Cloud-based 5G Spectrum Observatory

A Major Qualifying Project submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science

Submitted by:
Ian Casciola
Electrical and Computer Engineering

Sreeshti Chuke
Electrical and Computer Engineering

Aneela Haider
Electrical and Computer Engineering

Kevin Mbogo
Electrical and Computer Engineering

Joseph Murphy
Electrical and Computer Engineering

Project Advisor:
Alexander M. Wyglinski
Electrical and Computer Engineering
Worcester Polytechnic Institute

Submitted on: April 4, 2021

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review.

Abstract

The goal of this project was to build a low-cost, cloud-based spectrum observatory capable of monitoring multiple 5G frequency bands in real-time with the ability to remotely change the parameters of the radios during runtime. The observatory was implemented using GNU Radio, Python, JavaScript, and Node.js as well as multiple software-defined radios. The approach was successful, and the spectrum observatory serves as a good proof-of-concept for future works.

Acknowledgements

This project owes its success to the groups and individuals who helped by providing their time, knowledge, and resources to us. We would like to thank National Instruments and the Air Force Research Laboratory for the radios that were given for the completion of this project, as well as the Academic & Research Computing department at WPI for providing the hardware for the server. We want to thank Professor Wyglinski, our faculty advisor, for all the guidance he provided our team, and for the time he took out of his schedule to aid us each week. We want to thank Kuldeep Gill, a PhD student working under Professor Wyglinski for all the help he provided with getting the radios working, an extremely important part of our project. We would like to thank Adonay Resom, a Computer Science major at WPI, for his aid in the development of the server backend. Finally, we would like to Eleanor O'Neill, an industry professional web developer for her advice and help on the creation of the website.

Executive Summary

The development of a spectrum observatory is critical to communications as it provides a variety of functions. A 5G spectrum observatory allows anyone to monitor 5G frequency bands and access the data to make decisions for their task. Most observatories are designed for professional researchers and engineers and require expensive, high-end equipment [1]. There is a need for a smaller scale spectrum observatory that is inexpensive and can provide control of various parameters while displaying real-time results for anyone to access. The project has created a proof-of-concept implementation of a low-cost spectrum observatory that allows users to remotely view spectrum data and remotely change the operational parameters of the software-defined radios sampling.

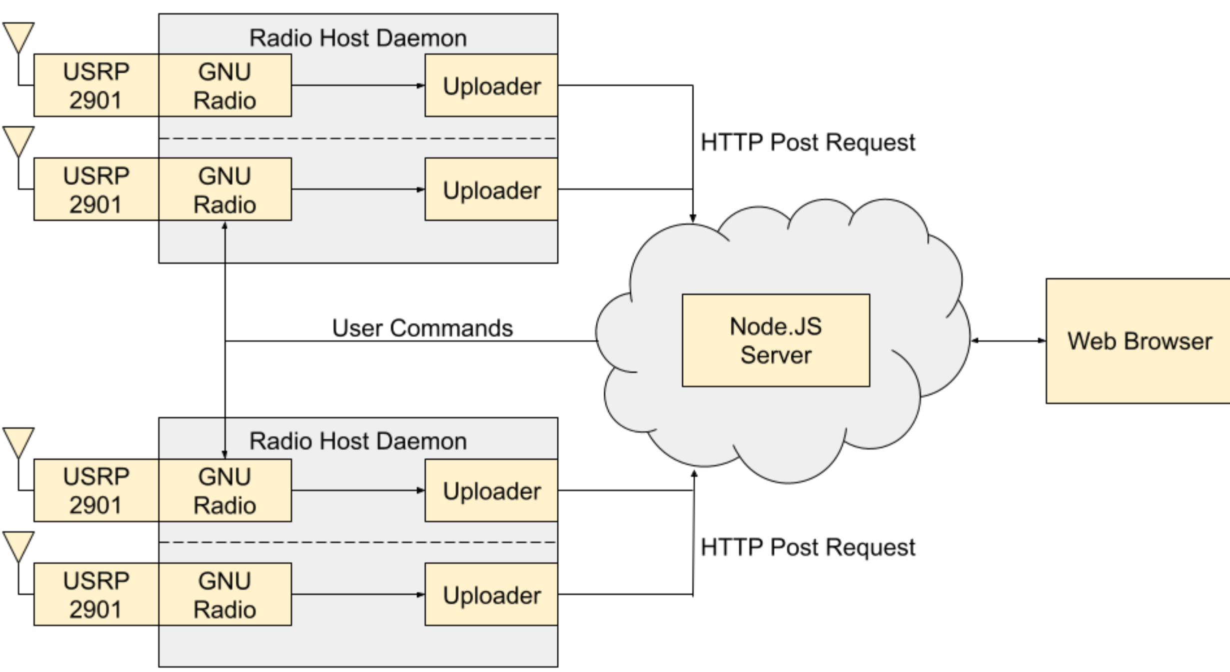

Figure ES.1: Block Diagram of Design Overview

Six different 5G frequency bands were evaluated to have a greater range of 5G data to speculate for busy frequencies or available frequencies. The spectrum data was tested using two different radios, each designated to a different 5G frequency band. The testbed was built with Ubuntu 18.04 using a USRP 2901 on a headless host computer. GNU Radio was used to sample

the data at each requested frequency band from the user. This flowgraph sampled data at 300,000 samples per second to maintain a minimal output data size.

The GNU Radio data was uploaded to the cloud server using a custom script that encoded and formatted the data before upload. A daemon control program paired the GNU Radio instance and the uploader script so that a single program had the ability to control multiple radios. The radio host program parsed two input arguments, one argument determined the number of GNU Radio instances to run, and the other argument was a string indicating a command for the daemon to start running, stop, or restart.

The server was originally implemented using the WebSocket protocol. With this type of implementation, the server begins by waiting for an event to fire, indicating the radio host has connected. The socket would then be opened and allow for data to be sent and received, and the server began parsing the data as it came in. The headers were separated from the data payload, and the data was converted from Base64 to binary so it could be arranged in a Comma Separated Values (CSV) file format. This approach worked well for a single radio but not for multiple. Socket implementation with several radios resulted in the server greatly slowing down. As a result, Hypertext Transfer Protocol (HTTP) was used in the final server implementation. This type of implementation worked better because the server was designed to have a specific address for data coming in from different radios. Some changes to data processing were made to increase the performance of the server, such as implementing worker threads and splitting the data across several files. After the data was converted to a CSV file, the files were stored in separate folders, with one designated for each radio.

The website used AJAX or "Asynchronous JavaScript and XML" to access the folders containing the CSV files, and using this data, calculated the magnitude of each pair of numbers using the magnitude equation. Once the calculations were complete, the spectrum data was graphed as an amplitude plot. Figure ES.2 is an example of this.

The final part of the project was implementing the ability to send commands backward, from the website to the radio host. The reason for the bi-directional data flow was to allow for the remote control of radio parameters, like center frequency, during run time. This was achieved by implementing an HTML form with two frequencies to choose from for each radio. This form would use an HTTP POST request to send the user's selected frequency value through the web interface to the server. The radio host computer used an HTTP GET request to receive these instructions

and change the radio parameters accordingly. This resulted in an easy and effective implementation to let the user change the center frequency of the radio remotely through a web interface.

Once the prototype was functional from end-to-end, the performance of this spectrum observatory was measured. It was noted that there was a delay in each step of the implementation, with the biggest delay occurring between data being stored on the server after being processed and data being displayed on the webpage. This delay kept compounding as more files were added to the server and the number of required computations to create the spectrograms increased. As more data was added, the time between the collection of data and displaying on the website went from less than five seconds to more than a minute. Since the time delay is very dependent on the speed of the internet and the machine displaying the webpage, the exact latency measurement varies. Finally, to check the accuracy of the data, a comparison was made between data being displayed on the website and data being displayed in QT GUI sink of GNU Radio. Similarities were noted between the visualization of data in both cases, even though the delay in end-to-end implementation made the comparison a challenge.

Overall, this project provided a comprehensive overview and proof-of-concept of a lower cost, cloud-based spectrum observatory. With more research and development being completed to expand the system, this implementation could be an excellent model for other lower-cost spectrum observatories.

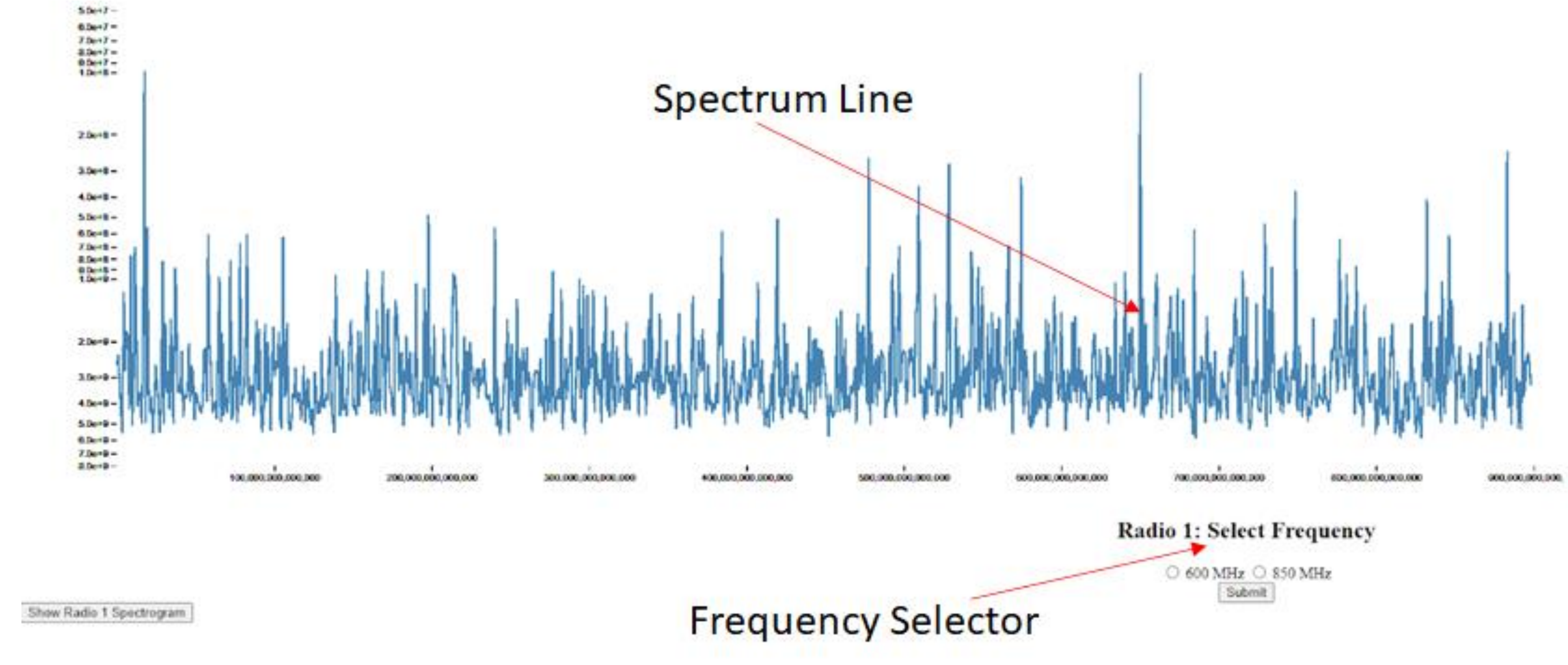

Figure ES.2: Web Display of Spectrum Data

Authorship Page

While all members of this team contributed to both the implementation of the spectrum observatory and writing of this report, different members had different focuses throughout the project. For a more in-depth breakdown of who contributed to which pieces of the project, refer to Appendix A.

Table of Contents

Abstract ii

Acknowledgements iii

Executive Summary iv

Authorship Page viii

Table of Contents ix

List of Figures xii

List of Tables 13

List of Acronyms xv

1. Introduction 1

- 1.1 Motivation 2
- 1.2 Current State of the Art 2
- 1.3 Technical Challenges 5
- 1.4 Contributions 6
- 1.5 Report Organization 7

2. Overview of Spectrum Sensing 8

- 2.1 Introduction of Frequency Spectrum 8
- 2.2 Introduction to Fifth Generation Technology 10
- 2.3 Software-Defined Radios 11
- 2.4 Types of Spectrum Display 14
- 2.5 Software Daemons and Program Control 17
- 2.6 Cloud Servers 18
- 2.7 Chapter Summary 22

3. Proposed Implementation 23

- 3.1 Problem Statement 23
- 3.2 Proposed Approach 23
- 3.3 Metrics for Success 23
- 3.4 Design Overview 24
  - 3.4.1 GNU Radio 27
  - 3.4.2 Cloud Infrastructure 28
  - 3.4.3 Cloud Server Software 30
- 3.5 Project Planning 31
- 3.6 Technical Deliverables 36

3.7 Chapter Summary 36

4. Sampling Frequency Spectrum 37
4.1 Configuring GNU Radio on a Headless System 41
4.2 Implementation of GNU Radio 38
4.3 Challenges 41
4.4 Chapter Summary 41

5. Program Control and Data Upload 42
5.1 Radio Host Control 42
5.2 Unix Daemon Implementation 44
5.3 Data Encoding and Upload 45
5.4 Chapter Summary 46

6. Cloud Infrastructure 47
6.1 Implementation 47
6.1.1 Socket implementation 47
6.1.2 HTTP Implementation 49
6.1.3 Data Processing and the Event Loop 51
6.2 Chapter Summary 54

7. Spectrogram Display 55
7.1 Implementation Overview 55
7.2 Displaying Data 57
7.3 Challenges 59
7.4 Chapter Summary 59

8. Remote Control of Radio Parameters 60
8.1 HTML Forms 60
8.2 HTTP Method GET/POST 61
8.3 GNU Radio Python Script 63
8.4 Chapter Summary 64

9. Results and Discussion 65
9.1 Data Transfer from Radio to Website 70
9.2 Frequency Change Request 70
9.3 Chapter Summary 70

10. Conclusion 71
10.1 Future Work 71

References 72

Appendix A: Project Authorship 78

Appendix B: Tutorial for Setting up Radio Host 80

Appendix C: Tutorial for Setting up Web Server 81

Appendix D: RadioHost.py Source Code 82

Appendix E: top_block.py Source Code 88

Appendix F: daemon.py Source Code 92

Appendix G: transer.py Source Code 96

Appendix H: server.js Source Code 100

Appendix I: worker.js Source Code 107

Appendix J: index.html Source Code 110

List of Figures

Figure ES.1: *Block Diagram of Design Overview* iv

Figure 1.1: *Illinois Institute of Technology Defunct Spectrum Observatory. From [5]* 1

Figure 1.2: *Spectrum Observatory concept diagram of the operations between the sensors and central server* 3

Figure 1.3: *Illinois Institute of Technology spectrum observatory. From [10][11]* 4

Figure 1.4: *FuNLab Spectrum Observatory. From [13]* 5

Figure 2.1: *FCC Frequency Allocations. From [17]* 9

Figure 2.2: *Figure 2.2: 5G Slicing architecture with NFV. From [25]* 11

Figure 2.3: *Block Diagram of SDR's Transmission Functionality. Adapted from [28]* 12

Figure 2.4: *Block Diagram of SDR's Receiving Functionality. Adapted from [28]* 13

Figure 2.5: *GNU Radio Flowgraph Example [30]* 14

Figure 2.6: *Waterfall Plot. From [32]* 15

Figure 2.7: *Amplitude Plot* 15

Figure 2.8: *An example of the functional layout of a Cloud system. Adapted from [43]* 19

Figure 2.9: *A comparison between a classic server and a virtualized server installation.* 20

Figure 2.10: *An outline of the services and tools provided by the three main cloud computing service models offered by many cloud hosting companies. Adapted from [43].* 21

Figure 3.1: *Block Diagram of Design Overview* 26

Figure 3.2: *Block Diagram of GNU Radio Data Sampling and Processing* 28

Figure 3.3: *Team Gantt Charts* 33

Figure 4.1: *Hardware Setup of Testbed in East Hall (Radio Host 1)* 38
Figure 4.2: *Flowgraph of Spectrum Analyzer* 39
Figure 5.1: *Overview of Processes and Threads* 42
Figure 5.2: *A sample of the Radio Daemon's log file* 43
Figure 5.3: *Data Upload Script Flow Diagram* 45
Figure 6.1: *A snippet of the code used to create the server along with creating the socket.* 48
Figure 6.2: *Server-side Data Processing* 48
Figure 6.3: *HTTP implementation of server using switch statements.* 50
Figure 6.4: *Web Server Processing as data is transmitted and received by the server.* 52
Figure 6.5: *Code sample of the function responsible for creating a worker thread.* 53
Figure 7.1: *Spectrogram Display on Website* 56
Figure 8.1: *HTML form output* 61
Figure 8.2: *Flow of Data using HTTP Methods.* 62
Figure 8.3: *Generation of python script top_block.py* 63
Figure 9.1: *Time to Draw Data per File* 66
Figure 9.2: *Change in Time to Open Files* 66
Figure 9.3: *Sample Spectrum Graph from the Website* 68
Figure 9.4: *Spectrum Graph from GNU Radio* 68
Figure 9.5: *Flow of the Frequency Change Request* 69

List of Tables

Table 1.1: *Specifications of other Spectrum Observatories* 5
Table 2.1: *Licensed Frequency Bands by Company* 8
Table 2.2: *Actions taken to Create Unix Daemon* 18
Table 2.3: *Pros and Cons of an On-Premise Server vs. a Cloud Server* 20
Table 3.1: *Advantages and disadvantages of signal sampling on GNU Radio* 27
Table 3.2: *Comparison between different commercial Cloud platforms* 29
Table 3.3: *Comparison of Node.JS versus Apache* 31
Table 4.1: *GRC Block Descriptions* 40
Table 8.1: *Comparison of HTTP GET vs. POST* 62

List of Acronyms

Abbreviation	Term
5G	Fifth Generation Cellular Networks
ADC	Analog to Digital Converter
AJAX	Asynchronous JavaScript and XML
AWS	Amazon Web Services
CSV	Comma Separated Value
DDC	Digital Down Converter
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
GRC	GNU Radio Companion
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IF	Intermediate Frequency
IoT	Internet of Things
PaaS	Platform as a Service
PID	Process ID
RF	Radio Frequency
SaaS	Software as a Service
SDR	Software-defined Radio
VM	Virtual Machine

1. Introduction

This chapter explains the significance of spectrum sensing and spectrum display utilizing a cloud-based system, the presence and impact of spectrum observatories in society currently, and the current development in data collection and display methods. This chapter ends by narrating this project's contribution to furthering spectrum observatory technologies.

1.1 Motivation

5G is a brand new technology still in its infancy, and yet it is making huge waves in industry with companies like Verizon or AT&T purchasing mid-band spectrums for over $40 billion each [4]. Throughout the lifetime of 4G wireless, there have been many projects [3][4][5] that would capture wireless signals across the 4G spectrum using one or more radios and display the spectrum on a website. These cloud-based spectrograms allow any internet user to view and analyze data being collected by any number of radios across the world in numerous cities, but many are now defunct and no longer operate. Figure 1.1 shows an example of one of the spectrograms that has since gone defunct. Its two axis, frequency and power, show it is a periodogram, but data is no longer being collected and is therefore not being displayed.

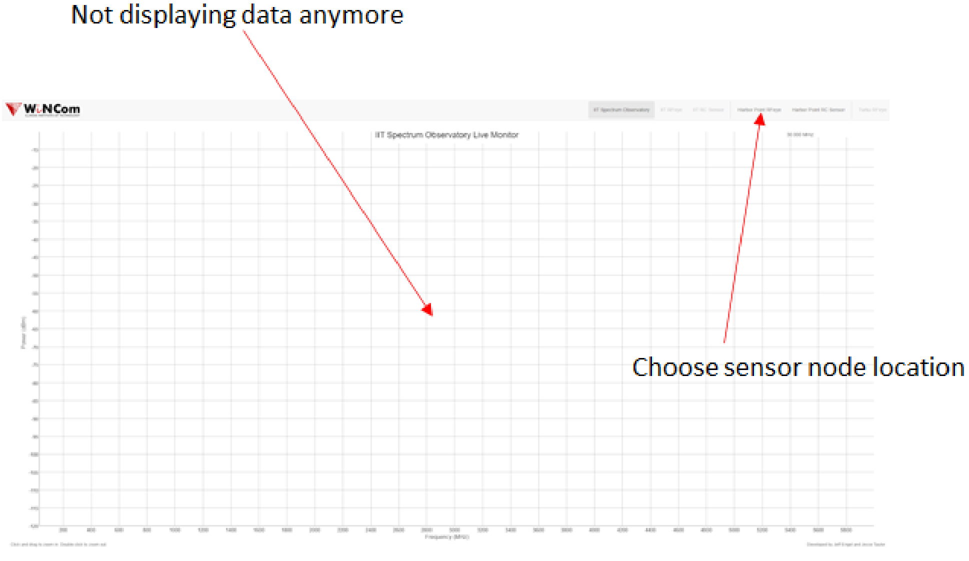

Figure 1.1: Illinois Institute of Technology Defunct Spectrum Observatory. From [5]

This project's goal is to accomplish the same functionality as the 4G spectrum observatories but in the 5G band. The possible uses for this technology include civilian hobbyists, researchers, and even national defense [6]. A 5G spectrum observatory allows interested parties to monitor the 5G bands and make decisions based on the data they have access to. A hobbyist interested in setting up an IoT device may access a 5G spectrum observatory to see the frequencies that are being commonly used, or see the frequencies they need to avoid. A researcher may use the technology to locate frequency hotspots or investigate the properties of a 5G antenna outputting energy in a band that is being monitored by the observatory. National defense uses include securing locations where VIPs need to travel, locating insurgent cells communicating on 5G bands, or finding the frequency that needs to be jammed to force a missile to fly off course from a friendly aircraft [6]. On a smaller scale, a spectrogram can be used to find local WiFi or Bluetooth frequencies that are being jammed by lower energy jammers [7][8].

The possible uses of spectrum observatories are wide, and the choice to make it cloud-based is purposeful. Allowing anybody in the world to access this technology, analyzing frequency data, monitoring captured signals, and changing the operating frequencies on the radios, allows this project to be accessible by those who normally cannot afford it. Opening the 5G band for use by anyone and everyone levels the technology playing field between professional researchers and developers from Silicon Valley tech companies with the everyday engineers who wish to develop something unique and personal. Costs for a spectrum observatory are high, the hardware alone costing around $5000 for a medium tier radio and computer to run it, not even taking into account the development costs and server costs [1]. Providing this technology at a lower cost, and on a more widely accessible scale allows any individual to become invested in the spectrum world.

1.2 Current State of the Art

Several spectrum observatories have been implemented to help conduct research that requires constant spectrum sensing and data collection at a large scale. Spectrum observatories are technologies deployed to capture and display spectrum data of a specific time or throughout a specific interval of time. These observatories were designed to overcome some of the challenges of developing and analyzing new wireless technologies. Careful handling is required to ensure the hardware and software of such spectrum observatories work properly to output correct data [5].

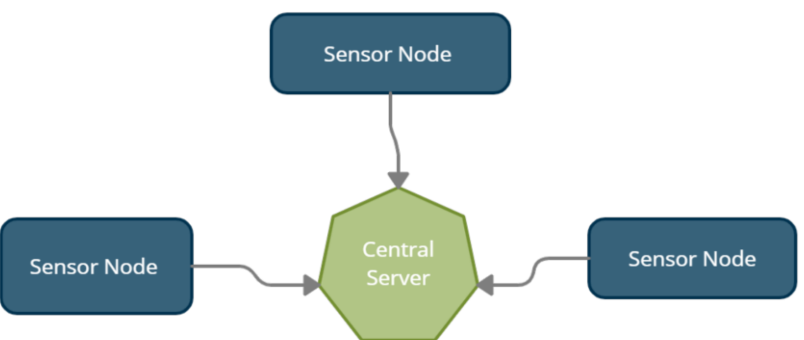

Figure 1.2 Spectrum Observatory concept diagram of the operations between the sensors and central server

Figure 1.2 illustrates the two main components of such spectrum sensing observatories are spectrum sensor nodes and a central server [5]. A spectrum sensor node typically consists of a software-defined radio (SDR). An SDR is a radio communication system used for modulation and demodulation of radio signals. The SDR is programmed to monitor a specific channel in the frequency spectrum at a certain time and collect data. This data is then sent to the server when it is stored and integrated to generate a plot displayed on a user interface such as a website. Figure 1.2 shows the basic concept of a spectrum observatory with the sensor nodes feeding information to the central server that computes and displays the data.

The Illinois Institute of Technology (IIT) created a spectrum observatory to assess the radio frequency (RF) spectrum utilization accurately in downtown Chicago. This spectrum observatory's main objective was to encourage the use of the spectrum more efficiently [9]. SDRs covering a wideband of 30 MHz to 6 GHz were used as sensor nodes in this project. Due to the large amounts of generated data from these sensor nodes, several different data storage solutions were evaluated and utilized to store the RF measurement data. Figure 1.3 represents the building where Illinois Institute of Technology housed the spectrum observatory along with the results they achieved.

