Distributed Systems - Surveillance System Project

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

I. Introduction

Our project simulates a real-world surveillance system that detects intruders in a compound based on video footage. Multiple cameras send video streams to an edge device, which preprocesses the data and queries a cloud-based face detection system. If an intruder is detected, an alarm is triggered in the specific compound.

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

Our system is organized into three distinct layers: IoT, edge, and cloud, each designed to handle specific tasks of the overall process. The IoT layer serves as the entry point for data, consisting of sensors like cameras and alarms that collect real-time information from the environment. The edge layer is responsible for local data processing, performing tasks like initial object detection to reduce the amount of data sent to the cloud. This layer ensures low-latency responses and minimizes network usage. The cloud layer handles more intensive tasks, such as advanced facial recognition and scaling resources based on demand.

The system assumes a network topology where multiple cameras at a single location are connected to one edge device. For the simulation, we chose two edge devices, each managing two cameras but only one alarm. The edge devices operate as separate entities while sharing a common cloud service. The cloud layer is designed to handle multiple edge devices simultaneously, with the capability to trigger location-specific alarms when intruders are detected.

The cameras are simulated using a section of the WiseNET dataset. Individual video files are loaded into camera containers running on a single EC2 instance (see Figure 1). Each container parses the video file into individual images. These image sequences are then sent to the assigned edge devices at regular intervals via Socket.IO, simulating a real-time video stream. Additionally, the EC2 instance hosts an alarm container for each edge/location, completing the Internet of Things (IoT) layer of the system.

The edge layer also consists of an EC2 instance, simulating two distinct edge devices located in different areas. At the edge, the image stream is processed using the YOLO (You Only Look Once) algorithm. When a person-like object is detected,

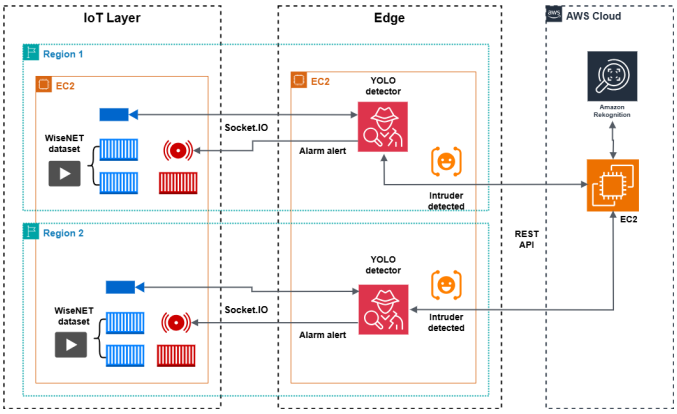

Fig. 1. System architecture, depicting three separate layers in 2 regions

the relevant image snippet is forwarded to the cloud layer via a REST API and waits for a response. If a positive response is received, the edge triggers the alarm in the corresponding location. With sufficient resources in the cloud, the system can scale dynamically to meet 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.

This architecture prioritizes performance and scalability. YOLO is used for fast, real-time object detection at the edge, minimizing latency. Amazon Rekognition handles facial recognition in the cloud, offloading computationally heavy tasks. WebSockets enable real-time, bidirectional communication for efficient image streaming with minimal delay. REST APIs ensure reliable data exchange, while EC2 instances simulate cameras and edge devices.

III. Implementation details

A. Layer implementation

We decided that the IoT layer will consist of two camera containers and one alarm container for each edge device. This setup is an ideal simulation choice, as it minimizes the system's computational requirements while maintaining the separation between individual devices. The number of cameras can be easily scaled to more devices, and the alarm containers can be triggered independently based on the edge device's response. We use the WiseNET dataset (set_3) for video simulation, which features two individuals appearing at different times—one simulating an intruder and the other an employee. The camera containers have access to the full set of

videos, but each container selects one based on a container-specific environment variable, *CAMERA*. The video sets are *.avi* files that are read sequentially and looped when they reach the end.

