1 Introduction

1.1 Background of Study

Underwater communication networks are vital in various domains such as scientific exploration,environmental monitoring, maritime security, disaster response, subsea infrastructure and maintenance, etc. The continuous advancement and deployment of underwater communication networks are quintessential for unlocking the full potential of the oceans to address the challenges faced by maritime industries.

1.2 Problem Statement

Traditional wireless communication technologies do not work underwater as electromagnetic waves are rapidly absorbed by water. Acoustic communication systems are used instead. Analog voice communication systems using acoustics are operational and commercially available today, but the voice quality is generally poor when the operating conditions are challenging. Digital data communication techniques have improved in recent years, and underwater modems have become available for computer-to-computer underwater data transfer.

1.3 Aim of the Project

The goal of this project is to be able to successfully record voice signals from a single webpage and transmit the voice signals to Node B in the UnetStack Framework, from which the compressed voice signals would be transmitted to Node A and then to a second webpage, which is listening for any datagrams, where the decompressed audio is played.

1.4 Requirements of the Project

For this project to be executed successfully, there were a number of things which were required to be in place. Firstly, a web page in HTML and JavaScript which was served from Node B that would be able to record the audio by a simple press of a button, stop recording the audio and send the audio, in bits using DatagramRequest. Secondly, a Groovy Agent had to be developed, that could handle sending of the DatagramRequest, DatagramNtf as well as being able to call the voice compression encoder and decoder to encode and decode the voice signals respectively. Lastly, is another web page in HTML and JavaScript served from Node A that would be able to play the decoded audio.

2 Lyra

For this project, Lyra was the choice of the voice compression technology

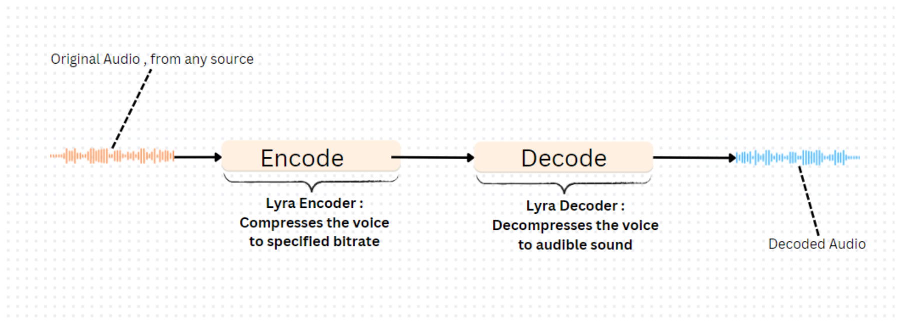
2.1 Introduction to Lyra

Lyra is a speech codec that offers high quality,low bitrate voice communication available in the lowest of networks. To achieve this, it utilizes traditional codec techniques and machine learning models trained on thousands of hours of data to create a method for compressing and transmitting voice signals (Luebs, 2022).

2.2 Lyra Architecture

Lyra simply is a 2-node network: an encoder and decoder. It encodes the input voice and compresses it to the specified bitrate and then decodes the encoded audio, playing back the decoded audio. **Figure 1** below, illustrates the Lyra Architecture for easy reference:

Figure 1: Lyra Encoder and Decoder Architecture



2.3 Installation of Lyra

We used a Linux virtual machine on Ubuntu to build Lyra as it offered the most user-friendly and fastest way to build on our machine. **Appendix A** covers the necessary steps and set-up requirements to build Lyra on a local machine.

2.4 Benefits of Using Lyra

For our project, we integrated the latest version of Lyra which is Lyra Version 2 (V2 for short). With the new architecture, the delay in the voice to be encoded and decoded is reduced from 100 ms with the previous version to 20 ms. In this regard, Lyra V2 is comparable to the most widely used audio codec. As compared to the previous versions, the encoding and decoding process offers better audio quality on a wider range of bitrates. Lastly, it is an open source release, which makes it widely available to be used in projects. Before implementing Lyra into our project, it is quintessential to understand the integral role of UnetStack in this project. As such, the next chapter discusses the UnetStack technology in more depth.