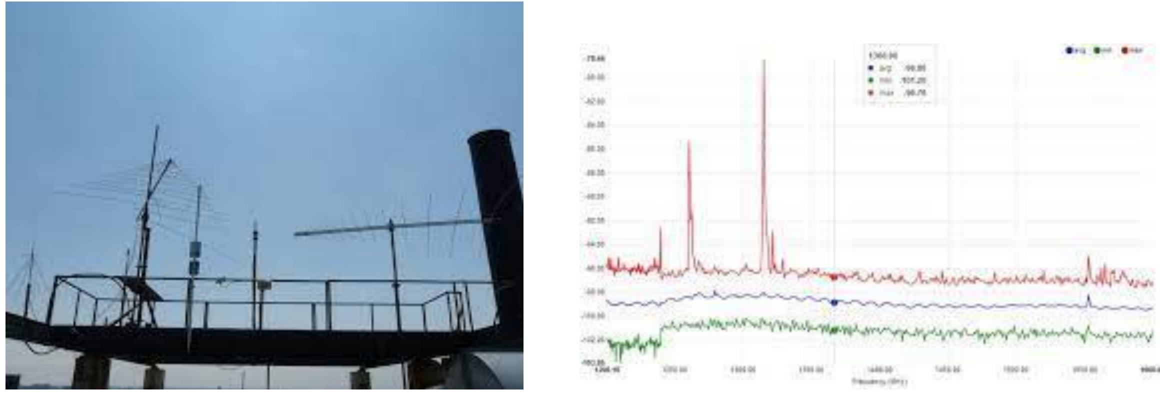

Figure 1.3 Illinois Institute of Technology spectrum observatory. From [10][11]

Microsoft created a spectrum observatory to map the number of active transmitters, their frequency, and their temporal characteristics to detect both rogue transmitters and opportunities for radio frequency access. Microsoft used advanced and wideband spectrum analyzers covering measurements from 50MHz to 2.5 GHz at different resolution bandwidths and housed the data using their own Azure database system [1]. This system required a co-located PC that processed the data and created summaries before uploading them to the Cloud [1].

FuNLab of the University of Washington developed a system named SpecObs that used a geolocation database and spectrum sensing with the intention to improve white space prediction in the frequency spectrum. The RF sensor nodes scanned certain frequencies periodically and then uploaded the data to a server. This sensing data was tagged with metadata such as a timestamp and locational data before being uploaded. The cloud server managed the storage and processing of data and calculated statistical characteristics white space availability [5]. Secondary users utilized this model to estimate channel availability and determine when the primary user did not occupy the channel. A primary user, in this case, refers to individuals or organizations to whom the channel is legally licensed [12]. Radios can be deployed at multiple locations to cover a particular geographical area in such a type of spectrum observatory [12]. With the above systems seemingly not currently available for public access, the project’s goal is to expand on these current systems and determine ways to lower the cost of deploying a cloud-based spectrum observatory without compromising the vital features. Figure 1.4 shows the system architecture implemented by FuNLab along with a scan of radio frequencies in the United States of America.

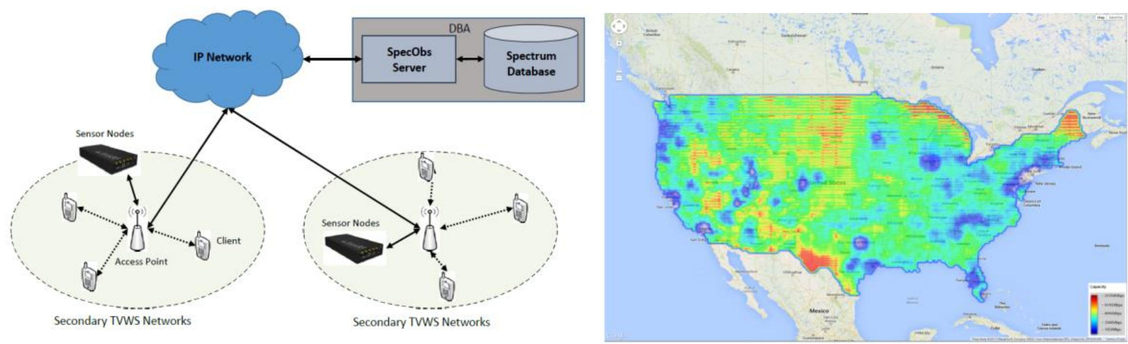

Figure 1.4: FuNLab Spectrum Observatory. From [13]

Table 1.1 summarizes the specifications of each of these spectrum observatories [1][9][12].

Table 1.1: Specifications of other Spectrum Observatories

Company/ Organization	**FuNLab of the University of Washington**	**Microsoft**	**The Illinois Institute of Technology (IIT**
Spectrum Sensing tool	Sensor nodes	Spectrum Analyzers (50MHz to 2.5 GHz)	SDR (30 MHz to 6 GHz)
Data Base	Cloud server	TX Miner	MongoDB
Storage	Cloud server	Blob Storage	DSNET

1.3 Technical Challenges

Most existing spectrum observatories concentrating on finding radio frequencies and displaying real-time results have high operating and development costs or have privileged access to RF equipment. Smaller-scale implementations of a spectrum observatory may not have access to such high budgets and equipment, limiting these implementation's ability.

MongoDB, a database platform, was used by the spectrum observatories developed by IIT and the University of Washington to store spectrum data and metadata [5][9]. MongoDB is a cloud-based database platform used by application developers to store large-scale datasets. Even though this software has a free version, the storage is not enough for continuous data generated by SDRs.

A paid subscription for this database would be ideal for storing spectrum data, but it is not cost-efficient for lower budget projects as the subscription can cost upwards of $97/month in addition to the costs of the hardware needed to host the database.

SDRs have various parameters such as center frequency, gain, and channel bandwidth. Although it is possible to change these parameters during runtime, it is usually done on the computer hosting the radio. Remotely changing these parameters requires the transfer of commands from the user interface to the host computers. This transfer often causes complications as commands are transferred between multiple programming languages and multiple, differently configured computer systems. As such, this ability to change commands is very uncommon. Some observatories let the user select and view the spectrum at different geographical locations by selecting an option on the web interface, but this does not require changing radio parameters. This user input method does not require data to transfer back to the radio host like changing radio parameters would [5][14].

1.4 Contributions

Cloud-based Spectrum Observatory:

The cloud-based Spectrum Observatory was implemented using several SDRs and a custom central cloud server. The sensor nodes were run by GNU Radio and sampled specific 5G spectrum bands, uploading the spectrum data to the central server. This platform was improved to support several radios, and acts as a proof-of-concept that has the ability to be expanded to support additional sensor nodes and create additional functionality for the user.

Web-based User Display:

End-user access to the data was created with an openly accessible website. The utilization of JavaScript on both the server and website allowed for the formatting and display of data as a spectrogram with data being transferred from the cloud server to the user. This functionality allows users around the world to access the sampled data without the need for physical access to the observatory.

Remote User-control of Radio Parameters:

In addition to the display, the user-accessible website provided functionality for remotely changing the SDR's sampling parameters during runtime. Commands from the user are passed through the cloud server and accessed by GNU Radio using custom functionality. This allows users to remotely control the spectrum band being sampled from a selection of pre-set frequencies, in addition to being able to remotely view the sampled data.

1.5 Report Organization

This report is organized into nine chapters as follows. Chapter 1, the Introduction, details the motivations for this project and the current state of the art for spectrum sensing and display regarding cloud-based systems. The Introduction concludes by briefly analyzing challenges in the current state of the art and explaining how this project contributes to current research. Chapter 2, Overview of Spectrum Sensing, outlines essential information on the technologies used in this project, such as SDRs and cloud platforms, as well as the types of data and processing required for displaying frequency spectrum data. Chapter 3, Proposed Implementation, summarizes possible solutions, outlines the general implementation, and defines the project's timeline. Chapters 4 through 7 describe the implementation methodology and the criteria for measuring success, with each chapter focusing on a specific segment of the final project. Chapter 8 presents findings and analysis on the data measured in the previous Methodology chapters. The report finishes with the Conclusion in Chapter 9. This chapter summarizes the project's contributions and makes recommendations on areas of future improvements.

2. Overview of Spectrum Sensing

2.1 Introduction of Frequency Spectrum

Radio spectrum is a portion of the electromagnetic spectrum used in modern technologies, especially telecommunication [14]. The radio spectrum and its internal partitioning are managed by the Federal Communications Commission (FCC) and the National Telecommunications and Information Administration (NTIA) [15][16]. Figure 2.1 highlights the different frequency allocated by the Federal Communications Commission.

As seen in Figure 2.1, frequency ranges from 9 kHz to 275 GHz have been allocated, and these allocated frequencies are used for different purposes. For example, broadcasting AM radio has been allocated frequencies between 535 kHz and 1605 kHz, FM radio spans between 87.8 MHz to 108 MHz, and radio navigation has the frequency range between 9 kHz and 14 kHz allocated to it [18]. Different telecommunications companies have licensed specific bandwidths for their commercial uses—Table 2.1 highlights these various cellular frequency band allocations [15][16]. Wi-Fi with a the standard 802.11 provides selective ranges for use, these include: 900 MHz, 2.4 GHz, 3.6 GHz, 4.9 GHz, 5 GHz, 5.9 GHz, 6 GHz and 60 GHz frequency bands.[15][16]

Table 2.1: Licensed Frequency Bands by Company

Provider	3G Frequency	4G Frequency	5G Frequency
AT&T	GSM/UMTS/HSPA+ 1900 MHz,850 MHz	1900, 1700/2100, 850, 700 2300	850 MHz, 39 GHz
T-Mobile	GSM/UMTS/HSPA+ 1900 MHz, 1700/2100 MHz,CDMA 1900 MHz, 800 MHz	1900, 1700/2100 700, 600,1900, 850, 2500	600 MHz, 28 GH, 39 GHz,2.5 GHz
Verizon	CDMA 850 MHz, 1900 MHz	1900, 1700/2100, 850, 700	28 GHz,39 GHz

Figure 2.1: FCC Frequency Allocations. From [17]

Four major frequency bands are used in technology today. There are low frequencies that range between 30 kHz to 300 kHz, high frequencies that have a range between 3 MHz to 30 MHz, ultra-high frequencies that have a range between 300 MHz to 3 GHz, and microwaves that range between 1 GHz to 1000 GHz [19][20]. Frequency bands that range between 24 GHz to 100 GHz are known as millimeter waves [21], and this range is primarily used for 5G. Other low bands and mid bands are also used by telecommunication companies, such as AT&T's 850 MHz band and T-Mobile's 600 MHz band.

2.2 Introduction to Fifth Generation Technology

5G refers to fifth generation mobile network technology, and it uses a network of cell sites to send encoded radio waves. Each cell site is connected to a network backbone through a wired connection [22]. 5G primarily uses millimeter waves because the use of millimeter waves provides higher bandwidth because higher carrier frequency translates to higher signal bandwidth. The portion of frequencies that range from 30GHz to 300 GHz are referred to as millimeter waves because the wavelengths range between 1-10 mm. Along with millimeter waves ultra-high frequencies (UHF) have started to be repurposed for the use of 5G.These higher frequency waves cannot penetrate objects such as walls and buildings, limiting the range of the higher frequency 5G cellular data [22]. It is reliable in densely populated areas such as parks, stadiums, and large cities.

Multi-access edge computing (MEC) is growth to cloud computing, it brings applications from centralized data centers to the network edge. This application creates a shortcut in the delivery of content between the user and host [23]. Some characteristics of MEC are real-time access to Radio Access Networks (RAN), high bandwidth, and low latency. Another integral part of the 5G architecture is Network slicing [23][24]. This technology along with Network Function Virtualization (NFV) allows multiple networks to simultaneously run on top of shared physical network infrastructure (wired connection) [24]. Figure 2.2 illustrates 5G slicing architecture with SDN/NFV architecture and its connection to a shared physical network infrastructure. This is needed for 5G architecture as the creation of end-to-end virtual networks includes network and storage functions, which can be managed by partitioning the networking resources depending on cases with different latency and availability.

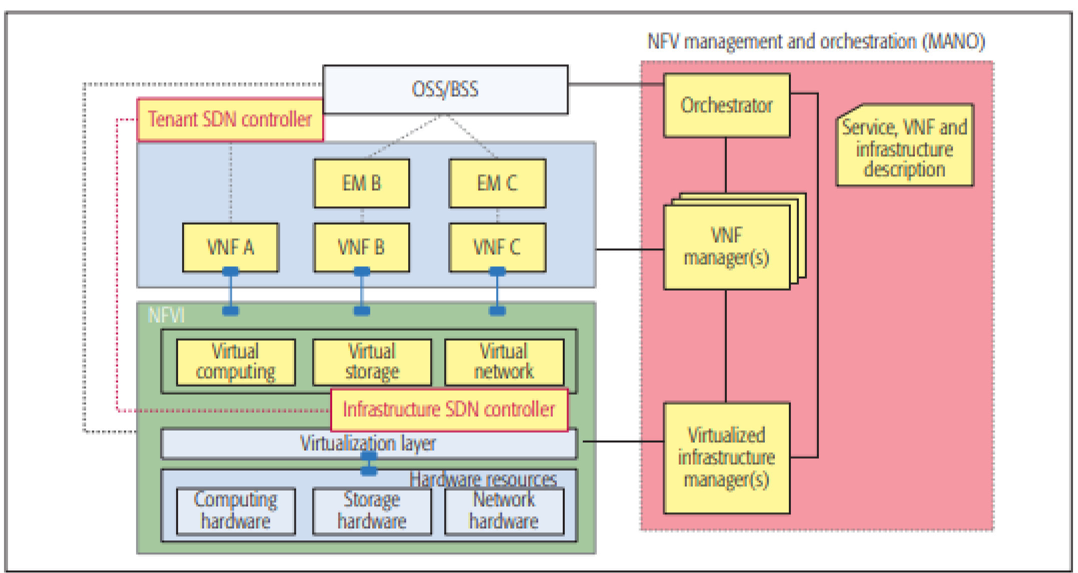

Figure 2.2: 5G Slicing architecture with NFV. From [25]

Another part of 5G architecture is known as beamforming, base stations transmit signals in multiple directions using Multiple In Multiple Out (MIMO) arrays using a lot of small antennas combined with the use of signal processing algorithms [23]. These algorithms are able to determine efficient paths for signals and data packets to travel to reach the intended destination. The use of small antennas enables large arrays to occupy the same area and help with reassigning beam direction several times per millisecond [23]. The use of a larger antenna density in the same area makes it possible to achieve narrow beams with the use of MIMO. This provides a high throughput with effective user tracking [23].

2.3 Software-Defined Radios

In the field of communication, it is not always feasible to exchange information between different types of equipment due to data loss and incompatibility. Thus, SDRs are used, as they have flexible architectures. The physical layer function of an SDR is defined through software such as GNU Radio, CubicSDR, and Simulink. Leveraging the programmable nature of the platform, SDRs can transmit and receive signals to produce various data types from a wide range of frequencies, and it is flexible enough to switch channels and change modulation schemes in real time [26].

The block diagrams of the transmitting and receiving functionality SDRs are shown in Figures 2.3 and 2.4, respectively. For the receiver end of the SDR, the radio frequency (RF) tuner first converts analog signals to intermediate frequency (IF). After this, the IF signal is passed to the analog to digital converter (ADC), which changes the signal's domain, and the output sample is passed onto the digital down converter (DDC). The DDC is the critical part of the system, and it contains three main components: a digital mixer, a digital oscillator, and a low-pass filter. The digital mixer and oscillator shift the IF samples to baseband, and the low-pass filter encloses the bandwidth of the final signal. The DDC output, baseband samples are passed onto the processing block, the digital signal processing (DSP). Similarly, the transmitter end of the SDR acts very much like the receiver; however, the process is in reverse order [27].

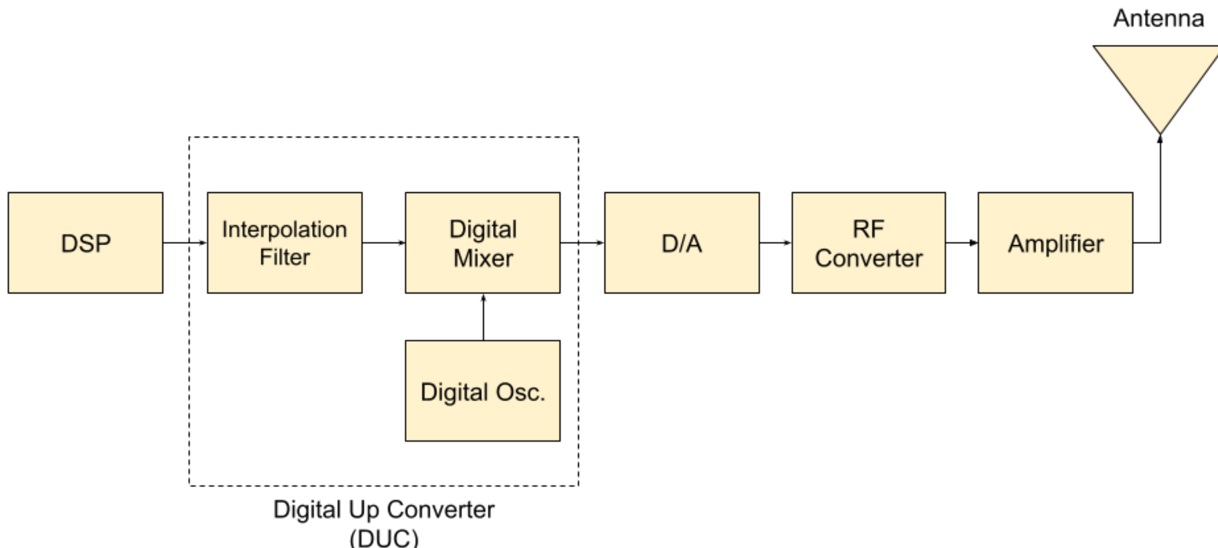

Figure 2.3: Block Diagram of SDR's Transmission Functionality. Adapted from [28]

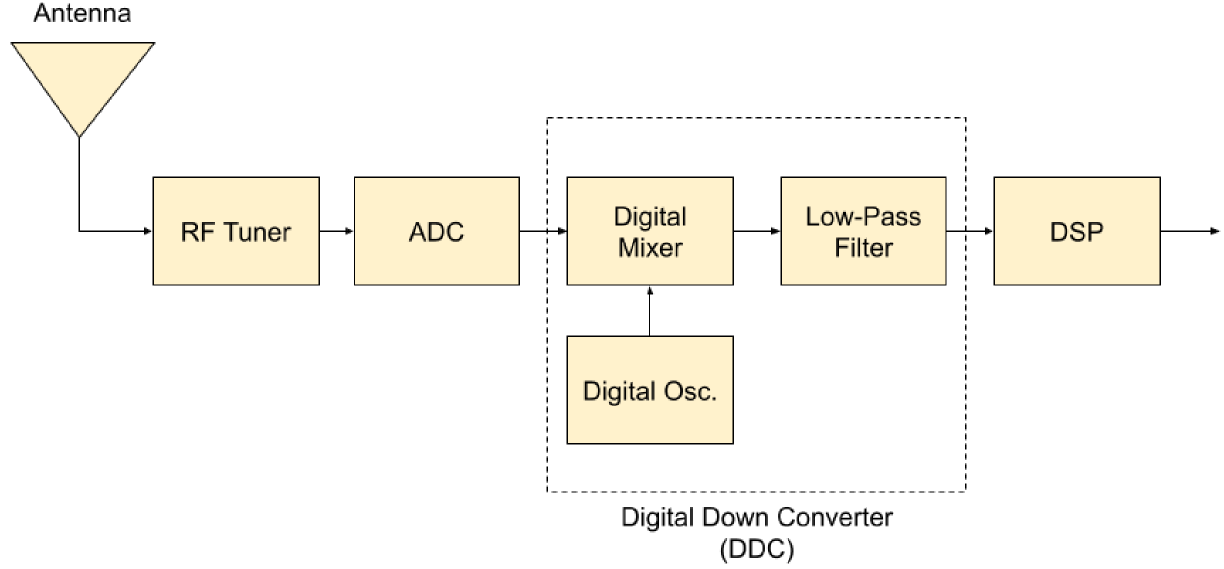

Figure 2.4: Block Diagram of SDR's Receiving Functionality. Adapted from [28]

GNU Radio is a free and open-source software development toolkit that uses signal processing blocks to program an SDR and can perform all necessary signal processing using the different default blocks or customized blocks. On GNU Radio, different applications can be written to receive data from the digital streams and push data into the digital streams using an external hardware device to transmit. The different element blocks in the software are capable of handling all the digital data and can be used to create filters, channel codes, demodulators, equalizers, and many others. The software is programmed by connecting these blocks so that data can be passed from one block to the other. It provides a user-friendly graphic user interface (GUI) for the user. Python files can also be generated from the flowgraph which can be modified based on user needs [29].

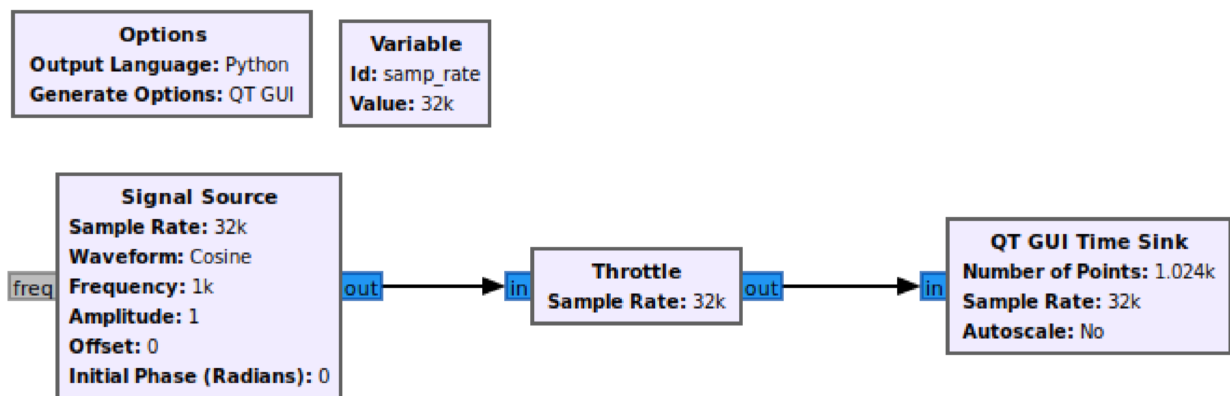

Figure 2.5: GNU Radio Flowgraph Example [30]

2.4 Types of Spectrum Display

Different types of plots are used to display spectrum data according to the user's need. The waterfall plot and the amplitude plot are the most common [5][9]. The waterfall plot is a three-dimensional plot that shows a two-dimensional phenomenon such as spectrum change over time, as seen in Figure 2.6. The vertical axis represents time, the horizontal represents frequency, and the third axis shows amplitude or power, represented by changes in color [31]. An amplitude plot in spectrum sensing displays the instantaneous power of a frequency channel, as seen in Figure 2.7. This type of plot is mainly used to determine whether a channel is active or not, whereas the waterfall plot is more widely used to analyze data on a larger scale [32].

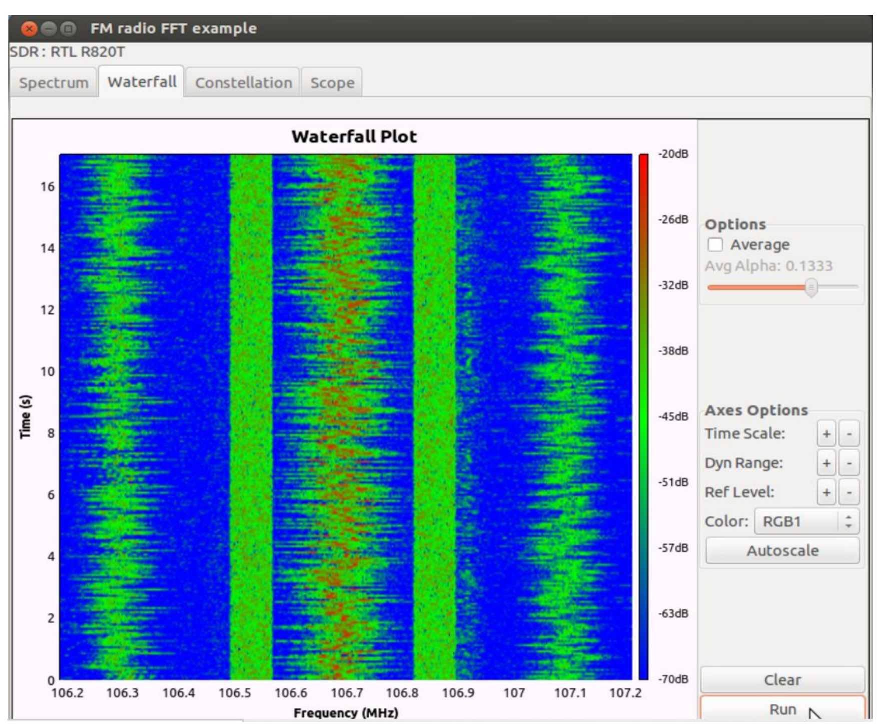

Figure 2.6: Waterfall Plot. From [32]

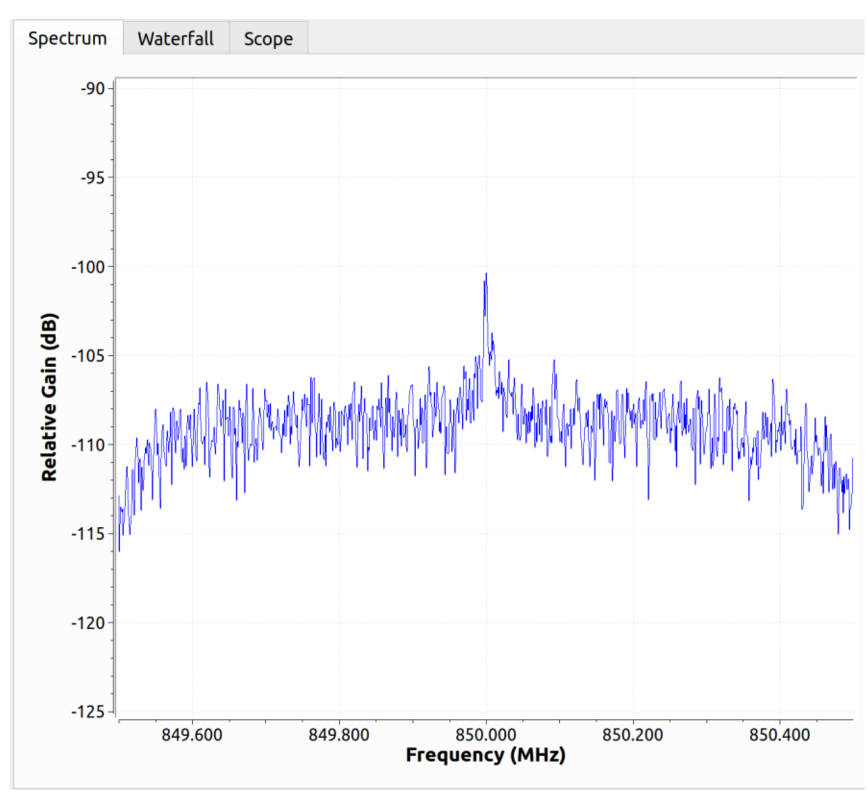

Figure 2.7: Amplitude Plot

These display methods have positives and negatives associated with their use related to computational power needed and usefulness of data displayed. An amplitude plot is an effective tool for showing a simplistic overview of a frequency spectrum and only requires magnitude data from the fast Fourier transform and the center frequency [33]. The waterfall plot requires the same magnitude and center frequency data but also requires the ability to store multiple time instances of the magnitude, drastically increasing the storage space necessary to use the plot. The other concern with waterfall plots is the code necessary to produce one. In a language such as Python [34], it is a simple matter using the matplotlib [35] library. However, when using JavaScript's D3 graphing library [36], there is no built-in functionality for building a waterfall graph, requiring custom-built methods for drawing the desired field instead.

