Digitized Radio Broadcast Seeker (DRBS)

by

Simbisai Mfunani Sithole

Submitted to the University of Cape Town in partial fulfilment of the requirements

for the degree

Master of Engineering

Supervisor:

Associate Professor Simon Winberg

Department of Electrical Engineering

Faculty of Engineering and the Built Environment

UNIVERSITY OF CAPE TOWN

Grad September 2025

Declaration

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This thesis/dissertation has been submitted to the Turnitin module (or equivalent similarity and originality checking software) and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

Signature of Author: . . .. Signed by candidate . . .

Cape Town

THE DATE: 16 February 2025

Abstract

The use of Software Defined Radio (SDR) has greatly increased the flexibility and programmability of radio systems due to the implementation of radio functions in software. The transfer of traditionally hardware-based signal processing to the software domain enables web SDR receivers to be hosted on the internet as data streaming services. Hosting SDR receivers online allows many users to tune in and listen to broadcast transmissions simultaneously. While significant advancements have been made in SDR hardware technology and making data easily accessible to users, developer productivity has been lacking. The market lacks a simple software development kit that would enable researchers and developers to experiment and create innovative software applications using existing data and components. The aim of this project is to develop an application framework that provides a collection of pre-built modules, components, code libraries, tools and Application Programmable Interfaces (API) that will allow developers to quickly create innovative applications using broadly available Software Defined Radio (SDR) data by leveraging pre-developed infrastructure. The Digitized Radio Broadcast Seeker (DRBS) application as an implementation of the application framework provides a platform for web streaming that aggregates web SDRs and helps developers search for broadcast transmissions of interest as well as build solutions around the streamed transmissions. The research project demonstrated that DRBS could be used to monitor Morse code signals to detect emergencies such as distress calls, weather alerts and other critical broadcasts. After conducting latency and average server response time performance tests, it was concluded that despite the additional infrastructure layer, the DRBS application did not add significant overhead to the signal processing path and could be considered for additional use cases such as identifying FM radio stations and analyzing spectrum usage across different geographical regions.

Acknowledgements

I would like to express my gratitude to my supervisor, A/Prof. Simon Winberg, for the timeous support, detailed and constructive feedback that I needed to complete this thesis. I fully appreciate A/Prof. Winberg's commitment and unquestionable integrity in his approach to work. His guidance was invaluable.

Nomenclature

AM	Amplitude Modulation
API	Application Programming Interface
CCM	CORBA Component Model
CMOS	Complementary Metal Oxide Semiconductor
CORBA	Common Object Request Broker Architecture
CW	Continuous Wave
DMR	Digital Mobile Radio
DRBS	Digitized Radio Broadcast Seeker
DRFS	Direct Radio Frequency Sampling
DSP	Digital Signal Processing
DVB	Digital Video Broadcasting
EHF	Extremely High Frequency
FFT	Fast Fourier Transform
FM	Frequency Modulation
FPGA	Field Programmable Gate Array
GNU	GNU's Not Unix
GPP	General Purpose Processors
HTTP	Hyper Text Transfer Protocol
IF	Intermediate Frequency
LO	Local Oscillator
LSB	Lower Side Band

NFM	Narrowband Frequency Modulation
OS	Operating System
OSGi	Open Service Gateway Initiative
RF	Radio Frequency
SCA	Software Communications Architecture
SDR	Software Defined Radio
SNR	Signal to Noise Ratio
SoC	System on a Chip
SSB	Single Side Band
UHF	Ultra High Frequency
UI	User Interface
USB	Upper Side Band
USRP	Universal Software Radio Peripheral
VHF	Very High Frequency
VM	Virtual Machine
VPC	Virtual Private Cloud
WFM	Wideband Frequency Modulation

Contents

List of Figures

List of Tables

1 Introduction

Digitized Radio Broadcast Seeker (DRBS) is the latest innovation in the web Software Defined Radio (SDR) and streamed data services domain that promises to increase developer productivity and encourage innovation by making it easier for developers to get started. DRBS is an application framework which provides a software development kit that saves developers time taken in developing a solution from the ground up. This lowers the learning curve and literally provides a foundation for developers to create applications quickly and deliver products to market much faster.

The objective of the research project is to create a development framework equipped with pre-built resources (modules, components, code libraries, tools, and APIs) specifically designed to accelerate the creation of innovative applications that process and utilize commonly available Software Defined Radio (SDR) data, leveraging pre-existing infrastructure. The DRBS also acts as a web streaming platform that aggregates web SDRs, helps developers search for broadcast transmissions of interest and build solutions around the streamed transmissions.

1.1 Background

In 1987, Martin Reiser in his insightful work ‘The Oberon System’, made a satirical observation: “The hope is that the progress in hardware will cure all software ills” [1]. At the time, computers were becoming faster, smaller, and more powerful, yet the software running on them was ballooning in complexity and slowing in performance. Reiser’s remark wasn’t a celebration, but a subtle lament, in recognition of the ever-expanding chasm between the pace of software development and the accelerating capabilities of the machines they were meant to command. Nearly four decades later, his words still resonate. While processors now boast billions of transistors, and storage and memory capacities have expanded exponentially, software often struggles to keep up – consuming more resources and introducing more bugs.

Wirth [2], an advocate of lean software and [3], termed this “software bloat” and described this as the tendency of users to want all features including the nice-to-haves thereby increasing software complexity. This feature creep makes software resource intensive negating hardware performance gains.

The SDR industry has seen similar characteristics where Radio Frequency (RF) front ends and Digital Signal Processing (DSP) chains support wider frequency ranges and

higher sampling rates which require more sophisticated software to handle the increased data availability. The trend analysis report [4], Akeela and Dezfouli [5] and Zitouni et al. [6] agree that whilst the rate at which signals are sampled, processed and presented to the application layer has grown significantly due to innovations in hardware, the advancements won't be very useful, if the software and tools don't follow. The adage prevails, hardware enables software and software drives hardware. An example of how hardware improvements have unlocked new capabilities for software to define, control, and adapt radio systems more flexibly is provided in [7], highlighting the evolution of SDR over three decades with more powerful Field Programmable Gate Arrays (FPGA) progressively being designed for intensive DSP.

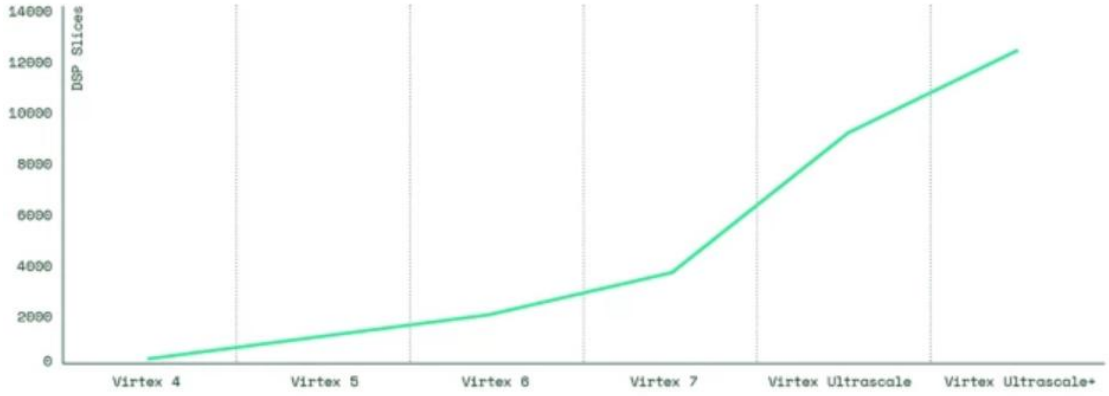

Figure 1.1 The number of DSP slices in each subsequent AMD Virtex FPGA generation continues to grow rapidly [7]**.**

Whilst the strides made by hardware vendors is apparent, the case for software productivity is less encouraging. According to [7] "…SDRs have fundamentally reached a point where the primary limitation on growth is in software, not hardware". Hardware used in SDRs has advanced so much in terms of speed, flexibility, and capability that it's no longer the main bottleneck for progress. Instead, the current constraint lies in complex software design, lack of development tools and skills.

1.2 Objectives

The research focussed on developing the DRBS, an application framework that provides pre-built code libraries, components and packages, Application Programmable Interfaces (API), tools, tutorials and documentation that allow developers to create new software applications using available streamed data services from remotely hosted web SDR receivers.

The objectives of the research project are:

a) To develop and implement DRBS application; an open source, web broadcast streaming tool that provides the ability to cycle across SDR receivers and automatically search for specific data.
b) To develop and document a customizable model and framework specification which can be reused by the research and developer community for SDR based streamed data applications.

1.3 Problem Description

Despite massive advancements in SDR hardware technology that have increased the availability of streamed data services, there is a lack of innovative software applications being developed, that is, developer productivity is lagging. For a developer to create a new application or use case, they must develop from scratch the code base for server-side logic, APIs and UI components, test, debug, deploy and document. All of which take time, and cost money. Developers must painstakingly go through the same process every time they want to develop a new application.

The market is devoid of a robust, open-source software development platform integrated with web SDR hosting applications where researchers and developers can easily build on top of pre-developed code bases, packages and tools to allow creation of new features and functionality quickly using already available infrastructure, resources and data.

There is a need for an application framework that provides a collection of reusable modules to perform common tasks, and pre-built APIs to connect to other platforms hosting web SDR receivers, thus reducing developer effort and allowing rapid building and testing of application specific solutions.

1.4 Research Questions

The research seeks to answer the following questions:

1) What are the common tasks that can be pre-developed providing base functionality from which developers can build-on?
2) Which modules, components, code libraries, packages, tools and APIs are required to perform these tasks?

3) To what extent can DRBS API integrate to an existing web SDR platform such as OpenWebRX and interrogate hosted SDR receivers?

4) Can morse code signal detection and decoding use case can be implemented to validate the system design and test the functionality as well as performance?

1.5 Terms of Reference - requirements and functions

The objective of this research project as outlined in Chapter 1.2 is to develop and document a customizable model and framework specification which can be reused by the research and developer community for SDR based streamed data applications. To validate the framework, DRBS application was developed; an open source, web broadcast streaming tool that provides the ability to cycle across SDR receivers and automatically search for specific data.

The requirements were inspired by platforms such as websdr.org, OpenWebRX, kiwiSDR and shinySDR which host remote SDR receivers. The general theme on the platforms was a landing page listing all the receivers hosted on the platform. The user connects to a receiver of choice by selecting the receiver to start streaming. The web landing page of each receiver typically shows a waterfall display with a simple control panel as shown below.

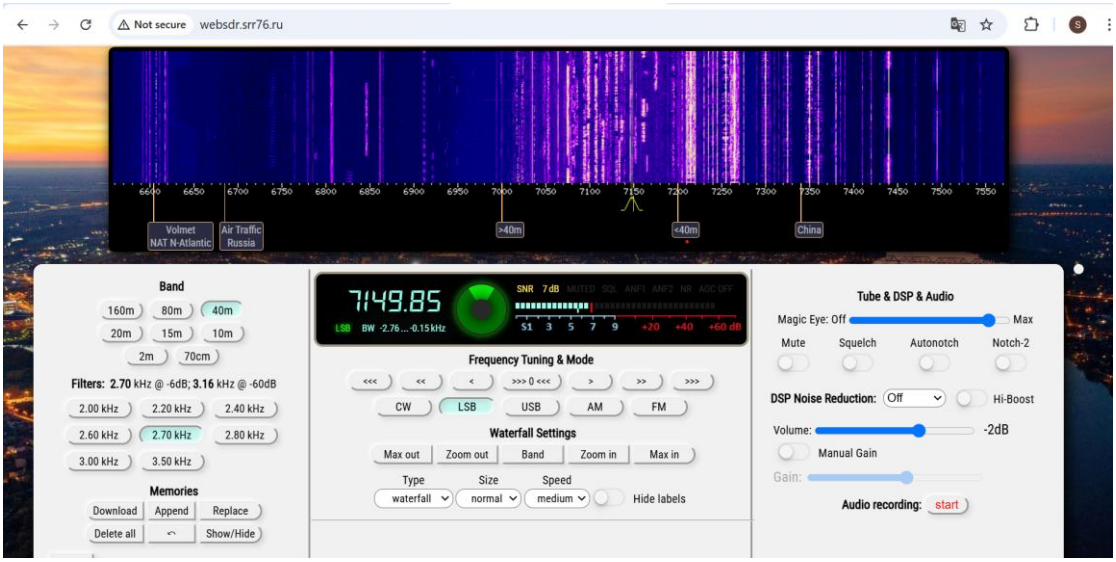

Figure 1.1: Websdr.org hosted SDR receiver web landing page [8].

The main user requirements for the DRBS application are:

R1. The application must be able to connect to a remote SDR and stream audio broadcasts and frequency information. The application must also display information of the connected SDR receiver.
R2. The application must have record, store and audio playback capabilities.
R3. The application must support hosting of local SDR devices for local experimentation.
R4. Feature list that must be exposed through the application's user interface. This includes basic user authentication, displaying SDR receiver information, waterfall and GPS location of receivers. User's ability to select frequency, modulation and control volume etc.
R5. The application must implement a use case.

To achieve these requirements, the application must be able to provide the following functionality:

F1. Integration to web SDR hosting platform

To meet the first requirement R1, an API is required. The API is the integration point to the web SDR hosting platform and is responsible for establishing and maintaining the connection. The API contains logic for pulling FFT and audio streams from receivers as well as exposing endpoints for control.

F2. Backend storage

The backend must be able to store I/Q data and audio signal recordings as well as the receiver list.

F3. Configuration and setup of localhost for local SDR support.

F4. UI display and control

The DRBS application must have a user-friendly interface to display information and control panel to change parameters such as frequency, modulation, volume etc. It also provides basic user authentication on login.

F5. DRBS application must scan receivers and search for morse code signals. The application then displays decoded morse code text.

Table 1.1 provides a summary of the tests performed during functional testing. Each requirement was translated into a test case with a defined testing procedure and expected results. In total, 13 test cases were performed with each test case being derived directly from the requirements thus providing a comprehensive assessment of the application's functionality goals.

Table 1.1 Breakdown of tests performed during functional testing.

Test ID	Description	Functions Checked	Requirements Tested
T1 – T4	Connect to OpenWebRX receiver, receive audio streams and frequency information. Display SDR receiver information.	F1	R1
T5	Record, store and play audio.	F2	R2
T6	Add local RTL-SDR tuner.	F3	R3
T7 – T11	Authentication. Display SDR receivers, waterfall and GPS location of receivers. User's ability to select frequency, modulation and control volume.	F4	R4
T12	Scan receivers and search for morse code signal. Display decoded morse code text.	F5	R5

1.6 Scope and Limitations

The research project seeks to design and implement an application framework that provides a collection of pre-built modules, tools and APIs that will allow developers to quickly create innovative applications using broadly available SDR data. The DRBS is a software development kit (devkit) that aggregates web SDRs, helps developers search for broadcast transmissions of interest and build solutions around the streamed transmissions.

The scope of the investigation involves:

- Identification of tasks that can be pre-developed to provide base functionality of the application framework.
- Developing DRBS application, which is a collection of modules, components, code libraries, packages, tools and APIs are required to perform these tasks.
- Implementation of morse code distress signal detection and decoding use case to validate the application's functionality.
- Testing the application thoroughly for functionality and performance.
- Developing document on how to get started with the application and how to use the devkit.

Some of the limitations of the investigation included lack of resources which forced the researcher to use low-cost RTL-SDR dongle (RTL2832U chipset) with low specifications. The low specification equipment affected the quality of the experiment results.

1.7 Document Outline

Section 2 reviews background literature on SDR overview, how cloud technology has democratized data access through web SDR, hardware advancements that have led to an increase in data availability as well as the software development models and frameworks being used currently by the developer community.

Section 3 presents the research methodology for this project. The section starts with outlining the general approach to the software development process, that is, requirements specification, design, development, testing and validation. The requirements specifications are derived from evaluation of existing web SDR platforms. The chapter proposes design approaches and implementation strategies before summarizing the testing methodology. The testing methodology was conducted in two parts i.e., functional testing which uses a set of test cases to verify API features and functionality of DRBS modules as specified by the requirements and performance testing; a metric based integrated approach of testing the end to system. The chapter ends by highlighting the documentation strategy for this project.

Section 4, the Prototype Design which includes a detailed requirements specification. A design of the system description; architecture and components are proffered. More detail is provided on the API design including modelling and validation. A mock up user interface of the morse code implementation is presented before concluding the chapter

with the experiment design, which describes the experiment setup for functional and performance testing.

Section 5 presents the experiment results and provides a brief discussion on the results analysis.

Section 6 concludes the research project. It provides a more generalized discussion of the results and further assesses them against the requirements specifications. It summarizes contributions to the research community and ends by laying the groundwork for future works.

2 Literature Review

This section studies the relevant literature required to provide a good base for the development project. Section 2.1 introduces SDR and provides a high-level description of SDR architecture. Section 2.2 presents web SDR, internet based SDR which allows many users to tune in and listen at the same time. Section 2.3 discusses the most popular application of SDR, FM broadcast monitoring. Section 2.4 concludes the chapter by highlighting the software frameworks available in the SDR industry.

2.1 SDR Overview

Historically, wireless communications relied pre-dominantly on radio hardware which made it expensive and inflexible to configuration changes. Traditional radio uses hardware transceivers to propagate electromagnetic waves through free space for the transfer of communication from one point to another [8]. Radio parameters such as frequency, modulation and transmit power could only be changed by some physical intervention.

Software Defined Radio (SDR) technology has however changed that, as “key elements of the radio are implemented in software” [9]. Software, on the other hand can easily be reprogrammed and reconfigured making it flexible hence less expensive. This flexibility and configurability allow SDR to be multi-functional, support multiple frequency bands, modulation and standards [10], [11].

According to the SDR Forum, Software Defined Radio is defined as "Radio in which some or all of the physical layer functions are software defined" [8]. In essence, SDR is the implementation of radio functions in software.

2.1.1 SDR Architecture

SDR is a radio communication system which consists of both hardware and software components. The level of implementation of radio functions in software may vary, in the ideal case, the Digital to Analog Convertor (DAC) and Analog to Digital Converter (ADC) are at the antenna such that digital baseband signals are converted directly to an RF signal and fed to the antenna [12], [13]. At a high level, the SDR architecture can be described by the block diagram given in Figure 2.1 which gives a brief overview of each component of the SDR system.

