Face Recognition Access Control System - Technical Report

I. Project Context

1.1 Context

In today's fast-changing world, the need for secure and efficient access control systems has become more important than ever. Traditional systems, such as key cards, passwords, or manual checks, are no longer enough to meet the demands of modern organizations. These older methods often fail to provide the level of security and convenience that both businesses and users require.

Our project addresses this challenge by creating a cutting-edge facial recognition access control system. Unlike traditional systems, this solution uses advanced technology to verify identities quickly, securely, and without physical contact.

This report explains how the system works, the challenges it solves, and the possibilities it offers for the future.

1.1.1 Problem Statement

Many organizations still use traditional access control methods. While these systems have worked for years, they come with serious limitations:

1. **Lost or Stolen Credentials:** Physical access cards can be lost, stolen, or duplicated. This creates a major security risk.
2. **Delays and Inefficiency:** During busy times, systems that rely on manual checks or swiping cards can cause delays. Long lines and waiting times are frustrating for users and inefficient for organizations.
3. **Manual Oversight:** Traditional systems often require staff to monitor access points, which increases labor costs and introduces human error.
4. **Health Concerns:** After the COVID-19 pandemic, people are more cautious about touching shared devices, like fingerprint scanners or keypads. Touchless solutions are now preferred.

Recognizing these problems, we developed a facial recognition system to address these shortcomings. Our solution is touchless, fast, and secure, making it ideal for modern access control needs.

1.1.2 Solution

Our solution combines artificial intelligence (AI) and facial recognition technology in a way that is both user-friendly and reliable. The system uses a JavaFX application for the user interface and a Flask-based backend to handle facial recognition tasks.

When a user approaches the system, their face is scanned and compared to the database in real time. If the face matches a stored profile, access is granted instantly. The system also keeps a record of every access attempt, which is useful for audits and security checks.

This approach eliminates the need for physical keys, cards, or passwords. Users only need their face to gain access, making the system both convenient and secure.

1.2 Project Architecture

1.2.1 Technical Design

Our system is built using a **client-server model**, which means the work is divided between two main parts:

1. **Frontend (User Interface):**

The frontend is the part of the system that users see and interact with.

   a. Built with **JavaFX**, it provides a simple and intuitive graphical interface.
   b. The application uses **OpenCV** to process video in real time, detecting faces as users approach the system.
   c. We used the **MVC (Model-View-Controller)** pattern, which keeps the code organized and easy to update in the future.

2. **Backend (Server):**

The backend is the part of the system that handles complex tasks like analyzing and matching faces.

   a. Built with **Flask**, a lightweight web framework.

b. Uses the **face_recognition library**, a powerful tool for facial detection and matching.
c. Stores user profiles and access logs in a **SQLite database**.

The frontend and backend communicate through **Flask REST APIs**. Data is exchanged in JSON format, which is lightweight and easy to process, ensuring fast communication between the two parts.

1.2.2 Development Approach

To ensure the system was reliable and easy to expand, we followed a modular development approach. Each part of the system was built as an independent component, which means we can update or improve one part without affecting the others.

We also implemented **role-based access control**:

- **Administrators**: Manage user profiles, configure the system, and view access logs.
- **Regular Users**: Simply scan their face to gain access.

This structure ensures that sensitive administrative features are protected, while regular users enjoy a simple, hassle-free experience.

II. Analysis and Design

2.1 Requirement Analysis

The system's requirements were carefully analyzed to ensure comprehensive coverage of both functional and non-functional aspects.

Functional Requirements:

1. User Authentication
    a. Real-time facial recognition
    b. Admin login with username/password

    c. User verification through facial matching
2. User Management
    a. Add new users with facial data
    b. View and manage user records
    c. Update user information
3. Access Control
    a. Real-time access verification
    b. Access logging
    c. History tracking
4. Administrative Functions
    a. User management dashboard
    b. Access history viewing
    c. System configuration

Non-Functional Requirements:

1. Performance
    a. Real-time face detection and recognition
    b. Quick response time for access decisions
    c. Efficient database operations
2. Security
    a. Secure storage of facial data
    b. Protected admin access
    c. Encrypted communication (encoding the user faces)
3. Usability
    a. Intuitive user interface
    b. Clear feedback on recognition status
    c. Easy navigation for administrators

2.2 UML Design

The system's architecture is documented through detailed UML diagrams that illustrate the relationships between components and the flow of operations.

