Distributed Systems - Surveillance System Project

1st Cedric Sillaber 2nd Alan Gallo 3rd Frantisek Sova

I. Introduction

In our surveillance system, cameras are simulated using the WiseNET dataset. The system is designed to capture video streams and process them efficiently for real-time object and facial recognition. The video data undergoes preprocessing at the edge, minimizing the workload on the cloud, reducing latency, and optimizing the overall performance of the surveillance system. The system leverages edge devices and cloud services to ensure scalability, fast response times, and effective monitoring.

II. System Architecture

The video files are stored in camera containers running on a single EC2 instance 1. Each container parses a certain video file into images using OpenCV. These sequences are then sent to edge devices at regular intervals through HTTP, simulating a real-time video stream. Communication between components is handled through a WebSocket, which is capable of managing large payloads and scaling efficiently. Additionally, the EC2 instance houses an alarm container, completing the IoT layer of the system.

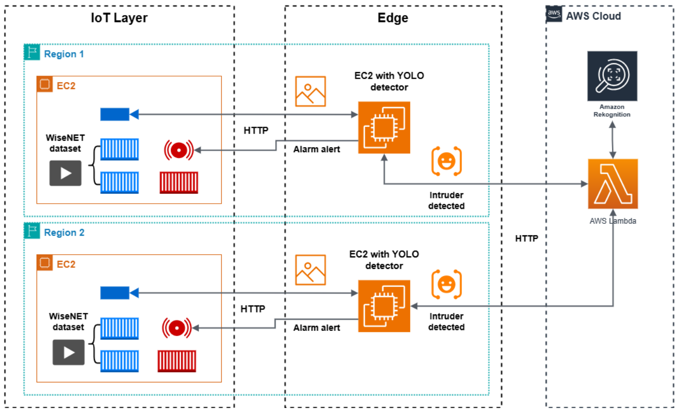


Fig. 1. Example of your architectural diagram.

The edge layer consists of a single EC2 instance simulating edge devices in different locations. At the edge, the video data is processed using the YOLO (You Only Look Once) algorithm to detect objects and people. When a person-like object is detected, the relevant data is forwarded to a separate instance in the cloud, ensuring the system can scale dynamically based on demand.

The cloud layer integrates with Amazon Rekognition to perform facial recognition, comparing the detected face against a collection of known individuals. If an unknown person is identified, the cloud server triggers an alert, which is sent back to the edge device and activates the alarm in the IoT layer. This architecture reduces the workload on the cloud, minimizes latency, and ensures the system can handle large volumes of video data in a responsive and efficient manner, making it a robust solution for surveillance applications.

III. Implementation Details

Our implementation consists of three distinct layers: IoT, Edge, and Cloud, each with specific responsibilities and components.

A. IoT Layer

The IoT layer consists of 4 camera containers and one alarm container for each edge device. We utilize the WiseNET dataset (set_3) for video simulation, which features two individuals appearing at different times in the scene - one simulating an intruder and another an employee. The camera containers are designed to have access to the complete set of videos, though each container selects only one based on a container-specific environment variable *CAMERA*. The video sets are *.avi-files* that are read sequentially and looped when the end is reached. These videos are processed using OpenCV's read() function to extract frame sequences. The system then transmits these frames to the edge device using Socket.IO library at intervals that simulate real-time frame rates. The alarm container maintains a connection to the edge server and remains in a waiting state until triggered by an event.

B. Edge Layer

The edge layer operates as an EC2 instance running the YOLO (You Only Look Once) algorithm for object detection. The implementation is Python-based and utilizes OpenCV for frame processing. Images from the IoT layer are received by the edge device using the Socket.IO library. When the system detects a person-like object using YOLO, it immediately forwards the relevant data to the cloud server through a REST API interface.

C. Cloud Layer

The cloud implementation centers around facial recognition and alert management through a REST API built using Python's Flask package to receive images from the edge layer. The system integrates directly with AWS Rekognition for facial recognition processing, with automatic provisioning of the Rekognition collection and known faces occurring at cloud server startup using boto3. When processing is complete, the result is returned in the HTTP response. Upon detection of an unknown person, the system triggers an alert, informs the edge layer through the HTTP response, and activates the appropriate alarm.

The system assumes a network topology where multiple cameras at a single location connect to one edge device. Each edge device manages multiple cameras but only one alarm. The cloud layer is designed to handle multiple edge devices simultaneously, with the capability to trigger location-specific alarms when intruders are detected.

D. Implementation Decisions

In the software stack where several options were available and we decided to go for specific ones. This section provides rationale for the choices made. This includes the programming language, communication libraries and detection algorithms.