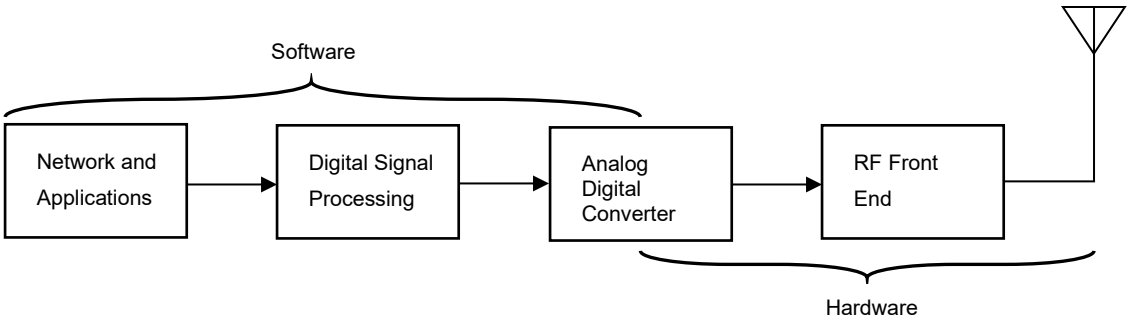

Figure 2.1 SDR architecture [13].

The following section provides a description of each of the major components of the SDR system.

i. Digital Signal Processing

The signal processing block is responsible for baseband processing. Baseband processing includes operations such as modulation and demodulation, coding and encoding, interleaving and de-interleaving, scrambling and descrambling [14].

Advancements in silicon manufacturing and Complementary Metal Oxide Semiconductor (CMOS) fabrication technology have made transistors cheap and widely accessible [12], [13], [14]. This has aided the development of SDR as electronic manufacturer's build highly complex hardware circuitry to improve processing capabilities of SDR systems. Some of the hardware platforms that can be used for signal processing in SDR deployments include:

General Purpose Processors (GPPs): According to [12], [14], these are register based microprocessors which provide the widest range applications. For example, X86/64 ARM architectures used in PCs. Due to their flexibility, availability, ease of programmability and support for a wide rand of OS's, Grayver in [12] suggests that GPP's are suitable for physical layer DSP – SDR functionality. Further studies in 2018 by [14], showed that GPPs may not be suitable after all for high throughput computing. Akeela et al. [14] cited an example of implementing IEEE802.11 standard using GNU's Not Unix (GNU) radio, the 20MHz sampling rate requires parallelism of GPP cores to avoid saturation.

Digital Signal Processors. Defined by [13], [14] as an improvement of GPPs, optimized to handle arithmetic operations particularly multiply accumulate (MAC). DSPs Reduced Instruction Set Computer (RISC) like architecture can deliver high speed performance for SDR deployments [14]. Additionally, hardware accelerators can be employed for complex operations albeit with the unintended consequence of increasing the power consumption. Grayver, [12] opines that the high-power consumption of DSPs coupled with limited OS support and the inherent optimization of instruction sets for specific applications make DSPs an unattractive alternative to GPPs.

*Field Programmable Gate Arrays (*FPGAs) are microchips consisting of arrays of configurable logic blocks surrounded by routing fabric [13], [14]. Their in-built parallelism allows for high-speed performance whilst consuming little power which makes FPGAs more desirable and cheaper than GPPs and DSPs [14]. Figure 2.2 depicts FPGA architecture and shows how it uses parallelism, adaptability, and hardware-level execution to achieve high performance.

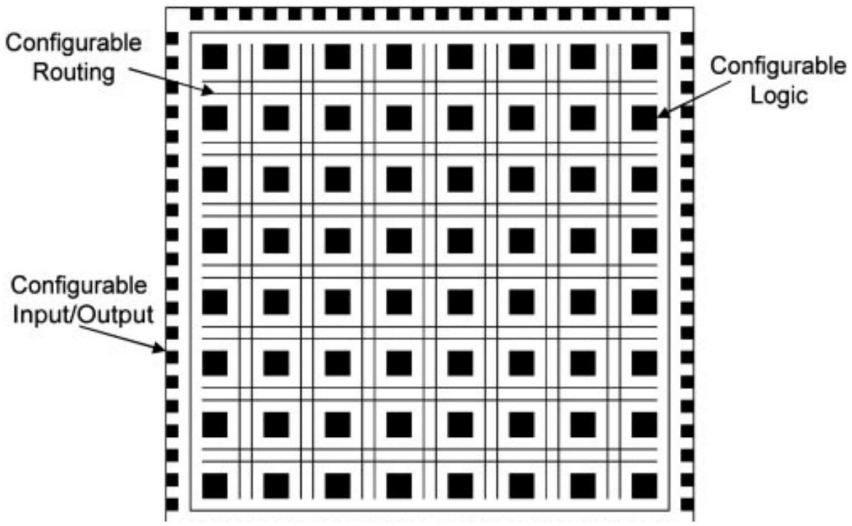

Figure 2.2 FPGA architecture [13].

SDR deployments generally use hybrid approaches which combine several hardware platforms and executable software blocks e.g. SDR system control can be handled by a GPP, logic and numeric operations such as MAC can be processed by a specialized DSP and FPGA based hardware accelerators may also be employed to improve performance [14].

ii. Digital Front End

Introduced in [13] as the bridge between RF and baseband signal processing whose output digital signal (in both transmit and receive directions) must have a defined bandwidth, center frequency and sample rate. The two basic functions of the digital front end (DFE) are summarized by [13], [14]:

- *Channelization* provides for up and down conversion of the signal between baseband and Intermediate Frequency (IF) before further frequency translation in the RF front end. On the transmit path, it up-converts the baseband signal received from signal processors. The DFE is connected to the DAC so that the output digital IF signal is converted to an analog signal before RF up-conversion. On the receive path, it receives the digital output of the ADC which is mixed with a carrier signal from the digital local oscillator to further down-convert the signal to baseband frequency.
- *Sample Rate Conversion* converts the sampling rate to match DSP or RF front end processing which allows for synchronization of timing, frequency, and phase.

iii. Analog to Digital and Digital to Analog Converter

In the transmitter, the DAC is responsible for producing an analog RF signal from a digital IF output signal of the DFE for transmission. In the receiver, the ADC is responsible for changing a continuous-time RF signal to a discrete-time binary signal for further digital processing [14].

Tuttlebee [13] presents three strategies for sampling in receivers, that is direct or Nyquist sampling, quadrature and IF sampling. Direct sampling must satisfy the Nyquist condition which specifies that the sampling rate must be at least twice the highest frequency component of the sampled signal. Anti-aliasing filters must be employed to remove harmonic frequency components. Quadrature sampling involves splitting the analog input into two signal components i.e., in-phase and quadrature which are orthogonal to each other. Whilst this technique reduces the sampling rate by two, it requires two phase-locked ADCs [13]. IF sampling, also known as sub-sampling, can also be employed typically in super-heterodyne receivers where the RF signal is first down converted to intermediate frequency before processing.

Inherently, converters are prone to quantization noise due to binary estimation of a continuous-time analog signal. Though unavoidable, oversampling is one of the methods that can be employed to mitigate quantization noise [13].

iv. RF Front End

The RF Front End (RFFE) block is responsible for the transmission and reception of RF signals. Grayver [12], Tuttlebee [13], Akeela and Dezfouli [14] summarize the basic functions as down and up-conversion of an RF signal, channel selection, interference rejection and amplification. Furthermore, they discuss the techniques for converting baseband digital samples to RF waveforms and vice-versa. Direct RF synthesis or direct conversion allows for the conversion of digital baseband to RF signals in one step thus lowering the cost and size of transceivers due to the fewer parts required [12], [13]. However, to avoid aliasing, the sample rate must be at least twice the carrier frequency, making it impractical for modern wireless communications e.g., may operate in the Extremely High Frequency (EHF) band.

Super-heterodyne transceivers on the other hand use two steps to transition from baseband to RF i.e., additional mixing and filtering stage is introduced to first convert the baseband signal to IF before further translating it higher carrier frequencies.

Zero-IF conversion is also popular due to its lower sampling rate and power consumption [12]. The sampling rate is proportional to the bandwidth of the signal and not the carrier frequency. Since the center frequency of the input signal is DC 0Hz, it suffers greatly from flicker noise and image rejection of the I and Q signals generated by the mixer [12]. Two DACs are also required for the IQ signals. Direct-IF conversion provides a solution to flicker noise by sampling at a rate proportional to IF, thus using a regular mixer instead of the complicated IQ mixer [12].

Modern SDR deployments require that receiver architectures have a high dynamic range (wideband) whilst filtering to protect the signal of interest against interference. [13]envisions this flexibility to be achieved through ‘pure digital RF front ends’ i.e., placing the ADC/DAC at the antenna thus essentially sampling RF waveforms. Unfortunately, due to the high sampling frequency involved, extremely high-power consumption is also expected.

v. Antenna

Antennas are passive devices that radiate and collect electromagnetic waves. For SDR to achieve the flexibility, certain requirements must be met by antennas.

Dynamic tuning wideband – from very low frequencies to extremely high frequencies [12]. Not practical since antennas are mechanical structures which rely on physical dimensions.

2.1.2 Radio on a chip

One of the key drivers of all programmable, 'radio on a chip' [13] SDR systems is the transfer of RF processing to the digital domain. Tuttlebee [13], Akeela and Dezfouli [14] agree that replacing analogue devices such as mixers, Local Oscillator (LO) and filters in the RF front end and replacing them with an RF sampling ADC effectively digitizes the RF front end and offers more flexibility and wider bandwidth [14]. However, operating the ADC at high frequencies also increases the system power consumption significantly. Tuttlebee [13], Xilinx and Inc [15] highlight this as one of the major drawbacks facing SDR evolution: data convertors must be able to support multi-gigabit sample rates without increasing the power consumption. RF front end and digital front-end interfaces must also be able to handle the massive amounts of data moved between the two processing blocks.

Secondly, [15] cites the need to increase the number of antennas to build MIMO systems able to cope with increasing capacity. Whilst this approach increases array gain, it increases the system complexity when interfacing with convertors. Lastly, the ever-increasing demands on SDR to be more flexible and reduce the time new products or features go to market. One of SDR's biggest selling points is flexibility and SDR is required to support a diverse range of applications on the same hardware [15]. Tuttlebee [13], Xilinx and Inc [15] propose SDR implementation on a single device. To reduce the complexity of the RF front end and reduce the time to market, Figure 2.3 [15] presents their "all programmable SoC and FPGA with integrated RF sampling data convertors".

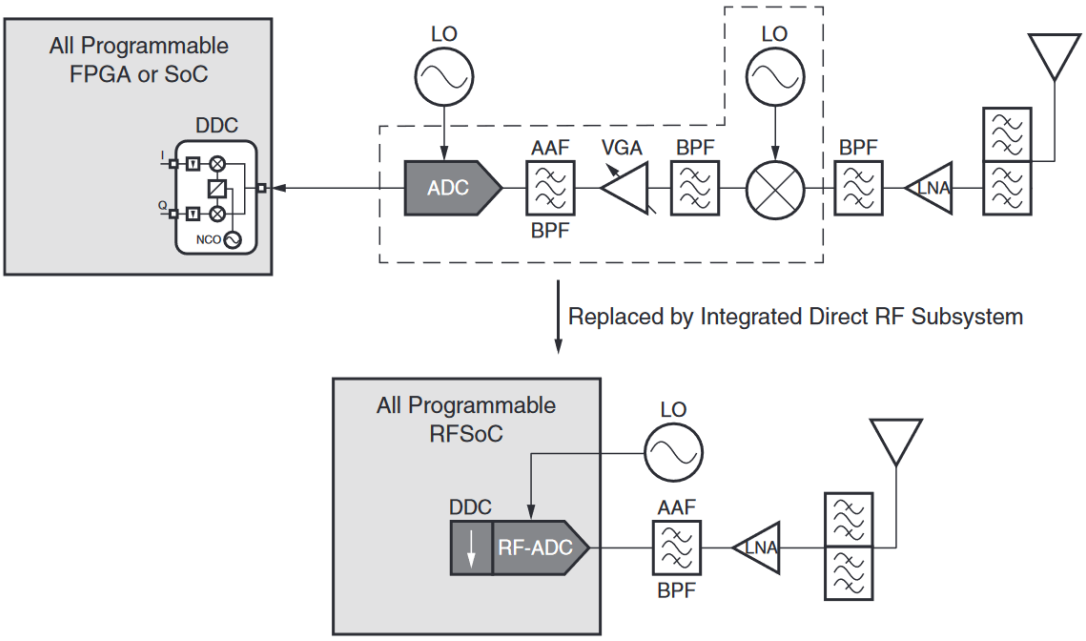

Figure 2.3 Comparison of a superheterodyne receiver to a direct RF sampling receiver [15].

2.2 Web SDR

A web SDR is an internet-based Software Defined Radio (SDR) receiver which allows many users to tune it and listen at the same time. It is a program that can connect to a Realtek RTLSDR, as well as to other types of SDR peripherals, and provide the facility to host a website (on the machine that the SDR peripheral connects to) that allows users with access to that website to tune in and request changes to the centre frequency, perform possibly multiple types of standard demodulation techniques, and stream either the raw or demodulated data to the user's PC. The Web SDR architecture pictured in Figure 2.4 provides a platform to create a network which connects several SDR radio devices to the internet through a website. A network of connected SDR nodes can provide wideband coverage across large geographical areas [16].

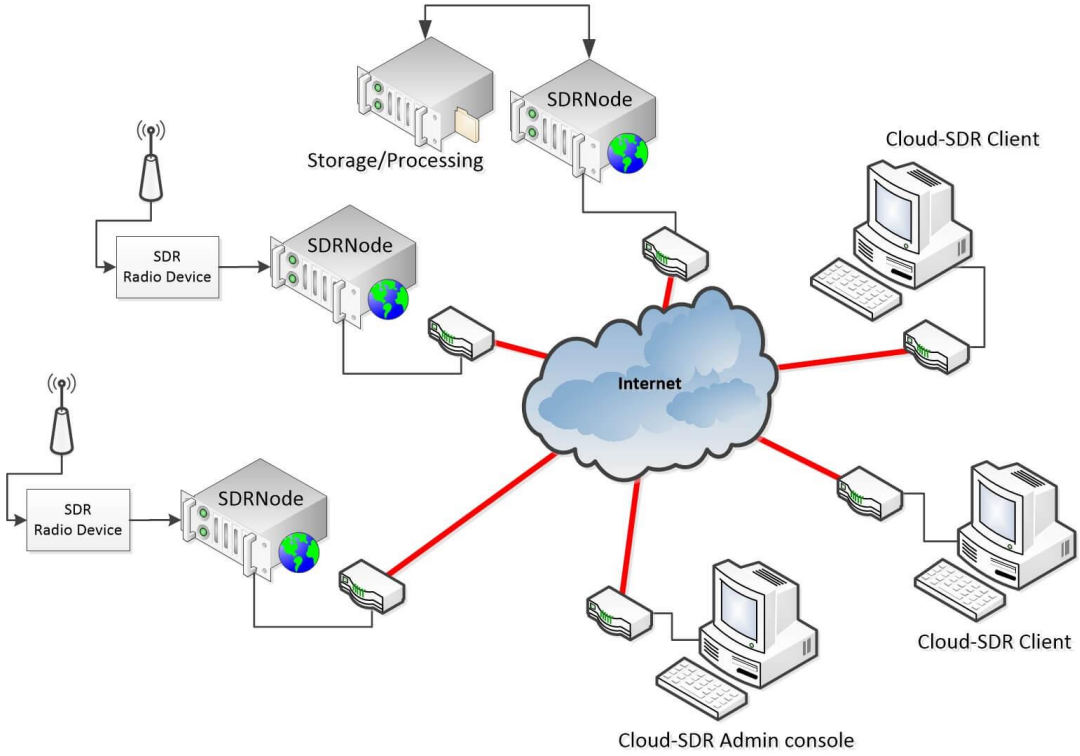

Figure 2.4 Web SDR architecture [16].

The Cloud SDR Admin console provides a centralized network management system with control capabilities to remotely manage SDR network nodes. The user can access the SDR network and tune in to various SDR radios though the Cloud-SDR Client.

There are several Web SDR implementations such as OpenWebRX [17], ShinySDR [18], WebRadio [19] and WebSDR.org which are independent platforms that provide specific monitoring applications over a limited frequency range but none of the publicly accessible implementations exhibit dynamic wideband spectrum sampling across multiple receiver feeds [20]. For example, websdr.org as of 12 February 2025, monitors 704 MHz of radio spectrum using 148 active Web SDR servers. Each server runs independently and hosts a specific application over a limited frequency range. To find a particular broadcast transmission such as a news or music channel, a listener may potentially have to separately tune into each of the 169 receivers and generate thousands of audio streams which they will have to listen to one by one. Making it a tedious and time-consuming process.

The ability to cycle across and scan separate Web SDR feeds on a single platform and automatically find a specific broadcast stream would make the system more effective and efficient. This could be useful for applications such as:

- listening to emergency calls for rapid response services such as emergency medical services, ambulance, police etc.
- triangulating a position from a signal source for tracking purposes e.g., aircrafts, boats etc.
- spectrum analysis and searching for RF noise sources
- security surveillance and RF signal sniffing etc. [20]

2.3 FM Broadcast Monitoring Approaches

Many years ago, before the Internet, shortwave provided a means to receive messages over the horizon. Shortwave radio users could get updates about world news from hundreds of kilometres away. Shortwave was named in early 1900s close to the invention of radio – and got its name because the waves were – at that time – considered shorter than the other options at an amazing (at that time) frequency of around 3-30MHz. But a nice characteristic of short-wave radio was that it could bounce between the ionosphere and Earth, propagating for long distances. Nowadays FM transmissions are prevalent for public radio broadcasts… but we can still 'pick up' these from continents away – not necessarily by tuning in to them directly but rather tuning in to them remotely through streamed sample data.

2.3.1 Hardware-based approaches

The RTLSDR dongle is a highly versatile, computer-based Software Defined Radio (SDR) receiver, that can be tuned to perform fairly wideband sampling (a width of 1MHz around an adjustable centre frequency) [20]. This can sample multiple FM or AM public radio broadcasts. Other types of transmissions and modulation can also be received with the right kind of digital processing performed.

Several architectures of SDR being implemented on integrated circuits such as RTLSDR and Universal Software Radio Peripheral (USRP) were studied and was summarized in this section. Monitoring FM broadcast is costly but often very necessary. Due to its wideband nature, broadcasts typically cover a large

geographical area. Juhana [16], Angelov et al. [21], Moshkov et al. [22], Juhana and Girianto [23] present SDR based FM broadcast monitoring systems. The architecture generally consists of a radio front end (e.g., Realtek RTL2832U) connected to a computer loaded with SDR software where all the signal processing is done. This is depicted in Figure 2.5 which shows a simplified block diagram of a Realtek RTL2832U radio front end connected to a computer which hosts the DSP functions through the USB port.