2.2.1 Use Case Diagram

```
@startuml Use Case Diagram
left to right direction
actor Admin
```

```
actor User

rectangle "Face Recognition Access System" {
    usecase "Login" as UC1
    usecase "Authenticate with Face" as UC2
    usecase "Manage Users" as UC3
    usecase "View History" as UC4
    usecase "Add New User" as UC5
    usecase "Delete User" as UC6
    usecase "Access Building" as UC7
}

Admin --> UC1
Admin --> UC3
Admin --> UC4
Admin --> UC5
Admin --> UC6

User --> UC2
User --> UC7

UC2 ..> UC7 : includes
UC5 ..> UC3 : extends
UC6 ..> UC3 : extends

@enduml
```

2.2.2 Class Diagram

```
```mermaid
classDiagram
    class AdminController {
        + void initialize()
        + void onSwitchSubjectToMainAdmin()
        + void onSwitchSubjectToUsers()
        + void onSwitchSubjectToAddAdmin()
    }

    class DatabaseConnection {
        + static Connection getConnection()
    }

    class History {
        - int id
        - String dateTime
```
```

```
    - User user
    + String getDateTime()
    + User getUser()
}

class HistoryController {
    + void initialize()
    - ObservableList~History~ getDataHistory()
}

class LoginController {
    + void initialize()
    + void switchToAdminView()
    - void switchToUserView()
    - void sendImageToBackEnd()
}

class User {
    - int id
    - String firstName
    - String secondName
    + String getFirstName()
    + String getSecondName()
    + int getId()
}

class UserAddController {
    + void initialize()
    - void saveUserToDatabase()
}



class UsersController {
    + void initialize()
    - ObservableList~User~ getDataUsers()
}

class ApiClient {
    + static List~History~ getHistory()
    + static List~User~ getUser()
}

%% Relationships
UserAddController -- DatabaseConnection
AdminController -- UserAddController
AdminController -- HistoryController
```

```
    AdminController -- UsersController
    HistoryController -- History
    HistoryController -- ApiClient
    ApiClient -- DatabaseConnection
    UsersController -- ApiClient
    UsersController -- User
    LoginController -- AdminController
```

2.2.3 Sequence Diagram

2.2.3.1 User Sequence Diagram

```mermaid
sequenceDiagram
    participant User
    participant Interface (Frontend)
    participant Backend
    participant Database

    User->> Interface (Frontend): Start Application
    Interface (Frontend)->>Interface (Frontend): Initialize Camera
    User->>Interface (Frontend): Face Detection
    Interface (Frontend)->>Backend: Send Face Data
    Backend->>Database: Verify User
    Database-->>Backend: User Data
    Backend-->>Interface (Frontend): Authentication Result
    Interface (Frontend)-->>User: Access Response
```

2.2.3.2 Adminastrator Sequence Diagram

```mermaid
sequenceDiagram
    participant Admin
    participant Interface (Frontend)
    participant Backend
    participant Database

    Admin->>Interface (Frontend): Enter Username and Password
    Interface (Frontend)->>Backend: Send Login Credentials
    Backend->>Database: Verify Admin Credentials
    Database-->>Backend: Admin Verification Result
    Backend-->>Interface (Frontend): Login Success/Failure
    Interface (Frontend)-->>Admin: Access Granted/Denied
```

```
    Admin->>Interface (Frontend): Add User
    Interface (Frontend)->>Backend: Send User Details
    Backend->>Database: Save New User
    Database-->>Backend: User Added Confirmation
    Backend-->>Interface (Frontend): Add User Success

    Admin->>Interface (Frontend): Delete User
    Interface (Frontend)->>Backend: Send User ID
    Backend->>Database: Remove User
    Database-->>Backend: User Deletion Confirmation
    Backend-->>Interface (Frontend): Delete User Success

    Admin->>Interface (Frontend): View Authentication History
    Interface (Frontend)->>Backend: Request Authentication Logs
    Backend->>Database: Retrieve Logs
    Database-->>Backend: Send Logs
    Backend-->>Interface (Frontend): Display Logs
    Interface (Frontend)-->>Admin: Show Authentication History
````

