Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет __Комп'ютерної інженерії та управління_____
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти ____другий (магістерський)_____
(рівень вищої освіти)

Модель багатоплатформового Digital Signage плеєра
(тема)

Виконав: студент 2 курсу, групи СКСм-20-1

_____Іванов М.Ю._____
(прізвище, ініціали)

Спеціальність *123 _Комп'ютерна інженерія*

Тип програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма_____

*Спеціалізовані комп'ютерні системи*_____
(повна назва освітньої прграми)

Керівник роботи проф. Свірь І.Б.__
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Чумаченко С.В.
(підпис) (прізвище, ініціали)

2021 р.

`

Харківський національний університет радіоелектроніки

Факультет_____Комп'ютерної інженерії та управління _____

Кафедра_____Автоматизації проектування обчислювальної техніки_____

Рівень вищої освіти____другий (магістерський)_____

Спеціальність_____123_ Комп'ютерна інженерія_____
(шифр і назва)

Тип програми _____Освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма _____Спеціалізовані комп'ютерні системи_____
(повна назва

ЗАТВЕРДЖУЮ:

Зав. кафедри ___ __
(підпис)

«__4__»___11____ 2021 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові_____Іванову Максиму Юрійовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи (проекту)_ Модель багатоплатформового Digital Signage плеєра
_____
_____

затверджена наказом по університету від «_04_»__11__2021 р. № 1635 Ст_____

2. Термін подання студентом роботи до екзаменаційної комісії _____20.12.2021_____

3. Вихідні дані до роботи (проекту)_____

Мова програмування C++_____

Raspberry Pi 4_____

Qt Creator IDE_____

Xibo CMS_____

_____

4. Перелік питань, що потрібно опрацювати у роботі _____

Аналіз предметної галузі та постановка задачі проектування _____

Розробка інфраструктри плеєєру та його модулів_____

Реалізація програмного забезпечення_____

_____

`

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) _14 слайдів_____
_____
_____
_____
_____

6. Консультанти розділів роботи (проекту)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

7. Дата видачі завдання 02.09.2021_____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проекту)	Термін виконання етапів роботи	Примітка
1	Отримання завдання	02.09.2021-08.09.2021	
2	Аналіз предметної області	09.09. 2021-15.09. 2021	
3	Аналіз джерел з предметної галузі	16.09.2021-29.09.2019	
4	Розробка архітектури плеєра	30.09. 2021-13.10.2021	
5	Розробка програмної частини	14.10. 2021-31.10. 2021	
6	Тестування плеєєра	01.11. 2021-15.11. 2021	
7	Оформлення пояснювальної записки	15.11. 2021-30.11. 2021	
8	Оформлення графічного матеріала	01.12. 2021-15.12. 2021	
9	Захист проекту	20.12. 2021-25.12. 2021	

Студент _____
(підпис)

Керівник роботи (проекту) _____ _____проф. Свірь І.Б.____
(підпис) (посада, прізвище, ініціали)

`

РЕФЕРАТ

Пояснювальна записка містить 70 сторінок, 29 рисунків, 2 таблиці, 12 джерел за переліком посилань.

БАГАТОПЛАТФОРМОВИЙ, DIGITAL SIGNAGE, RASPBERRY PI, МОДЕЛЬ, МУЛЬТИМЕДІА, АПАРАТНЕ ПРИСКОРЕННЯ

Метою роботи було створення моделі багатоплатформового плеєра, який відповідає сучасним потребам в інфраструктурі Digital Signage, реалізація обробки аудіо- та відеоконтенту, графічного стека в різних операційних системах з урахуванням особливостей апаратного прискорення.

В результаті роботи були розглянуті завдання по створенню багатоплатформового Digital Signage плеєра якій може працювати на Windows 10, Ubuntu Desktop та Ubuntu на Raspberry Pi. Розроблена програмна частина з усіма необхідними модулями. Плеєр був протестований з різними модулями і успішно запущений на різних платормах.

`

ABSTRACT

The explanatory note contains 70 pages, 29 figures, 2 tables, 12 sources according to the list of links.

MULTI-PLATFORM, DIGITAL SIGNAGE, RASPBERRY PI, MODEL, MULTIMEDIA, HARDWARE ACCELERATION

The aim of the work was to create a model of a multi-platform player that meets modern needs in the Digital Signage infrastructure, imlement the processing of audio and video content, the graphics stack on various operating systems taking into account hardware acceleration.

As a result, the tasks of creating a multi-platform Digital Signage player that can run on Windows 10, Ubuntu Desktop and Ubuntu on Raspberry Pi were implemnted. Software has been developed with all necessary modules. The player was tested with different modules and successfully launched on different platforms.

`

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення

РК – рідкокристалічний дисплей

HTML – HyperText Markup Language (мова розмітки гіпертексту)

HDMI – High Definition Multimedia Interface (мультимедійний інтерфейс високої чіткості)

LCD – Liquid Crystal Display (рідинно кристалічний дисплей)

UTP – Unshielded twisted pair (неекранована вита пара)

CMS – Content Management System (система керування вмістом)

IPTV – Internet Protocol Television (IP-телебачення)

SOAP – Simple Object Access Protocol (протокол обміну структурованими повідомленнями)

API – Application Programming Interface (прикладний програмний інтерфейс)

AGPL – Affero General Public License (загальна громадська ліцензія Affero)

ARM – Advanced RISC Machine (поліпшена RISC машина)

HEVC – High Efficiency Video Coding (високоефективне кодування відео)

XML – Extensible Markup Language (розширювана мова розмітки)

URI – Uniform Resource Identifier (уніфікований ідентифікатор ресурсів)

CSS – Cascading Style Sheets (каскадні таблиці стилів)

HTTP – Hypertext Transfer Protocol (протокол передачі гіпер-текстових документів)

FTP – File Transfer Protocol (протокол передачі файлів)

SMPT – Simple Mail Transfer Protocol (простий протокол пересилання пошти)

RPC – Remote procedure call (виклик віддалених процедур)

MAC – Media Access Control (управління доступом до носія)

USB – Universal Serial Bus (універсальна послідовна шина)

ВСТУП

Інфраструктура Digital Signage стає все більш популярною в наші, тому що відкриває безмежні можливості донесення різної інформації до звичайних людей. Під час, коли цифровізація ще не була настільки розповсюджена, було безліч способів донести якусь інформацію до людей, але вони відрізнялися між собою: плакати, рекламні щити, листівки тощо. У період цифровізація з'явилася можливість централізовано керувати будь-якими видами контенту. З'явилося безліч дисплеїв, починаючи від великих стендів у центрі міст, закінчуючи мініатюрними планшетами, які встановлюються на спинки крісел у літаках. Все це поєднує те, що на такі екрани та дисплеї можна виводити контент будь-якого типу, починаючи від оголошень, прогнозу погоди, курсу валют та закінчуючи звичайною рекламою. Якщо централізовано керувати та планувати весь цей контент з одного місця, а дисплеям разом із вбудованою апаратною підтримкою дати можливість показувати все це, можна побудувати велику інфраструктуру по доставці контенту, яка називається Digital Signage.

Як було зазначено, вся краса цього підходу полягає у віддаленому управлінні контентом і можливостями показувати майже все, на що здатне цільове апаратне забезбечення. Софт, який працює на даному апаратному забезпеченні та відтворює такий тип контенту називається плеєром. Такий плеєр повинен вміти працювати на пристроях різного ступеня потужності та запускатися на різних операційних системах. У рамках цієї роботи весь фокус був саме на реалізації такої моделі багатоплатформового плеєра Digital Signage.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Загадьні відомості про Digital Signage

Оскільки цифрові засоби відображення інформації все більше впливають на наше життя, можливості систем Digital Signage також розвиваються зі звичайного показу цифрових слайдів і відео, переміщаючи контент в більш просунуту площину, витягуючи, поєднуючи та комбінуючи контент з різних джерел, взаємодіючи з ним, відображає його в режимі реального часу.

Digital Signage відома як цифрова інсталяція, яка відтворює і відображає відео або інший мультимедійний контент в інформаційних або рекламних цілях. Вона може бути знайдена всюди: на цифрових світлодіодних екранах в місті, меню в кафе і ресторанах швидкого харчування на ЖК-екранах, електроному табло з розкладом на вокзалах і аеропортах і все це завдяки Digital Signage.

Digital Signage – це підкатегорія електронних екранів, наприклад LCD, LED, проєкція і «електронний папір». Однак розуміння сучасного Digital Signage вимагає більш глибокого розгляду його ролі, функцій і технічних можливостей.

Рисунок 1.1 – Digital Signage у дії

Digital Signage складається з трьох ключових компонентів:

– контент – набір аудіо, відео, зображень, графіки, тексту і багато чого іншого, зібраний, щоб розповісти унікальні історії для будь-якої кількості унікальної аудиторії в певних цілях;

– обладнання – фізичні, матеріальні пристрої та компоненти, такі як екрани (сенсорні або звичайні), медіаплеєри, медіаконтролери, комутатори та розподільники сигналів, кріплення, платіжні термінали, принтери, камери, датчики і т. д.;

– програмне забезпечення – цифрова інфраструктура, інакше CMS (Content Management System), що дозволяє створювати, розгортати, обробляти, управляти контентом і аналізувати його, розгорнутий на обладнанні.

Контент Digital Signage – це все, що відображається на екрані, включаючи текст, зображення, анімацію, відео та аудіо. Проте, «контент» може також належати до комплексного результату, створеному шляхом об'єднання різних джерел інформації. Контент – це те, що спонукає відвідувачів звернути та зафіксувати їх увагу за допомогою взаємодії з ними. Такий контент можна навіть персоналізувати в режимі реального часу за допомогою інтеграції з іншими системами, наприклад адаптуючи та прив'язуючи його зміст до певного часу або події.

Найбільш поширені сценарії використання Digital Signage:

– освіта: цифровий розклад, інформування та навчання;

– музеї: мультимедійні стенди, інтерактивна експозиція, каси, цифрові путівники та електронні таблички;

– виставки: інтерактивні каталоги продуктів і послуг, реєстрація відвідувачів;

– загальнодоступний: навігація та інформація (новини, погода, розклад, дорожній трафік і т. д.);

– корпоративний: інформування персоналу, обмін знаннями, бронювання переговорних і конференц-залів, координація нарад і зустрічей, навчання;

– маркетинг та торгівля.

Обладнання Digital Signage – це не просто відеоекран, на якому відображається контент за певним плей-листом. Існує безлічі варіантів обладнання, комбінуючи яке можна досягати своїх цілей.

Сенсорні мультитач-екрани дозволяють за допомогою дотику до поверхні екрану взаємодіяти з на екрані контентом. Існує дві найбільш популярні технології:

– ємнісна технологія – вбудовані в скло датчики, визначають місце розташування потоку струму, який потім визначається як подія торкання;

– інфрачервона технологія – коли об'єкт стикається з екраном, інфрачервоний промінь переривається, що призводить до визначення місця торкання.

Рисунок 1.2 – Digital Signage дисплей

Beacon маяки – це електронні пристрої з низьким енергоспоживанням, що передають унікальний ідентифікатор або URL-адресу. Привласнюючи маяк окремим елементам або відправляючи повідомлення, можна створити інформаційний контекстний міст між дисплеями або мобільним пристроєм.

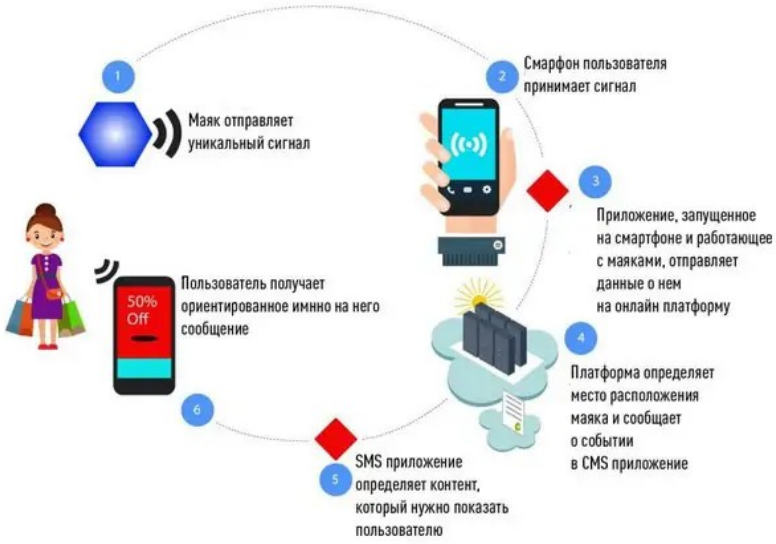

Рисунок 1.3 – Beacon маяк

RFID, NFC і інші зчитувачі – ці пристрої фіксують унікальні ідентифікатори, які можна використовувати для відтворення на екрані відповідної інформації.

Рисунок 1.4 – RFID датчики

Програмне забезпечення Digital Signage складається з чотирьох основних компонентів:

- створення та редагування контенту;
- поширення, планування та управління відтворенням контенту;
- управління медіплеєром;
- моніторинг.

Більшість представлених програмних рішень для Digital Signage являють надають систему управління контентом (CMS), пропонуючи спеціальний призначений для користувача інтерфейс (UI), що дозволяє користувачам завантажувати контент і керувати ним, який потім публікується на медіаплеєрі.

1.2 Digital Sigange плеєр

Digital Signage плеєр – це невеликий фізичний пристрій, який передає вміст на телевізор, монітор або інший цифровий дисплей. Такий медіаплеєр є важливою частиною будь-якої системи Digital Signage та має безліч варіантів використання. Він надає обчислювальні можливості й без них екран, монітор або інший дисплей не змогли б показувати вміст. Більшість таких медіаплеєрів підключаються до Інтернету, оскільки це дозволяє підключати екрани дисплея до вебсистеми керування вмістом Digital Signage (CMS). Платформи CMS для цифрових вивісок дозволяють користувачам зберігати, планувати та оновлювати контент.

Рисунок 1.5 – Digital Signage плеєр

Функції Digital Signage плеєра:

– здатність обробляти та показувати вміст на екрані, а саме відтворювати зображення, відео, графічне зображення та вміст веб-сторінок;

– підтримка декількох екранів;

– підключення до Інтернету за допомогою дротового з'єднання Ethernet або бездротове підключення через Wi-Fi.

Одним із варіантів вашого медіапрогравача є програвач цифрових вивісок на основі Raspberry Pi. Raspberry Pi – це мінікомп'ютер розміром з кредитну картку. Цей пристрій побудовано на основі технології з відкритим кодом і він має надзвичайно активну спільноту розробників та ентузіастів. За допомогою відповідного програмного забезпечення для цифрових вивісок програвач Raspberry Pi підтримує контент у форматі Full HD для зображень, відео, графічних зображень та навіть вмісту веб-сторінок у прямому ефірі.

1.3 Мета дослідження

Мета дослідження полягає в тому, щоб розробити модель багатоплатформового Digital Signage плеєра. При цьому необхідно продумати проектування та реалізацію плеєра з урахуванням особливостей платформ, під які реалізовуватиметься плеєр. У якості цільових платформ були обрані: Windows 10 і Ubuntu Desktop (обидва на Intel x86), Ubuntu на Raspberry Pi (ARM). Також необхідно врахувати обмеженість ресурсів на такій платформі як Raspberry та ефективно спроектувати компоненти, що вимагають багато ресурсів. Для цього необхідно реалізувати такі завдання:

1. Проаналізувати існуючі рішення у галузі Digital Signage
2. Проаналізувати особливості розробки графічних програм під різними операційними системами
3. Проаналізувати, як працює апаратне прискорення під різними операційними системами
4. Розробити та спроектувати архітектуру з урахуванням проведеного аналізу
5. Розробити модель кроссплатформенного Digital Signage плеєра
6. Протестувати результат на всіх платформах, під які проектувався плеєр

2 ОПИС ІНФРАСТРУКТУРИ DIGITAL SIGNAGE ТА ПЛЕЄРА

2.1 Огляд існуючих рішен в області Digital Signage

Перед тим, як перейти до розробки самого плеєра, необхідно проаналізувати існуючі рішення, а також виявити їх переваги та недоліки.

1. NoviSign – з понад 20 000 клієнтів, включаючи корпоративних клієнтів, таких як Disney та Hilton, NoviSign є відомим гравцем в індустрії програмного забезпечення для цифрових вивісок. NoviSign дозволяє користувачам створювати високоякісні дисплеї на замовлення, використовуючи готові шаблони, які можна налаштувати. Шаблони впорядковані за секторами, таки як гостинність або роздрібна торгівля, і дуже легко створити професійний дисплей незалежно від технічних можливостей.