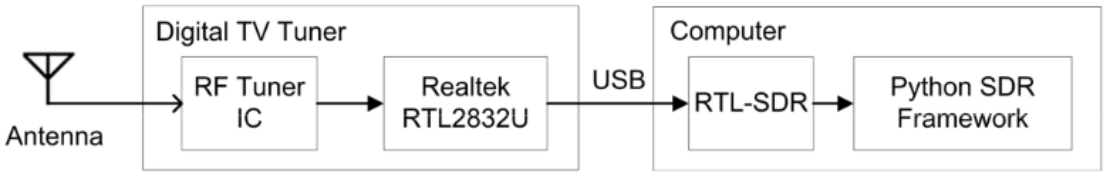

Figure 2.5 Block diagram of a low-cost SDR kit [24].

Uengtrakul and Bunnjaweht [24] designed the system in Figure 2.5 as an experimental kit to demonstrate through simulation, wireless communication and signal processing. Python SDR framework is employed to capture and process IQ samples.

Angelov et al. [21], Moshkov et al. [22] present case studies of their development of SDR systems to physically monitor FM radio broadcasts in Gabrovo and Russia, respectively. Both case studies use RTL-SDR RTL2832U demodulator and web servers for remote monitoring and control. [21] architecture consists of two parallel RTL-SDR modules designed to independently monitor VHF and UHF RF bands.

According to [25], signal processing at the sensing node is computationally intensive and reduces the performance of the network. A centralized processing architecture is proposed where the control centre built using GNU radio is responsible for signal analysis, displaying measurement results and listening to the audio signal. and remote stations are built using RTL-SDR and USRP devices. Remote SDR devices detect and capture the RF signal, digitize it and send it over the network to the control centre as I/Q samples. GNU radio is an open source SDR framework that uses processing blocks and allows signal processing to be implemented in software [25].

2.3.2 Crowdsourcing using IoT receivers

Spectrum monitoring systems are localized and often limited to specific geographical regions such as countries. Considering shortwave transmissions can travel hundreds of kilometres, it is prudent to consider a spectrum monitoring application which transcends geographical boundaries. From Section 2.3.1. it can be noted that hardware-based approaches are costly due to the number of sensing nodes required to cover a large geographical area. A distributed approach is therefore proposed.

Calvo-Palomino et al. [26] introduce Electrosense+; a web-based spectrum monitoring platform that relies on crowdsourcing. Other platforms such Microsoft Spectrum Observatory and BM Horizon, require expensive Software Defined Radios and offer poor scalability options whilst applications such as OpenWebRX, Airspy, WebSDR have limited applicability. Electrosense+ is a 'peer-to-peer spectrum sensor network' which incentivises users hosting IoT sensors with digital coins. The sensor node is a low-cost device consisting of an RTL-SDR dongle RF front end and Raspberry Pi signal processor. The platform uses an 'Automatic Channel Identification' algorithm which scans, selects, and decodes different spectrum applications such as FM or AM radio, ADS-B aircraft messages, AIS, ACARS and LTE.

2.3.3 Web application-based

Juhana [16], Girianto [23] propose an FM broadcast monitoring system that can be remotely accessed via the Internet. The system is integrated with OpenWebRX which allows monitoring of up to four stations and provides a web browser-based user interface for ease of configuration [23]. It also allows for live streaming and recording as well as independent and simultaneous user access.

2.4 Software Frameworks for SDR

One of the major benefits brought about by SDRs is the ability to be reconfigured through software which provides a high degree of flexibility, allowing for the same hardware to be used for various applications. Robert M. et al. [27] stresses the importance of software frameworks in developing SDR software applications in components (modular approach) making software engineers more efficient and productive in creating SDR based software solutions. Robert M. et al. [27] define two categories of software frameworks namely:

i) Processing frameworks such as GNU Radio, Octave, R and NextMidas which focus on the different digital signal processing stages of a system and how they communicate.

ii) System frameworks developed for complex applications and span across different infrastructure and integrated components. Examples include Common Object Request Broker Architecture (CORBA) Component Model (CCM) [28], Open Service Gateway Initiative (OSGi) [29] and Software Communications Architecture (SCA) [30]. CORBA Component Model (CCM) defines *components* in [28] as a specialized collection of features and is the most basic meta-type in CORBRA. CCM offers different models and frameworks such as the CCM Implementation Framework, Container Programming Model, packaging and deployment models [27]. CCM is a complex model that spans across several implementations. It also provides tools and runtime environment with specific code syntax and semantics making the developer learning curve very steep.

Open Service Gateway Initiative (OSGi) according to [27] is a Java-based set of specifications that provides guidelines and an API for implementation by a vendor. OSGi is an open-source, service-oriented initiative that also uses components (modules) to enable development, deployment and management of software applications across several Java VM instances [29].

Software Communications Architecture (SCA) is an open architecture developed by the U.S. Department of Defense specifically to standardize the development of SDR [31]. SCA, illustrated by Figure 2.6 also defines a component model that runs on COTS hardware through standards interfaces but does not prescribe implementation details. It provides a common environment to deploy applications across platforms with different hardware [31]. However, SCA relies on CORBRA middleware which [31] describes as “adding a layer of complexity”.

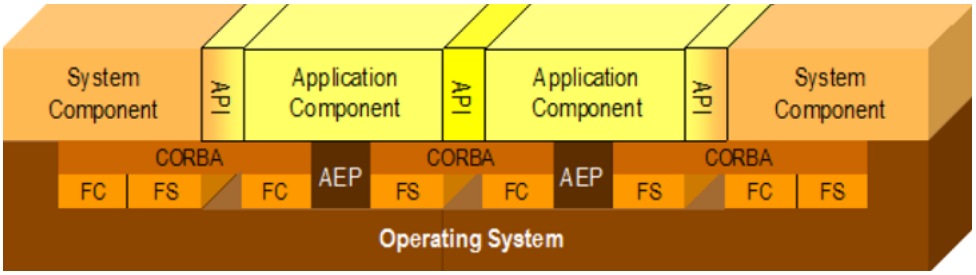

Figure 2.6 SCA architecture diagram [32].

After studying the system frameworks summarized above, [27] concluded that the three system frameworks are too generic and lacked the processing chains required for SDR implementation. Considering these shortcomings, [27] goes further to propose REDHAWK [32] which combines attributes of a processing framework (similar to GNU Radio) as well as a component model which can be deployed across distributed infrastructure environments thus making it both a specification and an implementation. According to [27], REDHAWK contributions include a service platform that manages the deployment, and configuration of components, base classes to support in the development of new components, code generators, hardware abstraction, data/messaging API, control API for the management of RF hardware, support for C++, Python or Java implementation etc.

It is evident that processing and system frameworks allow for configurability of SDR hardware and are essential in enabling the flexibility required in SDR technology. However, these frameworks are difficult to scale due to their complexity. A higher-level framework focussed on aggregation of already available SDR sampled/streamed data is required for larger scale innovation.

3 Methodology

3.1 Introduction

This thesis is premised on developing the DRBS, an application framework that provides pre-built code libraries, components and packages, APIs, tools, tutorials and documentation that allow developers to create new software applications using available streamed data services from remotely hosted web SDR receivers. To connect to remote SDR receivers, DRBS API must integrate to a web SDR platform that hosts a significant number of these receivers and provides a broad range of transmissions and modulations.

If not managed well, building a software platform can be a tedious task. Developers and software project managers must take due care to produce high quality software which meets the functional and performance needs of the user. The software development lifecycle (SDLC) defines seven (7) phases of software development i.e., project planning, requirements gathering and analysis, system design, coding or implementation, testing, deployment, and maintenance. Taya and Gupta [33] suggest an amalgamation of some of the phases thus generalizing the software development process into four phases i.e., requirements specification, design, development (or coding) and testing (or validation).

Software development approaches further categorize these phases into process models depending on the project or product. The five most popular software development process models according to [33], [34], [35], are Waterfall, Spiral, Incremental (Iterative), Rapid Application Development (RAD or Agile) and Rational Unified Process (RUP). The sequential and rigid approach of the Waterfall model makes it suitable for projects whose deliverables are clear, and a well-defined plan can be developed, it however does not incorporate the risk of uncertainty which is prevalent in most software projects [35]. Rapid and agile development models on the other end break the project into very small projects or software releases based on incremental functionality [35], [36], [33] argue that this highly flexible approach significantly reduces project risk as it quickly delivers a product to the user for early testing and acceptance. However, [36] supported by [33] have a diverging view, highlighting concerns that the project can easily degrade into a code and fix affair. The spiral model is often viewed as a compromise between the rigidity of the Waterfall model and the almost lack of processes of rapid application development models. Boehm [37] a major proponent of

spiral development, advances several characteristics which are key to the success of a spiral development undertaking in reducing cost overruns, completion delays and scope creeps. These characteristics include: the cyclic nature which is critical for continuous risk analysis and stakeholder engagement, inherent parallel execution, and the incremental nature where each iteration builds on the previous spiral prototype [37]. The characteristics are illustrated in Figure 3.1.

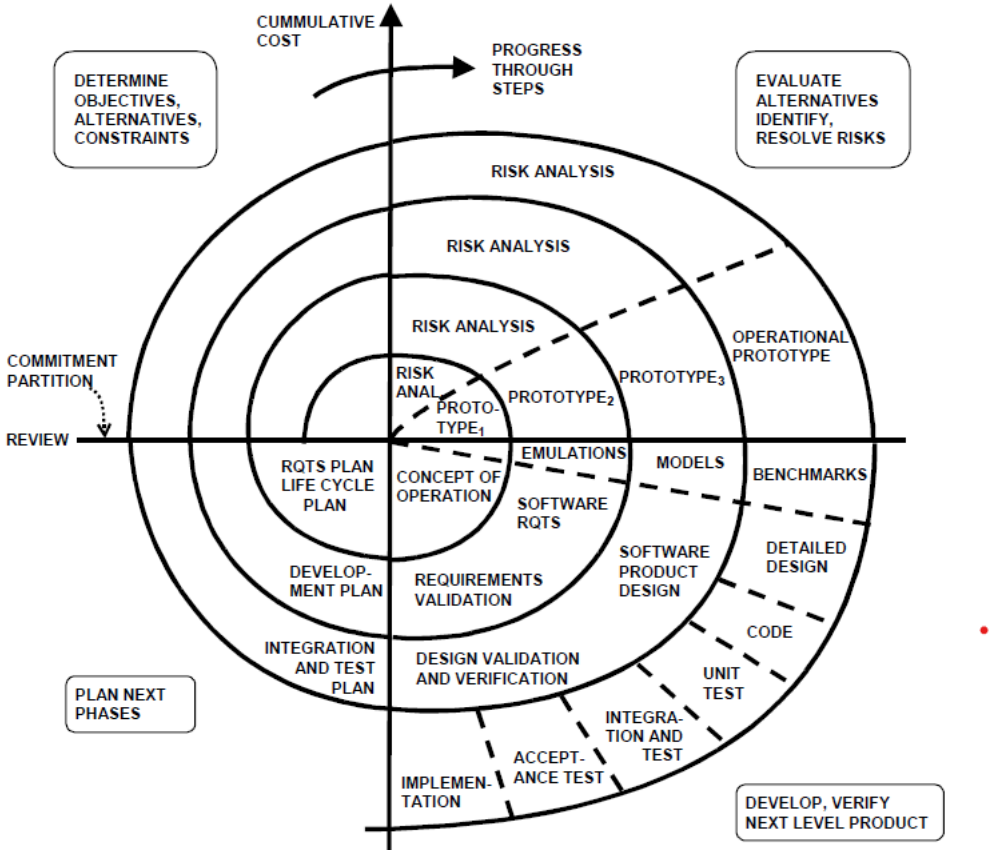

Figure 3.1 Spiral development process [37].

Boehm further advancements his argument in [38] and proposes further enhancements through “incremental commitment” and presents practical evidence from successfully implemented industry projects. Similarly, development of the DRBS application will follow the spiral model, with incremental changes added with each iteration to address the requirement for ongoing feedback through the testing loop. The first iteration will include planning and user requirements gathering, high level design of the proposed solution, development of the API and supporting software as well as functional testing. The first iteration may include initial performance tests for general metrics such as latency and response time.

The second iteration will incorporate feedback from the first round of functional and performance testing and will go through the requirements specifications (refinement of original specifications), low level design (more detailed), coding, and testing. Further iterations may be conducted to meet the minimum acceptance level for the application.

This chapter's structure will follow the main phases of the software development process i.e., planning, design, code, and test. Section 3.2 will start by evaluating existing web SDR platforms to establish the initial requirements i.e., requirements gathering and development of specifications. Section 3.3 presents the design methodology. The section considers best practise design approaches and discusses API modelling and system design workflow. A discussion on the implementation methodology which will guide the development and coding of the application will ensure in Section 3.4. Section 3.5 addresses the last phase of the software development process, testing. Functional and performance testing methods and techniques was considered to validate the design. Finally, Section 3.6 will provide a high-level outline of how the software application was documented.

3.2 Requirements Specifications

The Requirements Specifications section defines all functional, non-functional and performance requirements crucial for the successful development and deployment of the DRBS application.

3.2.1 Evaluation of existing web SDR platforms

Most of the existing published web SDR applications focus on development of a web server and browser-based client as well as hosting of SDR receiver hardware such as USRP, ADALM Pluto, RTL2832U tuners etc. on the web server. An API based server client communication is initiated when a client opens a web browser. Retzler and Horváth [39] uses rxws.py; a server-side WebSocket protocol for establishing and maintaining the connection to the server back end and JavaScript based front-end application.

Suzuki et al. [40] presents the case for a cloud-based storage approach which separates the UI from SDR development environment for processing distribution. The authors also implement WebSocket protocol for communication and JSON RPC 2.0 for sending payloads over the connection. JSON RPC 2.0 is responsible for data structuring and formatting over sockets and http connections [41]. JSON-RPC Working

Group [42] prototypes libWebSockets (LWS) for running http and WebSocket connections on a single TCP/IP port. LWS also runs on top of WebSockets and provides the ability to manage several connections on a single thread using a “non-blocking event loop” [43].

Table 3.1 Comparison of features of existing web SDR streamed data platforms [18], [44], [45].

Features	**WebSDR**	**OpenWebRX**	**Shiny SDR**	**KiwiSDR**
*Number of receivers	145	445		848
Supported audio transmissions and radio broadcasts	No client-side software	AM, FM, LSB, USB, DMR	AM, FM (narrowband and broadcast), SSB, CW	AM, AMN, CW, LSB, USB, CWN, NBFM
Supported telemetry data	No client-side software	APRS, WSJT-X	APRS, Mode S/ADS-B, WSPR, all those supported by rtl_433	WSPR, TDoA, WEFAX
Map locator	Yes	Yes	Yes	Yes
Waterfall display	Depends on connected web SDR	Persistent	Persistent	Persistent
Server OS supported	Linux	Linux	Linux	Windows, Linux, and Mac

Hardware support SDR devices	Not hardware specific, most SDR devices	Most SDR devices	RTL-SDR, HackRF, USRP	KiwiSDR (BeagleBone)
Web browser	Google Chrome, Mozilla Firefox, Microsoft Edge, Safari, Opera	Google Chrome, Mozilla Firefox	Google Chrome	Firefox, Chrome, Safari and Opera
Internet bandwidth restrictions	Minimum 100 kbit/s	About a 100 kbit/s uplink or greater	3Mbps or greater	
Mobility support	Android, iOS		Phone, tablet	Android, iOS
Source code	Closed source	Open source	Open source	Partially open source (derived from OpenWebRX source code)
Support for GNU Radio		Yes	Yes	
Multi-user support	Yes, independent, and simultaneous	Yes, independent, and simultaneous	No	Yes, up to 4 independent and simultaneous users

Documentation	FAQ	Manual or installation guide, developer guides, API	Manual or installation guide, developer guide	Manual or installation guide

**This number varies and depends on the number of user hosted receivers coming online and going offline.*

Table 3.1 summarizes the key features of existing web SDR streamed data platforms. KiwiSDR hosts the largest number of remote SDR receivers with over 800 receivers listed at any given time and more than 1000 users listening [44]. It also supports key audio and broadcast transmissions such as FM, AM, and CW as well WSPR telemetry data. However, having been developed using OpenWebRX code base, to the knowledge of the author, kiwiSDR source code is not freely distributed to the public. Like websdr.org, where the owner of the site has openly stated that they do not distribute the software. Whilst kiwiSDR and websdr.org are the most popular amongst users, their closed source approach is a major hinderance for the developer community. On the other hand, ShinySDR is not suitable for multiple users as it does not support independent sessions and is CPU resource intensive [18]. To the researcher's knowledge, there is no platform that is aggregating the already existing web SDR applications such as webSDR, OpenWebRX, kiwiSDR, Shiny SDR etc. The market is devoid of an open-source streamed data platform hosting a significant number of remote SDRs which researchers and developers can easily integrate with to experiment and develop innovative software applications using already available data.

For this research endeavour, DRBS prototype will integrate with OpenWebRX because it has the most accessible software and hosts a significant number of receivers, sufficient to validate the researcher's assumptions. OpenWebRX is distributed in Linux only but supports most SDR hardware devices such as RTL-SDR which was used for this research.

3.3 Design

This section outlines the design methodology adopted for the development of the DRBS application, with a focus on the overall design approach, API model, and architectural decisions that guided system implementation.

3.3.1 Design approaches

- Client-server model

  The client-server model is vital for separating the front end (client side) code which is executed in the browser from the backend (server) which will provide the API logic [45].
- Modularity

Modules or components are groups of related code which execute a particular function [46]. Modularization allows for loosely coupled yet cohesive code which is easy to read. The API design will create modules which will make variables and functions private for internal implementation and public when exporting functionality to other modules. This also makes the software scalable and easier to maintain.

- Open source

  For customization and modification of source code, the application will adhere to the criteria published by the *Open-Source Definition* [47] such as free redistribution, access to source code, technology neutrality etc. This is a feature that makes it useful for the research community.

- WebSockets

  WebSocket is a connection-oriented protocol that runs on top of TCP to enable full duplex communications between a client and server [48]. WebSocket protocol is well suited for our application because of the real-time (low latency) requirement as well as the potential for multi-user support. Unlike HTTP streaming which may incur server overload due to its reliance on long polling [40].

- Non-blocking and asynchronous architecture

  Asynchronous architecture allows the software to handle multiple requests using a single thread by releasing the thread once a request is received and using the callback function when a task is completed. Non-blocking allows the program to continue executing other tasks whilst waiting for a response. The Communications API and OpenWebRX APIs will use these features for maintaining responsiveness during real-time execution and handling multiple operations.

- Programming languages and frameworks

  DRBS application is a web browser-based application hence will rely on HTTP requests and responses. The interactive and dynamic UI functions make JavaScript the programming language of choice. OpenWebRX API for receiver metrics */metrics.json* and status */status.json* as well as feature detection *api/features* endpoints are

provided in JSON format [49]. For seamless, full stack execution; JavaScript fabric was implemented in Node.js i.e., React framework for frontend development and Express.js for the backend.