When displaying a spectrogram, the most important thing to consider is; what does each axis symbolize and what are its units. In the examples shown in Figures 2.3, the x-axis is labeled as frequency, and while this is partially correct, it does not tell the full story of what data is being stored and displayed. In reality, the x-axis represents the frequency bins collected from the Discrete Fourier Transform (DFT) earlier in the computation. The x-axis is dependent on: the center frequency at which the spectrum was collected, the sampling frequency, and the size of the DFT used to calculate the frequency magnitudes. The total range of frequencies, or bandwidth, is equivalent to the sampling frequency, with the minimum and maximum frequencies displayed defined in (1) and (2).

$$minimum\ frequency\ = f_c - f_s \tag{1}$$

$$maximum\ frequency\ =\ f_c + f_s \tag{2}$$

where f_c is the center frequency and f_s is the sampling frequency. Additionally, the described range of frequencies is split into equal parts that are averaged and displayed based on the size of the discrete fast Fourier transform. Each data point in the transform, called a frequency bin, holds the average number of frequencies in its range. The number of bins is described by (3).

$$Number\ of\ Bins\ = f_s/FFTsize \tag{3}$$

The y-axis is much simpler for the two display methods. In the amplitude plot, the vertical axis shows the given frequency's amplitude. This amplitude is commonly in decibels [dB] and creates a logarithmic scale [37]. In the waterfall chart, the vertical axis represents time, with each new line representing a new time instance. This results in the waterfall chart needing to retain past data for display in addition to displaying the most current set of data.

The final axis to be aware of is the z-axis, which is only relevant for the waterfall chart. The z-axis represents the magnitude of the corresponding frequency at the corresponding time and is represented by color. Commonly, high power frequencies trend towards a more red color, while background noise is commonly blue.

2.5 Software Daemons and Program Control

In computing, a daemon is a type of process that runs in the background and typically requires little or no input from the user, making them different from a normal process [38]. Daemons are typically long-running, with the daemon being created at system startup and running until termination when the system shuts down. This behavior makes a daemon process ideal for performing autonomous tasks such as serving web pages, like the Apache HTTP server daemon (httpd), or providing a network login ability, like the Secure Shell daemon (sshd) [38].

Characteristics that define a UNIX daemon are outlined below in Table 2.1. Due to this standard set of characteristics, many programming languages have a standard implementation for converting a regular program into a daemon [39]. Having a standard implementation allows the creator of the daemon not to have to ensure these creation steps are completed precisely and ensures misbehaving daemons are unlikely to be created.

Once the process has successfully been turned into a daemon, the process enters the main program to execute its intended behavior. This main program is often located inside an infinite loop due to the nature of daemons often running until system shutdown.

The characteristics of a daemon also mean that there are very few ways to interact with it during runtime (as is expected since these processes are meant to be autonomous) [40]. The two typical interactions a daemon has are to signal a restart or to signal shutdown. Each signal serves a specific purpose. By indicating a daemon should restart, it allows a user to change a configuration file to send new operational parameters to a daemon. Sending a daemon a signal to terminate allows it to perform any cleanup necessary to exit safely. Most often, the termination signal (SIGTERM) is sent to the daemon by the system as it shuts down. In this case, the daemon’s shutdown process must be short to ensure it can be completed before the system fully shuts down [38].

Table 2.2: Actions taken to Create Unix Daemon

Action Performed on Process	Reason
Perform a process fork and exit the parent process	Orphans the child to run in the background and ensures the child process is not a process group leader
Child process from previous fork sets its SID	Starts a new session for the child and frees the child from any association with a controlling terminal
Fork the child process again and exit the parent. This parent is the child in the previous step.	Ensures the child process is not the session leader and can not reacquire a controlling terminal
Clear the process unmask for the new child process	Ensures the process has the needed permissions when creating files and directories
Change the process's working directory (typically to the root directory)	Ensure the process's working directory can not be unmounted
Close all open file descriptors belonging to the process	Ensures stdin, stdout, and stderr are closed and not pointed to a terminal. Additionally, this helps avoid file descriptor leaks
Reopen file descriptors 0, 1, and 2 and point them at /dev/null	Ensures that if the daemon program tries to make calls to stdin, stdout, or stderr, these functions will not unexpectedly fail

2.6 Cloud Servers

With the introduction of the Internet of Things (IoT) and an increase in internet-connected devices, the concept of "The Cloud" and cloud computing has been growing in popularity [41]. When an item or utility is referred to as "cloud-enabled" or "cloud-based" (like this project), it typically refers to using the Cloud to transmit data and information. The Cloud, in this case, is a central server used to host, route, and store that data or information with the benefit that the cloud server has more resources and abilities available than any single device [42]. Figure 2.8 shows a typical layout of a cloud system. The user accesses one or more servers over the internet and these servers contain applications and databases that provide the user with the desired functionality.

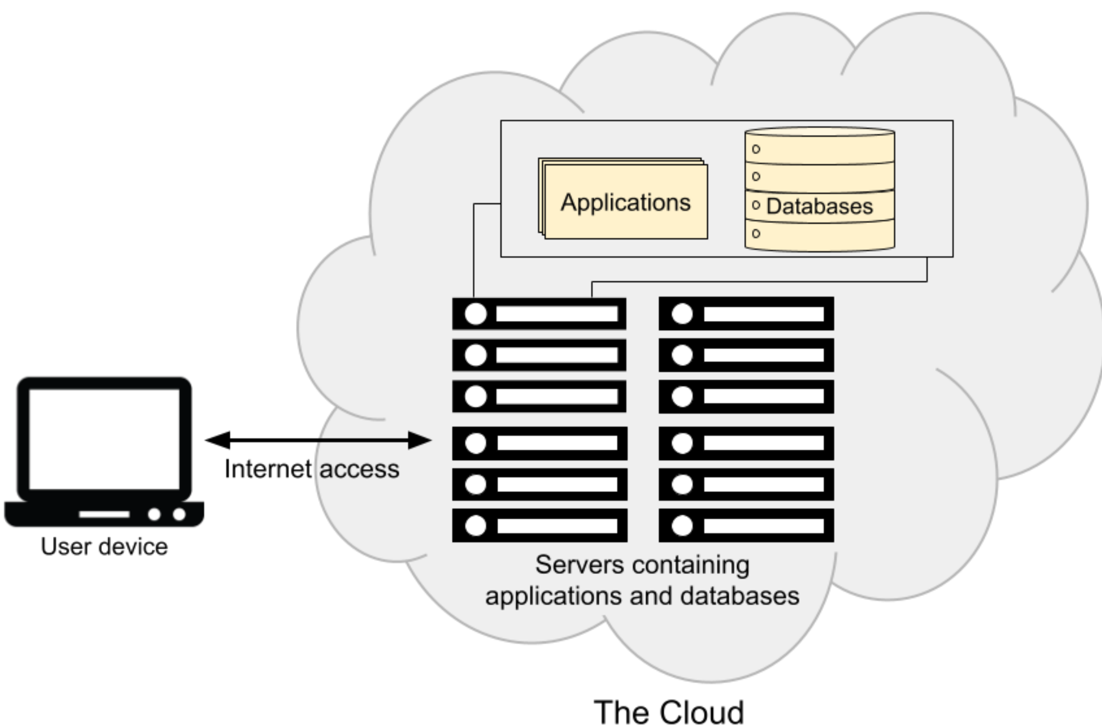

Figure 2.8: An example of the functional layout of a Cloud system. Adapted from [43]

In addition to hosting data and making said data available over the internet, cloud servers have additional processing power that can be used to support less powerful devices. One of the primary differences between an on-premise server (a server physically hosted and managed by the company using it) and a cloud server is that cloud servers are often virtualized so that the company subscribing to the provider has a lower subscription and operating cost for the server space [44]. Table 2.3 outlines several of the pros and cons between using an on-premise server and a cloud server.

Table 2.3: Pros and Cons of an On-Premise Server vs. a Cloud Server

Server Type	Pros	Cons
On-Premise Server	- Lower latency to server - Direct access to the hardware - Complete control of the server	- Expensive to maintain - Requires on-site technical support to manage the server - Requires owning/renting a physical location for the hardware
Cloud Server	- Do not need to manage physical server hardware - Less expensive - Professional technical support for the server	- Dependent on separate company for anything running on the server - Critical data is stored external to the company

Virtualization refers to splitting up the physical server hardware using software so that one physical server can act as multiple virtual servers sharing hardware. Figure 2.9 shows the difference between a traditionally server installation and a virtualized server installation. The coloring of each block corresponds to the next layer of the installation, with the virtualization installation containing an additional layer. In the classic server case, the physical hardware only supports one operating system instance as opposed to the virtualized instance that can run several instances of an operating system through the use of virtualization software.

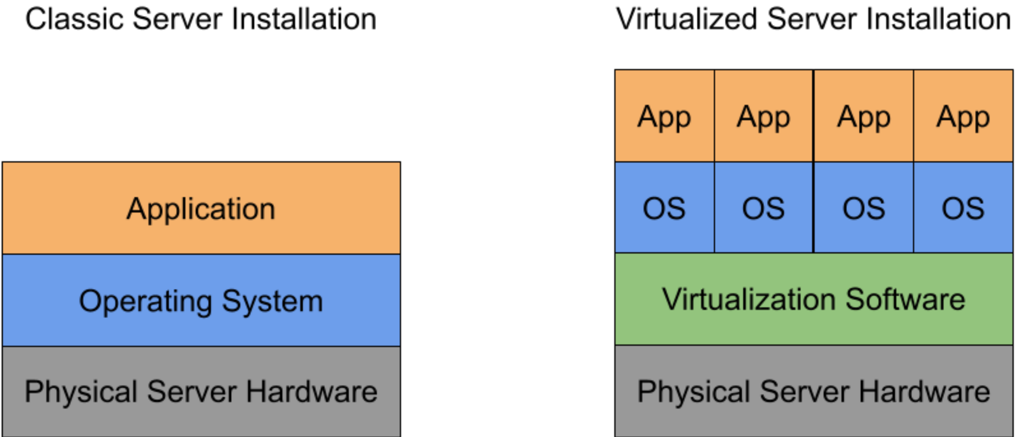

Figure 2.9: A comparison between a classic server and a virtualized server installation.

Cloud servers are typically sold by commercial providers such as Amazon or Microsoft, (Amazon Web Services (AWS) and Microsoft Azure, respectively). There are three main types of services that are sold, these being Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

The major difference in these three services is the amount of infrastructure and software managed by the provider versus the consumer. With IaaS, the consumer is provided with just the server infrastructure and is responsible for managing things such as operating systems and development tools. IaaS is commonly used for tasks like website hosting and data storage as these types of jobs require more control over the underlying systems on the server [45]. With PaaS, much of the lower-level software, such as the operating system and database management software, is managed by the provider so the consumer can focus development on tasks specific to their use case [46]. The final service, SaaS, requires the least amount of management from the consumer as the consumer is buying access to a preexisting piece of software. One very common example of SaaS is a web-based email service, like Gmail. The consumer purchasing access to a SaaS application does not have to worry about managing the servers hosting the application, and a majority of the workflow is managed by the service provider [47]. Figure 2.10 shows the three main service models of cloud computing and outlines what is provided to the client by the cloud hosting company. These three types of cloud services have different uses that must be considered when selecting what type of cloud platform is required for a given product.

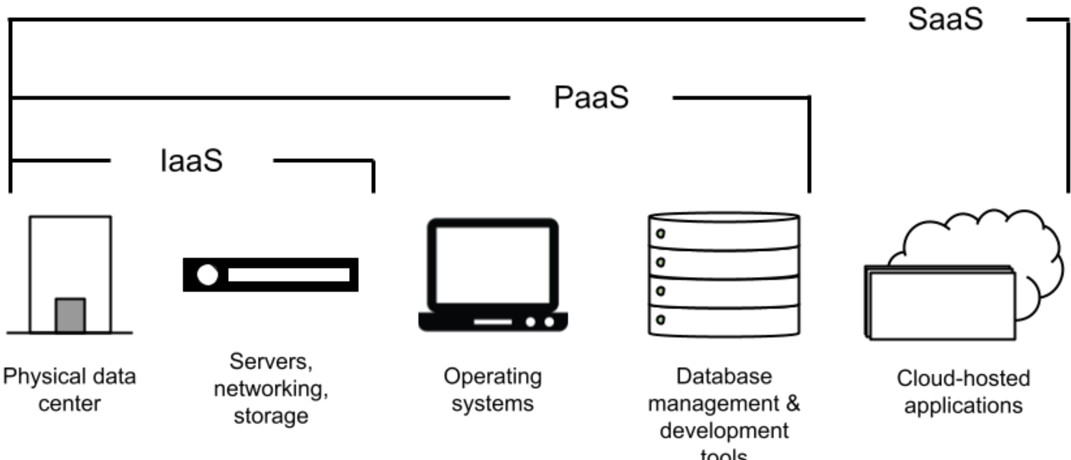

Figure 2.10: An outline of the services and tools provided by the three main cloud computing service models offered by many cloud hosting companies. Adapted from [43].

2.7 Chapter Summary

This chapter introduced the frequency spectrum, explained how spectrum data could be collected and displayed. Different components required to implement a spectrum sensing model were outlined, including the utilization of SDRs, GNU Radio, and software daemons. Furthermore, several types of graphs that are conventionally used to display spectrum data and a summary of cloud servers and cloud infrastructure were discussed.

3. Proposed Implementation

3.1 Problem Statement

The need for spectrum observatories by both professionals and hobbyists is high. As detailed in Chapter 1, a majority of existing spectrum observatories are not easily deployable and have both a high cost to set up and operate. For this reason, the goal of this project is to create a proof-of-concept of a lower-cost, more easily deployable spectrum observatory that is capable of sampling the 5G frequency spectrum. In addition to being less expensive to maintain, one of the goals is to also make the observatory more accessible to the average person by allowing internet access to both view the sampled data and interface with the radio's sampling parameters remotely. In accomplishing this goal, the final proof-of-concept will outline the fundamental structure of a lower-cost, more accessible spectrum observatory, with the ability to be expanded upon to create a more permanent, large-scale system.

3.2 Proposed Approach

This project aims to create a spectrum observatory capable of monitoring multiple 5G frequency bands that are updated in real-time and accessible online. Several SDRs will be used to collect data from various separate frequency bands, with the data being processed and stored using cloud infrastructure. The data from the radios will be processed and displayed as independent spectrum graphs that are located on a webpage acting as the point of user interaction for the observatory. This webpage also allows the user to select predefined center frequencies to monitor, updating the radios and data in real-time to provide a complete in-depth view of many different 5G frequency bands. The specific details as to how this approach will be implemented is detailed in Section 3.4, later in this chapter.

3.3 Metrics for Success

At the start of this project, there were four different goals established to evaluate the success of the project. Two of these were measurable metrics, latency and accuracy, while the other two were slightly more abstract accomplishments, successful storage and display of a 5G spectrum.

Regarding latency, the implementation is deemed successful if it is capable of collecting data for a spectrum, saving the data, sending that data to the web server, and writing the data to a display in less than a full minute end to end. The barriers to achieving this are all related to code optimization and data throughput speed. This is assessed by comparing the sample time metadata saved in each data file with the current time.

The metric for accuracy is based on the difference between the received signal in GNU Radio and the signal displayed on the website. The expectation for a successful display with accuracy is no more than a 5% difference in displayed magnitude values on the website and the actual frequency magnitude values analyzed inside GNU Radio.

The two abstract metrics for success, storage and display, are evaluated by their completion and robustness. A robust storage system for the data is able to take in the large amounts of data coming from the radios while also being able to serve that data to the website for display. A robust display on the website allows the user to accurately see the spectrum at a given frequency and the time the data was collected. The connection between the storage and the display is important for these metrics to be met. The quantity of data that is collected by the radios and needs to flow smoothly to the website is large and requires that the speed the data can travel through the entire implementation is fast enough to keep up. Having a slow connection would result in the failure to meet the metric for latency, causing data to slowly build up in a queue until it can be displayed and the display to fall behind the sampling stage of the system.

3.4 Design Overview

To achieve the goal of this project, the frequency spectrum in the sub-six gigahertz range, which ranges from 70 MHz to 6 GHz, will be observed. To conduct this spectrum sensing, several SDRs (USRP 2901s) will be utilized. Each SDR will be programmed to scan at the frequency band requested by the user on the client interface. The SDR will be programmed using GNU Radio which will scan the requested frequency band and store RF data as a ".dat" file on the host computer.

In order to utilize the data from the separate SDRs to create the final spectrum graphs outlined in the project goal, all datasets must be sent to a centralized location where they can be processed and formatted. To accomplish this, a cloud infrastructure will be utilized to process and

display the data. Using a software daemon and an uploader script, data from the host computer will be sent to the cloud server to be prepared for display to the user.

As each radio is responsible for data of distinct frequency bands, it is important to keep track of the stored data and catalogue them according to the SDR metadata (e.g., sampling frequency, timestamp, etc.). This makes it easier for the developer to integrate the information to create four separate spectrum displays, one for each radio, on the web interface.

To achieve remote control of the radio parameters through the web interface, the cloud platform also needs the ability to send data in the opposite direction, from the web browser back to the radio host computer. This can be implemented using HTTP GET and POST requests and HTML forms to create the user input on the website. The user input from these forms is sent to the server using an HTTP POST request. The server stores this command and sends it to the host computer when it receives an HTTP GET request. GNU Radio on the host computer sends this request, and once it receives the user input, it changes the radio parameter accordingly. Figure 3.1 on the following page shows a block diagram of the design overview.

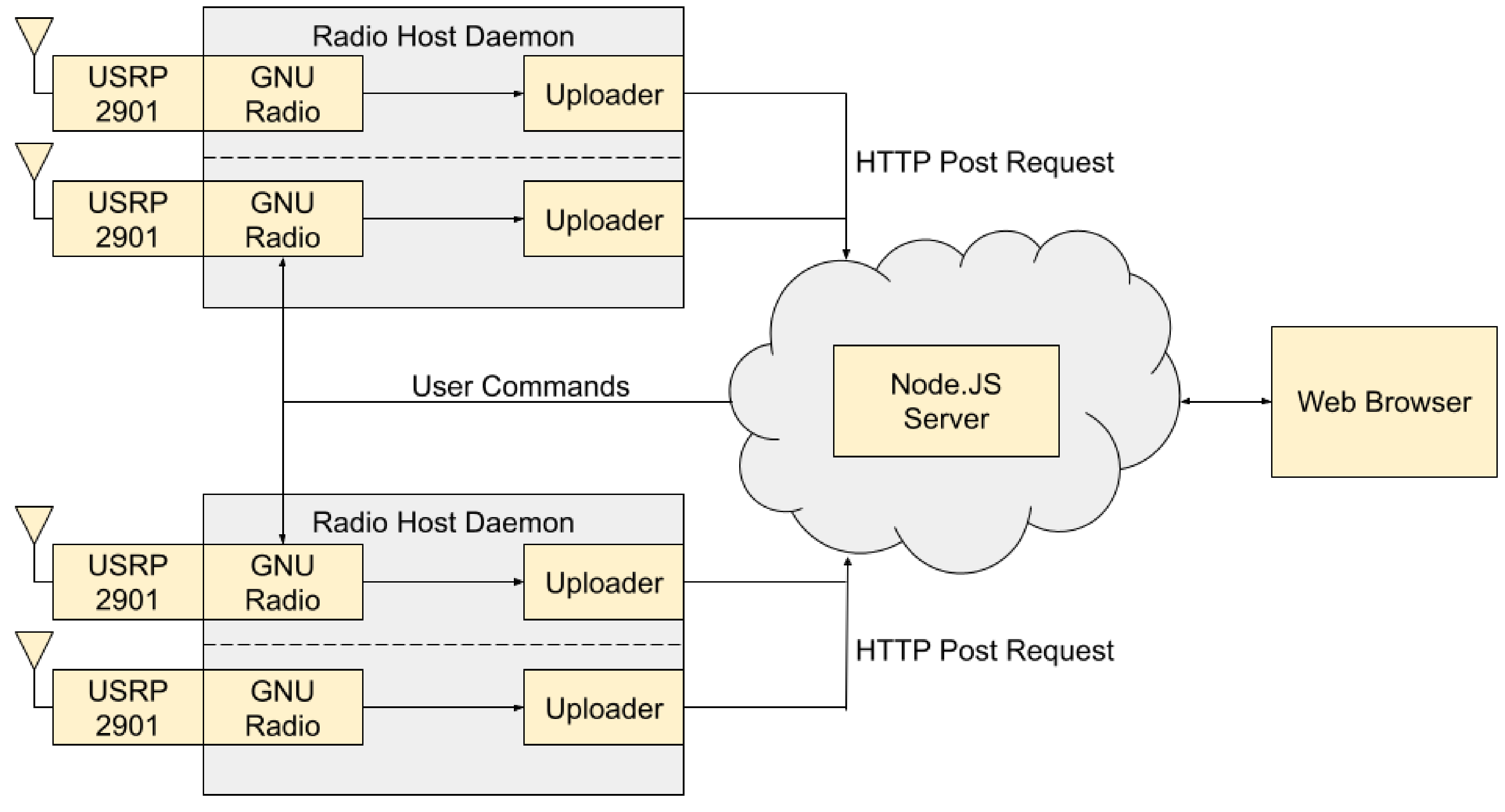

Figure 3.1: Block Diagram of Design Overview

3.4.1 GNU Radio

In order to successfully process radio signals, an SDR requires a software that can perform digital signal processing at high speed. GNU Radio offers both graphical design approaches along with development in Python and C++. GNU Radio Companion (GRC) is a graphical design approach in GNU Radio which allows users to create and execute signal processing applications by utilizing a drag-and-drop method. With GRC users can easily connect different module blocks graphically to receive and transmit data from the radio. It uses a block diagram to represent the flow of the project within the software. This graphical feature in GNU Radio has a signal generator block and interfaces to sound cards. It also can read and write data to the file system without any additional hardware [48]. Although it allows users to easily process signals, there are some advantages and disadvantages of using GNU Radio as seen on Table 3.1. Even with few limitations of GNU Radio, the advantages of the program largely overcomes them such as the ability to easily design a system to process analog and digital signals at high speed at reduced cost.

Table 3.1: Advantages and disadvantages of signal sampling on GNU Radio

Advantage	Disadvantage
• Free open source toolkit • Ability to generate data from both command prompt and/or GRC • Programmable using Python and graphical feature • Real time data transfer	• Outputs large data size file • High noise in RF • Processing delay

As shown in the block diagram of GNU Radio in Figure 3.2, SDR is programmed in GRC by first sampling signals from an input device (a signal source) at some sampling rate. The first block on GNU Radio is the signal source which directs the program to receive samples of signals at specific frequency bands with a specified sampling rate. The samples provided from radio devices are complex numbers with an I and Q component. These samples are processed by calculating the fast fourier transform (FFT) and saving the file on the designated source. The simplicity of the block diagram allows users to construct an SDR easily. Each of the module blocks

includes interchangeable options such as frequency, sample rate, and bandwidth. GRC allows the user to visually see the port connections of the same data type or data type converter of the block diagram.