Інтерфейс дуже простий для навігації та розділення на три окремі розділи: оголошення, списки відтворення та екрани. Інформація легко інтегрується в дисплеї через RSS-канали, віджети та програми, а весь текст і зображення також можна повністю налаштувати. Є планувальник платформи або функція списку відтворення, інтуїтивно зрозуміла система, яка дозволяє користувачам завантажувати вміст і влаштовувати, коли і де він буде відтворюватися, так само, як у Spotify або iTunes.

2. Yodeck продається як програмне рішення для цифрових вивісок, яке «народилось і виросло в Хмарі». Теоретично це має означати, що сервіс є швидким і надійним. Однак, щоб запустити плеєр з одним монітором, потрібно придбати програвач Yodeck на основі Rasberry Pi. Крім того, річні плани починаються з щомісячної плати за екран, і передплатники отримують необмежену кількість плеєрів безкоштовно.

Платформа має всі основні функції, які ви очікуєте, як-от сумісність з відео та аудіо, а також ряд віджетів, які дозволяють транслювати вміст від сторонніх постачальників, включаючи BBC і CNN. Інтерфейс Yodeck

зрозумілий, а планування моніторів особливо просте, оскільки вже наявні шаблони.

3. OnSign TV – це просте у використанні програмне забезпечення для цифрових вивісок, сумісне з кількома операційними системами, включаючи Android, Windows, Chrome OS, macOS, Phillips, Linux, Samsung SSP тощо. OnSign TV надає повний список сумісних програвачів і необхідне для них обладнання. Після налаштування технологія стає неймовірно зручною для користувачів. OnSign TV працює на основі широкого спектру безкоштовних програм, які відображають велику кількість інформації, від спортивних результатів до місцевого прогнозу погоди. Також є можливість створити власні програми в HTML5 або попросити розробників платформи зробити це за вас. Інтерфейс працює за принципом перетягування, тому можна завантажити потрібний вміст, будь то програма чи інший медіа, і запланувати його відтворення на моніторі.

4. TelemtryTV – це чудовий варіант програмного забезпечення для вивісок для користувачів із різним бюджетом, з безкоштовною версією для одного користувача, обмеженою одним пристроєм. Стандартний інтерфейс браузера добре продуманий, авторитетний і чітко позначений. Домашня сторінка містить відеоуроки, які допоможуть розпочати роботу, але інтерфейс досить зрозумілий. За допомогою вбудованого редактора можна створювати основні дисплеї.

2.2 Графічний стек в операційних системах

Під час проектування такого софту як плеєр, необхідно врахувати багато моментів, пов'язаних з візуалізацією контенту. Плеєр має показувати безліч видів контенту, проте рендеринг такого контенту може відрізнятися залежно від обраної операційної системи. Зважаючи на те, що метою проекту є створення багатоплатформового плеєра, було прийнято рішення

проаналізувати графічний стек двох найпопулярніших операційних систем: Windows та Linux.

Почнемо наш огляд с графічного стека Linux. Сьогодні все набагато складніше, ніж раніше, але в перші часи існував лише один фрагмент програмного забезпечення, який мав прямий доступ до графічного обладнання: сервер X. Цей підхід спростив графічний стек, оскільки йому не потрібно було синхронізувати доступ до графічного обладнання між кількома клієнтами.

У ці перші дні програми виконували всі свої дії непрямо, через сервер X. Використовуючи Xlib, вони надсилали команди рендерингу по протоколу X11, які сервер X буде приймав, обробляв та переводив на фактичні апаратні команди з іншого боку сокета. Переклад – це робота драйвера: він приймає купу команд відтворення та переводить їх в апаратні команди, як очікується цільовим графічним процесором. Драйвери, які були написані спеціально для X-сервера, стали модулями самого сервера та невіддільною частиною його архітектури. Ці драйвери називаються драйверами DDX, і їх роль у стеку графіки полягає у підтримці 2D-операцій, експортованих Xlib та необхідних для реалізації X-сервера.

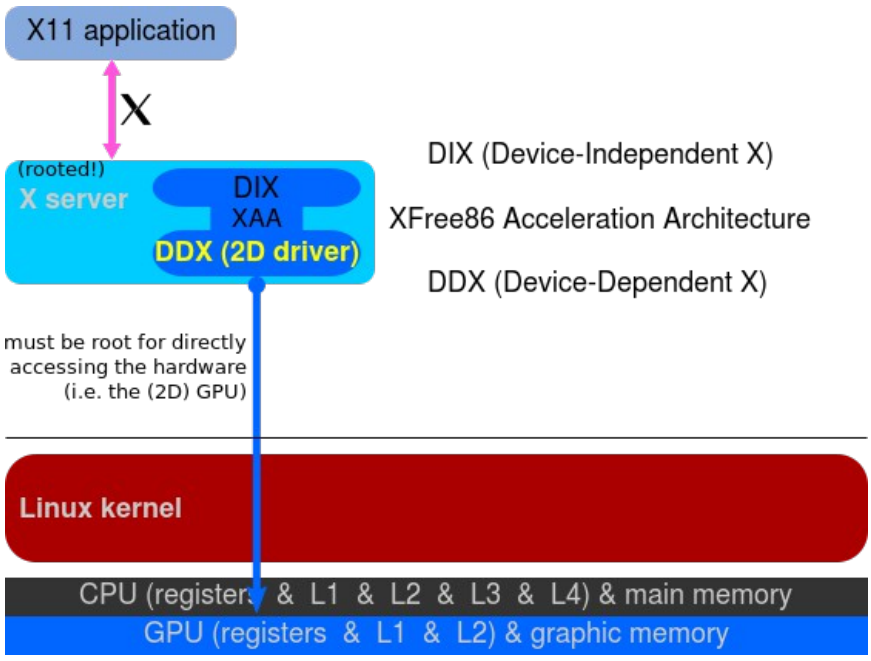

Рисунок 2.1 – DDX драйвери в X сервері

Наведене вище охоплює 2D-графіку, оскільки це був сервер X раніше. Однак поява апаратного забезпечення 3D-графіки істотно змінило сценарій. У Linux 3D-графіка реалізована через OpenGL, тому люди очікували впровадження цього стандарту, який би скористався перевагою нового 3D-обладнання, тобто апаратного прискорення libGL.so. Однак у системі, де лише X-серверу було дозволено отримувати доступ до графічного обладнання, не могло бути libGL.so, який би спілкувався безпосередньо з апаратним забезпеченням 3D. Рішення полягало в тому, щоб забезпечити реалізацію OpenGL, яка б надсилала команди OpenGL на сервер X через розширення протоколу X11 і дозволяла серверу X перетворювати їх на фактичні апаратні команди, як це робилося раніше для 2D-команд.

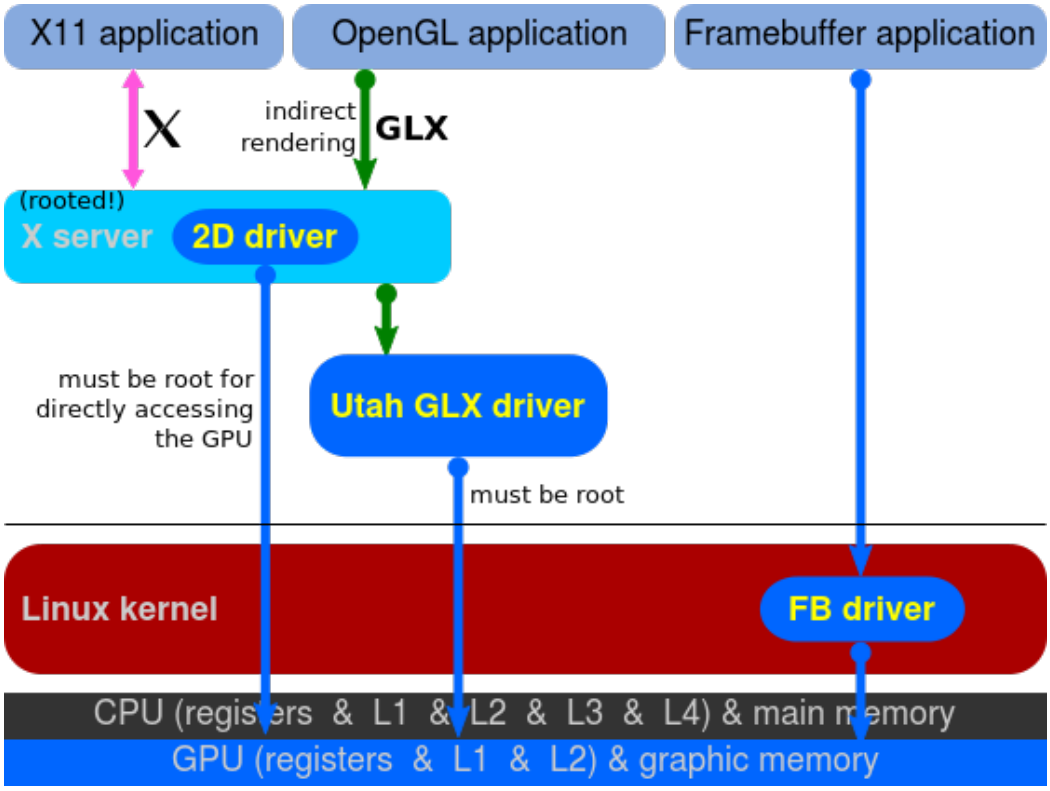

Рисунок 2.2 – OpenGL з непрямим рендерингом

Однак цього рішення недостатньо для інтенсивних 3D-додатків, які вимагають відтворення великої кількості 3D-примітивів зі збереженням високої частоти кадрів.

Інфраструктура прямого рендерингу (DRI) – це нова архітектура, яка дозволяє клієнтам X безпосередньо спілкуватися з графічним обладнанням. Іншою важливою частиною DRI є Direct Rendering Manager (DRM). Тут ядро обробляє такі важливі аспекти, як апаратне блокування, синхронізація доступу, відеопам'ять тощо. DRM також надає API, який може використовуватися для надсилання команд і даних у форматі, придатному для сучасних графічних процесорів, що ефективно дозволяє користувачу спілкуватися з графічним обладнанням. Для кожного графічного процесора існують різні драйвери DRM.

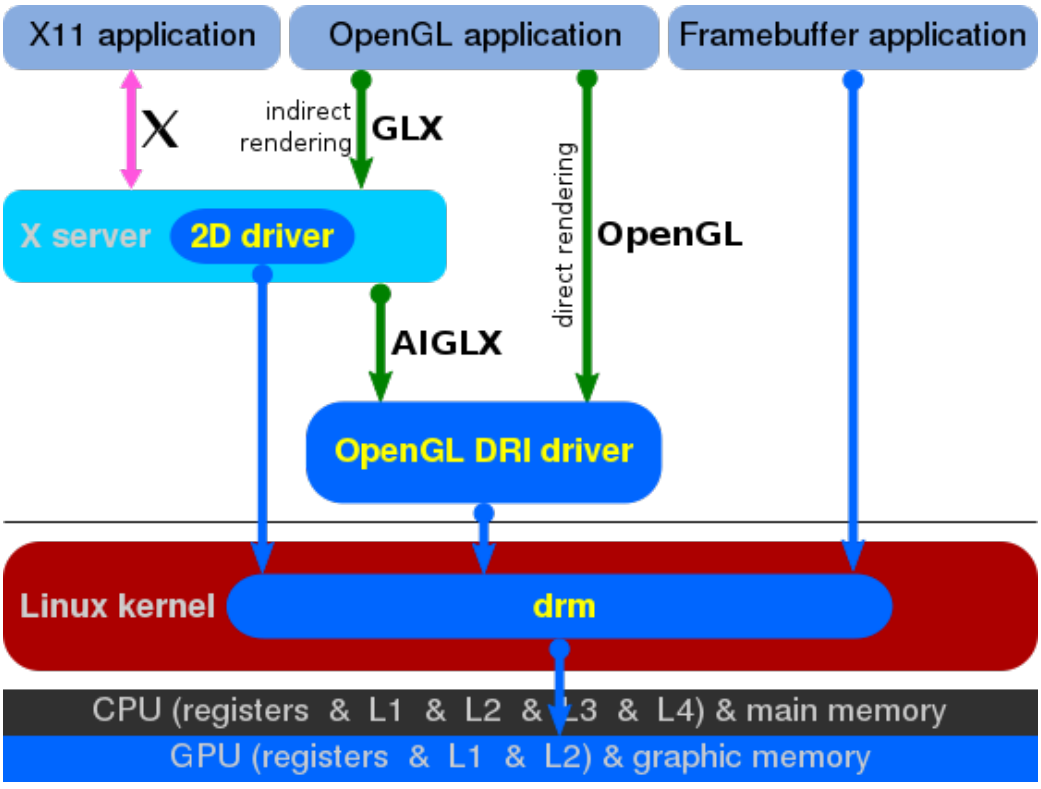

Рисунок 2.3 – OpenGL с прямим рендерингом

Для того, щоб використовувати OpenGL, потрібен ще один програмний продукт, який із використанням інфраструктури, наданої DRI/DRM, реалізує OpenGL API, поважаючи вимоги до сервера X. Mesa – це програмна реалізація специфікації OpenGL, яку програми на основі OpenGL можуть використовувати для виведення 3D-графіки в Linux. Mesa може забезпечити

прискорену 3D-графіку, скориставшись архітектурою DRI, щоб отримати прямий доступ до базового графічного обладнання під час реалізації OpenGL API.

Коли 3D-програма працює в середовищі X11, вона виводить свою графіку на вікно сервера X. З DRI це відбуватиметься без втручання сервера X, тому, між ними має бути виконана певна синхронізація, оскільки X-сервер все ще володіє вікном, на яке рендерить Mesa, і відповідає за відображення його вмісту на екрані. Ця синхронізація між додатком OpenGL та сервером X є частиною DRI. Реалізація GLA Mesa використовує DRI для спілкування з сервером X і досягнення цього.

Mesa також використовує DRM для багатьох речей. Спілкування з графічним обладнанням відбувається шляхом надсилання команд (наприклад, «намалювати трикутник») та даних (наприклад, координат вершин трикутника, їх колірних атрибутів, нормалей тощо). Цей процес зазвичай передбачає виділення буфера в графічному обладнанні, де всі ці команди та дані копіюються, щоб графічний процесор мав доступ до них та виконував свою роботу. Це забезпечується драйвером DRM, який управляє відеопам'яттю і дає API (наприклад, Mesa) для використовування в просторі користувача. DRM також необхідний, коли нам потрібно виділити та керувати відеопам'яттю в Mesa, тому такі речі, як створення текстур, завантаження даних до текстур, виділення кольору, глибини або буферів трафарету тощо, вимагають використання API DRM для цільового обладнання.

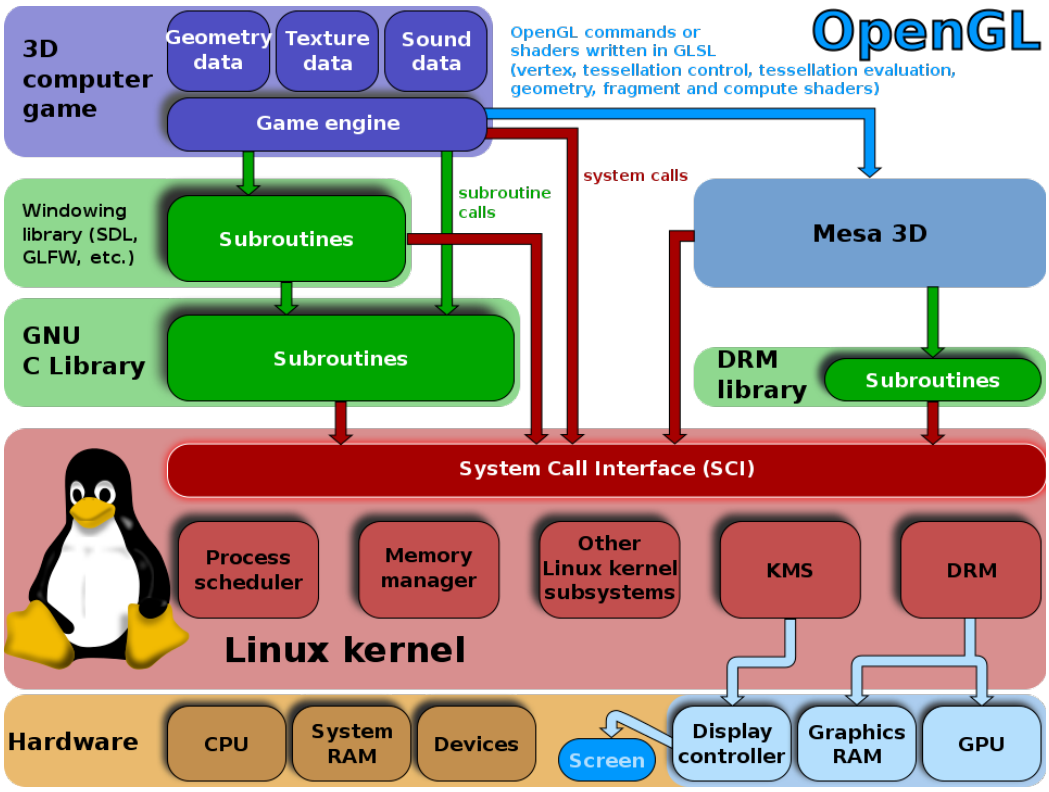

Рисунок 2.4 – OpenGL/Mesa в контексті 3D додатків

Перейдемо до графічного стека Windows. Windows надає кілька графічних інтерфейсів C++/COM:

- GDI – це оригінальний графічний інтерфейс для Windows;
- GDI+ – був представлений у Windows XP як спадкоємець GDI;
- Direct3D – для підтримки 3D-графіки;
- Direct2D – це сучасний API для 2D графіки, спадкоємець GDI та GDI+;
- DirectWrite – це макет тексту та механізм растеризації. GDI або Direct2D може бути використаний для малювання растрового тексту;
- DXGI – виконує завдання низького рівня, такі як представлення кадрів для виводу.