We process these videos using OpenCV's *read()* function to extract frame sequences. The system then transmits these frames at short intervals (2 seconds) to the edge device via the Socket.IO library. This significantly reduces the edge device's workload while maintaining low system latency. The alarm container maintains a connection to the edge server and remains in a waiting state until triggered by an event.

We implemented the edge layer 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. Each incoming image is analyzed for person-like objects. When such an object is detected, YOLO creates a snippet of the object and immediately forwards it to the cloud server via a REST API, ensuring a response. In the case of a positive detection, an alarm is triggered using Socket.IO for communication back to the IoT layer.

The cloud layer is dedicated to facial recognition and alert management. It uses a REST API built with Python's Flask package to receive images from the edge layer. The system integrates with AWS Rekognition for facial recognition, with the Rekognition collection and known faces automatically provisioned at cloud server startup using boto3. After processing, the result is returned in the HTTP response.

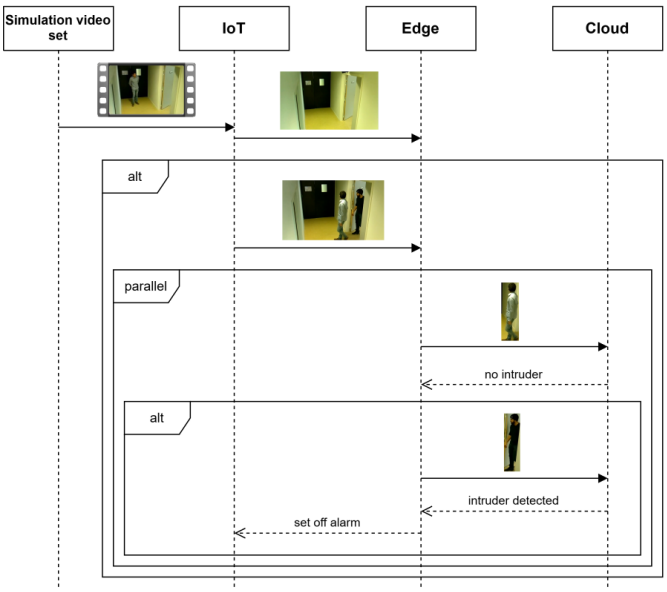

Fig. 2. Sequence diagram of the system

Figure 2 shows the sequence diagram of the system. The camera containers send frames to the edge device. The edge device processes the frames, extracts persons and sends them to the cloud. The cloud processes the frames and sends a response back to the edge. The edge triggers the alarm if an intruder is detected.

B. Implementation Decisions

In our software stack, we had several options to choose from, and we made specific decisions based on our requirements. This section explains the rationale behind our choices, including the programming language, communication libraries, and detection algorithms.

1) Communication: The distributed system comprises several components that must communicate efficiently to function as a cohesive unit. Specifically, there are two primary communication interfaces:

- 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 on top of the network layer. It is a consistent communication model that requires minimal overhead for setting up the connection.

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

Our choice of communication model is based on the assumption that data transfer between the edge and cloud is less frequent than between IoT and edge. Ideally, only a few requests per minute are sent to the cloud, as the edge device filters out most non-relevant frames. This reduces the load on the cloud server and ensures efficient processing of critical data. 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 response for our computation.

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

- YOLO: preprocessing images on edge
- AWS Rekognition: facial recognition on cloud

We chose AWS Rekognition for facial recognition due to its powerful capabilities and ease of use. Having tested this service earlier in the semester, we found it to be a reliable and efficient solution for our needs.

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 wraps *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.

To integrate YOLO and Rekognition into our system, we encapsulated their functionalities within wrapper classes. For instance, we created a wrapper class for YOLO, allowing us to use the *yoloDetection.analyzeImage(image)* function. This function internally runs the YOLO image detection algorithm and returns *True* if a person is detected, otherwise *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: two camera instances + alarm instance
- EDGE - layer: two edge instances 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.

