Ti9tek.AI documentation

- An AI & Iot and Distribued system based solution

for Real-Time Data Processing and Automation

.

Elabored by AI & Data science and Iot engeering students

Contents

Introduction

In today's interconnected world, the integration of Artificial Intelligence (AI) and the Internet of Things (IoT) has revolutionized the way we approach automation, efficiency, and data-driven decision-making. This project explores the development of a distributed AI-powered IoT system designed to process video streams from multiple sources in real time. The system demonstrates a scalable and modular architecture that combines advanced AI-based video analysis with distributed task management.

The core objective of this project is to create an intelligent system capable of detecting specific behaviors or objects in video feeds using state-of-the-art AI models. This is achieved by distributing the computational workload across multiple worker nodes, ensuring efficiency, fault tolerance, and scalability. Additionally, the system integrates a real-time notification mechanism using WhatsApp, allowing stakeholders to be alerted immediately when the system detects predefined events of interest.

This report delves into the design, implementation, and testing phases of the project, highlighting key components such as video streaming, distributed task allocation, AI-based video analysis using object detection models like DETR (DEtection TRansformer), and the notification system. The architecture leverages Python's socket programming for video transmission, XML-RPC for task management, and PyWhatKit for real-time WhatsApp notifications.

By combining IoT and AI in a distributed framework, this project provides a robust solution for real-time video surveillance and monitoring, making it ideal for applications in security, traffic management, and industrial automation. The following sections detail the system's architecture, code implementation, and the testing process, providing a comprehensive view of the project's functionality and potential real-world applications.

Chapter 1

Design System

Welcome to the beginning of an exciting journey. Let us explore the fundamental building blocks of a sophisticated system—two key components that, when seamlessly integrated, form the foundation of a powerful solution: the **environment** and the **server**.

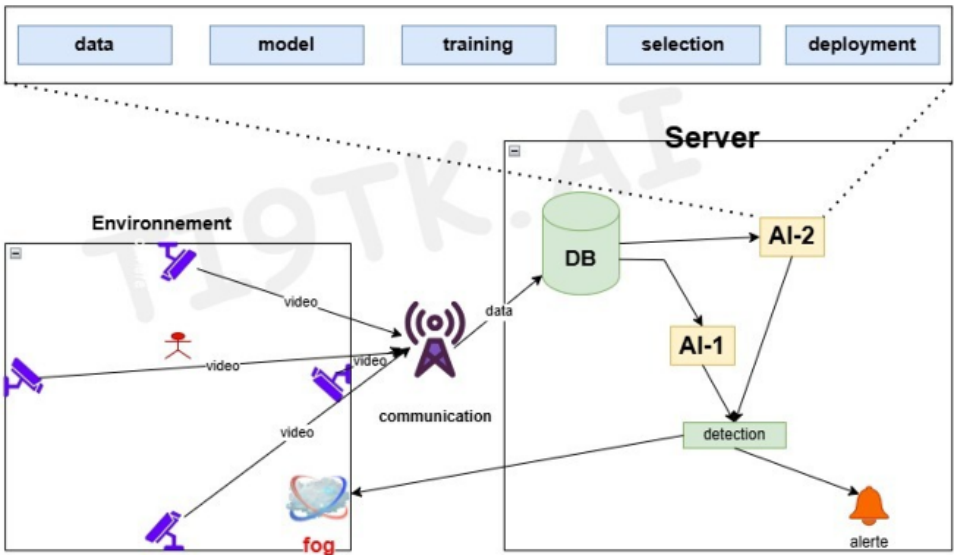

Figure 1.1: First conception of the design system

At first glance, this may seem like a straightforward concept. However, there is much more beneath the surface. Together, the environment and server constitute the core architecture that drives an efficient and seamless user experience. The real value emerges when these components are aligned, each fulfilling its distinct role to ensure the smooth operation of the entire system.

You may be wondering, ”What makes the environment and server so crucial? Why are they so integral to the system?” The answer lies in their complementary nature. Rather than simply existing side by side, these two components enhance one another, creating a harmonious and highly efficient system that functions with precision and reliability.

Chapter 2

The Environnement

Let's start with the **environment**. Think of it as the stage where everything unfolds. It's the setting that holds all the action, the context where processes run and data flows. The environment is where your system's components come to life, interact with one another, and execute the tasks they were designed for.

