Industrial Internship Program

Final Report

Front End Development for OpenVideoWalls

Student Name

Sira Kongsiri

AI and Computer Engineering

CMKL University

***Fall/Spring/Summer 20xx* Semester**

Project Title: *Open Video Walls*
Student Intern: *Mr.Sira Kongsiri*
Supervisor Name: *Dr. Jinjun Xiong*
Company: *University at Buffalo, Department of Computer Science and Engineering*

Abstract

This report documents my internship experience developing the frontend interface for OpenVideoWalls, an innovative open-source system for building video walls using recycled heterogeneous displays. The project addresses the growing environmental concern of electronic waste by repurposing discarded yet functional display screens into synchronized video wall systems. My contribution focused on creating a responsive and intuitive React-based frontend that communicates effectively with the system's backend and the client side raspberry PI, enabling users to control and configure the video wall easily. The interface facilitates stream management, display synchronization settings, layout configuration, and monitoring system performance. Through this internship, I gained valuable experience in frontend development, real-time data visualization, WebSocket communication, endpoint development, and collaborative software development within an environmentally conscious project. The developed frontend successfully complements the OpenVideoWalls system's capabilities, making it more accessible to users without technical expertise and contributing to the overall goal of promoting sustainable electronic waste management through practical reuse solutions.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Profesor Jinjun Xiong and my supervisor Amir Nassereldine, for their guidance, support, and mentorship throughout this internship. Their expertise and feedback were invaluable in shaping my approach to frontend development for the OpenVideoWalls project.

I extend my appreciation to the entire OpenVideoWalls team for their collaboration and willingness to share knowledge. Special thanks to the backend developers who patiently worked with me to establish effective communication interfaces between the frontend and backend components.

I am also grateful to CMKL University and the University at Buffalo for providing this opportunity and creating an environment conducive to learning and professional growth.

Finally, I thank my academic advisors and peers who offered encouragement and constructive criticism that helped refine my work on this project.

Table of Contents

Chapter 1

Introduction

1.1 Problem Statement

Electronic waste, particularly discarded display screens, represents a significant environmental challenge. According to the Global E-waste Monitor report[5] referenced in the OpenVideoWalls paper[1], approximately 5.9 billion kilograms of discarded screens and monitors were produced globally in 2022, with only 22.3% being properly recycled. Traditional recycling methods for these devices are resource-intensive, requiring substantial labor and energy, and often involve hazardous chemicals like arsenic, mercury, and cadmium that pose environmental and health risks.

Existing video wall systems, which could potentially repurpose these screens, are typically built with homogeneous displays and proprietary software, making them expensive and inaccessible for general use. These systems lack open technical documentation, clear universal metrics for evaluation, and user-friendly interfaces that would enable widespread adoption of display recycling through video wall implementation.

A significant challenge in this context is the absence of an intuitive, accessible frontend interface that would allow users without technical expertise to configure, control, and monitor video wall systems built from recycled heterogeneous displays. Without such an interface, the broader adoption of the OpenVideoWalls system would be limited, hindering its potential environmental impact.

1.2 Project Solution Approach

My approach to addressing the frontend needs of the OpenVideoWalls system involved developing a React-based web application that serves as the control center for the video wall system. This solution provides an intuitive interface for configuring, controlling, and monitoring the video wall, making the system accessible to users with varying levels of technical expertise.

The frontend solution integrates with the existing OpenVideoWalls backend components, including the Server, Relay, and Client modules described in the research paper. It enables users to:

1. Configure the layout and arrangement of heterogeneous displays in the video wall
2. Upload, manage, and control video content across displays
3. Monitor the synchronization status and performance metrics in real-time
4. Adjust synchronization parameters to optimize the video wall experience
5. Visualize system diagnostics and identify potential issues

The frontend communicates with the backend through RESTful APIs for configuration management and WebSocket connections for real-time monitoring and control. This approach ensures responsive interaction while maintaining system performance.

By creating an accessible frontend interface, this project contributes to the overall goal of the OpenVideoWalls system: to provide an open-source, low-cost solution for recycling display screens into functional video walls, thus reducing electronic waste and promoting sustainable reuse practices.

1.3 Project Objectives

The specific objectives of my frontend development work for the OpenVideoWalls project were:

1. **Develop an intuitive user interface**: Create a user-friendly frontend that enables users with varying technical expertise to configure and control video walls built with recycled heterogeneous displays, thereby increasing the accessibility and adoption potential of the OpenVideoWalls system.
2. **Implement real-time monitoring capabilities**: Design and develop visualization components that display synchronization metrics, system performance, and potential issues in real-time, allowing users to ensure optimal video wall operation.
3. **Establish effective backend communication**: Create robust communication channels between the frontend and backend components using RESTful APIs for configuration management and WebSocket connections for real-time data exchange and control.
4. **Enable comprehensive video content management**: Implement features for uploading, organizing, and scheduling video content across multiple displays while maintaining synchronization and quality.
5. **Support flexible display configuration**: Build interface components that allow users to define and adjust the physical layout of heterogeneous displays in the video wall, ensuring correct content distribution and appearance.

