Bachelor's thesis

Information and Communications Technology

2024

Otto Laitinen

Developing a Raspberry Pi Audio Player for Commercial Spaces

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2024 | 32 pages

Otto Laitinen

Developing a Raspberry Pi Audio Player for Commercial Spaces

The aim of this thesis was to develop a reliable, cost-effective, and continuously operating Raspberry Pi-based audio player for commercial environments. The purpose of the player is to meet the needs of a Finnish media company specializing in background music solutions.

As a result of the thesis, a player that combines streaming and local, royalty-free music, ensuring uninterrupted playback even without an internet connection, was created. The player utilizes technologies such as SQLite database, VLC media player, Python, and APScheduler Python library for scheduled content playback. Additionally, the player uses Hazelcast for real-time data synchronization with the server. The architecture allows for remote management without a direct user interface.

Testing demonstrated the functional efficiency and reliability of the player and highlighted its potential for widespread use in commercial environments. There are future development plans for the player that could expand its functionality and market reach. The player is currently in commercial use, thereby promoting the use of IoT applications in commercial audio systems.

Keywords:

Raspberry Pi, commercial audio systems, streaming technology, Hazelcast, VLC Media Player, IoT-applications

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2024 | 32 sivua

Otto Laitinen

Raspberry Pi -äänisoittimen kehittäminen kaupallisiin tiloihin

Työn tavoite oli kehittää luotettava, kustannustehokas ja jatkuvasti toimiva Raspberry Pi -pohjainen äänisoitin kaupallisiin ympäristöihin. Soittimen tarkoitus on täyttää taustamusiikkiratkaisuihin erikoistuvan suomalaisen mediayrityksen tarpeet.

Työn tuloksena oli soitin, joka yhdistää suoratoiston ja paikallisen, tekijänoikeusvapaan musiikin, varmistaen keskeytymättömän toiston myös ilman internet-yhteyttä. Soitin hyödyntää teknologioita, kuten SQLite-tietokantaa, VLC-mediasoitinta, Pythonia ja APScheduler Python -kirjastoa sisällön aikataulutettuun soittamiseen. Lisäksi soitin käyttää Hazelcastia reaaliaikaiseen datan synkronointiin palvelimen kanssa. Arkkitehtuuri mahdollistaa etähallinnan ilman suoraa käyttöliittymää.

Testausvaiheet osoittivat soittimen toiminnallisen tehokkuuden ja luotettavuuden ja korostivat sen potentiaalia laajalle käytölle kaupallisissa ympäristöissä. Soittimelle on tulevaisuuden kehityssuunnitelmia, jotka voisivat laajentaa sen toiminnallisuutta ja markkina-aluetta. Soitin on tällä hetkellä kaupallisessa käytössä, ja edistää näin IoT-sovellusten käyttöä kaupallisissa äänijärjestelmissä.

Asiasanat:

Raspberry Pi, kaupalliset äänijärjestelmät, suoratoistoteknologia, Hazelcast, VLC-Mediasoitin, IoT-sovellukset

Contents

List of Figures

List of Pictures

List of abbreviations

API	Application Programming Interface
CSV	Comma-separated Values
DDL	Data Definition Language
IoT	Internet of Things
IP	Internet Protocol
SSH	Secure Shell
UI	User Interface
URL	Uniform Resource Locator
VPN	Virtual Private Network
WSL	Windows Subsystem for Linux

1 Introduction

The objective of the thesis is to document the development of an audio player for scheduled content for Raspberry Pi (Picture 1), a compact and cost-effective single-board computer [1]. This project addresses the needs of a Finnish media company, from now on referred to as the customer, specializing in background audio solutions for commercial spaces. This initiative is undertaken at the behest of Fidera, a Finnish IoT and software development company, under which the customer is categorized as a client project.

Challenges such as copyright compliance, customization constraints, cost considerations, and dependency on continuous internet connectivity have created a need for a more adaptable and resilient audio delivery system for the commercial audio landscape. In response, the custom audio player, from now on referred to simply as the player, designed in this project addresses these needs by ensuring continuous operation, even under challenging conditions.

Challenges such as copyrights regarding playing music in commercial spaces, customization limits of the content, and dependency on internet connection have created a need for a resilient and cost-effective audio delivery system. The player designed in this project addresses these needs by ensuring continuous operation, even under challenging conditions.

The player primarily uses streaming to play background music, which reduces costs compared to using locally stored audio files. Additionally, it automatically switches to a locally stored library of royalty-free music during internet outages, ensuring that music continues without interruption. Besides scheduling streams, it also supports scheduling advertisements and announcements from local storage, making it useful for commercial settings where messages or promotions need to be aired at specific times regardless of internet connectivity.

The player’s plug-and-play functionality eliminates the need for a direct UI, simplifying deployment and enhancing usability. The players are remotely controlled and managed via a server-side admin UI.

This thesis documents the project from an idea to deployment, keeping the main focus on the player software, starting with customer requirements and the selected technologies. It then delves into the architectural decisions, detailing the development process and strategies. The testing phase outcomes are critically examined to evaluate the system's reliability and effectiveness. The thesis concludes by assessing the project's impact and suggesting potential future enhancements.

Picture 1. Raspberry Pi used in the project.

2 Customer Requirements

This chapter discusses some of the main initial requirements set by the customer.

2.1 Playback Functionalities

The player should support advanced stream playback capabilities, including an offline music library filled with a selection of royalty-free music. This setup should include health monitoring and an automatic transition to the local music library during streaming failures to ensure uninterrupted service. Additionally, interruptions in internet service or stream feed should not affect the scheduled playback of advertisements and announcements, meaning that the player should download and store necessary audio files locally beforehand from a remote server.

2.2 Updates

The player’s software upgrade process should be simple, enabling players to independently fetch and apply software updates from a central server, minimizing manual maintenance.