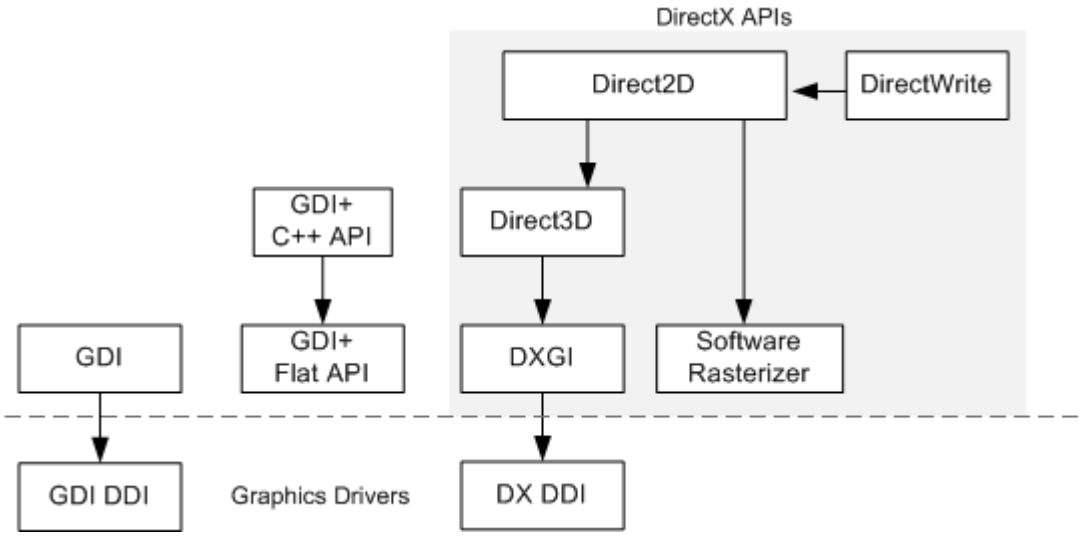

Рисунок 2.5 – Графічний стек Windows

Direct2D підтримує повністю апаратне прискорене альфа-змішування (прозорість). GDI має обмежену підтримку альфа-змішування. Більшість функцій GDI не підтримують альфа-змішування, хоча GDI підтримує альфа-змішування під час операції bitblt. GDI+ підтримує прозорість, але альфа-змішування виконується центральним процесором.

Апаратно прискорене альфа-змішування також дозволяє згладжувати. Аліасинг – це артефакт, викликаний вибіркою безперервної функції. Наприклад, коли криву лінію перетворюють у пікселі, аліасинг може викликати нерівні види. Будь-яка техніка, яка зменшує артефакти, викликані аліасингом, вважається антиаліасингом. У графіці антиаліасинг здійснюється шляхом змішування країв із фоном.

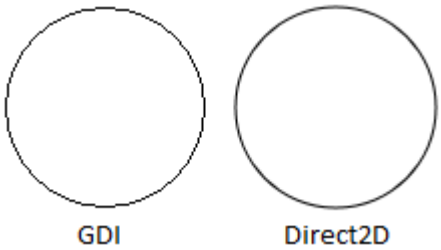

Рисунок 2.6 – Приклад аліасинга та антиалісинга

GDI не підтримує антиаліасинг, коли малює геометрію (лінії та криві). GDI може малювати текст з антиаліасингом за допомогою ClearType; але в іншому випадку текст GDI має аліасинг. Аліасинг особливо помітний для тексту, оскільки зазубрені лінії порушують дизайн шрифту, роблячи текст менш читабельним. Хоча GDI+ підтримує антиаліасинг, він застосовується центральним процесором, тому ефективність не така хороша, як в Direct2D.

Direct2D підтримує векторну графіку. У векторній графіці математичні формули використовуються для представлення ліній та кривих. Ці формули не залежать від роздільної здатності екрана, тому їх можна масштабувати до довільних розмірів. Векторна графіка особливо корисна, коли зображення потрібно масштабувати для підтримки різних розмірів монітора або дозволу екрану.

2.3 Обробка медіаконтенту

Весь контент, який передбачається показувати за допомогою плеєра, надходить ззовні. Після того, як файли і потрібні ресурси були завантажені, їх потрібно відобразити на екрані в правильному порядку. Те, яким чином майбутній контент буде розташований на екрані і скільки часу він буде там перебувати, залежить від конфігурації, яка поставляється мережею разом з ресурсами. Дана конфігурація є файлом у форматі XML і має наступну структуру:

Лістинг 2.1 – Конфігурація макета

```
<?xml version="1.0" encoding="UTF-8"?>
<lay w="1800" h="1050" colbg="white" ver="3" bakcimg="back.jpg">
   <reg id="1" uid="1" w="1500" h="150" t="40" l="30">
      <med id="1" medtype="txt" ren="home" dur="5">
      </med>
   </reg>
   <reg id="2" uid="1" w="1500" h="600" t="150" l="30">
      <med id="2" medtype="img" ren="home" dur="5">
         <opt>
            <src>second.png</src>
         </opt>
```

```
      </med>
      <med id="3" medtype="img" ren="home" dur="5">
         <opt>
            <src>third.png</src>
            <scaling>inTheCenter</scaling>
            <horAr>inTheLeft</horAr>
            <vertAl>inTheMiddle</vertAl>
         </opt>
      </med>
      <med id="4" medtype="img" ren="home" dur="5">
         <opt>
            <src>fourth.png</src>
            <scaling>inTheCenter</scaling>
            <horAr>inTheRight</horAr>
            <vertAl>inTheMiddle</vertAl>
         </opt>
      </med>
   </reg>
</lay>
```

Даний файл називається макетом (layout). Будь-який макет (layout), який був завантажений мережею з нашої CMS матиме схожу структуру. Розглянемо її докладніше. Усередині такого файлу може знаходитися тільки один макет, тому якщо у нас буде кілька макетів, то нам потрібно завантажувати кілька файлів подібного формату.

Кожному вузлу в файлі відповідає сутність в програмному коді. З огляду на те, що обробка і відтворення медіаконтенту є центральною і однією з найбільш комплексних завдань в рамках реалізації плеєра, то необхідно визначитися з інструментами, які будуть використовуватися для досягнення даної мети. Для малювання вікон і віджетів була обрана бібліотека GTK+. API даної бібліотеки надано мовою програмування C, але ми створили C++ обгортку навколо даного інтерфейсу, щоб спростити взаємодію з C++ кодом. Кожна з трьох основних сутностей (layout, region, media), яка описана в XML файлі, має власний GTK+ віджет, з яким вона асоціюється. Для сутностей верхніх рівнів (layout, region) це будуть GTK+ контейнери, а для медіа – конкретний віджет, який розміщується в відповідних контейнерах. Крім GTK+, були використані і інші бібліотеки, взаємодія з якими буде описана нижче.

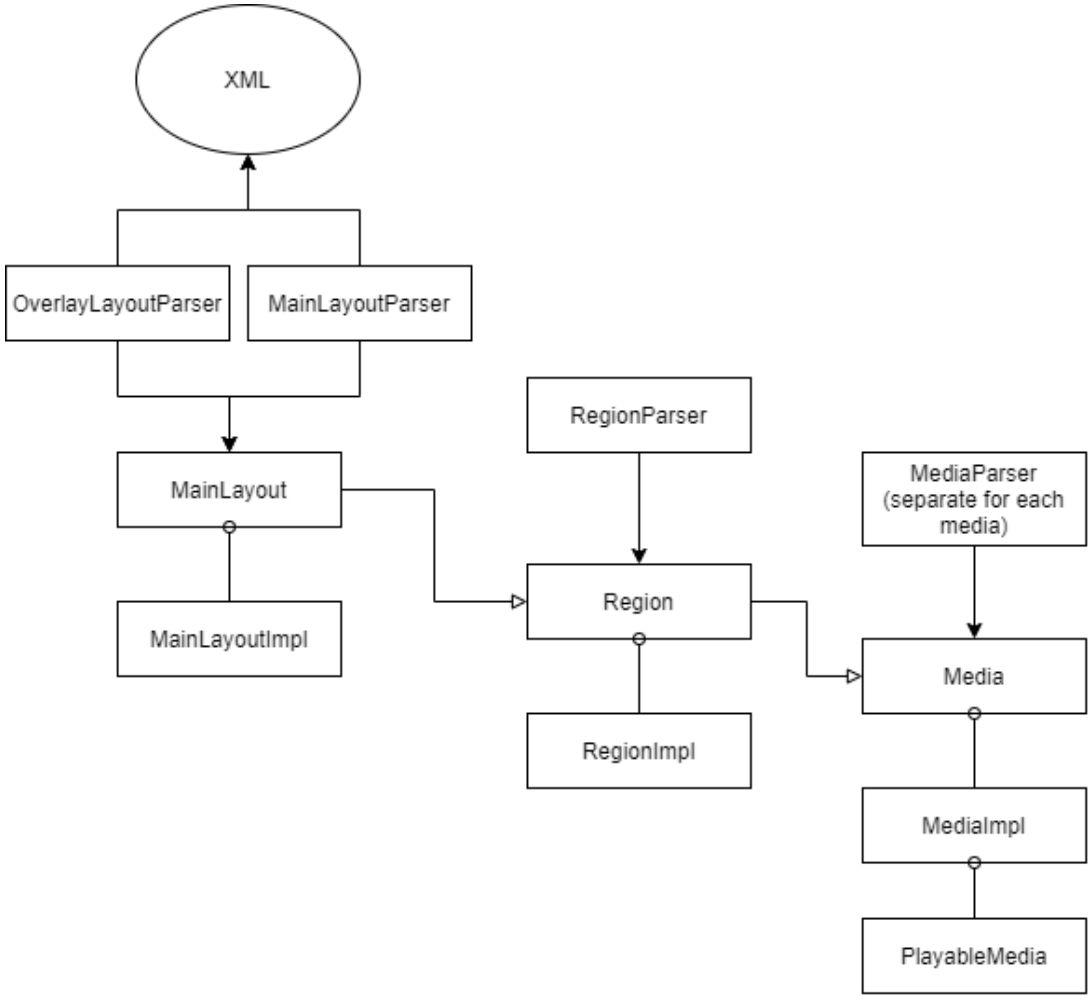

Рисунок 2.7 – Архітектура обробки медіконтенту

2.4 Огляд графічної бібліотеки GTK+

GTK+ – це бібліотека для створення графічних інтерфейсів користувача. Бібліотека створена мовою програмування C. Бібліотеку GTK+ також називають набором інструментів GIMP. Спочатку бібліотека була створена під час розробки програми маніпулювання зображеннями GIMP. З того часу GTK+ став одним з найпопулярніших наборів інструментів під Linux та BSD Unix. Сьогодні більшість програмного забезпечення графічного інтерфейсу у світі з відкритим кодом створено у Qt або GTK+. GTK+ – це об'єктноорієнтований інтерфейс прикладного програмування. Об'єктноорієнтована система створюється за допомогою системи Glib Object,

яка є базою для бібліотеки GTK+. GObject також дозволяє створювати мовні прив'язки для різних інших мов програмування. Прив’язки до мов існують для C++, Python, Perl, Java, C# та інших мов програмування.

GTK+ використовує наступні бібліотеки:

- glib – це бібліотека загального призначення, вона надає різні типи даних, рядкові утиліти, дозволяє повідомляти про помилки, вести журнал повідомлень, працювати з потоками та інші корисні функції програмування;
- pango – це бібліотека, яка дає можливість інтернаціоналізації;
- ATK – це інструментарій доступності, він пропонує інструменти, які допомагають людям з обмеженими фізичними можливостями працювати з комп’ютерами;
- GDK – це обгортка навколо низькорівневих функцій малювання та вікон, що надаються базовою графічною системою, він обробляє примітивні малюнки, растрову графіку, курсори, шрифти, а також події вікна та функцію перетягування;
- GDKPixbuf – це інструментарій для завантаження зображень та маніпулювання піксельним буфером;
- cairo – це бібліотека для створення 2D векторної графіки.

2.5 Опис основних сутностей плеєра

Для того, щоб перевести приклад у форматі XML в програмний код, необхідно представити кілька додаткових сутностей для обробки контенту. Центральна сутність, яка одночасно є і кореневою, називається MainLayout. Вона є і контейнером для сутностей іншого виду – регіонів. Так як ця сутність є центральною, то на ній лежить обов'язок відслідковувати свій внутрішній стан, а також стан усіх внутрішніх контейнерів, що є її частиною. Коли макет закінчує виконання, він повідомляє про це за допомогою сигналу expired(). Крім цього, він також збирає статистику про себе і про внутрішні контейнери, що повідомляється за допомогою сигналу statReady().

Лістинг 2.2 – Інтерфейс класа MainLayout

```
class ML
    {
    public:
        virtual ~ML() = default;

        virtual void bgSet(ShPtr<Img>&& back) = 0;
          virtual void regAdd(UnPtr<Reg>&& reg, Coord l, Coord t, int
zCoord) = 0;
        virtual LayExpEvent& expEv() = 0;
        virtual LayReadyEvent& statCollected() = 0;
        virtual LayStatReadyEvent& medStatCollected() = 0;
        virtual void again() = 0;
        virtual ShPtr<BaseWidg> getView() = 0;
        virtual int UID() const = 0;
    };
```

Спускаючи ієрархією нижче, наступними йдуть регіони. Регіони – це те, що знаходиться у макеті. На відміну від кореневого компонента регіонів може бути кілька. Кожен регіон є контейнером для медіаконтенту. Т.к. регіонів може бути кілька, у них є координати, які задають розташування на макеті.

Лістинг 2.3 – Інтерфейс класа Region

```
class Reg
    {
    public:
        virtual ~Reg() = default;

        virtual void mediaAddTo(UnPtr<Med>&& media) = 0;
        virtual void regBegin() = 0;
        virtual void regEnd() = 0;
        virtual RegExpEv& expEv() = 0;
        virtual const ListOfMed& mediaList() const = 0;
        virtual ShPtr<BaseWidg> getView() = 0;
    };
```

Медіа – це кінцевий елемент ієрархії. Цей термін досить загальний, але в межах даного макета він означає будь-який вид контенту (видимий і невидимий), який може відтворюватися плеєром. На даний момент у плеєрі підтримуються такі види контенту: зображення, веб-контент, аудіо та відео. Компонуючи їх по-різному і поміщаючи в різні регіони, можна досягти практично будь-якого макета.

Цей клас надає спільний інтерфейс взаємодії з будь-яким медіа, не прив'язуючись до конкретної реалізації. Кожен медіа має унікальний ідентифікатор. Будь-яке видиме медіа, тобто те, що має view(), може бути розташоване по-різному згідно з властивостями align та valign. Більш того, у кожного медіа може бути ще одне вкладене медіа, якщо це необхідно. Також збір статистики, який контролюється на кілька рівнів вище, можна включати та вимикати індивідуально для кожного медіа модуля.