Chapter 2

Background

2.1 Fundamental Theory and Concepts

2.1.1 Video Wall Systems[1]

Video walls are large display systems consisting of multiple screens arranged in grid or custom layouts to show synchronized content across all displays. As outlined in the OpenVideoWalls paper, these systems have applications in retail, museums, entertainment venues, and public spaces. The core challenge in video wall systems is maintaining precise synchronization across all displays to ensure a coherent visual experience[1].

Traditional video wall systems rely on proprietary hardware and software solutions, specialized controllers, and typically require homogeneous displays with identical specifications. In contrast, the OpenVideoWalls approach aims to support heterogeneous displays with varying specifications, sizes, and capabilities, making it suitable for recycled screens.

The synchronization challenge is particularly complex[2] with heterogeneous displays, as each screen may have different processing capabilities, refresh rates, and response times. The OpenVideoWalls system addresses this through a combination of server-side timestamp embedding and client-side frame management, as detailed in the research paper.

2.1.2 Frontend Development for Control Systems

Control systems for multimedia equipment like video walls require specialized frontend development approaches that differ from conventional web applications. These interfaces must balance simplicity for casual users with the power and flexibility required by technical operators.

Key considerations for control system frontends include:

- **Real-time responsiveness**: The interface must respond immediately to user inputs and reflect the current system state without noticeable delay.
- **Visual feedback**: Users need clear visual cues about system status, operations in progress, and potential issues.
- **Intuitive representation**: Complex technical concepts must be represented in visually intuitive ways that match users' mental models.
- **Error prevention**: The interface should guide users toward correct operations and prevent configurations that could cause system failures.
- **Flexibility vs. guardrails**: The system must provide both flexibility for advanced users and guardrails for less experienced operators.

For the OpenVideoWalls frontend, these principles informed the design of layout configuration tools, content management interfaces, and synchronization control panels.

2.1.3 Real-time Data Visualization

Effective monitoring of video wall systems requires real-time visualization of complex performance metrics. The OpenVideoWalls paper defines four key metrics that are essential for evaluating video wall performance:

1. **Dropped Frames**: Counting frames omitted during playback
2. **Frame Duration**: Measuring the time each frame is displayed
3. **Stream Difference**: Quantifying temporal disparities between identical frames on different displays
4. **Frame Offset**: Measuring discrepancies between scheduled and actual presentation times

Visualizing these metrics effectively requires techniques from the field of data visualization, including time-series charts, and simplified representations that make technical data accessible to non-technical users.

2.2 Technologies

2.2.1 React.js[6]

React.js served as the primary frontend framework for the OpenVideoWalls control interface. Created by Facebook (now Meta) and released as an open-source project in 2013, React has become one of the most popular JavaScript libraries for building user interfaces.

React's component-based architecture was particularly well-suited to the OpenVideoWalls frontend, allowing us to create reusable UI elements for different parts of the control system. The virtual DOM (Document Object Model) implementation in React enabled efficient updates to the interface when system status changed or user interactions occurred, without requiring a full page refresh.

Key React features utilized in this project include:

- Functional components with hooks for state management
- Context API for sharing state across component hierarchies
- React Router for navigation between different control panels
- Error boundaries to gracefully handle component failures

2.2.2 TypeScript[7]

TypeScript was used to enhance code quality and developer experience by adding static typing to JavaScript. This was particularly valuable for the OpenVideoWalls frontend, as it helped prevent common runtime errors and provided better tooling support for refactoring and code navigation.

The use of TypeScript interfaces for component props, API responses, and state management ensured consistent data structures throughout the application. For example, in the StartSRT component:

```
const StartSRT: React.FC<{ setOutput: (msg: string) => void }> = ({ setOutput })
=> {
  // Implementation details
};
```

This explicit typing made the codebase more maintainable and reduced bugs related to incorrect prop types or data structures.

2.2.3 Vite.js[8]

Vite served as the build tool and development server for the OpenVideoWalls frontend. Vite's fast hot module replacement (HMR) significantly improved the development experience by providing near-instantaneous feedback when making changes to components.

The project leveraged Vite's environment variable system for configuration management, allowing the frontend to easily connect to different backend environments:

```
const apiUrl = import.meta.env.VITE_API_URL;
```

This approach simplified deployment across development, testing, and production environments without requiring code changes.

2.2.4 FFmpeg Integration[4]

The frontend interacted with FFmpeg, a powerful multimedia framework used in the backend for video processing and streaming. While FFmpeg itself ran on the server side, the frontend provided the interface for controlling its operation and configuring its parameters.

The integration with FFmpeg enabled the OpenVideoWalls system to divide video content across multiple displays, accounting for different screen arrangements and orientations. The frontend components allowed users to control how video content was processed and distributed without requiring direct knowledge of FFmpeg command-line syntax.