2.3 Real-Time Remote Control

The player should use remote management through a pre-existing server-side UI, which will be briefly described in subchapter 4.1. The player should be configurable through this UI, allowing functions such as assigning the player to a specific customer, creating scheduled content, and setting audio levels for each type of content. The communication should be real-time between the players and the server for consistent and immediate operations across all deployed players.

2.4 Database and Data Synchronization

The integration of a lightweight local database is another crucial requirement. It should be synchronized with a central server database to manage schedules and other player-specific data.

3 Selected Technologies

This chapter details the selected software components for the player to fulfill the customer’s requirements that were discussed in the previous chapter.

3.1 VLC

VLC, a free and open-source cross-platform multimedia player [2], is chosen for this project primarily due to its versatility and light resource demands, making it very suitable for use on Raspberry Pi.

Another critical factor in selecting VLC is its Python bindings, which allow for easy integration with the system's Python-based architecture. These bindings enable the scripting of audio playback functionalities directly within Python, aligning well with the project's requirements for automated and dynamic audio management.

3.2 Python

Python, an object-oriented, high-level programming language [3], is chosen for its inherent ability to execute shell scripts, perform system-level commands, and manage file and logging operations, making it a fit and simple choice for Raspberry Pi. These capabilities allow for easy control over the audio playback environment, including the dynamic execution of operational scripts, seamless management of log files for monitoring system health, and manipulation of files which are essential for maintaining an up-to-date content library.

Python pip library APScheduler, a library that allows the scheduling of tasks at specified times [4], is chosen to streamline the management of scheduling audio playback. Additionally, asyncio library, which allows the execution of concurrent code [5], is used to ensure smoother non-blocking simultaneous operations.

3.3 Hazelcast

Hazelcast, an open-source In-Memory Data Grid with high-speed processing [7], is employed for real-time data synchronization and communication with the remote server.

Hazelcast maps can be used for storing messages between the players and the server (Chapter 4, Figure 2), and these maps can be either unique to a specific player or belong to multiple players. For example, all players have their unique identification number, which can be used to ensure a specific player only listens to the changes in the maps that belong to it.

By leveraging Hazelcast, the system ensures that all instances of the player have the correct content and configuration settings, critical for maintaining a good user experience across multiple devices.

3.4 SQLite

SQLite, an in-process library that implements a serverless, zero-configuration, transactional SQL database engine [6], serves as the local database solution and enables the system to maintain high functionality levels even when offline. This lightweight, disk-based database does not require a separate server process, simplifying deployment and minimizing resource usage.

4 Operational Infrastructure and System Environment

This chapter shows a peek into the context of the ecosystem the player sits in, as well as the environment on the Raspberry Pi where the Python software runs.

4.1 Architectural Overview

The existing infrastructure uses a server-side UI for various user interactions, a central server and database for managing business logic and data, and a separate stream server for serving audio streams (Figure 1). This thesis will not dive into the details of these components but shows the underlying context the player sits in, and what is being developed around the player itself in terms of surrounding services.

The UI was originally designed for various purposes for the customer. To accommodate the functionalities of the player, the UI is being enhanced alongside the player development to include features for managing the audio players directly—such as assigning players to specific customers or locations, scheduling audio content, monitoring player status, and configuring device-specific settings.

The central server acts as the center of the backend operations, handling the data and processing of user requests. It serves as the intermediary that ensures data flow between the UI and the stream server, as well as between the UI and the physical audio players. With the integration of the player, additional features are developed to manage player-related data.

The pre-existing streaming server is responsible for creating and serving the audio streams based on the data it receives from the central server, managing the real-time broadcasting needs. The stream server does not interact directly with the players, but the central server coordinates with the streaming server to ensure that the content delivered to the players is correctly managed.

This architectural setup supports the existing operations and seamlessly integrates the player.

