Open-Source-Spectrum-Access-System
    .git
cd Open-Source-Spectrum-Access-System/rfnest
chmod +x rfnest_install.sh
```

Before installation, special care must be taken to ensure that some of the files have the correct file path to the program that runs the CEC as shown in the code below

Listing 3.1: cec.service

```
[Unit]
Description=runs the channel emulator controller in the background
After=multi-user.target
[Service]
Type=simple
ExecStartPre=sudo /sbin/ip route add 224.0.0.0/4 dev eth2
ExecStart=/home/rfnest/rftest/Software/cec/run_cec.sh
[Install]
WantedBy=multi-user.target
```

Listing 3.2: cec_ com.service

```
[Unit]
Description=gets information from the cec and sends it to appropriate
    nodes
PartOf=cec.service
After=cec.service
[Service]
Type=simple
ExecStart=/usr/bin/python /home/rfnest/control/RFNestInfo.py
[Install]
WantedBy=cec.service
```

If all paths are okay, the networking multicast aspect of the CEC is started and the services started. As shown in previous code, the cec.service will start the next communication service in cascade. The user also needs to ensure that the right interface ”eth2” is chosen.

Further information on installation of RFNEST software can be found at [11].

3.2.2 Setting Up

For the experiment, the RFNEST was calibrated with the provided caliberation files. In this case, to utilize the 800 MHz band, an 850.001.calib was selected. These can be found in the Calibration folder. The cec.xml file is configured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE eventagent SYSTEM "file:///usr/share/emane/dtd/eventagent.dtd
    ">
```

```
<eventagent name="Channel Emulator Controller" library="cec">
  <param name="publishpathlossevents" value="on"/>
  <param name="publishlocationevents" value="on"/>
  <param name="processpathlossevents" value="on"/>
  <param name="processlocationevents" value="on"/>
  <param name="basicmodel" value="on"/>
  <param name="antennapattern" value="off"/>
  <param name="analog" value="on"/>
  <param name="fadingmodel" value="off"/>
  <param name="testmode" value="off"/>
  <param name="emanetestmode" value="off"/>
  <param name="locationtestmode" value="off"/>
  <param name="testcount" value="20"/>
  <param name="testincrement" value="0.0001"/>
  <param name="freespacemodel" value="on"/>
  <param name="freespacewavelength" value="0.353"/>
  <param name="tworaymodel" value="off"/>
  <param name="longdistpathlossmodel" value="off"/>
  <param name="longdistpathlossexponent" value="2.0"/>
  <param name="hatamodelurban" value="off"/>
  <param name="hatamodelsuburban" value="off"/>
  <param name="hatamodelrural" value="off"/>
  <param name="hatapcsmodelurban" value="off"/>
  <param name="hatapcsmodelsuburban" value="off"/>
  <param name="hatafrequency" value="2400"/>
  <param name="frequency_calib" value="850.001.calib"/>
  <param name="antennaprofile" value="antenna/antenna1.xml"/>
```

```
  <param name="dopplerincrement" value="9.31322575"/>
  <param name="systemloss" value="0.0"/>
  <param name="number_of_nodes" value="16"/>
  <param name="manual_control" value="off"/>
  <param name="legacy_analog_matrix" value="on"/>
</eventagent>
```

3.2.3 Running

```
sudo ip route add 224.0.0.0/4 dev eth2
sudo systemctl start cec.service
rfnest@rfnest:~/rftest/Software/rfview$ ./rfview.sh
```

The rest of the scenario setup is done by the user in the RFView GUI.

3.3 Installation

As previously noted, each node can be any of the subtypes of the whole system. Therefore, the same installation procedure works for all.

```
git clone https://github.com/ShemK/Open-Source-Spectrum-Access-System.
    git
cd Open-Source-Spectrum-Access-System
sudo ./install.sh
```

The installation script installs all dependencies required for the system to run. Installation has been tested for Ubuntu 14.04. Notable libraries installed are:

- python-pip: For installing python packages
- postgres psql: PSQL database
- libpqxx: C++ client API for PostgreSQL
- php5: PHP Engine
- apache2: Webserver
- gpsd: gps client

3.3.1 Background Services

The installation also creates background services that help with the functioning of the whole system:

- sdr_phy.service: This is a background service that will run the sensing for a given host machine. When started, it will run two python scripts needed for sensing as previously noted in Chapter 2. This service was created such that a remote user can log out, and the sensing would continue without interruption..
- fakegps.service: It starts the GPSFAKE process that provides a manual location to the virtual machine.
- aggregator.service: Waits for information from all sensors and redirects it to the database
- gps_proxy.service: Waits for information from external entity such as the RFNEST to manipulate GPS information for a specific node

- cbsd_ control.service: Expects commands from a central_ controller which designates the type of node i.e. either a PU or an SU. The central_ controller is a program that is used to designate each nodes functionality.

3.4 Using the SAS

The current experimental testbed is implemented with a global IP address of 128.173.221.41

3.4.1 Logging In

The logging in procedure is the same as the one provided by [8]. A computer with a program that can run Secure Socket Shell(SSH) should be utilized. With Linux based machines, this can be easily accomplished through the terminal.

```
ssh -XC shemk@128.173.221.41 -p 7011
```

The user-name is provided by CORNET, and the port is 70xx with xx representing the node number. Therefore, the port for node 11 is 7011. The firewall reroutes the ssh information to the appropriate node.
With windows systems, Putty will be more useful or Windows BASH subsystem can be utilized to obtain a platform that can use SSH. Information on logging in will not be repeated in this work, but can be obtained from [8].

3.4.2 Starting the SAS

The user moves to the folder with the software. The first thing to do is to write a configuration file that assigns different functionality to the nodes. The configuration file is called

cornet.cfg.

```
shemk@CORNET-Node-1-11-Port-7011:~$ cd Open-Source-Spectrum-Access-System/
shemk@CORNET-Node-1-11-Port-7011:~/Open-Source-Spectrum-Access-System$ ls
aggregator  cbsd  control  crts  README.md  rem engine  server  services
shemk@CORNET-Node-1-11-Port-7011:~/Open-Source-Spectrum-Access-System$ cd control/
shemk@CORNET-Node-1-11-Port-7011:~/Open-Source-Spectrum-Access-System/control$ ls
cornet.cfg  cornet_controller.py  README.md
shemk@CORNET-Node-1-11-Port-7011:~/Open-Source-Spectrum-Access-System/control$ vi cornet
cornet.cfg            cornet_controller.py
shemk@CORNET-Node-1-11-Port-7011:~/Open-Source-Spectrum-Access-System/control$ vi cornet.cfg
```

Figure 3.7: Configuring the whole experiment

Listing 3.3: cornet.cfg

```
#configuration file for nodes

log_path = "/users/shemk/sas_logs";

SAS = {
  nodeList = (
    {
      nodeID = 1;
      ip = "192.168.1.21";
      log = true;
    }
  );
};

PU = {
  nodeList = (
    {
      nodeID = 1;
      ip = "192.168.1.17";
```

```
      random_distribution = "poisson";
      max_time = 5;
      min_time = 0;
      scenario_controller = "SC_SAS_PU";
      pu_type = "interferer"
      run_time = 240.0;
      log = true;
      information_parser = "192.168.1.21";
    }
  );
};

SU = {
  nodeList = (
    {
      nodeID = 1;
      ip = "192.168.1.20";
      random_distribution = "poisson";
      max_time = 5;
      min_time = 0;
      grouped = ("192.168.1.21","192.168.1.20");
      sas = "192.168.1.21";
      scenario_controller = "SC_Cbsd";
      guard_band = 3; #in MHz
      cbsd = {
        fccId = "cbd563";
        cbsdCategory = "A";
```

```
      userId = "cbd3";
      cbsdSerialNumber = "hask124ba";
      cbsdInfo = "yap";
    };
    log = true;
    information_parser = "192.168.1.21";
  },
  {
    nodeID = 2;
    ip = "192.168.1.18";
    random_distribution = "poisson";
    max_time = 5;
    min_time = 0;
    grouped = ("192.168.1.17","192.168.1.16");
    sas = "192.168.1.21";
    scenario_controller = "SC_Cbsd";
    guard_band = 3; #in MHz
    cbsd = {
      fccId = "cbd561";
      userId = "cbd1";
      cbsdCategory = "A";
      cbsdSerialNumber = "hask124ba";
      cbsdInfo = "yap";
    };
    log = true;
    information_parser = "192.168.1.21";
  },
```

```
    {
      nodeID = 3;
      ip = "192.168.1.12";
      random_distribution = "poisson";
      max_time = 5;
      min_time = 0;
      grouped = ("192.168.1.12","192.168.1.11");
      sas = "192.168.1.21";
      scenario_controller = "SC_Cbsd";
      guard_band = 3; #in MHz
      cbsd = {
        fccId = "cbd564";
        userId = "cbd4";
        cbsdCategory = "A";
        cbsdSerialNumber = "hask124ba";
        cbsdInfo = "yap";
      };
      log = true;
      information_parser = "192.168.1.21";
    }
  );
};

SENSOR = {
  nodeList = (
    {
      nodeID = 1;
```

```
            ip = "192.168.1.18";
            log = true;
        },
        {
            nodeID = 2;
            ip = "192.168.1.19";
            log = true;
            information_parser = "192.168.1.21";
        }
    );
};

CHANNEL_EMULATOR = {
    emulatorList = (
        {
            ip = "192.168.1.178";
            nodeList = (
                {
                    port = 0;
                    ip = "192.168.1.19";
                },
                {
                    port = 1;
                    ip = "192.168.1.17";
                },
                {
                    port = 2;
```

```
            ip = "192.168.1.17";
          },
          {
            port = 3;
            ip = "192.168.1.18";
          },
          {
            port = 4;
            ip = "192.168.1.21";
          },
          {
            port = 5;
            ip = "192.168.1.21";
          },
          {
            port = 6;
            ip = "192.168.1.20";
          },
          {
            port = 7;
            ip = "192.168.1.20";
          }
        );
      }
    );
};
```

The above configuration file is one of the main important part of the experiment. Commands in this configuration file are sent to each of the nodes so that it saves the user from manually starting each part of the system. It is split up into SAS,SU,PU,CHANNEL_ EMULATOR.

- CHANNEL_ EMULATOR informs the RFNEST to which IP address GPS information has to be sent. For example, any information concerning Port 7 should be sent to Node 10 at 192.168.1.20. This information is sent to the gps_ proxy.service for that particular node.
- SENSOR informs the SAS which nodes are going to be sensors. Furthermore, it also tells the sensors to which ip address, sensor information has to be sent.
- SU configuration provides an SU with information about the HTTP address of the SAS and basic information like the fccId, serialNumber. This information that would be needed for registration at the SAS. Other information that is needed is the ip address of the node to which the SU should communicate to for performance analysis. This is because the underlying CRTS being used, would require two nodes to have bi-directional communication.
- PU just gets information on how often it should run, and what type of PU it is.