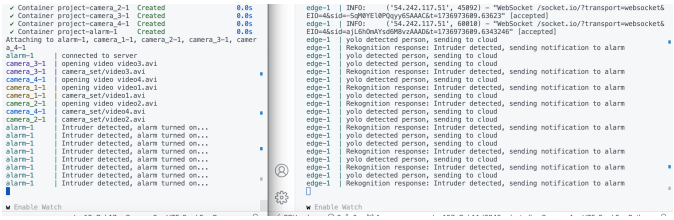

Fig. 3. Sucessful detection of intruders.

VI. Evaluation

For the evaluation of our distributed surveillance system, we conducted extensive performance testing across all three layers (IoT, edge, and cloud) to assess the system's efficiency and reliability. Our evaluation focused on several key metrics: processing latencies at both edge and cloud layers, YOLO detection accuracy ratios, and round-trip time (RTT) distributions under different deployment scenarios. We analyzed these metrics in both local Docker environments and EC2 deployments to understand the system's behavior under various conditions.

As stated in the Deployment section, we used three EC2 instances representing IoT, edge and cloud layer respectively. The edge instance was deployed on a t2.medium instance, while the cloud instance was deployed on a t2.micro instance. The IoT instance was also deployed on a t2.micro instance. The most compute is needed in the edge layer, as the YOLO algorithm is computationally expensive. The cloud layer is the least computationally expensive, as the facial recognition is offloaded to AWS Rekognition.

We decided to use parts of the WiseNET dataset to evaluate our distributed system. The videos are stored and processed on the IoT to send frames to the system. For the experiment we chose a dataset with two individuals appearing at different times in an office scenario. One person simulates an intruder and the other an employee. In the following we focus also on two metrics that are dependent on the dataset. The YOLO detection ratio is calculated as the number of persons detected divided by the total number of frames processed. The intruder

detection ratio is calculated as the number of intruders detected divided by the total number of frames processed. As we had no time to manually check every result, we cannot verify the correctness of the detection.

Initially our deployment evaluations showed low latencies on the cloud. The figure 4 shows the latencies of the cloud processing. The latencies are very low, at a median latency of 78.1ms. Therefore we decided to simulate latencies in the system.

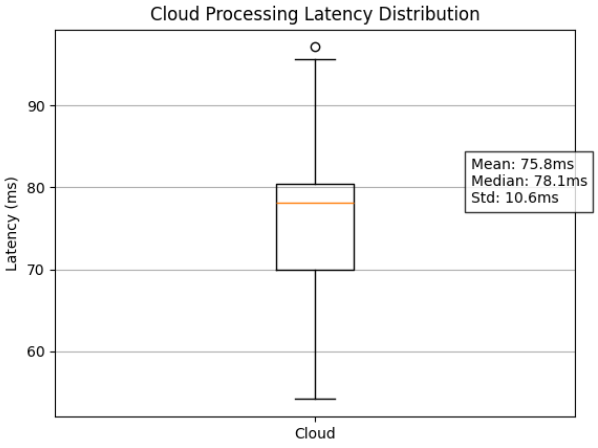

Fig. 4. Boxplot of cloud processing latencies

Here we added gaussian ($x \sim \mathcal{N}(150, 20)$) latencies to edge-cloud connection. In our assumption, edge devices are at the same location as the cameras, so the latencies are very low. The cloud is located in a different region, so the latencies are higher.

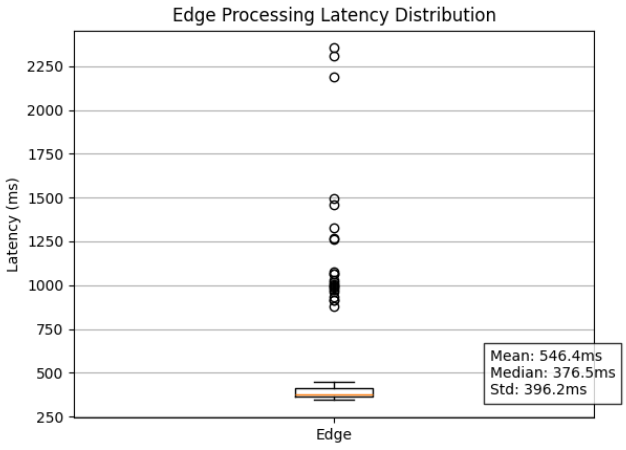

Fig. 5. Edge processing latencies showing YOLO object detection performance on t2.medium