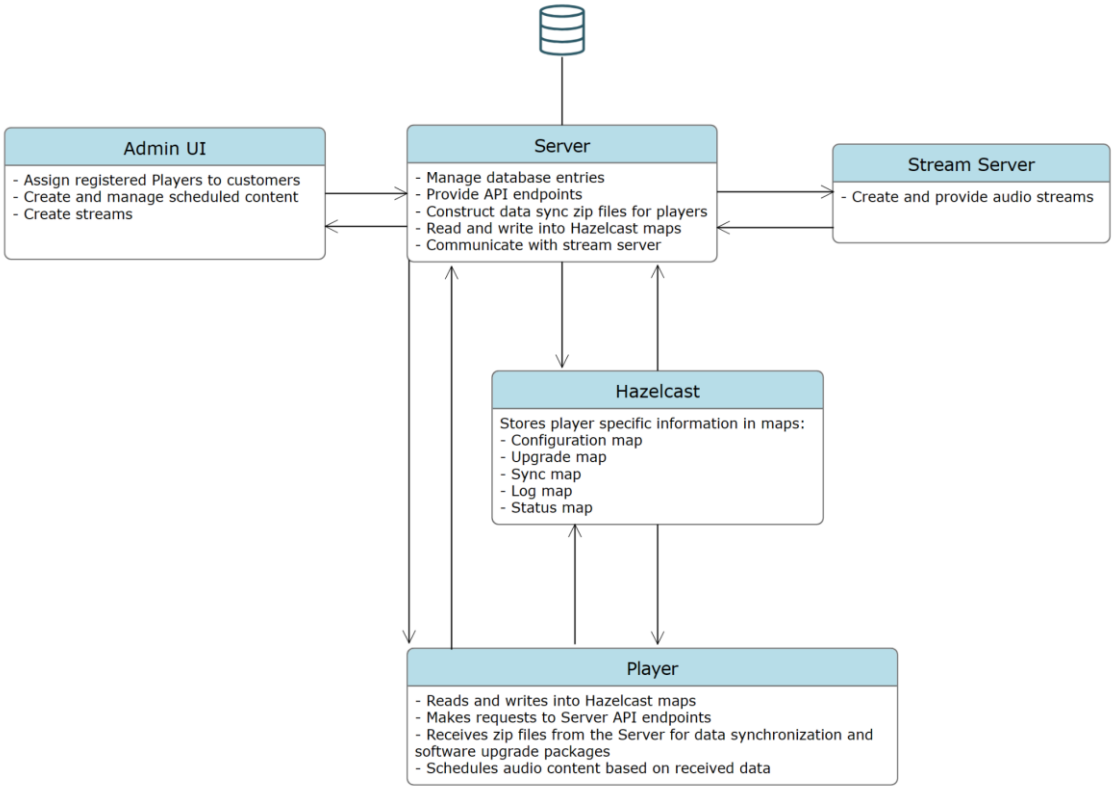

Figure 1. Simplified flow chart of the operational infrastructure.

4.2 System Environment

The operational framework of the player relies on the implementation of custom shell scripts and systemd services. Systemd is a system and service manager for Linux operating systems, responsible for initializing and managing system services and processes [8].

A basic systemd service that starts a Python program on boot and restarts it on failure is created for system resilience. This file specifies the working directory of the program, configures it to start at boot, and defines behavior for restarting the program if it crashes (Figure 2).

```
[Unit]
Description=Player Service
After=network.target

[Service]
ExecStart=/home/user/player/main
WorkingDirectory=/home/user/player
StandardOutput=inherit
StandardError=inherit
Restart=always
KillMode=process
User=user

[Install]
WantedBy=multi-user.target
```

Figure 2. A systemd service for auto-starting software.

Additionally, another systemd service addresses the Raspberry Pi's limitation of lacking a real-time clock. It ensures the system's time is accurate upon startup, a process indicated by the creation of a file. This file's existence can be detected by the Python program, allowing accurate timekeeping for precise initiation of scheduled tasks, further discussed in Chapter 5.

The system also has operational automation scripts included in it. For example, a script that adjusts network configurations (Figure 3) to meet specific customer needs. This adaptability is crucial for specifying which IP configurations the player should operate under, to address the needs of locations that require their players to operate in a local network for security reasons.

ip_config.sh

```bash
#!/bin/bash

IP_ADDRESS_MODE=$1  # Always required ('static' or 'dhcp')
IP_ADDRESS=$2       # Required for 'static'
GATEWAY=$3          # Required for 'static'
NAMESERVER1=$4      # Required for 'static'
NAMESERVER2=$5      # Optional
NAMESERVER3=$6      # Optional
NETMASK=$7          # Required for 'static' (in CIDR notation)

# Determine active ethernet connection name
determine_connection_name() {
    local connection_name=$(nmcli -t -f TYPE,NAME con show --active | grep -E 'ethernet' | cut -d':' -f2 | head -n 1)
    if [[ -n "$connection_name" ]]; then
        echo "$connection_name"
    else
        echo "No active Ethernet connection found." >&2
        exit 1
    fi
}

CONNECTION_NAME=${8:-$(determine_connection_name)}

# Set STATIC
set_static() {
    DNS="$NAMESERVER1"
    [ -n "$NAMESERVER2" ] && DNS="$DNS,$NAMESERVER2"
    [ -n "$NAMESERVER3" ] && DNS="$DNS,$NAMESERVER3"
    sudo nmcli con mod "$CONNECTION_NAME" ipv4.addresses $IP_ADDRESS/$NETMASK
    sudo nmcli con mod "$CONNECTION_NAME" ipv4.gateway $GATEWAY
    sudo nmcli con mod "$CONNECTION_NAME" ipv4.dns "$DNS"
    sudo nmcli con mod "$CONNECTION_NAME" ipv4.method manual
    sudo nmcli con up "$CONNECTION_NAME"
}

# Set DHCP
set_dhcp() {
    sudo nmcli con mod "$CONNECTION_NAME" ipv4.addresses "" ipv4.gateway ""
    sudo nmcli con mod "$CONNECTION_NAME" ipv4.method auto
    sudo nmcli con up "$CONNECTION_NAME"
}

# Main
if [[ "$IP_ADDRESS_MODE" == "static" ]]; then
    if [[ -z "$IP_ADDRESS" || -z "$GATEWAY" || -z "$NAMESERVER1" || -z "$NETMASK" ]]; then
        echo "For static configuration, IP_ADDRESS, GATEWAY, NAMESERVER1, and NETMASK are required."
        exit 1
    fi
    set_static
elif [[ "$IP_ADDRESS_MODE" == "dhcp" ]]; then
    set_dhcp
else
    echo "Invalid IP address mode: $IP_ADDRESS_MODE"
    echo "Usage: $0 <mode> <IP_ADDRESS> <GATEWAY> <NAMESERVER1> [NAMESERVER2] [NAMESERVER3] <NETMASK> <CONNECTION_NAME>"
    echo "Modes:"
    echo "  static: Requires IP_ADDRESS, GATEWAY, at least one NAMESERVER, NETMASK in CIDR notation, and CONNECTION_NAME."
    echo "  dhcp: Only requires the mode and CONNECTION_NAME; other parameters are ignored."
    exit 1
fi
```

Figure 3. Script for setting IP configurations on a Raspberry Pi.

For remote access, the system leverages SSH and VPN, providing a secure and encrypted channel for system administrators and developers to perform maintenance tasks and updates. SSH allows secure command-line access between two computers [9], allowing detailed system interaction, while VPN usage ensures a protected network environment for accessing the system from remote locations [10]. These tools were chosen for their robust security

features, including strong encryption and authentication mechanisms, making them ideal for sensitive operations. Remote access allows direct manual access to the filesystem of any player (Figure 4), making it useful for troubleshooting.

```
ottol@DESKTOP-GR03PH4:~$ ssh username@customer-player-ip-address
```

Figure 4. Accessing a customer's player remotely via SSH from a local machine.