After configuration, the cornet_ controller.py is used to send commands to each nodes and also start some of the background services in each node. The background services would require that each node has the same SSH login information. If the nodes have different login information, then the corresponding background services will need to be started manually at each node.

```
python cornet_controller.py -h
```

The above help option will list how to start each of the nodes. For the sensors, the python program starts each of the sensors by remotely starting the sdr_phy.service. For either the PU or SU, command are just sent to the corresponding cbsd_control.service which controls the functionality of either the PU or SU.

At this point of the system, the SU or PU are not started automatically, but the user has to log into the corresponding nodes to actually start them. The instructions on their behavior will have already been provided by information from cornet.cfg file.

```
ssh -XC shemk@128.173.221.41 -p 7010
cd Open-Source-Spectrum-Access-System/cbsd/
python radio.py
```

The above code will start the SU. Starting the PU requires the same process.

```
ssh -XC shemk@128.173.221.41 -p 7007
cd Open-Source-Spectrum-Access-System/pu/
python primary_user.py
```

With every required process running, the last process to run is the engine behind the decision making of the SAS. At this point, the SAS receives all sensor information, it receives requests from the SUs. The last thing to do, is to make decisions on the availability of the spectrum.

Different users will create their SAS Cognitive Engines for decision making. The current engine is called the ProtoEngine and its current implementation is covered in section 3.5.

3.5 Cognitive Engine Implementation

Current prototype of the derived class of the Cognitive Engine is a reactive-based engine which uses historical data to make a decision on the presence of a PU. After fetching sensor

data from the REM, the engine converts the occupancy metric values into received power at the sensor in a particular frequency band.

The Expected Noise Floor for a particular SDR is calculated from the inherent quantization noise of its ADC. The quantization noise provided by a typical ADC ADS62P49 is -73.4 dBm according to its data sheet[25] for a 20MHz channel. Since noise power is spread across frequency bins after an FFT, the new noise power in each been is calculated by

$$NoiseFloor = -73.6 - 10 * \log(Fs/2) \tag{3.1}$$

where Fs is the number of FFT bins and 73.6 is the typical quantization noise power[25]. With the theoretical Noise Floor calculated the reverse of eqn 2.1 is used to calculate the power in a specific frequency channel. Historical data for each channel is held in memory. The median value of the power form from the single frequency hop is used for further calculations. Using the median value reduces on the number of outliers and false positives that arise from the sensors.

With knowledge of the power with which the PU is transmitting, the distance from each sensor to the PU is interpolated using various theoretical path loss equations. Since all experimental scenarios were accomplished over free-space path loss, the free-space path loss equation was used.

$$PossibleDistance = (10^{\frac{PUTxPower - ReceivedPower}{20}} * 3e8)/4\pi \tag{3.2}$$

The possible distance of the PU is then stored in the REM database for each sensor. Two methods are then utilized to make a decision on the presence of a Primary User.

1. The rudimentary method that is used involves informing every SU that is in the same vicinity as the PU to switch to a different available channel. The only issue with this

method is the lack of knowledge of the position of the primary user.

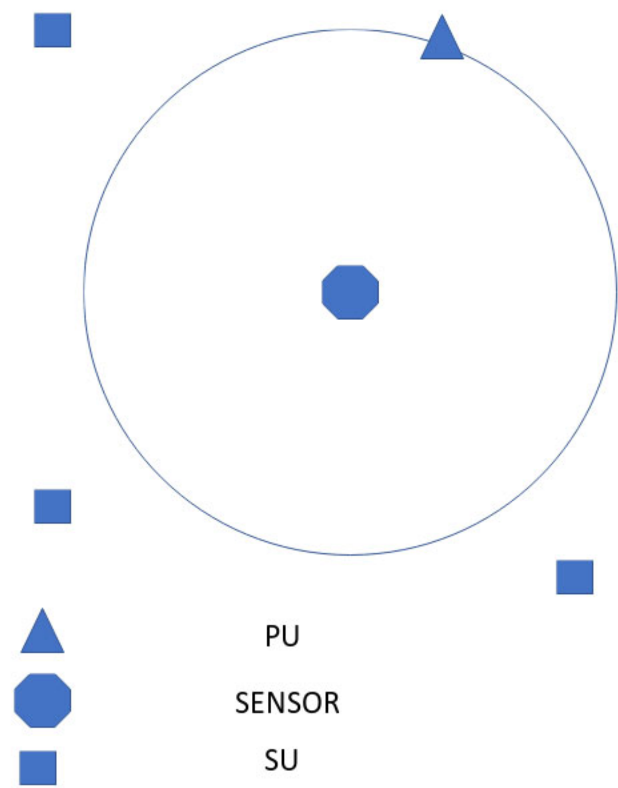


Figure 3.8: Rudimentary PU detection with a single sensor

The issue with this detection is that the SAS does not know the location of the PU. Therefore, a flag will be raised for all SUs that share the frequency band. Therefore, SUs that are potentially far away will also have to skip that frequency channel as shown in Figure 3.8.

2. The second method used involves, trying to obtain the possible location of the PU through Trilateration. In this case, the channel is made free for SUs in particular spatial areas that will not experience interference from the PU, and whose interference would not affect the PU.

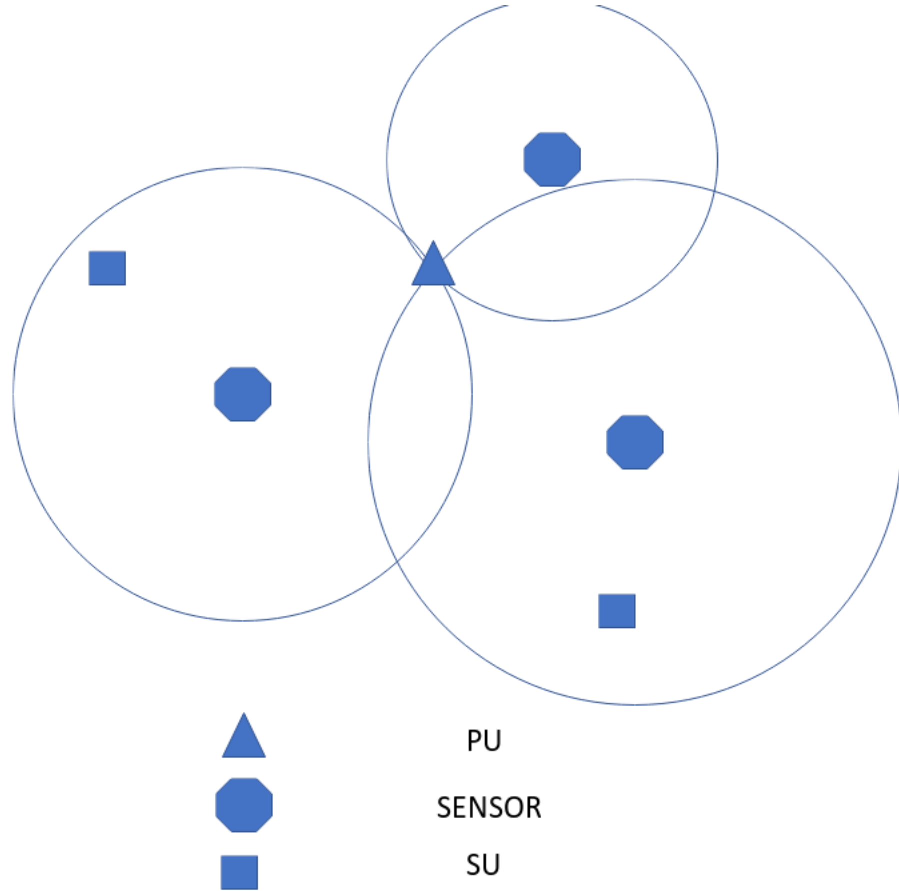


Figure 3.9: Trilateration Detection Using Three Sensors

3.6 Data Logging and Visualization

An extra program that can be utilized, either on the testbed or an external machine to gather data from all modules. UDP information packaged a GNURadio PMT are sent to one process called MailingService.py. The process transfers the data to an appropriate IP. By appropriate IP, the testbed should be able to ping that particular computer. This can happen if the computer and the testbed are under the same subnetwork such that routers can move packets to the appropriate IP address, or if the computer is a Virtual Private Network (VPN) associated with the testbed. In this case, computers connected to the Virginia Tech network can collect the data, but computers outside Virginia Tech's Network will need to

connect through a VPN.

The IP address associated with data collection should be added to the MailingService.py script for relaying purposes. Users can create their own program to read the data, but a Qt program was written to get the data, and log it into csv files, while showing in real-time what is happening with the entire system. Data are logged when a Primary User begins transmission and stopped when transmission ends. The logged data can be utilized for post processing.

Chapter 4

Results

The experimental results were obtained from the testbed in the Chapter 3. With 20dB attenuation on RFNEST, and an inherent 30dB attenuation from the RFNEST, the total attenuation of any transmission signal from the SDR is 50dB. To counter this attenuation, a 50 dB software gain was configured in the RFNEST scenario. The 50dB gain is added to the channel emulation rather than actual gain in power.

Initial measurements were done with a USRP N210 and the Textron SA2500 Spectrum Analyzer to determine if the above attenuation logic was correct. The first control experiment was done by connecting the N210 through a 20dB attenuator at 850 MHz and its output power was noted. The next experiment involved adding an RFNEST and increasing the scenario power to counter any inherent attenuation. Both experiments produced matching results.

4.1 PU Detection

From the results, the PU is assumed to be more wide-band than the SU, which simplifies Power Detection. With PU detection, results are based on latency between transmission and detection, the latency between PU transmission and location detection, the error between PU location and the actual location of the PU. The error between calculated sensor-PU

distance and the actual sensor-PU distance.

With a PU transmitting for 4 minutes, the following results were obtained for the estimated location of the PU. Each sensor is able to make a possible calculation of the distance between the PU and itself. All percentages are based on over 600 possible distance measurements.