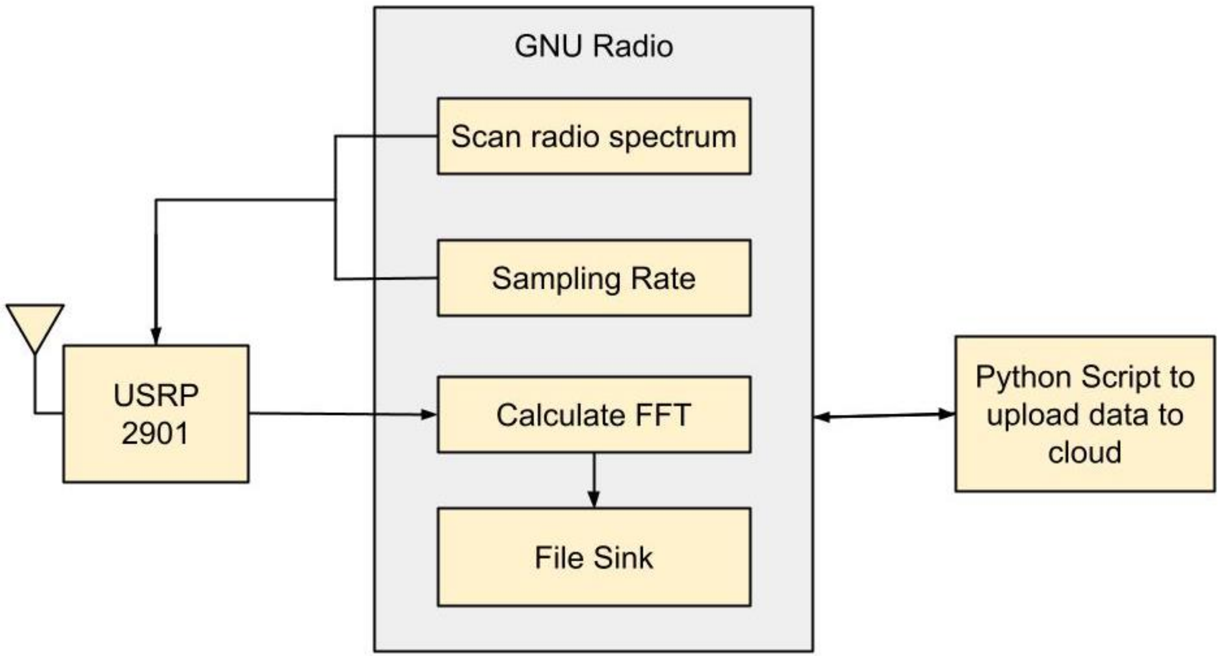

Figure 3.2: Block Diagram of GNU Radio Data Sampling and Processing

For this project, the GNU Radio platform was selected for digital signal processing due to its simple design and graphical features, and the ability to utilize Python or C++ programming to expand the software's functionality for more complex applications. As seen in Figure 3.2, the processed sample file is saved using File Sink and is sent to a Python Script for uploading to the cloud server which will be discussed in greater detail in section 5.4. Even with few limitations of GNU Radio, the advantages of the program largely overcomes them such as the ability to easily design a system to process analog and digital signals at high speed at reduced cost.

3.4.2 Cloud Infrastructure

To create a cloud-based spectrum observatory, server hardware to host the cloud platform was required. The options for web and cloud infrastructure were evaluated during the planning phase, with two main options being considered. These primary options are commercial infrastructure and self-hosted infrastructure. In this context, self-hosted infrastructure refers to the

server hardware hosted and managed by WPI's Academic & Research Computing (ARC). The primary benefit of using commercial server infrastructure over a self-hosted solution is that commercial options, such as Amazon Web Services (AWS) or Microsoft Azure, have much of the web backend foundation in place, so that development would be focused specifically on the spectrum observatory. Below, Table 3.2 outlines the differences between the three major cloud platform providers selected to be evaluated [49][50][51]. While these services all have different features, the important points are that they all have database platforms, compute service options, similar pricing models. Because the intent for the project was to create a smaller scale, less expensive cloud-based spectrum observatory, these commercial options are not worthwhile. The compute services and database platforms are not needed for the project, and the benefit of the platform's pre-established framework does not outweigh the cost of needing to pay for the service.

Table 3.2: Comparison between different commercial Cloud platforms

	AWS	**Azure**	**Google Cloud**
Pricing	Pay as you go (hourly basis)	Pay as you go (minute basis)	Pay as you go (minute basis)
Compute Services	EC2 (IaaS) Elastic Beanstalk (PaaS)	Azure Virtual Machine (Azure's strongest focus is PaaS Compute)	Compute Engine (IaaS) App Engine Application Platform (PaaS)
Database Engine Support	Amazon Aurora, MySQL, Microsoft SQL, PostgreSQL, and Oracle, NoSQL	Azure SQL, NoSQL	MySQL, PostgreSQL, or SQL Server
Hybrid/Multi-cloud Support	General systems are confined to the AWS family (hybrid cloud platform is very new)	Open to hybrid cloud systems	Anthos - Enterprise level hybrid/multi cloud platform
Software Integrations	More support with the open-source community	Integration primarily focused on Microsoft's own products and Windows dev tools	Attempting to support open source more, not as established as AWS

After concluding that commercial options were not reasonable to accomplish the low-cost goal of the project, using a self-hosted server infrastructure was deemed the best option. This solution allows the implementation to be low cost, as WPI's server infrastructure is free, and allows the implementation to be more portable as it is not dependent on features only available on a commercial platform. WPI's infrastructure most matches the IaaS options from commercial providers as things like network security are already set up and managed by WPI. This decision meant that more work needed to be dedicated to setting up the server running the platform, but overall meets the goals of the project best.

3.4.3 Cloud Server Software

The backend runtime environment used to create and run the server was evaluated and selected between two different open-source options, Node.JS and Apache. These software had key differences and were analyzed to choose one that would suit the project's needs best. These needs included the ability to quickly process large amounts of data, the ability to process large amounts of incoming requests, and be a system that was easily and rapidly deployable.

Node.JS is an asynchronous event driven software that uses JavaScript to build networks and applications. Due to the software's asynchronous architecture, it is possible to create event driven loops [52]. An event driven loop occurs when certain events occur, for example receiving and transmitting requests are both events that can fire [52]. The event loop manages the incoming requests while delegating I/O tasks to different worker threads as I/O processing takes much longer to complete [52]. This multi-threaded approach allows the server to handle file processing tasks without blocking new requests from executing. The system is malleable as the events are programmable to cater to the clients needs, and because of the event driven architecture, the program is very scalable.

Apache is a web server application software widely used in the world. Apache has a module-based architecture, this means that it takes a service or task and breaks it apart in attempts to manage and carry out the work [53]. Modules help server administrators terminate and restart individual processes without needing to restart the entire server [53]. These modules are used for different operations such as security URL rewriting, caching, and password authentication. Apache handles incoming requests as a single thread and can create and destroy processes depending on the administrators purpose [53]. Apache, with it's simple configuration along with the use of

modules and an environment that is friendly and digestible for new programmers, makes it the most widely used web based software.

Table 3.3: Comparison of Node.JS versus Apache

Software	**Apache**	**Node.JS**
Efficiency	Single Thread	Worker threads
Architecture	Module-base	Asynchronous Event Driven
Data Processing	Request rate falls short behind Node.JS	Request rate shows a high processing rate

Table 3.3 is a comparison between two open source web-based server software. The software is compared across three criteria, efficiency, architecture, and data processing . This comparison helped inform the decision as to which software was the better option for the needs of the project.

From Table 3.3 it is clear that Node.JS was the best choice to meet the needs of the project. Node.JS has an event driven architecture which makes it efficient regarding receiving data and transmitting data, as the program will automatically sleep while waiting for an event to trigger the program. The use of worker threads to accomplish tasks makes it possible for Node.JS to perform different actions simultaneously and the asynchronous structure helps in management of programs along with requests from clients. Although Node.JS has a complex architecture to use and can be hard to keep track of processes, it is the most suitable for high traffic situations which will be needed in this project due to the large amount of data processing and requests sent to the server.

3.5 Project Planning

Google Sheets was utilized to organize the timeline of the project. The tasks needed to be completed and the deadlines and milestones for these tasks are outlined in the Gantt Charts in Figures 3.3(a), 3.3(b), and 3.3(c). Academic years at WPI are divided into four quarters, known as terms, where each term consists of 7 weeks. The Gantt Charts shown in the figures below were divided into three sections as the project was completed in three terms (A term 2020, B term 2020,

and C term 2021). The week of a task deadline is highlighted in red and milestones in yellow in the chart. The rest of the weeks were noted for work in progress and highlighted in blue.

A majority of the literature review for the project, which included research on cloud infrastructures, GNU Radio, and other similar spectrum observatories, was conducted over the summer leading up to A term and continuing a few weeks into the beginning of A term. In A term, the team focused on setting up the testbed for the SDRs, consisting of one or two USRP 2901s connected to a headless host computer with Ubuntu and GNU Radio installed. As testing was conducted on each USRP, the implementation of the server also took place. By the end of B term, each module of the project (e.g., data collection by the Radios, server execution, website implementation, and remote control of radio parameters) was working separately. In C term, the team focused on combining the modules to implement the initial prototype and expanding the prototype to multiple radios. During this last stage of the project implementation, the final report was also written alongside development.

Tasks	Division of Labor			A-Term							
Week			Summer	1	2	3	4	5	6	7	Break
Report Writeups/Presentations	**Leader**	**Assigned**									
Initial Proposal	Joe	All									
Status Updates	Sreeshti	All									
Results/Conclusions	Ian	All									
Final Presentation	Kevin	All									
Revisions	Aneela	All									
Final Report	Joe	All									
Initial Research											
Spectrum Sensing and GNU Radio	Sreeshti	Sreeshti & Aneela									
Cloud Infrastructure	Joe	Joe & Kevin									
Real Time Web Display	Ian	Ian & Joe									
Radios & Data Collection											
Install LINUX	Joe	All									
Install GNU Radio	Sreeshti	Joe & Sreeshti									
Set up radios with 4 channels separated	Joe	All									
Intial Data Collection Test	Sreeshti	Sreeshti & Aneela									
Set up Data Processing and Event Loop	Joe	All									
Cloud Infrastructure											
Determine Platform to use	Kevin	All									
Upload Data to Cloud	Joe	Joe & Kevin									
Remote Control of Radio Parameter											
Create HTML Form	Aneela	Aneela & Ian									
Implement HTTP GET/POST	Aneela	All									
Webpage Display Real Time											
Determine Platform to use	Ian	All									
Sync data between radios	Ian	Ian & Joe									
			Complete:	WIP:	Milestone:						

Figure 3.3(a): A Term Gantt Chart

Tasks	Division of Labor		B-Term							
Week			1	2	3	4	5	6	7	Winter Break
Report Writeups/Presentations	**Leader**	**Assigned**								
Initial Proposal	Joe	All								
Status Updates	Sreeshti	All								
Results/Conclusions	Ian	All								
Final Presentation	Kevin	All								
Revisions	Aneela	All								
Final Report	Joe	All								
Radios & Data Collection										
Set up radios with 4 channels separated	Joe	All								
Intial Data Collection Test	Sreeshti	Sreeshti & Aneela								
Set up Data Processing and Event Loop	Joe	All								
Cloud Infrastructure										
Determine Platform to use	Kevin	All								
Upload Data to Cloud	Joe	Joe & Kevin								
Store data from radio	Kevin	Kevin & Joe								
Sync cloud with webpage	Ian	Ian &Kevin								
Remote Control of Radio Parameters										
Create HTML Form	Aneela	Aneela & Ian								
Implement HTTP GET/POST	Aneela	All								
Integrate into GNU Radio Flow Graph	Aneela	Aneela & Sreeshti								
Webpage Display Real Time										
Determine Platform to use	Ian	All								
Sync data between radios	Ian	Ian & Joe								
Limit delay time in updating webpage	Ian	Ian & Kevin								
			Complete:	WIP:	Milestone:					

Figure 3.3(b): B Term Gantt Chart

Tasks	Division of Labor		C-Term						
Week			1	2	3	4	5	6	7
Report Writeups/Presentations	**Leader**	**Assigned**							
Initial Proposal	Joe	All							
Status Updates	Sreeshti	All							
Results/Conclusions	Ian	All							
Final Presentation	Kevin	All							
Revisions	Aneela	All							
Final Report	Joe	All							
Radios & Data Collection									
Set up radios with 4 channels separated	Joe	All							
Set up Data Processing and Event Loop	Joe	All							
Cloud Infrastructure									
Store data from radio	Kevin	Kevin & Joe							
Sync cloud with webpage	Ian	Ian &Kevin							
Remote Control of Radio Parameters									
Create HTML Form	Aneela	Aneela & Ian							
Implement HTTP GET/POST	Aneela	All							
Integrate into GNU Radio Flow Graph	Aneela	Aneela & Sreeshti							
Webpage Display Real Time									
Determine Platform to use	Ian	All							
Sync data between radios	Ian	Ian & Joe							
Limit delay time in updating webpage	Ian	Ian & Kevin							
			Complete:	WIP:	Milestone:				

Figure 3.3(c): C Term Gantt Chart

3.6 Technical Deliverables

By the end of the project, the following will have been completed and delivered:

1. A fully functioning radio host including radios, a control program, and an upload script to upload the data to the server. This radio host samples the data to be displayed to the end-user and uploads the data to the cloud server through the use of the upload script.
2. A cloud server that receives and processes the data in preparation for the website display. This is one of the primary goals of the project, as creating a real-time cloud-based spectrum observatory relies on a server to both process and provide the webpages and data needed to create a remote display for the user.
3. A front-end website users can access to see live spectrum data being sampled by one or more radios, with the data being updated in real-time. This is the other primary goal of the project; by pairing this live display with the cloud server, the user can access a full spectrum observatory.
4. A user-interface that a user can interact with to change the parameters of the radio (in this case, the center frequency) in real-time while the radio is running. This specific implementation proves it is possible to have remote control over a radio's running parameters and can be expanded to implement more functionality such as changing the radio's sampling rate or other runtime parameters.

3.7 Chapter Summary

This chapter discussed the current challenges with spectrum observatories, how these issues will be tackled, and the approach that will be used to evaluate the success of the project. Different design options were discussed in this chapter, and detailed explanations were given as to why GNU Radio combined with a self-hosted cloud server is the best option for creating a real-time cloud-based spectrum observatory. A plan for implementation of the project was introduced, providing a general overview of the architecture of the project. Gantt charts were used to demonstrate the timeline for each task of the project. Lastly, an overview of the items to be delivered at project completion was provided, as well as the relevance of these deliverables.

4. Sampling Frequency Spectrum

This chapter discusses the process of designing a data receiver utilizing GNU Radio Companion for specific frequency bands using an SDR. The program took in frequency requests from user input on the web interface and extracted data from specified bands.

4.1 Configuring GNU Radio on a Headless System

The radio host computer responsible for running the sampling radios and the upload program and control daemon outlined in the next chapter consisted of a recycled computer provided by WPI. The radio host ran a headless configuration of Ubuntu 18.04, with GNU Radio release version 3.8.2 and Python version 3.8 installed for both the GNU Radio script and uploader script.

For a detailed tutorial on how to install and configure this radio host, refer to Appendix B. Prior to setting up the GNU Radio, a Linux system, Ubuntu 18.04, was installed on the server. An in-depth tutorial of the Ubuntu Server installation can be found in [54], all of the installation settings were kept the same. Next, Python and GNU Radio are required to be installed. Python 3.6 is preinstalled with Ubuntu and this can be verified using the following command:

```
$ python3 --version
```

A set of commands is in a package for GNU Radio 3.8 installation through terminal commands:

```
$ sudo add-apt-repository ppa:gnuradio/gnuradio-releases-3.8
$ sudo apt-get update
$ sudo apt install gnuradio
```

The above three lines of commands adds the package repository to Ubuntu's package manager, updates the list of packages the manager can install and then installs the GNU Radio package. To verify the installation of the software following command can be used:

```
$ gnuradio-config-info -v
```

The code for running the Radio Host can be found and cloned from the following git repository:

```
$ git clone https://github.com/MQPSpectrumObservatory/SpectrumObservatory-Radios.git
```

Now, the USRP 2901 can be connected to the host computer via USB and run the Radio Host using:

```
$ python3 RadioHost.py 1 start
```

Figure 4.1 shows the teams setup of Testbed for Radio Host 1 in WPI dormitory, East Hall using the campus network.

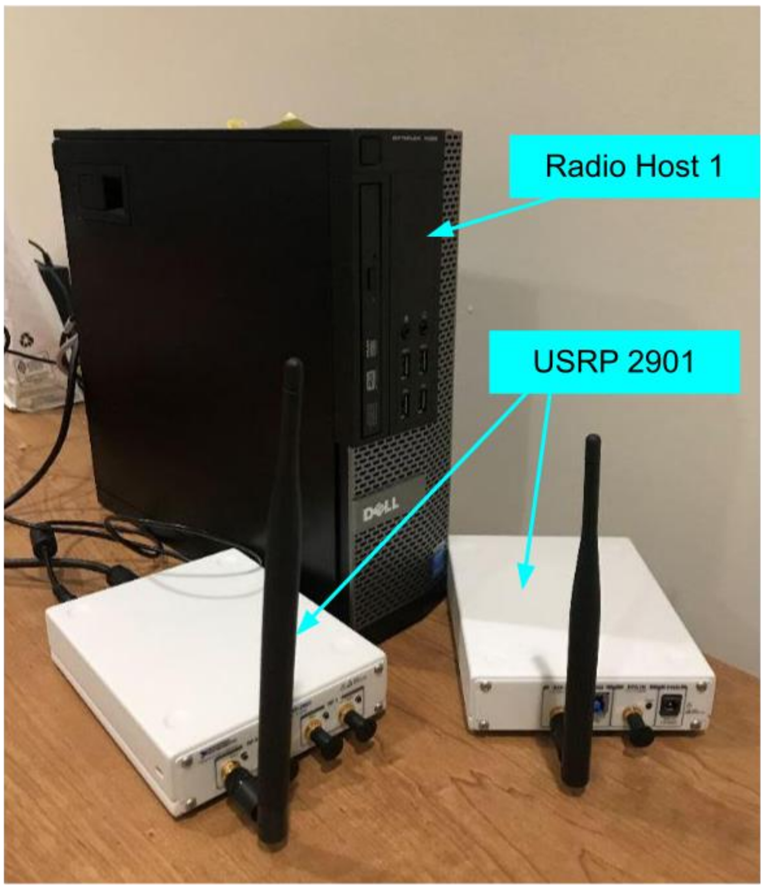

Figure 4.1: Hardware Setup of Testbed in East Hall (Radio Host 1)

4.2 Implementation of GNU Radio

To generate data on the headless host computer, GNU Radio needed to use a non-graphical flowgraph so that the data generation and request could be prompted from the command prompt or the website. Thus, graphical sink and graphical variable control blocks were not used. As shown in Figure 4.1, the generate option in the "Options" block was set to "No GUI" for the flowgraph to generate without using blocks dependent on a user interface. The received signal was processed within GNU Radio using a fast Fourier transform (FFT) algorithm, and this was completed using the FFT block. This block required a vector of floats or complex values as input, and for this implementation, the input type was set to complex. To limit the size of the output data, the sample

rate of the spectrum analyzer flowgraph was set to 300 kHz. The size of the FFT was set based on the number of samples at a time used in each iteration, which is 1024 samples for this frequency sampling.

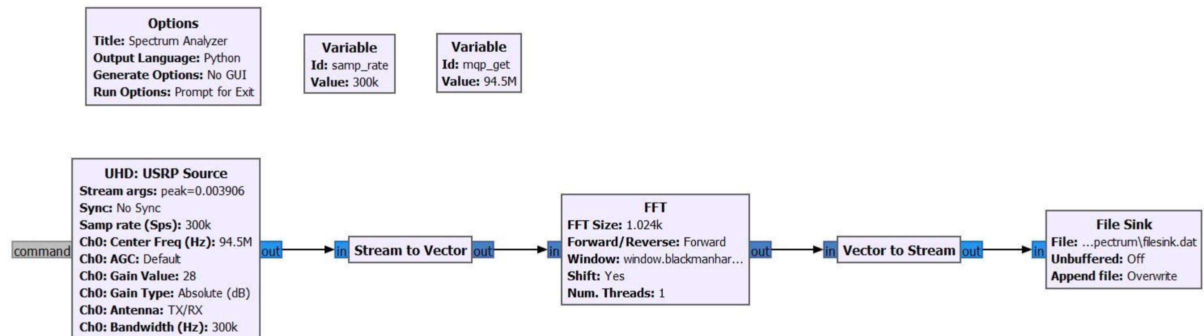

Figure 4.2: Flowgraph of Spectrum Analyzer

As seen in Figure 4.1, two constant variables were set for this flowgraph. The variable "sample_rate" was the sample rate used in both the USRP Source block and FFT block for data sampling and conversion. The bandwidth of the channel was also set at the same value as the sample rate. The variable "mqp_get" stored the center frequency and was used together with the code setting new center frequencies based on user input. The value of the frequency band requested in "mqp_get" was directed to the "Center Frequency" parameter of the USRP Source block. This triggered the SDR to extract data samples from this specific center frequency.

The output of the USRP Source block was a complex sample. The FFT block requires a certain number of samples at a time to calculate an FFT; however, this cannot be done using the output from the USRP Source block directly. Thus, the Stream to Vector block was used to combine all 1024-samples output from the USRP Source block. The Stream to Vector block takes 1024 samples as an input and converts it to a single vector of 1024 samples. The complex FFT block was able to calculate FFT using the output of Stream to Vector. After the FFT had been performed on the signal, the output was then converted back to 1024 samples using the Vector to Stream block. This block is a reverse of the Stream to Vector block; it takes the single vector stream and converts it into a stream of 1024 samples. Once the vector had been converted, the 1024 samples were saved to a .dat file. This .dat file will be appended each time a new sample is written. The full Python file for this GNU Radio flowchart can be found in Appendix E.

Table 4.1: GRC Block Descriptions

Block Term	Function of Block
Options	It sets special parameters for the flow graph. Only one option block is allowed per flow graph.
Variable	Maps a value to a unique variable. The variable can be used in the parameter field of another block by simply using the Variable Block's ID. The parameters are ID and Value (R).
UHD: USRP Source	The USRP source block receives samples and writes to a stream. The source block provides receiving data to downstream processing blocks.
Stream to Vector	Converts a stream of items into a stream of vectors containing Num Items. The input stream can itself be of vectors. The parameters are Num Items and Vec Length.
FFT	Takes in a vector of floats or complex values and calculates the FFT. The parameters are FFT Size, Forward/Reverse, Window, Shift, Num Threads.
Vector to Stream	Convert a stream of vectors into a stream of items. The parameters are Num Items and Vec Length.
File Sink	Used to write a stream to a binary file. This file can be read into any programming environment that can read binary files. The parameters are File (R), Unbuffered, and Append File.

4.3 Challenges

The biggest challenge of frequency sampling on GNU Radio was using a lower sample rate to maintain a smaller data size, allowing for a smaller data file to be transferred and increasing the speed of the overall system. A sample rate of less than 300 kHz would not allow signals to be detected with the antenna and radios used for the project. To get the more detailed signals, a higher

sample rate would be required; however, a high sample rate resulted in a larger data file size, increasing the data upload time and conversion time on the server. Thus, maintaining the sample rate at the 300 kHz boundary was the most reasonable selection to meet both requirements of small data size and high resolution.

4.4 Chapter Summary

This chapter discussed the design and implementation of the sampling spectrum analyzer using GNU Radio Companion. It required selecting a sample rate to maintain low data size and signal detecting and calculating the FFT for signal processing for various frequency bands requested from the website. The flow graph sampled specific frequencies requested from the user on the website and saved the sampled signal on the host computer for the server to push it to the cloud.

5. Program Control and Data Upload

This chapter discusses the steps taken to develop and set up the program responsible for uploading the GNU Radio data to the cloud server and the program which handled pairing the GNU Radio instance and the uploader script to allow the single program to control multiple radios. The control program in its entirety exists as four Python files with the main file instantiating the classes found in the other three Python files.

5.1 Radio Host Control

The program responsible for running both GNU Radio and the uploader script was a daemon process that acted as the entry point for the program. The main program would turn itself into a daemon, the method in which this occurred is outlined in section 5.2, create several child processes equal to the number of radios being run, and set up each child process to run two threads, one thread for GNU Radio and one thread for the uploader script. Figure 5.1 shows the overview of these child processes, threads, and connections between them.