5 Player Development

This chapter outlines the development process of the player starting from the development strategy and environment, and lastly looking at the Python modules developed to achieve the required functionalities.

5.1 Development Strategy

The development strategy for the player and the services around it centers around an Agile approach, tailored to accommodate a small team working under tight deadlines. The core development team is composed of three key members, each specializing in a different aspect of the project. One member focuses on developing the server-side UI, another one develops the back-end services, and the third one develops the audio player. This organizational structure enables each major component of the project to receive focused attention.

The Agile methodology is customized to accommodate the team's size and remote work conditions. Daily stand-up meetings facilitated via video conferencing are important in maintaining project momentum and alignment. During these sessions, team members report on their progress, identify problems, and plan subsequent activities. This frequent communication is crucial for sustaining coherence among remotely located team members.

Slack serves as the primary communication platform, enabling frequent exchanges among team members. This tool is crucial for maintaining continuous communication flow, ensuring all members are promptly updated on changes and developments, thus preserving team cohesion and information synchronicity.

Engagement with the customer occurs via email when needed and on a weekly basis. These interactions allow for showcasing features, discussion of

forthcoming features, and integration of feedback from testing phases. These interactions help with the identification of software bugs, enhancement of feature sets, and alignment of project timelines with customer expectations, thereby optimizing the development lifecycle for a tight schedule.

Task management is mainly conducted using Google Docs Sheets, a web-based document management platform that allows users to create, edit, and collaborate on documents online [11]. This tool tracks progress and prioritizes tasks from low to high urgency, enabling dynamic adjustment of workloads and priorities based on evolving project requirements (Figure 5).

Feature	Affects	Feature Part	Overall Status	Priority	Feature Release Goal
Build and install	Player (lite)	Build flow for Lite player	In Production	HIGH	BETA
Build and install / Playback	Player (lite)	Audio configuration (round 1)	In Production	HIGH	BETA
Build and install	Player (lite)	Installation scripts (no auto installer) and instructions	In Production	HIGH	BETA
Build and install	Player (lite)	Linux image with autoinstaller etc	In Production	MEDI...	MVP
Player registration	Player (lite)	Resolve mac address	In Production	HIGH	MVP
Player registration	Player (lite), Server	Send mac address to server, create Device	In Production	HIGH	MVP
Player registration	Player (lite), Server	Command channel to use	In Production	HIGH	MVP
Player registration	Player (lite), Server	Attach unassigned device to customer	In Production	HIGH	MVP
Player registration	Player (lite)	Player init with customer identifiers etc	In Production	HIGH	MVP
Sync and Upgrade	Player (lite), Server	Separate upgrade files for Lite	In Production	HIGH	BETA
Sync and Upgrade	Server	Content service and server side handling for Lite upgrade	In Production	HIGH	BETA
Sync and Upgrade	Player (lite)	Upgrade on Lite player side (download, unpack and replace files)	In Production	HIGH	BETA
Sync and Upgrade	Player (lite), Server	Media sync	In Production	HIGH	BETA
Sync and Upgrade	Player (lite)	Sync local database	In Production	HIGH	BETA
Sync and Upgrade	Server	Add first / latest sync and upgrade info to PlayerStatus	In Production	HIGH	BETA / MVP
Player Management	Server	Add proper "Player" entity to the ecosystem	In Production	HIGH	BETA
Player Management	Server	Register player and license information in the Player entity (also on Toneco mainl	In Production	HIGH	BETA
Player Management	Player (lite), Server	Add player command channel to hazelcast (communication via macAddress)	In Production	MEDI...	MVP
Player Management	Player (lite), Server	Send status logs to server on player errors (uses existing PlayerStatusLog)	In Development	HIGH	MVP
Player Management	Player (lite), Server	Send & save player hardware information to server	In Development	LOW	FINAL
Player Management	Server, MusicTools	Player/Customer Management APP (MVP) to MusicTools UI	In Production	HIGH	FINAL
Player Management	Server, MusicTools, Pla	Send configurations and commands to player via MusicTools UI	In Production	MEDI...	FINAL
Player Management	Server, MusicTools, Pla	Save player configurations to server (now only available via hazelcast)	In Production	MEDI...	BETA
Player offline functionality	Player (lite)	Baseline offline functionality	In Production	HIGH	BETA
Player offline functionality	Player (lite)	Improved and stabilized offline functionality	Ready	HIGH	MVP
Player offline functionality	Player (lite)	Fully stabile offline scheduling for announcements	Ready	HIGH	MVP

Figure 5. A snippet of task management in Google Docs.

5.2 Development Environment

The development environment is designed to closely emulate the target deployment platform, the Raspberry Pi. This section describes the integrated tools used to create a suitable development environment.

A key component for setting up the development environment is utilizing WSL to mirror the Linux-based environment of the Raspberry Pi. WSL allows developers to run a Linux environment directly on Windows without needing a separate machine or dual-boot setup [12].

Python environments are managed using virtual environments to encapsulate dependencies. Dependency lists are maintained in a requirements file, used by the pip tool to manage package installations, ensuring synchronized dependencies across environments, and ensuring that code developed and tested under WSL behaves as consistently as possible when deployed to the Raspberry Pi hardware.

However, as mentioned in subchapter 4.2, the Raspberry Pi includes certain system services and scripts that may not translate perfectly due to differences in hardware architecture and system configurations between WSL and the actual Raspberry Pi device, meaning continuous testing between both devices is necessary.

5.3 Modules

This subchapter shows how the software structure encapsulates specific functionalities within distinct asynchronous modules, promoting a clean separation of concerns.