2.2.5 SRT Protocol Integration[2]

The Secure Reliable Transport (SRT) protocol, a key component of the OpenVideoWalls system as described in the research paper, required frontend integration for controlling stream settings and monitoring performance.

While the protocol itself operates at the backend level, the frontend provided interfaces for:

- Configuring SRT stream parameters
- Monitoring stream health and performance
- Controlling the SRT service lifecycle
- Managing stream routing to different displays

This integration was achieved through API endpoints that communicated with the SRT Relay component of the OpenVideoWalls system, allowing frontend users to control streaming behavior without needing direct knowledge of the underlying protocol.

2.2.6 Docker Integration

The OpenVideoWalls system used Docker containers to simplify deployment of the SRT streaming server. The frontend provided controls for managing these containers, including starting and stopping the Docker service that hosted the Simple Realtime Server (SRS)[3].

The Docker integration in the frontend allowed users to control the system infrastructure without requiring command-line access or Docker expertise. This approach made the system more accessible to non-technical users while maintaining the deployment benefits of containerization.

Chapter 3

Methodology

3.1 *Design*

3.1.1 User Research and Requirements Analysis

The design process for the OpenVideoWalls interface was guided primarily by personal exploration and an intuitive understanding of what would make the system user-friendly and effective. I considered the types of users who might interact with the system, including:

- **Technical operators:** Individuals familiar with managing traditional video wall systems
- **Commercial users focused on sustainability:** Organizations interested in reducing e-waste through display reuse
- **Hobbyists and makers:** Individuals who enjoy building tech projects with recycled hardware
- **Exhibition designers:** Professionals setting up visual displays for museums, galleries, or retail environments

Based on this informal analysis and with direction from my professor and advisors to create a frontend aligned with my own vision, I identified several key interface goals:

- **Simplicity:** A clean UI that hides technical complexity
- **Visual configuration:** Easy-to-use tools for arranging and managing screen layouts
- **Real-time feedback:** Indicators for server and stream status
- **Content control:** Basic mechanisms for uploading and scheduling video playback
- **Troubleshooting support:** Clear messaging for common system states and errors

The design was also heavily influenced by modern web development practices. I used React.js for its component-based architecture, which allowed me to structure the UI around modular features like stream control, screen settings, and upload panels.

3.1.2 UI/UX Design Principles

The final UX/UI design for the OpenVideoWalls frontend was guided by several core principles:

- **Progressive disclosure:** Complex features were organized in layers, with basic functions immediately accessible and advanced options available through progressive interaction.
- **Spatial metaphor:** The interface used spatial representations that matched the physical reality of the video wall, making it easier for users to understand the relationship between interface actions and physical outcomes.
- **Consistency:** UI elements, terminology, and interaction patterns were standardized across the application to reduce cognitive load and make the system more learnable.
- **Feedback loops:** Every user action received immediate visual feedback, and ongoing processes provided progress indicators to maintain user confidence.
- **Error prevention:** The interface guided users toward correct actions and provided validation to prevent configuration errors that could affect video wall performance.

These principles informed the visual design, which used a clean, modern aesthetic with a focus on readability and clear visual hierarchy. The color scheme emphasized contrast for readability while giving the whole front page a clean and minimal look.

3.2 *Implementation*

3.2.1 Frontend Architecture

The frontend application was structured following a modular architecture to ensure maintainability and separation of concerns, based on the directory structure provided:

```
src/
├── assets/              # Static assets for the application
├── components/          # Reusable UI components
│   ├── NetworkConfiguration.tsx   # Network settings configuration
│   ├── ScreenLayout.tsx        # Display layout management
│   ├── StartDocker.tsx         # Docker service initialization
│   ├── StartSRT.tsx            # SRT service controls
│   ├── StatusDisplay.tsx       # System status visualization
│   ├── StopDocker.tsx          # Docker service termination
│   ├── StopSRT.tsx             # SRT service termination
│   └── SystemControls.tsx      # Central system control panel
├── hooks/               # Custom React hooks
│   ├── useScreenManagement.tsx # Display management logic
│   ├── useStreamSettings.tsx   # Stream configuration hooks
│   └── useSystemCommands.tsx   # System command abstractions
├── FileUpload.tsx       # Content upload component
├── SRTControlPanel.tsx  # SRT protocol configuration
├── SystemDiagram.tsx    # System visualization component
├── SystemStatusBar.tsx  # Status monitoring component
├── index.css            # Global styles
├── main.tsx             # Application entry point
└── vite-env.d.ts        # TypeScript environment definitions
```