1) Communication: The distributed system consists of several components that need to communicate in an efficient way - that's what makes it a distributed system. Namely two main interfaces need communictation:

- IOT - Edge communication
- Edge - Cloud communication

For the **IOT - Edge** communication we decided to use socket.io. This Python library enables communication with websockets. As we need to send a constant stream of data, this communication model seems to be the best fit. The protocol includes a full-duplex communication channel build ontop of the network layer. Its a consistent communication model that requires minimal overhead for setting up the connection.

This model handles many frames transfered very well and is very efficient. In comparision to HTTP that would require a new connection for every frame, this model is much more efficient.

For the **Edge - Cloud** communication we decided to use a REST API. This approach was implemented using Python's *Flask* package to allow the distributed processing of the data.

Our assumption for the choice of the communication model was that the data transfer between Edge and Cloud is not as frequent as between IOT and Edge. Ideally only a few requests per hour are sent to the cloud, as there are only few intruders in the office. When we would have tens of thousands of requests to the server per minute, we would argue there is something wrong in the office.

Given our assumption, the overhead of setting up a new connection for every request is not as significant. Having a constant connection open to the cloud would be a waste of resources.

The REST API is a stateless communication model that uses HTTP requests to transfer data. It is simple to implement and easy to understand.

Whenever the preprocessing model (YOLO) detects a person in a frame, it sends the frame in a HTTP request to the cloud for further processing. The cloud immediately processes the image and sends the response back in the HTTP response. We efficiently use the request and respons for our computation.

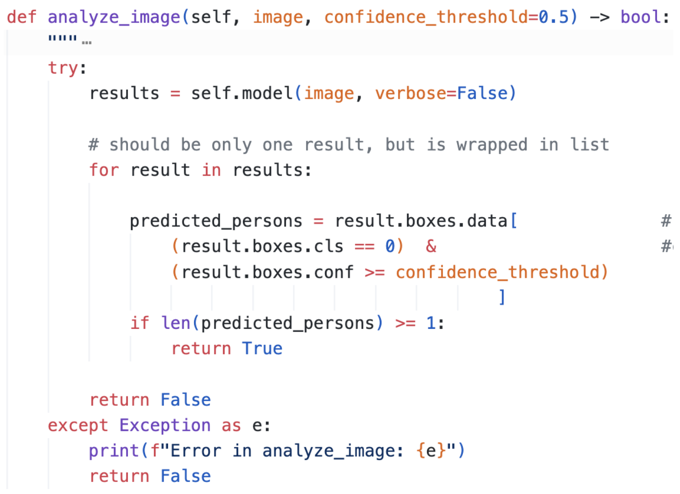
2) Dection Models: In the system two detection models are used, namely:

- YOLO: preprocessing images on Edge
- AWS Rekognition: facial recognition on Cloud

It was obvious to us to use Rekognition, as we already tested this AWS service in the course of the semester. It is a very powerful tool for facial recognition and very easy to use.

YOLO was chosen as the detection model for the edge, as it is a very efficient object detection model. It is able to process images in real-time and is very accurate. There were multiple problems in setting up the model in Docker, especially with dependencies for the *pytorch* library. We found a workaround using *ultralytics* package, that wrappes *torch* and special classes for YOLO into one package. Unfortunately the package is exceptionally big and exceeds the maximum image size of 1GB. We had no time to further optimize the size of the edge layer and decided to use the large image.

YOLO and Rekognition can be easily encapsulated and used in the system. For example, we implemented a wrapper class for yolo usage, so we can use simply the *yoloDetection.analyseImage(image)* function. Internally this function runs the yolo image detection, and returns True if a person was detected, otherwise False.



```
def analyze_image(self, image, confidence_threshold=0.5) -> bool:
    """...
    try:
        results = self.model(image, verbose=False)

        # should be only one result, but is wrapped in list
        for result in results:

            predicted_persons = result.boxes.data[                #
                (result.boxes.cls == 0)  &                        #
                (result.boxes.conf >= confidence_threshold)
                                                        ]
            if len(predicted_persons) >= 1:
                return True

        return False
    except Exception as e:
        print(f"Error in analyze_image: {e}")
        return False
```

IV. Testing Setup

In the first weeks of implementing the distributed system, we solely focused on simulating disjoint networks in Docker, utilizing the Docker-Networks feature.

For every layer, we created a separate Dockerfile.[layer]. A simple `docker-compose.yaml` file was used to orchestrate the containers. In this orchestration, the camera containers were connected to the edge device, which in turn was connected to the cloud server. The IOT containers are connected to the same network as the edge, the edge in addition is connected to the cloud network. The following *.yaml* shows the configuration for networking in docker containers.

```
networks:
  simulation_network:
    driver: bridge
  simulation_network_edge_cloud:
    driver: bridge

# then in the container specification:
cloud:
    build:
    context: .
    dockerfile: docker/Dockerfile.cloud
    image: cloud
    networks:
    # subscribe to the network
    - simulation_network_edge_cloud
```

This simple setup allowed us to test the communication between the layers and the correct processing of the data. Deployment was easy then, just a matter of spinning up the containers on the EC2 instances and adapting the IP addresses in the config file.