5.3.1 Database Module

The database module is dedicated to managing all SQLite database interactions, which includes executing queries, updating records, and retrieving data. It also handles converting the data from provided DDL and CSV files contained in a compressed file received from the remote server to synchronize the local SQLite database to match the remote server.

5.3.2 Main Module

The main module serves as the application's entry point, handling the initialization and coordination of all other modules. It sets the environment variables by reading some of them from a configuration file (Figure 6) in the

software's working directory, which supports easy switching between development and production environments.

```
software.version=1.7.7
BASE_URL=https://example.fi
HAZELCAST_SERVER=example.example.fi:11000
PLAYER_REGISTER_URL=https://example.fi/np/client/players/unassigned
SYNC_UPDATE_URL=https://example.fi/np/do/update
UPGRADE_SERVICE_BASE_URL=https://example.fi/np/do/upgrade
TIMEZONE=Europe/Helsinki
ENVIRONMENT=dev
JINGLE_API_KEY=123-abc-123-abc-123
```

Figure 6. Player configuration file.

The main module is responsible for initializing the system's logging facilities to track successes and failures during runtime. It also manages the player's registration with the server on the first boot, which enables the server to create and identify an instance of the player. Before advancing to the main operational phase, where it activates other modules, the module waits for a clock synchronization script to confirm synchronization through the creation of a flag file, as mentioned in subchapter 4.2.

All modules collect logs throughout the operation to record important successes and failures. An example from the first boot (Figure 7) shows an initial failed attempt by the database module to fetch configurations from a database table that does not yet exist. After creating and populating the configuration table with initial data from the server's response to a registration API request, subsequent entries show successful configuration. The execution of the clock synchronization check is also logged.

```
INFO - [Main] Player registered | Config: {'id': 1, 'mac': 'd83add2ba36e-48e4', 'stream_volu
': 'C3AN9-IUNLC-150C4-EBRS8-WEELS-YU0A2-V8H85-8VQ44-SNKJK-CI7PU-4K', 'playerId': 26, 'softwa
B Rev 1.5', 'image_version': 'PROD_v1.7.4', 'ipAddressMode': 'dhcp', 'ipAddress': None, 'gat

INFO - [Clock Sync] Starting clock synchronization check.
INFO - [Clock Sync] Clock synchronized within initial 30 seconds.
```

Figure 7. Main module logs.

5.3.3 Hazelcast Module

The Hazelcast module is essential for managing real-time interactions and synchronizing data with the server. It initializes a client and sets up key data structures such as maps for status, play logs, configurations, and synchronization messages. Upon startup, the module begins by sending periodic status messages to the server, reflecting the player's state. This process involves adding status information to the status map, which the server uses for troubleshooting and managing the player's state.

The module’s main role is to monitor various maps for server commands, ensuring real-time communication and data flow. For example, if an IP configuration is needed at startup, the module executes the IP script (Chapter 4.2, Figure 4). It also handles additional settings listed in the configuration map, such as volume or license settings. When an upgrade is necessary, a specialized helper module manages the software upgrade process. Similarly, for new data synchronization needs, another helper module downloads the necessary files and updates the database.

The module also adds entries to the play log map at specific intervals. These logs, which help track the player's history, are collected by the player module discussed in Chapter 5.3.5. All these operations are also recorded in a log file (Figure 8).

```
ite-player $ grep -E "\[Hazelcast\]" blackbox.log
 INFO - [Hazelcast] Sending periodic status updates to the server every 30
 INFO - [Hazelcast] No IP configuration update needed at startup.
 INFO - [Hazelcast] UPGRADE map entry found: True
 INFO - [Hazelcast] Processing UPGRADE update: {"payload":{"environment":"
05-08T11:51:54.913577183Z"}
 INFO - [Hazelcast] UPGRADE required: False
 INFO - [Hazelcast] UNASSIGNMENT map entry found: False
 INFO - [Hazelcast] DEACTIVATION map entry found: False
 INFO - [Hazelcast] VOLUME map entry found: False
 INFO - [Hazelcast] LICENSE map entry found: True
 INFO - [Hazelcast] LICENSE config updated | Config: {'id': 1, 'mac': 'd83
tId': 20393, 'contId': 50262, 'license': 'C3AN9-IUNLC-150C4-EBRS8-WEELS-YU
00000000000', 'deviceId': 138, 'hardware': 'Raspberry Pi 4 Model B Rev 1.5
 None, 'nameserver2': None, 'nameserver3': None, 'netmask': None}
 INFO - [Hazelcast] [Startup Sync] ZIP file processed. Size: 245128 bytes
 INFO - [Hazelcast] Initializing SYNC map
 INFO - [Hazelcast] IP map entry found: False
 INFO - [Hazelcast] Adding playlog entries
 INFO - [Hazelcast] No log entries to add. The JSON file is empty.
 INFO - [Hazelcast] Service initialized and maps created.
```

Figure 8. Hazelcast module logs.

5.3.4 Scheduler Module

The scheduler module is responsible for managing the execution of scheduled audio content. This module uses the capabilities of the APScheduler library mentioned in Chapter 3.2.

One of the primary functions of the module is to create CRON jobs. CRON jobs are scheduled commands or scripts that are executed at specified times. These jobs are needed for automating the playback of audio. For example, it might schedule a stream to begin at a specific time, followed by local advertisements at scheduled breaks (Figure 9), ensuring that content plays at the correct time and in the correct order.

The module interacts directly with the player module, by initiating stream playback, starting local MP3 playback if network conditions do not favor streaming, and managing the playback of advertisements or announcements.

A key feature of this module is the monitoring loop, which runs continuously to assess the state of the player and the active CRON jobs. This monitoring is vital

for adapting to changes in the playback schedule or resolving conflicts between scheduled tasks and current playback. By constantly evaluating the system's status, the module ensures that the application adheres to its intended operational schedule and adjusts dynamically to any changes or interruptions, like internet outages.