3.3.2 API Model

The model, design, build, test, and document API workflow was adopted for consistency whilst incorporating agile approaches such as parallelization to meet delivery timelines. The 5 step API modelling process proposed in [50] was used to develop a customizable API. The participants who will interact with the application include internal and external developers, users, OpenWebRX platform and SDR receiver hardware such as RTL-SDR. A UI-driven approach was used to identify the activities that would be undertaken by the participants. Each activity will then be broken down into steps to address the functional requirements. The steps were categorised into groups i.e., API endpoints which was mapped into HTTP methods in the API design phase. A user interface wireframe was also created using *draw.io* to visualize the methods and account for all the activities.

3.3.3 Approaches to System Design

Now that the design choices have been made, designing the actual API for the systems that will receive the data was the next step. The API model identified resources and activities as well as their relationships. API endpoints were then be mapped into HTTP methods. Expected HTTP response codes were added based on how each API endpoint should behave including error handling procedures. The API modelling process was presented in section 3.3.2. Section 4 will cover Low Level Design (LLD) aspects and go into further details of the system design and writing the API specification. It will also get into detail on the selected technology architecture and how the system components interact i.e., RTL-SDR hardware, DRBS and OpenWebRX software applications, hosting, storage etc. A use case for morse code sensing, detection and decoding flow of execution was also presented.

3.4 Implementation

The next step was building the API and developing the system. A hybrid API first/design first approach was used. DRBS application is anchored on the integration with OpenWebRX hence the focus on developing the API functionality then building software components around the API. OpenWebRX API is publicly available and provides the primary endpoints required to start the API design process. JavaScript based frameworks and libraries such as Node.js, Express JS and React were then be used to create models and routes, generate code from the API specification as well as implementing API endpoints and backend logic. Clear definition of endpoints and data was critical for communication between the different components. A user interface mock design was required upfront (or at least early in the development) for this approach to be effective.

Nonetheless, since OpenWebRX is a third-party application, the behaviour of the API may not be as expected hence the requirement for parts of the software to be developed incrementally with corresponding iterations of the API design as proposed in Section 3.1 which proposes the use of Boehm's Spiral Model. This allows for a flexible approach as API requirements may change during development. The downside, however, is that the code may not follow best practises and difficult to document. In general, the implementation methodology will follow the outline below:

- High-level API design
- Implement code to test the endpoints and request/response methods
- Refine and standardise API specification

3.5 Testing

Experiments were conducted to test DRBS application functionality and performance, and a case study was used to validate the system design. Three types of testing methods were used for validation i.e., unit testing, functional testing, and performance testing. Unit testing, also known as code level testing

[50] will test specific code blocks for internal functionality during the development phase.

3.5.1 Functional testing

Functional tests were conducted against the API specification to confirm end to end functionality of components and end points. Functional testing used a modular approach that verified if certain components of the system were producing the expected output from a user perspective. Some of the fundamental tests that was conducted include tuning into OpenWebRX receivers – display receiver status and GPS location, receiving and storing audio streams as well as frequency information, ability to select modulation, frequency, and volume control etc. The algorithm to scan and search for specific information such as morse code signals will also be tested and verified by text display of decoded morse code.

A product test document was designed and used to test API features.

3.5.2 Performance testing

This section provides an overview of the performance testing methodology and metrics to be used for testing the application.

1) *End to end latency*. The message handling performance testing methodology like the one used by *Becker and Starobinski* in [51] will used to evaluate the performance of data passing capabilities of the API. A local RTL-SDR receiver was used to benchmark and compare a locally acquired signal against a remotely sampled signal.
2) *Average server response times* were measured using API performance assessment tools.
3) *Morse code signal detection and decoding.* An RTL-SDR receiver (DVB-T TV tuner with RTL2832U chip) connected to a Linux host can be used to scan the air interface for FM spectrum band signals typically from local FM radio stations such Star FM (89.7MHz, ZiFM 106.4MHz in Harare) [20]. Because morse code is rarely in use anymore, except by researchers and amateur radio enthusiasts, morse code transmissions were emulated by playing morse code audio signals from online recordings. A cheap FM

modulator (transmitter) device can be used to convert these morse code recordings into an RF signal for local broadcast. The transmitted signal can then be picked up by the RTL-SDR receiver. Experiments were conducted to verify the DRBS can scan and detect morse code signal and then decode the detected morse code audio signal into text.

3.6 Documentation

This research's approach towards documentation aimed at providing customizable, comprehensive, complete, and simple to understand reference documents to allow adaption of the open source, web SDR API by academia and the developer community at large. The fundamental documents which were provided include:

i. Source code control through GitHub.
ii. A README file which provides a set of instructions for installing the application.
iii. A tutorial which is essentially a getting started guide that underlines the basic steps for running the application.

4 Prototype Design of the Digitized Radio Broadcast Seeker (DRBS) Application

This chapter presents the low-level design of DRBS main application, a web browser based SDR aggregation platform which will make it easier for developers to search and find transmissions of interest. Reducing platform complexity will allow developers the opportunity to focus on creating software applications. The prototype design incorporated the architectural decisions proposed in *Section 3.3.1 Approaches to System Design* such as client-server model, modularity, and non-blocking as well as asynchronous architecture.

4.1 Initial Requirements

This section describes the functional requirements for the DRBS prototype.

R1. The application must be able to connect to a remote SDR and stream audio broadcasts and frequency information. The application must also display information of the connected SDR receiver.

a. OpenWebRX API should establish and maintain connection to OpenWebRX.

b. The API must be able to pull FFT (frequency) and audio streams from OpenWebRX receivers.

c. The API should expose endpoints for receiver details and UI configuration such as GPS and waterfall display, modulation etc.

R2. The application must have record, store and audio playback capabilities.

a. The backend must allow for I/Q data and audio signal recordings storage as files.

R3. The application must support hosting of local SDR devices for local experimentation.

a. Application must support local RTL-SDR tuners.

R4. Feature list that must be exposed through the application's user interface.

a. The system must be able to do basic user authentication on login.

b. The application must have a user-friendly interface to display information about SDR receivers pulled from OpenWebRX.

c. The display must show a real time OpenWebRX-like waterfall as well as GPS location of receivers.

d. The user must be able to select a centre frequency within the available radio frequency spectrum from signal source or SDR receiver.

e. The user must be able to change receiver parameters such as modulation, volume etc.

f. The application must be able scan across multiple feeds and attempt to search for specific information such as morse code signals.

R5. The application must implement a use case.

a. The interface must display decoded morse code text.

4.2 System Description

This section describes components of the main DRBS application and how they interact with each other. The section starts with an overview of the main application (DRBS) architecture and then details the development project structuring.

4.2.1 DRBS Architecture

Figure 4.1 is a high-level depiction of DRBS architecture. The DRBS application is hosted on a single machine, the separation between the DRBS backend and the user (DRBS client front end) is a logical separation. The diagram also illustrates how data sources such as OpenWebRX instances (receivers) and physical SDR receivers locally hosted on the machine are connected to the main DRBS application.

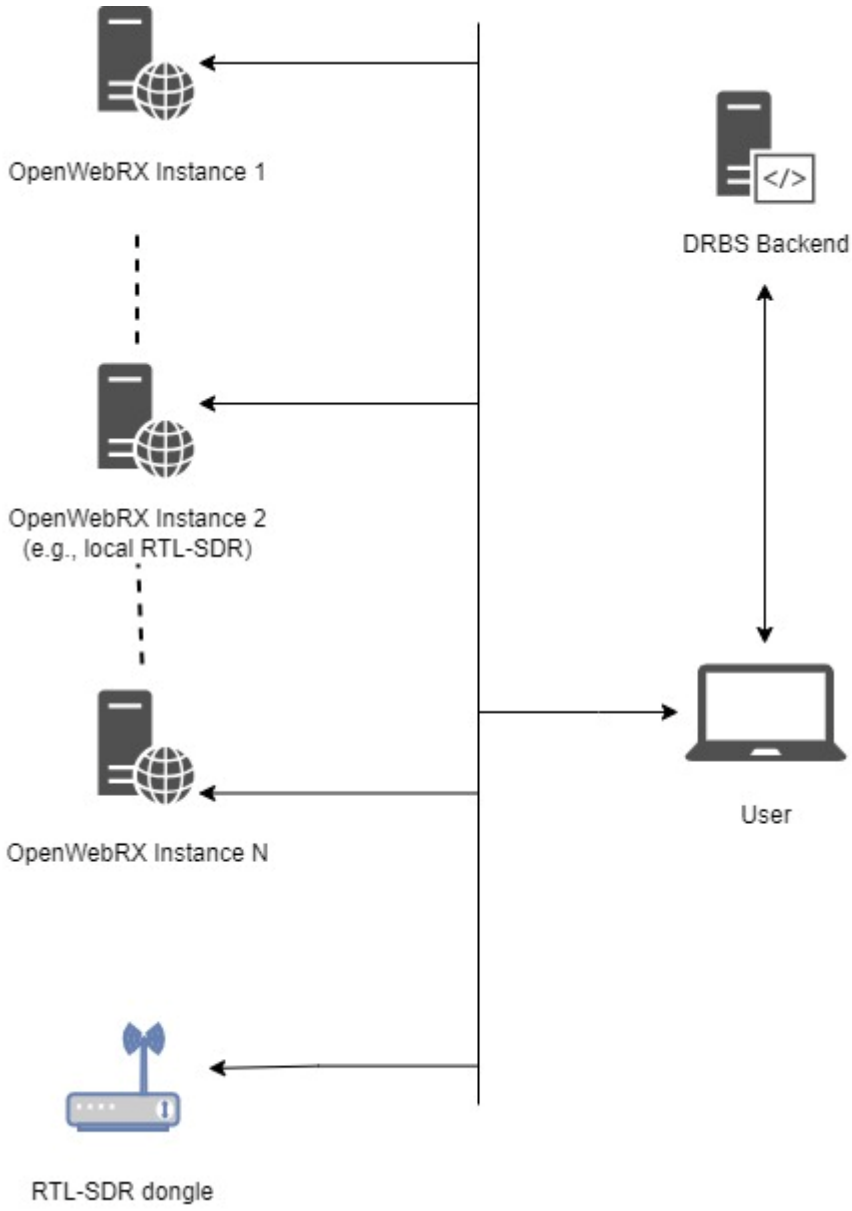

Figure 4.1 Architecture of DRBS main application.

4.2.2 Project Structure

This section refers to Section 3.3.1, which discusses the design approaches. Node.js application framework was selected for implementing the DRBS application as a project because it ticked most of the boxes. Below is a summary of Node.js selling points and why it was selected for implementing the main DRBS application.

- Node.js application is not just a framework but also a runtime environment i.e., it provides end to end code execution through frameworks such as Express and React and communicates with the OS [52].
- Node.js is a web application framework thus suits DRBS application which is front end heavy i.e., a lot of functionality was implemented in UI.
- Supports modularity through node modules. This enables hardware abstraction, grouping of code based on function makes it easy to read and document at a high level whilst keeping the low-level detail abstracted. Modularity is also important for scalability, adding functionality is a matter of introducing a new module. This segmentation allows developers the ability to test new code and new features without disrupting current functionality.
- Node.js is a free, open-source, cross-platform [52]. This is critical for developer community engagement.
- Node.js uses directories which are important for separating the server side (backend) from the front end (frontend) logic [45].
- Supports non-blocking asynchronous request handling [53].

The Node.js main application project root folder contains the following major directories:

- ***api*** folder which contains project source code for DRBS application backend logic.
- ***ui*** folder containing files with front end logic for handling requests. DRBS uses Single Page Architecture (SPA) hence most of the functionality is executed in the front end. The folder contains user control components such as authentication, frequency and modulation controls, receiver selection etc. The directory also contains css styling, html, and client side javascript files for web page rendering.
- *README.md* and *design* folders contain project documentation.

DRBS Frontend

The structure of the ui directory comprises multiple sub-directories and files for end-to-end execution logic. DRBS front-end serves two primary purposes i.e., client end execution and providing communication with the server backend application. Figure 4.2 below illustrates the sub-directories and file collections hosted under this directory.

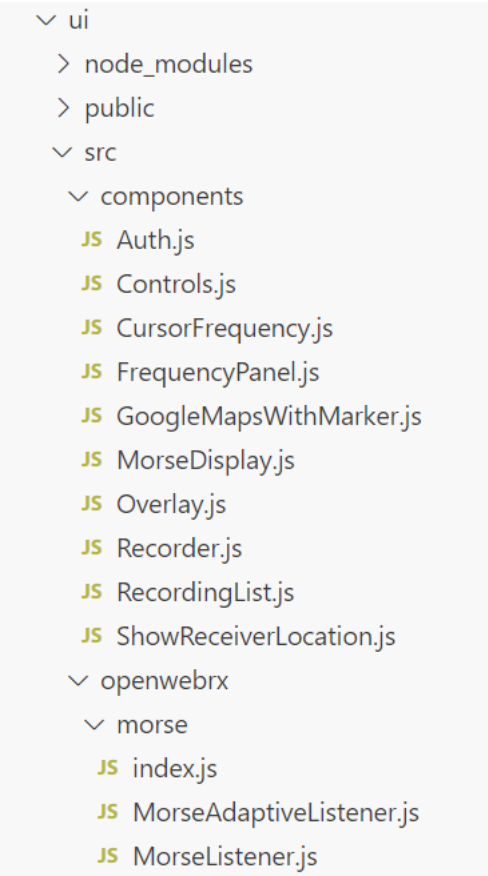

Figure 4.2 UI directory structure.

The front-end provides the code for DRBS to run in a web browser and for an interactive user interface. For efficient, JavaScript fabric full stack execution; React framework was used for frontend development. JavaScript modules are used for authorization, control, frequency, and Google maps panel displays etc.

The structure and content of the webpage are defined by HTTP whilst CSS controls the visual presentation and layout of the elements. A user interface design wireframe is presented in Figure 4.3 is a user interface mock design that was created using an online interface design platform for web applications.

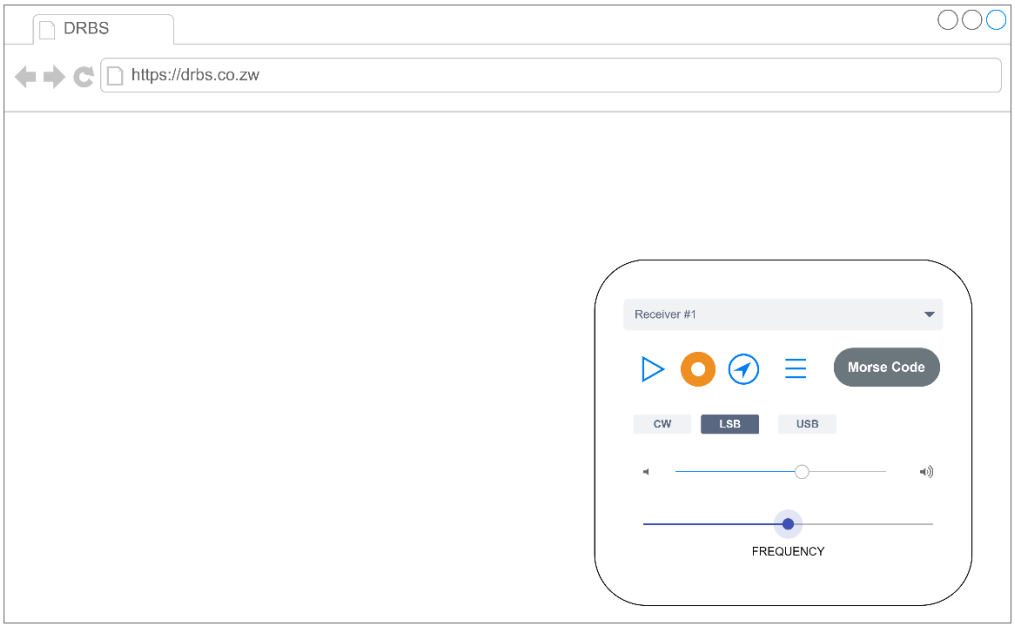

Figure 4.3 DRBS application user interface mock design.

DRBS Backend

The *api* directory provides the core functionality of the server backend. This section summarizes some of the key elements of the backend such as OpenWebRXClient, main.js, receivers and recordings sub directories. Figure 4.4 illustrates the sub-directories and file collections hosted under the api (DRBS backend) directory.

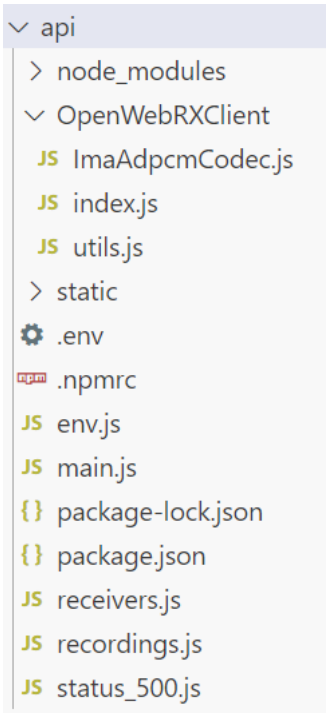

Figure 4.4 API (DRBS backend) directory structure.

OpenWebRXClient is a class that was used to connect to OpenWebRX web application using webSockets protocol. WebSockets will also be employed for internal communication between the frontend and backend.

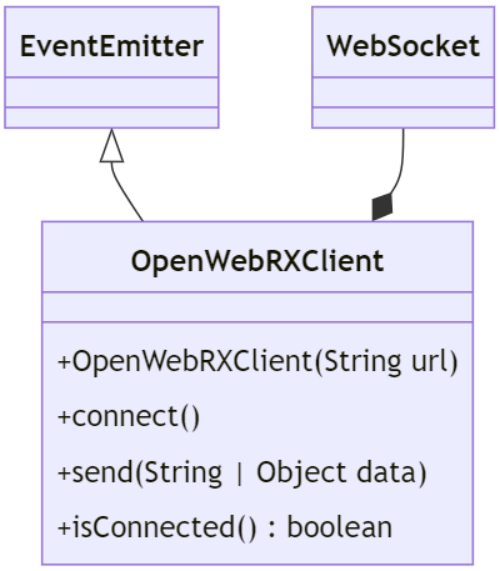

Figure 4.5 OpenWebRXClient class diagram.

main.js module contains all the high-level logic including setting up of the core environment, configuration of middlewares and routes as well as starting the server. The initialization logic includes code for activating express.js which is a web application framework that works with Node.js to simplify https, manage middleware etc. [54]. This involves setting environment variables, configuring external services and databases. It is also used to specify which port to listen on and start the server. Middleware functions are “a stack of functions that have access to the request and response object as well as the next middleware function to call the next middleware function in the stack” [55]. They process incoming requests before reaching the end point. *main.js* file contains executable logic that loads and applies middleware modules globally to the application instance including built-in modules such as *express.json()* and third party middlewares such as morgan. Morgan is a third-party middleware for logging third-party http requests [56]. Routes define the endpoints and map them to specific request handlers e.g., import route files *receivers* and *recordings*. The *app.use ()* object is used to apply the routes. Below is a code snippet of the *main.js* file containing initialization, routes and middleware functions execution logic.

```
require('dotenv').config();
require('./env');

const express = require('express');
const morgan = require('morgan');
const receivers = require('./receivers');
const recordings = require('./recordings');

const app = express();

// middlewares
app.use(express.static(`${__dirname}/static`));
app.use(express.json());

if (process.env.NODE_ENV !== 'test')
   app.use(morgan('tiny'));

// routes
const api = express.Router();
app.use('/api', api);

api.use('/receivers', receivers);
api.use('/recordings', recordings);

// initialization
const PORT = process.env.PORT;

(async () => {
   // start server
   app.listen(PORT, () => {
      if (process.env.NODE_ENV !== 'test')
         console.log("Server started!!!");
   });
})();
```

Figure 4.6 Main.js code snippet.