Лістинг 2.4 – Інтерфейс класа Media

```
class Med
    {
    public:
        virtual ~Med() = default;

        virtual void widgSet(const ShPtr<BaseWidg>& widg) = 0;
        virtual void setAdditionalMedia(UnPtr<Med>&& med) = 0;
        virtual bool isActive() const = 0;
        virtual void medStart() = 0;
        virtual void medEnd() = 0;

        virtual void setActiveStat(bool enable) = 0;
        virtual bool isStatActive() const = 0;
        virtual int id() const = 0;

        virtual MedEndEvent& endEvent() = 0;
        virtual StatCollectedEvent& statCollectedEv() = 0;

        virtual MedPos::HorAl horAl() const = 0;
        virtual MedPos::VertAl vertAl() const = 0;
        virtual ShPtr<BaseWidg> getView() = 0;
    };
```

2.6 Опис парсера

До цього часу був описаний лише формат зберігання кінцевого макету, який має бути відтворений у плеєрі. Для того, щоб перевести цей формат у програмний код і ті сутності, які плеєр знає та розуміє, необхідно написати відповідні парсери. Парсери працюють так само, як і влаштований початковий формат даних. Кореневим елементом є парсер основного макету, який уже запускає парсери регіонів. Парсери регіонів, у чергу, запускають

парсери медіа модулів. Завдяки такому ланцюжку, плеєр може ініціалізувати всі необхідні компоненти.

Лістинг 2.5 – Інтерфейс класа MainLayoutParser

```
class ParserML
{
public:
    virtual ~ParserML() = default;
    struct Excep : BaseExcep
    {
         Excep(const std::string& area, int uidLay, const std::string&
msg);
    };
    UnPtr<BaseML> MLParseById(int uidLay);
protected:
    UnPtr<BaseML> getLay(const PropTree& xml);
    MLOpts getOpt(const PropTree& xml);
    Opt<Src> getBackSrc(const PropTree& xml);
    ClrHex getBackClrHex(const PropTree& xml);

    virtual ShPtr<BaseImg> backGet(const MLOpts& opts);
    void regSuppl(BaseML& lay, const PropTree& xml);

private:
    int uidLay_;
};
```

Після того, як всі необхідні макети були переведені в той формат, який необхідний плеєру, вони додаються в спеціальний менеджер, який використовує планувальник для подальшого планування та показу всіх відповідних макетів.

Лістинг 2.6 – Інтерфейс класа ManagerOfLayouts

```
class ManagerOfLayouts
{
public:
         ManagerOfLayouts(LayoutTimeSched&  timeSched,  CollectorStat&
statColl, LocalCache& fc);
    void retrieveML();
    void fetchOverlays();
    MLLoadedEvent& mlLoadedEv();
    SupplLayoutsLoadedEvent& supplLaysLoadedEv();
};
```

2.7 Опис медіа віджетів

Робота з медіа модулем ділиться на дві частини. Перша частина полягає у визначенні сутності, яка знаходиться в ієрархії та є частиною регіону. Ця сутність не робить ніяких припущень про те, який тип контенту можна показувати плеєром, але при цьому вона визначає базовий мета-функціонал, який буде доступний будь-якому типу контенту. Вона вже була описана у попередніх підрозділах. Друга частина полягає у визначенні безпосередньо медіа віджетів, які і будуть малюватись на екрані та визначати, який вид контенту на даний момент повинен відтворюватися.

На даний момент у плеєрі підтримується 4 види медіа: зображення, веб-контент, аудіо та відео. Найскладніше в реалізації є аудіо та відео, адже це найбільш витратний контент з погляду обчислювальних ресурсів, тому вони будут розглянуті окремою

Почнемо із зображень. Даний модуль легко реалізується за допомогою GTK+ віджету і не вимагає додаткової обробки. У плеєрі є 2 види зображень: однотонні картинки, які просто заповнені одним кольором, та зображення, які завантажені з якогось файлу.

Лістинг 2.7 – Інтерфейс класа Image

```
class Img : public BaseWidg
    {
    public:
        enum class AspectRat
        {
            NotAspect,
            Aspect
        };

        virtual void imgFillWith(const ClrHex& clr) = 0;
          virtual void loadFrom(const Src& src, AspectRat ratAspect) =
0;
    };
```

Далі ми розглянемо модуль малювання веб-контенту. Даний модуль повинен малювати як нескладні сторінки HTML, так і відкривати звичайні веб-сайти в інтернеті. Тобто фактично нам необхідно вбудувати невеликий веб-браузер у плеєр. Як інструмент для реалізації цієї ідеї необхідно було

знайти той, який легко інтегрується з GTK+, адже це основний інструмент побудови віджетів та малювання всіх вікон усередині плеєра. Таким чином, був обраний WebKitGTK.

WebKitGTK – це повнофункціональний порт механізму візуалізації WebKit, який підходить для проектів, які потребують будь-якої веб-інтеграції: від гібридних додатків HTML/CSS до повноцінних веб-браузерів. Він пропонує повну функціональність WebKit і корисний у широкому діапазоні систем від настільних комп'ютерів до вбудованих систем, таких як телефони, планшети та телевізори. Після додавання підтримки WebKit2 можна створювати програми, які використовують веб-платформу з підвищеною безпекою та швидкістю реагування. Підтримується повнуаінтеграцію відео у вміст сторінки та елемент HTML canvas. WebKitGTK може використовувати графічний процесор для забезпечення плавного компонування та прокручування сторінок, а також тривимірних перетворень CSS і 3D HTML-канваса (відомого як WebGL). Це робить WebKitGTK придатним для цілого ряду ігор і додатків для візуалізації.

Лістинг 2.8 – Інтерфейс класа WebView

```
class ContentWeb : public BaseWidg
    {
    public:
        enum BackTransp
        {
            NotTransp,
            Transp
        };

        virtual void restart() = 0;
        virtual void webpageFrom(const Src& src) = 0;
        virtual void backTranspEnbl() = 0;
    };
```

Робота з аудіо та відео є найбільш складною, адже такий вид контенту дуже вимогливий до ресурсів і необхідно використовувати всі обчислювальні потужності комп'ютера, щоб досягти максимальної продуктивності. На початку підрозділу було зазначено, що робота з медіа модулем ділиться на 2 частини. До цього часу ми описували лише медіа віджети, а основний медіа

компонент залишався таким самим для всіх віджетів. Однак ситуація з аудіо та відео відрізняється, адже тут тривалість показу контенту не статична і визначається вмістом самого контенту, тобто тривалістю аудіо або відео доріжки. Щоб плеєр зміг підтримувати цю поведінку, необхідно розширити існуючу реалізацію медіа компонента.

Лістинг 2.9 – Інтерфейс класа PlayableMedia

```
class MedCanBePlayed : public BaseMed
{
public:
        MedCanBePlayed(const MedOpts& opts, UnlPtr<BaseMedPlayer>&&
player);

protected:
    void evBegin() override;
    void evEnd() override;

private:
    void medFinished(const MedOpts& opts);
    void evMedFinished();

private:
    UnPtr<BaseMedPlayer> medPlay_;
};
```

Для реалізації більш спеціалізованого медіа компонента необхідно впровадити додаткову сутність медіа програвача. Ця сутність буде безпосередньо взаємодіяти з аудіо та відео потоком, внутрішніми кодеками та здійснювати рендеринг відео на віджет GTK+. Щоб зрозуміти, які кодеки будуть підтримуватись, необхідно проаналізувати, які кодеки можуть бути використані. Спочатку розглянемо відеокодеки.

Відеокодек стискає необроблений відеофайл, щоб його можна було легко транслювати онлайн. Він складається з кодера і декодера. Кодер стискає аналоговий відеофайл, тоді як декодер розпаковує відеофайл на пристрої, щоб підготувати відео до відтворення. Відеокодеки працюють таким чином, що застосовують алгоритми для стиснення відеофайлу у відповідний формат контейнера. Це полегшує зберігання та передачу файлу, оскільки стиснутий файл має значно менший розмір, ніж необроблений

формат. Коли відеофайл знаходиться на цільовому пристрої, відеофайл розпаковується, щоб користувач міг його переглянути. Необроблені відео та аудіофайли мають дуже великий розмір. Це ускладнює потокове передавання через Інтернет, оскільки для відтворення відео буде потрібно багато пропускної здатності та пам’яті.

Щоб будь-який відеокодек працював, йому потрібно стискати кадри. Існує два типи стиснення кадрів – міжкадрове та внутрішньокадрове стиснення. За допомогою внутрішньокадрового стиснення кожен кадр стискається окремо. Це також можна вказати як стиснення зображення, яке застосовується до кожного кадру відео. Міжкадрове стиснення ідентифікує всі статичні частини в кадрах і знову використовує їх у наступних кадрах.

Вибір відеокодека залежить від того, на яких пристроях відео буде транслюватися. Розглянемо найпоширеніши кодеки.

H.264/AVC – це найбільш широко використовуваний відеокодек для потокового відео. Він підтримується всіма пристроями. При однаковому розмірі файлу він дає набагато кращу якість відео, ніж попередні відеокодеки.

H.265 кращий за H.264 з точки зору того, що він дає вдвічі менший бітрейт, і добре підходить для відтворення відео з високою роздільною здатністю. Але у нього є недолік: H.265 використовує приблизно втричі більше ресурсів, ніж H.264

AV1 користується величезним успіхом у багатьох виробників пристроїв, платформ смартфонів, настільних операційних систем і браузерів, які його підтримують

VP9 – це безплатний відеокодек, розроблений Google. Він підтримується всіма продуктами Google, наприклад Android, Chrome і YouTube. З точки зору продуктивності, він забезпечує кращу якість відео при тому ж бітрейті, ніж H.265, але він не підтримується пристроями Apple.

Як тільки буде визначено найкращий відеокодек для доставки вмісту через потоковий канал, сервер стисне відео RAW у бітові потоки та

передасть бітові потоки користувачеві через безпечне з'єднання HTTP. Формат бітового потоку варіюється від одного відеокодека до іншого. Як тільки потоковий сервер встановлює зв'язок з пристроєм користувача, декодер кодеку на пристрої користувача почне відтворювати відео з бітових потоків.

Тепер розглянемо аудіокодеки.

Аудіо з втратою – це техніка стиснення, яка не розпаковує аудіофайли до їхньої початкової кількості даних. Методи втрат забезпечують високий ступінь цифрового стиснення, що призводить до зменшення розміру файлів. У цих випадках деякі звукові хвилі видаляються, що позначається на якості звуку в аудіофайлі.

Аудіо без втрат – це техніка стиснення, яка декомпресує аудіофайли до їх початкової кількості даних. Методи без втрат можуть забезпечити високий ступінь цифрового стиснення, але немає втрат у розмірі або якості звуку. Музичні формати стиснення без втрат включають FLAC, ALAC та WMA Lossless.

Нестиснене аудіо – це аудіофайл, до якого не застосовується стиснення. Звук у нестиснених аудіофайлах залишається таким самим, як і під час запису. Приклади включають формати PCM, AIFF та WAV.

Розглянемо формати аудіокодеків.

FLAC забезпечує аудіо якості CD у розмірі файлу, меншому за фактичний компакт-диск. Цей формат не сумісний з усіма пристроями та програмним забезпеченням, але він чудово звучить через Bluetooth та потокове передавання.

WAV – це нестиснений аудіоформат, який чудово використовувати, якщо потрібно отримати оригінальний записаний матеріал без втрати якості звуку. Звук у цих файлах чіткий.

MP3 є одним з найпопулярніших аудіокодеків. У файлах MP3 використовується стиснення з втратами, що значно ущільнює звук. Цей

метод стиснення є універсальним і працює практично на всіх пристроях відтворення.

WMA – як правило, ці файли менші за їх незжаті аналоги та за функціональністю подібні до файлів MP3 та FLAC. Хоча WMA пропонує універсальність, вона не сумісна з усіма пристроями, особливо з пристроями Apple.

ALAC звучить ідентично оригінальному записаному аудіо, але він стискається до меншого розміру без відкидання бітів. ALAC працює переважно з продуктами Apple, що робить формат трохи обмеженим для людей без пристроїв iOS.

Ogg Vorbis забезпечує винятковий звук при більш низькій швидкості передачі даних, ніж інші формати з втратами. Єдиний недолік Ogg Vorbis полягає в тому, що він стискає аудіо та відкидає дані для менших розмірів файлів. Однак він швидко передає звук і чудово звучить через Bluetooth.

AAC – це ще один кодек з втратами, який надає невеликі аудіофайли та відмінно працює для потокової передачі в режимі онлайн. Файли AAC не корисні, якщо вам потрібна майже репліка оригінального запису, оскільки біти відкидаються. Розмір стисненого файлу ідеально підходить для мобільних пристроїв.

AIFF – це аудіофайл, який повністю не стиснений і може відтворюватися як на комп'ютерах Mac, так і на ПК. AIFF ідентичний аудіо якості CD, але його великі файли збільшують час завантаження і займають значний простір, роблячи цей формат менш ідеальним для портативних пристроїв. Потік в AIFF можливий, але він не використовується широко.

DSD – це нестиснений аудіоформат надзвичайно високої роздільної здатності. DSD навіть вищий за якістю, ніж формати завантаження CD та HD, такі як FLAC та ALAC. Оскільки це високоякісний аудіокодек, для відтворення файлів DSD часто потрібні цифрові аудіоконвертери, якщо вони не сумісні з типовим комп’ютером. DSD є вимогливим кодером, що робить

його недоцільним для потокової передачі. Однак це найкращий звук, який можна отримати через Bluetooth.

Лістинг 2.10 – Інтерфейс класа MediaPlayer

```
class BaseMedPlayer
    {
    public:
        virtual ~BaseMedPlayer() = default;

        virtual void fileFrom(const Src& src) = 0;
        virtual void volSet(int vol) = 0;
        virtual void setScaling(ScalingType type) = 0;
        virtual void videoWindowEnabled() = 0;
        virtual void videoWindowDisabled() = 0;
         virtual void videoWindowSet(const ShPtr<WinVideo>& winVideo) =
0;
        virtual const ShPtr<WinVideo>& videoWindowGet() const = 0;

        virtual void beginVideo() = 0;
        virtual void endVideo() = 0;
        virtual MedFinishedEv& finishedEvent() = 0;
    };
```

Як було зазначено вище, щоб досягти максимальної продуктивності, необхідно використовувати всі ресурси комп'ютера. Один із таких способів – апаратне прискорення.

Апаратне прискорення – це процес, за допомогою якого програма розвантажуватиме певні обчислювальні завдання на спеціалізовані апаратні компоненти в системі, забезпечуючи більшу ефективність, ніж це можливо у програмному забезпеченні, яке працює лише на центральному процесорі загального призначення.

Апаратне прискорення поєднує в собі гнучкість процесорів загального призначення, таких як центральні процесори, з ефективністю повністю налаштованого обладнання, такого як графічні процесори та ASIC, що підвищує ефективність на порядок. Наприклад, процеси візуалізації можуть бути завантажені на відеокарту, щоб забезпечити швидше та якісніше відтворення відео та ігор, а також звільнити центральний процесор для виконання інших завдань.

Розглнеямо найбільш поширене обладнання для прискорення.

GPU – спочатку розроблені для обробки руху зображення, зараз GPU використовуються для обчислень, що включають величезну кількість даних, прискорюючи частини програми, а решта продовжує працювати на ЦП. Величезний паралелізм сучасних графічних процесорів дозволяє користувачам миттєво обробляти мільярди записів.

FPGA – напівпровідникова інтегральна схема, визначена мовою опису обладнання (HDL), призначена для того, щоб дозволити користувачеві налаштувати значну більшість електричних функцій.

ASIC – інтегральна схема, спеціально налаштована для певної мети або застосування, що покращує загальну швидкість, оскільки вона зосереджена виключно на виконанні однієї функції.

Програмне прискорення належить до техніки реалізації максимально можливих системних функцій у програмному забезпеченні та делегування критично важливих функцій до спеціалізованого зовнішнього обладнання з метою скорочення часу виконання програми. Хоча програмне прискорення є вигідним для обмеженої кількості спеціальних застосувань, поява сучасних інструментів, таких як FPGA та ASIC, скасували обмеження апаратного прискорення до повністю фіксованих алгоритмів, роблячи апаратне прискорення вигідним для більш широкого кола інтенсивних завдань.

