Geo-Distributed Application for Early Forest Fire Detection

1st Noah Polig 2nd Damian Klotz 3rd Jonas Mayr

I. Introduction

Our project will simulate a distributed system for early forest-fire detection.

II. System Architecture

We orchestrate the three layers, IoT, Edge and Cloud like so:

IoT Layer:
We simulate several *camera* devices on laptops. Each camera produces a frame stream of a certain forest area.
We also simulate *radios* as a second type of an IoT-device. These devices pose as an alarm for a detected fire and as a warning system for nearby areas.
Each radio is assigned to a respective camera and the pair of them are inside a *forest area*.

Edge Layer (ECS):
We run the Edge notes on AWS ECS and call them *weather stations*.
They take the video streams from the cameras in their region and do a light, fast analysis using simple OpenCV checks. Furthermore, they only send important information to the Cloud — for example, only frames that look suspicious (possible fire). Hence we can reduce bandwidth and keep the system efficient. Additionally, *weather stations* are tasked to track the wind direction and the wind speed of the region they are in.

Cloud Layer (EC2 + Akka):
In the Cloud, we run Akka-based services on EC2-instances. We call these services *Orchestrators*.
They receive frames from the weather stations and do a deeper analysis to confirm if there is really a fire. If that's the case, they trigger an alarm in the corresponding area and check if any other area is in danger based on the wind direction/speed. As previously stated, this data is also sent by the weather stations. For this to work, the areas are defined in the Cloud with corresponding coordinate-values, which describe their geo-distribution. The calculation itself is done in an AWS Lambda function. If any other area is in danger, the Orchestrator also sends a warning to them. This warning comes with a severity-value which tells the alarm how urgent the situation is (computed from wind speed and distance to fire-source).
Moreover, all messages are logged within Redis.

Communication:
We use NATS as the message system between all layers. This includes the frame stream (IoT → Edge → Cloud), the wind values (Edge → Cloud) and the alarm/warning messages (Cloud → IoT).

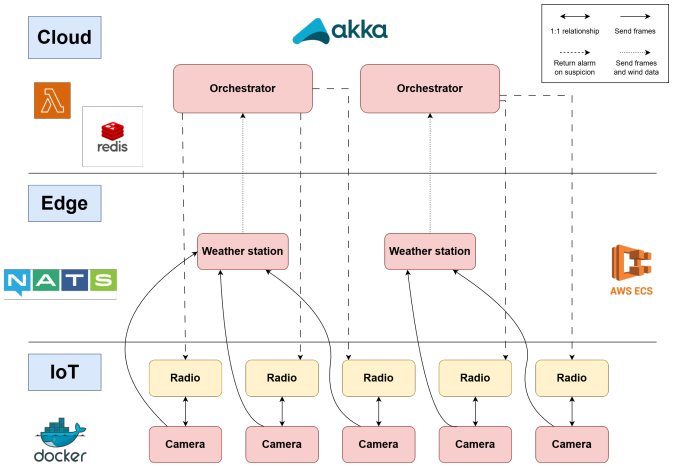

Fig. 1. Architectural diagram.

III. Implementation Details

IoT Layer:
Camera Simulator:
The camera simulator reads image files (frames) from a local folder and sends them continuously to the Edge layer using NATS messaging. This creates a simple stream that behaves similar to a real camera feed. The frames are published to a subject with the format `area.<area>.frame`. The FPS value can be configured, so the camera can send more or fewer frames per second, which helps us test system behaviour under different loads. The program is written in Python using `asyncio`, so it can stream frames without blocking execution.

Alarm Radio:
The alarm radio listens for alert messages coming from the Cloud instead of sending frames. Each radio subscribes to a subject based on its area (`alerts.<area>`). When the Cloud detects a fire, it publishes a message to this subject and the radio prints a warning message on the screen. This allows us to see alarms in real time and check if the communication works as expected. The program also uses Python and runs

continuously in a loop waiting for messages.

Edge Layer:
Each weather station has a list of associated areas. They then subscribe to their respective areas using the Subject `area.<area>.frame`. Upon reception of a frame they do **Preprocessing** and **Packaging**.

Preprocessing (Smoke and Fire Detection):
On the Edge we run a lightweight image processing step before sending frames to the Cloud. The goal is to filter out harmless frames early. We do not want to upload every frame, because this would create a lot of traffic. Instead, we check the frame first and only continue with frames that might show smoke or fire. This makes the whole pipeline faster and reduces network usage.

The preprocessing is done in two steps. In the first step we try to detect smoke in the frame. Smoke is important for early detection, because it often appears before visible flames. For the detection we first convert the frame from BGR to HSV, because HSV makes it easier to work with colors. Then we create a grey color mask with a lower and upper HSV bound. Grey pixels often have low saturation and medium brightness, which we can detect with very simple rules. We count how many pixels fall into this grey range and divide it by the total pixel count. If this smoke value is higher than our threshold value, we mark the frame as suspicious. In this case we directly send the frame to the Cloud and skip the fire check, because smoke alone is already a warning.

If no smoke was found, we do the second step where we check for fire. This step works similar to the smoke detection, but we use different color bounds. This time we look for red and orange tones, which are typical fire colors. We create a mask for these colors, count the pixels inside it and calculate a fire ratio. If this ratio is higher than our fire threshold, the frame is marked as suspicious and sent to the Cloud. If the value is lower, we ignore the frame and do not send it further.

This preprocessing step is simple and lightweight. It can run in real time without high CPU load. The Edge only forwards frames that contain smoke or fire indicators, so the Cloud does not get flooded with useless frames. This reduces bandwidth, makes the system more scalable and improves performance. Even though this solution is basic, it already helps to detect possible danger early and makes the whole system more efficient.

Packaging:
The weather stations also send two metrics regarding the wind in the region: Speed and Direction. These are simulated random values computed in the edge. Additionally, the edge also sends a timestamp-value for potential logging purposes and other necessary metadata. All this is then sent as Subject `region.region.processed` to the cloud.

Summary:
Both IoT components are simple, lightweight and easy to run. We containerized them using Docker, which allows us to start multiple cameras or radios in parallel for scalability tests. The same goes for the Edge component.

IV. Evaluation

Stress your application to prove the correctness of your implementation, be aware of its main limitations. Explain first the experiments done (e.g., vary the number of input events), then introduce and discuss the results obtained.

V. Conclusions and future work

Summarize your solution described in this report, as well as honestly mention the current limitations and the areas that could be explored in future work.