4.3 API Design

Section 3.3.2 proposes the 5-step model, design, build, test, and document API workflow process. This section will focus on how steps 1 and 2 of the API workflow process were implemented to enable step 3 i.e., building of the DRBS API.

4.3.1 API model

This section provides a high-level description of how the resources and API endpoints interact as well as how the endpoints was mapped into HTTP methods.

Endpoint	Resource	Method	Description
/api/receivers	receivers	GET	Get list of receivers
/api/recordings	recordings	POST	Start recording audio
/api/recordings/:id/stop	recordings	POST	Stop recording audio
/api/recordings/:id/cancel	recordings	POST	Cancel audio recording
/api/recordings	recordings	GET	Fetch audio recording

Figure 4.7 API endpoints to HTTP methods mapping.

4.3.2 Validating the API model

To validate the API model, below sequence diagram illustrates the request-response flow. Three key actors were identified i.e., application User Interface, API and OpenWebRXClient class.

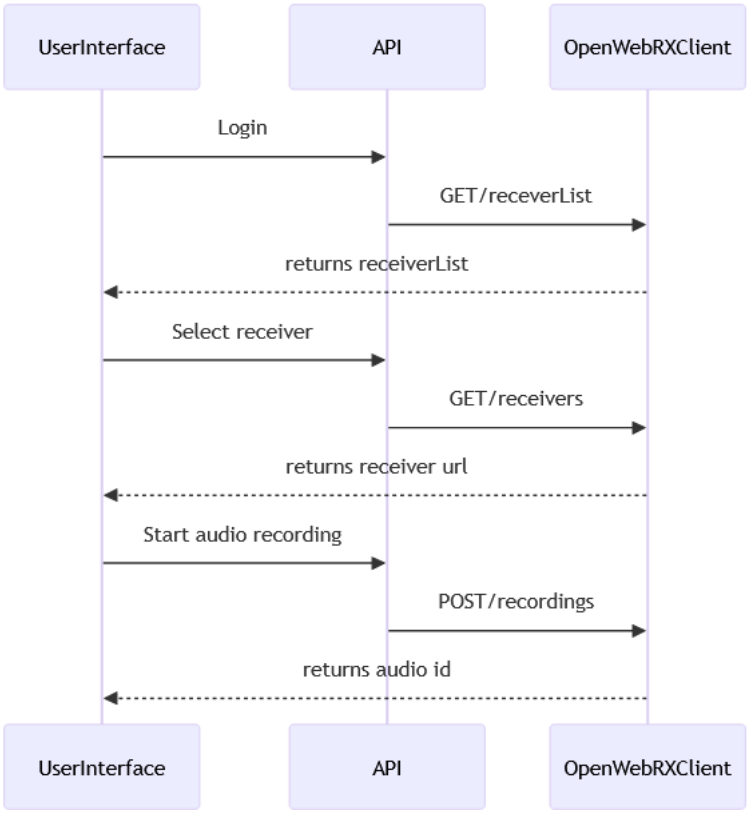

Figure 4.8 Request-response flow sequence diagram.

4.4 Use Cases

The DRBS prototype was developed with the main aim of providing an easy to use, one page web interface for detecting, playing, and recording audio signals from web SDR streaming data services. The sensing, detection and decoding of morse code distress signals was used to validate the system design and test the functionality of DRBS application. Figure 4.9 below summarizes the use cases that will be implemented by the DRBS prototype.

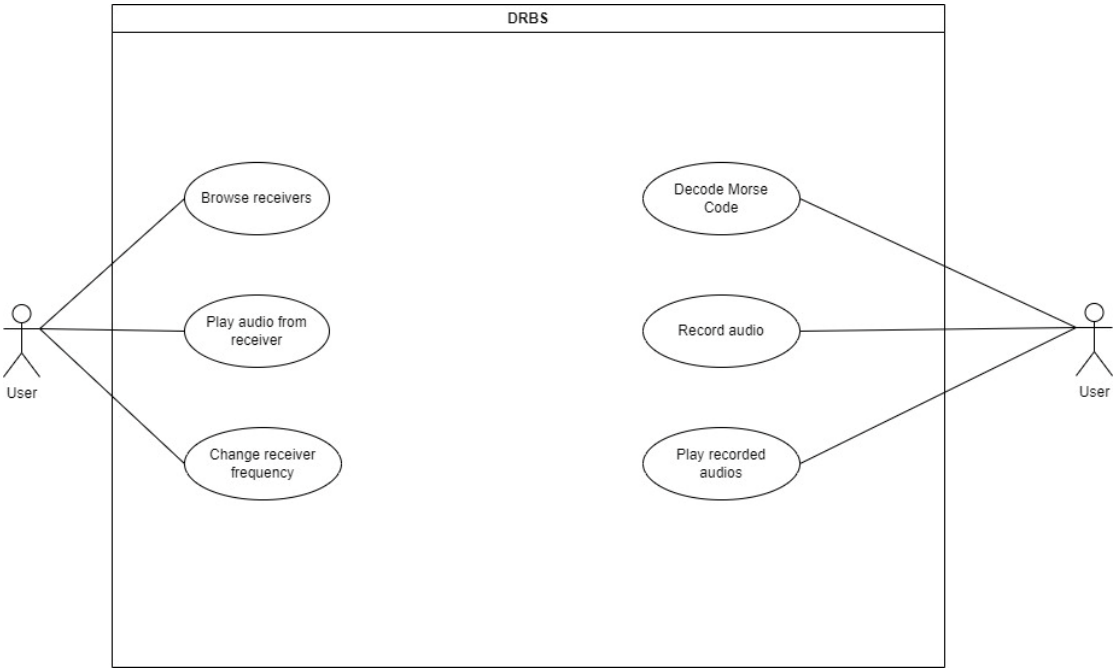

Figure 4.9 DRBS application use cases.

4.5 Experiment Design

To conclude this chapter, this section describes the experiments conducted for testing the functionality and performance of the DRBS application. Three tests were identified in *Section 3.5.2*, namely end to end latency, average server response times and morse code signal detection and decoding experiments.

4.5.1 Functional Testing

Section 4.1 details the requirements specifications and further breaks them down into sub-items. Each requirement was translated into a test case with a defined testing procedure and expected results. In total, 13 test cases were performed with each test case being derived directly from the functionality requirements thus providing a comprehensive assessment of the application's ability to meet design goals. Table 4.1 consolidated the 13 tests into 5 experiments and the results are presented in Section 5.1.

Table 4.1 Experiments performed during functional testing.

Experiment No.	Test ID	Description	Functions Checked	Requirements Tested
1	T1 – T4	Connect to OpenWebRX receiver, receive audio streams and frequency information. Display SDR receiver information.	F1	R1
2	T5	Record, store and play audio.	F2	R2
3	T6	Add local RTL-SDR tuner.	F3	R3
4	T7 – T11	Authentication. Display SDR receivers, waterfall and GPS location of receivers. User's ability to select frequency, modulation and control volume.	F4	R4
5	T12	Scan receivers and search for morse code signal. Display decoded morse code text.	F5	R5

4.5.2 Performance Testing

This section discusses the experiments conducted for testing the performance of the DRBS application under varying conditions. The two tests conducted are namely:

- Experiment 6 – Latency and server response times
- Experiment 7 – Decoding Morse Code

Experiment 6 – Latency and server response times

Experiment 7 evaluated the performance of the application using measurable performance metrics i.e., latency and server response time. Requirement R4.b. from *Section 4.1* specifies that the application's user interface must be user-friendly. To meet this requirement, the application must be consistently responsive to user control

and actions, with minimal user waiting time for responses. Latency, throughput and server response time are critical components for user satisfaction.

For reliability and stability, the DRBS application was hosted in a virtual private cloud (VPC) container. Hosting the application in the cloud also allows the performance of the application to be tested in an isolated environment with dedicated resources. A Linux based virtual machine (VM) was used to host the DRBS application and a network monitoring tool. The VM was configured with 2 virtual CPUs, 8GB memory and 100GB storage. Zabbix network monitoring tool was used to monitor the performance of the application.

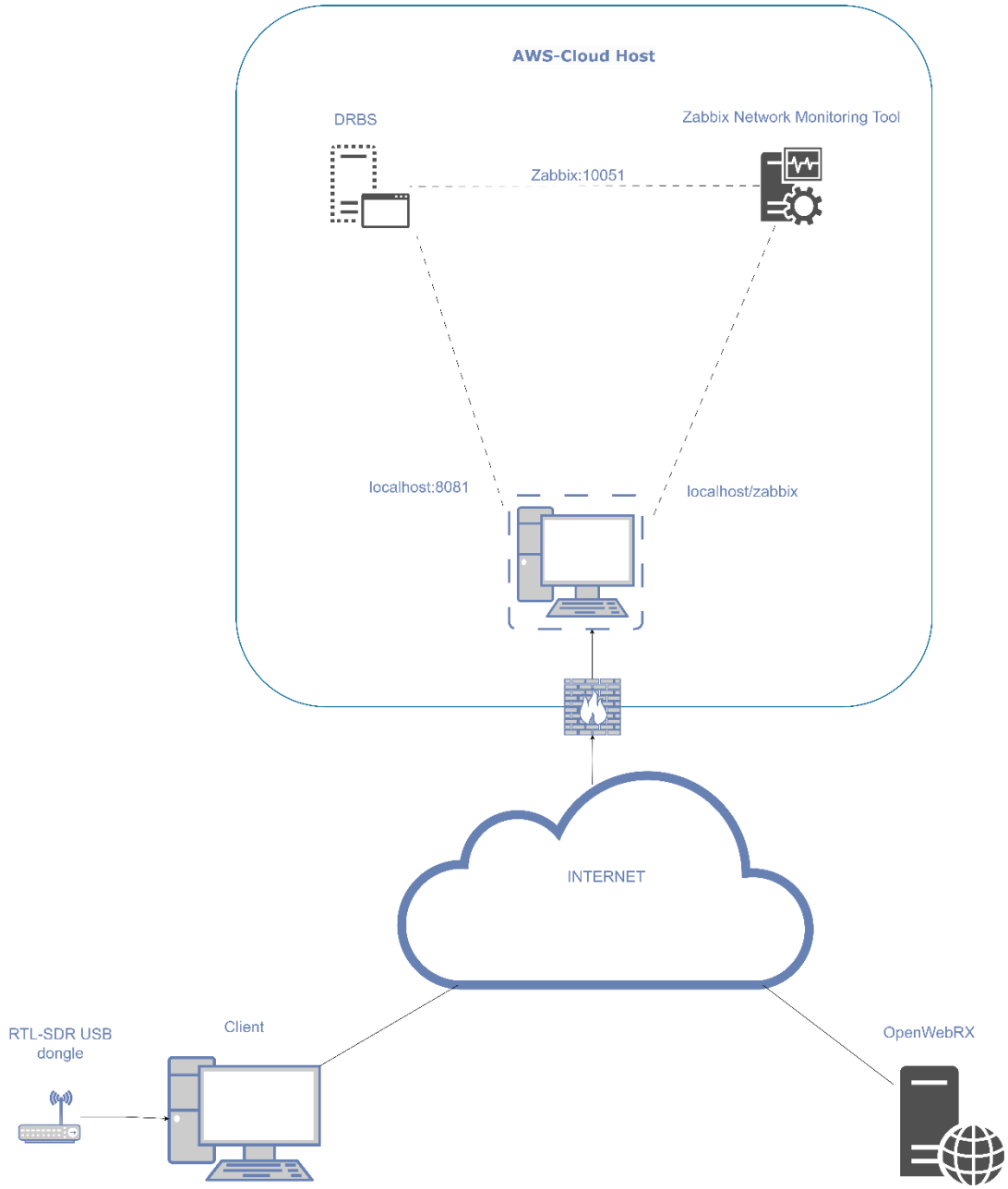

Figure 4.10 Experiment setup for latency and response time in VPC hosted DRBS application.

Test Scenario 1 – Baseline testing

- Setup the experiment as shown in Figure 4.10.
- Stream local radio stations continuously using RTL-SDR receiver whilst continuously monitoring key metrics such as latency and server response times over a 72-hour period.

Test Scenario 2 – Endurance testing

- Setup the experiment as shown in Figure 4.10.
- Use Zabbix network monitoring tool to collect performance logs from DRBS streamed data from the four (4) connected remote OpenWebRX SDR receivers. Continuously monitoring key metrics such as latency and server response times over a 48-day period.
- Plot graphs of the 4 receivers over the 48-day period for the two metrics.

Test Scenario 3 – Stress testing

- Setup the experiment as shown in Figure 4.10.
- Use Postman network monitoring tool to simulate peak load conditions e.g. 10,000 server requests from 10 virtual users.
- Test performance of DRBS API application and plot graphs of the two metrics.

The results for the 3 test scenarios are presented in Section 5.2.1.

Experiment 7 – Decoding Morse Code

In addition, to validate the functionality of DRBS as an application framework allowing diverse use cases to be implemented whilst re-using the underlying core modules of the application. An experiment on the use case was therefore conducted to evaluate the applications performance when executing a specific task, in this case; decoding morse code. The experiment goal was to assess how well the decoder translates morse code audio signal input into the correct text output.

In the experiment setup shown in Figure 4.11, a low-cost RTL-SDR dongle connected to a small whip antenna was used to pick up FM broadcast signals transmitted by the FM modulator. The FM modulator transmits recorded audio signals from an audio player such as a smartphone's mp3 or m4a player. The smartphone is connected to the FM modulator via Bluetooth.

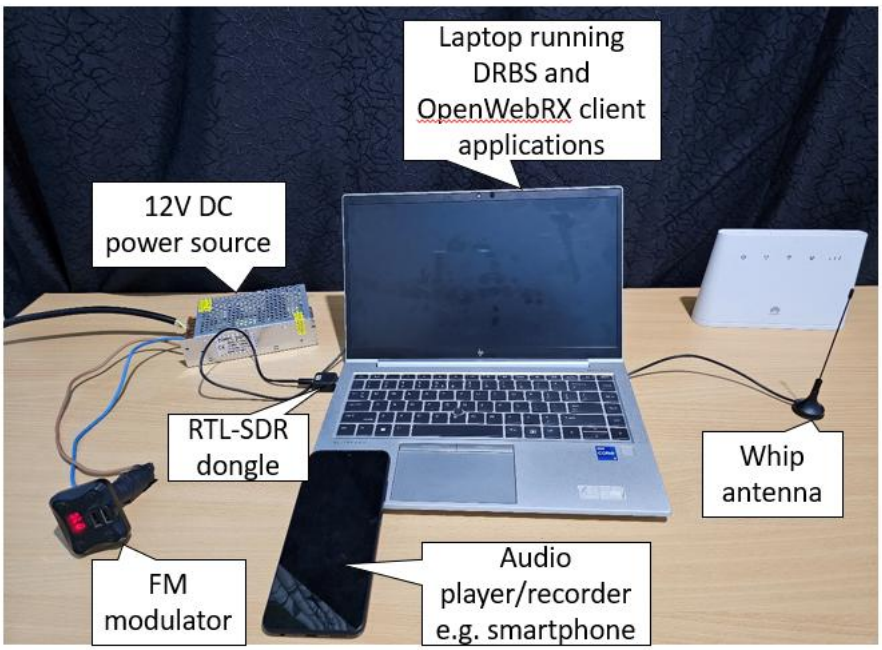

Figure 4.11 Local RTL-SDR experiment setup for morse code detection and decoding tests.

For morse code signal detection and decoding tests, the experiment was conducted as described below.

- The experiment was set up as described in Figure 4.11.
- Morse code audio signals were emulated from 3rd party online recordings on a normal smartphone and broadcast by a low-cost FM transmitting device. An RTL-SDR receiver was used to receive the audio signals and send them to the DRBS application for decoding.
- Perform *Baseline Test* – Generate test sample of morse code audio signal with known outputs. Capture decoded text and measure accuracy.
- Conduct *Stress Test* – Generate test sample with varied transmission rates i.e., lengths of dots and dashes. Capture decoded text and measure accuracy.
- Draw table of results showing decoding accuracy of the baseline test sample and the distorted sample. The results are presented in Section 5.2.2.

5 Results and Discussion

This chapter summarizes the functional and performance results of the experiments conducted in Section 4.5.

5.1 Functional Testing Results

With reference to the testing methodology discussed in Section 3.5.1, Section 4.1 developed a testing plan which involved 13 test cases. These 13 test cases were consolidated into 5 experiments as consolidated in Table 4.1. The results of the 5 experiments conducted for functional testing are summarized below.

Experiment 1 was based on requirement R1 for testing the establishment and maintenance of a connection to OpenWebRX, pulling FFT and audio streams from OpenWebRX receivers. DRBS illustrated that it could successfully connect to OpenWebRX via the API and stream data. This is evidenced by the screen capture below of DRBS streaming data from an OpenWebRX receiver on a public IP address showing FFT streams pulled from the receiver.