За допомогою сучасних відеокарт часто можна розвантажувати завдання з кодування та декодування відео з ЦП, щоб зменшити споживання електроенергії та зробити більше ресурсів доступними для решти системи. Однак для цього розвантаження потрібне як апаратне забезпечення, так і програмне забезпечення, і останнє за останні роки зазнало значної еволюції. Підтримка апаратного прискорення відео в Linux розділена на різні API з різними рівнями підтримки.

Розглянемо три основні API, які використовуються.

VA-API – підтримується в Intel, AMD та NVIDIA (лише через драйвери з відкритим кодом Nouveau). Широко підтримується програмним

забезпеченням, включаючи Kodi, VLC, MPV, Chromium та Firefox. Основне обмеження – відсутність будь-якої підтримки у власних драйверах NVIDIA.

VDPAU – повністю підтримується на AMD і NVIDIA (як фірмових, так і Nouveau). Підтримується більшістю настільних програм, таких як Kodi, VLC та MPV, але взагалі не підтримує Chromium або Firefox. Основними обмеженнями є погана та неповна підтримка Intel та непрацююча робота з браузерами для прискорення вебвідео.

NVENC/NVDEC – фірмовий API, підтримуваний виключно NVIDIA. Підтримується лише в кількох основних додатках (FFmpeg та OBS Studio для кодування, FFmpeg та MPV для декодування). Основним обмеженням є обмежена підтримка програмного та апаратного забезпечення по всій платі через його фірмовий характер.

Лістинг 2.11 – Апаратне прискорення за домопогою OpenGL плагіна

```
sinkForVideo_= gst_element_factory_make("openglsink", "sinkforgtk");
binForSink_ = gst_element_factory_make("openglsink", "openglsink");

g_object_set(binForSink_, "sink", sinkForVideo_, nullptr);
g_object_set(mainElement_, "video-sink", binForSink_, nullptr);

GtkWidget* widgetVidSink = NULL;
g_object_get(sinkForVideo_, "widget", &widgetVidSink, nullptr);
videoWin_ = createShPtr<VideoWinGtk>(Glib::wrap(widgetVidSink));
g_object_unref(widgetBidSink);
```

2.8 Мережевий модуль

Ще один модуль, який важливо розглянути в рамках реалізації моделі програвача, є мережевий модуль. Без мережного модуля плеєр практично нічого не може, адже весь контент, який він отримує, надходить через мережу. Щоб реалізувати передачу даних через мережу, необхідно зрозуміти, які є цілі та завдання у плеєра, а потім вибрати необхідні протоколи для їх реалізації. Взаємодія з мережею буває наступною:

– інтервал збору мета-даних;

– завантаження файлів із зовнішніх серверів;

– механізм спілкування швидкими повідомленнями, щоб виконати якусь необхідну операцію позачергово.

Як видно, всі 3 завдання є абсолютно різними, тому для реалізації кожного необхідно різні інструменти. Ми розглянемо кожне з цих завдань у деталях, щоб ухвалити рішення щодо подальшої реалізації.

Почнемо з інтервалу збору мета-даних. Це процес обміну статусом із сервером, який здійснюється з певною частотою, яка задається в налаштуваннях. В рамках цього інтервалу плеєр повідомляє про свій стан та передає всю необхідну інформацію, яка потрібна серверу. Сервер, у свою чергу, передає додаткову мета-інформацію програвачу. Розглянемо список запитів, які використовуються в межах цього інтервалу.

RegisterDisplay – найважливіший запит, який дозволяє зареєструвати плеєр на сервері, передати всі необхідні ключі шифрування та обмінятися найважливішими даними, без яких подальші запити мають сенс. Якщо плеєр вже зареєстровано на сервері, плеєр робить авторизацію.

RequiredFiles – не менш важливий запит, який запитує список файлів, необхідних для завантаження. Цей список містить весь майбутній контент програвача, який буде відтворюватися в певному вигляді. Тут, як правило, містяться посилання на пряме завантаження всіх файлів.

Schedule – запит, який отримує розклад відтворення макетів. У попередніх розділах було сказано, що плеєр може відтворювати кілька макетів у різний час, таким чином створюючи чергу відтворення. Правила та порядок відтворення цієї черги плеєр отримує разом із цим запитом.

SubmitStats/SubmitLogs/SubmitScreenshot - допоміжні та необов'язкові запити, які потрібні, щоб повідомити сервер про свій стан. Усі попередні запити раніше отримували інформацію від сервера, тоді як ці запити навпаки відправляють інформацію на сервер. Статистика потрібна, щоб повідомити, скільки часу кожен макет відтворювався на екрані. Логи потрібні як допоміжна інформація для розробника та виявлення потенційних проблем.

Скріншот потрібен, щоб захопити поточний стан екрана і переконатися, що контент відповідає очікуванням.

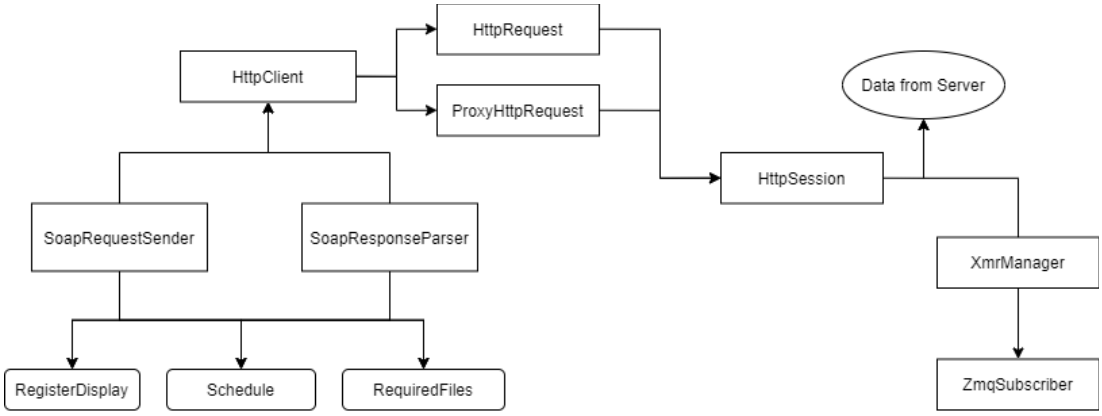

Рисунок 2.8 – Мережева архітектура

Як видно, всі ці запити є досить високорівневими, тому для реалізації нам достатньо взяти протокол прикладного рівня. HTTP є хорошим варіантом, але є ще більш підходящий протокол, який може працювати поверх HTTP – SOAP. Цей протокол визначає певний формат передачі повідомлення, тому ідеально підходить для опису та передачі всіх описаних вище запитів.

Крім описаного інтервалу та його запитів, нам необхідно ще й завантажувати файли, список яких ми отримали через запит RequiredFiles. Це досить легко зробити за допомогою звичайних HTTP запитів і не обмежувати себе більш високорівневими протоколами.

Як було видно, HTTP є ключовим протоколом у реалізації міжмережевої взаємодії, тому необхідно реалізувати HTTP клієнт, через який здійснюватиметься вся взаємодія із сервером.

Лістинг 2.12 – Обробка SOAP запиту та відповіді

```
template <typename R>
    HttpResult<R> soapReceived(const ResponseHttp& rsp)
    {
        auto [errorResponse, resultResponse] = rsp;
        if (errorRessponse) return HttpResult<R>{errorResponse, {}};

        HttpParser<R> p(httpResult);
        return p.parse();
```

```
    }

    template <typename R, typename Req>
    Future<HttpResult<R>> post(const Src& src, const Req& soap)
    {
        static_assert(std::is_assign<Req> && std::is_constr<Req>);

        VersaParser<Req> vp{soap};

                            return   ClientForHttp::getInst().post(src,
vp.toString()).then([](Future<ResponseHttp> futureRes) {
            return soapReceived<R>(futureRes.get());
        });
    }
```

Реалізація HTTP клієнта є досить тривіальним завданням з огляду на те, що нам не потрібно реалізовувати всі типи запитів, які передбачені в рамках протоколу HTTP. Для наших цілей підійдуть лише два типи: GET і POST. Більше того, іноді необхідно здійснювати запити до сервера не безпосередньо, а через якийсь проміжний проксі-сервер. Причин для цього може бути безліч, але важливо, щоб плеєр умів це підтримувати. Для цього ми залишаємо можливість настроювати проксі-сервер усередині HTTP клієнта.

Лістинг 2.13 – HTTP-клієнт

```
class ClientForHttp
{
public:
    ~ClientForHttp();

    static ClientForHttp& getInst();

    void closeClient();
    void setupProxy(const Opt<Src>& src);
    Future<ResponseHttp> get(const Src& src);
       Future<ResponseHttp>  post(const  Src&  src,  const  std::string&
body);
};
```

Надсилання запитів HTTP та отримання відповідей може здійснюватися в рамках однієї HTTP сесії. Ця сесія буде встановлювати з'єднання, проводити авторизацію, надсилати дані та отримувати відповідь від сервера. Одночасно таких сесій може бути відкрито відразу кілька, а

запити на сервер можуть надсилатися паралельно в декілька потоків. Такий підхід дозволить скоротити час очікування, якщо ми маємо запити, які не вимагають чіткої послідовності. Наприклад, це застосовується для завантаження списку файлів, де не важливо, який файл буде завантажено першим або останнім. Так само в рамках цієї сесії може здійснюватися обробка помилок, пов'язаних з відповіддю від сервера HTTP. Важливо це робити в рамках окремої сесії, адже якісь інші запити можуть бути успішними.

Лістинг 2.14 – Сесія кожного HTTP запита

```
class SessionForHttpConn : public std::shared_ptr_enabled<
SessionForHttpConn>
{
public:
    SessionForHttpConn(boost::asio::io_context& ioc);

          Future<HttpResponseResult> reqSend(const Src& src, const
boost::http_req<boost::http_body>& req);
    void shutdownSession();

private:
    void onFinished(const boost::err& code);
    void resultSetForHttp(const HttpResponseResult& result);

    template <typename ResultClbk>
       void hostResolve(const Src::Host& httpHost, const Src::Port&
httpPort, ResultClbk clbk);
                 void httpHostResolved(const boost::err& code,
resolver::result_types res);

    template <typename ResultClbk>
         void sessionConnect(resolver::result_types res, ResultClbk
callback);
    void sessionConnectionEstablished(const boost::err& code);

    template <typename ResultClbk>
    void sessionHandshake(ResultClbk callback);
    void sessionHandshakeFinished(const boost::err& code);

    template <typename ResultClbk>
    void sessionWrite(ResultClbk clbk);
      void sessionRequestWritten(const boost::err& code, std::size_t
amountOfBytes);

    template <typename ResultClbk>
    void sessionRead(ResultClbk clbk);
        void sessionRequestRead(const boost::err& code, std::size_t
amountOfBytes);
};
```

Ще одна сутність, яка потрібна для взаємодії з сервером – це сам HTTP запит. HTTP протокол чітко визначає структуру того, як має виглядати такий запит, тому є необхідність винести це в окремий клас. Тут встановлюється версія протоколу, з якою відбувається робота, заповнюються всі необхідні поля для авторизації та, нарешті, заповнюється саме тіло запиту.

Лістинг 2.15 – Об’єкт HTTP запита

```
class Req
{
public:
     Req(boost::request_type type, const Src& src, const std::string&
httpBody) :
        type_(type),
        src_(src),
        httpBody_(std::move(httpBody))
    {
    }

    boost::http_req<boost::http_body> get()
    {
        boost::http_req<boost::http_body> buildReq;

        buildReq.method(type_);
        buildReq.setPath(src_.get());
        buildReq.setVer(“1.1”);
                            buildReq.setField(boost::http_host_field,
getHost(src_.creds()));
        if (auto infoUsr = src_.creds().usrInfo())
        {
            buildReq.set(boost::http_authorization_field,
                                                        "Basic   "   +
crypto::convertToBase64(infoUsr.value()));
        }
        buildReq.setBody(std::move(httpBody_));
        buildReq.request_prep();

        return buildReq;
    }
private:
    std::string getHost(const Src::Creds& creds)
    {
        auto&& srcHost = creds.getHost();

        if (auto valPort = creds.getOptPort())
        {
            return srcHost + ":" + valPort->getStr();
        }

        return srcHost;
    }
```

```
private:
    boost::request_type type_;
    Src src_;
    std::string httpBody_;
};
```

2.9 Модуль Watchdog

Весь софт в рамках інфраструктури Digital Signage має мету працювати цілодобово і без перерви. Це особливо актуально для плеєрів, що постійно відтворюють контент. Проте будь-який софт завжди містить у собі потенційні баги, які можуть призвести до аварійного завершення роботи програми. Щоб боротися з цим, було придумано механізм моніторингу стану плеєра та його подальшого перезапуску у разі аварійного завершення роботи. Для цього необхідно реалізувати моніторинг процесу та його поточного стану. Коли програма завершується, вона завершується з якимось конкретним кодом чи сигналом. Цей код або сигнал може повідомити про причину завершення програми, що допоможе розрізняти нормальне завершення роботи від аварійного завершення роботи. Перехопивши та проаналізувавши цей сигнал, ми можемо перезапускати плеєр у потрібні моменти. Однак для того, щоб перехоплювати сигнали, необхідно встановити обробники сигналів для конкретного процесу і при цьому знати унікальний ідентифікатор запущеного процесу. Якщо реалізувати цей механізм у рамках двох незалежних процесів, то може виникнути безліч складнощів з коректною реалізацією, тому було прийнято рішення, що процес плеєра буде дочірнім для процесу, який здійснює моніторинг стану. Для того, щоб завершити роботи плеєра без подальшого перезапуска, у вікні плеєра існує спеціальна кнопка, яка посилає сигнал батьківському процесу, що виконує моніторинг, і плеєр завершується без перезапуску.

Лістинг 2.16 – Батьківський процес плєра

```
class PlayerMonitor
{
public:
    PlayerMonitor(const std::string& binPath);
    void startPlayer();

private:
    static void setHandlerFoprSignal(int signum);
    void catchSignals();

private:
    static inline int playerProcessId = -1;
    static inline bool isActive = false;

private:
    std::string playerBinaryPath_;
};
```

3 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

У ході дослідження цієї роботи були проаналізовані інші існуючі Digital Signage плеєри. Щоб перевірити нашу модель багатоплатформового плеєра, необхідно перевірити його роботу на різних платформах. Як платформи для тестування були обрані 3 платформи: Windows 10, Ubuntu Desktop і Ubuntu на Raspberry Pi 4. Перші дві платформи на архітектурі Intel x86, у той час як Raspberry Pi 4 на ARM.

Як приклади контенту, який відтворюватиметься на даних платформах, було обрано 4 різних макета.

Перший макет складається з 4 різних картинок на однотонному фоні або з картинкою на фоні, які знаходяться на різних шарах, тобто вони відрізняються координатою z. Цей приклад добре ілюструє, як плеєр рендерит зображення.

Другий макет складається із двох HTML файлів, які малюють текст на чорному тлі. Це добре ілюструє роботу вбудованого браузерного двигуна, який повинен вміти рендерувати HTML-код.

Третій та четвертий макети присвячені аудіо та відео програванню. Було вирішено створити 2 макети: окремо програється аудіодоріжка на чорному фоні та окремо відтворюється відео у Full HD якості. Це було зроблено тому, що хоча програмне аудіо та відео використовують один і той же плеєр для відтворення, відео ще додатково має візуальний ряд. Для того, щоб краще протестувати аудіо доріжку окремо, для неї було створено окремий макет.

Таблиця 3.1 – Результати виконання

Макет	Платформа	Використання пам’яті RAM у мегабайтах	Середній час запуску у секундах
Зображення	Windows 10 (x86)	152	0.33
Зображення	Ubuntu Desktop (x86)	127	0.39
Зображення	Ubuntu on Raspberry Pi (ARM)	98	0.29
HTML-контент	Windows 10 (x86)	335	0.73
HTML-контент	Ubuntu Desktop (x86)	298	0.78
HTML-контент	Ubuntu on Raspberry Pi (ARM)	301	0.75
Аудіо	Windows 10 (x86)	553	0.89
Аудіо	Ubuntu Desktop (x86)	563	0.87
Аудіо	Ubuntu on Raspberry Pi (ARM)	548	0.84
Full HD відео	Windows 10 (x86)	1001	1.04
Full HD відео	Ubuntu Desktop (x86)	987	1.15
Full HD відео	Ubuntu on Raspberry Pi (ARM)	901	1.1

Як видно з отриманих результатів, аудіо- та відео-контент вимагає найбільше ресурсів, а відео-контент найвибагливіший за ресурсами. Це пов'язано з тим, що у Full HD відео більша роздільна здатність і висока частота кадрів, що призводить до більшої кількості рендерингу та

навантаження на апаратне забезпечення. Цікаво, що HTML-контент не такий витратний за ресурсами, хоча фактично це означає внутрішній запуск веб-браузера. Це пов'язано з тим, що як приклад були використані прості HTML-сторінки без використання CSS та JavaScript. Це дозволило значно скоротити навантаження на вбудований веб-браузер, адже не довелося залучити інтепретатор для мови JavsScript. Більше того, бібліотека, яка використовується для рендеринга веб-контенту, сама по собі є однією з найлегших у порівнянні з такими аналогами як Chromium Embedded Framework та іншими.