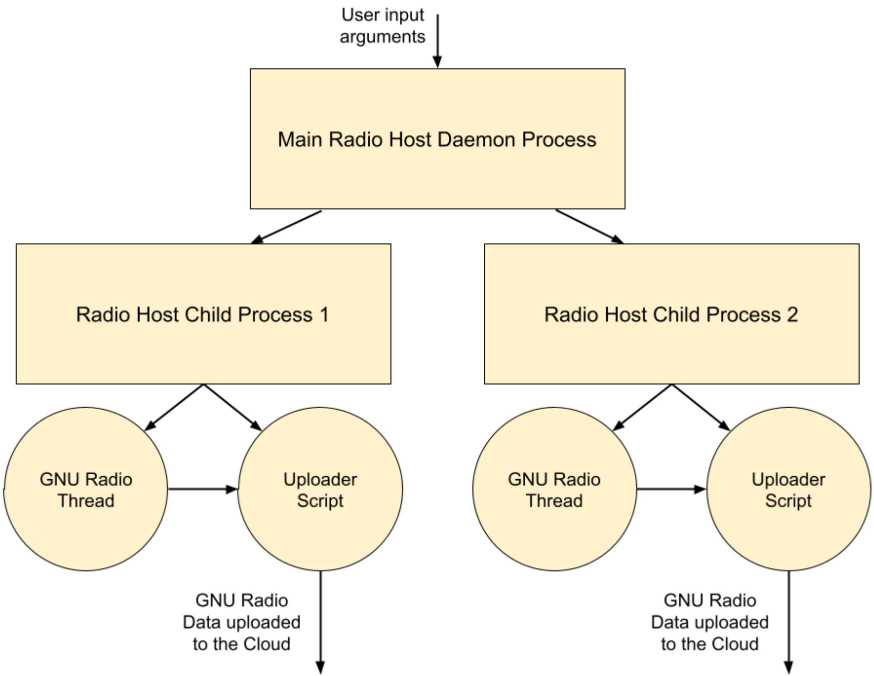

Figure 5.1: Overview of Processes and Threads

When the program was first called, it began by parsing the arguments passed to it. The script took two arguments, one argument indicating the number of radios to run and one argument indicating a daemon command. This first argument was either "1" or "2" as the main program could handle running either one or two instances of GNU Radio and the data uploader. This limit was arbitrarily chosen, and if run on a more powerful computer, the program could likely be configured to run more instances in parallel. The second argument was either the string "start," "stop," or "restart" and indicated to the daemon to either start running, stop running, or restart. The implementation of these control signals for the daemon will be detailed in section 5.2. The command to run the program took the following format with the arguments changing as needed:

```
jrmurphy ~
python3 RadioDaemon.py 1 start
```

After parsing the arguments, the program would set up a logger using Python's built-in "logging" library, allowing it to report informational states and errors to a log file. Figure 5.2 is a sample of the log file. This sample shows the program starting up as well as the first two samples being processed for uploading. The log file includes a timestamp for each message, the log level of the message, the process the message is from, and the details in the message. This log allows the user to monitor the uploading process to ensure the server is properly receiving the data as well as ensuring the program is starting up and shutting down correctly.

```
2021-04-03 08:03:19,831 [INFO] Start Radio Process 1
2021-04-03 08:03:19,835 [INFO] [Process: 1] Starting GNU Radio Thread
2021-04-03 08:03:26,651 [INFO] [Process: 1] Starting Upload Thread
2021-04-03 08:03:26,652 [INFO] [Process: 1] Header Number: 1
2021-04-03 08:03:26,653 [INFO] [Process: 1] Segment too small, skipping

2021-04-03 08:03:26,653 [INFO] [Process: 1] Header Number: 2
2021-04-03 08:03:26,716 [INFO] [Process: 1] Response from server: <Response [200]>
```

Figure 5.2: A sample of the Radio Daemon's log file

After the initial setup, the program entered its main loop, where it began both GNU Radio and the uploader. The program created either one or two instances of a child process class, started each of the child processes, and then waited for them to join during termination. Each child process class had similar behavior to the parent. The class created and started two thread classes—one thread class for the GNU Radio instance and one thread class for the uploader script. The child process then waited for a signal to terminate and end both threads. Both thread classes created an

instance of the GNU Radio class and the uploader class respectively. The behavior of the GNU Radio class was described in Chapter 4, and the behavior of the uploader class is described in section 5.4. The main code body for these process and thread classes can be found in the file *RadioHost.py* in Appendix D.

5.2 Unix Daemon Implementation

For the main portion of the radio host control program to act as a daemon, it implemented a daemon class written in Python. Originally the program used a Python library for converting itself into a daemon; however, this library could no longer be used due to a hardcoded limitation forbidding daemon processes to create child processes. This limitation was introduced to protect against the creation of orphan processes (processes that still exist even after the parent process has been terminated). This was simply a protective measure, not a limitation, and was bypassed by using a custom daemon class. The full code of this daemon class can be found as *daemon.py* in Appendix F.

The class contained five functions, four of which were used to control the daemon, and one to be overridden when the dameon class was subclassed. The four control functions, start, daemonize, stop, and restart, contained standard behavior for the daemon that would remain identical across implementations of the class. The final function, run, was left unimplemented in the daemon class as this function was the function that ran the main program loop of the daemon and was left to be overridden by the process subclassing the daemon class.

The daemon began when the start function was called. This function would first check if the daemon already had an instance running by checking if a process ID (PID) file existed for the daemon. If this file existed, the program would exit, indicating that the daemon was already running. Once the program determined the daemon was not already running, it would call the daemonize function. This function would perform the process outlined in Table 2.2 to create a daemon. After the daemonize function, the run function would be called, and the program would begin its main loop of code as a daemon.

The stop and restart functions were invoked by rerunning the program with either the stop or restart arguments. The restart function is called the stop function and then the start function to restart the daemon. The stop function would first check if the daemon was running through the

same process as the start function, and if it found a PID file, it would send a termination signal to the ID stored in the file until the process ended.

5.3 Data Encoding and Upload

The Python script responsible for uploading data to the cloud server took four steps to prepare and upload the data to the server for each sample generated. These four steps were parsing the incoming data from GNU Radio to separate the metadata from the data payload, encoding the data payload to human-readable characters, formatting and creating the JSON file to be uploaded, and sending the file to the server on an HTTP request. Figure 5.3 below shows a diagram of these four steps and how they interact.

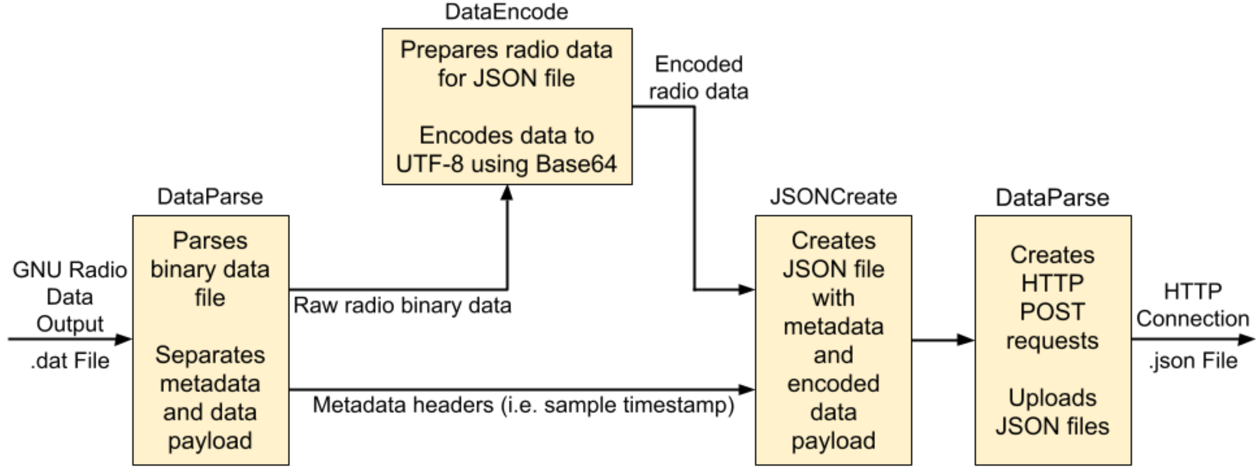

Figure 5.3: Data Upload Script Flow Diagram

The first step, parsing the binary data from GNU Radio, primarily utilized a Python script bundled with GNU Radio. Calling this script converted the metadata out of its polymorphic type into a Python dictionary. With the metadata separated from the main data payload, the script then converted the binary data into Base64, allowing the data to be represented as alphanumeric characters. Base64 represents six bits as a single character, and while this grows the dataset by roughly 33% due to each Base64 character being written as an eight-bit UTF-8 character in the file, it does avoid transmitting raw binary, which could be unintentionally interpreted as a control signal during the transfer. With the data prepared to be sent, the next step was to format this data

into the JSON file that would be uploaded to the server. A sample of the formatted JSON file is shown below. (In a proper file the “payload” field would be much longer).


```
{
    "metadata":
    [
        {
            "rx_time":1.0807631283682877,
            "rx_sample":299999.9990650252,
            "rx_freq":900000000,
            "radio_num":1
        }
    ],
    "payload":"QLcNusAcZDqu"
}
```


Python’s built-in JSON library handled creating this JSON file and the formatting of the file’s data. Lastly, the JSON file was sent to the cloud server using Python’s HTTP library “requests”. The program would send the file and await a 200 OK response from the server. Once the dataset was successfully sent to the server, the program would wait for the next dataset to be generated and repeat the process until the output data from GNU Radio stopped. At that point, the loop continually processing and uploading data would terminate, and the class would handle its own cleanup. This cleanup consists of closing and removing any leftover data files and joining with its parent (as it exists in a thread).

This uploader script exists as a Python class in file *transfer.py*, and the full source code can be seen in Appendix G.

5.4 Chapter Summary

In this chapter, the process of running GNU Radio and uploading data from the radios was outlined. The program, when run, would turn itself into a daemon and create child processes for both GNU Radio and the uploader script. Once the program was signaled to terminate, it would close these child processes before the parent daemon itself would finally exit. This uploading would ensure the data from the radios would arrive at the cloud server for processing and display on the front-end webpage.

6. Cloud Infrastructure

Cloud computing has revolutionized general purpose and computing in recent years. Cloud architecture has several advantages, such as reducing cost, providing flexibility, and consolidating servers. The emergence of real-time applications such as video streaming, cloud-based gaming, and telecommunication management has created a demand for real-time performance. This section highlights the server's implementation and challenges and solutions associated with the real-time application of data associated with the project.

6.1 Implementation

The web server used in this project was hosted on a virtual machine (VM) provided by WPI. This VM ran the server distribution of Ubuntu 18.04 and ran the server using version 12.20.1 of Node.js. Node Package Manager (npm) was used for managing external libraries.Node.js can be installed using different commands

- $ curl -fsSL https://deb.nodesource.com/setup_12.x | sudo -E bash -
- $ sudo apt-get install -y nodejs

After the installation is done to verify everything is working order the commands $ node -v and $ npm -v can be used. The code used to build the server can be found in the Github repository in Appendix C.

6.1.1 Socket implementation

The web server initially used the WebSocket protocol for transferring data to and from the server. The use of the "net" library built into Node.js made it possible to create a network API that creates a stream-based socket.

```
net.createServer( connectionListener: function(sock : Socket ) {//creates a server with a listener for a socket
    console.log('CONNECTED:',sock.remoteAddress,':',sock.remotePort);

    // prints socket info
    var buffers = []
    sock.setEncoding("ascii"); //set data encoding (either 'ascii', 'utf8', or 'base64')
    sock.on( event: 'data', listener: function(incomingData) {// event name  of data,  listens for data
        console.log(sock.bytesRead+ " bytes");

        buffers.push(incomingData)
        console.log("receiving data batches...")


    }).on( event: 'end', listener: function(){
        console.log("finished receiving all batches");
        console.log(sock.bytesRead+ " final bytes");
```

Figure 6.1: A snippet of the code used to create the server along with creating the socket

Figure 6.1 is a snippet of the code as used with the net class. The first line of code, `net.createServer`, was a way to initialize a server with the net class library with a listener function inside that was the socket. The server would listen for a connection for the socket and when the socket was connected It would print out the number of bytes and create an empty buffer that could hold the data. The data was pushed onto the buffer until all the data was transmitted.

As the data would come in, the server would log the number of bytes and begin parsing the data to find headers. The headers were removed along with converting the encoding from Base64 to binary. Having the code in binary form would help format the data to the CSV files. The arrangement of data began with arranging the bits into 32-bit arrays and then splitting the arrays into two arrays per row. This made it so there would be 64 bits in a row, but each column would have two 32-bit words. This structure was needed as it would be easier to reference the data parse through the data to extract the information. The data was then written into a CSV file with the format and saved for the website to reference.

Figure 6.2: Server-side Data Processing

Figure 6.2 illustrates the system of events that occur starting with the incoming of data to the creation of the CSV file using the socket implementation. This figure is a visual representation of the server's process.

While looking for different solutions to multiple radio implementations, it was determined using sockets was not a viable method. In the socket implementation, there would be either several independent connections with one per radio or a single connection shared between several radios. This presented several issues, such as a slower data processing rate, and extra difficulties implementing many separate data pipelines due to the consequence of these multiple connections slowing the server down. Using one pipeline for the data would be viable however the parsing of data would slow down the server, and the server would need to classify commands back to the radio to specify what radio it would be trying to communicate with. The solution to this problem was a different architecture than bi-directional sockets. The new protocol considered was Hypertext Transfer Protocol (HTTP). This architecture was different as there would be no direct data connections between the radio host and server. Instead, a message (also referred to as a request) would be created with a sender and a receiver and would not consist of an always-open communication path. Having requests and responses made bi-directional communication a possibility. The architecture made it so that the system was dynamic and asynchronous.

6.1.2 HTTP Implementation

To create the new server implementation, new code was written with the main differences being the references to the HTTP class along with the switch cases for different GET and POST scenarios that the server would handle.

```
61  const server = http.createServer( requestListener: function (req : IncomingMessage , res : ServerResponse ) {
62      console.log(req)
63      switch (req.method) {
64          case "GET":
65              var dir = path.join(__dirname, 'public');
66              var req_path = req.url.toString().split( separator: '?')[0];
67              var filteredPath = req_path.replace(/\/$/, '/index.html');
68
69              var file = path.join(dir, filteredPath);
70              if (file.indexOf(dir + path.sep) !== 0) {
71                  sendCode(res,  code: 403,  msg: "403 forbidden");
72                  break;
73              }
74              var type = mime[path.extname(file).slice(1)] || 'text/plain';
75              var s = fs.createReadStream(file);
76              s.on( event: 'open',  listener: function () {
77                  res.setHeader( name: 'Content-Type', type);
78                  s.pipe(res);
79              });
80
```

Figure 6.3: HTTP implementation of server using switch statements

Figure 6.3 shows the implementation used to create a HTTP based server. This server was created by referencing the HTTP library, as seen on line 61. The switch statement was used to process incoming POST and GET requests. The GET statement refers to any information that other clients need from the server. The POST statement refers to any information clients try to send to the server. It was important to clarify to the server what action was occurring.

Using a server built with HTTP made everything smoother as the radio host would transmit the data, and the server would have an address for where the data would appear. Using this method made it much easier to expand the number of radio nodes the server could support. Each new radio would be provided with a specific address to interact with. This reduced the likelihood of data overlapping as each dataset would arrive at its specified address. Additionally, each address could have different functions that could process the data differently, allowing individual radio behavior if needed. The implementation of HTTP requests used switch statements to parse and route the request efficiently and these statements were made to identify what actions were occurring in the server. For example, if the radio host were requesting information, the radio host would need to specify this to the server by referencing the GET case and the address it would like to receive the information from. If the radio wanted to transmit data, it would need to specify this to the server by referencing the POST case. This made it so the server would know whether it would be receiving information or sending information. The processing of data was done in the POST case when data was uploaded, and the processing of data was not changed from the socket

implementation. The addressing was added to the GET case to specify what information the radio host would request. This implementation is the current implementation, and the full source code of the server can be seen in Appendix H.

While implementing the HTTP server helped solve some of the major issues such as the dynamicity of the architecture, other large issues, such as the conversion rate were problems. The conversion of bytes to a UTF-8 string of binary took some time but not as long as writing the CSV file. This took too much time for the project to be considered real-time.

6.1.3 Data Processing and the Event Loop

The high amount of data processing required to convert the incoming JSON file to the required formatting in the outgoing CSV file caused the server to have slowdowns and not be as responsive as necessary. The long amount of time taken by the data processing caused the server's event loop to become blocked, and as such, if a request came in while the server was still processing the previous request, the new request could not be handled until the previous finished. To resolve these issues, several changes were implemented. The primary changes involved the introduction of worker threads, optimization of the data processing functions, and changing each incoming set of data to be smaller but sent more frequently.

Optimization of the data processing primarily consisted of ensuring the data was iterated through as few times as possible. Since the required processing required iterating through the data several times, one of the most effective optimizations was ensuring the data was only iterated through as many times as strictly needed. By doing several operations per iteration instead of iterating through for each operation, the processing took less time but still was not as quick as needed for a real-time server.

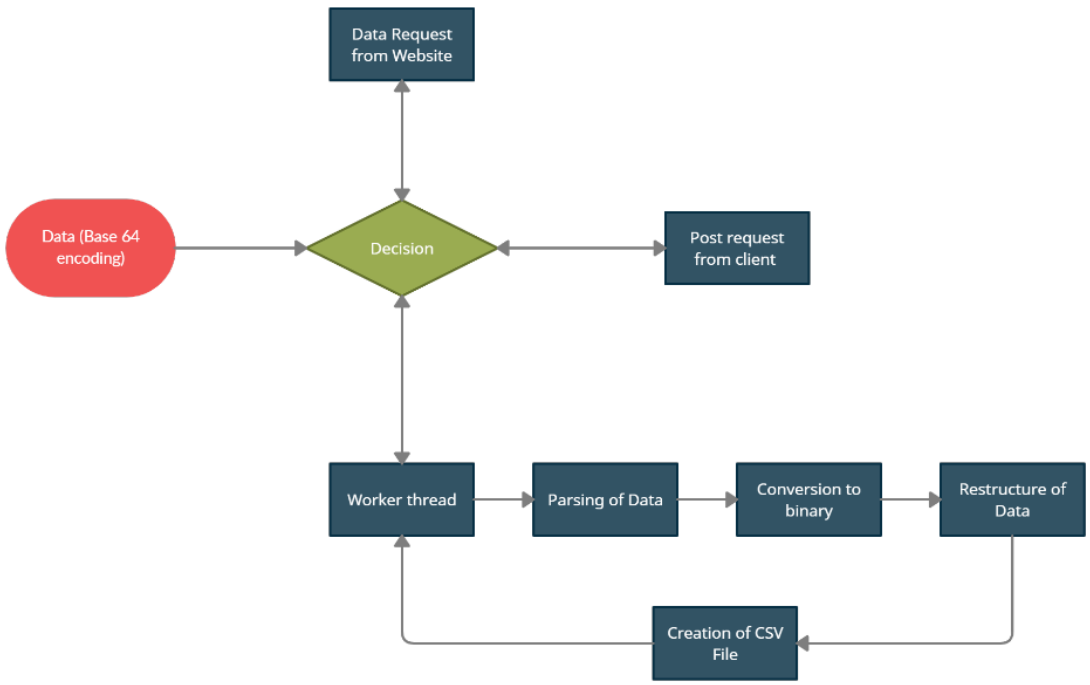

Figure 6.4: Web Server Processing as data is transmitted and received by the server

Figure 6.4 illustrates the threading of commands and requests that occur starting with the processing of data to the creation of the CSV file using the HTTP implementation along with event loops. In this figure, the Base64 data and the post requests are inputs that feed into the system the server's main event loop as the main event loop is what delegates action to the worker thread to work on the data. The data is parsed, converted to binary and restructured to be written onto the csv file. Once the worker thread is done the main event loop stores the data to be referenced by the website. The post request is an input to the system as the main event loop forwards this request to the GNU radio for the change of frequency.

This figure is a visual representation of the server's process, the worker thread focuses on the data conversion while the main event loop listens and awaits for data or requests coming from the client or website.

The two changes that had the most significant increase in server performance were decreasing the size of the incoming data (and, as a result, increasing the frequency of the incoming data) and delegating the data processing to a worker thread instead of it blocking the main event loop thread. Due to the nature of the data and the operations being performed, working on smaller

sets instead of one large set took significantly less time despite the total amount of data being operated on remained the same. Additionally, since the processing was being performed in more frequent but shorter bursts, it gave the server more opportunities to handle different events, such as responding to a request from a client for the main website HTML files. The other fix put in place to ensure the server would be responsive was to delegate the data processing to a separate worker thread. By creating a worker thread, the server's event loop is free to continue to listen for events instead of needing to perform the data processing. The main thread called the worker thread and passed the data, and the worker thread processed the data in the same method as before but did not block the event loop. Once the worker thread was finished, it joined with the main thread, allowing the server to use the processed data as necessary. While the implementation of worker threads did not noticeably speed up the server in this small-scale implementation, it ensures the server can stay responsive if the implementation is scaled up and the server needs to handle a larger amount of data and/or a larger amount of incoming requests.

The creation of the worker threads was done using a built-in Node.js library. The code executed by the worker thread existed in a separate javascript file (in this case, this file is *worker.js* and can be seen in Appendix I), and the file was provided to the instance of the worker thread object when it was created. Figure 6.5 shows the code responsible for the initialization of a worker thread.

```
// Spawn a worker thread that runs the file worker.js
function processData(pathname, reqBody) {
    return new Promise((resolve, reject) => {
        const worker = new Worker('./worker.js', { workerData: { pathname, reqBody } });
        worker.on('message', resolve);
        worker.on('error',   reject);
        worker.on('exit', (code) => {
            if (code !== 0)
                reject(new Error(`Worker stopped with exit code ${code}`));
```

Figure 6.5: Code sample of the function responsible for creating a worker thread

The function begins by creating a Promise. This is a data structure in Node.JS that is related to asynchronous work. The Promise indicates to the program that the function will return data at some point in the future, but the data is not currently present. In this case the Promise acts as a placeholder for the indication that the worker thread completed its task successfully. Next, the constructor for a new Worker is called. The constructor is passed the new JavaScript file it is to execute, as well as any data needed. Once the worker is finished it can send several events that

the Promise uses as the data it was holding place for. In this implementation, if the Worker responds with a message, the Promise knows the Worker finished successfully.

6.2 Chapter Summary

This chapter discussed the different implementations used and the different reasons as to why the implementations were chosen or switched. The movement of data was examined from when it arrived at the server, to when it was saved for the website to reference. The different functions used were highlighted and their purposes explained. The solution to the slower response time was explained using the knowledge of event loops and worker threads.

7. Spectrogram Display

This chapter describes the methodology and thought process for the creation of the website that is used to display the spectrogram, as well as some of the problems that arose during the coding process. The website as all websites are, using a combination of HTML code for the visual design and javascript for the background processes, in conjunction with the use of the D3 visualization library for file IO and creating the graph.

7.1 Implementation Overview

The D3 javascript library is a tool utilized in industry mainly as a method for showing single-use statistics related to business trends or other statistical measurements. The website uses this library as well due to the method in which files are saved on the server rather than using a constant stream of data. The current method works as a proof-of-concept for a cloud-based spectrum observatory but can be improved upon by changing the way the data is served and the way the data is displayed.

On the server-side, separate folders exist that represent each radio the website is expecting to display and the data from each radio is stored in each of the separate folders. The website works by utilizing asynchronous functions called AJAX or "Asynchronous Javascript and XML" that allows the website to call functions and carry out processes that fall outside the usual order of the program. The website uses a new instance of AJAX for each radio, and inside each asynchronous block, the code performs multiple functions. The first function is to access the folder for the corresponding radio and retrieve a list of all the files that contain data inside it. The function loops through each of these files, creating a new JavaScript file request for each of the files, calculating the magnitude of each pair of numbers in the data files using the magnitude equation, and graphing that data as a line.

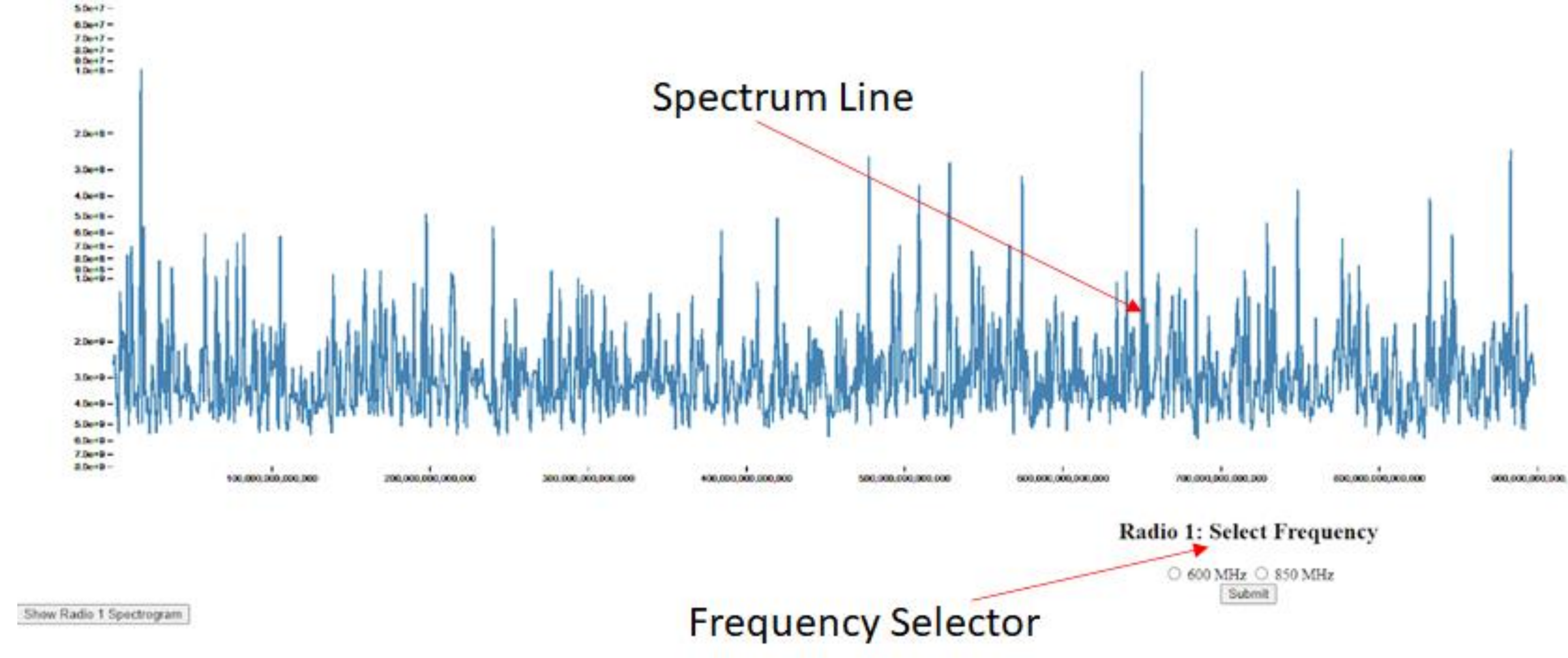

Figure 7.1: Spectrogram Display on Website

The final functionality of the website is to allow a user to change the frequency of one of the radios. It performs this function by displaying a clickable button that, when clicked, creates a push request containing the number for the selected frequency that is then handled on the server-side.