4.1.1 Localization

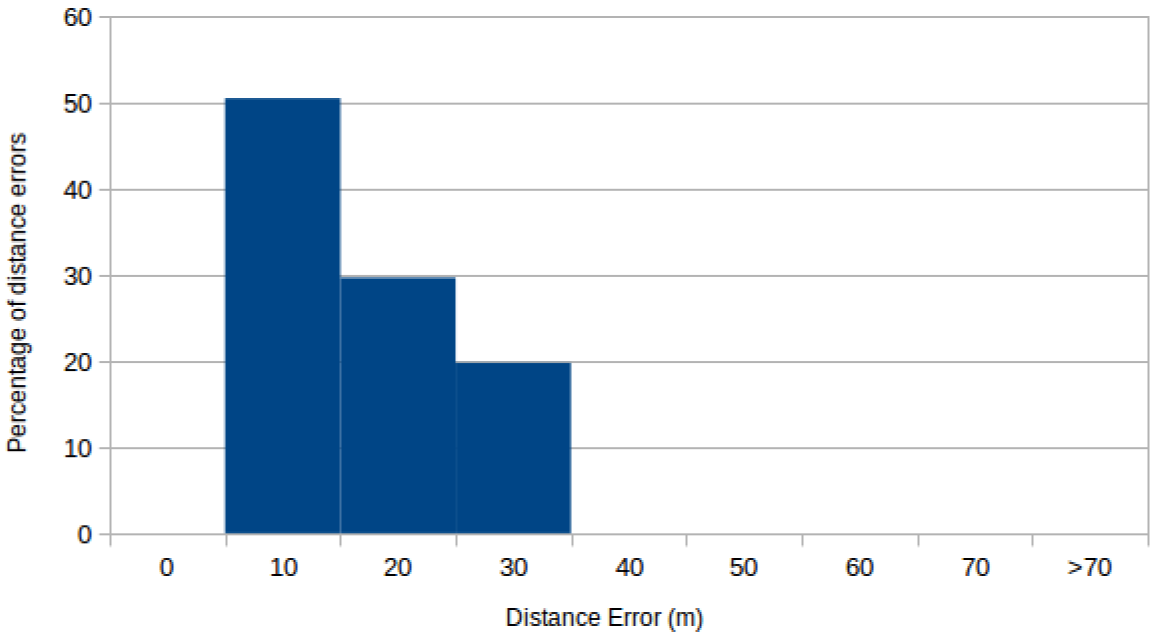


Figure 4.1: Histogram Showing the Error in Possible Distance between the Sensor 1 and the PU

As can be seen from the graph, each sensor had about 50% of its distances below 10m from the actual distance and about 70% with an error below 20m. Differences in histograms is dependent on the sensor being used. Since each SDR differs from the ideal SDR.

The estimated distances obtained are then utilized to find the position of the PU using eqn 1.3.

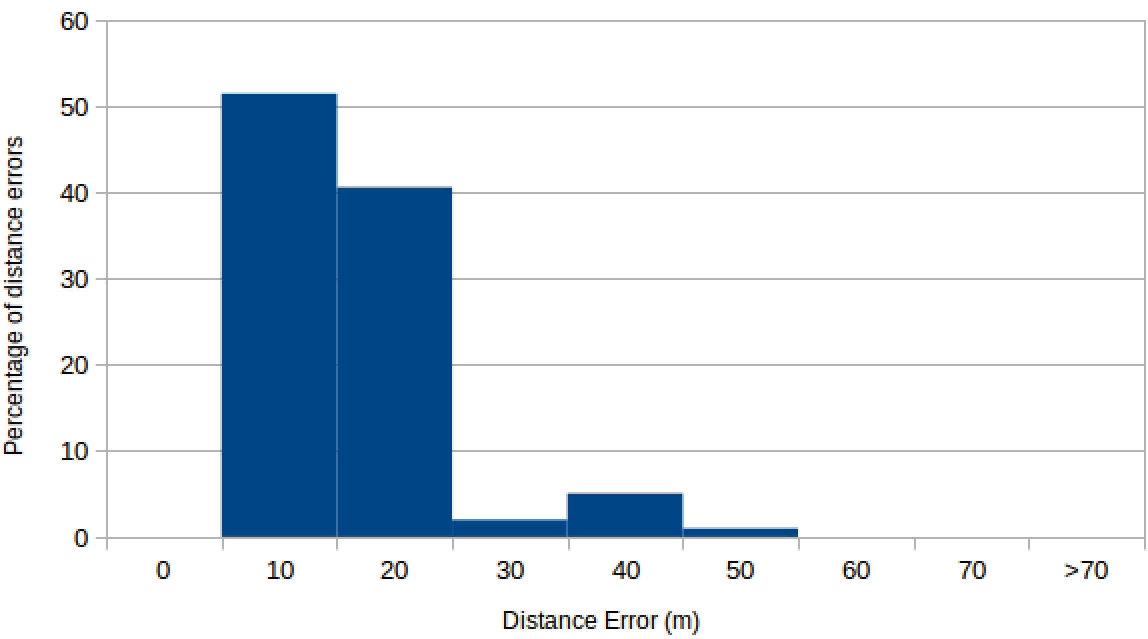


Figure 4.2: Histogram Showing the Error in Possible Distance between the Sensor 2 and the PU

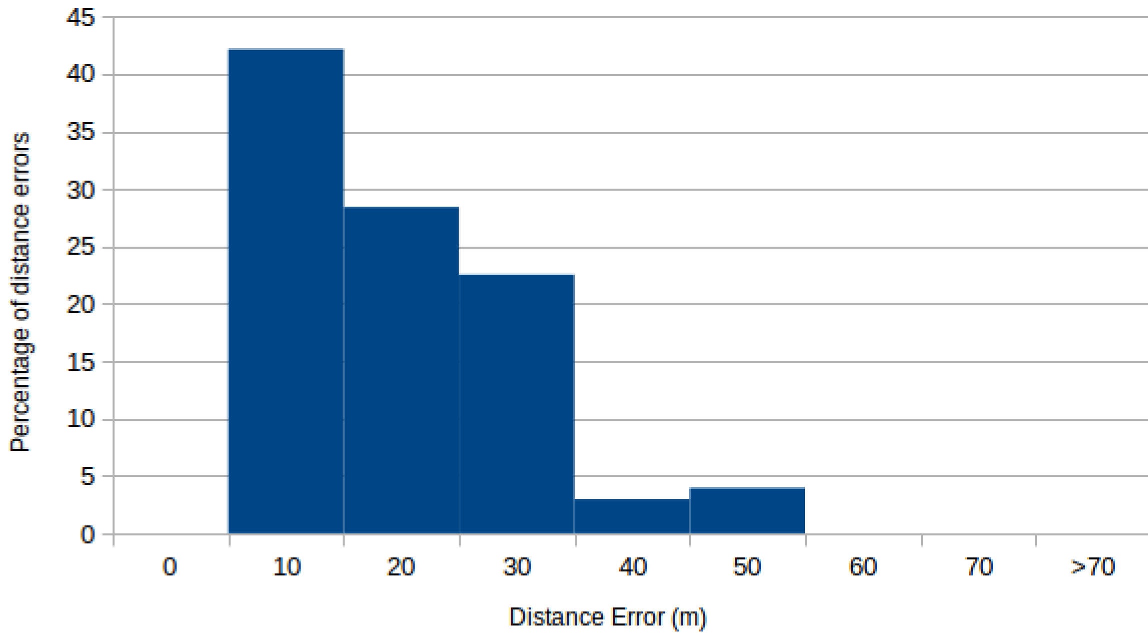


Figure 4.3: Histogram Showing the Error in Possible Distance between the Sensor 3 and the PU

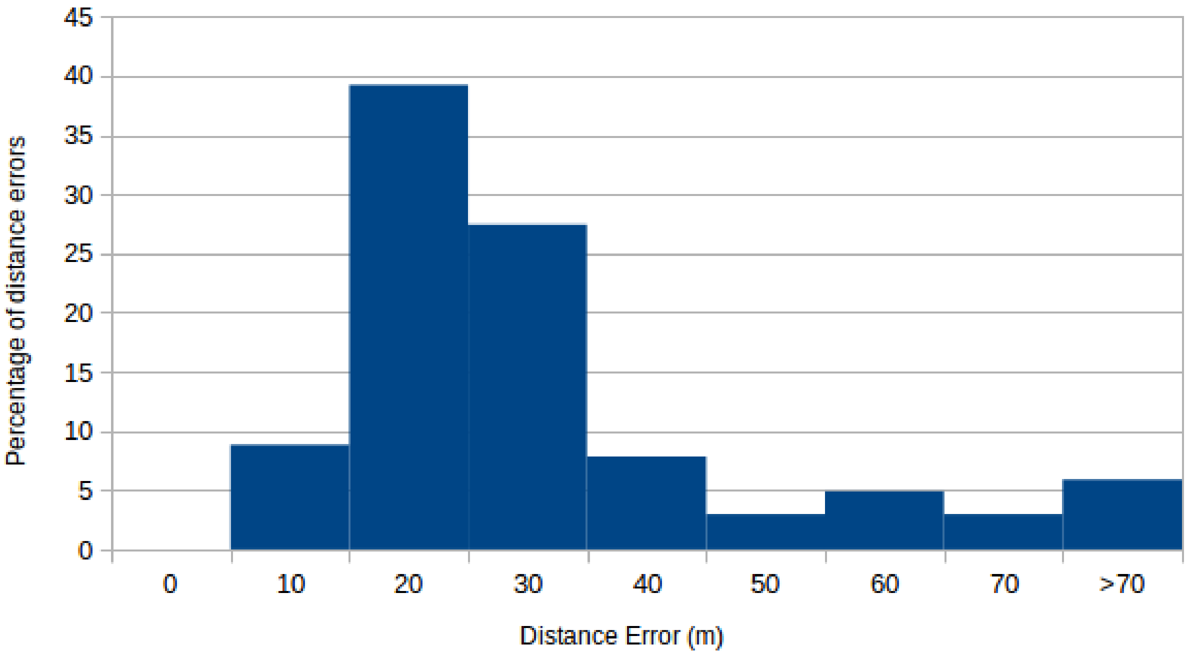


Figure 4.4: Histogram Showing the Error in Possible Distance after Trilateration

Based on signal power received, the trilateration produces error of less than 20m for about 50% of the measurements, with 70% having an error of less than 30m.

Xuiyan [31] showed an improvement in the signal strength localization techniques by averaging the signal strength values. Due to the system's flexibility, a moving average of 5 sensor data points was implemented to test the theory, and compare with the previous method of using the median value.

As can be shown in Fig 4.5, it improves the percentage of errors less than 10m, but also increases the number of errors above 30m. Increase in errors below 10m is due to reducing of short erratic deviation that occur while measuring power. Increase in errors above 30m is due to the fact that if a sensor gets poor errors for a longer time, they greatly skew the average.