UnetStack

3.1 Introduction To UnetStack

UnetStack is a collection of technologies to extend communication networks underwater. A unet comprises or consists of many nodes(e.g underwater sensor nodes, Autonomous Underwater Vehicles (AUVs), gateway buoys, ground stations, boats/ships) that generate, consume or relay data over a variety of links. Unet nodes are equipped with one or more network interfaces that allow communication over some of these links (Chitre, 2014). UnetStack has several components to develop, simulate, test and deploy unets: Unet Framework, Unet Basic stack, Unet Premium Stack, Unet Simulator, Unet IDE and Unet Audio (Chitre, 2014). For simplicity, our project will be using the Unet Simulator. In order to achieve programmatic interaction with UnetStack from external programmes, UnetSocket is used. In the next section we discuss, more in detail how UnetSocket functions.

3.2 Unet Agents

Agents are the basic building blocks of the UnetStack. They exchange messages, provide services and implement protocols(Chitre,). The goal of the agent in this project is to act as a middle-man, whereby its responsibilities is to initiate DatagramRequest, DatagramNtf as well as call the Lyra Encoder/Lyra Decoder, depending if the.

3.3 Unet Simulator

The Unet simulator is able to simulate Unets with many nodes on a single computer. It can run in realtime simulation mode for interactive testing of agents and protocols, working to provide the user with the same user experience as in a real Unet. It can also be run in discrete event simulation mode to perform a large number of simulations in a short time, allowing Monte Carlo testing and performance evaluation of network protocols. In order to integrate UnetStack Stimulator into the project, we have to first set up the UnetStack Simulator.

3.4 Setting Up UnetStack Simulator

The purpose of the UnetStack Simulator is to set up 2 Nodes: Node A and Node B between which the transmission of data would take place. **Appendix B, section 2.1** covers in detail the steps to set up the UnetStack Simulator.

3.5 Unet.js Library

This library extends the browser-based fjage.js API to simplify interaction with agents and services in UnetStack, a framework for underwater network simulations and communication. It allows developers to easily get and set parameters on agents (like modem or physical layer agents) using high-level JavaScript functions, by wrapping the low-level ParameterReq message system. It also enhances the AgentID objects with utility functions and adds convenient access to UnetStack's predefined services (such as PHYSICAL, LINK, DATAGRAM, etc.). Overall, it streamlines parameter control and agent communication for real-time web interfaces connected to UnetStack simulations or deployments.

4 Integration of Lyra and UnetStack into Project

4.1 Introduction of Groovy Agent

A Groovy Agent, by the name LyraProcessor.groovy was introduced to handle incoming DatagramRequest and DatagramNtf and based on the respective scenario, it could call the Lyra Encoder or Lyra Decoder to encode and decode the audio respectively depending on receiving DatagramRequest and DatagramNtf. LyraProcessor.groovy would be running on both the nodes in the UnetStack before the recording began.

4.2 Integration of UnetStack simulator with project

The webpages to record the audio and play the decoded audio respectively would be served from the nodes B and A of the UnetStack framework, so that the nodes could send and receive the audio in bits from the respective webpages.

4.3 Overall Process

The overall architecture of the project is illustrated in the Figure 2 below.