2.0.1 Smart cameras

The environnement will be constutued essentially with 4 cameras placed in all stages of the specific buildings.

Figure 2.1: Smart cameras

Since it called smart it may require that it's connected to wifi able to send in real time the image recognition.

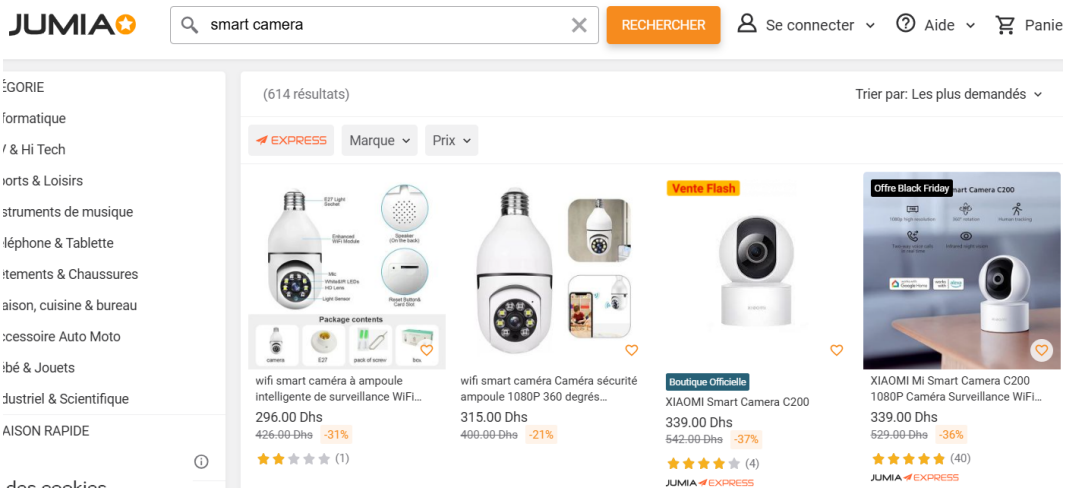

Figure 2.2: Smart Cameras

Overall, the price ranges from 300 DH to over 2000 DH on the Jumia website.

Since we cannot afford to purchase all the smart cameras, we have developed an alternative solution: instead of buying new cameras, we decided to utilize the cameras already available on our PCs. By adapting them, we aim to replicate the functionality of smart cameras—potentially even enhancing their capabilities.

2.1 Protocol of communication

In this setup, the cameras on our PCs will serve as publishers, transmitting data in real-time. These cameras will capture the required images and send them over the network. On the other side, one of the PCs will act as the subscriber, receiving and processing the information sent by the cameras. This setup mimics the behavior of smart cameras while leveraging the existing hardware, making the solution more cost-effective.

For communication between the cameras (publishers) and the receiving PC (subscriber), we need to choose an appropriate protocol to facilitate smooth data transmission. The choice of communication protocol is crucial because it will define how efficiently and reliably the data is transferred across the network. The protocol must ensure that the data is delivered in real-time, with minimal delay, and it must also handle any potential data loss or errors that may occur during transmission.

We explored several communication protocols, including TCP and MQTT, to evaluate their suitability for our system.

1. **TCP (Transmission Control Protocol)** is a reliable, connection-oriented protocol, ensuring that all data sent from the publisher is received by the subscriber without errors. However, it comes with higher overhead due to its error-checking and connection-handling features, which can result in latency issues, especially when transmitting large amounts of image or video data in real-time.

2. **MQTT (Message Queuing Telemetry Transport)** is a lightweight, publish-subscribe messaging protocol that operates over TCP/IP. It is well-suited for systems requiring low-bandwidth,

low-latency communication, such as IoT applications. While MQTT provides efficient communication, its reliance on a broker and the need for an active connection may not always be ideal for real-time image transmission.

After testing these protocols, we decided to use **UDP (User Datagram Protocol)** for this system. UDP is a connectionless protocol that is much faster and more efficient than TCP because it eliminates the need for connection management and error-checking overhead. This makes UDP well-suited for applications where real-time performance is a priority, even at the cost of occasional data loss. In our case, transmitting image data, where real-time speed is more critical than absolute reliability, UDP provides an ideal solution.