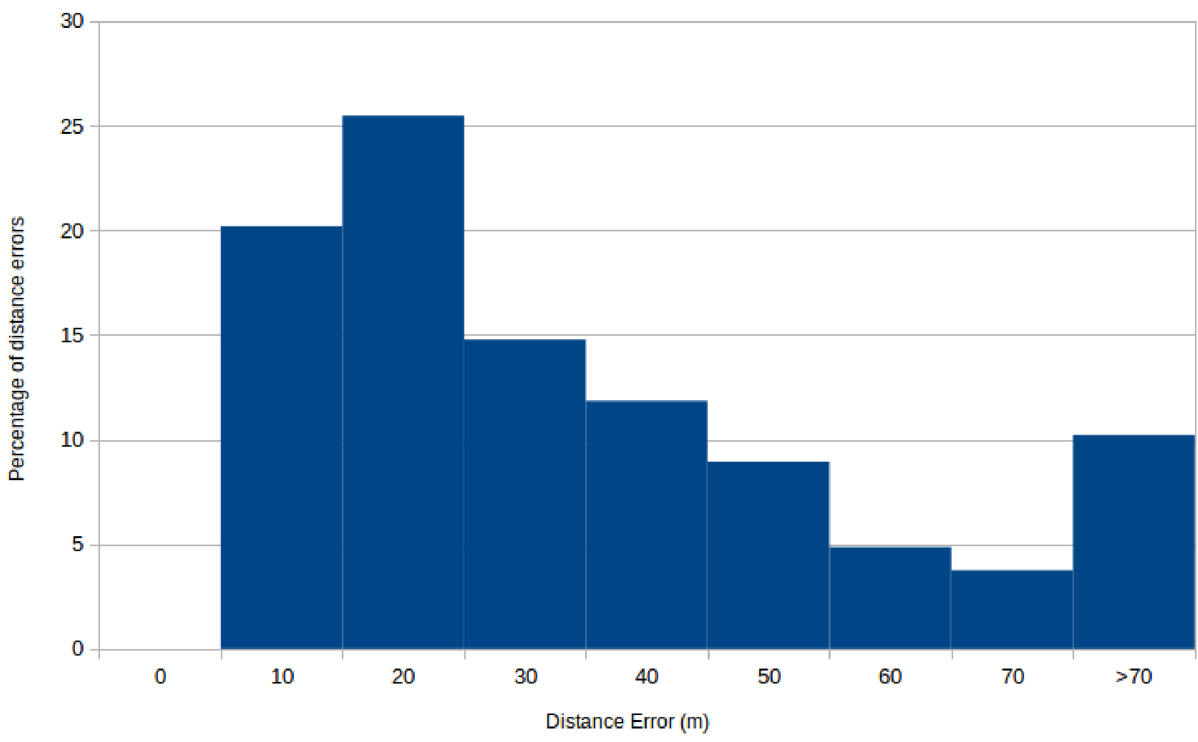


Figure 4.5: Histogram Showing the Error in Possible Distance after Trilateration after Moving Average

4.1.2 Detection Latency

Because of the huge transfer of data from each module to modules to data collection, the latency between PU transmission and detection is affected by many factors. These include CPU speed utilized to analyze the data, the network speed between the different modules, the network connection between the modules and the data collection for post-processing. In all these results, the data were collected at a computer which is not part of the testbed. With this in mind, latency is measured in seconds since a fine grained measurement will vary widely.

As shown in the table above, the average latency for first sensor detection is below 1s. Not all sensors are able to detect the PU within 1s, and therefore, the location detection average

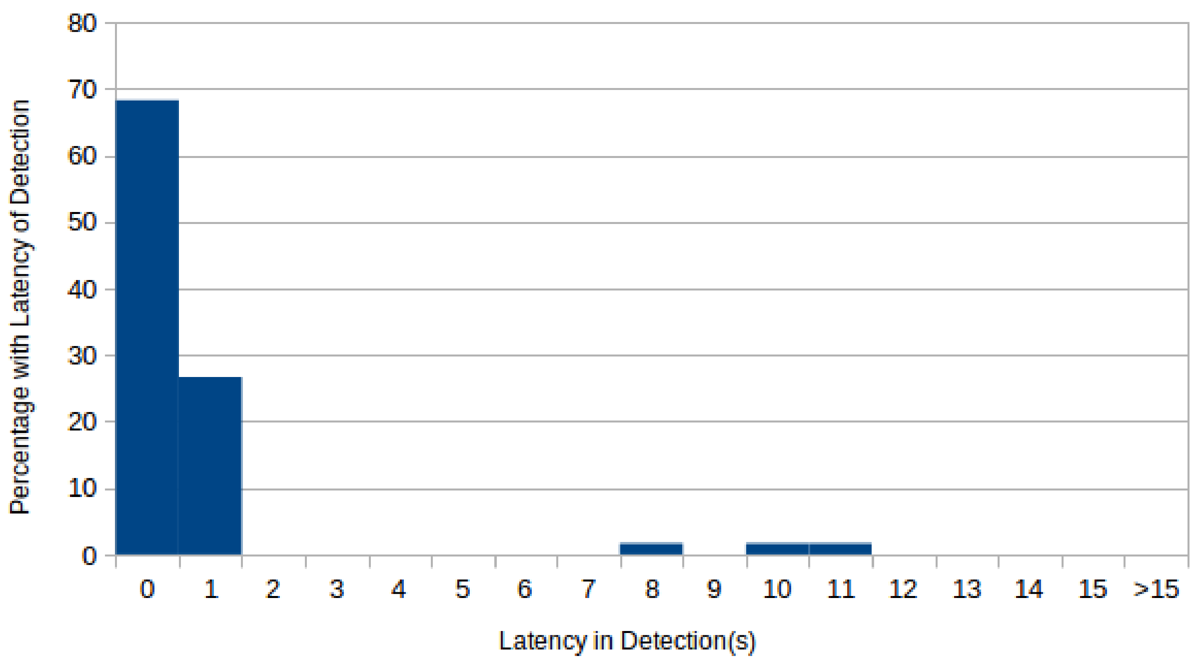


Figure 4.6: Histogram Showing Latency of Detection for a single sensor

Table 4.1: Average Latency after 50 PU transmissions in seconds

First Sensor Detection Latency	All Sensor Detection and PU location detection Latency
0.75	5.38

latency increases to about 2s.

4.2 SU Performance

Since this is all Frequency and Spatial allocation of channels, the SU performance drops only when the PU starts transmitting. Therefore, performance drops within the time between PU transmission and detection. During data collection, the performance of the SU is provided in real-time and logged for post-processing. Frequency changes are informed with a second after the PU has been detected due to the heartbeat interval between the SU and the SAS.

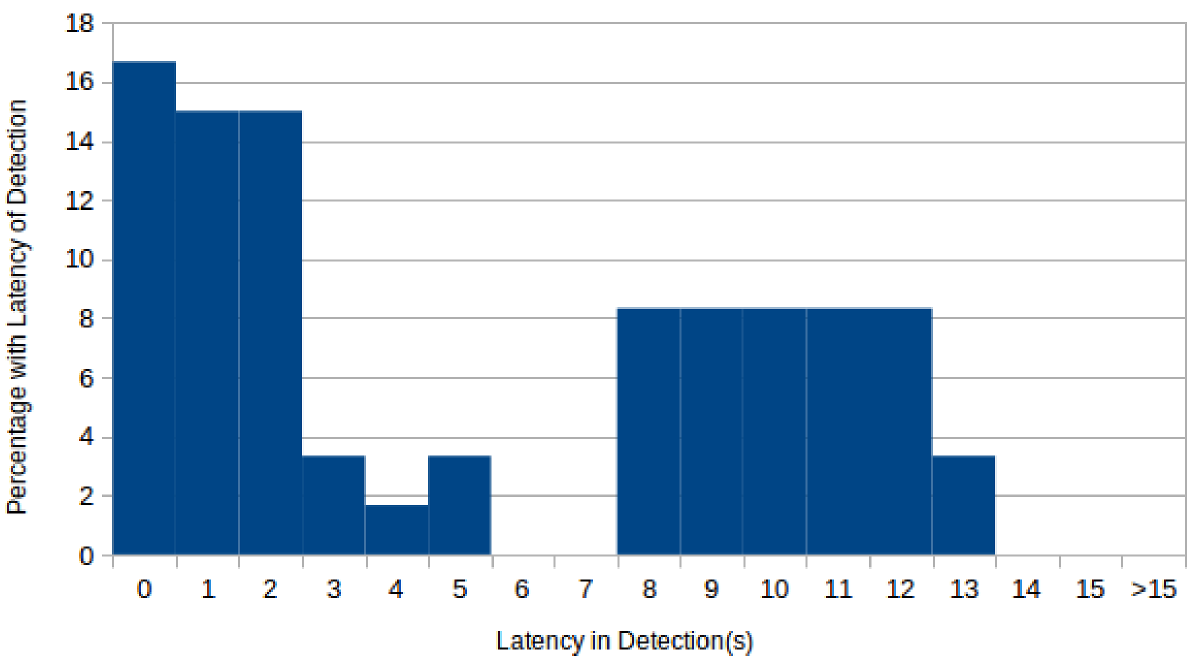


Figure 4.7: Histogram Showing Latency of Detection with Localization

4.3 Real Time Sensor Performance

Real Time Sensor Performance is also provided to obtain the current occ value of the sensor across the whole spectrum is also available for display. These real-time displays provide researchers a visual perspective of how their current implementation is working.

An example is shown with figure 4.9, which shows that a sensor has detected a wide-band signal in the 870 MHz band.