3.2.2 Component Development

The `SRTControlPanel` serves as the central component of the OpenVideoWalls frontend, orchestrating the interaction between various subcomponents and custom hooks. Below is a concise version of this component that illustrates the core architecture while omitting some implementation details for clarity:

```
const SRTControlPanel: React.FC = () => {
  // State management
  const [output, setOutput] = useState('Waiting for input...');
  const [orientation, setOrientation] = useState<'horizontal' |
'vertical'>('horizontal');

  // Custom hooks for different functional domains
  const { systemStatus, handleSystemCommand } = useSystemCommands(setOutput);
  const { screens, screenCount, handleIPChange } = useScreenManagement(setOutput);
  const { streamSettings, handleFileUpload } = useStreamSettings(setOutput);

  return (
       <div className="srt-control-panel">
       <SystemControls {...systemProps} />
       <StatusDisplay {...statusProps} />
       <NetworkConfiguration {...networkProps} />
       <ScreenLayout {...layoutProps} />
       </div>
  );
};
```

This implementation demonstrates several key architectural decisions:

1. **Separation of Concerns**: The component delegates specific functionality to custom hooks (`useSystemCommands`, `useScreenManagement`, and `useStreamSettings`), keeping the main component focused on composition and state coordination.
2. **Prop Drilling Pattern**: The component passes state and callback functions down to child components, establishing clear data flow throughout the application.
3. **Centralized State Management**: The component maintains local state for orientation and output messages, while delegating domain-specific state to custom hooks.

4. **TypeScript Integration**: The use of TypeScript with explicit type annotations (like `'horizontal' | 'vertical'` for orientation) enhances code reliability and developer experience through static type checking.

This architecture reflects modern React best practices and provides a solid foundation for the OpenVideoWalls frontend, allowing for easy maintenance and future extensions.

3.2.3 Custom React Hooks

To facilitate the backend communication necessary for the OpenVideoWalls system, I implemented a set of custom React hooks that encapsulate specific domains of functionality, with each component using the standard fetch API for HTTP communication. A representative example is the StartSRT component:

```
import React from 'react';

const StartSRT: React.FC<{ setOutput: (msg: string) => void }> = ({ setOutput })
=> {
  const handleStart = async () => {
       try {
       const res = await fetch(`${import.meta.env.VITE_API_URL}/start_srt`, {
       method: "POST",
       headers: {
       'Content-Type': 'application/json'
       },
       body: JSON.stringify({}) // Empty object, but with proper JSON content type
       });

       const data = await res.json();
       if (res.ok) {
       setOutput(`✅ SRT started. ${data.message || ''}`);
       } else {
       setOutput(`❌ SRT start failed: ${data.error}`);
       }
       } catch (err) {
       setOutput("❌ Failed to connect to backend for SRT start.");
       }
  };
  return <button onClick={handleStart}>Start SRT</button>;
};

export default StartSRT;
```

This component demonstrates the pattern used throughout the application:

- Each functional component accepts a `setOutput` function to provide feedback to users
- Communication with the backend uses the standard fetch API with proper error handling
- Environment variables (`import.meta.env.VITE_API_URL`) are used to configure backend URLs
- Success and error states are clearly communicated to users with visual indicators (✅/❌)

The custom hooks I developed (`useSystemCommands`, `useScreenManagement`, and `useStreamSettings`) build upon this foundation, abstracting away the details of API communication and providing a clean, React-oriented API for the UI components. By delegating specific functionality to these specialized hooks, the main components remain focused on composition and presentation, leading to a more maintainable and testable codebase.

3.2.4 API Communication Pattern

The frontend components in the OpenVideoWalls system follow a consistent pattern for API communication, using the fetch API for HTTP requests. Each component that needs to communicate with the backend implements a similar approach:

- **Environmental Configuration:** Backend API URLs are configured through environment variables using Vite's environment variable system (`import.meta.env.VITE_API_URL`), making the application easily deployable to different environments.
- **Standard Error Handling**: All API calls include consistent error handling with clear user feedback, as demonstrated in components like StartSRT and StopSRT.
- **Status Feedback:** Each component uses the `setOutput` function passed from parent components to provide clear status updates to users, including success confirmations and error details.

- **Typed Interfaces:** TypeScript is used throughout to ensure type safety in API communications, with properly defined interfaces for request and response data structures.

This standardized approach to API communication helps maintain consistency across the application and simplifies debugging when issues arise. It also makes the codebase more maintainable as new developers can quickly understand the pattern and apply it to new components.

The custom hooks build upon this foundation by encapsulating related API calls and state management logic, providing a higher-level interface for the main components. For example, the `useSystemCommands` hook manages the state and API calls related to Docker and SRT service control, while the `useScreenManagement` hook handles display configuration and network settings.

3.2.5 Frontend-Backend Communication

The frontend components communicate with the Flask backend through a set of RESTful API endpoints. Based on the backend code provided, the communication architecture includes:

Configuration Management API:

- The `/set_screen_ips` endpoint allows the frontend to save screen IP addresses, screen count, and orientation configuration.
- The `NetworkConfiguration` component sends this data to maintain persistent system settings.

Docker Container Control:

- The `/start_docker` and `/stop_docker` endpoints enable the frontend to manage the SRS (Simple Realtime Server) Docker container.
- The `SystemControls` component uses these endpoints to control the streaming infrastructure.