The edge processing performance shown in Figure 5 demonstrates that our YOLO-based object detection system operates with reasonable efficiency on the t2.medium instance. The median processing time of 376.5ms indicates that the edge layer can process multiple frames per second. While there are some outliers reaching up to 2250ms, the majority of processing times fall within an acceptable range (std: 396.2ms). This performance is suitable for real-time surveillance, as our system processes frames at 1-second intervals. Our YOLO detection statistics (Figure 6) show a detection rate of 17.9%, with 20 detections out of 112 total frames processed. This ratio indicates that our edge preprocessing effectively filters out frames without human presence, significantly reducing the load on the cloud layer. For the tested office scenario from the WiseNET dataset, this detection rate aligns with our expectations of occasional human presence rather than continuous detection.

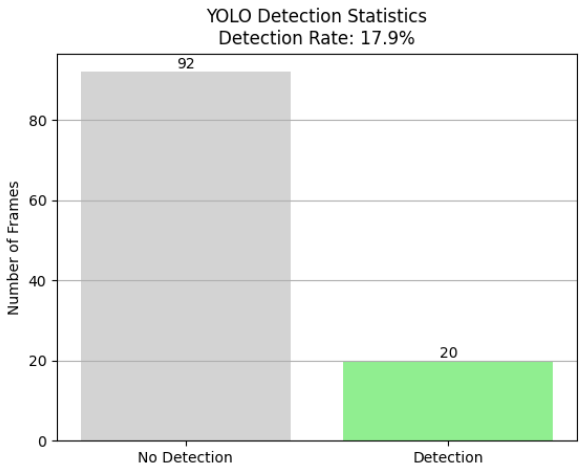

Fig. 6. YOLO detection ratios - number of persons detected / total frames processed

The end-to-end system performance is captured in the RTT distribution (Figure 7). With a median RTT of 467.2ms and a mean of 698.9ms, the system demonstrates responsive performance even with our simulated Gaussian network latency ($x \sim \mathcal{N}(150, 20)$ ms) between edge and cloud. The standard deviation of 556.1ms indicates some variability in processing times, but the overall distribution remains within acceptable bounds for our surveillance use case. However the RTT heavily relies on the amount of frames send to the edge device and the computing power of the edge processing instance. In VI-A we discuss the latency analysis in more detail. Note that the RTT is calculated for both cases with and without intruders, meaning only a portion of the frames are sent to the cloud. Therefore, the RTT is not simply the sum of the median cloud latencies and the YOLO processing time.

In Figure 8 we demonstrate the intruder detection ratios, showing the number of intruders detected relative to the total number of frames processed. The system detected 20 intruders out of 44 frames. Again, this performance cannot be verified without a manually labelled dataset.