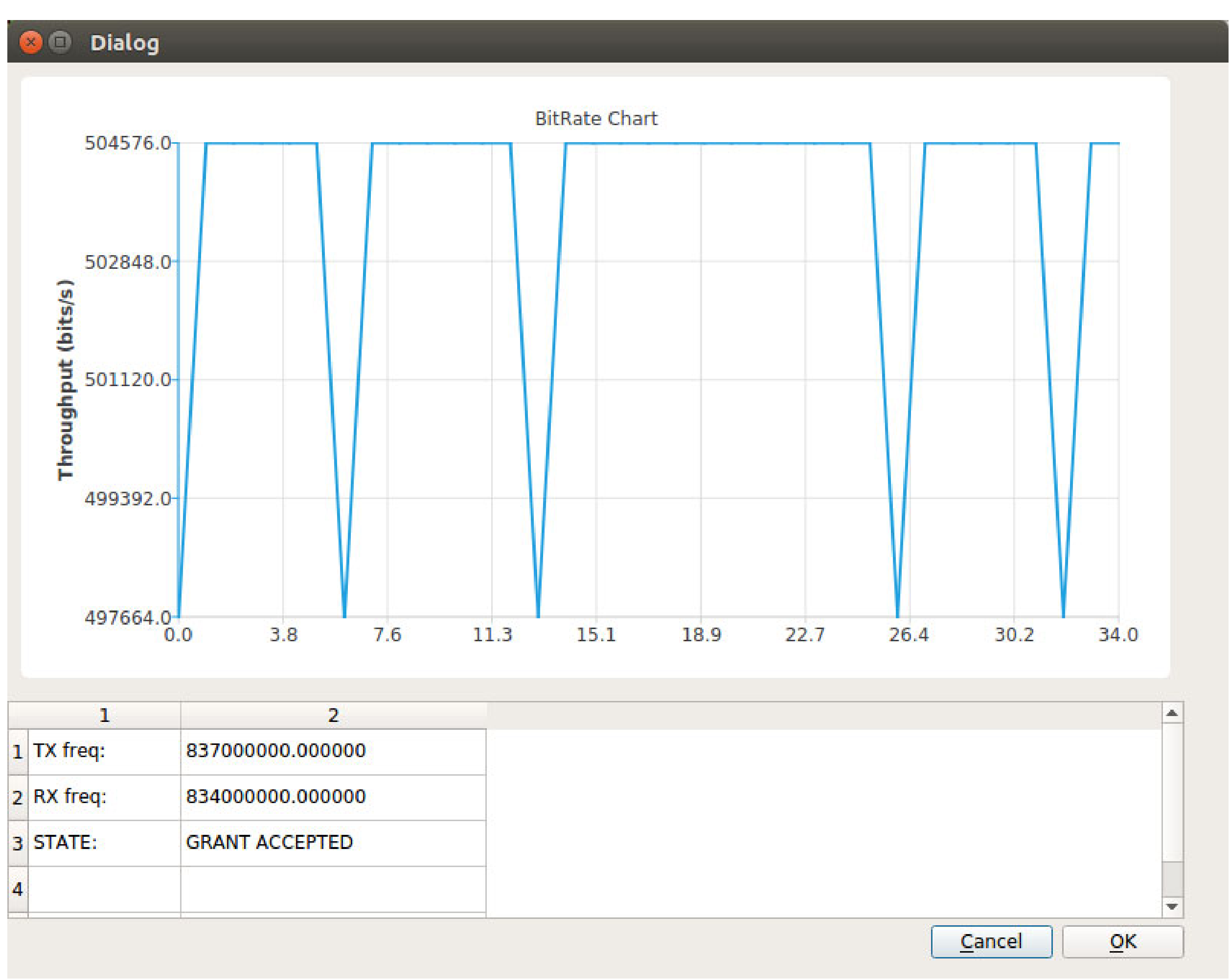
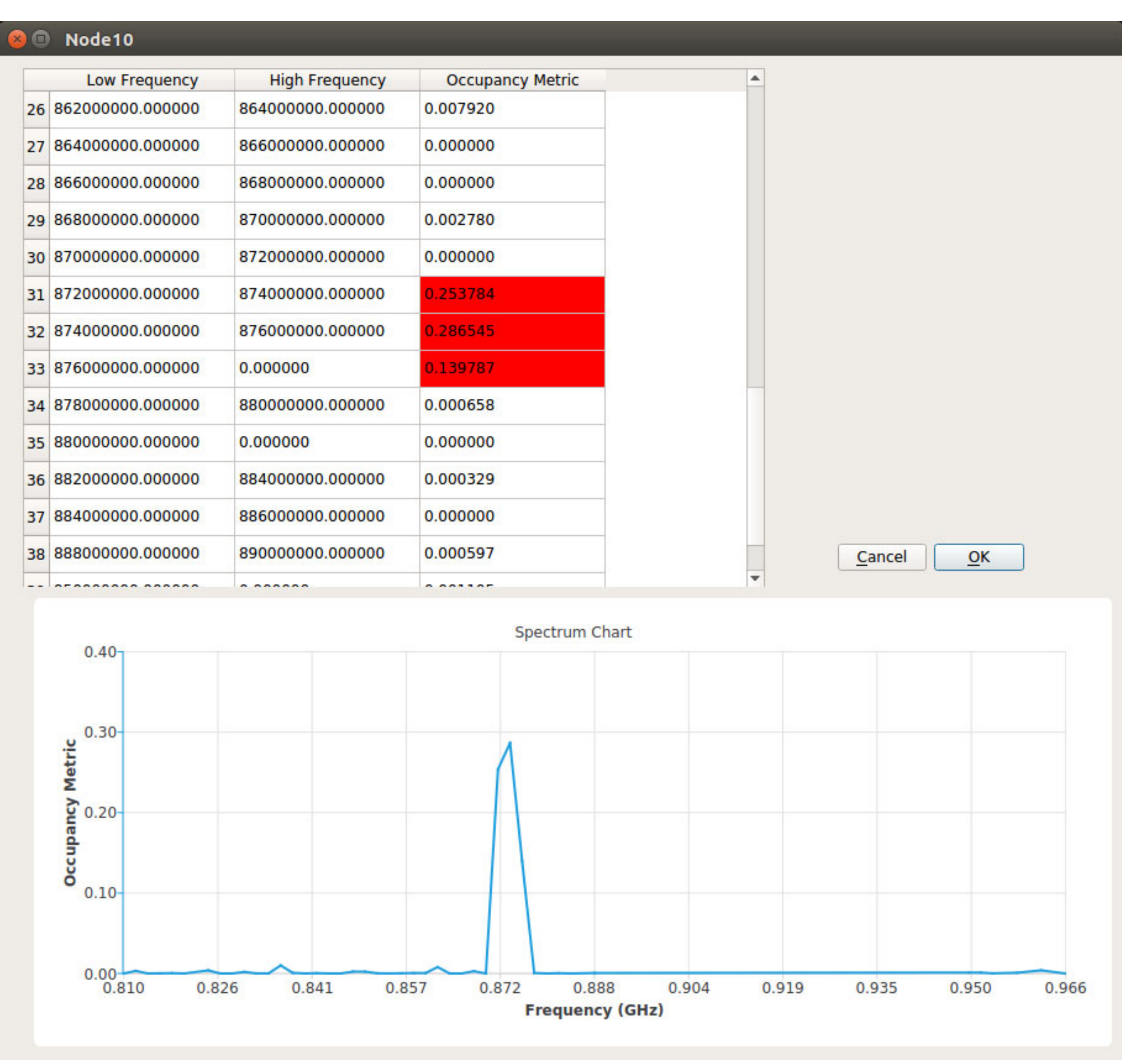


Figure 4.8: Real Time SU Performance Display



	Low Frequency	High Frequency	Occupancy Metric
26	862000000.000000	864000000.000000	0.007920
27	864000000.000000	866000000.000000	0.000000
28	866000000.000000	868000000.000000	0.000000
29	868000000.000000	870000000.000000	0.002780
30	870000000.000000	872000000.000000	0.000000
31	872000000.000000	874000000.000000	0.253784
32	874000000.000000	876000000.000000	0.286545
33	876000000.000000	0.000000	0.139787
34	878000000.000000	880000000.000000	0.000658
35	880000000.000000	0.000000	0.000000
36	882000000.000000	884000000.000000	0.000329
37	884000000.000000	886000000.000000	0.000000
38	888000000.000000	890000000.000000	0.000597

Figure 4.9: Real Time Sensor Display

Chapter 5

Conclusion

The current system accomplishes the basic functionalities that an SAS is expected to archive.

- It has an interconnected system of sensors that can detect the PU
- A central engine that is able to protect the PU and inform the SUs within 6s, which is lower than current expected threshold of 60s. It also outperforms [19] which provided latency of greater than 10s.
- An interconnected network between the SAS and the CBSDs to ensure that SAS-CBSD connection is robust

In conclusion, this research system is currently a minimum viable product which offers a basis through which more channel allocation algorithms and machine learning can be utilized. As shown in the results, the infrastructure works and produces good preliminary results that are better than what is currently stipulated[7] which is a latency of 60s. The instructions provided in this research show that it can be easily deployed over a large license area with existing INTERNET as the back-haul between the CBSDs and the SAS for that particular area.

Chapter 6

Future Work

1. Network Load Balancing and Performance: The total CBSDs that can communicate with the SAS is hugely dependent on available network and computational resources. Therefore, no formal results would be obtained with this performance. Using Apache JMeter, the performance can be calculated. Instructions on how to utilize Jmeter to get performance analytics are mentioned below:

   (a) Download Apache Jmeter from https://jmeter.apache.org/

   (b) Run Jmeter using:

   ```
   ./jmeter.sh
   ```

   (c) Use the following start.xml file as a basis for the creating a virtual CBSD that requests channels from the SAS. It is found in the jmeter folder from https://github.com/ShemK/O Source-Spectrum-Access-System.git

   (d) Open the xml file and make appropriate changes to the JSON request files

2. Local REM: There is a database at each node, and each node has the same software. Therefore, there is a possibility that CBSD could potentially work as sensors in durations in which they are not requesting the channel. Then decisions can be made by the CBSD before getting confirmation from the SAS. In such a scenario, PU detection is shared among the CBSDs, and the sensors. Furthermore, during transmission, the

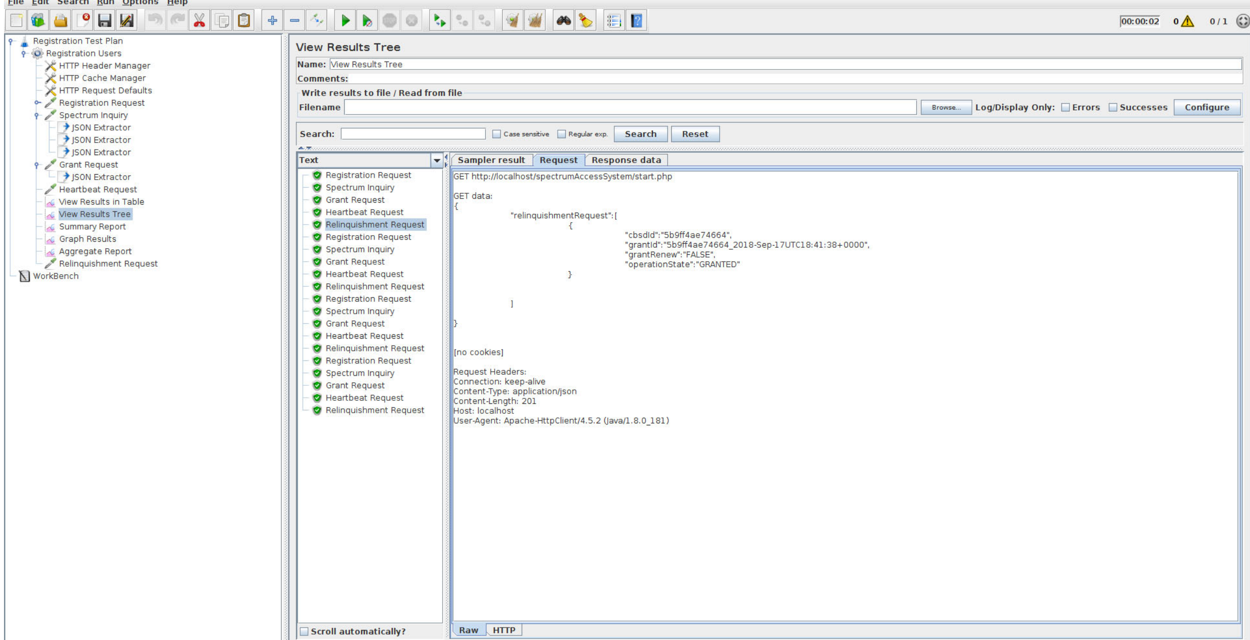


Figure 6.1: Jmeter Application

CBSD which are utilizing SDRs, have the ability for a full-duplex system, which provides an avenue for CBSDs to actively detect a PU or even more viable channels. As shown in figure 2.4, all that is required is for the CBSD to write raw IQ received data to the shared memory of the sensor process.