7.2 Displaying Data

The display of the data is implemented using the D3 library which takes several steps between data being collected and put on the server, and actually showing the spectrogram on the website. The first step after getting the names of each of the files the data is stored on is to access the files and store the data onto a more accessible local variable. This is accomplished by using a function in the D3 library called d3.text() that reads in the contents of a file as a selection of strings. It then uses another function d3.csvParseRows() that parses through the string variable storing the data and interprets it as a CSV file that can be separated into its real and imaginary components. Below is the code that is used to both access the data stored in the files, parse it into separate rows so that each sample is an array with two positions that store the two parts of each frequency sample, and then assign each part to its own variable for future calculations.

```
d3.text(pathName, (function (d) { //Reads in the CSV as a text file
a = d3.csvParseRows(d);
samp_freq = a[1][0];
a.forEach((item, index) => {
  I.push(parseInt(item[0], 2));
  Q.push(parseInt(item[1], 2));
});
```

After the data has been properly parsed, the next step is to change it from real and imaginary frequency samples into magnitude data for display. This is done with a simple function that uses the magnitude equation (4). The numbers that result from this represent the y-axis values that will be displayed on the final graph. Directly after the magnitude is calculated, the numbers for the x-axis need to be computed for display in a future function. This axis is easier than the magnitude function to calculate and uses (1) and (2) to get the minimum and

maximum frequency values that will be displayed, and then (3) to create all of the in-between points that make up the x-axis, both of these functions are shown below.

$$X = \sqrt{A^2 + B^2}$$

(4)

```
function calcMag(I, Q, mag) {
	for (i = 3; i < I.length; i++) {
			mag.push(Math.sqrt((I[i] * I[i]) + (Q[i] * Q[i])));
	}
	return mag;
}

freq = linspace(samp_freq-I[0]/2, samp_freq+I[0]/2, 1024)
```

With all of the data properly received, calculated into its final sets, and ready to be displayed, the final step is to create the objects necessary for showing each axis, the background that sits behind the graph, and the line itself that is graphed. This part utilizes the D3 library again. The first step is to form the background of the graph that the axes and line will be laid on top of, the only variables that need to be set on this are the height and width of the window the graph will be drawn in.

```
var svg = d3.select("#my_dataviz")
    .append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .append("g")
    .attr("transform",
      "translate(" + margin.left + "," + margin.top + ")");
```

With the background defined, the next step is to draw the x and y axis that correspond to the frequency and magnitude. This is completed by defining the maximum and minimum values of each axis by getting the maximum and minimum of the arrays that hold the frequency and magnitude data, and by defining the scale of the axis. The frequency scale is linear, while the magnitude axis is logarithmic. Finally, the line that represents the spectrum needs to be

displayed, this is completed by accessing the x-axis of frequency values, and the magnitude data and appending a path type D3 object to the graph, rendering the line itself.

```
svg.selectAll("path")
  .datum(data)
  .attr("fill", "none")
  .attr("stroke", "steelblue")
  .attr("stroke-width", 1.5)
  .attr("d", d3.line()
    .x(function (d, idx) {return x(xaxis[idx])})
    .y(function (d, idx) {return y(data[idx])})
  );
```

7.3 Challenges

The largest challenges that hold the website back are related to CPU usage and memory issues that arise during website runtime. These challenges exist due to the method the website uses to gain access to the data that is displayed. The website is required to ask for the contents inside 100 plus files, it performs this action very quickly, and then does this multiple times as the files inside change and update. This process is extremely taxing on the CPU as the browser is asked to load in large amounts of data from an external data source over Wi-Fi.

The memory problems come from the way that browsers load files and create the objects necessary to draw the spectrum line. Even after the data has been overwritten in the variable that is being used to draw the line, the browser will still hold onto the data, slowly taking up more and more memory.

7.4 Chapter Summary

This chapter discusses the implementation and challenges associated with the website used for the display of data. The website works by gaining access to the files that are used to store all of the spectrogram data which is the cause of a memory leak due to how internet browsers function keeping the data cached even when it is no longer being used. It then creates the background for the display of the graph and the line that represents the spectrogram, and loops through each file in the associated radio's folder for display purposes.

8. Remote Control of Radio Parameters

In order to remotely change the center frequency of the SDRs during run-time, the establishment of data flow from the client's web browser back to the cloud server and radio host is required. This chapter explains the process of implementing control of radio parameters remotely from the client interface of the website. User input is collected using HTML forms and then by utilizing HTTP GET/POST this backward flow of data is implemented.

8.1 HTML Forms

An HTML form is a section of an HTML document that is used to collect user input. They consist of web content, markup and unique elements called controls/input. Input can be text entries, checkboxes, menus or radio buttons. These inputs are usually labeled to help the user understand what information is being requested. An HTML form is "completed" by a user by modifying these inputs, for example - by entering text, selecting an item on a menu, clicking on a radio button or by checking a box. Modification of controls/input is conducted before the form is submitted to a web server for processing. In this project, since spectrum data will be displayed on a website, an HTML form was the most suitable option to get user input.

Each SDR was designated with two center frequencies for spectrum sensing. The user would be allowed to select one of the two center frequencies for viewing spectrum data at a time. As a result, the control type selected for this HTML form was radio buttons. Radio buttons are like checkboxes but when the radio buttons share the same input name, they are mutually exclusive. This type of selection is perfect for center frequency as the SDR can only scan at one specific frequency channel at a time.

The pseudo code below shows the HTML form element just like any other HTML element had a start tag (green highlight) and an end tag (pink highlight). The start tag included the attributes (bright red text color) "action" and "method". The "action" attribute is used to denote the HTTP URI so the server can process the data. The "method" attribute used in this form is HTTP POST as data is submitted to the server.

```
<h2> **Element Content** </h2>

<form action="URI" method='POST'> **Element Content** </form>
```

Two radio buttons using the input tag were created with the same name “CenterFreq”. The value of the first radio button was set to be "600000000" and the second was "850000000" to denote frequencies 600MHz and 850MHz. The HTML source can be found in Appendix J. Below is a snippet of the source code for the HTML form.

```
<form action="http://spectrumobservatory.wpi.edu/post1" method='POST'
        style="text-align:center">
  <input type="radio" id="CenterFreq" name="CenterFreq" value="600000000">
  <label for="CenterFreq">600 MHz</label>
  <input type="radio" id="CenterFreq" name="CenterFreq" value="850000000">
  <label for="CenterFreq">850 MHz</label><br>
  <input type="submit" value="Submit">
</form>
```

When one of the radio buttons is selected their respective value is sent to the server where the server stores this data and sends it to the host computer. When a radio button is selected it is highlighted in blue. Figure 8.1 shows the output of the HTML form when displayed on the Website.

Radio 1: Select Frequency

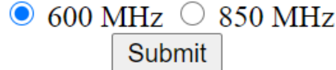

Figure 8.1: HTML form output

8.2 HTTP Method GET/POST

The Hypertext Transfer Protocol (HTTP) was created to establish communication between clients and servers. GET and POST are the two most common HTTP methods. GET is used when the user needs to request data from a specific resource/server. On the other hand, POST is used to

send data to a server with the intent of creating or updating a resource. A comparison of GET and POST is shown in Table 8.3.1.

Table 8.1: Comparison of HTTP GET vs. POST

GET	POST
Can be cached	Never cached
Remain in browser history	Does not remain in browser history
Has length restrictions	No length restriction
Cannot modify data of a server resource	Can modify data of a server resource

For this project, both the HTTP GET and POST requests were utilized as shown in Figure 8.2. The POST request was used to send the user input (from the HTML Form) to the server as discussed in section 8.1. This was set up using the attribute method of the HTML form. As described in Section 6.1.3, the POST request was set up ro be an input to the Web Server. This updates the center frequency variable of the server according to user input and the main event loop of the Web Server forwards this request to the GNU Radio. During the testing stage, GNU Radio was used to program the SDRs and a python file was generated called top_block.py. This python script was used by the host computer later to scan the spectrum. In Figure 8.2 it can be seen that HTTP GET was used in this python script to fetch the frequency value from the server and change the center frequency of the SDR.

Figure 8.2: Flow of Data using HTTP Methods.

8.3 GNU Radio Python Script

After the value of the center frequency has been stored in the server, the next step is to fetch this information. This will be accomplished using an HTTP GET in the python script called top_block.py.

When the GNU Radio Companion executed the spectrum sensing flowgraph, the top_block.py file was generated (shown in Figure 8.3).

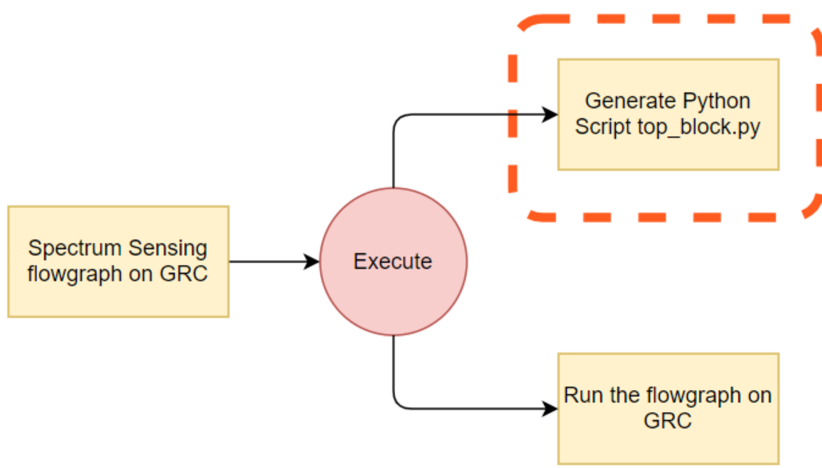

Figure 8.3: Generation of python script top_block.py

It looks like any other python script with the modules being imported at the top. In this segment, a few new modules (requests, time, threading) were added. These modules are required to fetch the data from the server. To use HTTP methods on Python, the module requests need to be imported to that python script. The module "time" was added because the GET request will be sent every five seconds. The threading module was imported to implement this while loop using a separate thread.

A function MQP_HTTP_Parser() was created to write a while loop for parsing the data from HTTP GET. Inside the while loop, a variable freq1 was set to hold the value of the GET request. Once the content of the GET request was received using the command (requests.get()).content, it was seen that the value of the frequencies was in the form of a string instead of a float. To fix this issue, Python's built-in function int() was used to convert the variable from a string to an integer. Below is a snippet of the while loop in the python script.

```
def MQP_HTTP_Parser(self):
        while True:
            freq1 =
int((requests.get(f'http://spectrumobservatory.wpi.edu/freq{self.radio_num}')).content)
            if(freq1 != self.get_center_freq): # Only set frequency if it is different
(save on overhead)
                self.set_center_freq(freq1)
            time.sleep(5)
```

The URL of the GET request was replaced by the address where the server stored the value of the center frequency. At this point, the thread waited for five seconds to create a delay between GET requests. Another function called startThread() was defined to create a thread that executes the while loop. This threading function calls the initial HTTP parsing function.

```
def startThread(self):
        self.t1 = threading.Thread(target = self.MQP_HTTP_Parser)
        self.t1.daemon = True
        self.t1.start()
```

The final step is calling the threading function in the main() of top_block.py thus changing the parameter of the radio during run time. The source code for top_block.py can be found in Appendix E.

8.4 Chapter Summary

This chapter discussed the flow of data from the front-end (webpage) to the computer hosting the SDRs. It requires writing code in three different programming languages HTML (creating the HTML form), JavaScript (creating a simple test server), and Python (modifying top_block.py). The HTML form utilized radio buttons that helped the user select one of the two frequencies for each radio which was then sent to the server. The data was further sent to the host computer from the server. This data was then parsed and ultimately used to change the frequency of the SDR during runtime.

9. Results and Discussion

This chapter discusses and explains the results of the website display, data transfer from the radios to the cloud, and the data collection from the radios by examining the delay between collection and display, accuracy of the displayed spectrum, and the delay between a user pressing the button to change the frequency the radios are being sampled.

9.1 Data Transfer from Radio to Website

Each step the data takes before it can finally be displayed on the website causes a delay. These delays are measured inside the functioning of each section, and added together make up an average delay for the end-to-end system functionality. The largest delay is the time it takes for the data to go from being stored on the server to being displayed on the website. As the radio stores a new file worth of samples every second, and the website takes longer than one second to display every file the delay between the data being collected and getting displayed rises. Compounding this with the slowing of the display rate on the website results in the time between collection and display going from less than five seconds to far more than a minute. Getting exact numbers for the time delay from the radio computer to the server is not a feasible task. This is due to the time delay being entirely reliant on the speed of the internet connection between each of the steps. Figure 9.1 shows the amount of time it took for each file to be accessed and drawn, the data shows a strong linear upward trend showing that as more files get requested the total time takes longer. Figure 9.2 shows the change in the amount of time it takes to open and display the data files, further showing that it stays as a linear increase.

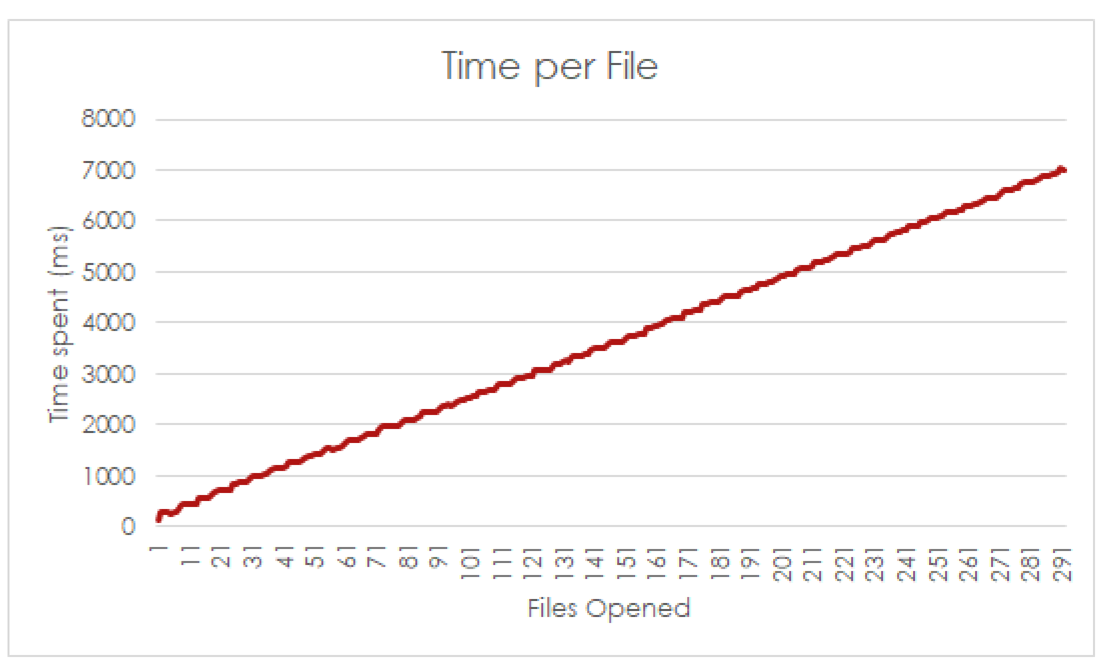

Figure 9.1: Time to Draw Data per File

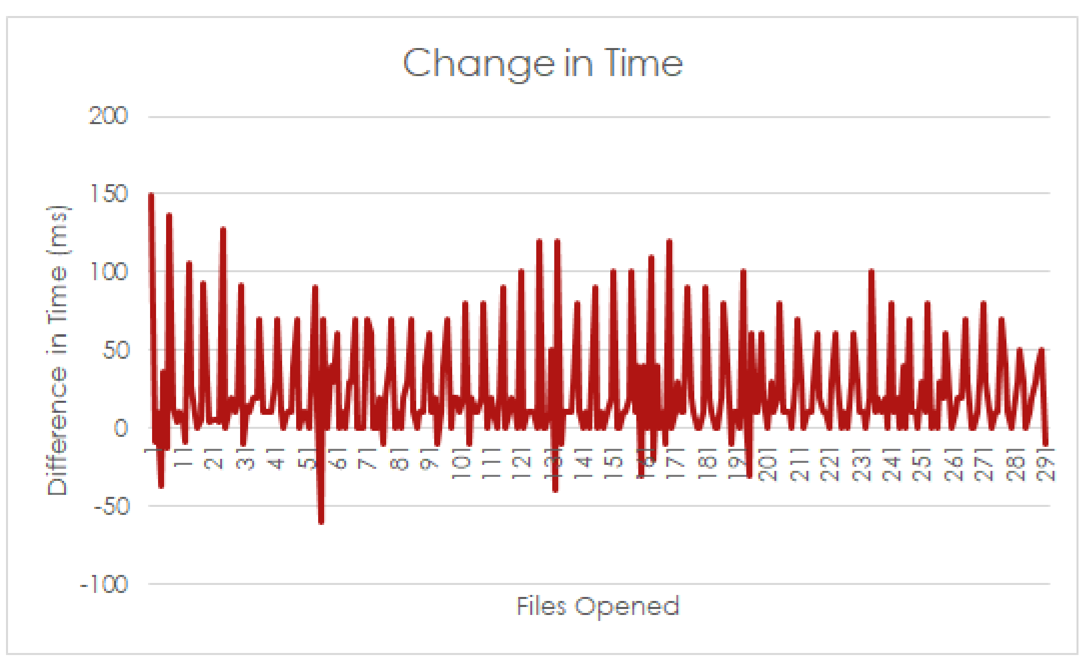

Figure 9.2: Change in Time to Open Files

The second measurable result is the accuracy of the data. There are only a few times data could become corrupted and thus display inaccurately, specifically during transfer of data from the radio computers to the server when the data is encoded, sent to the server, and then decoded, and

during the conversion of the binary data to a CSV. Each of these points is evaluated for accuracy by comparing to the raw data that was first output from the radios and is still stored on the computer. The other way to confirm the accuracy of the spectrum data is to compare the visualization graph on the website to the spectrum graph that can be created using GNURadio. Using the first method shows that the data received on the website is the exact same as the data collected by the radios, meaning there is 100% accuracy from end-to-end. Using the second method is more difficult due to the delay between collection and display, but also shows that there is a similarity between the graph the website shows and the graph GNURadio creates, implying the accuracy confirmed by comparing the exact data files is representative of the accuracy of the entire end-to-end implementation.

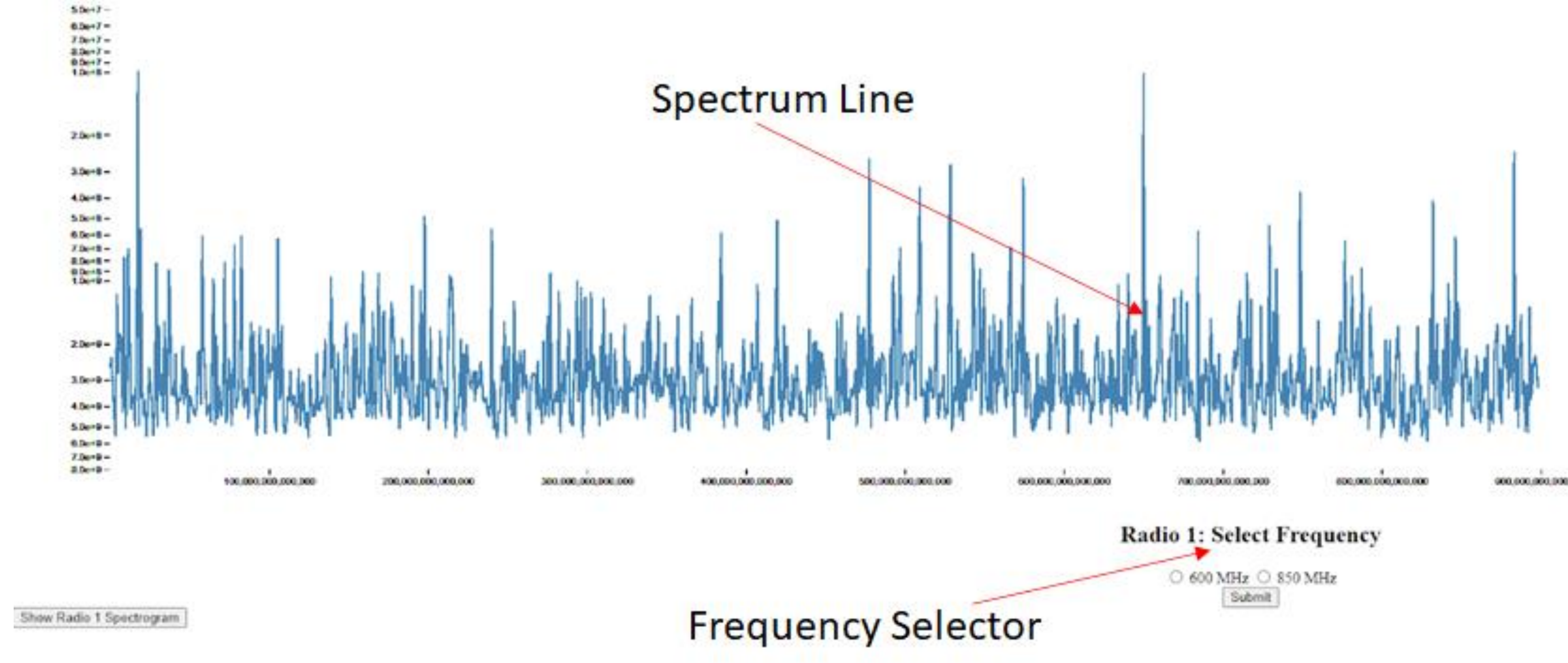

Figure 9.3: Sample Spectrum Graph from the Website

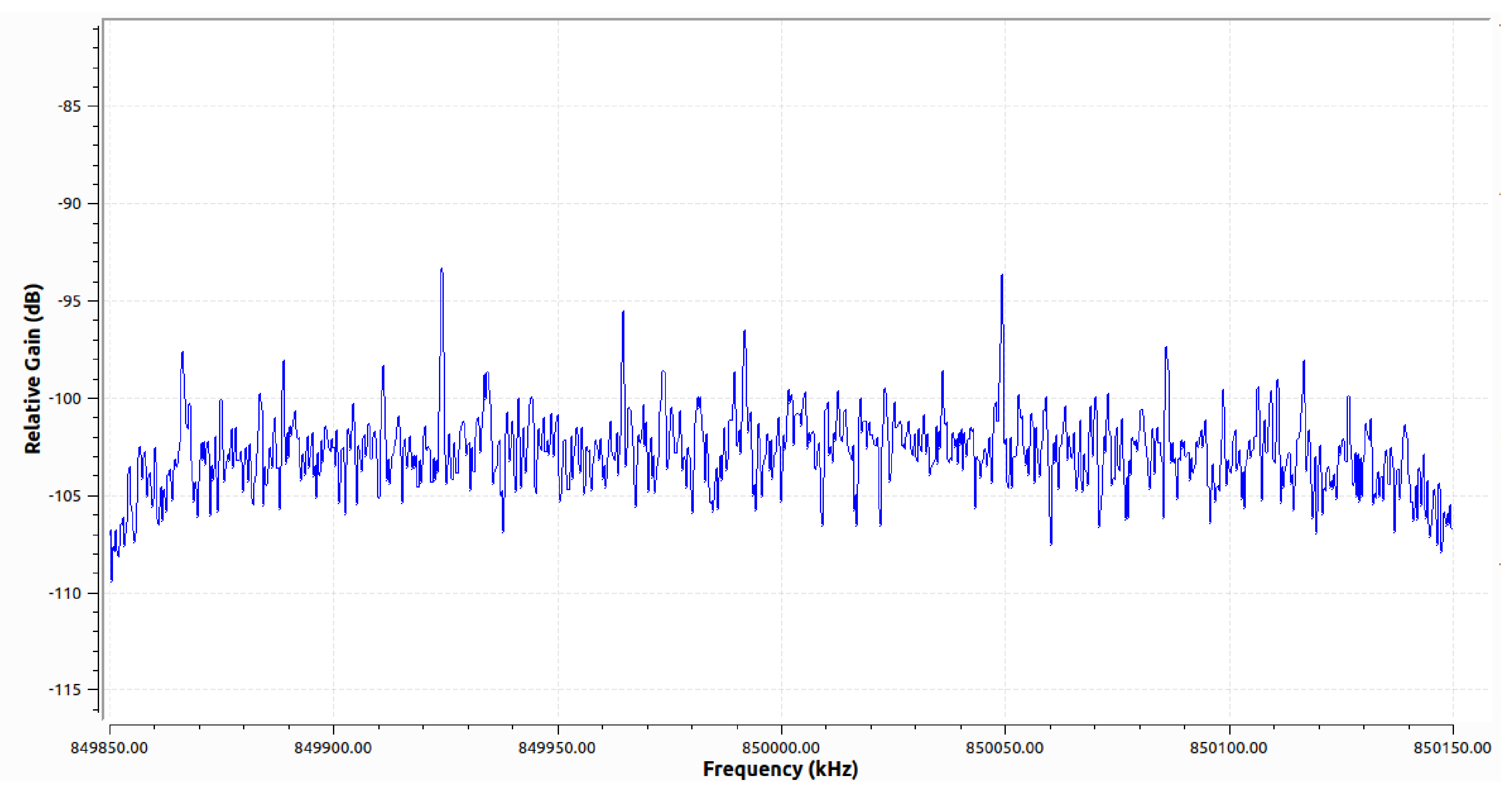

Figure 9.4: Spectrum Graph from GNU Radio

9.2 Frequency Change Request

The other main function that the website has is to send back a request to the radios to change the center frequency the data is being recorded at. This process is completed by choosing a frequency and clicking submit, creating an HTML form that is sent through the web server to the python script that controls the radios, and finally, changes the radio's frequency, this flow is shown in figure 9.5. The time for this, much like the delay for data to be sent, is affected by the internet connection the user is operating on, though it is more limited in the amount of time it can take due to GNURadio asking for the frequency to center too every five seconds. This means that the minimum amount of time it would take to change the center frequency is five seconds.