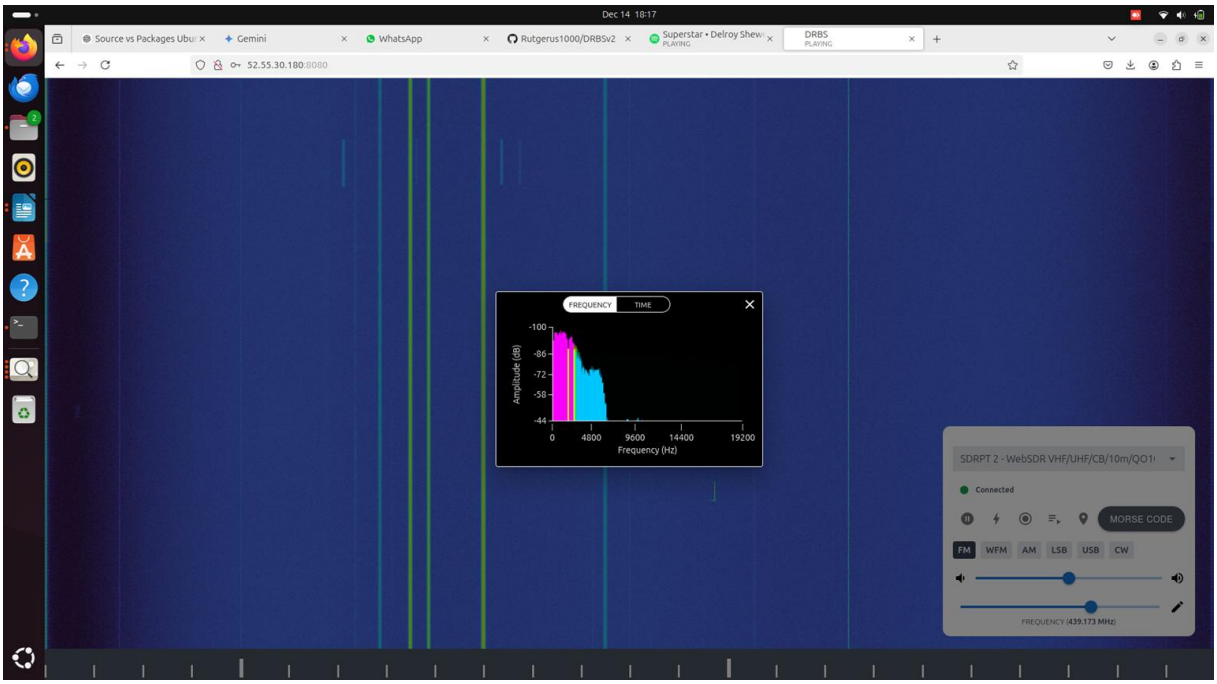

Figure 5.1 DRBS FFT visualizer with waterfall plot of OpenWebRX receiver data stream in the background.

Experiment 2 was based on requirements R2 and Function F2 which specified audio file storage capability. The experiment tested UI record button, DRBS communication API ability to communicate with the backend for storing of audio recordings and fetch function for audio retrieval. The test result was successful.

Experiment 3 tested requirement R3 i.e., support local RTL-SDR tuners. A local RTL-SDR USB dongle was successfully added to DRBS receiver list as illustrated in Figure 5.2 below.

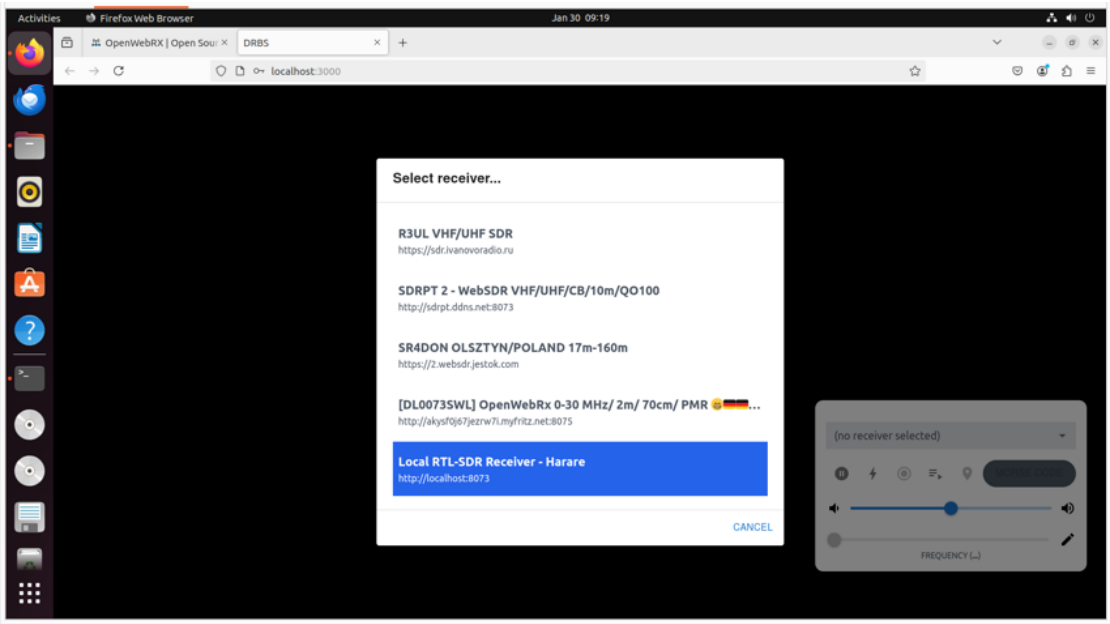

Figure 5.2 DRBS receiver list showing Local RTL-SDR Receiver hosted locally.

Experiment 4 tested DRBS features as presented by the user interface. These are represented by requirement R4a on the requirements list presented in Section 4.1. R4.a. states that the system must be able to do basic user authentication on login. Figure 5.3 below shows DRBS login page requesting user credentials.

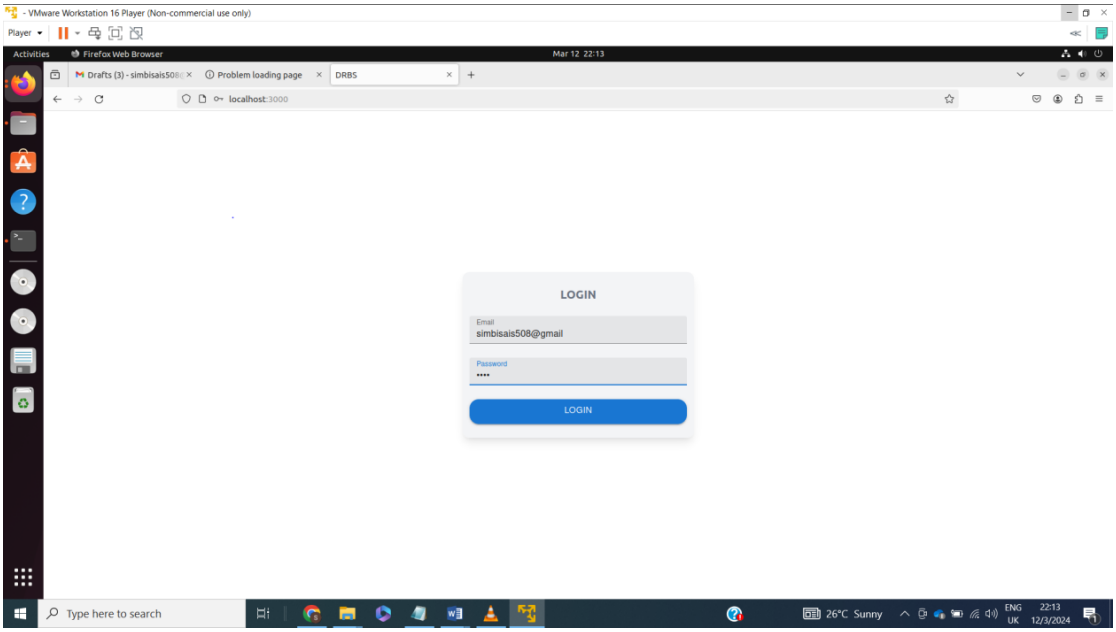

Figure 5.3 DRBS user login page.

Samples of other test results conducted in Experiment 4 such as R 4.c. which required the application to display information about SDR receivers pulled from OpenWebRX, showing a real time OpenWebRX-like waterfall as well as GPS location of receivers. The location feature of DRBS was tested successfully and is shown in Figure 5.4 below.

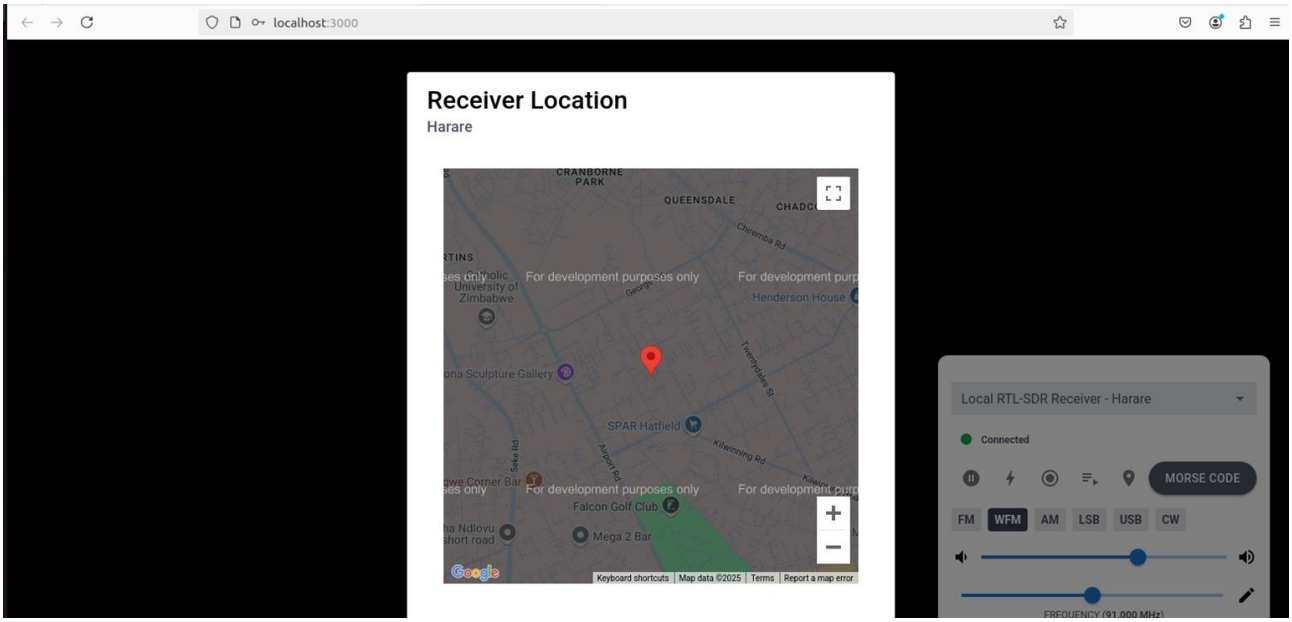

Figure 5.4 DRBS application GPS location feature.

Other successfully tested features included center frequency, modulation and audio volume control.

Of particular importance was **Experiment 5** which tested the ability of the application to scan across multiple feeds, attempted to search for morse code signal and display decoded morse code text. The application was able to manually scan successfully, detect morse code and convert morse code audio signals into text as displayed in Figure 5.5.

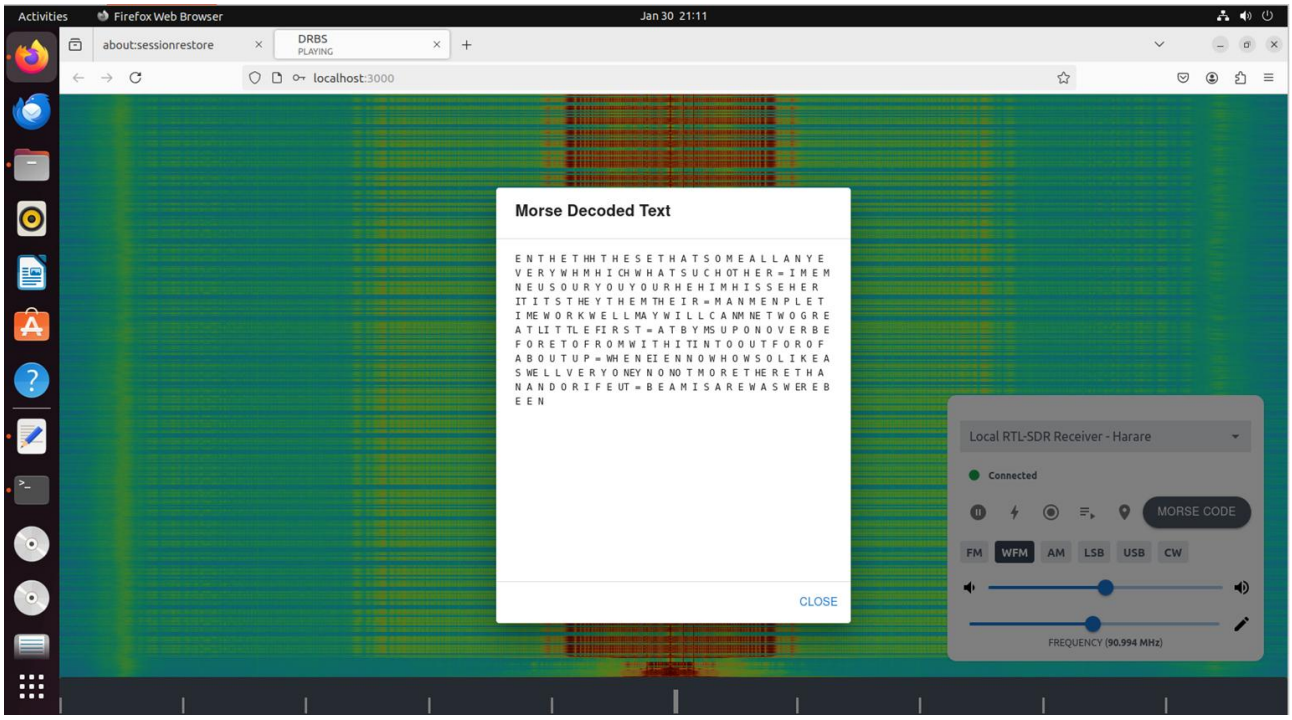

Figure 5.5 DRBS UI displaying decoded morse code text.

As illustrated in experiments 1 to 4, the DRBS application performed all the functions and generally met the needs identified in the design phase. In particular, the application

provided a wide range of features testament to the “openness” of OpenWebRX API. Thus, development proceeded towards a more integrated experiment-based testing of the system in addition to the modular tests presented above.

5.2 Performance Testing

This section summarizes the results of the experiments conducted for end-to-end performance testing of the DRBS application.

5.2.1 Experiment 6 – Latency and server response times

DRBS application was cloud hosted on a Linux based virtual machine (VM). Zabbix and Postman network monitoring tools were used to monitor the performance of the application. As per experiment setup described in Section 4.5.2, three tests were conducted, and the results are as follows:

- Test 1: Baseline testing
  Baseline test where DRBS streamed local radio stations using RTL-SDR receiver whilst continuously monitoring key metrics such as latency and server response times over a 72-hour period. The results are presented in Figures 5.6 and 5.7.
- Test 2: Endurance testing
  The four (4) connected remote OpenWebRX SDR receivers were then endurance tested (persistent streaming) with Zabbix network monitoring tool collecting performance logs over a 48-day period. The network monitoring tool automatically generated graphs of the 4 receivers over the period for the two metrics. The results are presented in Figures 5.8 and 5.9.
- Test 3: Stress testing
  DRBS was stress tested under peak load conditions; in this case 100 virtual users were simulated generating over 17,000 requests in one minute. The results are presented in Figure 5.10.

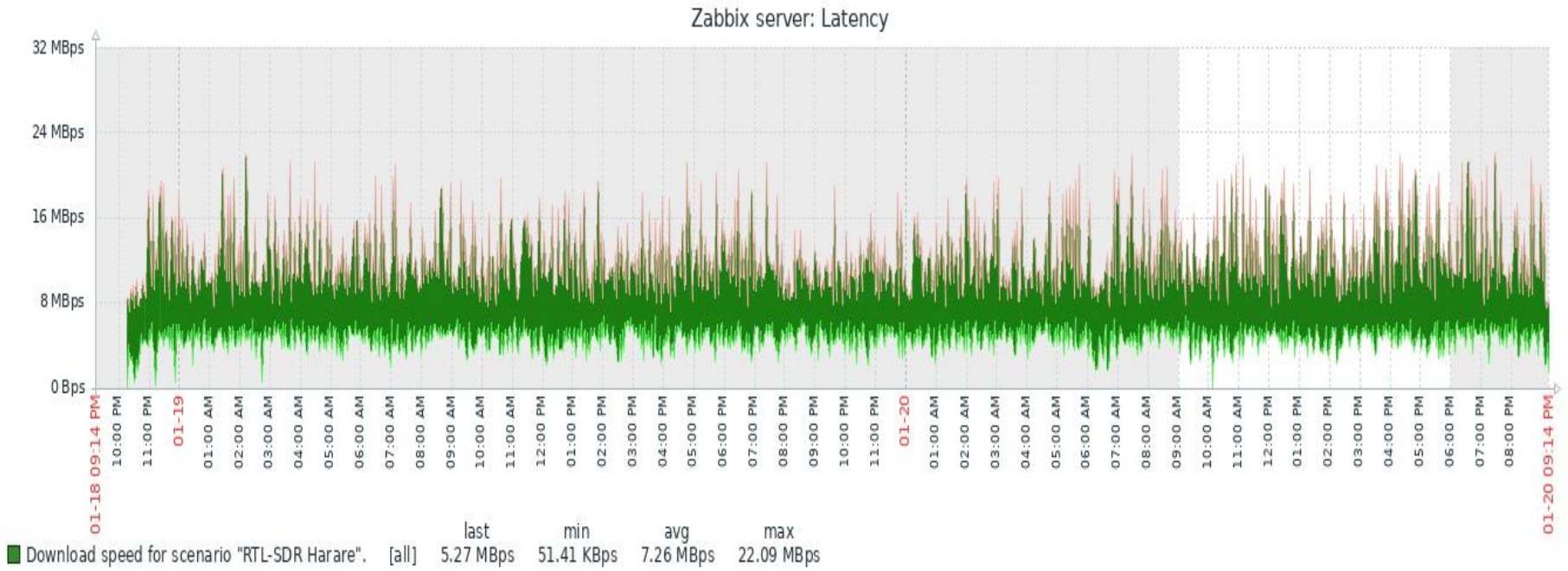

Figure 5.6 Latency graph for the local SDR receiver used for baseline values.

