

Distributed Face Recognition System - docs

Antonio Labinjan

July 11, 2025

Contents

1 Overview	2
2 System Architecture	2
3 Core Components	2
4 Technology Stack	3
5 Setup and Usage	3
5.1 Running the Central Server	3
5.2 Running a Node	3
6 Data Flow	3
7 Design Principles	4
8 Privacy and Security	4
9 Logging Interface	4
10 Example Outputs	4
11 Future Improvements	4
12 Key Functions and Routes (Draft)	4
13 Maintainer	5

1 Overview

This project enables face recognition in a distributed setting where low-cost client nodes perform local embedding extraction and offload classification to a centralized server.

Main characteristics:

- Lightweight on edge devices
- Scalable across multiple clients
- Centralized for efficient indexing and classification
- Privacy-aware, avoiding raw image transfer
- Face segmentation with MediaPipe Face Mesh
- Dynamic threshold tuning via live grid search

2 System Architecture

The architecture consists of:

- **Nodes (Clients)**: Perform detection, segmentation, embedding.
- **Central Server**: Performs classification and logging.
- **Embedding Queue**: Ensures fair processing order.

Diagram: Insert system architecture diagram image here.

3 Core Components

Nodes

- Detect faces using Haar cascades.
- Segment faces using MediaPipe Face Mesh.
- Extract CLIP embeddings.
- Compare to previous embedding to avoid duplicates.
- Send to server for classification.

Embedding Queue (Server-side)

- Buffers requests from nodes.
- Ensures sequential processing.

Central Server

- Loads dataset and builds FAISS index.
- Classifies embeddings using k-NN.
- Logs detection results.
- Provides web UI at `/log/html` and `/threshold_stats`.
- Auto reload dataset on restart.

Live Threshold Tuner

- Tests multiple thresholds per classification.
- Tracks classification performance per threshold.
- Results available via web.

4 Technology Stack

Component	Technology
Embedding Model	CLIP (openai/clip-vit-base-patch32)
Classifier	FAISS (L2)
Web Interface	Flask + JS
Face Detection	OpenCV Haar Cascade
Face Segmentation	MediaPipe Face Mesh
Communication	HTTP (JSON REST API)
Parallelism	Python threading + Queue
Containerization	Docker

5 Setup and Usage

5.1 Running the Central Server

Option A: Docker (Recommended)

```
docker pull antoniolabijnan/face-rec-central_server:latest
docker run -p 6010:6010 -v $(pwd)/dataset:/app/dataset antoniolabijnan/face-rec-
central_server:latest
```

Option B: Manual

```
pip install -r requirements.txt
python server.py
```

5.2 Running a Node

```
python node.py
```

6 Data Flow

1. Node detects and segments face.
2. CLIP embedding extracted and normalized.
3. Embedding sent via POST to server.
4. Server enqueues and classifies.
5. Result is logged and displayed.

7 Design Principles

- Nodes handle embedding; server handles classification.
- Queue ensures fairness and prevents overload.
- No images transferred, only vector data.
- Web UI provides live insight.

8 Privacy and Security

- No image/video transfer.
- Only numeric embeddings sent.
- Server performs all identity resolution.
- GDPR-compliant design.

9 Logging Interface

- Live view: `http://<server_ip>:6010/log/html`
- Threshold view: `http://<server_ip>:6010/threshold_stats`

10 Example Outputs

Include screenshots of:

- Log interface
- Threshold stats

11 Future Improvements

- WebSocket-based updates
- Persistent DB logging (e.g. SQLite)
- Central monitoring dashboard
- Node authentication
- Auto-retraining from confirmed captures

12 Key Functions and Routes (Draft)

Server Functions

- `add_known_face`, `load_dataset`, `build_index`
- `classify_face`, `classify_worker`, `check_for_intruder_alert`

Server Routes

- /classify, /log, /log/html, /
- /threshold_stats, /ping, /active_nodes/html, /intruder_alerts, /reload_dataset

Node Functions

- segment_face, classify_worker

13 Maintainer

Antonio Labinjan

<https://github.com/AntonioLabinjan>
Dockerhub: face-rec-central_server