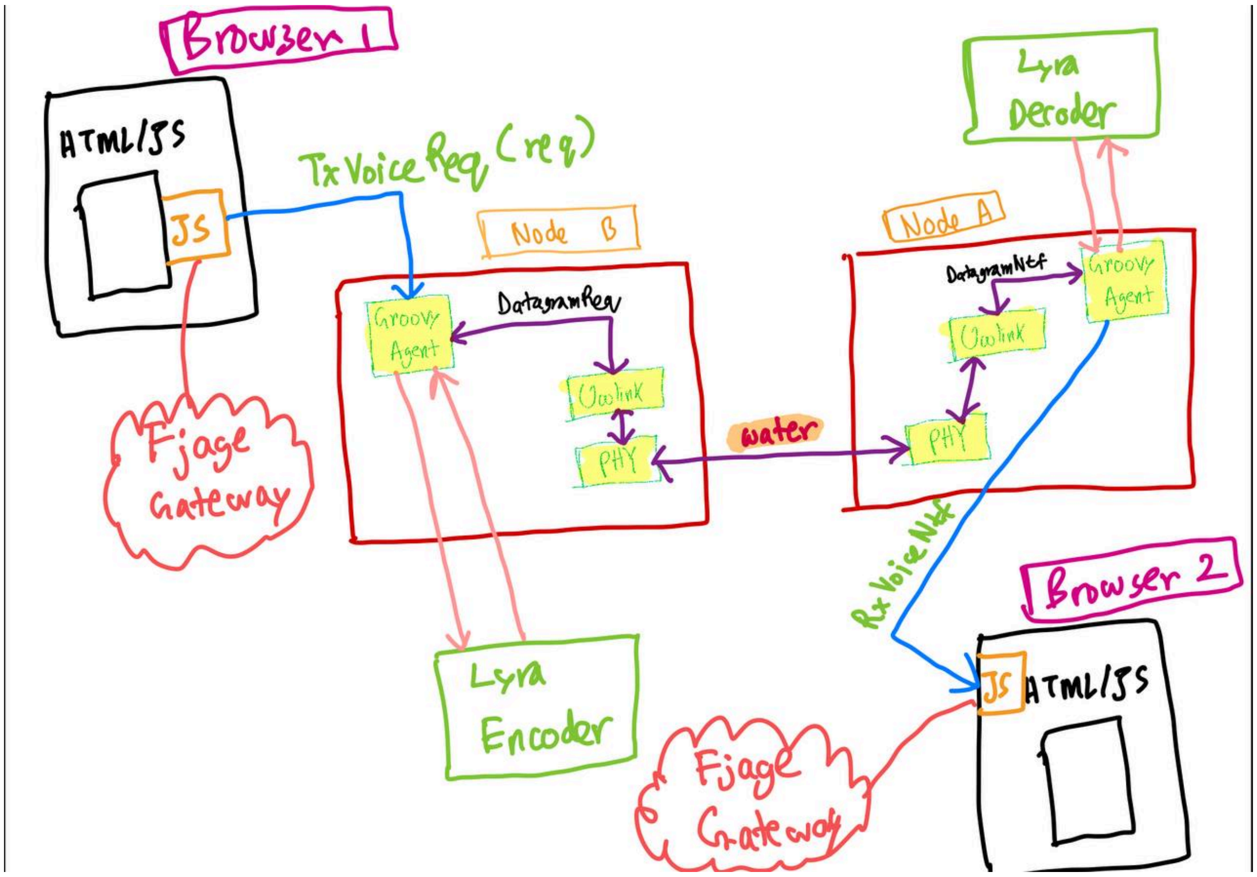


Figure 2: Overall Architecture

From our first website (**"Browser 1"**), which is served on **Node B**, users can initiate audio recording by clicking the "Record" button. Once the **stop** button is pressed, the recording ends, and clicking **"Send DatagramRequest"** triggers the transmission. The core objective is to convert the recorded audio into bits and transmit it from **1** to the first node in UnetStack, **Node B,** from which the webpage is served from by a DatagramRequest.

Upon receiving the DatagramRequest, LyraProcessor in **Node B** invokes the **Lyra Encoder**, which compresses the voice signal by calling the Lyra Encoder. After encoding the voice signals successfully, these bits are then forwarded as a DatagramRequest to the **Uwlink** layer. Once the Uwlink layer receives this request, it sends a DatagramNtf to **Node A**, signaling the arrival of the compressed voice data.

At Node A, the **Lyra decoder** is activated by LyraProcessor to decompress the bitstream back into an audible audio signal. This decoded audio is then published to the appropriate topic(). In this context, topic() refers to our second web page (**"Browser 2"**), which is subscribed to the Groovy agent, named LyraProcessorGroovy() and is served from **Node B**. **2** listens for any DatagramNtf from LyraProcessorGroovy() agent and once **2** receives the decoded audio, the audio is seamlessly played back on the website. To better understand the respective parameters, we will need to delve deeper into the scripts employed in the next section.