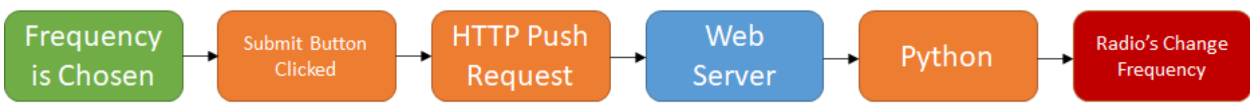

Figure 9.5: Flow of the Frequency Change Request

9.3 Chapter Summary

This chapter discussed the results related to the timing of data transfers from the radios to the website and back again for changing the center frequency, and investigated the slow down points that keep the website from functioning smoothly. By graphing the response time for data on the website to be displayed the setbacks on data visualization are brought to the forefront and serve as a jumping off point for any future project that would want to improve on this implementation.

10. Conclusion

The project utilized several SDR (USRP 2901s) along with a fully built server to create a real time spectrum observatory. The SDRs were used to detect activity in different ranges to provide for a wholesome frequency spectrum. The use of a server was needed to help manage reorganization of information and helped with bi-directional information flow. The front end was created to display the frequency spectrum and was made to adapt for changes in the spectrum. This was utilized with a variety of programming languages such as Node.js, and Python. By implementing a flowgraph and modifying the GNU Radio generated Python script it was possible to transmit data from the GNU Radio to the server. The use of Node.js helped connect the server's back end to the front end. These connections helped pass data in an efficient manner and made it possible for the system to operate in real time specifications.

10.1 Future Work

There are many ways for this project to be expanded and optimized to make further achievements in the detection and relay of activity in the spectrum. The detection of radio frequencies can be improved upon by increasing the detection range or adding a few more radios to accurately detect specific ranges. The stream of data between the GNU Radio and server can be made smoother, for example instead of the data having to write the file and transfer it before arriving to the server the GNU Radio would instead stream data directly to the web server. This could be made possible using an in-built functionality in GNU Radio to utilize the ZeroMQ protocol. Further research would be needed to find a better way to transfer data from the back end to the front end, instead of referencing the data in the front end and transferring several CSV files. Further work could also be done on the front end spectrum display. Instead of using a D3 real time-line chart other possible alternatives could be used. Using the current D3 real time chart makes for a slow integration of data.

References

[1] “Microsoft's Spectrum Observatory project opens up for increased collaboration,” *Microsoft On the Issues*, 13-Jul-2015. [Online]. Available: https://blogs.microsoft.com/on-the-issues/2014/04/08/microsofts-spectrum-observatory-project-opens-up-for-increased-collaboration/.

[2] J. Walko, “Verizon Commits $45.5B for 5G Mid-Band Spectrum -,” *EETimes*, 03-Mar-2021. [Online]. Available: https://www.eetimes.com/verizon-commits-45-5b-for-5g-mid-band-spectrum/. [Accessed: 29-Mar-2021].

[3] M. Z. Zheleva et al., "Enabling a Nationwide Radio Frequency Inventory Using the Spectrum Observatory," in IEEE Transactions on Mobile Computing, vol. 17, no. 2, pp. 362-375, 1 Feb. 2018, doi: 10.1109/TMC.2017.2716936.

[4] “Spectrum observatory,” Fundamentals of Networking Laboratory, 2021. [Online]. Available:https://depts.washington.edu/funlab/projects/software-defined-wireless-networks/spectrum-observatory/. [Accessed: 10-Feb-2021].

[5] S. Roy et al., "CityScape: A Metro-Area Spectrum Observatory," 2017 26th International Conference on Computer Communication and Networks (ICCCN), Vancouver, BC, 2017, pp. 1-9, doi: 10.1109/ICCCN.2017.8038427.

[6] “Space aggressors jam AF, allies,” *U.S. Air Force*, 13-Feb-2017. [Online]. Available: https://www.af.mil/News/Article-Display/Article/1081947/space-aggressors-jam-af-allies-systems/. [Accessed: 16-Mar-2021].

[7] J. Ataya and M. Farah, “Jamming Security for LTE Networks,” Worcester Polytechnic Institute, Worcester, Massachusetts, MQP AW1 - CELL, 2020.

[8] Y. Brown and C. Teng, “LTE Frequency Hopping Jammer,” Worcester Polytechnic Institute, Worcester, Massachusetts, 2019.

[9] G. Noorts et al., "An RF spectrum observatory database based on a Hybrid Storage System," 2012 IEEE International Symposium on Dynamic Spectrum Access Networks, Bellevue, WA, 2012, pp. 114-120, doi: 10.1109/DYSPAN.2012.6478122.

[10] *2014 Publications*. [Online]. Available: http://www.cs.iit.edu/~wincomweb/wincomnew/pub2014.html. [Accessed: 25-Mar-2021].

[11] Illinois Institute of Technology Spectrum Observatory https://www.nitrd.gov/nitrdgroups/images/7/79/illinois_institute_of_technology_-_dennis_roberson.pdf [Accessed: 25-Mar-2021].

[12] Z. Luo, C. Lou, S. Chen, S. Zheng and S. Li, "Specific primary user sensing for wireless security in IEEE 802.22 network," 2011 11th International Symposium on Communications & Information Technologies (ISCIT), Hangzhou, 2011, pp. 18-22, doi: 10.1109/ISCIT.2011.6089728.

[13] “Spectrum observatory,” *Fundamentals of Networking Laboratory*. [Online]. Available: https://depts.washington.edu/funlab/projects/software-defined-wireless-networks/spectrum-observatory/. [Accessed: 25-Mar-2021].

[14] What is Radio Spectrum?, 21-Sep-2017. [Online]. Available: https://www.transportation.gov/pnt/what-radio-spectrum. [Accessed: 11-Mar-2021].

[15] C. Holmes, Cell Phone Networks and Frequencies Explained: 5 Things To Know, 21-May-2020. [Online]. Available: https://www.whistleout.com/CellPhones/Guides/cell-phone-networks-and-frequencies-explained. [Accessed: 22-Feb-2021].

[16] “Radio Spectrum Allocation,” *Federal Communications Commission*, 03-Feb-2021. [Online]. Available: https://www.fcc.gov/engineering-technology/policy-and-rules-division/general/radio-spectrum-allocation. [Accessed: 16-Mar-2021].

[17] "United States Frequency Allocation Chart," *United States Frequency Allocation Chart | National Telecommunications and Information Administration*. [Online]. Available: https://www.ntia.doc.gov/page/2011/united-states-frequency-allocation-chart. [Accessed: 29-Mar-2021].

[18] "U.S. Spectrum Allocations 300 - 3000 MHz," *UNITED STATES SPECTRUM ALLOCATIONS*, Nov-2002. [Online]. Available: https://transition.fcc.gov/Bureaus/OPP/working_papers/oppwp38chart.pdf. [Accessed: 15-Mar-2021].

[19] Types of frequencies and wavelengths in the radio frequency spectrum. (2019, June 10). Retrieved February 15, 2021, from https://www.signalbooster.com/blogs/news/types-of-frequencies-and-wavelengths-in-the-radio-frequency-spectrum

[20] W. Amplifiers, "Cell Phone Frequency Bands by Provider," *WilsonAmplifiers.com*, 16-Mar-2021. [Online]. Available: https://www.wilsonamplifiers.com/blog/frequencies-by-provider/. [Accessed: 16-Mar-2021].

[21] Accton, "The Emergence of 5G mmWave," Accton Technology, 15-Jan-2021. [Online]. Available: https://www.accton.com/Technology-Brief/the-emergence-of-5g-mmwave/ [Accessed: 15-Feb-2021].

[22] M. Simkó and M.-O. Mattsson, "5G Wireless Communication and Health Effects-A Pragmatic Review Based on Available Studies Regarding 6 to 100 GHz," *International journal of environmental research and public health*, 13-Sep-2019. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6765906/. [Accessed: 16-Mar-2021].

[23] *5G Network Architecture. Core, RAN, & Security Architecture For 5G*, 19-Mar-2021. [Online]. Available: https://www.viavisolutions.com/en-us/5g-architecture#:~:text=5G%20Architecture,-The%20primary%20goal&text=5G%20utilizes%20a%20more%20intelligent%20architecture%2C%20with%20Radio%20Access%20Networks,creating%20additional%20data%20access%20points. [Accessed: 29-Mar-2021].

[24] M. Shariat, Ö. Bulakci, A. De Domenico, C. Mannweiler, M. Gramaglia, Q. Wei, A. Gopalasingham, E. Pateromichelakis, F. Moggio, D. Tsolkas, B. Gajic, M. R. Crippa, and S. Khatibi, "A Flexible Network Architecture for 5G Systems," *Wireless Communications and Mobile Computing*, 11-Feb-2019. [Online]. Available: https://www.hindawi.com/journals/wcmc/2019/5264012/. [Accessed: 29-Mar-2021].

[25] J. Ordonez-Lucena and P. Ameigeiras, "Network Slicing for 5G with SDN/NFV: Concepts, Architectures, and Challenges," *IEEE Xplore*. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7926921. [Accessed: 04-Apr-2021].

[26] M. N. O. Sadiku and C. M. Akujuobi, "Software-defined radio: a brief overview," *IEEE Potentials*, vol. 23, no. 4, pp. 14–15, Oct. 2004.

[27] L. F. Chaparro, "Chapter 0 - From the Ground Up!," in *Signals and Systems Using MATLAB (Third Edition)*, 3rd ed., A. Akan, Ed. Academic Press, 2019, pp. 3–56.

[28] Engineering Journal, 2014. Wireless Innovation, "What is Software Defined Radio?," Wireless Innovation Forum, 2015.

[29] GNU Radio Project, "About GNU Radio," *GNU Radio*. [Online]. Available: https://www.gnuradio.org/about/. [Accessed: 12-Mar-2021].

[30] "Guided Tutorial GRC," Guided Tutorial GRC - GNU Radio. [Online]. Available: https://wiki.gnuradio.org/index.php/Guided_Tutorial_GRC. [Accessed: 30-Mar-2021].

[31] "QT GUI Waterfall Sink," *QT GUI Waterfall Sink - GNU Radio*. [Online]. Available: https://wiki.gnuradio.org/index.php/QT_GUI_Waterfall_Sink. [Accessed: 16-Mar-2021].

[32] "QT GUI Frequency Sink," *QT GUI Frequency Sink - GNU Radio*. [Online]. Available: https://wiki.gnuradio.org/index.php/QT_GUI_Frequency_Sink. [Accessed: 16-Mar-2021].

[33] Primer, "Fundamentals of Real-Time Spectrum Analysis," Tekatronix, 2015

[34] “Welcome to python.org.” [Online]. Available: https://www.python.org/. [Accessed: 04-Apr-2021].

[35] “Visualization with Python” *Matplotlib*. [Online]. Available: https://matplotlib.org/. [Accessed: 04-Apr-2021].

[36] M. Bostock, “Data-Driven Documents,” *D3.js*. [Online]. Available: https://d3js.org/. [Accessed: 04-Apr-2021].

[37] K. R. Rao, “Properties of the DFT,” in *Fast fourier transform - algorithms and applications*, Lieu de publication non identifié: Springer, 2013.

[38] M. Kerrisk, “Chapter 37: Daemons,” in The Linux Programming Interface, No Starch Press, 2010.

[39] B. Finney, “PEP 3143 -- Standard daemon process library,” Python.org, 26-Jan-2009. [Online]. Available: https://www.python.org/dev/peps/pep-3143/. [Accessed: 11-Mar-2021].

[40] M. C. R. Chintalapati and R. D. Mellencamp, “Method, Apparatus & Computer Program Product for Dynamic Administration, Management, and Monitoring of Daemon Processes,” 24-Sep-2002.

[41] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A survey,” Computer Networks, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[42] C.-C. Chang, H.-L. Wu, and C.-Y. Sun, “Notes on ‘Secure authentication scheme for IoT and cloud servers,’” Pervasive and Mobile Computing, vol. 38, pp. 275–278, Jul. 2017.

[43] Cloudflare, “What is the Cloud?: Cloud Definition,” *Cloudflare*. [Online]. Available: http://www.cloudflare.com/learning/cloud/what-is-the-cloud. [Accessed: 01-Apr-2021].

[44] IBM Cloud Education, “Cloud Servers,” IBM, 19-Apr-2019. [Online]. Available: https://www.ibm.com/cloud/learn/cloud-server. [Accessed: 02-Mar-2021].

[45] "What is IaaS?," Infrastructure as a Service | Microsoft Azure. [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-iaas/. [Accessed: 2-Mar-2021].

[46] "What is PaaS?," Platform as a Service | Microsoft Azure. [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-paas/. [Accessed: 2-Mar-2021].

[47] "What is SaaS?," Software as a Service | Microsoft Azure. [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-saas/. [Accessed: 2-Mar-2021].

[48] GNU Radio Project, "GNU Radio Wiki," *GNU Radio*. [Online]. Available: https://wiki.gnuradio.org/index.php/Main_Page. [Accessed: 12-Mar-2021].

[49] Amazon Web Services, "Cloud Products," Amazon. [Online]. Available: https://aws.amazon.com/products/. [Accessed: 16-Mar-2021].

[50] Microsoft Azure, "Directory of Azure Cloud Services: Microsoft Azure," *Directory of Azure Cloud Services | Microsoft Azure*. [Online]. Available: https://azure.microsoft.com/en-us/services/. [Accessed: 16-Mar-2021].

[51] Google Cloud, "Products and Services | Google Cloud," *Google Cloud*. [Online]. Available: https://cloud.google.com/products. [Accessed: 16-Mar-2021].

[52] Node.js, "About Node.js," *Node.js*. [Online]. Available: https://nodejs.org/en/about/. [Accessed: 04-Apr-2021].

[53] *Apache Overview HOWTO: Apache*. [Online]. Available: https://tldp.org/HOWTO/Apache-Overview-HOWTO-2.html#:~:text=Apache%201.3%20has%20a%20modular,party%20modules%2C%20providing%20extended%20functionality. [Accessed: 04-Apr-2021].

[54] Canonical, "Install Ubuntu Server," *Ubuntu*. [Online]. Available: https://ubuntu.com/tutorials/install-ubuntu-server. [Accessed: 12-Mar-2021].

Appendix A: Project Authorship

The following chart outlines each section of the report, the sections' primary and (if applicable) secondary author, and the sections' primary and secondary editors. In this case an "author" indicates the individual(s) who wrote major parts of each section. "Editor" indicates the individual(s) that primarily edited each section.

Section	Author 1	Author 2	Editor 1	Editor 2
Abstract	Joseph	-	Ian	-
E.S.	Aneela	Sreeshti	Joseph	-
1	Sreeshti	-	Aneela	Joseph
1.1	Ian	-	Aneela	Kevin
1.2	Kevin	-	Joseph	Sreeshti
1.3	Aneela	-	Joseph	Ian
1.4	Joseph	Sreeshti	Ian	Aneela
1.5	Joseph	-	Kevin	Sreeshti
2.1	Kevin	-	Ian	Joseph
2.2	Kevin	-	Joseph	Aneela
2.3	Sreeshti	-	Joseph	Aneela
2.4	Aneela	Ian	Sreeshti	Kevin
2.5	Joseph	-	Aneela	Ian
2.6	Joseph	-	Ian	Aneela
2.7	Aneela	-	Kevin	Sreeshti
3.1	Joseph	-	Ian	Aneela
3.2	Joseph	-	Ian	-
3.3	Ian	-	Joseph	Sreeshti
3.4	Aneela	-	Ian	Joseph
3.4.1	Sreeshti	-	Aneela	Joseph
3.4.2	Joseph	-	Ian	Aneela
3.4.3	Kevin	-	Joseph	-
3.5	Aneela	-	Sreeshti	Kevin

3.6	Joseph	Kevin	Sreeshti	Aneela
3.7	Aneela	-	Kevin	Ian
4	Sreeshti	-	Joseph	-
5	Joseph	-	Kevin	-
6	Kevin	Joseph	Sreeshti	-
7	Ian	-	Aneela	-
8	Aneela	-	Ian	-
9	Ian	-	Sreeshti	-
10	Kevin	-	Aneela	-

In addition to the breakdown of contributions to the report, there were five major pieces of the implementation with one member of the team taking primary responsibility for each. The GNU Radio flowchart and data sampling portion of the project was overseen by Sreeshti. Joseph was responsible for the uploader script, as well as the daemon program responsible for running the entire radio host. Kevin was the primary developer for the cloud server, with Joseph assisting with the multi-threading and optimization portions of the web server. Ian created the front-end display that the user sees, and Aneela was responsible for the portion of the web page allowing user input as well as the functionality in GNU Radio to change the radio's center frequency based on data from the server.

Appendix B: Tutorial for Setting up Radio Host

The first part of setup is installing Ubuntu 18.04 Server:

- An in-depth tutorial for installing Ubuntu Server can be found in [54]
- All of the installation settings are the same as listed with the exception of personal information such as user login

After the operating system is installed the next step is installing required software:

- Python 3.6 comes installed with Ubuntu and can be verified using the following command:
  $ python3 --version

- GNU Radio 3.8 can be installed through the package manager using the following terminal commands:
  - $ sudo add-apt-repository ppa:gnuradio/gnuradio-releases-3.8
    - This command adds the package repository to Ubuntu's package manager
  - $ sudo apt-get update
    - This command updates list of packages the manager can install
  - $ sudo apt install gnuradio
    - This command installs the GNU Radio package
- GNU Radio can be verified with the following command: $ gnuradio-config-info -v

The code for running the Radio Host can be found and cloned from the following git repository

$ git clone https://github.com/MQPSpectrumObservatory/SpectrumObservatory-Radios.git

- This repository contains four Python scripts, the following modifications are required:
  - On line 96 of "top_block.py" and line 18 of "transfer.py", the URL must be changed to be directed towards the address of the server being used.

At this point it should be possible to plug in the USRP 2901 via USB and run the Radio Host using:

$ python3 RadioHost.py 1 start

Appendix C: Tutorial for Setting up Web Server

The steps for installing Ubuntu Server for the Web Server are the same as listed in Appendix B

- The tutorial for this can be found in [54]

After the operating system is installed the next step is installing required software:

- Node.js can be installed using the following two commands:
  - $ curl -fsSL https://deb.nodesource.com/setup_12.x | sudo -E bash -
  - $ sudo apt-get install -y nodejs
    - These commands add Node v12 to the package repository and install it
- The commands $ node -v and $ npm -v can be used to verify both Node.js and its package manager are installed

The code for running the server can be found and cloned from the following git repository

$ git clone https://github.com/MQPSpectrumObservatory/SpectrumObservatory-Webserver.git

- This code should not require any modification to run

Once the code base is cloned, navigate into the directory and run the command $ sudo npm install

- This command will install any third-party libraries and modules needed to run the server

After the libraries are installed, it should be possible to run the server using the command $ sudo npm start

- Note: The server must be started with elevated privileges as it must bind itself to port 80

Appendix D: *RadioHost.py* Source Code

```
#!/usr/bin/env python3
## Imports
import argparse         # Parse commandline args (to set number of radios)
import logging          # Logging library
import multiprocessing  # Separate child process for each Radio Instance
import signal           # Kernel signal handling
import sys              # Exit program
import threading        # Seperate thread for GNU Radio and EncodeTransfer
import time             # Program sleep

from top_block import (
    top_block
) # GNU Radio top block class

from transfer import (
    Transfer
) # Transfer class

from daemon import (
    Daemon
) # Daemon class

## Daemon subclass
class RadioDaemon(Daemon):

    # Override init to have args attribute
    def __init__(self, pidfile, args):
        Daemon.__init__(self, pidfile) # Run parent constructor

        self.args = args

    # Override run with main process code
    def run(self):

        # Set up signal handler
        signal.signal(signal.SIGTERM, signalHandler)
```

```
        # Set up child processes
        p1 = RadioChild(1)
        if self.args.n == 2:
            p2 = RadioChild(2)

        # Start the child processes
        logging.info("Start Radio Process 1")
        p1.start()

        if self.args.n == 2:
            logging.info("Start Radio Process 2")
            p2.start()

        try:
            while True:
                time.sleep(0.5)

        except ServiceExit:
            # Terminate the running threads.
            # Signal the GNU Radio thread to stop
            p1.shutdown_flag.set()
            if self.args.n == 2:
                p2.shutdown_flag.set()

            # Wait for the processes to close...
            p1.join()
            if self.args.n == 2:
                p2.join()

            logging.info('Exiting main program')

## Radio child-process sublass
class RadioChild(multiprocessing.Process):

    # Override init to have radioNum attribute
    def __init__(self, radioNum):
        multiprocessing.Process.__init__(self) # Run parent constructor
```

```python
        self.radioNum = radioNum
        self.shutdown_flag = multiprocessing.Event()

    def run(self):
        # Startup code
        g = GNURadioJob(self.radioNum)
        t = EncodeTransferJob(self.radioNum)

        logging.info(f"[Process: {self.radioNum}] Starting GNU Radio Thread")
        g.start()
        time.sleep(5)
        logging.info(f"[Process: {self.radioNum}] Starting Upload Thread")
        t.start()

        # Spin until shutdown is signaled
        while not self.shutdown_flag.is_set():
            time.sleep(0.5)

        # Shutdown code
        g.shutdown_flag.set()
        t.shutdown_flag.set()

## GNU Radio thread subclass
class GNURadioJob(threading.Thread):

    # Override init to have radioNum attribute
    def __init__(self, radioNum):
        threading.Thread.__init__(self) # Run parent constructor

        self.radioNum = radioNum
        self.shutdown_flag = threading.Event()

    def run(self):
        # Startup Code
        tb = top_block(self.radioNum)
        tb.start()
        tb.startThread()

        # Spin until shutdown is signaled
```

```
        while not self.shutdown_flag.is_set():
            time.sleep(0.5)

        # Shutdown Code
        tb.stop()
        tb.wait()
        logging.info(f"[Process: {self.radioNum}] GNU Radio Shutdown")

## Encode and Transfer Script thread subclass
class EncodeTransferJob(threading.Thread):

    # Override init to have radioNum attribute
    def __init__(self, radioNum):
        threading.Thread.__init__(self) # Run parent constructor

        self.radioNum = radioNum
        self.shutdown_flag = threading.Event()

    def run(self):
        # Startup Code
        tx = Transfer(self.radioNum)
        tx.run()

        # Spin until shutdown is signaled
        while not self.shutdown_flag.is_set():
            time.sleep(0.5)

        # Shutdown Code
            # Transfer class handles its own shutdown

## Custom exception for termination
class ServiceExit(Exception):
    pass

def signalHandler(signum, frame):
    raise ServiceExit

## Setup daemon and start
def main():
```

```
    # Argument parsing (sets number of radios in use and daemon commands)
    parser = argparse.ArgumentParser(description="Daemon controlling radios and
data upload")
    parser.add_argument("n", type=int, help="Number of radios in use")
    parser.add_argument("d", type=str, help="Command for daemon (start, stop,
restart)")

    args = parser.parse_args()

    # Check if args are invalid
    if args.n < 1 or args.n > 2:
        parser.error("Number of radios must be 1 or 2")
    if args.d != "start" and args.d != "stop" and args.d != "restart":
        parser.error("Daemon command must be start, stop, or restart")

    # Create instance of daemon
    daemon = RadioDaemon('./RadioDaemon.pid', args)

    # Daemon command handling
    if "start" == args.d:


        # Setup logger
        logging.basicConfig(filename="RadioDaemon.log",
                        format='%(asctime)s [%(levelname)s] %(message)s',
                        filemode='w')

        logger = logging.getLogger()
        logger.setLevel(logging.INFO)

        daemon.start()

    elif "stop" == args.d:
        daemon.stop()

    elif "restart" == args.d:
        daemon.restart()
```

```
    sys.exit(0)

if __name__ == "__main__":
    main()
```