3. SAS-SAS Communication: The need for interface between SAS from different areas would provide an ability to reduce interference among CBSDs stationed at the edges.

4. Code Modification: The appendix section provides important information on the different modules that other researchers can modify to get there own system up and running. The REM can be modified by adding more tables to it, to accomplish any other attribute that is needed. The SAS Engine would also be modified to work on the information that has been added to the database. The structure is shown in the appendix.

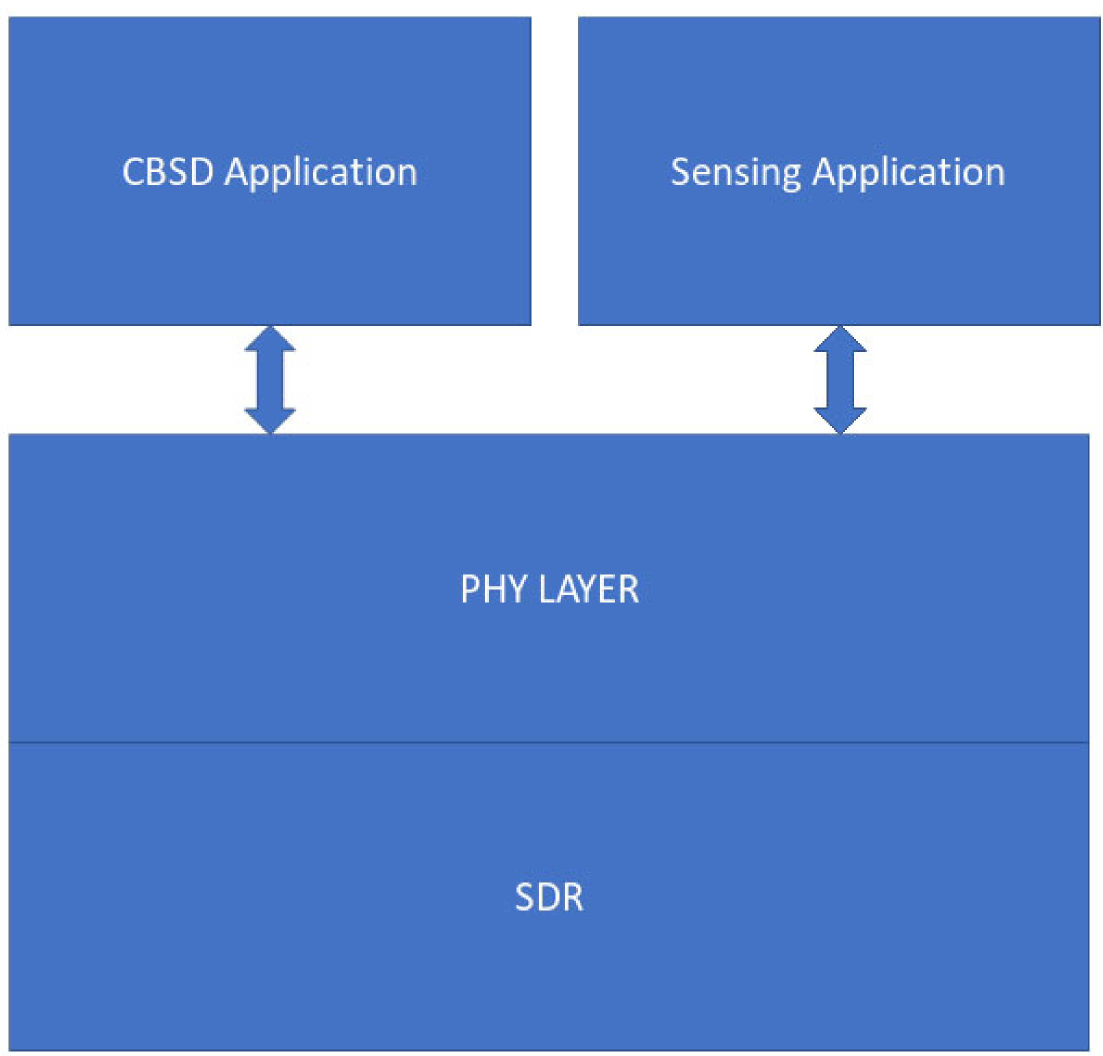


Figure 6.2: Dual CBSD and Sensor

Bibliography

[1] Apache license. URL https://httpd.apache.org/docs/2.4/license.html.

[2] The php license. URL http://www.php.net/license/3_01.txt.

[3] LLC AdjacentLink. Extendable mobile ad-hoc network emulator (emane). Internet: https:// github. com/ adjacentlink/ emane,[January 2, 2016].

[4] National Marine Electronics Association et al. NMEA 0183–Standard for interfacing marine electronic devices. NMEA, 2002.

[5] Daniel P Bovet and Marco Cesati. Understanding the Linux Kernel: from I/ O ports to process management. " O'Reilly Media, Inc.", 2005.

[6] Tim Bray. The javascript object notation (json) data interchange format. Technical report, 2017.

[7] FC Commission et al. In the matter of amendment of the commission's rules with regard to commercial operations in the 3550-3650 mhz band: Further notice of proposed rulemaking, 2014.

[8] Daniel DePoy. Cognitive radio network testbed (cornet): Design, deployment, administration and examples. In Masters Thesis, Virginia Polytechnic Institute and State University, 2012.

[9] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/ 1.1. Technical report, 1999.

[10] Joseph D. Gaeddert. Liquid-dsp. URL http://liquidsdr.org/.

[11] RFnest User Manual. Intelligent Automation,Inc.

[12] RFnest Product Specifications. Intelligent Automation,Inc, 2017.

[13] Shem Kikamaze, Vuk Marojevic, and Carl Dietrich. Spectrum access system on cognitive radio network testbed. In Proceedings of the 11th Workshop on Wireless Network Testbeds, Experimental evaluation & CHaracterization, pages 99–100. ACM, 2017.

[14] Chang Wook Kim, Jihoon Ryoo, and Milind M Buddhikot. Design and implementation of an end-to-end architecture for 3.5 ghz shared spectrum. In Dynamic Spectrum Access Networks (DySPAN), 2015 IEEE International Symposium on, pages 23–34. IEEE, 2015.

[15] Maxim Krasnyansky and Maksim Yevmenkin. Universal tun/ tap device driver. URL: http:/ / www. kernel. org/ pub/ linux/ kernel/ , FILE: people/ marcelo/ linux-2.4/ Documentation/ networking/ tuntap. txt, 2007.

[16] Mitchell Lazarus. Fcc names spectrum access administrators, Dec 2016. URL https://www.commlawblog.com/2016/12/articles/unlicensed-operations-and-emerging-technologies/fcc-names-spectrum-access-administrators/.

[17] Wes McKinney. pandas: a foundational python library for data analysis and statistics. Python for High Performance and Scientific Computing, pages 1–9, 2011.

[18] Travis E Oliphant. A guide to NumPy, volume 1. Trelgol Publishing USA, 2006.

[19] Marko Palola, Marko Höyhtyä, Pekka Aho, Miia Mustonen, Tero Kippola, Marjo Heikkilä, Seppo Yrjölä, Vesa Hartikainen, Lucia Tudose, Arto Kivinen, et al. Field trial of the 3.5 ghz citizens broadband radio service governed by a spectrum access sys-

tem (sas). In Dynamic Spectrum Access Networks (DySPAN), 2017 IEEE International Symposium on, pages 1–9. IEEE, 2017.

[20] Stefanos Papadakis and Apostolos Traganitis. Wireless positioning using the signal strength difference on arrival. In Mobile Adhoc and Sensor Systems (MASS), 2010 IEEE 7th International Conference on, pages 674–681. IEEE, 2010.

[21] GNU Radio. Home page. URL http://www.gnuradio.org/.

[22] Eric Raymond et al. gpsd- a gps service daemon. 2005.

[23] Munawwar M Sohul, Miao Yao, Xiaofu Ma, Eyosias Y Imana, Vuk Marojevic, and Jeffrey H Reed. Next generation public safety networks: a spectrum sharing approach. IEEE Communications Magazine, 54(3):30–36, 2016.

[24] Eric Sollenberger, Ferdinando Romano, and Carl Dietrich. Test & evaluation of cognitive and dynamic spectrum access radios using the cognitive radio test system. In Vehicular Technology Conference (VTC Fall), 2015 IEEE 82nd, pages 1–2. IEEE, 2015.

[25] Dual Channel 14-/12-Bit, 250-/210-MSPS ADC With DDR LVDS and Parallel CMOS Outputs. Texas Instruments.

[26] John Viega, Matt Messier, and Pravir Chandra. Network security with openSSL: cryptography for secure communications. " O'Reilly Media, Inc.", 2002.

[27] WinnForum. Requirements for commercial operation in the u.s. 3550–3700 mhz citizens broadband radio service band, 2016.

[28] Justin Yackoski, Babak Azimi-Sadjadi, Ali Namazi, Jason H Li, Yalin Sagduyu, and Renato Levy. Rfnest™: Radio frequency network emulator simulator tool. In Military Communications Conference, 2011-MILCOM 2011, pages 1882–1887. IEEE, 2011.

[29] H Birkan Yilmaz, Tuna Tugcu, Fatih Alagoz, and Suzan Bayhan. Radio environment map as enabler for practical cognitive radio networks. IEEE Communications Magazine, 51(12):162–169, 2013.