5 Organization of Scripts

To achieve the objective, we used the **Groovy** to implement a custom **fjåge agent named "LyraProcessor"**, which handled encoding and decoding of audio using the **Lyra voice codec** within the **UnetStack simulator** environment. Additionally, we developed two separate **web pages using HTML and JavaScript**:

- **"Browser 1"** is responsible for **recording audio from the user and sending it to an agent via a datagram**, and

- **"Browser 2"** is used to **monitor the target agent (e.g., "mahir") for any** `DatagramNtf`, **receive audio data, and play it back in real time**.

These pages were served from **Node B and Node A** of the UnetStack simulator respectively, enabling bidirectional communication and audio streaming between the nodes through the use of fjåge Gateways and the UnetStack messaging system. See **Appendix C** for all the 3 scripts and how they work.

5.1 Function of LyraProcessor script

The purpose of the LyraProcessor script(covered in **Appendix C, section 3.3**) is to define a custom fjåge agent called `LyraProcessor`, which enables the encoding and decoding of audio files using Google's Lyra voice codec within a UnetStack-based underwater communication system. When the agent starts up, it subscribes to the `DATAGRAM` and `LINK` services to enable message-based interactions. Upon receiving a `DatagramReq` message with audio data under the `USER` protocol, it saves the audio to a temporary file, converts it into a valid 16 kHz mono WAV format using `ffmpeg`, and then compresses it using the Lyra encoder. The resulting `.lyra` encoded audio is sent as another `DatagramReq` to the appropriate communication link. Conversely, when the agent receives a `DatagramNtf` containing a `.lyra` file, it decodes the file back into a WAV format using the Lyra decoder and sends the resulting audio data out as a `DatagramNtf` with the `DATA` protocol.

5.2 Function of Web Pages

This script “Browser 1” (covered in **Appendix C, section 3.1**) is designed to record audio from the user's microphone using the browser's `MediaRecorder` API and send the recorded audio as a datagram message to a remote agent using the fjåge framework. When the user clicks the "Start Recording" button, the script captures audio input, stores it in chunks, and waits for the "Stop Recording" button to be pressed. Once recording stops, the audio chunks are combined into a single WAV-formatted blob. This blob is then converted into a byte array (`Uint8Array`) and sent as a `DatagramReq` message to a specified agent (named `mahir`) via the fjåge `Gateway` interface. The script uses fjåge’s messaging system to define and send the datagram with specified `to` and `protocol` values.

This script ”Browser 2”(covered in **Appendix C, section 3.2**) serves as a web-based **monitoring and audio playback interface** for a fjåge agent named **"mahir"** in a UnetStack environment. When the page loads, it establishes a connection to the fjåge system using the `Gateway` module and checks if the agent "mahir" is currently active. If the agent exists, the script subscribes to its message topic and starts listening for `DatagramNtf` messages, which are typically used for receiving data such as audio payloads. When such a message is received, it extracts the audio byte array from the message, converts it into an `ArrayBuffer`, decodes it using the browser’s `AudioContext`, and plays the audio back in real time.

Appendix

1 Appendix A: Building Lyra

1.1 Set-up requirements:

1. Google's Bazel System (5.00 and above [as of October 2023])
2. Python3
3. Numpy

1.2 Steps to Build Lyra:

1. Clone the GitHub repository containing Lyra to Linux machine using the following command :

    git clone https://github.com/google/lyra.git

2. On the command terminal navigate to the directory where Lyra was cloned to

3. Generating the necessary binaries by putting in the following commands on my terminal:
    a. `mkdir build`
    b. `cd build`
    c. `cmake ..`
    d. `cmake --build`

4. Build Lyra using bazel using the following:

```
bazel build -c opt lyra/cli_example:encoder_main
```

5. After completing the build of Lyra , we can proceed with first encoding the speech in the following format:

```
bazel-bin/lyra/cli_example/encoder_main
--input_path=/home/m4hirr4mos/Downloads/Female_fast_wav.wav
--output_dir=/home/m4hirr4mos/Downloads/Compressed\ Music
--bitrate=3200
```

[input path- indicates the directory of the input WAV file
output path- indicates where the output compressed file is going to be placed
bitrate- specifies the bitrate we want to compress the voice down into]