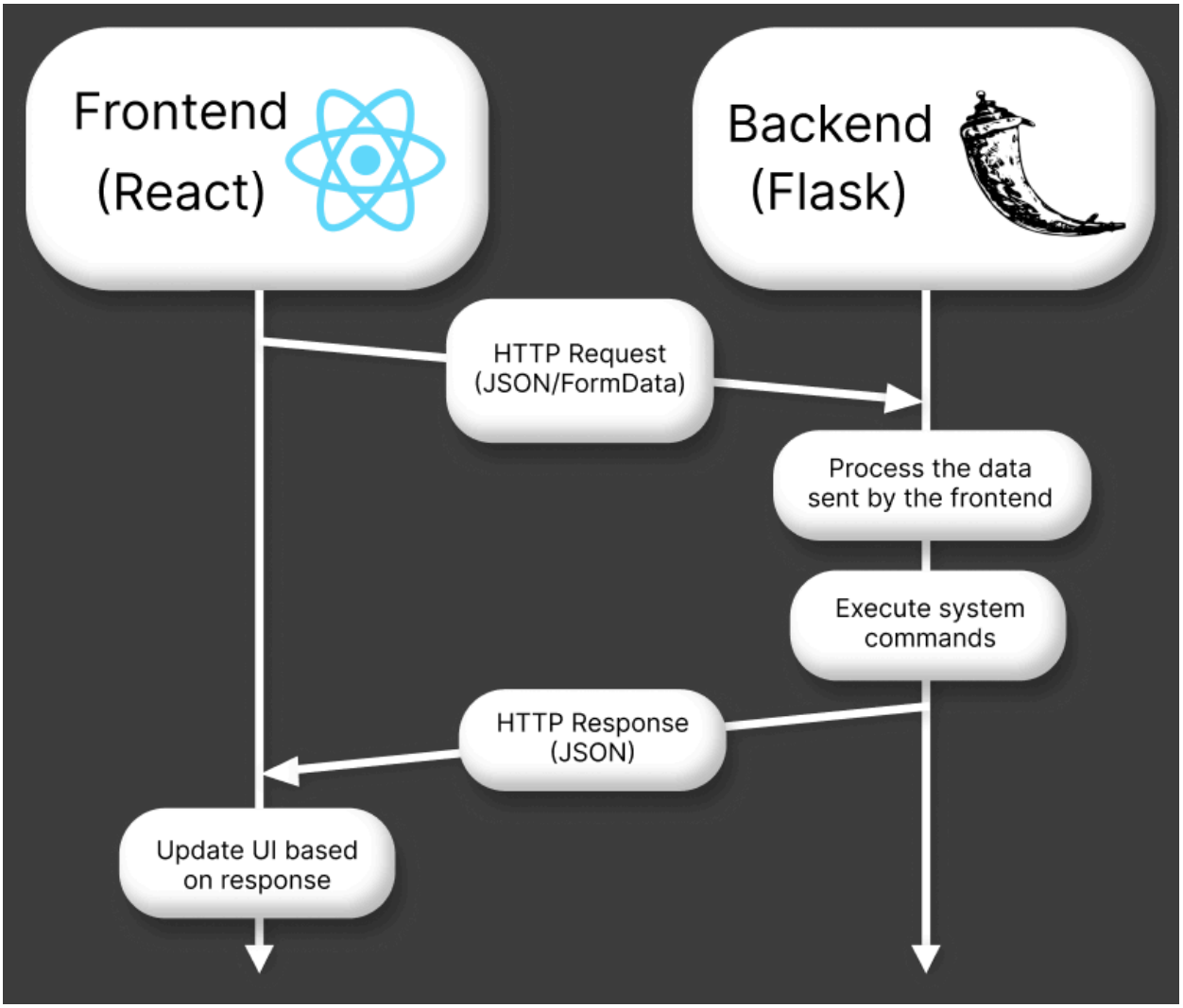
SRT Stream Management:

- The `/start_srt` and `/stop_srt` endpoints allow the frontend to start and stop video streams with proper segmentation for the video wall.
- The `SRTControlPanel` component provides interface controls that connect to these backend functions.

Content Upload:

- The `/upload_video` endpoint facilitates video file uploads and returns metadata about the uploaded video.
- The `FileUpload` component manages this file transfer and displays the returned resolution information.

The communication flow follows a standard request-response pattern:



Error handling is implemented on both sides of the communication:

- **The backend** captures errors with try/except blocks and returns appropriate HTTP status codes and error messages
- **The frontend** uses the setOutput function to display status messages and error notifications to the user

For example, when the SRTControlPanel initiates a video stream, the following data flow occurs:

- The frontend collects configuration data (screen count, orientation, IPs) from its state
- This data is sent as a JSON payload to the /start_srt endpoint
- The backend processes this request, generates an FFmpeg command with the appropriate filter complex for splitting the video
- The backend returns detailed information about the stream configuration
- The frontend updates its UI to reflect the new streaming state

This tight integration between frontend and backend ensures that the OpenVideoWalls system operates as a cohesive whole, with the React frontend providing an intuitive interface to the FFmpeg-based video processing backend.