[30] Tevfik Yucek and Huseyin Arslan. A survey of spectrum sensing algorithms for cognitive radio applications. IEEE communications surveys & tutorials, 11(1):116–130, 2009.

[31] Xiuyan Zhu and Yuan Feng. Rssi-based algorithm for indoor localization. Communications and Network, 5(02):37, 2013.

[32] Josh Zimmerman and Jonathan Clark. Unit testing. Principles of Imperative Computation, 2012.

Appendices

Appendix A

Radio Environment Map

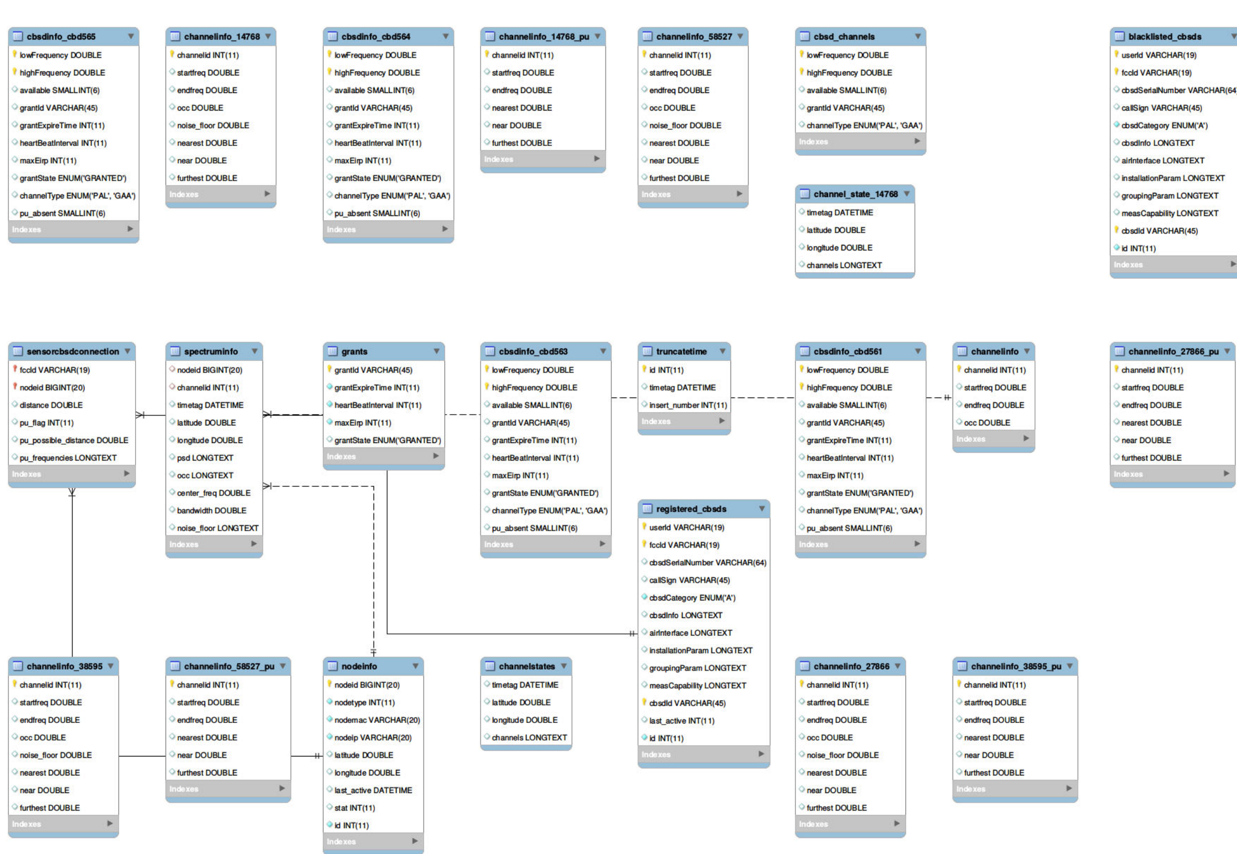


Figure A.1: Entity Relationship Diagram of the REM

A.1 Sensor Information

At the central REM, all information from each sensor is stored in tables called channelinfo_id where id is the node identification of the sensor. All information from the Central Sensor Aggregator is stored in each of these tables. These tables only store current information from the sensor.

When a new sensor is connected to the network, the REM has an internal trigger that automatically creates the table for that particular sensor. To ensure that each sensor has its own unique id, the mac address of network interface is used. Each sensor id is the hash value of the mac address. The location and node information is stored in the nodeinfo table. The SAS Cognitive Engine would be required to constantly read these tables to obtain sensor information. After extrapolating the distance between the sensor and the incumbent, such information is stored in the channelinfo_id_pu table. The sensor information and the calculated information are not grouped in the same table because two programs (Central Sensor Aggregator and SAS Cognitive Engine) cannot write to the same table without the introduction of deadlocks.

A.2 CBSD Information

CBSD information is also stored in its own tables that are isolated from the sensor tables, except with the sensorcbsdconnection table. sensorcbsdconnection is a table that is made to measure the difference in physical attributes between each sensor and each CBSD. In its current state, the table measures the distance between each CBSD and each sensor.

Each CBSD will have the cbsdinfo_id where id is the FCC identification for that CBSD. Any grant information will be stored in that table for that particular CBSD. This is the

table that the SENSOR-CBSD interface uses when allocating channels to a CBSD. There is a flag in each CBSD table that is updated by the SAS Cognitive Engine to show whether a PU communication will be affected by that CBSD.

Appendix B

SAS-CBSD Interface

The SAS-CBSD interface contains two PHP classes. The json_ listener.class.php object is the interface that listens and replies to incoming JSON requests from the CBSD. Each function is made for handle each of the requests. This class depends on either the serverpsql.class.php for connection to the PSQL database or the serversql.class.php if an SQL database is used instead as shown in Figure B.2.

The communication between the SAS and CBSD is shown in Figure B.1. The SAS-CBSD interface then inquires the REM for any the response needed by the CBSD through its serverpsql.class.php.