Additionally, UDP allows us to transmit multiple packets of data simultaneously without waiting for acknowledgments, reducing latency.

We will also implement techniques such as packet sequencing and error handling at the application level to mitigate the effects of potential data loss, ensuring the system remains functional even if some packets are lost during transmission.

2.2 Publishers Setup

To highlight the functionalities of the publisher, we have defined a UML diagram to explain it:

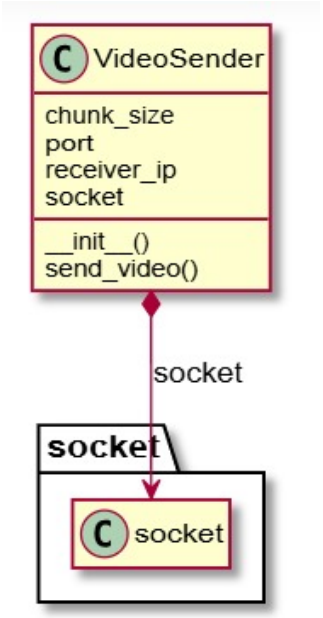

Figure 2.3: UML diagram for the environment setup

The diagram depicts a class named `VideoSender` and its relationship with another class named `socket`.

1. **Class VideoSender :**
    - **Attributes:**

(a) `chunk_size`: Represents the size of data chunks that will be sent in the video transmission.

(b) `port`: The port number on which the video data will be sent.

(c) `receiver_ip`: The IP address of the recipient of the video data.

(d) `socket`: Holds a reference to a `socket` object, which is used for network communication.

- **Methods:**

(a) `__init__()`: The constructor method, responsible for initializing the `VideoSender` object and setting up necessary parameters such as `chunk_size`, `port`, and `receiver_ip`.

(b) `send_video()`: Sends the video data to the specified `receiver_ip` and `port`, using the `socket` object to establish a connection and transmit the video data in chunks.

2. **Relationship with the socket Class :**

The diagram shows a composition relationship between the `VideoSender` class and the `socket` class. This means:

(a) **Ownership**: The `VideoSender` class owns the `socket` object.

(b) **Lifetime**: The `socket` object's lifetime is tied to the `VideoSender` object. When the `VideoSender` object is destroyed, the `socket` object is also destroyed.

3. **Interpretation :**

This class diagram represents a simplified model of a video streaming application. The VideoSender class encapsulates the logic for sending video data over a network. It uses a socket object to establish a connection with the receiver and transmit the video data in chunks.

Below is an example of a Python script for sending video over UDP of a publisher :

Listing 2.1: PublisherUDP.py - Video Sender Script

```
1
2  import cv2
3  import socket
4  import numpy as np
5
6  class VideoSender:
7      def __init__(self, receiver_ip= 'YOUR IP ADRESS' , port='500x',
           chunk_size=8192):
8          self.receiver_ip = receiver_ip
9          self.port = port
10         self.chunk_size = chunk_size
11         self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
12
13     def send_video(self, video_source=0):
14         try:
15             cap = cv2.VideoCapture(video_source)
16             if not cap.isOpened():
17                 raise IOError("Cannot open video source")
18
19             while True:
20                 ret, frame = cap.read()
21                 if not ret:
```

```
                    print("Failed to grab frame")
                    break

                # Compress the image with reduced quality
                encode_param = [int(cv2.IMWRITE_JPEG_QUALITY), 30]
                result, encoded_frame = cv2.imencode('.jpg', frame,
                    encode_param)
                data = encoded_frame.tobytes()

                # Send the image in chunks via UDP
                for i in range(0, len(data), self.chunk_size):
                    chunk = data[i:i+self.chunk_size]
                    self.socket.sendto(chunk, (self.receiver_ip, self.
                        port))

                # End of frame marker
                self.socket.sendto(b'END', (self.receiver_ip, self.port))

                # Optionally show the video locally
                cv2.imshow(f'Sending to {self.receiver_ip}:{self.port}...
                    ', frame)

                if cv2.waitKey(1) & 0xFF == ord('q'):
                    print("Exiting video stream...")
                    break

        except Exception as e:
            print(f"Video streaming error: {e}")

        finally:
            cap.release()
            cv2.destroyAllWindows()
            self.socket.close()

# Example usage
if __name__ == "__main__":
    sender = VideoSender()
    sender.send_video()
```

Explanation of the `PublisherUDP.py` Code

The `PublisherUDP.py` script is designed to capture video from a camera, compress the video frames, and transmit them over a network using the **UDP protocol**. This enables real-time video streaming between two computers. The video data is sent in smaller *chunks* to ensure efficient transmission.