6. We can now proceed with decoding the compressed speech:

```
bazel-bin/lyra/cli_example/decoder_main
--input_path=/path/to/compressed.lyra
--output_path=/path/to/output.wav
```

[input path-indicates the location of the encoded speech
output path-indicates the location of where to place the decoded speech
to]

2 Appendix B: UnetStack

2.1 Setting up Unet Simulator

The following details the steps to set up the UnetStack Simulator on the local machine:

Step 1: Download the latest version UnetStack Community Version from:
https://unetstack.net/#downloads
Step 2: Navigate to the bin/unet samples/2-node-network.groovy directory and replace the script with the **2NodeNetworkScript in Appendix C section 3.1.**
Step 3: Re-run the following command after navigating to the UnetStack directory on your local machine: `bin/unet samples/2-node-network.groovy.`
Step 4: You should get the following output:

```
2-node network
--------------

Node A: tcp://localhost:1101, http://localhost:8081/
Node B: tcp://localhost:1102, http://localhost:8082/
```

This sets up the 2 Node Network, Node A and Node B for the UnetStack Simulator. They can be opened by pressing on the respective links "http://localhost:8081/" for Node A and "http://localhost:8082/" for Node B. Then select the shell button to open up the integrated development environment (IDE) where the various data will be displayed during data transmission and receiving. In the next chapter, we will observe how UnetStack and Lyra are jointly integrated into our project.

3 Appendix C: Browser 1, Browser 2 and LyraProcessor script

3.1 Browser 1 Script

```
!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Audio Recorder with Datagram Request</title>
</head>
<body>
    <button id="startButton">Start Recording</button>
    <button id="stopButton" disabled>Stop Recording</button>
    <button type="button" id="sendBtn" disabled>Send Datagram</button>

    <script type="module">
        import { Gateway, Services, MessageClass } from '/js/unet.js';

        var gw = new Gateway();

        let mediaRecorder;
        let chunks = [];
        const DatagramReq = MessageClass('org.arl.unet.DatagramReq');

        // Function to start recording
        function startRecording() {
            navigator.mediaDevices.getUserMedia({ audio: true })
                .then(function (stream) {
                    mediaRecorder = new MediaRecorder(stream);

                    // Collect audio data
                    mediaRecorder.ondataavailable = function (e) {
                        chunks.push(e.data);
                    };

                    // When recording stops
                    mediaRecorder.onstop = function () {
                        // Create a single Blob from chunks
                        const audioBlob = new Blob(chunks, { type:
'audio/wav' });
                        chunks = []; // Reset chunks

                        // Send the recorded audio directly as a datagram
                        sendDatagram(audioBlob);
                    };

                    mediaRecorder.start();
                    console.log('Recording started...');
                    document.getElementById('startButton').disabled =
true;
                    document.getElementById('stopButton').disabled =
false;
                })
                .catch(function (err) {
                    console.error('Error accessing microphone:', err);
                });
        }

        // Function to stop recording
        function stopRecording() {
            if (mediaRecorder && mediaRecorder.state === 'recording') {
                mediaRecorder.stop();
                console.log('Recording stopped...');
                document.getElementById('startButton').disabled = false;
                document.getElementById('stopButton').disabled = true;
            }
        }

        // Function to send the recorded audio as a datagram
        function sendDatagram(audioBlob) {
            const arrayBufferPromise = audioBlob.arrayBuffer(); // Get the
ArrayBuffer directly from the Blob

            arrayBufferPromise.then(arrayBuffer => {
                const uint8Array = new Uint8Array(arrayBuffer);

                console.log('Sending a datagram...');

                // Create Datagram Request
                var req = new DatagramReq();
                req.recipient = gw.agent('mahir'); // Recipient agent
                req.to = 232;
                req.protocol = 32;
                req.data = Array.from(uint8Array); // Convert Uint8Array
to array for datagram
                //req.protocol = gw.Protocol.USER;

                gw.send(req);
                console.log("Datagram has been sent");
            });
        }

        // Event listener for the start button
        document.getElementById('startButton').addEventListener('click',
startRecording);

        // Event listener for the stop button
        document.getElementById('stopButton').addEventListener('click',
stopRecording);
    </script>
</body>
</html>
```