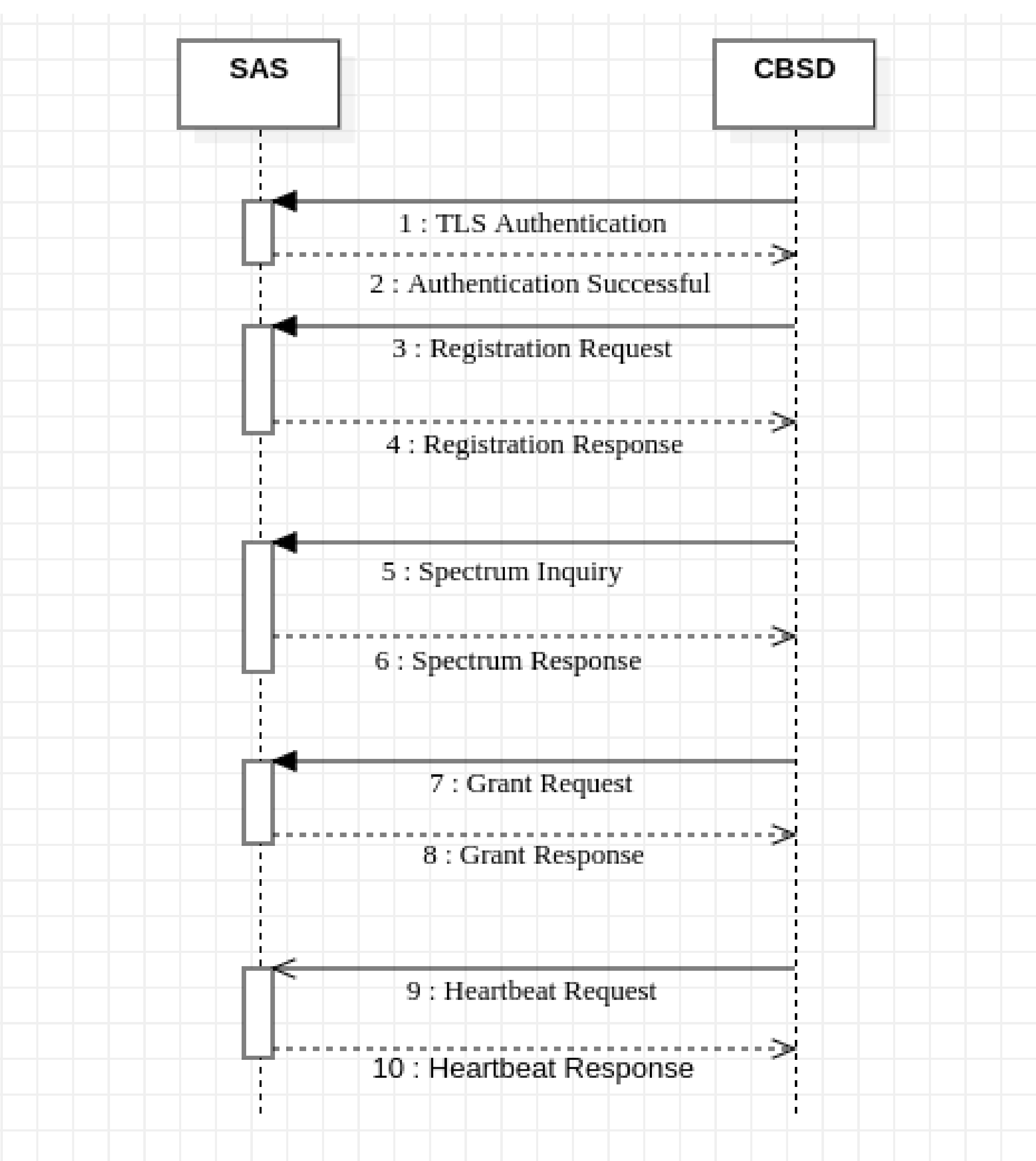


Figure B.1: Request-Response Communication Sequence

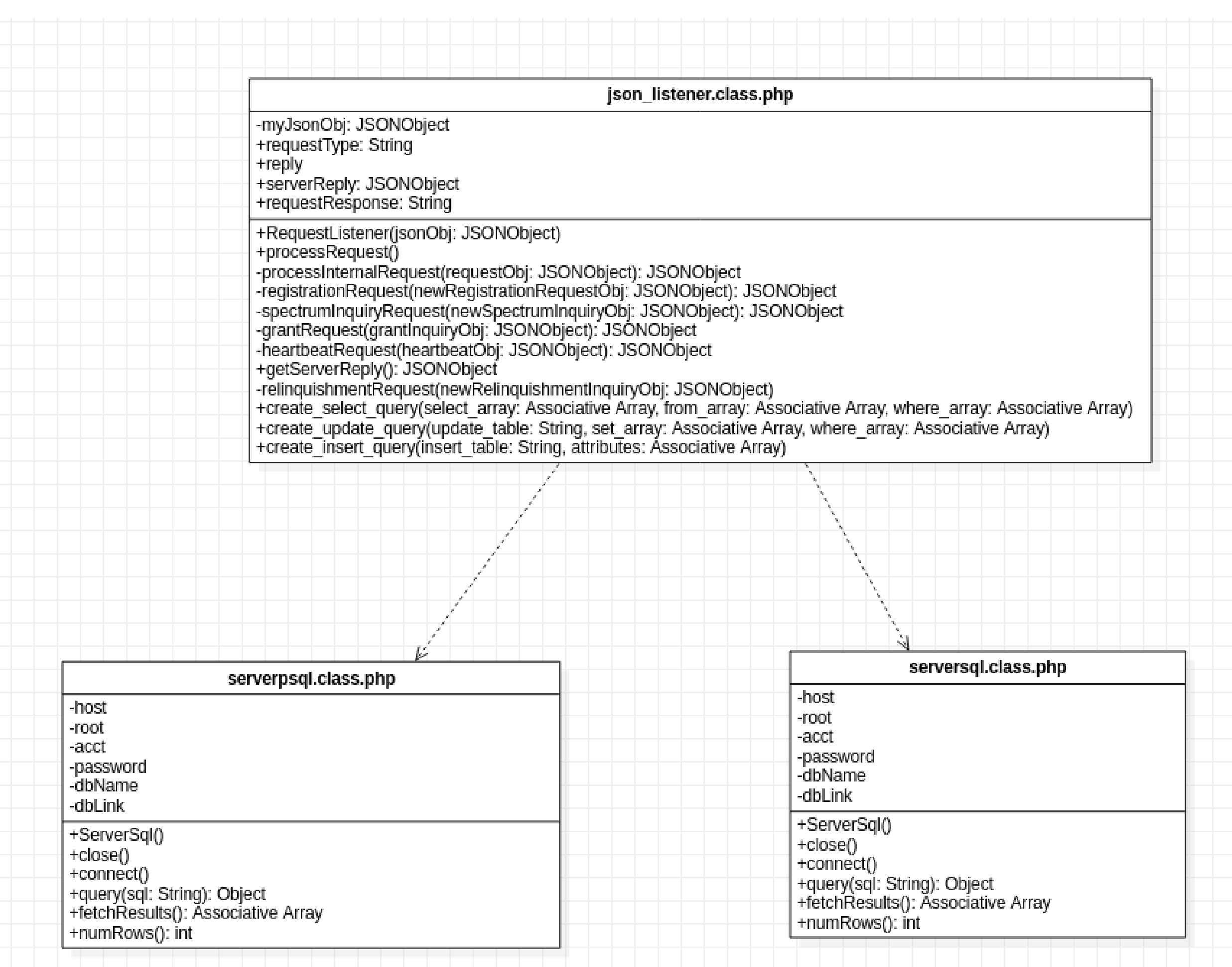


Figure B.2: Class Diagram for the SAS-CBSD Interface

Appendix C

SAS Engine

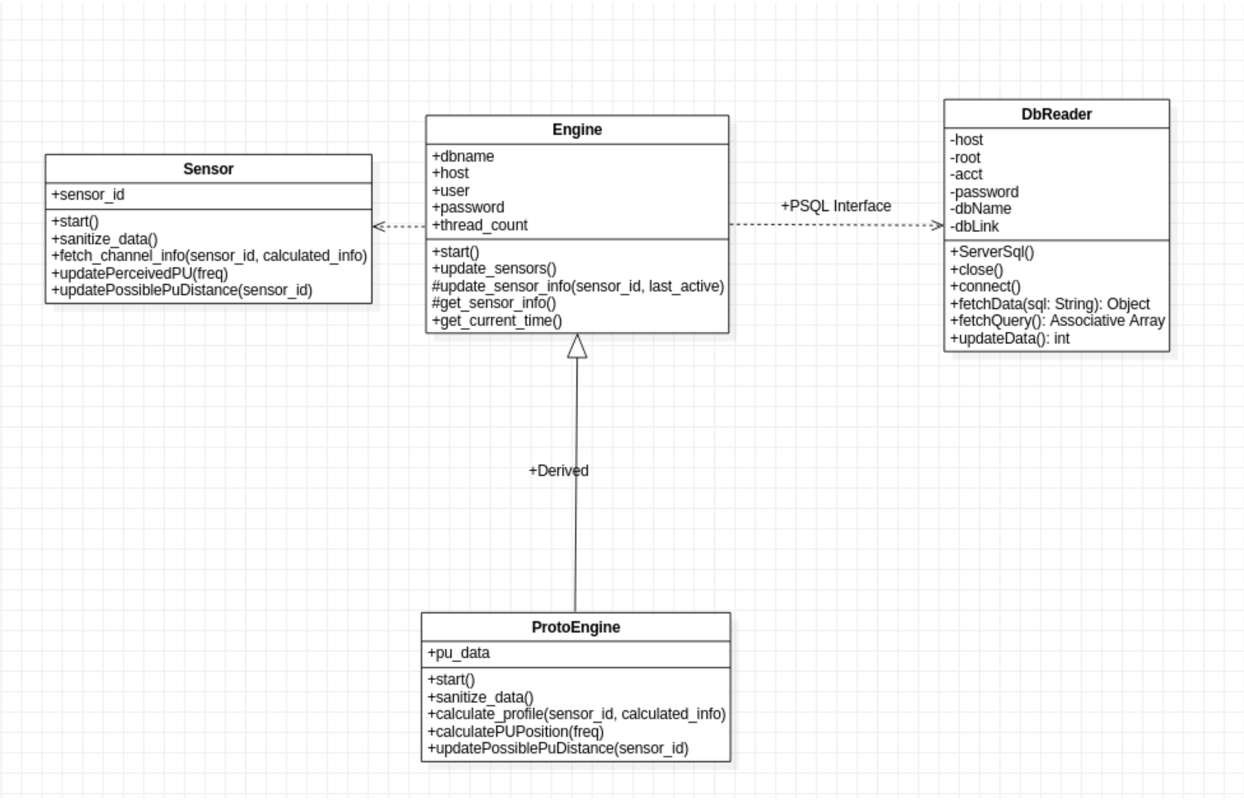


Figure C.1: Class Diagram for the SAS Engine

The SAS Engine contains the Engine which is the base class as explained in 2.2.3. The ProtoEngine is derived from Engine, and researchers are expected to derive future classes as exemplified. The DbReader is the interface that is used to read from the database and write to it.

Appendix D

Software Interconnectivity

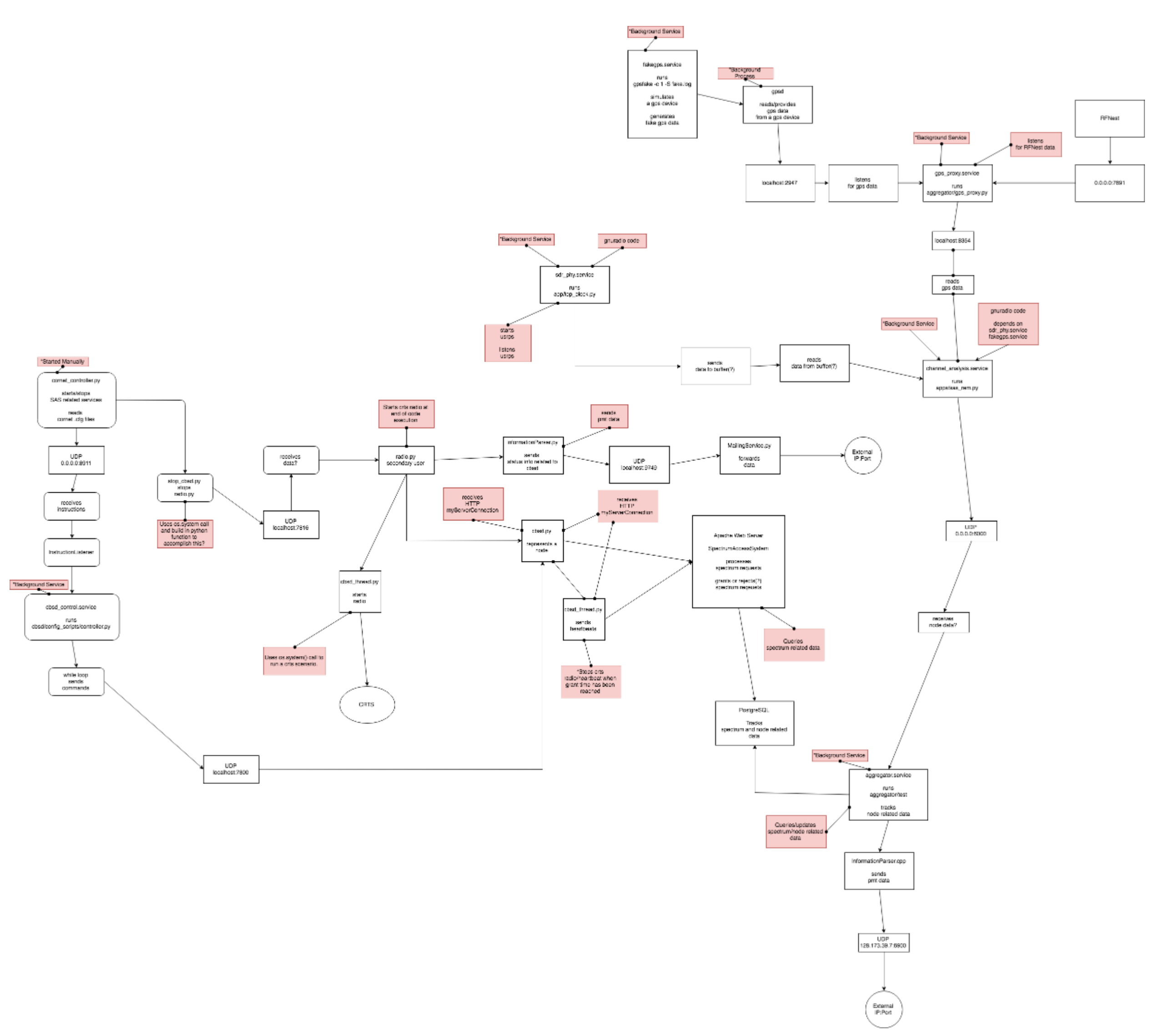


Figure D.1: Current Software Interconnection