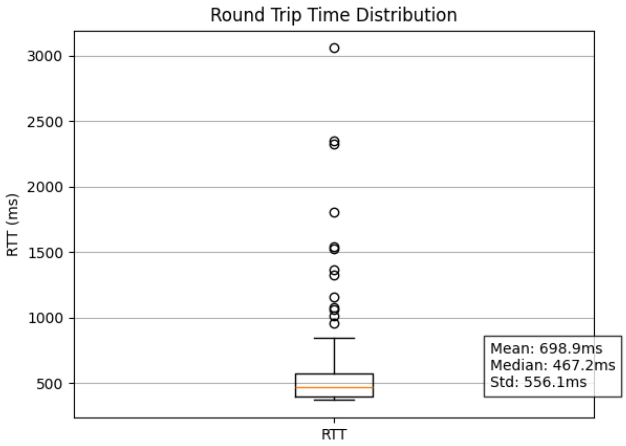

Fig. 7. Round-trip time (RTT) boxplot showing end-to-end system latency

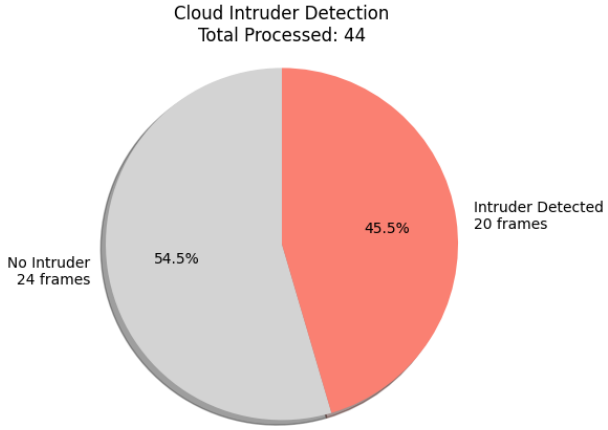

Fig. 8. Intruder detection ratios - number of intruders detected / total frames processed

A. *Latency analysis*

The main bottleneck of the distributed intruder detection system lies in the edge layer. The edge layer's implementation consists of a worker thread and a queue. The worker sequentially processes frames from the queue. Whenever the rate of new incoming frames is higher than the processing rate, the queue will fill up, leading to increased latencies. This is the main reason for the high standard deviation in the RTT distribution. This problem depends mainly on the amount of frames sent per minute per device and the number of IoT devices connected to a single edge device.

There are three variables to adjust in this setup:

- limit the rate of sending frames of the cameras.
- limit the number of connected devices per edge device.
- increase the resources of the edge device and deploy multiple worker threads.

Our performance analysis focused on varying the frame transmission rates to the edge device. The framerate of the WiseNET dataset is 30 fps. We implemented a mechanism to send only every nth frame.

We conducted the experiment for two IoTs per edge device with different frame rates. The results are shown in Figure 9 and Figure 10.

Figure 9 shows the RTT boxplot for the system with 20 frames skipped. This means that $\frac{2}{3}$ frames per second. This resulted in significantly increased latencies with a median of 13400ms. The system was not able to process the frames in time and the round-trip times increased exponentially.

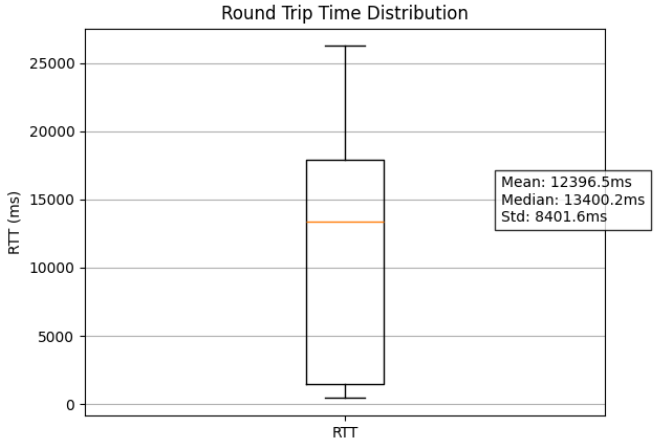

Fig. 9. Round-trip time (RTT) boxplot showing end-to-end system latency (on Docker with 20 frames skipped)

Figure 10 shows the RTT boxplot for the system with 60 frames skipped, meaning only 0.5 frames per second are sent to the edge device. In this configuration, the system processed the frames in a timely manner, resulting in significantly lower round-trip times with a median of 350ms.

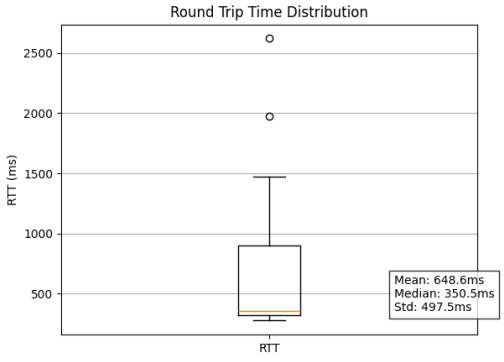

Fig. 10. Round-trip time (RTT) boxplot showing end-to-end system latency (on Docker with 60 frames skipped)

This analysis shows the importance of the right parameters for the system.