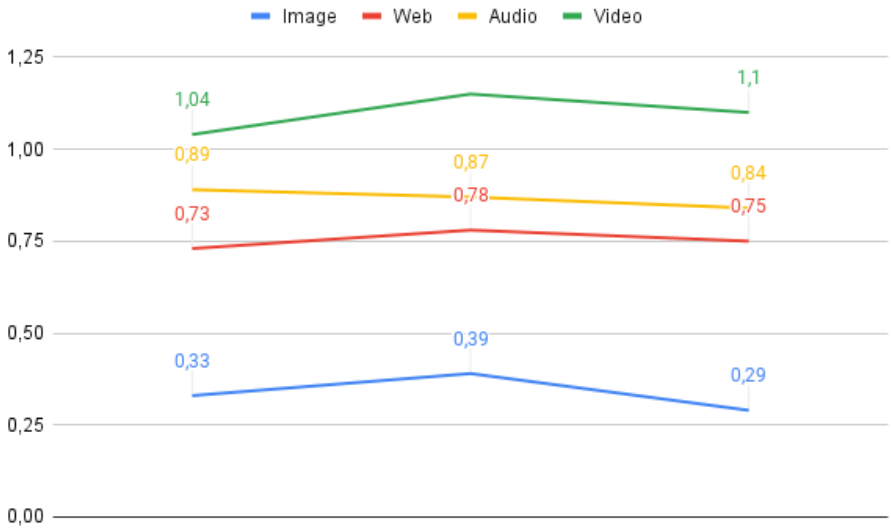

Рисунок 3.1 – Графік середнього часу

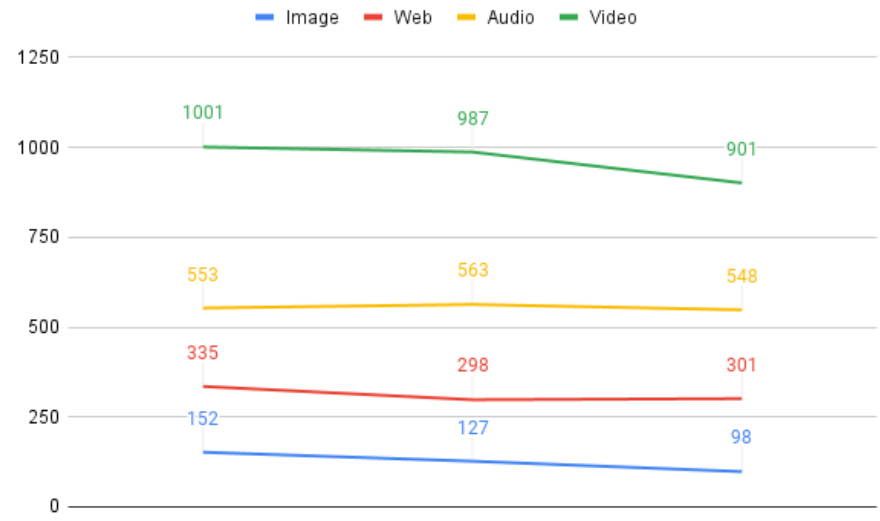

Рисунок 3.2 – Графік використання RAM

Якщо проаналізувати графіки, то буде видно, що споживання ресурсів залежить в основному від типу контенту, який відтворюється, а не від архітектури процесора чи операційної системи. Це не дивно, адже навіть якщо ОС якимось чином оптимізована під той чи інший вид контенту, це не вплине глобально на результати, тому що, наприклад, висока роздільна здатність та частота кадрів у відео від цього не зміниться. Незважаючи на те, що веб-контент вимагає відносно небагато ресурсів, це може змінитися, якщо веб-сторінка також почне містити відео- або аудіо-контент.

Крім дослідження споживання апаратних ресурсів, важливо перевірити, що плеєр працює коректно. Для цього було впроваджено модуль статистики, який відстежує, коли кожен тип медіа з'являється на екрані та зникає з нього. Це дозволяє переконатися, що макет відпрацював так, як нам і потрібно. Такий механізм називається Proof Of Play.

Статистика Proof of Play – це записи, зібрані плеєром і передані в CMS, яка фіксує, коли гравець відіграв елемент. Записи макета представляють час відтворення макета. Записи віджетів представляють час відтворення віджета, який може бути віджетом, специфічним для макета (наприклад, текстом) і може мати пов’язаний медіа-елемент бібліотеки (наприклад, відео). Збір статистики можна ввімкнути/вимкнути на рівні налаштувань плеєра. Коли статистика увімкнена, вона записується для всіх подій, які відтворює плеєр.

Type	Display ID	Display	Layout ID	Layout	Widget ID	Media	Tag	Number of Plays	Total Duration	Total Duration (s)	First Shown	Last Shown
layout	15	XfT - QM49H	340	PoP Layout	0	No media		3	0day 0hr 0min 46sec	46	2020-04-24 09:53:09	2020-04-24 09:53:55
media	15	XfT - QM49H	340	PoP Layout	1030	test screen.jpg		3	0day 0hr 0min 31sec	31	2020-04-24 09:53:14	2020-04-24 09:53:55
widget	15	XfT - QM49H	340	PoP Layout	1033	text		4	0day 0hr 0min 20sec	20	2020-04-24 09:53:09	2020-04-24 09:54:00

Рисунок 3.3 – Записи Proof of Play

Кожен запис містить корисну інформацію, скільки і яку тривалість мав кожен конкретний тип контенту. Більше того, запис містить інформацію про дату старту першого відтворення та дату закінчення останнього відтворення.

Все це дозволяє зрозуміти, чи працює плеєр відповідно до наших очікувань. Це може здатися зайвим, але при плануванні макета завжди можна або щось проґавити, або сам плеєр може накопичувати помилки під час відтворення, тому важливо переконатися, що вся інформація була доставлена та відтворена коректно.

Робота з мережею є важливою складовою у роботі плеєра, тому що всю потрібну мета-інформацію плеєр отримує через мережу. Більш того, всі файли, необхідні для коректної роботи, плеєр також отримує по мережі. Якщо відповідь від сервера в якомусь випадку виходить занадто довгою, то є необхідність оптимізувати цю частину коду, щоб прискорити взаємодію з сервером і підвищити якість роботи плеєра. Для початку розглянемо низку запитів, які вимагають всю необхідну мета-інформацію під час інтервалу збору даних. Список запитів, які вимірюються: RegisterDisplay, RequiredFiles, Schedule, SubmitLogs, SubmitStats, SubmitScrenshot.

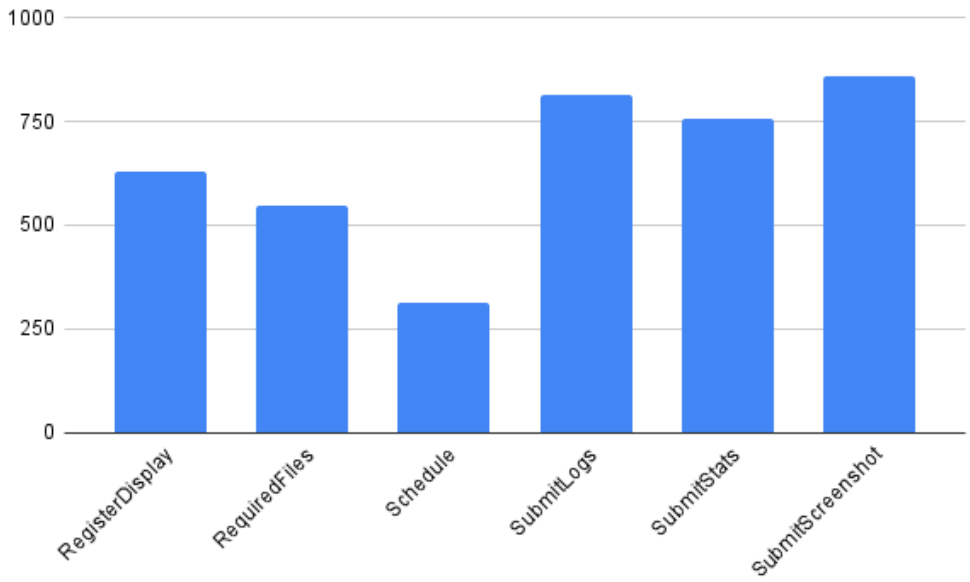

Рисунок 3.4 – Середній час відповіді на поширені запити

Проаналізуємо отриманий результат. Середній час відповіді знаходиться у прийнятному діапазоні, тому критичних покращень робити не доведеться. Можемо відзначити, що виділяється серед усіх запит Schedule,

час якого становить трохи більше 300 мілісекунд. Це нескладно пояснити тим, що запит на актуальний розклад макетів є досить простим з точки зору сервера, необхідно лише сформувати з готової БД результат у форматі XML та надіслати відповідь назад.

Розглянемо два інших запити (RegisterDisplay і RequiredFiles), результати яких знаходяться приблизно посередині. У випадку з RegisterDisplay – це ключовий запит, який виконує авторизацію або реєстрацію плеєра, і в цьому випадку серверу потрібно витратити якийсь час і звірити всі дані у себе в базі. RequiredFiles є витратним з точки зору часу збору всіх файлів, які необхідно завантажити з локальної бази сервера. Така операція може зайняти певну кількість часу, що збільшує час відповіді сервера.

Нарешті необхідно розглянути 3 запити, що зайняли найбільшу кількість часу. Вони відрізняються від перших трьох тим, що якщо перші три запитували інформацію, яку сервер діставав або зі своєї бази, або з локального кеша, то останні запити роблять рівно навпаки – вони надсилають велику кількість даних на сервер, які необхідно зберегти в певне сховище. Час відповіді збільшується з огляду на те, що збільшується час відправлення, тому що тепер тіло запиту не порожнє і містить певну кількість байт, яку потрібно передати через мережу. Більше того, операція запису даних майже завжди виконуватиметься довше, ніж операція читання, тому що тут не можна скористатися заготовленим кешем або швидким пошуком по базі даних. Різниця в часі між цими трьома запитами невелика, але вона все ж таки є. SubmitLogs відправляє набір логів у форматі XML, але при цьому максимальна кількість обмежена 500 записами, що дозволяє не навантажувати сервер занадто сильно і надсилати дані порційно. SubmitStats працює аналогічно і має обмеження на максимальну кількість записів, яке він може відправити за один запит. У випадку SubmitScreenshot все працює по-іншому. Даний запит дозволяє надіслати знімок вікна програвача, який робить сам програвач. Такий механізм є ще одним способом переконатися в

тому, що плеєр в даний момент показує саме той макет, який був запланований, і показує його коректно і без збоїв, тобто картинка з часом змінюється. З огляду на те, що тут як дані для відправки використовується бінарне уявлення самого знімка, швидкість відправки може варіюватися в залежності від того, який розмір був у вікна, в якому був зроблений даний знімок. Як приклад було зроблено повноекранний знімок 1920x1080, тому цей запит вийшов найдовшим. Якщо картинка буде меншою, то час відповіді може скоротитися, але не істотно.

Крім запитів про мета-інформацію, є ще один важливий тип запитів, який вимагає окремого та детального аналізу – запит на завантаження файлу. Дані запити немає сенсу вимірювати окремо, тому що час відправлення та отримання сильно залежить від розміру файлу, який завантажується. Якщо файл дуже маленький і є простим HTML-документом, то завантаження може відбутися менше, ніж за секунду. У випадку великих файлів, таких як Full HD відео, це може зайняти навіть кілька хвилин, в залежності від швидкості інтернету. Проте такі файли можна завантажувати різними методами, тому буде проведено аналіз порівняння цих двох методів.

Перед самим аналізом необхідно описати ці методи. Перший метод полягає у прямому HTTP GET запиті, який запитує весь контент файлу повністю. Якщо файл занадто великий, то HTTP-клієнт може використовувати алгоритм зчитування даних по блоках. Другий метод – використовувати SOAP API та вказувати розміри та відступи для кожного блоку окремо. Більше того, кожне завантаження кожного блоку також є окремим запитом. Таким чином, якщо файл дуже великий і розбивається на велику кількість блоків, такий спосіб може виявитися невигідним у використанні. Тим не менш, сервер, на який надсилаються дані запити, може використовувати спеціальний внутрішній кеш, який дозволяє швидше формувати та надсилати блоки даних для SOAP API. У разі невеликих файлів це може значно прискорити час відповіді.

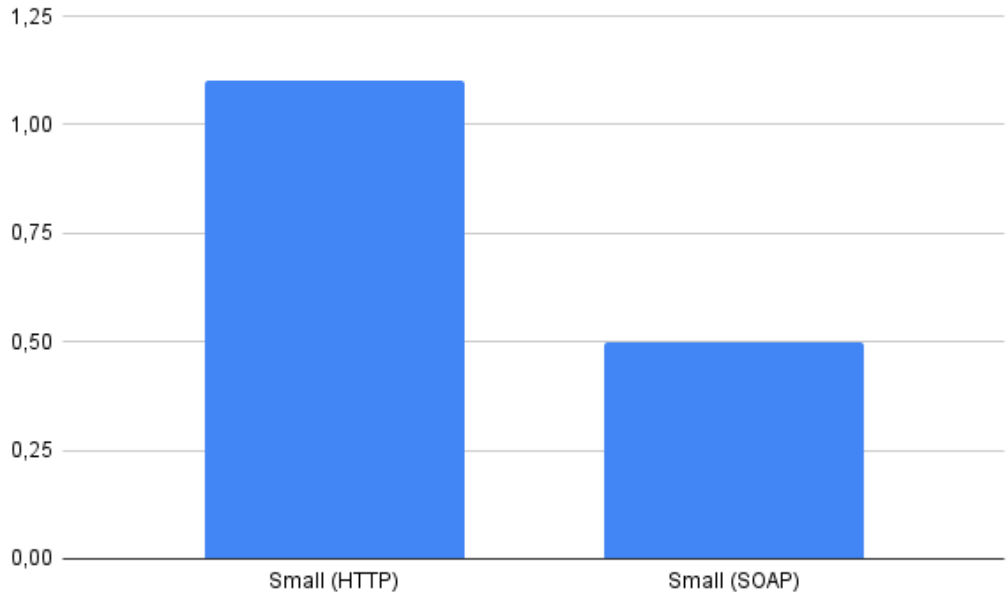

Рисунок 3.5 – Середній час завантаження маленького файла

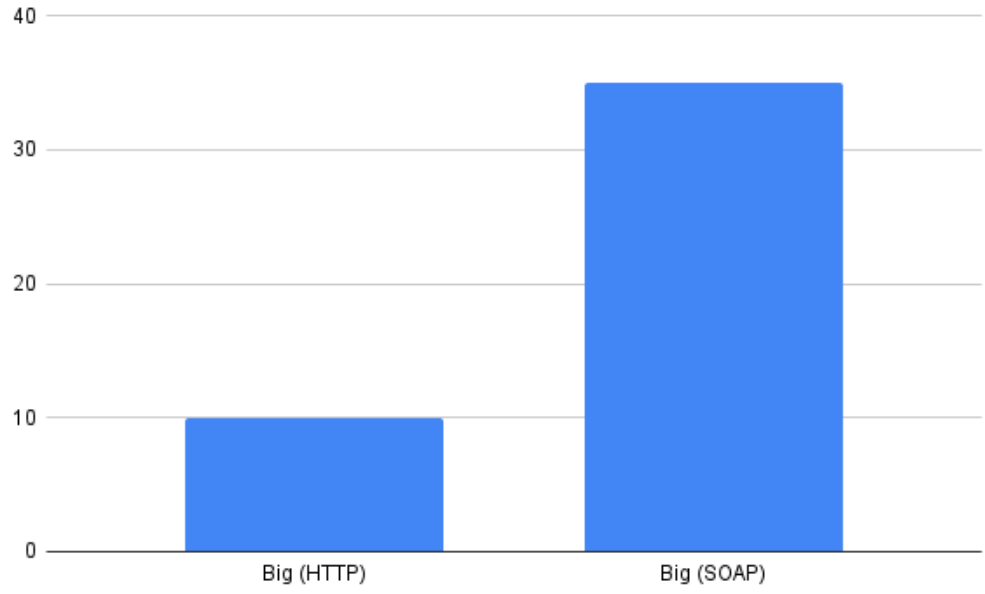

Рисунок 3.6 – Середній час завантажання великого файла

Після аналізів результату стало зрозуміло, що наші початкові припущення виявились вірними. Тест був проведений на 2 файли різного розміру, які на графіку для простоти названі "Small" і "Big". Маленький XML файл був розміром менше 1 кілобайта, тоді як великим файлом було Full HD відео розміром близько 200 мегабайт. Різниця в розмірах була істотною, тому

результати виявилися більш ніж наочними. Для маленького файлу метод завантаження безпосередньо через HTTP GET запит виявився менш ефективним, ніж його аналог через SOAP API. Як було зазначено, файли такого маленького розміру не б'ються на блоки, тому дуже швидко надаються сервером. Різниця трохи більша, ніж у 2 рази. Щодо великих файлів, то тут ситуація зворотна: SOAP-завантаження програє майже в 4 рази. Великі файли розбиваються на безліч маленьких блоків фіксованого розміру, а в SOAP кожен такий блок завантажується окремим запитом.