Key Concepts and Components

1. **IP Address and Port:**
    - IP Address (`self.receiver_ip`): An *IP address* is a unique identifier for a device on a network, similar to a phone number in a communication system. In the code, the

`receiver_ip` is set to `'YOUR IP ADRESS '`, which should be the IP address of the receiving computer. The sender and receiver must be on the same network for successful communication.

- Port (`self.port`) : A *port* is a communication endpoint on the receiving device. Think of it as a specific "door" through which data can be sent or received. The port is set to `500x`, where `x` depends on the number of publishers. Both the sending and receiving computers must use the same port for communication. Note that for each PC used, the port should be unique to avoid conflicts and ensure proper communication.

2. **Socket (Communication Channel)**

   It's is an interface that allows communication between two devices over a network.

   In this script, a **UDP socket** is used (`socket.SOCK_DGRAM`). UDP (User Datagram Protocol) is a faster, connectionless protocol ideal for real-time applications like video streaming. Unlike TCP (Transmission Control Protocol), UDP does not guarantee the delivery of every packet, but it has lower latency, making it suitable for applications where speed is more critical than guaranteed delivery.

3. **Video Capture and Compression :**

   - **Video Capture**: The script uses `cv2.VideoCapture(video_source)` to capture video from the camera. By default, `video_source=0` uses the first connected camera (usually the webcam).
   - **Frame Compression**: Each video frame is compressed using JPEG encoding with a reduced quality setting of 30

4. **Sending Data in Chunks:**

   The video frames are converted to byte data using `encoded_frame.tobytes()`. The data is then split into smaller *chunks* of size `self.chunk_size` (set to `8192` bytes or 8 KB). These chunks are sent over the network using the `socket.sendto()` method, ensuring efficient and reliable transmission.

Chapter 3

The server

However, the environment, while crucial, doesn't work alone. It needs direction. That's where the **server** comes in. The server is the conductor, the one calling the shots and ensuring that everything works as it should.

3.1 First Implementation :

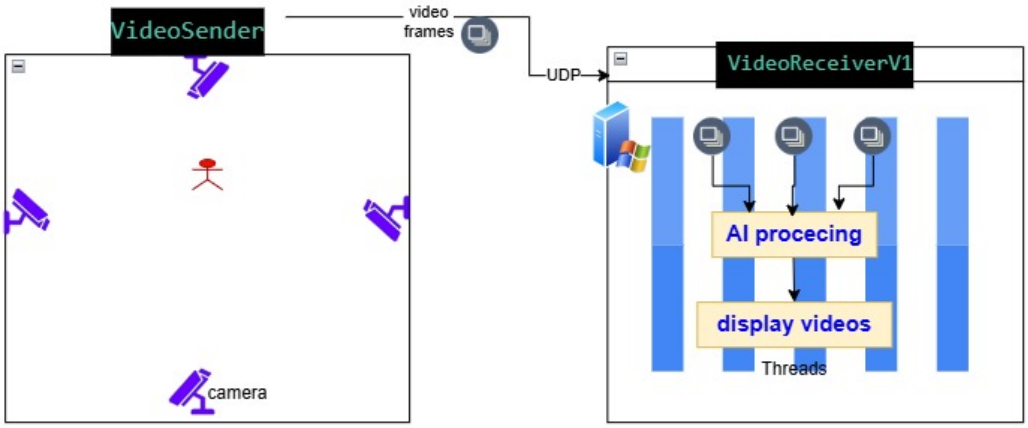

Figure 3.1: Conception of the first implementation

Our first approach, as mentioned earlier, was to adapt one PC to act as a subscriber, receiving data from the senders (publishers). The diagram illustrates that the VideoReceiver receives data in the form of frames via the UDP protocol, as previously discussed. After that, each predefined port will receive its respective frames, perform AI processing, and display the videos.