3.2 Browser 2 Script

```
<!DOCTYPE html>

<html>

<head>

  <meta charset="utf-8">

  <title>Monitor Agent Mahir</title>

  <script type="module">

    import { Gateway, MessageClass } from '/js/unet.js';



    window.addEventListener('DOMContentLoaded', async () => {

      const gw = new Gateway();

      const audioContext = new (window.AudioContext ||
window.webkitAudioContext)();



      function display(msg) {

        const li = document.createElement('li');

        li.textContent = msg;

        document.getElementById('messages').appendChild(li);

      }
```

```
try {

  // Step 1: Check if agent "mahir" exists

  const rsp = await gw._msgTxRx({

    action: 'containsAgent',

    agentID: 'mahir'

  });



  const exists = rsp?.answer === true;



  if (exists) {

    display("✅ Agent 'mahir' is running.");

    console.log("✅ Agent 'mahir' is running.");



    // Step 2: Subscribe to mahir's topic

    const mahirTopic = gw.topic(gw.agent("mahir"));

    await gw.subscribe(mahirTopic);

    display("📡 Subscribed to: " + mahirTopic.name);

    console.log("📡 Subscribed to topic:", mahirTopic.name);



    // Step 3: Listen for DatagramNtf messages from mahir

    gw.addMessageListener(async msg => {
            console.log("📥 Received:", msg);

            if (msg.__clazz__ === "org.arl.unet.DatagramNtf" && msg.data)
{
              display(`🎧 Audio received from ${msg.from}
(${msg.data.length} bytes)`);

              try {
                // Debug: log the type and contents of msg.data
                console.log("🔍 msg.data type:", typeof msg.data,
msg.data);

                // Step 1: Convert to Uint8Array safely
                let audioBytes;
                if (msg.data instanceof Uint8Array) {
                  audioBytes = msg.data;
                } else if (Array.isArray(msg.data)) {
                  audioBytes = new Uint8Array(msg.data);
                } else {
                  throw new Error("msg.data is not a valid audio byte
array.");
                }
```

```
          // Step 2: Convert to ArrayBuffer
          const arrayBuffer = audioBytes.buffer.slice(
            audioBytes.byteOffset,
            audioBytes.byteOffset + audioBytes.byteLength
          );

          // Step 3: Decode and play audio
          const audioBuffer = await
audioContext.decodeAudioData(arrayBuffer);
          const source = audioContext.createBufferSource();
          source.buffer = audioBuffer;
          source.connect(audioContext.destination);
          source.start();
        } catch (err) {
          console.error("❌ Audio decoding failed:", err);
          display("⚠️ Failed to decode and play audio.");
        }

        return true;
      }
```

```
              return false;

            });



          } else {

            display("❌ Agent 'mahir' is NOT running.");

          }



        } catch (e) {

          console.error("⚠️ Error:", e);

          display("⚠️ Error occurred while checking or subscribing.");

        }

      });

    </script>

  </head>

  <body>

    <h1>Agent Monitor: Mahir</h1>

    <ul id="messages"></ul>

  </body>

  </html>
```