В результаті можна зробити висновок, що SOAP API підходить для завантаження маленьких файлів, а прямий HTTP GET запит підходить для великих файлів. У зв'язку з тим, що для відтворення контенту потрібна безліч маленьких допоміжних файлів, без яких відтворення контенту неможливе або некоректне, SOAP API залишається дуже важливим і актуальним методом для завантаження.

Залишився останній вид взаємодії з мережею, який необхідно розглянути – push-messaging. Відправка звичайних HTTP-запитів відмінно підходить для більшості рутинних дій, які необхідно здійснити плеєру: запит мета-даних, відправлення даних та завантаження файлів. Всі ці операції поєднує те, що вони заплановані та ініціюються насамперед із боку плеєра. Таким чином, плеєр виступає в ролі HTTP-клієнта та посилає всі ці запити на сервер. Це працює відмінно, але є ситуації, коли серверу потрібно повідомити плеєру якусь інформацію. Прикладом такої інформації може бути зміна запланованого розкладу або запит на позачергове надсилання знимку поточного стану екрана. Щоб це реалізувати, плеєру необхідно бути напоготові і чекати повідомлень від сервера. Реалізований HTTP-клієнт не підходить для реалізації цього завдання, адже він займається виключно надсиланням даних та отримання відповідей на запити. Нам необхідно реалізувати HTTP-сервер на стороні плеєра, тоді і плеєр, і віддалений сервер одночасно буде виступати і клієнтом, і сервером. Важливо внести ясність і розмежувати термінологію. Коли мова йде про віддалений сервер, то ми

говоримо про якесь фізичне апаратне забезпечення, що знаходиться досить далеко і виступає в ролі постачальника даних для плеєра. Для зв'язку з сервером необхідно мати доступ до глобальної мережі Інтернет. Якщо ж ми говоримо про клієнт-серверну архітектуру, згадуючи про те, що плеєр може і клієнтом, і сервером, то йдеться про відкриття сокету в рамках однієї програми – нашого плеєра.

Після того, як стало зрозуміло, що необхідно мати відкритий серверний сокет з боку програвача, необхідно вибрати правильний протокол для реалізації. Спочатку йшлося про вибір HTTP як цільового протоколу для реалізації даної моделі, однак до цього протоколу можуть бути питання в рамках нашого завдання. Сервер відправлятиме дуже прості команди, що складаються з 2 або 3 слів, тому немає необхідності реалізувати досить затратний HTTP-сервер. Альтернативою може бути реалізація черги повідомлень між одержувачем та відправником. Така модель ідеальна підходить для нашого завдання та задовольняє простоті реалізації. Більш того, таку чергу легко розпаралелити на кілька брокерів надсилання повідомлень, тому це ще й прискорює процес доставки та обробки повідомлень. Тим не менш, важливо провести порівняльний аналіз і переконатися, що це припущення дійсно було вірним.

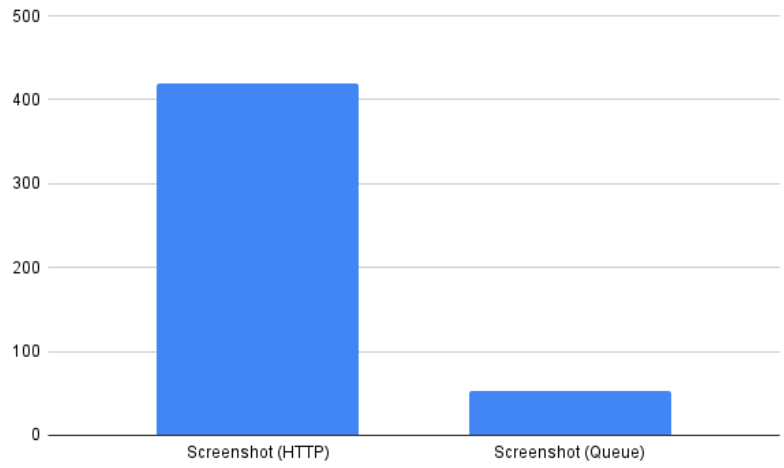

Рисунок 3.7 – Середній час відповіді на команду знимка екрану

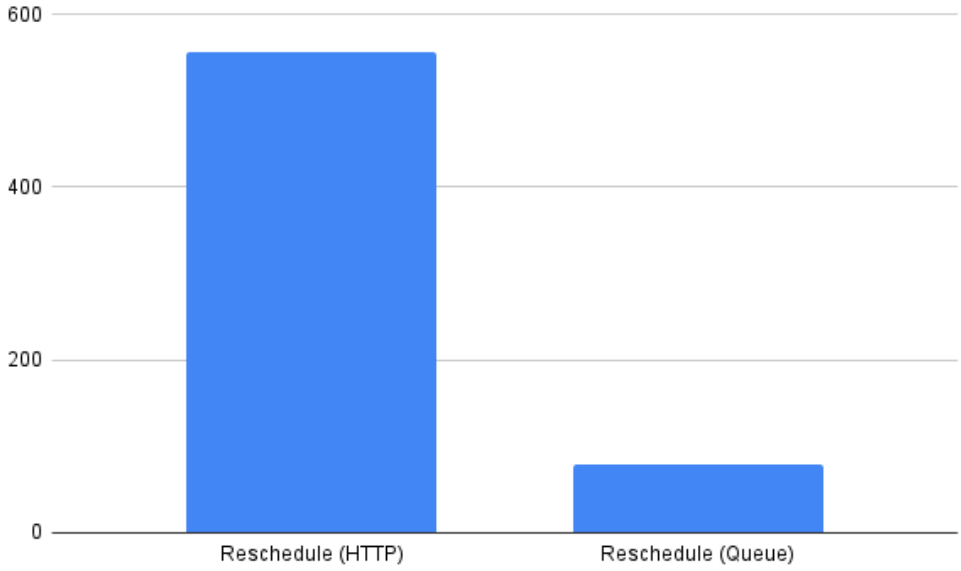

Рисунок 3.8 – Середній час відповіді на команду перепланування

Ми досліджували дві команди, які були згадані вище: команда перепланування розкладу і команда на запит знімка екрана. Обидві ці команди відносяться до класу термінових команд, тому важливо, щоб вони були швидко доставлені сервером та виконані плеєром. Порівнюючи отримані результати, можна зробити однозначний висновок, що варіант з сервером HTTP на кілька порядків повільніше, ніж варіант з чергою повідомлень. Це пояснити нескладно, адже для обробки HTTP-запиту його необхідно прийняти та встановити з'єднання, правильно проаналізувати заголовок та тіло запиту. Всі ці дії суттєво уповільнюють процес. У разі черги повідомлення канал зв'язку відкритий і доступний постійно, необхідно лише відправити в цей канал потрібне повідомлення, як одержувач на іншому кінці його відразу ж отримає, тому що з'єднання вже вставлено і жодних заголовків немає.

У ході досліджень, пов'язаних з роботою мережі, було проведено безліч тестів, у ході яких було встановлено, що хоча HTTP є досить популярним протоколом передачі даних, він далеко не завжди підходить для реалізації завдань певного типу.

Досі всі дослідження були націлені на безпосередню роботу плеєра, проте є ще один важливий аспект реалізації моделі плеєра – компіляція проекту.

Компіляція проекту – це дуже важлива та невід'ємна частина розробки. Якщо ігнорувати проблеми, пов'язані з компіляцією, можна істотно уповільнити процес розробки. У таких мовах як C++ це особливо важливо і актуально, адже тут компіляція зазвичай відбувається повільніше, порівняно з іншими більш високорівневими мовами.

Справа в тому, як у C++ підключаються залежності. Коли розробляється якийсь компонент, він рідко є самостійним і найчастіше залежить від існуючих і готових компонентів. Щоб використовувати готові компоненти, їх необхідно підключити до проекта. У таких мовах, як C#, така дія відбувається через ключове слово import. Це дозволяє імпортувати символи з інших компонентів, які називаються модулями. Сам модуль визначає, які символи потрібно експортувати назовні для включення в інші проекти. Це дозволяє реалізувати досить високорівневі абстракції та приховувати деталі реалізації.

У C++ історично склалося все зовсім інакше. При реалізації компонентів зазвичай використовується набір класів. Кожен клас має інтерфейс, який описується у заголовному файлі, та реалізацію, яка описується у файлі реалізації. Такий поділ допомагає розділяти деталі та реалізації та інтерфейс класа. Тим не менш, навіть при такому поділі приватна секція класу все ще доступна в заголовному файлі, що істотно ускладнює повну інкапсуляцію об'єкта. У C++ немає поняття символів, що імпортуються і експортуються, включається все, що описано в заголовному файлі. Саме це створює проблему тривалої компіляції проекту. Якщо приватна секція буде змінена, що може відбуватися досить часто, адже це частина реалізації, а не зовнішнього інтерфейсу, всі класи і компоненти, які від неї залежать, потрібно буде компілірувати заново. Це відбувається тому,

що замість символів, що експортуються та імпортуються, використовується проста підстановка тексту.

Працює це таким чином: вміст файлу підключається через директиву #include, далі під час компіляції проекту відбувається попередня обробка всіх директив, що призводить до підстановки всього вмісту файлу. Цей процес відбувається рекурсивно, тому за великого графа залежностей компіляція може займати колосальну кількість часу. Один із способів вирішення проблеми – винести приватну секцію в окремий вкладений клас та визначити цей клас вже у файлі реалізації. Це дозволяє суттєво зменшити час компіляції, проведемо відповідні виміри.

Таблиця 3.2 – Результати сборки

Назва сборки	Час
No Pimpl (Debug)	10:21
No Pimpl (Release)	5:17
Yest Pimpl (Debug)	5:47
Yest Pimpl (Release)	2:32

Розглянемо результати. Метод, про який йшлося вище, називається Yest Pimpl. Компіляція була запущена як у налагоджувальному, так і релізному режимах. Це зроблено тому, що час і процес збирання в них може дуже відрізнятися. З результатів видно, що без запропонованого способу компіляція в релізному режимі займається майже стільки ж, скільки з цим способів у режимі налагодження. Можемо зробити висновок, що ця ідіома допомагає збільшити швидкість компіляції проекту, що позитивно впливає на процес розробки загалом.

4 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

В результаті роботи була створена модель кроссплатформенного Digital Signage плеєра. Однак, щоб зрозуміти, як він працює, необхідно простежити цикл від початку запуску сервера до відтворення контенту на екрані.

Для початку нам необхідно вибрати сервер, встановити та запустити його. Як сервер був вибраний Xibo CMS, який знаходиться у вільному доступі і поширюється під ліцензією AGPL 3.0. Цей сервер використовує контейнери Docker для простоти розгортання. Використовуємо команду docker-compose, щоб загорнути та запустити вказаний сервер. Перед розгорткою необхідно відредагувати файл конфігурації, в якому потрібно встановити пароль та вказати порти, за яким буде доступна CMS. Після того, як сервер запущений, він доступний на локальному хості порту, вказаному у файлі конфігурації.

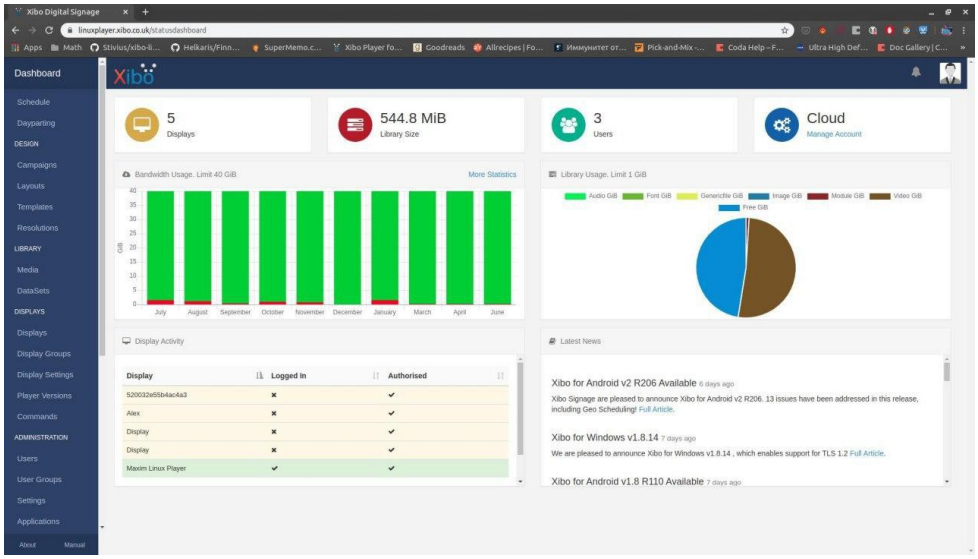

Рисунок 4.1 – Головне меню Xibo CMS

Ця CMS має безліч налаштувань, які на даний момент нас не цікавлять. Зараз нам потрібно підготувати все для першого запуску плеєра. При першому встановленні сервера в комплекті вже йде стандартний макет, який буде використовуватися за замовчуванням. Це зручно, тому що немає необхідності вникати в процес створення власного макета з нуля, досить просто підключити свій плеєр і перевірити працездатність циклу запуску та

роботи. Тим не менш, враховуючи, що наше завдання протестувати плеєр у різних умовах, найкраще створити кілька макетів із різним типом контенту.

Після створення всіх потрібних макетів можна перейти до зв'язування плеєра і сервера. Досі ми нічого не сказали серверу про плеєр, що підключається, тому що перше з'єднання ініціюватиме саме плеєр як клієнт. Для цього необхідно зібрати весь список необхідних параметрів, які використовуватимуться для підключення:

– ім'я хоста;
– порт;
– секретний ключ CMS;
– опціонально проксі-сервер.

Проксі-сервер зараз використовуватись не буде, а хост та порт ми отримали, коли проводили початковий запуск сервера. Залишилося дізнатися останню частину – секретний ключ. Його можна отримати в налаштуваннях самої CMS у розділі Settings. Тепер необхідно ввести всі ці параметри у програвач.

Для конфігурації плеєра було створено окрему програму, яка поставляється в комплекті з самим плеєром. Це було зроблено для того, щоб розділити ту частину програвача, яке відповідає за планування та відтворення контенту з частиною, яка займається виключно конфігурацією. Конфігурація потрібна лише один раз при старті або при подальшому підключенні до іншого сервера.