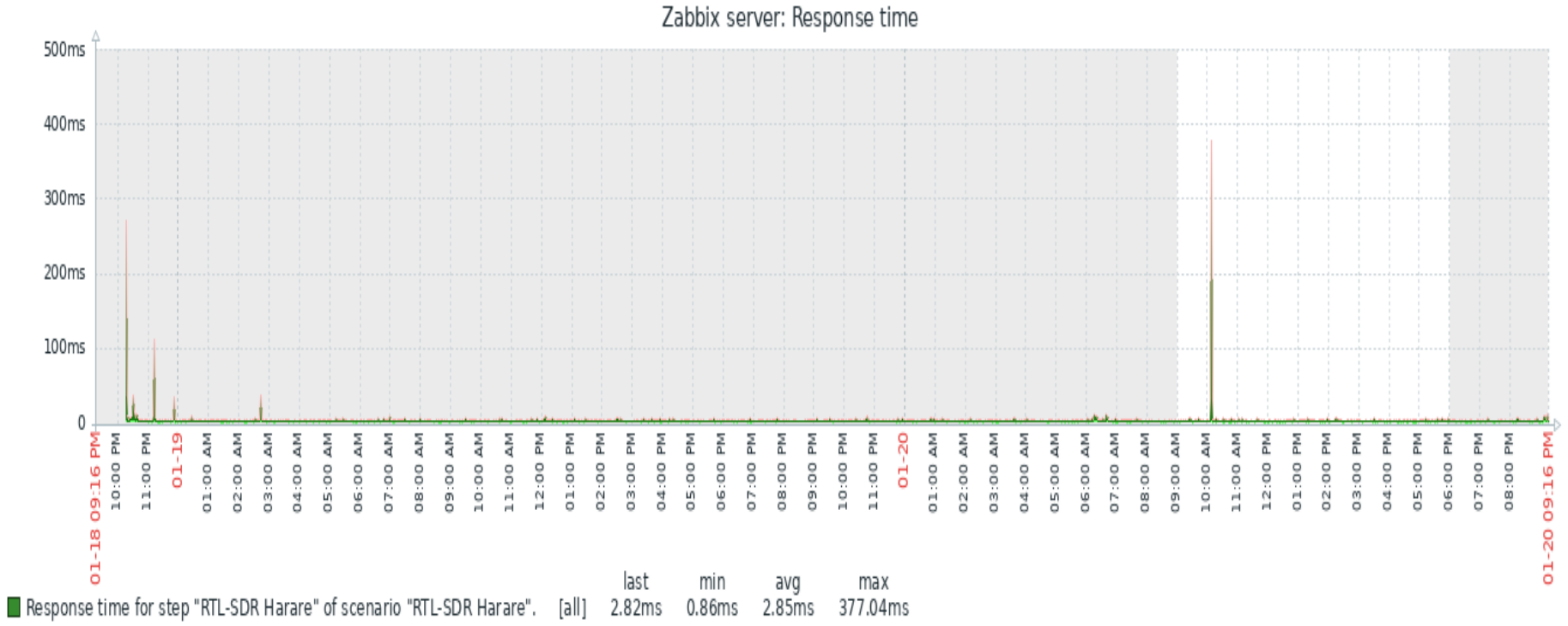

Figure 5.7 Server response time for the local SDR receiver giving baseline values.

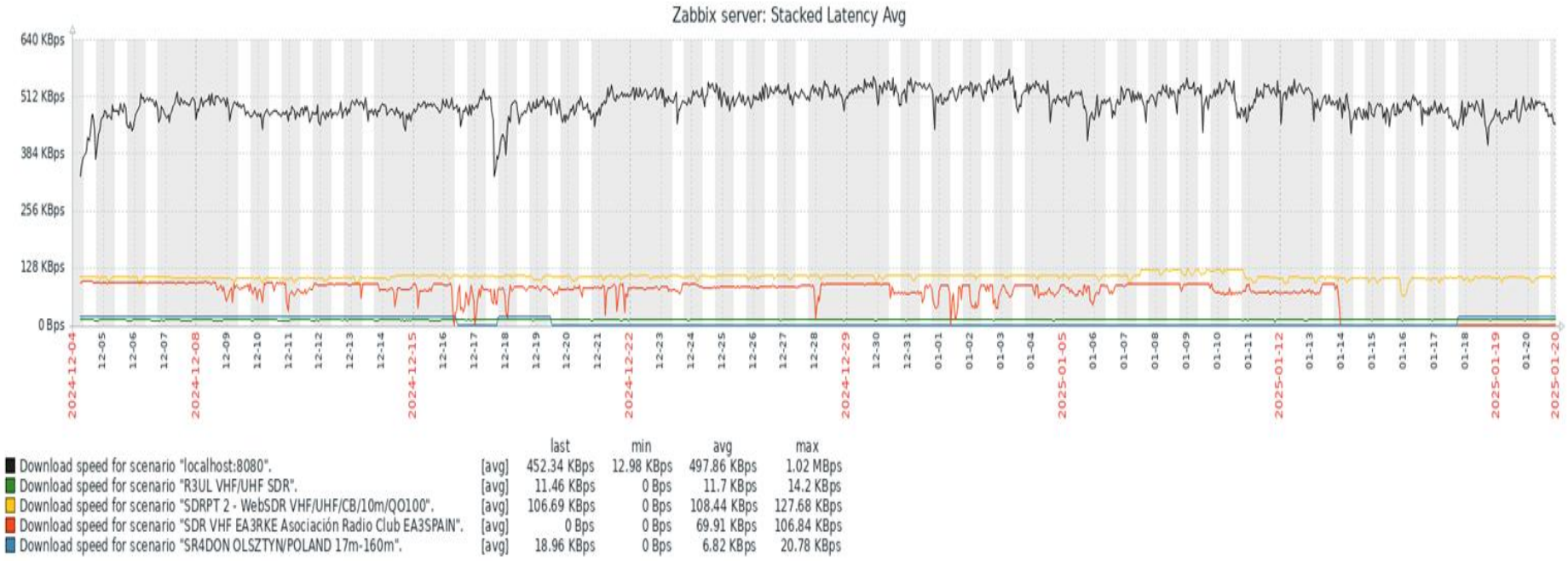

Figure 5.9 Stacked server response time graphs for the 4 receivers.

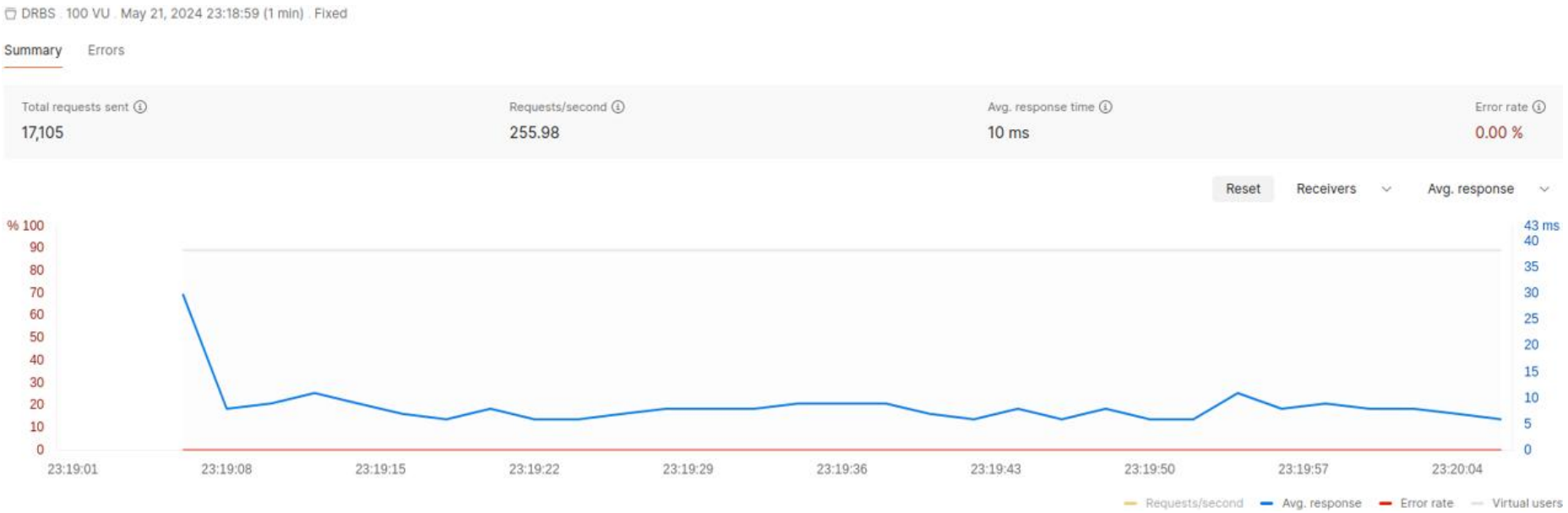

Figure 5.10 DRBS average server response times in peak load conditions.

Experiment 6 Analysis

It can be noted from the baseline tests i.e., Figures 5.6 and 5.7, the average latency is measured as 7.2 Mbps. Zabbix monitoring tool reports latency in bits per second i.e., throughput. Whilst latency measures the time it takes a bit to travel from point A to B, throughput on other hand measures the number of bits transmitted from A to B. High

throughput generally means a lot of data is being transferred from A to B at a fast rate (low latency). An average server response time of 2.85ms was also measured.

Figure 5.8 and 5.9 illustrates the two graphs for stacked latency and stacked server response time. The results from the 4 receivers added in DRBS receiver list were stacked on the same graph for ease of comparison. SR4DON OLSZTYN/POLAND had the highest implied latency (6.82 kbps throughput) whilst server response times measured between 300 and 500ms.

The results of the stress tests done locally on DRBS API using Postman which simulated 100 virtual users sending 17,105 requests in a 1-minute interval. Figure 5.10 shows as average server response time of 10ms. The result shows a marked difference between the server response times of 300-500ms being measured by Zabbix over the network connecting to remote servers. This test shows that on average DRBS API is contributing 10ms to the total time it takes a remotely hosted server to respond. Other factors that may contribute to this huge time delay include network related issues such as distance and traffic congestion as well as server-side processing

5.2.2 Experiment 7 – Decoding Morse Code

The experiment was setup as depicted in Figure 4.11. A simple test called the “100 most common English words” test [57] which measured the decoder’s ability to identify 108 words sent with varying transmission rates as follows:

12 Words per Minute (WPM), 18 WPM and 28 WPM. At least ten tests were done for each transmission speed and averaged out to produce three sets of results. The metric used for the evaluation was:

- Word Accuracy: (Number of Correct Words / Total Words) * 100

In all three tests, it was noted that the decoder ended up with more words (254) than the 108 words transmitted. Two scenarios were considered for reporting:

i) Word $\text{accuracy}_{\text{actual}}$ based on the number of words correctly decoded from the actual test set (108 words), ignoring the extra generated words.

ii) Word accuracy$_{extended}$ based on the number of words correctly decoded out of the total words produced by the decoder, including the extra words (254 words).

Test 1 – At 12 WPM:

i) Word accuracy$_{actual}$ = 80 / 108 x 100 = 74.4%

ii) Word accuracy$_{extended}$ = 80 / 254 x 100 = 31.4%

Test 2 – At 18 WPM:

i) Word accuracy$_{actual}$ = 68 / 108 x 100 = 62.9%

ii) Word accuracy$_{extended}$ = 68 / 263 x 100 = 25.8%

Test 3 – At 28 WPM:

i) Word accuracy$_{actual}$ = 59 / 108 x 100 = 54.6%

ii) Word accuracy$_{extended}$ = 59 / 360 x 100 = 16.3%

Table 5.1 Summary of experiment 7 results.

Test No.	Transmission speed	Number of Correct Words	Number of Extra Words	Word accuracy$_{actual}$	Word accuracy$_{extended}$
1.	12 WPM	80	146	74.4%	31.4%
2.	18 WPM	68	155	62.9%	25.8%
3.	28 WPM	59	252	54.6%	16.3%

Experiment 7 Analysis

The results as summarized by Table 5.1 show that whilst the word accuracy based on the actual set of words transmitted is encouraging i.e., the number of words that could be identified is quite high as illustrated by the high percentage word accuracy$_{actual}$ of 83.3%, 72.2 and 63.8 for 12, 18 and 28 WPM respectively. The result however does not reflect the lack of meaning of decoded text due to the extra words. Word

$accuracy_{extended}$ penalises the decoder for generating extra words and generally provides a better reflection of the performance of the decoder from a word accuracy perspective.

5.3 Discussion of Results

The general findings from the experiment results are that:

- DRBS API is a lightweight and lean software application,
- DRBS application had the minimum functionality required for a web SDR broadcast streaming tool,
- DRBS could detect and decode morse code signals albeit with low audio to text conversion accuracy.

The discussion below concerns some of the limitations or unexpected results and their implications for the project.

5.3.1 Experiment 6 – Latency and server response times

The stress test results (Figure 5.10) were particularly interesting because it means the overall contribution to the total time delay in normal operating conditions is much less than the average 10ms measured from over 17,000 requests from 100 users, all in 1 minute. It also highlights some potential limitations of the testing methodology and consequently the importance of the results. Since the API is hosted locally, and the tests were run on the local machine or VPC equivalent, the lack of network effects and other real-world factors will often result in consistently very low server response times. Using Postman for API testing is also known for resulting in low response times due to Postman serving responses from saved caches. Clearing Postman cache after each response is not realistic if you're sending over 17,000 requests. However, the results bode well for application responsiveness and user satisfaction.

5.3.2 Experiment 7 – Decoding Morse Code

As expected, the accuracy of the decoder decreases as the speed of sending and receiving increases. The decoder has less and less time to process the dots and dashes thus increasing the chance of error. Notwithstanding, the researcher acknowledges the lack of morse code decoder accuracy as alluded in Section 5.2.2 Experiment 2 Analysis. However, Section 4.1, requirement R5. states that "The interface must display decoded morse code text" to meet the basic needs of the

application in decoding emergency and distress signals which typically consists of a few characters such “SOS” or “Mayday”. The low accuracy may suffice for the intended application.

More work needs to be done to identify the specific conditions leading to errors (e.g., long dashes) recording where the decoder fails. Other issues which were apparent during testing that may have contributed to the low accuracy is the low audio signal quality that was being transmitted by the FM modulator and the low SNR at the RTL-SDR receiver. Both are low cost, low specification devices. Enhance filtering is required to better isolate the Morse code signal from noise.

6 Conclusions and Future Work

This chapter presents the culmination of a rigorous research investigation, from its initial conceptualization to the presentation of the key findings. As meticulously documented in the results and conclusions chapter, the extensive testing and experimentation undertaken for this project have yielded demonstrably successful outcomes. This research has not only addressed but has also effectively delivered on:

a) Developing and implementing DRBS application; an open source, web broadcast streaming tool that provides the ability to cycle across SDR receivers and automatically search for specific data.

b) Developing and documenting a customizable model and framework specification which can be reused by the research and developer community for SDR based streamed data applications.

Based on the experiments and results obtained in Chapter 6, some conclusions can be drawn in relation to the thesis objectives presented above. DRBS application, an open source, web broadcast streaming tool pulling data from hosted SDR receivers was successfully developed as verified by the functional test results presented in Chapter 5.1. Morse code signal detection and decoding application was implemented to test the ability to search for specific data.

Design approaches i.e., open source, modular, non-blocking and asynchronous architecture proposed in Chapter 3, were used to develop an application framework as envisaged by Objective b). DRBS contribution to the developer community as a devkit include:

- code libraries for base functionality such as authorization, waveform controls, audio engine, receiver location etc.
- development environment,
- deployment model based on Node.js application framework, Express for backend and React for frontend development,
- OpenWebRX API providing code structure for integration with 3rd party web SDR receiver hosting platforms as well as communications API for internal communications,

- testing methodology and tools such as Zabbix network monitoring tool and Postman,
- documentation in the form of a README file to provide installation instructions, Git source control for code base versioning and a getting started guide in the form of a tutorial.

Despite the morse code decoder accuracy concerns reported in Section 5.3, the morse code use case validates as a POC, that DRBS can provide a software development kit with pre-built modules, code libraries and components for basic functionality, for developers to create software applications quickly and save on development time as well as simplifying installation and debugging.

Future Work

It was evident from the morse code POC results that more work needs to be done in future iterations to improve the detection accuracy of the decoder. DRBS could be a vital tool in monitoring and keeping watch for emergency transmissions including distress calls, weather alerts or other critical broadcasts. The DRBS application being inherently able to monitor FM spectrum, additional use cases such as identifying FM radio stations based on music genre could also be considered for future iterations. This could be enhanced by involving a machine learning model to classify audio streams based on musical characteristics. Another aspect worth pursuing is analysing spectrum across different geographic regions to provide insights into how radio frequencies are being utilized. This could potentially be used to detect shortages on FM broadcasts. DRBS could also be useful as a web-based spectrum monitoring platform that relies on crowdsourcing.

Future work is not just required to improve the DRBS application and get better results. To build a robust and versatile application framework, components must be developed to provide base functionality across multiple platforms, devices, languages and Operating Systems.

7 References

[1] M. Reiser, “The Oberon System: User Guide and Programmer’s Manual - OberonCore.” Accessed: Feb. 13, 2025. [Online]. Available: https://oberoncore.ru/library/reiser_the_oberon_system_user_guide_and_programmers_manual

[2] N. Wirth, “A Plea for Lean Software,” *Computer (Long Beach Calif)*, vol. 28, no. 2, pp. 64–68, 1995, doi: 10.1109/2.348001.

[3] “Code Complete: A Practical Handbook of Software Construction.” Accessed: Feb. 13, 2025. [Online]. Available: https://www.researchgate.net/publication/319770462

[4] “Software Defined Radio: Past, Present, and Future - NI.” Accessed: Feb. 13, 2025. [Online]. Available: https://www.ni.com/en/perspectives/software-defined-radio-past-present-future.html

[5] R. Akeela and B. Dezfouli, “Software-defined Radios: Architecture, State-of-the-art, and Challenges,” *Comput Commun*, vol. 128, pp. 106–125, Apr. 2018, doi: 10.1016/j.comcom.2018.07.012.

[6] R. Zitouni, H. Bouaroua, and B. Senouci, “Hardware-Software Codesign for Software Defined Radio: IEEE 802.11p receiver case study,” Mar. 2020, Accessed: Feb. 13, 2025. [Online]. Available: https://arxiv.org/abs/2003.09525v2

[7] “Software Defined Radio: Past, Present, and Future - NI.” Accessed: Feb. 13, 2025. [Online]. Available: https://www.ni.com/en/perspectives/software-defined-radio-past-present-future.html

[8] SDR Forum, “SDRF Cognitive Definitions,” 2007. Accessed: Aug. 17, 2024. [Online]. Available: http://www.sdrforum.org/pages/documentLibrary/documents/SDRF-06-R0011-V1_0_0.pdf

[9] C. R. Johnson, W. A. Sethares, and A. G. Klein, *Software Receiver Design Build Your Own Digital Communications System in Five Easy Steps*. 2011.

[10] M. Dillinger, K. Madani, and N. Alonistioti, *Software Defined Radio Architectures, Systems and Functions*. 2003.

[11] T. Ulversoy, “Software defined radio: Challenges and opportunities,” *IEEE Communications Surveys and Tutorials*, vol. 12, no. 4, pp. 531–550, Dec. 2010, doi: 10.1109/SURV.2010.032910.00019.