Appendix E: *top_block.py* Source Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

#
# SPDX-License-Identifier: GPL-3.0
#
# GNU Radio Python Flow Graph
# Title: Spectrum Analyzer
# GNU Radio version: 3.8.1.0

from gnuradio import blocks
from gnuradio import fft
from gnuradio.fft import window
from gnuradio import gr
from gnuradio.filter import firdes
import sys
import signal
from argparse import ArgumentParser
from gnuradio.eng_arg import eng_float, intx
from gnuradio import eng_notation
from gnuradio import gr, blocks
from gnuradio import uhd
import threading
import time
import requests
import pmt

class top_block(gr.top_block):

    def __init__(self, radioNum):
        gr.top_block.__init__(self, "Spectrum Analyzer")


        ##################################################
        # Variables
        ##################################################
```

```
        self.samp_rate = samp_rate = int(300e3)
        self.center_freq = center_freq = 900e6
        self.radio_num = radioNum

        # Create custom PMT metadata containing the assigned radio number
        key0 = pmt.intern("radio_num")
        val0 = pmt.from_long(self.radio_num)
        extra_meta = pmt.make_dict()
        extra_meta = pmt.dict_add(extra_meta, key0, val0)

        ##################################################
        # Blocks
        ##################################################
        self.uhd_usrp_source_0 = uhd.usrp_source(
            ",".join(("", "")),
            uhd.stream_args(
                cpu_format="fc32",
                args='peak=0.003906',
                channels=list(range(0,1)),
            ),
        )
        self.uhd_usrp_source_0.set_center_freq(center_freq, 0)
        self.uhd_usrp_source_0.set_gain(28, 0)
        self.uhd_usrp_source_0.set_antenna('RX2', 0)
        self.uhd_usrp_source_0.set_bandwidth(samp_rate, 0)
        self.uhd_usrp_source_0.set_samp_rate(samp_rate)
        # No synchronization enforced.
        self.fft_vxx_0 = fft.fft_vcc(1024, True, window.blackmanharris(1024),
True, 1)
        self.blocks_vector_to_stream_0 =
blocks.vector_to_stream(gr.sizeof_gr_complex*1, 1024)
        self.blocks_stream_to_vector_0 =
blocks.stream_to_vector(gr.sizeof_gr_complex*1, 1024)
        self.blocks_file_meta_sink_0 =
blocks.file_meta_sink(gr.sizeof_gr_complex*1, f'./sample{self.radio_num}.dat',
samp_rate, 1, blocks.GR_FILE_FLOAT, True, 300000, extra_meta, False)
        self.blocks_file_meta_sink_0.set_unbuffered(False)
```

```
        ###################################################
        # Connections
        ###################################################
        self.connect((self.blocks_stream_to_vector_0, 0), (self.fft_vxx_0, 0))
        self.connect((self.blocks_vector_to_stream_0, 0),
(self.blocks_file_meta_sink_0, 0))
        self.connect((self.fft_vxx_0, 0), (self.blocks_vector_to_stream_0, 0))
        self.connect((self.uhd_usrp_source_0, 0),
(self.blocks_stream_to_vector_0, 0))

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.uhd_usrp_source_0.set_samp_rate(self.samp_rate)
        self.uhd_usrp_source_0.set_bandwidth(self.samp_rate, 0)

    def get_center_freq(self):
        return self.center_freq

    def set_center_freq(self, freq):
        self.center_freq = freq
        self.uhd_usrp_source_0.set_center_freq(self.center_freq, 0)

    def MQP_HTTP_Parser(self):
        while True:
            freq1 =
int((requests.get(f'http://spectrumobservatory.wpi.edu/freq{self.radio_num}')).co
ntent)
            if(freq1 != self.get_center_freq): # Only set frequency if it is
different (save on overhead)
                self.set_center_freq(freq1)
            time.sleep(5)

    def startThread(self):
        self.t1 = threading.Thread(target = self.MQP_HTTP_Parser)
        self.t1.daemon = True
        self.t1.start()
```

```python
## NOTE: GNURadio generated main function not used
def main(top_block_cls=top_block, options=None):
    tb = top_block_cls()

    def sig_handler(sig=None, frame=None):
        tb.stop()
        tb.wait()
        print("Exiting")
        sys.exit(0)

    signal.signal(signal.SIGINT, sig_handler)
    signal.signal(signal.SIGTERM, sig_handler)

    tb.start()
    tb.startThread()

    try:
        input('Press Enter to quit: ')
    except EOFError:
        pass

    tb.stop()
    tb.wait()

if __name__ == '__main__':
    main()
```

Appendix F: *daemon.py* Source Code

```
#!/usr/bin/env python3

# Implementation of a daemon class
# Taken from:
http://web.archive.org/web/20131017130434/http://www.jejik.com/articles/2007/02/a
_simple_unix_linux_daemon_in_python/

# This class handles creating a process and daemonizing itself
# To use it you simply subclass it and implement the run method

import atexit, logging, os, sys, time
from signal import SIGTERM

class Daemon:
    def __init__(self, pidfile):
        self.pidfile = pidfile

    def daemonize(self):
        # UNIX double fork mechanism
        try:
            pid = os.fork()
            if pid > 0:
                # Exit first parent
                sys.exit(0)
        except OSError as err:
            logging.critical('Fork #1 failed: {0}\n'.format(err))
            sys.exit(1)

        # Decouple from parent environment
        # NOTE: we do not change the working directory (current working directory
             is not at risk of being unmounted)
        os.setsid()
        os.umask(0)

        # Do second fork
        try:
```

```
            pid = os.fork()
            if pid > 0:
                # Exit second parent
                sys.exit(0)
        except OSError as err:
            logging.critical('Fork #2 failed: {0}\n'.format(err))
            sys.exit(1)

        # Redirect standard file descriptors into /dev/null
        sys.stdout.flush()
        sys.stderr.flush()
        si = open(os.devnull, 'r')
        so = open(os.devnull, 'a')
        se = open(os.devnull, 'a')

        os.dup2(si.fileno(), sys.stdin.fileno())
        os.dup2(so.fileno(), sys.stdout.fileno())
        os.dup2(se.fileno(), sys.stderr.fileno())

        # Write pidfile
        atexit.register(self.delpid)
        pid = str(os.getpid())
        with open(self.pidfile, 'w+') as f:
            f.write(pid + '\n')

    def delpid(self):
        os.remove(self.pidfile)

    def start(self):
        # Check for a pidfile to see if daemon is already running
        try:
            with open(self.pidfile, 'r') as pf:
                pid = int(pf.read().strip())
        except IOError:
            pid = None

        if pid:
            message = "pidfile {0} already exists."
            logging.error(message.format(self.pidfile))
```

```
            sys.exit(1)

        # Start the daemon
        self.daemonize()
        self.run()

    def stop(self):
        # Get the pid from the pidfile
        try:
            with open(self.pidfile, 'r') as pf:
                pid = int(pf.read().strip())
        except IOError:
            pid = None

        if not pid:
            message = "pidfile {0} does not exist."
            logging.warn(message.format(self.pidfile))
            return # this is not a fatal error if the daemon is restarting (do
                   not exit program)

        # Try to kill the daemon process
        try:
            while 1:
                os.kill(pid, SIGTERM)
                time.sleep(0.1)
        except OSError as err:
            e = str(err.args)
            if e.find("No such process") > 0:
                if os.path.exists(self.pidfile):
                    os.remove(self.pidfile)
            else:
                logging.error(e)
                sys.exit(1)

    def restart(self):
        # Restart the daemon
        self.stop()
        self.start()
```

```
def run(self):
    # Should be overwritten when Daemon is subclassed
    # Is called when the daemon finishes start() or restart()
    pass
```

Appendix G: *transer.py* Source Code

```
#!/usr/bin/env python3

## Imports
import base64   # Encoding binary data
import json     # Creating JSON file
import logging  # Logging library
import os       # File path manipluation
import pmt      # GNU Radio header parsing
import requests # Creating HTTP requests
import sys      # Reading program arguments
import time     # Program sleep

from gnuradio.blocks import parse_file_metadata # GNU Radio header parsing

class Transfer:
    def __init__(self, radioNum):
        self.radioNum = radioNum
        self.HOST = "http://spectrumobservatory.wpi.edu/data"
# host of webserver
        self.BINNAME = "sample.dat"
# name of the input file (gets overriden on init)
        self.HEADERS = {'Content-type': 'application/json', 'Accept':
                        'text/plain'} # Headers for POST request
        self.NITEMS = 300000

        self.inFile = None


    ## Called to update the data file name based on assigned radio number
    def setBINNAME(self):
        self.BINNAME = f'sample{self.radioNum}.dat'


    ## Called when the transfer program is terminated
    # Cleans up open files, removes temporary files and exits (terminating the
      thread)
```

```python
def stop(self):
    logging.info(f"[Process: {self.radioNum}] Program termination\n")
    self.inFile.close()
    os.remove(self.BINNAME)
    sys.exit(0)


## This will be called to parse header data out of the dat file
# Takes in an open file descriptor and returns a python dictionary
# NOTE: This function is built on the gr_read_file_metadata program from GNU
  Radio
def parseHeaders(self):

    # read out header bytes into a string
    header_str = self.inFile.read(parse_file_metadata.HEADER_LENGTH)

    # Convert from created string to PMT dict
    try:
        header = pmt.deserialize_str(header_str)
    except RuntimeError:
        logging.info(f"[Process: {self.radioNum}] Could not deserialize
                     header\n")
        self.stop()

    # Convert from PMT dict to Python dict
    info = parse_file_metadata.parse_header(header)

    if(info["extra_len"] > 0):
        extra_str = self.inFile.read(info["extra_len"])

    # Extra header info
    try:
        extra = pmt.deserialize_str(extra_str)
    except RuntimeError:
        logging.info(f"[Process: {self.radioNum}] Could not deserialize extra
                     headers\n")
        self.stop()

    info = parse_file_metadata.parse_extra_dict(extra, info)
```

```
        return info

    ## Main running function
    def run(self):

        ## Set the data file name (changes based on radio instance)
        self.setBINNAME()

        ## Open the binary data file
        self.inFile = open(self.BINNAME, "rb")

        headerNum = 0 # Number of headers read

        ## Main loop:
        # 1. Read GNU Radio output file
        # 2. Parse header info
        # 3. Check if final segment
        # 4. Encode payload
        # 5. Create JSON
        # 6. Send JSON with HTTP POST
        while(True):

            # Read in bin file to parse header metadata
            headerData = self.parseHeaders()
            headerNum += 1
            logging.info(f"[Process: {self.radioNum}] Header Number:
                         {headerNum}")

            # Size of each data segment
            ITEM_SIZE = headerData["nitems"]
            SEG_SIZE  = headerData["nbytes"]

            # Check if sample is too small
            # GET request interrupts GNU Radio's loop causing it to prematurely
             reinsert a header
            if ITEM_SIZE < self.NITEMS:
                self.inFile.read(SEG_SIZE)
```

```
        logging.info(f"[Process: {self.radioNum}] Segment too small,
          skipping\n")
        continue

    # Pull out relevant header info
    rx_time      = headerData["rx_time"]
    rx_rate      = headerData["rx_rate"]
    rx_freq      = pmt.to_python(headerData["rx_freq"])
    radio        = pmt.to_python(headerData["radio_num"])

    # Encode data payload from bin file into base64 ascii characters
    inputBinary = self.inFile.read(SEG_SIZE)
    encodedData = (base64.b64encode(inputBinary)).decode('ascii')

    # Create JSON file using encoded payload and header metadata
    jsonFormat = {"metadata":{"rx_time" : rx_time, "rx_sample" : rx_rate,
     "rx_freq" : rx_freq, "radio_num" : radio}, "payload" : encodedData}
    jsonFile = json.dumps(jsonFormat, indent=4)

    # Send this JSON file to the WebServer with an HTTP POST
    r = requests.post(url=self.HOST, data=jsonFile, headers=self.HEADERS)
    logging.info(f"[Process: {self.radioNum}] Response from server: %s\n"
                 %r)

    # Wait to send next segment (segments are generated at a rate of 1
     per second)
    time.sleep(1)
```

Appendix H: *server.js* Source Code

```
/* ---- Global Variables ---- */
// Default packages
const fs = require('fs');
const http = require('http');
const path = require('path');
const url = require('url');
const {Worker} = require('worker_threads');

// Third party npm packages
const finalhandler = require('finalhandler');
const serveIndex = require('serve-index');
const serveStatic = require('serve-static');

const port = 80;
const mime = {
    html: 'text/html',
    txt:  'text/plain',
    css:  'text/css',
    gif:  'image/gif',
    jpg:  'image/jpeg',
    png:  'image/png',
    svg:  'image/svg+xml',
    js:   'application/javascript'
};
// Point to the static directory to serve
const index = serveIndex('public');
const serve = serveStatic('public');
// Hold radio frequency values for remote control
let freqVal1 = '900000000';
let freqVal2 = '900000000';
let freqVal3 = '900000000';
let freqVal4 = '900000000';


/* ---- HTTP Server Processing & Event Loop ---- */
const server = http.createServer(function (req, res) {
```

```
const pathname = url.parse(req.url).pathname;
switch (req.method) {
    case "GET":
        console.log("GET: %s", pathname);

        // Filter and format path name and serve any static file matching
        let dir = path.join(__dirname, 'public');
        let req_path = req.url.toString().split('?')[0];
        let filteredPath = req_path.replace(/\/$/, '/index.html');
        let file = path.join(dir, filteredPath);

        if (file.indexOf(dir + path.sep) !== 0) {
            sendCode(res, 403, "403 forbidden");
            break;
        }

        let type = mime[path.extname(file).slice(1)] || 'text/plain';
        let s = fs.createReadStream(file);

        s.on('open', function () {
            res.setHeader('Content-Type', type);
            s.pipe(res);
        });

        // if not serving static file/directory
        s.on('error', function () {
            switch (pathname) {
                case '/freq1':
                    console.log("Sending frequency request on /freq1");
                    res.end(freqVal1);
                    sendCode(res, 200, "OK");
                    break;

                case '/freq2':
                    console.log("Sending frequency request on /freq2");
                    res.end(freqVal2);
                    sendCode(res, 200, "OK");
                    break;
```

```
case '/freq3':
    console.log("Sending frequency request on /freq3");
    res.end(freqVal3);
    sendCode(res, 200, "OK");
    break;

case '/freq4':
    console.log("Sending frequency request on /freq4");
    res.end(freqVal4);
    sendCode(res, 200, "OK");
    break;

// Called when front-end requests contents of public/data
case '/data1/':
    const done = finalhandler(req, res);    // handler to
     write response
    serve(req, res, function onNext(err) {  // serve the
     indexes of the files in directory
        if (err) return done(err);
        index(req, res, done);
    })
    break;

case '/data2/':
    done = finalhandler(req, res);          // handler to
     write response
    serve(req, res, function onNext(err) {  // serve the
     indexes of the files in directory
        if (err) return done(err);
        index(req, res, done);
    })
    break;

default:
    console.log("Client made GET to %s and invoked 404",
     url.parse(req.url).pathname);
    sendCode(res, 404, "404 not found");
    break;
```

```
            }
        });
        break;

    case "POST":
        console.log("POST: %s", pathname);

        // Data upload handling (full pathname is used in filename)
        if (pathname.startsWith('/data')) {
            let reqBody = '';
            req.on('data', function (data) {
                reqBody += data;
                if (reqBody.length > 1e7 /*10MB*/ ) {
                    sendCode(res, 413, "Request too large");
                }
            });
            req.on('end', function () {
                sendCode(res, 200, "OK");
                console.log("Received %d bytes", req.socket.bytesRead);
                // This function returns a promise that resolves when its
                     worker thread finishes
                processData(pathname, reqBody).then(console.log("Worker
                            finished"));
            });

        } else {

        let freq;
        switch (pathname) {
            case '/post1':
                freq = '';
                req.on('data', function (rcdata) {
                    freq += rcdata;
                });
                req.on('end', () => {
                    // Need to remove "CenterFreq="
                    freqVal1 = freq.toString().split("=")[1];
                    console.log("Logged a frequency of %s on /post1",
                                 freqVal1);
```

```
            // Respond with "204 No Content" to avoid page
                   redirecting
            res.writeHead(204)
            res.end()
        });
        break;

    case '/post2':
        freq = '';
        req.on('data', function (rcdata) {
            freq += rcdata;
        });
        req.on('end', () => {
            freqVal2 = freq.toString().split("=")[1];
            console.log("Logged a frequency of %s on /post2",
                         freqVal2);
            res.writeHead(204)
            res.end()
        });
        break;

    case '/post3':
        freq = '';
        req.on('data', function (rcdata) {
            freq += rcdata;
        });
        req.on('end', () => {
            freqVal3 = freq.toString().split("=")[1];
            console.log("Logged a frequency of %s on /post3",
                         freqVal3);
            res.writeHead(204)
            res.end()
        });
        break;

    case '/post4':
        freq = '';
        req.on('data', function (rcdata) {
            freq += rcdata;
```

```
                    });
                    req.on('end', () => {
                        freqVal = freq.toString().split("=")[1];
                        console.log("Logged a frequency of %s on /post4",
                                    freqVal);
                        res.writeHead(204)
                        res.end()
                    });
                    break;

                default:
                    console.log("Client made POST to %s and invoked 404",
                                url.parse(req.url).pathname);
                    sendCode(res, 404, "Not found");
                    break;
            }}
            break;

        default:
            sendCode(res, 405, "Incorrect Method");
            break;
    }
}).listen(port);
console.log("Server started on port %d\n", port);

/* ---- Helper Functions ---- */
// Spawn a worker thread that runs the file worker.js
function processData(pathname, reqBody) {
    return new Promise((resolve, reject) => {
        const worker = new Worker('./worker.js', { workerData: { pathname,
reqBody } });
        worker.on('message', resolve);
        worker.on('error',   reject);
        worker.on('exit', (code) => {
            if (code !== 0)
                reject(new Error(`Worker stopped with exit code ${code}`));
        });
    });
};
```

```
function sendCode(res, code, msg) {
    fs.readFile('public/status/' + code + '.html', function (error, content) {
        if (error) throw error;
        res.writeHead(code, msg, {'Content-type': 'text/html'});
        res.end(content, 'utf-8');
    })
}
```

Appendix I: *worker.js* Source Code

```
const fs = require('fs');
const {workerData, parentPort} = require('worker_threads');
const {pathname, reqBody} = workerData;

/* ---- Main Worker Loop ---- */

// Parsing JSON and extract fields
const jsonData = JSON.parse(reqBody);

const payloadData = jsonData.payload;
const metadata = jsonData.metadata;

// Formatting metadata for CSV file
const rx_time =  metadata.rx_time;
const rx_sample = metadata.rx_sample;
const rx_freq =  metadata.rx_freq;
const radio_num =  metadata.radio_num;
const metadata_line = rx_time + "," + rx_sample + "\n" + rx_freq + "," +
radio_num;

// Convert data payload to binary string
const binary_string = textToBin(payloadData);

// Split binary string into 1024-sample sized chunks
const bin_array_in_chunks = splitString(binary_string, 65536);

// Convert each binary segment into a csv
for(let i = 0; i < bin_array_in_chunks.length; i++) {
    convertBinToCSV(pathname, bin_array_in_chunks[i], i, metadata_line);
}

// Let parent know worker finished, fulfilling promise
parentPort.postMessage({filename: workerData, status: 'Done'});


/* ---- Helper functions ---- */
```

```
function Arraycreator(byte) {
    const inArray = byte.match(new RegExp('.{1,' + 32 + '}', 'g')); // split
            every 32 characters into a index in array
    const newArr = [];
    while (inArray.length) {
        newArr.push(inArray.splice(0, 2)); // split every two array indecies into
                                                   sub-array
    }
    return newArr;
}

function arrayToCSV(arr, delimiter = ',') {
    return arr.map(
        v => v.map(
            x => (isNaN(x) ? `"${x.replace(/"/g, '""')}"` : x)
            ).join(delimiter)
        ).join('\n');
    }

function zeroPad(num, places = 8) {
    return String(num).padStart(places, '0');
}

// Decodes the data out of Base64 and converts to a binary string
function textToBin(text) {
    let txt = new Buffer.from(text, 'base64').toString('binary');
    let output = [];

    // TODO Optimize this?
    for (let i = 0; i < txt.length; i++) {
        let bin = txt[i].charCodeAt().toString(2);
        output.push(Array(bin.length + 1).join('') + zeroPad(bin, 8));
    }

    return output.join("");
}

function convertBinToCSV(pathname, binary_string, index, metadata_line) {
    const bin_array = Arraycreator(binary_string);
```

```
    let bindata = arrayToCSV(bin_array);

    let finaldata = metadata_line + "\n" + bindata;

    // file format is data###.csv where ### is a 3 digit number representing the
            index of the current file
    let new_file_name = 'data' + index.toString().padStart(3, '0') + '.csv';

    fs.writeFile(`public/data${radio_num}/` + new_file_name, finaldata, function
(err) {
        if (err) return console.log(err);
    });
}

function splitString (string, size) {
    let re = new RegExp('.{1,' + size + '}', 'g');
    return string.match(re);
}
```

Appendix J: *index.html* Source Code

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <!-- Load d3.js -->
    <script src="https://d3js.org/d3.v4.js"></script>
    <script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
    <script type="text/javascript">
    function makeChart() {
      // set the dimensions and margins of the graph
      var margin = {top: 10, right: 30, bottom: 30, left: 100},
        width = 1600 - margin.left - margin.right,
        height = 512 - margin.top - margin.bottom;

      // append the svg object to the body of the page
      var svg = d3.select("#my_dataviz")
        .append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
        .append("g")
        .attr("transform",
          "translate(" + margin.left + "," + margin.top + ")");

      var dir = "data1/";
      var fileextension = ".csv";
      $.ajax({
        //This will retrieve the contents of the folder if the folder is
             configured as 'browsable'
        url: dir,
        success: function (data) {
          //List all matching file names in the page
          $(data).find("a:contains(" + fileextension + ")").each(function () {
            var a = [];
            var I = [];
            var Q = [];
```

```
        var mag = [];
        var freq = [];
        var samp_freq;

        // this makes the path relative to the current page location
        var filename = this.href.replace(window.location.pathname,
               "").replace("http://", "");
        var pathName = dir + filename.substring(filename.lastIndexOf('/') +
               1);
        d3.text(pathName, (function (d) { //Reads in the CSV as a text file
          a = d3.csvParseRows(d); //Parses the text file as a csv, a is an
                                  array of each row
          samp_freq = a[1][0];
          a.forEach((item, index) => { //Each item in 'a' is an arraysize 2
                                  with I and Q
            I.push(parseInt(item[0], 2)); //Stores I of each time sample
            Q.push(parseInt(item[1], 2)); //Stores Q of each time sample
          });
          mag = calcMag(I, Q, mag);
          freq = linspace(samp_freq-150000, samp_freq+150000, 1024);
          graph(freq, mag.slice(1, freq.length), svg);
        }));
      });
    }
  });
}

  // Graphing the data
function graph(xaxis, data, svg) {
        // set the dimensions and margins of the graph
  var margin = {top: 10, right: 30, bottom: 30, left: 100},
    width = 1600 - margin.left - margin.right,
    height = 512 - margin.top - margin.bottom;

  // Add X axis
  var x = d3.scaleLinear()
    .domain([d3.min(xaxis), d3.max(xaxis)])
    .range([0, width]);
  svg.append("g")
```

```
    .attr("transform", "translate(0," + height + ")")
    .call(d3.axisBottom(x));

  // Add Y axis
  var y = d3.scaleLog()
    //.domain([d3.min(data), d3.max(data)])
    .domain([50000000, 8000000000])
    .range([0, height]);
  svg.append("g")
    .call(d3.axisLeft(y)
    .tickFormat(d3.format("0.00001e")));


  svg.selectAll("path")
    .datum(data)
    .attr("fill", "none")
    .attr("stroke", "steelblue")
    .attr("stroke-width", 1.5)
    .attr("d", d3.line()
      .x(function (d, idx) {return x(xaxis[idx])})
      .y(function (d, idx) {return y(data[idx])})
    );

}

function calcMag(I, Q, mag) {
  for (i = 3; i < I.length; i++) {
    mag.push(Math.sqrt((I[i] * I[i]) + (Q[i] * Q[i])));
  }
  return mag;
}

function linspace(startValue, stopValue, cardinality) {
  var arr = [];
  var step = (stopValue - startValue) / (cardinality - 1);
  for (var k = 0; k < cardinality; k++) {
    arr.push(startValue + (step * k));
  }
  return arr;
```

```
    }

    </script>
  </head>
  <body>
    <!-- Create a div where the graph will take place -->
    <div id="my_dataviz" width="1100" height="440"></div>

    <h2 style="text-align:center">Radio 1: Select Frequency</h2>

    <form action="http://spectrumobservatory.wpi.edu/post1" method='POST'
            style="text-align:center">
      <input type="radio" id="CenterFreq" name="CenterFreq" value="600000000">
      <label for="CenterFreq">600 MHz</label>
      <input type="radio" id="CenterFreq" name="CenterFreq" value="850000000">
      <label for="CenterFreq">850 MHz</label><br>
      <input type="submit" value="Submit">
    </form>

    <button onClick="makeChart()">Show Radio 1 Spectrogram</button>

  </body>
</html>
```