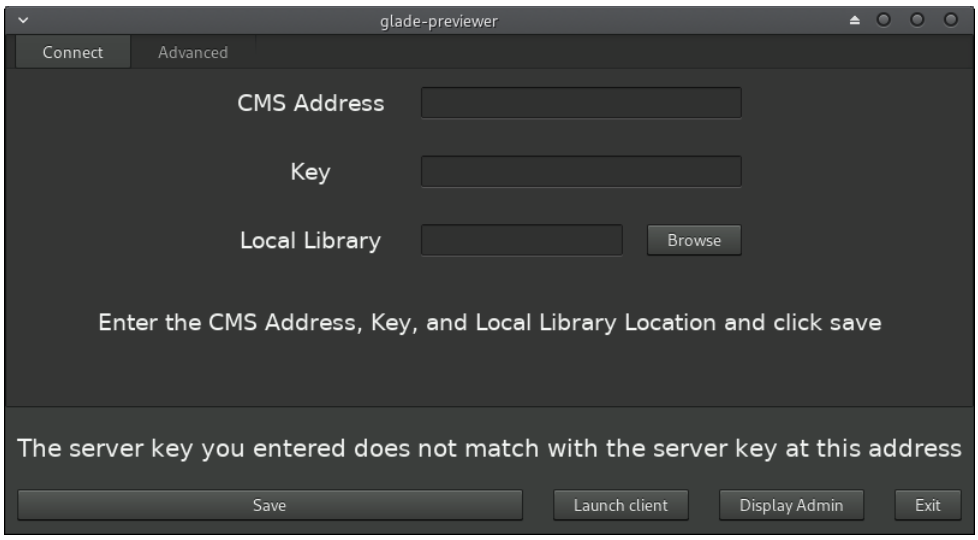

Рисунок 4.2 – Окно конфігурації плеєра

У полі адреси ми вводимо хост та порт, також потрібно вказати схему HTTP або HTTPS. На нашому сервері HTTPS використовується за замовчуванням. Наступним полем вводимо ключ, який є умовним паролем до сервера, щоб до нього не підключалися випадкові плеєри. Крім цього, можна вибрати папку, в якій зберігаються всі завантажені матеріали з сервера. Вибирати папку необов'язково, і у разі порожнього поля плеєр сам вибере папку для зберігання, яка найкраще підходить для конкретної операційної системи. Крім стандартних налаштувань, є також і просунуті установки, в яких можна задати параметри проксі-сервера і змінити апаратний ключ. Апаратний ключ – це унікальний ідентифікатор програвача, який генерується з декількох платформо-незалежних компонентів, якщо не вводити його вручну.

Після збереження всіх параметрів, плеєр спробує підключитися до сервера і видасть повідомлення про те, що чекає на схвалення з боку адміністратора сервера. Справа в тому, що навіть за допомогою всіх ключів і паролів, необхідно вручну авторизувати плеєр в панелі CMS. Після цього знову зберегти параметри і, у разі успіху, плеєр видасть повідомлення про те, що він готовий почати роботу.

```
<?xml version="1.0" encoding="utf-8"?>
<cmsAddress>https://linuxplayer.xibo.co.uk/</cmsAddress>
<key>egDeed06</key>
<localLibrary>&amp;quot;/home/stivius/Projects/Upwork/xibo-linux/build-debug/bin/resources&amp;quot;</localLibrary>
<username/>
<password/>
<domain/>
<displayId>3cf6b5aa5dce8b60e7bde7296d44df49</displayId>
```

Рисунок 4.3 – Формат зберегання конфігурації

Після збереження налаштувань вони зберігаються у форматі XML в окремому файлі і за бажання можуть бути відредаговані вручну. Тим не менш, формат збереження може бути змінено в майбутньому, тому краще їх змінювати через програму конфігурації.

Тепер, коли плеєр повністю налаштований, його можна запускати. Під час запуску плеєра відкривається 2 вікна: основне вікно, на якому малюється та відтворюється контент, та допоміжне вікно, на яке виводяться логи та помилки плеєра. Такі вікна найкраще розділяти, щоб логи не заважали основній роботі програвача і використовувалися тільки в крайньому випадку. Під час розробки це вікно дуже корисно та інформативно, адже містить усю послідовність дій, яка виконується під час запуску плеєра.

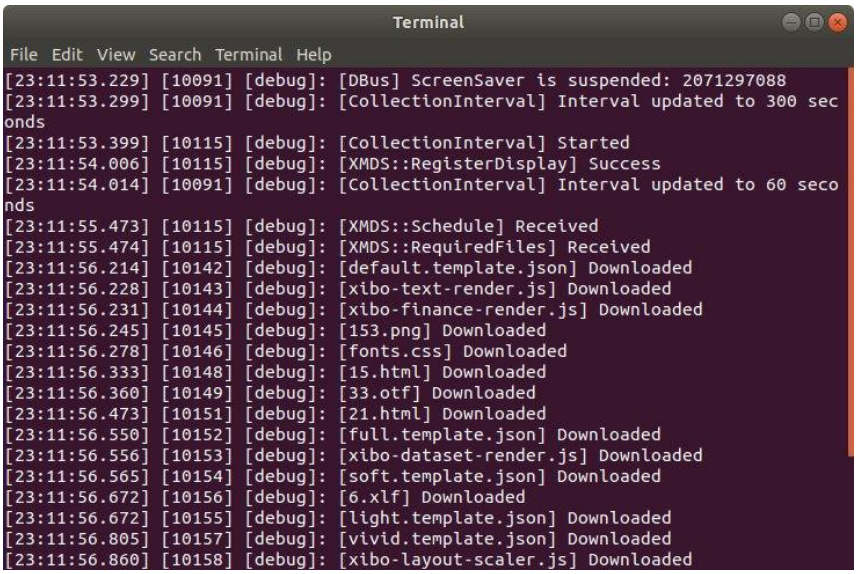

Рисунок 4.4 – Вікно додаткової інформації

У вікні видно, що відбувся запуск інтервалу збору даних, почали виконуватись запити, які ми докладно описували у попередніх розділах. Крім

цього, видно, як почалося масове завантаження файлів, які будуть використовуватися для відтворення контенту.

Тепер можна подивитися на результат, що вийшов.

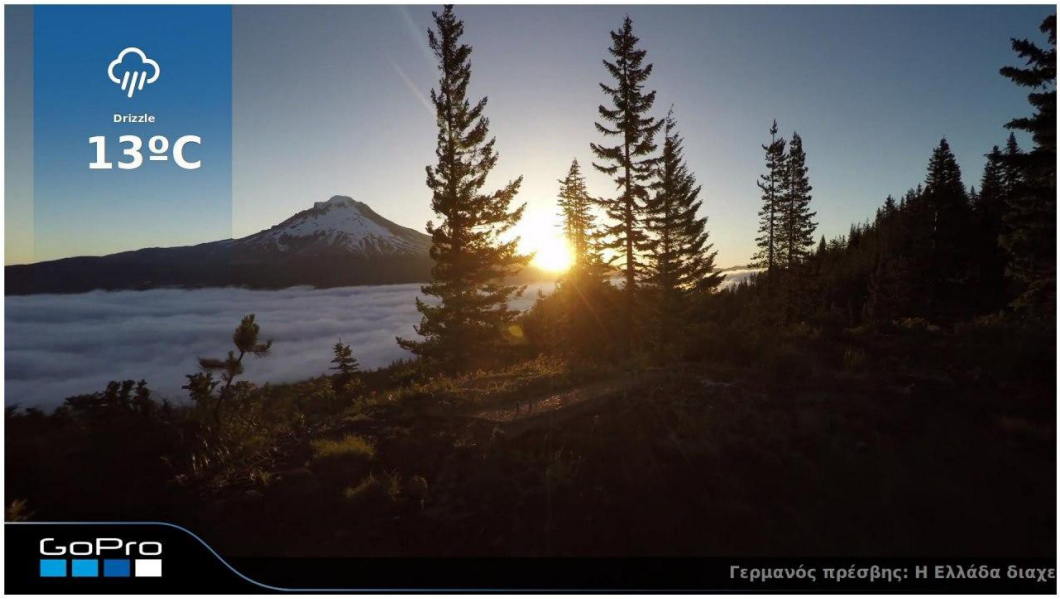

Рисунок 4.5 – Результат роботи плеєра

Даний макет один із найскладніших для відтворень і включає відразу кілька типів медіа-контенту. Також він чудово ілюструє роботу відразу кількох компонентів, тому розглянемо його докладніше. На задньому фоні працює відео у Full HD якості, тобто вікно розміром 1920×1080. Поверх цього відео внизу працює стрічка новин, яка виконується веб-модулем. Стрічка новин містить текст різними мовами, щоб продемонструвати підтримку юнікоду. Ще на шар вище розташована напівпрозора картинка з написом GoPro і градієнтною синьою смужкою зліва. І на останньому шарі розташований прогноз погоди, який також реалізується через веб-модуль. Таким чином, ми бачимо взаємодію всіх основних типів контенту, за винятком аудіо, хоча і воно представлене у вигляді звукового супроводу в самому відео. Такий макет є один із найвимогливіших за ресурсами.

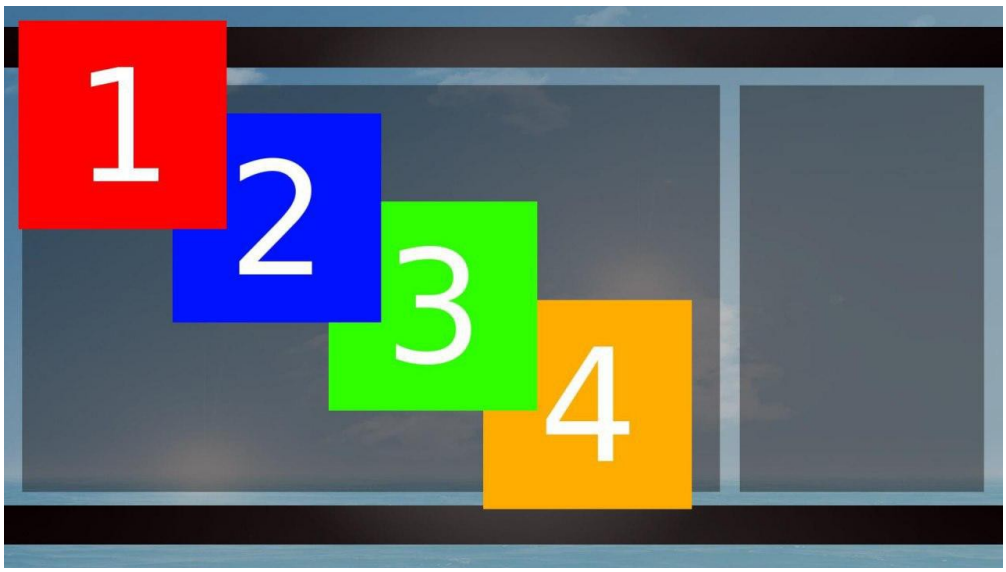

Рисунок 4.6 – Результат роботи плеєра

Рисунок 4.7 – Результат роботи плеєра

На цих двох макетів використовуються ті модулі, які ми вже бачили у дії, але кожен робить акцент на своєму. Перший макет компонує зображення в певному порядку та змінює їх місцями після закінчення заданого інтервалу. Другий макет акцентує увагу на короткому відео, яке працює на фоні картинки. Як видно, конструювати макет можна абсолютно з різних типів медіа, що дозволяє створювати контент будь-якої складності та практично будь-якого масштабу.

Остання частина плеєра, що залишилася не розглянутою – це екран статусу плеєра. Відтворення є дуже динамічним та мінливим процесом, може заплановано багато макетів, які змінюватимуться з часом. Не завжди очевидно, який поточний стан, хоч і ми вже описували безліч способів перевірити його. Всі описані способи робили це віддалено, а екран стану запускається прямо під час роботи плеєра окремим вікном і виводить всю інформацію по пунктах.

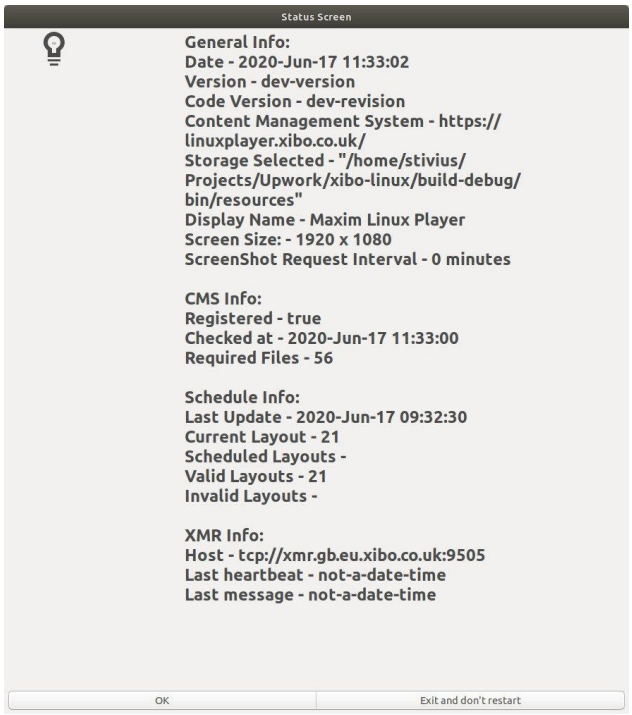

Рисунок 4.8 – Статус екран плеєра

Ми проаналізували отримані результати та запустили плеєр, щоб переконатися, що результати відповідають споконвічним очікуванням.

ВИСНОВКИ

В результаті роботи був розроблений багатоплатфомовий плеєр, який можна використовувати в рамках інфраструктури Digital Signage. Під час розробки було вирішено ряд завдань, пов'язаних з обробкою медіаконтенту, взаємодії з мережею і запуском плеєра з урахуванням того, що він повинен працювати декількох платформах. Плеєр підтримує запуск на Windows 10, Ubuntu Desktop (обидва на Intel x86) та на Ubuntu під Raspberry Pi 4 (ARM).

У процесі реалізації відбулося знайомство з графічними стеками Windows і Linux, а також із графічною бібліотекою GTK+, яка допомагає у створенні графічного інтерфейсу. З огляду на те, що сама бібліотека написана на мові C, ми використовували gtkmm – інтерфейс на мові C++. Крім цього, ми використовували бібліотеку GStreamer для обробки аудіо- та відеоконтенту. Бібліотека має багато можливостей, тому що інтегрована з системою плагінів. Завдяки цьому відео працює на прийнятній швидкості навіть на такому пристрої, а велика кількість плагінів дозволяє підтримувати багато різних кодеків і форматів.

Для того, щоб вищезгаданий медіаконтент постійно оновлювався і був в актуальному стані, була розроблена мережева архітектура з використання стека TCP/IP. Для отримання миттєвих повідомлень ми використовували ZeroMQ бібліотеку. Все це було зроблено так само з використанням багатопоточності, що дозволяє плеєру збільшити швидкість роботи.

В роботі є можливість поліпшити деякі модулі:

– додати можливість підвантаження медіа-модулів під час виконання роботи програми;

– розширити кількість доступного контенту;

– продовжувати роботу над поліпшенням продуктивності відео;

– оновити движок для веб-контенту, щоб підтримувати найновіші стандарти.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Digital Signage – что это такое? [Електронний ресурс] / Альянс Про – Режим доступу: www/ URL: https://tran.su/2018/10/10/digital-signage-dlja-teh-perevodchikov – 2018 р.

2. SOAP [Електронний ресурс] / W3C – Режим доступу: www/ URL: https://www.w3.org/TR/soap – 2007 р.

3. Docker. Зачем и как [Електронний ресурс] / softaria – Habr – Режим доступу: www/ URL: https://habr.com/ru/post/309556 – 2016 р.

4. Хмарні обчислення [Електронний ресурс] / Wikipedia – Режим доступу: www/ URL: https://en.wikipedia.org/wiki/Cloud_computing – 2020 р.

5. Raspberry Pi 4 Reduced Schematics [Електронний ресурс] / Raspberry Pi Foundation – Режим доступу: www/ URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/schematics/rpi_SCH_4b_4p0_reduced.pdf – 2019 р.

6. Player API Overview [Електронний ресурс] / Xibo Signage LTD – Режим доступу: www/ URL: https://xibo.org.uk/docs/developer/player-api-overview – 2019 р.

7. XMR – Push Messaging [Електронний ресурс] / Xibo Signage LTD – Режим доступу: www/ URL: https://xibo.org.uk/docs/setup/xmr-push-messaging – 2019 р.

8. Adam B Singer. Practical C++ Design: From Programming to Architecture [Текст] / Adam B Singer – Apress, 2017 – 266 с.

9. Make a GTK Web Browser with Glade [Електронний ресурс] / CodeNerd – Режим доступу: www/ URL: https://prognotes.net/2019/12/make-a-gtk-web-browser-with-glade – 2019 р.

10. Codec [Електронний ресурс] / Wikipedia – Режим доступу: www/ URL: https://en.wikipedia.org/wiki/Codec – 2020 р.

11. Hardware-accelerated video decoding [Електронний ресурс] / GStreamer Freedesktop – Режим доступу: www/ URL: https://gstreamer.freedesktop.org/documentation/tutorials/playback/hardware-accelerated-video-decoding.html?gi-language=c – 2018 р.

12. Bjarne Stroustrup. The C++ Programming Language [Текст] / Bjarne Stroustrup – Addison-Wesley Professional, 2013 – 1366 с.