[12] E. Grayver, *Implementing software defined radio*, vol. 9781441993328. Springer New York, 2013. doi: 10.1007/978-1-4419-9332-8.

[13] W. H. W. Tuttlebee, *Software Defined Radio: Enabling Technologies*. Wiley, 2003.

[14] R. Akeela and B. Dezfouli, “Software-defined Radios: Architecture, state-of-the-art, and challenges,” Sep. 01, 2018, *Elsevier B.V.* doi: 10.1016/j.comcom.2018.07.012.

[15] Xilinx and Inc, “An Adaptable Direct RF Sampling Solution (WP489),” 2019. [Online]. Available: www.xilinx.com

[16] T. Juhana, “Web-based FM Broadcasting Monitoring System,” in *2016 International Symposium on Electronics and Smart Devices (ISESD): November 29-30, 2016, Bandung, Indonesia*, IEEE, 2016, pp. 311–314. Accessed: Aug. 17, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/7886740/

[17] “OpenWebRX web-based software defined radio.” Accessed: Feb. 13, 2025. [Online]. Available: https://www.openwebrx.de/

[18] “About ShinySDR.” Accessed: Dec. 05, 2024. [Online]. Available: https://shinysdr.switchb.org/

[19] “WebRadio.fm - Christmas Internet Radio - Christmas Music.” Accessed: Feb. 13, 2025. [Online]. Available: https://webradio.fm/

[20] “About RTL-SDR,” rtl-sdr.com. Accessed: Aug. 17, 2024. [Online]. Available: https://www.rtl-sdr.com/about-rtl-sdr/

[21] K. K. Angelov, S. M. Sadinov, and P. G. Kogias, “Development of a Simple SDR-based System for Monitoring of VHF and UHF Radio Frequency Bands,” in *2020*

XXIX International Scientific Conference Electronics - ET2020, IEEE, 2020, pp. 1–4.

[22] V. V. Moshkov, R. V. Glazkov, and N. V. Babaev, “Web SDR-based QoS Monitoring System for Wireless Networks Analysis,” in *Proceedings of the 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus): January 27-30, 2020, St. Petersburg and Moscow, Russia*, St. Petersburg Electrotechnical University “LETI,” 2020, pp. 42–44. Accessed: Aug. 17, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/9039293/

[23] T. Juhana and S. Girianto, “An SDR-Based Multistation FM Broadcasting Monitoring System,” in *TSSA: 2017 11th International Conference on Telecommunication Systems Services and Applications: 26-27 October 2017*, IEEE, 2017, pp. 162–165. Accessed: Aug. 17, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/8272943/

[24] B. Uengtrakul and D. Bunnjaweht, “A Cost-Efficient Software Defined Radio Receiver for Demonstrating Concepts in Communication and Signal Processing using Python and RTL-SDR,” in *2014 Fourth International Conference on Digital Information and Communication Technology and its Applications*, IEEE, 2014, pp. 394–399.

[25] W.-T. Chen, K.-T. Chang, and C.-P. Ko, “Spectrum monitoring for wireless TV and FM broadcast using software-defined radio,” *Multimed Tools Appl*, vol. 75, no. 16, pp. 9819–9836, 2016, Accessed: Aug. 17, 2024. [Online]. Available: https://link.springer.com/content/pdf/10.1007/s11042-015-2764-5.pdf

[26] R. Calvo-Palomino *et al.*, “Electrosense+: Empowering People to Decode the Radio Spectrum.” [Online]. Available: https://www.researchgate.net/publication/329305354

[27] M. Robert, Y. Sun, T. Goodwin, H. Turner, J. H. Reed, and J. White, “Software Frameworks for SDR,” *Proceedings of the IEEE*, vol. 103, no. 3, pp. 452–475, 2015, doi: 10.1109/JPROC.2015.2391176.

[28] “About the CORBA Component Model Specification Version 3.0.” Accessed: Feb. 12, 2025. [Online]. Available: https://www.omg.org/spec/CCM/3.0/About-CCM

[29] “OSGi Working Group | The Eclipse Foundation.” Accessed: Feb. 12, 2025. [Online]. Available: https://www.osgi.org/

[30] S. Kim, J. Masse, S. Hong, and N. Chang, “SCA-based component framework for software defined radio,” *Proceedings - IEEE Workshop on Software Technologies for Future Embedded Systems, WSTFES 2003*, pp. 3–6, 2003, doi: 10.1109/WSTFES.2003.1201349.

[31] C. R. Aguayo González, C. B. Dietrich, and J. H. Reed, “Understanding the software communications architecture,” *IEEE Communications Magazine*, vol. 47, no. 9, pp. 50–57, Sep. 2009, doi: 10.1109/MCOM.2009.5277455.

[32] “REDHAWK.” Accessed: Feb. 12, 2025. [Online]. Available: https://redhawksdr.org/

[33] S. Taya and S. Gupta, “Comparative Analysis of Software Development Life Cycle Models,” *INTERNATIONAL JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY (IJCST)*, vol. 2, no. 4, pp. 536–539, Dec. 2011, [Online]. Available: www.ijcst.com

[34] A. Alshamrani and A. Bahattab, “A Comparison Between Three SDLC Models,” *International Journal of Computer Science Issues (IJCSI)*, vol. 12, no. 1, pp. 106–111, Jan. 2015, Accessed: Oct. 30, 2024. [Online]. Available: https://www.proquest.com/docview/1660801422

[35] A. Mujumdar, P. M. Chawan, and G. Masiwal, “Analysis of various Software Process Models,” *International Journal of Engineering Research and Applications (IJERA)*, vol. 2, no. 3, pp. 2015–2021, Jun. 2012, Accessed: Nov. 02, 2024. [Online]. Available: https://www.researchgate.net/publication/316510707

[36] A. Govardhan, “A Comparison Between Five Models of Software Engineering,” *International Journal of Computer Science Issues*, vol. 5, no. 7, pp. 94–101,

Sep. 2010, Accessed: Nov. 02, 2024. [Online]. Available: https://www.researchgate.net/publication/258959806

[37] B. W. Boehm, “A spiral model of software development and enhancement,” *Computer (Long Beach Calif)*, vol. 21, no. 5, pp. 61–72, Aug. 2002, Accessed: Oct. 30, 2024. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=59

[38] S. Koolmanojwong, B. Boehm, and J. A. Lane, “The Incremental Commitment Spiral Model for Service-Intensive Projects,” pp. 94–115, Sep. 2012, doi: 10.4018/978-1-4666-2503-7.CH005.

[39] A. Retzler and P. Horváth, “Software Defined Radio Receiver Application with Web-based Interface,” 2014. Accessed: Aug. 28, 2024. [Online]. Available: https://sdr.hu/static/bsc-thesis.pdf

[40] H. Suzuki, Y. Kawakita, and H. Ichikawa, “Remote implementation of GNU radio-based SDR development environment,” in *Proceedings - Asia-Pacific Conference on Communications, APCC 2016*, Institute of Electrical and Electronics Engineers Inc., Oct. 2016, pp. 355–360. doi: 10.1109/APCC.2016.7581497.

[41] JSON-RPC Working Group, “JSON-RPC 2.0 Specification.” Accessed: Aug. 28, 2024. [Online]. Available: https://www.jsonrpc.org/specification

[42] Ondřej Lysoněk, “Web SDR receiver,” 2017. Accessed: Aug. 28, 2024. [Online]. Available: https://is.muni.cz/th/kp8qb/thesis.pdf

[43] A. Green, “libwebsockets.org.” Accessed: Aug. 28, 2024. [Online]. Available: https://libwebsockets.org/

[44] “Introduction to using the KiwiSDR.” Accessed: Dec. 05, 2024. [Online]. Available: http://kiwisdr.com/ks/using_Kiwi.html

[45] GoalKickercom, “Node.js Notes for Professionals.” Accessed: Aug. 28, 2024. [Online]. Available: https://nvkarta.com/project/library/uploads/engineering/programming/Node%20JS%20Notes%20For%20Professionals.pdf

[46] J. Higginbotham, *Principles of Web API Design*. Pearson Education, Inc., 2022. Accessed: Aug. 28, 2024. [Online]. Available: https://ptgmedia.pearsoncmg.com/images/9780137355631/samplepages/9780137355631_Sample.pdf

[47] Open-Source Initiative, “The Open-Source Definition,” https://opensource.org/osd/. Accessed: Aug. 28, 2024. [Online]. Available: https://opensource.org/osd

[48] Internet Engineering Task Force (IETF), “WebSocket Protocol,” 2011. Accessed: Aug. 28, 2024. [Online]. Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc6455.txt.pdf

[49] J. Ketterl, “API,” GitHub. Accessed: Aug. 28, 2024. [Online]. Available: https://github.com/jketterl/openwebrx/wiki/API

[50] D. K. Casey, J. Higginbotham, and K. Casey, “A Practical Approach to API Design from Principles to Practice,” 2016. [Online]. Available: http://leanpub.com/restful-api-design

[51] J. K. Becker and D. Starobinski, “Snout: A Middleware Platform for Software-Defined Radios,” *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, pp. 644–657, Mar. 2023, doi: 10.1109/TNSM.2022.3215626.

[52] “Node.js — Run JavaScript Everywhere.” Accessed: Feb. 14, 2025. [Online]. Available: https://nodejs.org/en

[53] S. Pasquali, “Mastering Node.js.” 2013. [Online]. Available: http://www.digitalbreakdown.net/sandbox/Ebooks/Mastering-Node.js.pdf

[54] “Express - Node.js web application framework.” Accessed: Dec. 10, 2024. [Online]. Available: https://expressjs.com/

[55] “Using Express middleware.” Accessed: Dec. 10, 2024. [Online]. Available: https://expressjs.com/en/guide/using-middleware.html

[56] “Express morgan middleware.” Accessed: Dec. 10, 2024. [Online]. Available: https://expressjs.com/en/resources/middleware/morgan.html

[57] “ON6ZQ | CW/CW practice -100 most common English words.” Accessed: Feb. 14, 2025. [Online]. Available: https://on6zq.be/w/index.php/CW/100MostCommonEnglishWords

Appendix A How to get started with DRBS

A.1 README

This section is extracted from the Github page which hosts the project files and code libraries. This project is a web application that allow users to listen to public web based SDRs, record the feed and decode Morse Code

To try out the web app on your computer, follow the instructions below:

1. Clone the repository
2. Install dependencies
3. Navigate to the repository directory and then run the following commands:

   # backend

   cd api

   npm install

   # frontend

   cd ../ui

   npm install --force

4. Running the backend
   From repository directory and then run the following commands:

   cd api

   npm start

5. Running the frontend

   From repository directory and then run the following commands:

   cd ui

   npm start

A.2 Tutorial – Basic user guide

Introduction

The DRBS application is a web broadcast streaming tool that provides the ability to cycle across SDR receivers and search for specific data. This section is a basic user guide that can assist the user to get started.

1. Software installation and setup

Refer to the READ ME file provided in Appendix A-1 for downloading the DRBS software, complete installation process with step-by-step instructions.

Prerequisites:

- User has read the READ ME file and has successfully installed DRBS software and has successfully started backend and frontend servers.

2. Login page

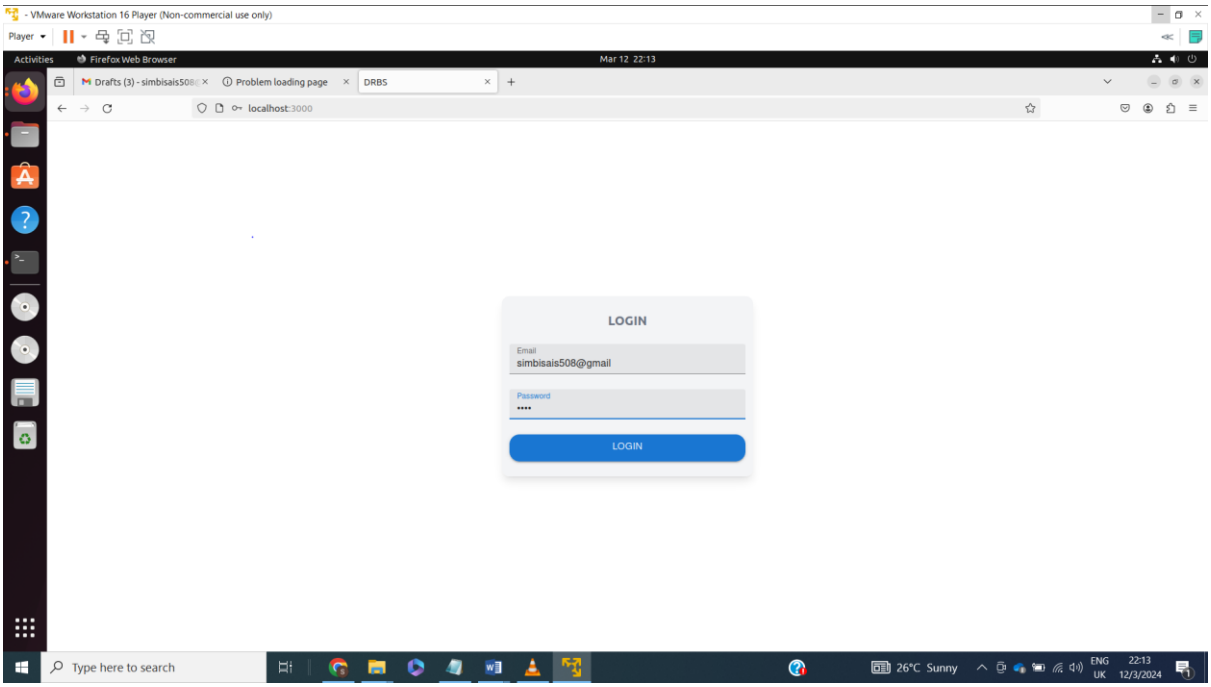

Figure A.1 Login page.

Enter Username (Email) and Password

Please note that you must be manually registered by the administrator to become a user.

3. Exploring the DRBS Application Interface

There are two main components on the landing page: Receiver List and the Control Panel.

3.1 Selecting a receiver

Go to the receiver list and double click on the receiver of choice to connect to the receiver.

Once connected to a receiver, a waterfall display showing signal history over time is displayed.

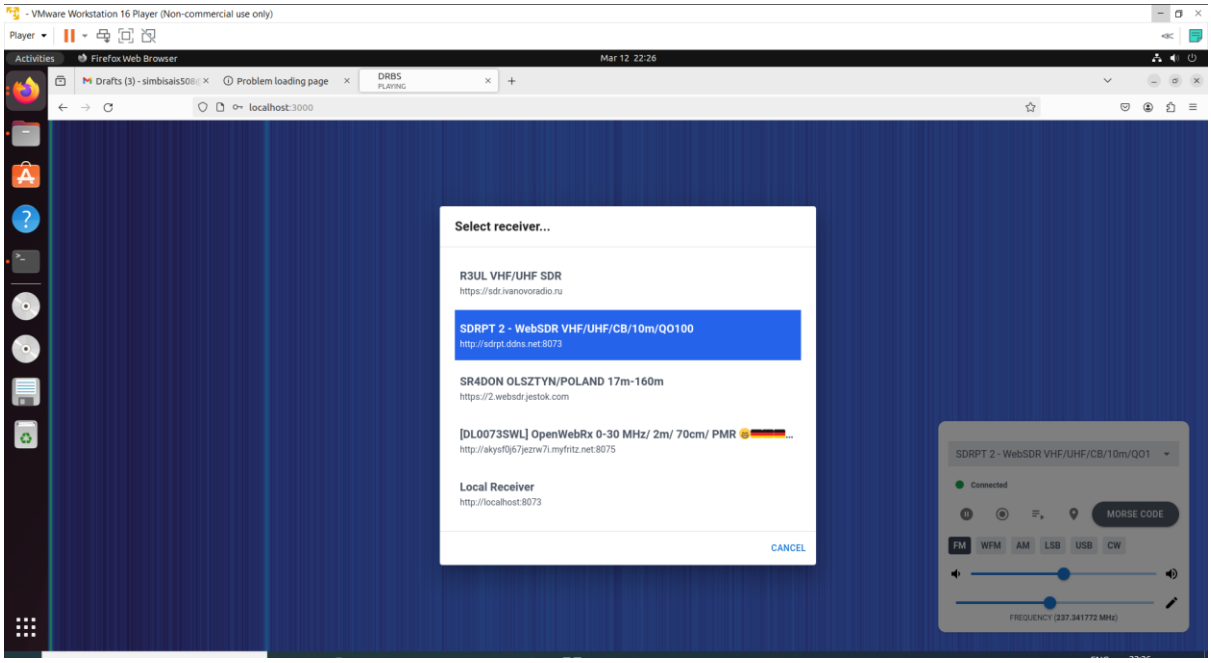

Figure A.2 Landing page.

3.2 Control panel

The control panel is located on the bottom right corner of the screen. It has several functions such as:

- Frequency tuning controls

- Audio control
- Modulation modes: AM, FM, WFM, LSB, USB, CW
- Audio recording options
- FFT visualizer display
- Select receiver option
- Display receiver GPS location
- Morse code detector and decoder

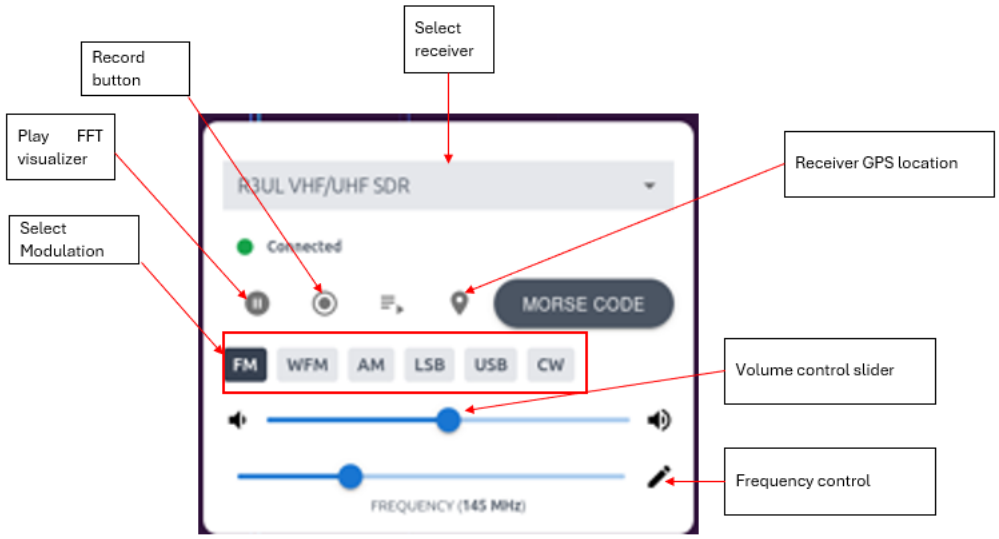

Figure A.3 Control panel.