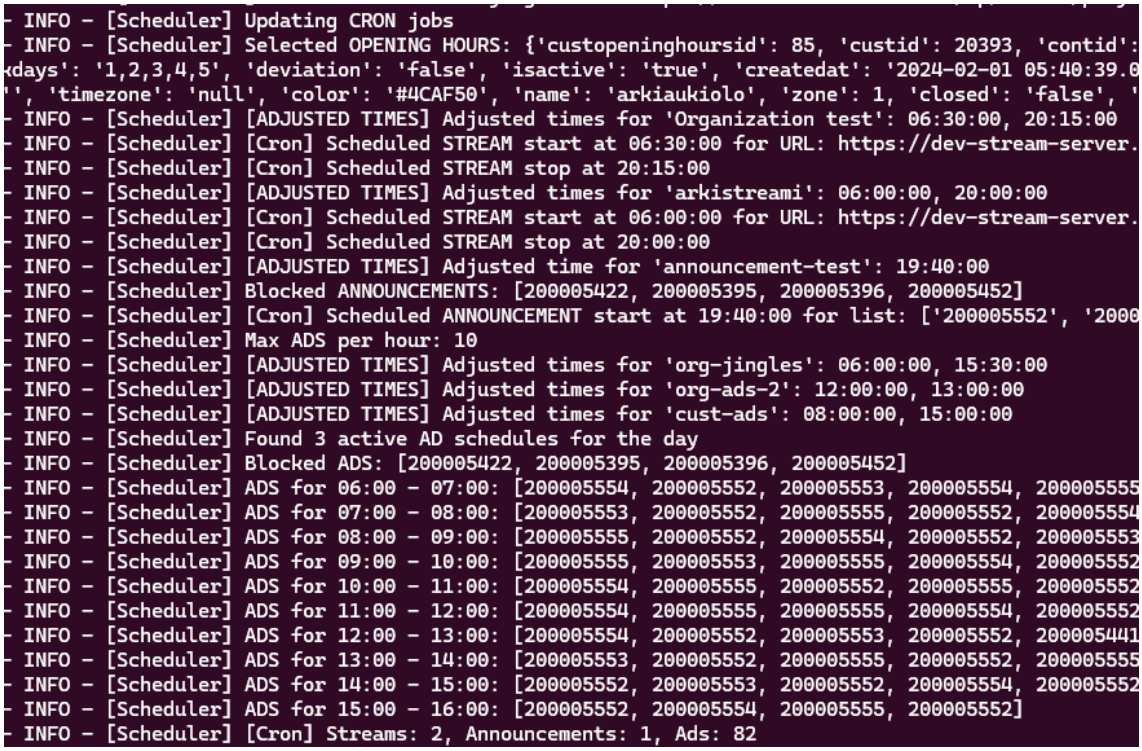

Figure 9. Scheduler module logs.

5.3.5 Player Module

The player module is dedicated to managing audio playback via the commands it receives from the scheduler module (Figure 10). It begins with the initialization of the VLC media player, setting up event listeners to track and respond to various playback events. The module supports multiple playback options, including streaming from URLs, playing royalty-free local MP3 files, and handling sequences of advertisements and announcements, also stored as local MP3 files. It also implements audio management techniques, such as adjustable volume settings and fade-in effects for smoother transitions.

```
INFO - [Player] Starting STREAM https://dev-stream-server.
INFO - [Player] Starting AD list: ['200005552']
INFO - [Player] Player state 3, Playing item: ./jingles/20
INFO - [Player] Sequence list ended.
INFO - [Player] Starting STREAM https://dev-stream-server.
INFO - [Player] Starting AD list: ['200005554']
INFO - [Player] Player state 3, Playing item: ./jingles/20
INFO - [Player] Sequence list ended.
INFO - [Player] Starting STREAM https://dev-stream-server.
```

Figure 10. Player module logs.

Moreover, the module is responsible for maintaining a detailed play log of all played items in a JSON file (Figure 11), which serves as a resource for monitoring play history.


```
[
    {
        "timestamp": "2024-05-09T05:00:02.026722Z",
        "jingleId": null,
        "streamId": 56,
        "scheduleEntryId": 65,
        "listId": null,
        "playerId": 78,
        "deviceId": 36,
        "playType": "stream"
    },
    {
        "timestamp": "2024-05-09T05:15:00.126870Z",
        "jingleId": 200005482,
        "streamId": null,
        "scheduleEntryId": null,
        "listId": null,
        "playerId": 78,
        "deviceId": 36,
        "playType": "announcement"
    },
    {
        "timestamp": "2024-05-09T05:15:25.923684Z",
        "jingleId": null,
        "streamId": 56,
        "scheduleEntryId": 65,
        "listId": null,
        "playerId": 78,
        "deviceId": 36,
        "playType": "stream"
    },
```


Figure 11. Play log entries.

5.3.6 Helper Modules

This section mentions some of the additional supporting modules supplementing the core modules discussed earlier.

The API module is designed to handle communication between the player and external services via API requests and inform about internet outages (Figure 12).