3.2.6 Cross-Origin Resource Sharing (CORS) Implementation

A critical aspect of the frontend-backend integration was implementing proper Cross-Origin Resource Sharing (CORS) to enable secure communication between the React frontend and Flask backend when running on different origins. In the OpenVideoWalls project, this was accomplished using the Flask-CORS extension as seen in the backend implementation:

```
from flask import Flask, request, jsonify
from flask_cors import CORS
import subprocess
import json
import os
import traceback
```

```
import psutil

app = Flask(__name__)
CORS(app, supports_credentials=True)
```

The CORS implementation served several important purposes in the project:

- **Security Enhancement:** By explicitly defining which origins could access the API, the backend reduced the risk of unauthorized access.
  The `supports_credentials=True` parameter enabled the frontend to send authenticated requests when needed.
- **Local Development Support:** During development, the React frontend typically ran on a different port (e.g., 3000) than the Flask backend (port 5000). CORS configuration allowed seamless communication between these local development servers.
- **Production Deployment Flexibility:** The CORS setup provided flexibility for various deployment scenarios, such as hosting the frontend on a content delivery network (CDN) while keeping the backend on a separate server.
- **Browser Security Compliance:** Modern browsers enforce the Same-Origin Policy, which restricts how documents or scripts loaded from one origin can interact with resources from another origin. The proper CORS headers enabled the frontend to make cross-origin requests without browser security restrictions.

In the OpenVideoWalls implementation, CORS was configured at the application level to apply to all routes. This broad configuration was appropriate for our use case since all API endpoints needed to be accessible from the frontend. For a production environment with more stringent security requirements, the CORS policy could be refined to specific routes or origins as needed.

The frontend React components were designed with awareness of these CORS considerations, particularly when making fetch requests to the backend API endpoints. The custom hooks (`useSystemCommands` , `useScreenManagement` , and `useStreamSettings`) encapsulated the proper handling of cross-origin requests, including setting appropriate headers and handling CORS-related errors.

3.3 *Testing*

3.3.1 Unit Testing

Unit tests were written for individual components and utility functions using Jest and React Testing Library. The testing strategy focused on:

- **Component rendering:** Verifying that components rendered correctly with different props
- **User interactions:** Testing that components responded appropriately to user events
- **State management:** Confirming that components maintained and updated state correctly
- **Error handling:** Ensuring components gracefully handled error conditions

For example, here's a simplified test for the StartSRT component:

```
import { render, screen, fireEvent } from '@testing-library/react';
import StartSRT from './components/StartSRT';

test('calls the correct API endpoint when clicked', async () => {
  // Mock fetch API
  global.fetch = jest.fn().mockResolvedValue({
      ok: true,
      json: async () => ({ message: 'SRT service started
successfully' })
  });

  // Mock output function
  const mockSetOutput = jest.fn();

  // Render component with props
  render(<StartSRT setOutput={mockSetOutput} />);

  // Find and click the button
  const startButton = screen.getByText('Start SRT');
  fireEvent.click(startButton);

  // Verify fetch was called with correct URL and method
```

```
  expect(global.fetch).toHaveBeenCalledWith(
      expect.stringContaining('/start_srt'),
      expect.objectContaining({ method: 'POST' })
  );

  // Wait for async operation and verify output was set
  await new Promise(resolve => setTimeout(resolve, 0));
  expect(mockSetOutput).toHaveBeenCalledWith(
      expect.stringContaining('✅ SRT started')
  );
});
```

These tests helped ensure that components worked correctly in isolation and maintained their expected behavior during refactoring.

3.3.2 Integration Testing

Integration tests verified that different parts of the application worked correctly together. These tests focused on:

- Hook interactions: Testing that custom hooks properly integrated with components
- State flow: Verifying that state changes triggered appropriate UI updates
- Feature workflows: Testing end-to-end workflows like configuring a display layout

For API interactions, a simulated backend environment was created to test frontend-backend communication without relying on the actual Flask server:

```
// Mock API service for testing
const createMockApiService = () => {
  const mockScreenIPs = {};
  let mockScreenCount = 2;
  let mockOrientation = 'horizontal';

  return {
      setScreenIPs: async (data) => {
      Object.assign(mockScreenIPs, data.ips);
      mockScreenCount = data.screenCount;
      mockOrientation = data.orientation;
      return { success: true };
      },

      startDocker: async () => {
      return { success: true, container_id: 'mock-container-123' };
      },

      startSRT: async () => {
      return { success: true, pid: 12345 };
      },

      // Additional mock endpoints...
  };
};
```

Using this mock service, integration tests could verify that user interactions triggered the correct series of API calls and UI updates.

3.3.3 Academic Evaluation and Feedback

I presented the application to my professor for academic evaluation and feedback. This review process focused on both technical implementation and usability aspects from an educational perspective.