III. Demonstration

3.1 Front-End (JavaFX)

The front-end is built using JavaFX and is organized into multiple controllers and views:

- LoginController: Handles user authentication. It captures images from the webcam using OpenCV, sends them to the Flask server for facial recognition, and switches to the appropriate dashboard (Admin or User) based on the result.
- AdminController: Manages the admin dashboard, allowing admins to view user history, manage users, and add new users. It dynamically loads different FXML views (e.g., Admin-TableView.fxml, Admin-TableUsers.fxml) based on user actions.
- HistoryController and UsersController: Handle the display of user history and user lists, respectively, by fetching data from the Flask API.
- UserAddController: Allows admins to add new users by entering their details and selecting an image, which is then saved to the database.

3.2 Back-End (Flask Server)

The Flask server (server.py) handles facial recognition and database interactions:

- Facial Recognition: When an image is uploaded from the JavaFX client, the server uses the face_recognition library to detect faces, encode them, and compare them against stored encodings in the database. If a match is found, the user's information is returned, and their login history is updated.
- API Endpoints: The server provides endpoints for fetching user history (/FRapi/history), retrieving user data (/FRapi/users), and uploading images for facial recognition and updating the history (/upload).

3.3 Database (SQLite)

The SQLite database (ProjectFR.db) stores:

- User Data: Includes user IDs, names, image paths, and facial encodings.
- History: Tracks user login attempts with timestamps and user IDs.

3.4 How It Works

1. Login: The user logs in via the JavaFX interface. The webcam captures an image, which is sent to the Flask server for facial recognition. If a match is found, the user is allowed to enter.
2. Admin Dashboard: Admins can view user history, manage users, and add new users. The UI dynamically loads different views based on admin actions.
3. Facial Recognition: The Flask server processes uploaded images, compares them against stored encodings, and updates the database with login history.

IV. Assessment

4.1 Project Assessment

Our project has successfully accomplished its primary objectives, demonstrating significant progress and effectiveness. The facial recognition system we developed is characterized by several key attributes:

- **Fast:** It is capable of processing faces in real time, ensuring that there is minimal delay in recognition. This rapid processing allows for seamless interactions, making it highly efficient for various applications.

- **Accurate:** The system exhibits a high level of accuracy, as it can effectively recognize users even when they are in different lighting conditions. This adaptability enhances its reliability, ensuring that it performs well in a variety of environments.

- **User-Friendly:** Feedback from both administrators and regular users indicates that they found the system to be easy to navigate and operate. Its intuitive design contributes to a positive user experience, making it accessible to individuals with varying levels of technical expertise.

4.2 Team Assessment

The project's success reflects the dedication and collaboration of the two-person team, where each member played a critical role:

4.2.1. Technical Expertise

Both team members demonstrated strong technical skills:

- One focused on the **frontend**, building a user-friendly JavaFX interface and integrating OpenCV for real-time video processing.
- The other concentrated on the **backend**, developing the Flask REST API, implementing the `face_recognition` library, and managing the SQLite database for secure data storage.

Together, we ensured smooth communication between components and high system performance. Importantly, we contributed to each other's areas when needed.

4.2.2 Strengths and Areas for Growth

- **Strengths:** Our ability to divide tasks efficiently and focus on their strengths resulted in a well-designed, functional system.
- **Improvements:** Allocating more time for testing and cross-training could further enhance their workflow and adaptability in future projects.

4.3 Future Enhancements

The current system provides a solid foundation, but there’s room for growth. In the future, we plan to:

- Store data in the **cloud** for better scalability and easier management.
- Create a **mobile app** to let administrators manage the system remotely.

These features will make the system even more powerful, while keeping its current strengths intact.