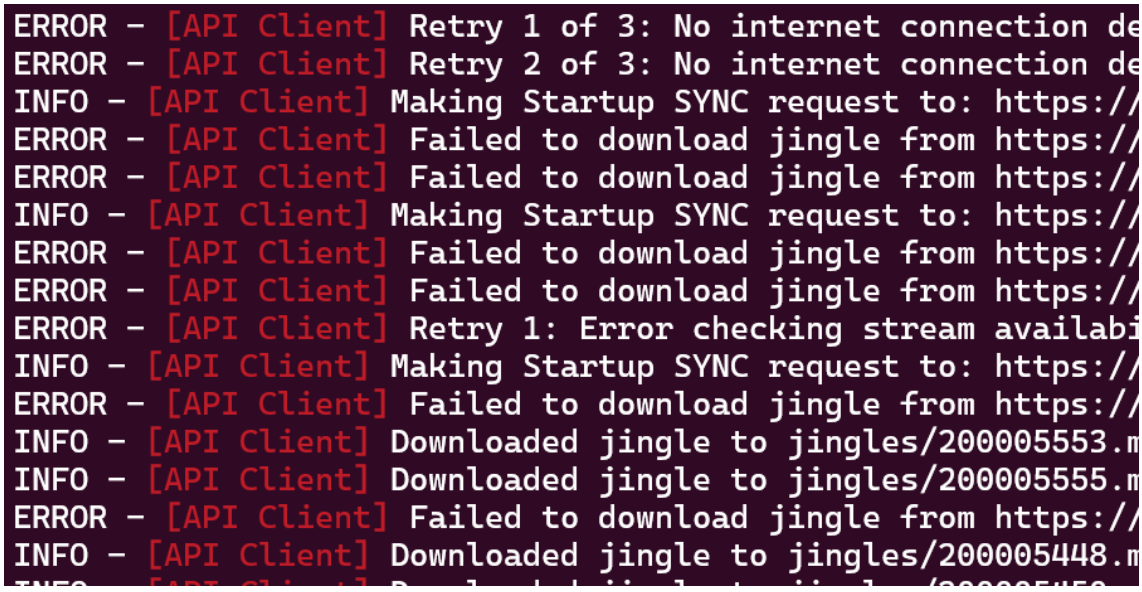

Figure 12. API module logs.

The upgrade module, invoked by the Hazelcast module, operates by constructing URLs to download specific upgrade packages from the server based on device and user identifiers, then fetches and prepares these packages for installation (Figure 13). Upon successfully retrieving and extracting the upgrade files, the handler executes a script to apply the update (Figure 14).

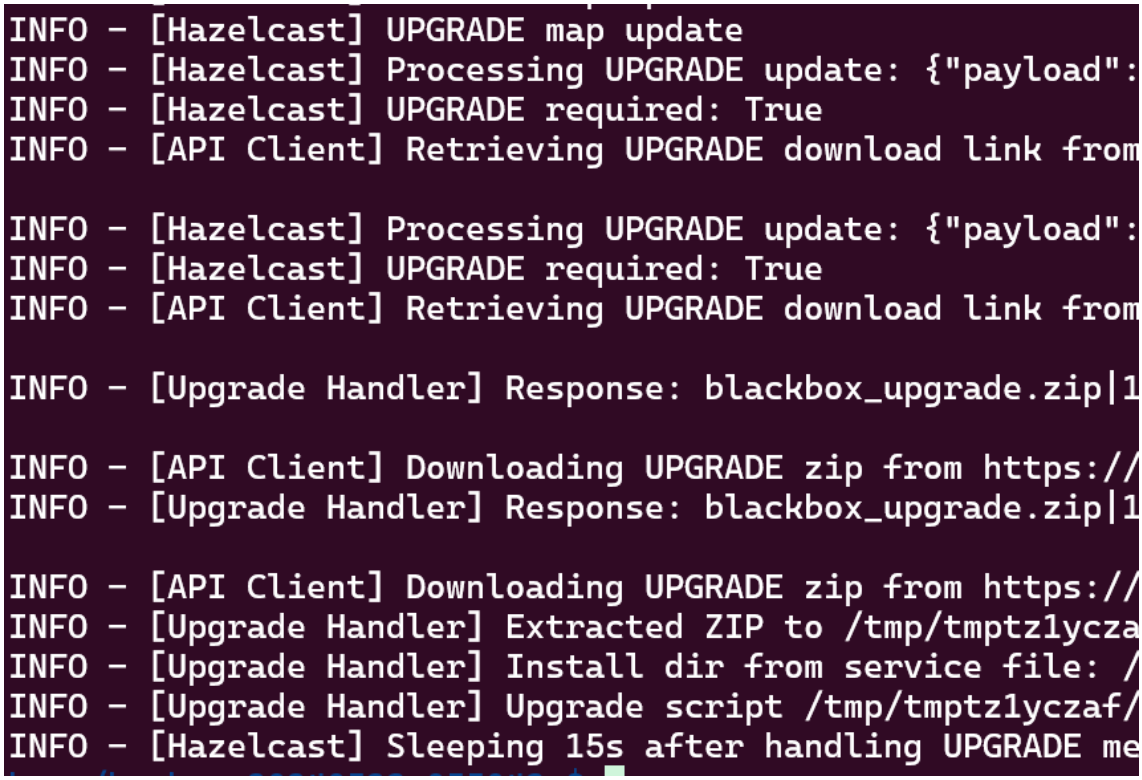

Figure 13. Upgrade module logs.

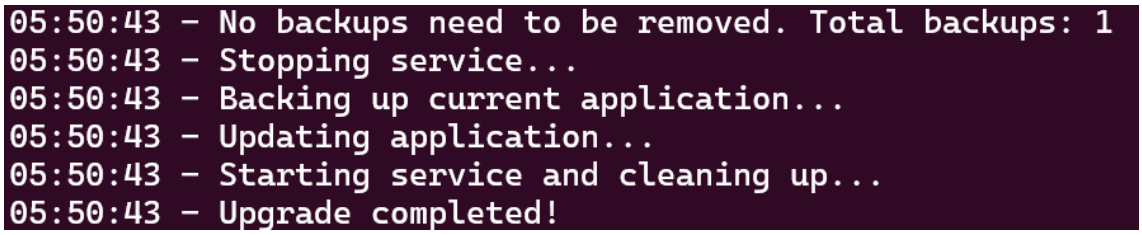

Figure 14. Upgrade script logs.