The feedback sessions with my professor included:

- **Code review**: Examination of the React component architecture, TypeScript implementation, and custom hooks
- **Interface demonstration**: Walkthrough of the main application features and workflows
- **Technical discussion**: Analysis of the integration approach with the backend system
- **Academic assessment**: Evaluation of how the implementation met the project objectives

Feedback from these sessions was categorized into several areas:

- **Architecture considerations**: Suggestions for improving component organization and state management
- **Implementation quality**: Assessment of code quality, TypeScript usage, and adherence to React best practices
- **Technical documentation**: Recommendations for better documenting the API integration points
- **Academic alignment**: Evaluation of how well the implementation demonstrated course concepts

This academic feedback provided valuable guidance for refining the implementation before project completion. The professor's perspectives helped ensure that the frontend not only met the functional requirements of the OpenVideoWalls system but also demonstrated sound software engineering principles and academic rigor.

Chapter 4

Results

4.1 Frontend Interface Overview

The completed OpenVideoWalls frontend provides a comprehensive interface for controlling and monitoring video wall systems built with recycled heterogeneous displays. The interface is centered around the `SRTControlPanel` component, which serves as the main control center for the entire system. This panel integrates four key functional areas:

- **System Controls:** Provides buttons and indicators for managing the underlying Docker containers and SRT streaming services. Users can start and stop services with visual feedback on the current system status.
- **Status Display:** Shows real-time output messages, error logs, and status information. This component also includes stream settings controls and file upload functionality for video content.
- **Network Configuration:** Allows users to configure the IP addresses and network settings for each display in the video wall. The interface adapts based on the number of screens and their orientation.
- **Screen Layout:** Enables visual configuration of the screen arrangement, including setting the screen count, orientation (horizontal or vertical), and assigning specific video streams to each display.

This interface design makes complex video wall configuration accessible to users with varying levels of technical expertise, supporting the project's goal of promoting display recycling by lowering technical barriers.

4.2 Key Features Implemented

Several key features were implemented in the frontend to address the specific challenges of managing video walls with recycled heterogeneous displays:

- **Flexible Display Configuration:** The ScreenLayout component allows users to configure both the number of displays and their orientation (horizontal or vertical). This flexibility is crucial for accommodating different types of recycled displays and physical installation constraints.
- **Stream Assignment System:** The interface enables users to assign specific video streams to each display in the video wall. This feature is essential for the proper division of content across multiple displays with potentially different specifications.
- **Network Management:** The `NetworkConfiguration` component provides tools for configuring the IP addresses of each display, ensuring proper communication between the server and client displays as described in the OpenVideoWalls paper.
- **System Service Controls:** The `SystemControls` component provides intuitive buttons for starting and stopping the Docker containers and SRT streaming services that form the backend of the OpenVideoWalls system. Visual indicators show the current status of each service.
- **Real-time Status Feedback:** The `StatusDisplay` component shows immediate feedback on system operations, error messages, and status updates. This feedback helps users troubleshoot issues during setup and operation.
- **Video Processing Integration:** The frontend integrates with FFmpeg-based video processing on the backend, which handles video segmentation for the video wall. The interface provides controls for uploading video content and configuring how it's split across displays.

4.3 Performance and Usability

The frontend implementation was evaluated based on several key criteria to ensure it met the project requirements:

Interface Responsiveness:

The React application maintains responsive interaction even during system operations Updates to the status display occur in real-time without blocking the user interface State updates are optimized to minimize unnecessary re-renders

Cross-browser Compatibility:

The interface functions correctly in modern browsers (Chrome, Firefox, Safari)
No browser-specific features are used that would limit accessibility

Error Handling:

The system provides clear error messages when operations fail
Network errors and backend communication issues are gracefully handled
User input validation prevents configuration mistakes

User Experience:

The logical grouping of related controls enhances usability
The interface adapts to different screen sizes and orientations
Visual feedback confirms user actions and system state changes

Based on feedback from the project supervisor and academic advisors, the interface successfully abstracts the technical complexity of the OpenVideoWalls system. The design decisions were validated by academic review, confirming that the frontend would make the system accessible to potential users with different levels of technical expertise.

4.4 Academic and Expert Feedback

Throughout the development process, I participated in weekly meetings with my advisor and professor to review progress and align on project direction. These meetings focused primarily on functional requirements and overall system performance rather than specific implementation details.

Key aspects of the supervision process included:

- Regular progress updates where I demonstrated new features and functionality
- Discussion of system requirements and how the frontend should support the OpenVideoWalls concept
- Clarification on how the application should work from an end-user perspective
- General guidance on project timeline and priorities

The supervision maintained a focus on the application's functionality and usability, with my advisors providing direction on what the system should accomplish rather than prescribing specific coding approaches or implementation techniques. But on some occasions, my supervisor has also introduced me to some systems of services that had made the development of OpenVideoWalls much smoother and faster. This allowed me to exercise creativity and technical judgment combined with the knowledge given by my advisors to determine the best ways to implement the required features.