V. Deployment Setup

The entire distributed system runs on three AWS EC2 instances:

- IOT - layer: camera instances
- EDGE - layer: edge instance with YOLO
- CLOUD - layer: cloud instance with AWS Rekognition

The IoT layer components (4 camera containers and 1 alarm container) are consolidated on a single t2.micro instance, which is sufficient for simulating the video streams and alarm functionality.

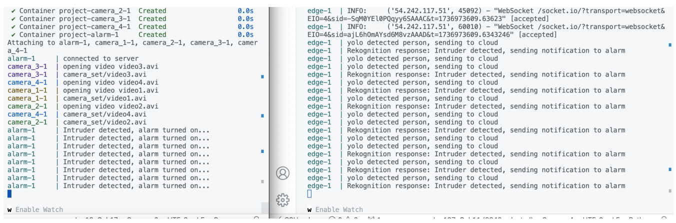
The edge layer operates on a t2.medium instance. This increased computational capacity is necessary due to the resource demands of the YOLO algorithm for real-time person detection.

The cloud layer, hosting the AWS Rekognition integration and REST API, runs on a t2.micro instance, as the facial recognition processing is offloaded to AWS services.

This deployment configuration provides a cost-effective setup while ensuring adequate processing power where needed, particularly for the computationally intensive edge layer operations.

VI. Evaluation

Evaluation of the response time and scalability (number of devices and traffic) to prove the correctness of your implementation. The more detailed the better.



```
 ✔ Container project-camera_2-1  Created                     0.0s
 ✔ Container project-camera_3-1  Created                     0.0s
 ✔ Container project-camera_4-1  Created                     0.0s
 ✔ Container project-alarm-1     Created                     0.0s
Attaching to alarm-1, camera_1-1, camera_2-1, camera_3-1, camer
a_4-1
alarm-1     | connected to server
camera_3-1  | opening video video3.avi
camera_3-1  | camera_set/video3.avi
camera_4-1  | opening video video4.avi
camera_1-1  | opening video video1.avi
camera_1-1  | camera_set/video1.avi
camera_2-1  | opening video video2.avi
camera_4-1  | camera_set/video4.avi
camera_2-1  | camera_set/video2.avi
alarm-1     | Intruder detected, alarm turned on...
alarm-1     | Intruder detected, alarm turned on...
alarm-1     | Intruder detected, alarm turned on...
alarm-1     | Intruder detected, alarm turned on...
alarm-1     | Intruder detected, alarm turned on...
alarm-1     | Intruder detected, alarm turned on...
alarm-1     | Intruder detected, alarm turned on...
alarm-1     | Intruder detected, alarm turned on...

w Enable Watch
```

```
edge-1  | INFO:     ('54.242.117.51', 45092) - "WebSocket /socket.io/?transport=websocket&
EIO=4&sid=-SqM0YEl0PQqyy6SAAAC&t=1736973609.63623" [accepted]
edge-1  | INFO:     ('54.242.117.51', 60010) - "WebSocket /socket.io/?transport=websocket&
EIO=4&sid=ajL6hOmAYsd6M8vzAAAD&t=1736973609.6343246" [accepted]
edge-1  | yolo detected person, sending to cloud
edge-1  | Rekognition response: Intruder detected, sending notification to alarm
edge-1  | yolo detected person, sending to cloud
edge-1  | yolo detected person, sending to cloud
edge-1  | Rekognition response: Intruder detected, sending notification to alarm
edge-1  | yolo detected person, sending to cloud
edge-1  | Rekognition response: Intruder detected, sending notification to alarm
edge-1  | yolo detected person, sending to cloud
edge-1  | Rekognition response: Intruder detected, sending notification to alarm
edge-1  | yolo detected person, sending to cloud
edge-1  | yolo detected person, sending to cloud
edge-1  | Rekognition response: Intruder detected, sending notification to alarm
edge-1  | yolo detected person, sending to cloud
edge-1  | yolo detected person, sending to cloud
edge-1  | Rekognition response: Intruder detected, sending notification to alarm
edge-1  | yolo detected person, sending to cloud
edge-1  | Rekognition response: Intruder detected, sending notification to alarm
edge-1  | yolo detected person, sending to cloud
edge-1  | Rekognition response: Intruder detected, sending notification to alarm

w Enable Watch
```

Fig. 2. Sucessful detection of intruders.