Together, these modules form an asynchronous application capable of fulfilling the customers' requirements outlined in Chapter 2 and allow easy troubleshooting and testing via the collected logs.

6 Testing

Testing is conducted using both the WSL development environment and the Raspberry Pi, ensuring that the software performs reliably in its intended environment. As shown in Chapter 5, various log files are collected to assist in testing. Testing occurs iteratively as new functionalities are developed. Initially, tests are performed within the WSL environment, with subsequent testing on the actual Raspberry Pi hardware by the developers and the customer.

Deployment to the Raspberry Pi is facilitated through SSH by moving the project files to the machine, allowing rapid testing of changes in the target environment. The configuration file (Section 5.2.3, Figure 7) is used to define if the tests are conducted in the development or production environment. This facilitates easy context switching without altering code and allows the customer to test their players in the production environment with real data.

The accurate execution of scheduled CRON jobs is tested by creating schedules, checking the scheduler module logs (Section 5.3.4, Figure 10), and listening to the player to see if it starts playing the correct schedule at the specified time.

Network resilience is tested by manually disconnecting network services to simulate outages (Section 5.3.4, Figure 10). This testing demonstrates the system's ability to switch to locally stored content, ensuring continuous playback during interruptions.

The software upgrade process is specifically tested on the Raspberry Pi to ensure new versions do not disrupt ongoing operations. Additionally, the upgrade process is designed to allow the player to continue and recover from its operations even if an update cannot be performed. This is achieved by storing a copy of the previous version of the software on the machine, allowing for rollbacks in case of issues (Section 5.3.6, Figure 15).

Database challenges, such as lock situations, arose as a problem during customer testing. These are addressed by implementing functionality to detect a

lock situation, remove the local database and its related files, and exit the software (Figure 15). The software will start again automatically by a dedicated systemd service (Section 4.2, Figure 2), and on startup, the player will request sync files from the server again and create a new fresh database with timely data.

```
ERROR - [Main] *** CRITICAL *** Detected potential database lock issue.
INFO - Deleted folder: sync_files_20240511_080147
INFO - Deleted file: player.db-journal
INFO - Deleted file: temp_sync.zip
INFO - Deleted file: player.db
INFO - [Main] Exiting.
```

Figure 15. Handling database issues.

Additionally, customer feedback is integral to the development cycle, with weekly demos provided to the customer for evaluation. The customer also informs about any bugs or anomalies by email, to allow for immediate fixes. This ongoing engagement is crucial for refining the system's resilience and responsiveness.

7 Conclusion and Future Development

This thesis has successfully demonstrated the development of an audio player specifically designed for Raspberry Pi, addressing the needs of the customer. Throughout this project, several key challenges were tackled, including the implementation of remote-control capabilities, a streamlined upgrade process, precise schedule timing calculations, and reliable operation during internet outages.

One cornerstone of this project was the adaptation of Raspberry Pi's capabilities to meet the performance demands while maintaining efficient use of resources. The choice of the selected technologies was proven successful in achieving an optimal balance between system performance and resource utilization.

Currently operational in about ten locations, the audio players are scheduled for a wider rollout by the customer. This expansion underscores the practical viability and user acceptance of the system. The feedback from these deployments will be invaluable in guiding future enhancements.

The knowledge acquired from overcoming the complexities associated with system dependencies, software logic, and error recovery has been overwhelming. Additionally, the experience gained in remote system management and resource optimization will serve as a useful asset in future projects.

In the immediate future, the project is set to evolve further. Plans are already underway to adapt the player for a different hardware platform, which will broaden the potential user base. Additionally, an enhancement of the player to support multiple audio outputs playing different schedules from a single device, catering to more complex audio scheduling needs, is underway. Continuous improvement of the software through optimization and refactoring remains a priority.

In conclusion, this thesis fulfills the initial objectives and lays a good foundation for future advancements.

8 References

[1] Opensource, "What is a Raspberry Pi?" [Online]. Available: https://opensource.com/resources/raspberry-pi. [Accessed 5 May 2024].

[2] VideoLAN, "VLC: Official site - Free multimedia solutions for all OS!" [Online]. Available: https://www.videolan.org/. [Accessed 5 May 2024].

[3] Python, "What is Python? Executive Summary" [Online]. Available: https://www.python.org/doc/essays/blurb/. [Accessed 5 May 2024].

[4] Apscheduler, "Apscheduler | Advanced Python Scheduler" [Online]. Available: https://boringowl.io/en/tag/apscheduler/. [Accessed 5 May 2024].

[5] Python, "asyncio — Asynchronous I/O" [Online]. Available: https://docs.python.org/3/library/asyncio.html. [Accessed 5 May 2024].

[6] SQLite, "About SQLite" [Online]. Available: https://www.sqlite.org/about.html. [Accessed 5 May 2024].

[7] Hazelcast, "Hazelcast Overview" [Online]. Available: https://docs.hazelcast.com/imdg/3.12/hazelcast-overview. [Accessed 5 May 2024].

[8] systemd, "System and Service Manager" [Online]. Available: https://systemd.io/. [Accessed 5 May 2024].

[9] SSH Academy, "What is SSH (Secure Shell)?" [Online]. Available: https://www.ssh.com/academy/ssh. [Accessed 5 May 2024].

[10] OpenVPN, "Business VPN For Secure Networking" [Online]. Available: https://openvpn.net/. [Accessed 5 May 2024].

[11] Google, "How to use Google Docs" [Online]. Available: https://support.google.com/docs/answer/7068618?hl=en&co=GENIE.Platform%3DDesktop. [Accessed 5 May 2024].

[12] Microsoft Learn, "What is Windows Subsystem for Linux" [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/about. [Accessed 5 May 2024].