The feedback received was primarily focused on ensuring that the frontend would effectively support the OpenVideoWalls system's core goals of making video wall technology accessible and promoting the reuse of display hardware. This guidance was valuable in maintaining alignment between the frontend implementation and the broader project objectives outlined in the research paper.

Chapter 5

Conclusions

Current Progress and Future Work

5.1 Summary of Current Accomplishments

During the first phase of this internship, I have made significant progress in developing the React-based frontend for the OpenVideoWalls system with the following key accomplishments:

- **Established the core control interface**: Implemented the foundational SRTControlPanel component that organizes the main functional areas of the application, including system controls, status display, network configuration, and screen layout.
- **Implemented system service controls**: Created components for managing Docker containers (`StartDocker.tsx` , `StopDocker.tsx`) and SRT streaming services (`StartSRT.tsx` , `StopSRT.tsx`), enabling basic control of the video wall infrastructure.
- **Developed network configuration tools**: Built the `NetworkConfiguration.tsx` component for configuring IP addresses of client displays, establishing the communication channels needed for the video wall system.
- **Created flexible display layout management**: Implemented the `ScreenLayout.tsx` component that allows configuring the number and orientation of displays in the video wall, supporting the key goal of accommodating heterogeneous recycled displays.
- **Integrated with backend services**: Established communication patterns between the frontend and Flask backend using the fetch API, with proper error handling and status feedback to users.

These initial accomplishments have established a solid foundation for the OpenVideoWalls frontend, with key functionality in place for basic video wall configuration and control.

These accomplishments collectively contribute to making the OpenVideoWalls system more accessible and user-friendly, potentially increasing its adoption and environmental impact through greater reuse of discarded displays.

5.2 Issues and Obstacles

Several challenges are being addressed in the ongoing development:

1. **State management complexity**: As the application grows, managing state across multiple components has become increasingly complex. The custom hooks approach (`useSystemCommands`, `useScreenManagement`, and `useStreamSettings`) is showing promise but requires further refinement.
2. **Error handling robustness**: The current error handling for backend communication needs strengthening to handle various failure scenarios consistently, especially for longer-running operations like video processing.
3. **Configuration persistence**: Ensuring user configurations reliably persist between sessions remains a challenge that requires better coordination with the backend's configuration management system.

5.3 Future Directions

The following key features are planned for the next phase of development:

- **Video Feed Grouping System:** A critical planned feature is the implementation of a grouping mechanism that will allow one frontend instance to control multiple video feeds. This will enable more complex video wall configurations where displays can be organized into logical groups that show different content streams.
- **Enhanced Content Management:** Expanding the existing file upload functionality to provide better organization and preview of video content before deployment to the video wall.
- **Improved Monitoring Dashboard:** Developing more comprehensive monitoring tools to visualize the synchronization metrics defined in the OpenVideoWalls paper.

- **User Documentation:** Creating integrated help and documentation to assist users in setting up and configuring their recycled display video walls.
- **Performance Optimization:** Identifying and addressing performance bottlenecks, particularly in the real-time status updates and configuration management.

5.4 Timeline for Completion

The current development timeline anticipates the following milestones:

- **Mid-Term** (**Current Stage**)**:** Core functionality established, basic control and configuration components implemented.
- **Video Feed Grouping Implementation:** Estimated completion in 3 weeks, this will be the next major focus area.
- **Content Management Enhancements:** Planned for weeks 4-5 of the remaining internship period.
- **Monitoring and Documentation:** Scheduled for the final 2-3 weeks of the internship.
- **Final Testing and Refinement:** The last week will be dedicated to comprehensive testing and addressing any remaining issues.

The project is progressing according to schedule, with regular weekly meetings with advisors ensuring alignment with the overall OpenVideoWalls system goals.

References

1. *Zhu, Z., Tang, Z., Nassereldine, A., Xiong, J., & Wei, S. (2024). OpenVideoWalls: an Open-Source System for Building Video Walls with Recycling Heterogeneous Displays. ACM Multimedia Asia (MMASIA '24).*
2. *Haivision. (2023). Secure Reliable Transport (SRT) Protocol. Github Repository. Retrieved from https://github.com/Haivision/srt*
3. *Simple Realtime Server. (2024). Retrieved from https://github.com/ossrs/srs*
4. *Tomar, S. (2006). Converting video formats with FFmpeg. Linux Journal, 2006(146), 10.*
5. *Global E-waste Monitor. (2024). The Global E-waste Monitor 2024 – Electronic Waste Rising Five Times Faster than Documented E-waste Recycling. United Nations.*
6. *React Team. (2023). React: A JavaScript library for building user interfaces. Retrieved from https://reactjs.org/*
7. *TypeScript Team. (2023). TypeScript: JavaScript with syntax for types. Retrieved from https://www.typescriptlang.org/*
8. *Vite.js Team. (2023). Vite: Next Generation Frontend Tooling. Retrieved from https://vitejs.dev/*