3.3 LyraProcessor.groovy

```
import org.arl.fjage.*
import org.arl.unet.*
import java.nio.file.Files
import java.nio.file.Path
import java.nio.file.Paths

class LyraProcessor extends UnetAgent {

    @Override
    void startup() {
        subscribeForService(Services.DATAGRAM)
        subscribeForService(Services.LINK)
    }

    static String getFileType(String filePath) {
        return filePath.substring(filePath.lastIndexOf('.') +
1).toLowerCase()
    }

    static def lyraEncoderBin =
"/home/m4hirr4mos/lyra/bazel-bin/lyra/cli_example/encoder_main"
    static def lyraDecoderBin =
"/home/m4hirr4mos/lyra/bazel-bin/lyra/cli_example/decoder_main"
    static def ffmpegBin = "/snap/bin/ffmpeg" // Path to ffmpeg binary

    static Path convertToValidWav(Path inputPath) {
        Path fixedWavPath =
Paths.get("/home/m4hirr4mos/temp/fixed_file.wav")
        def command = [
            ffmpegBin, "-y",
            "-i", inputPath.toString(),
            "-acodec", "pcm_s16le", "-ar", "16000", "-ac", "1",
            fixedWavPath.toString()
        ] as String[]

        println("Converting audio to valid WAV format with command:
${command.join(' ')}")
```

```
        ProcessBuilder builder = new ProcessBuilder(command)
        builder.inheritIO()
        Process process = builder.start()
        process.waitFor()

        if (process.exitValue() != 0) {
            throw new RuntimeException("Audio conversion failed with exit
code: ${process.exitValue()}")
        }

        println("Conversion successful: ${fixedWavPath}")
        return fixedWavPath
    }

    static Path encodeWithLyraEncoder(Path inputPath, Path outputDir) {
        def bitrate = 3200
        def outputPath = Paths.get("/home/m4hirr4mos/temp/temp_file.lyra")
        def command = [
            lyraEncoderBin,
            "--input_path=${inputPath.toString()}",
            "--output_dir=${outputDir.toString()}",
            "--bitrate=${bitrate}"
        ] as String[]
        println("Running Lyra encoder with command: ${command.join(' ')}")
        ProcessBuilder builder = new ProcessBuilder(command)
        builder.inheritIO()
        Process process = builder.start()
        process.waitFor()
        if (process.exitValue() != 0) {
            throw new RuntimeException("Encoding failed with exit code:
${process.exitValue()}")
        }
        println("Encoding completed")
        return outputPath
    }

    static Path decodeWithLyraDecoder(Path encodedPath, Path outputDir) {
        def outputPath =
Paths.get("/home/m4hirr4mos/temp/temp_file_decoded.wav")
        def command = [
            lyraDecoderBin,
            "--encoded_path=${encodedPath.toString()}",
            "--output_dir=${outputDir.toString()}",
            "--bitrate=3200"
        ] as String[]
        println("Running Lyra decoder with command: ${command.join(' ')}")
        ProcessBuilder builder = new ProcessBuilder(command)
        builder.inheritIO()
        Process process = builder.start()
        process.waitFor()
        if (process.exitValue() != 0) {
            throw new RuntimeException("Decoding failed with exit code: 
${process.exitValue()}")
        }
        println("Decoding completed")
        return outputPath
    }

    static Path processFile(Path filePath, Path outputDir, boolean 
isEncoding) {
        return isEncoding ? encodeWithLyraEncoder(filePath, outputDir) : 
decodeWithLyraDecoder(filePath, outputDir)
    }

    Message processRequest(Message msg) {
        if (msg instanceof DatagramReq && msg.protocol == Protocol.USER) {
            Path tempFilePath = Paths.get(System.getenv("HOME"), "temp", 
"temp_file.wav")
            Files.write(tempFilePath, msg.data)

            def uwlink = agentForService(org.arl.unet.Services.LINK)
            Path validWavFile = convertToValidWav(tempFilePath) // Convert 
before encoding

            Path encodedFilePath = processFile(validWavFile, 
Paths.get(System.getenv("HOME"), "temp"), true)
            byte[] encodedData = 
Files.readAllBytes(Paths.get("/home/m4hirr4mos/temp/fixed_file.lyra"))

            send new DatagramReq(
                recipient: uwlink,
                to: msg.to,
                protocol: Protocol.USER,
                data: encodedData
            )

            return new Message(msg, Performative.AGREE)
        }
    }

    @Override
    void processMessage(Message msg) {
        if (msg instanceof DatagramNtf && msg.protocol == Protocol.USER) {
            Path tempFilePath = Paths.get(System.getenv("HOME"), "temp",
"temp_file.lyra")
            Files.write(tempFilePath, msg.data)

            Path decodedFilePath =
processFile(tempFilePath,Paths.get(System.getenv("HOME"), "temp"), false);

            byte[] decodedData = Files.readAllBytes(decodedFilePath)

            send new DatagramNtf(
                recipient: topic(),
                protocol: Protocol.DATA,
                data: decodedData
            )
        }
    }
}
```