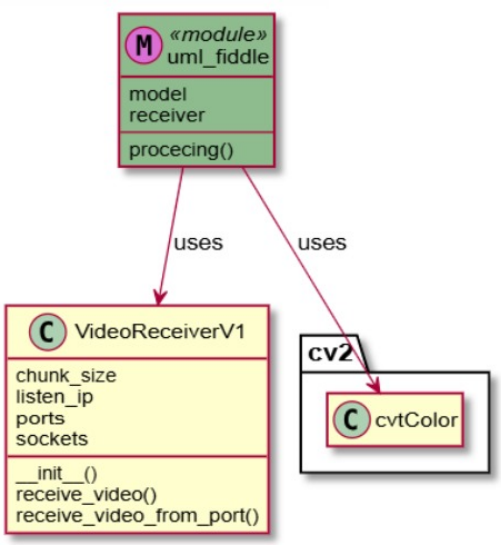

Figure 3.2: UML diagram to explain the VideoReceiver architecture

The diagram depicts a class named `VideoReceiverV1` and its relationship with a module named `uml_fiddle` and a class named `cv2.cvtColor`.

1. **Attributes:**
   - `chunk_size`: This represents the size of data chunks that will be received in the video transmission.
   - `listen_ip`: The IP address on which the video data will be received.
   - `ports`: This holds a list of port numbers on which the video data will be received.
   - `sockets`: This attribute holds a list of socket objects, which are used for network communication.
2. **Methods:**
   - `__init__()`: This is the constructor method, responsible for initializing the `VideoReceiverV1` object. It sets up the necessary parameters like `chunk_size`, `listen_ip`, and `ports`.
   - `receive_video()`: This method is responsible for receiving the video data from the specified `listen_ip` and `ports`. It uses the sockets to establish connections and receive the video data in chunks.
   - `receive_video_from_port()`: This method is a helper function used by `receive_video()` to receive video data from a specific port.
3. **Relationship with `uml_fiddle` Module:**

   The diagram shows a `uses` relationship between `VideoReceiverV1` and `uml_fiddle`. This means that:
   - **Dependency:** The `VideoReceiverV1` class depends on the `uml_fiddle` module.
   - **Usage:** The `VideoReceiverV1` class uses functions or classes provided by the `uml_fiddle` module for some of its operations.

4. **Relationship with `cv2.cvtColor` Class:**

   The diagram shows a `uses` relationship between `VideoReceiverV1` and `cv2.cvtColor`. This means that:

   - **Dependency:** The `VideoReceiverV1` class depends on the `cv2.cvtColor` class.
   - **Usage:** The `VideoReceiverV1` class uses the `cv2.cvtColor` class to convert the received video data from one color space to another (e.g., from BGR to RGB).

5. **Interpretation:**

   This class diagram represents a simplified overview of how `VideoReceiverV1` interacts with external modules and performs its operations, with clear dependencies on `uml_fiddle` and `cv2.cvtColor`.

Below is an example of a Python script for receiving video over UDP of a publisher :

Listing 3.1: SubscriberUDP.py - Video Receiver Script

```
import cv2
import socket
import numpy as np
import threading

class VideoReceiver: def __init__(self, listen_ip='0.0.0.0', ports=[5001,
    5002,5003], chunk_size=8192):
        self.listen_ip = listen_ip
        self.ports = ports self.chunk_size = chunk_size
        self.sockets = [] # List to hold the sockets.

         sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            sock.bind((self.listen_ip, port))
            sock.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, 65536)
                # Increase buffer size
            self.sockets.append(sock)
            print(f"Socket bound to port port}")

   def receive_video_from_port(self, sock, port):
        data = b''
        try:
            while True:
                while True:
                    chunk, addr = sock.recvfrom(self.chunk_size)
                    if chunk == b'END':  # End of frame signal
                        break
                    data += chunk

                if data:
                    print(f"Received data on port {port}: {len(data)}
                        bytes")
                    np_data = np.frombuffer(data, dtype=np.uint8)

                    if np_data.size > 0:
                        frame = cv2.imdecode(np_data, cv2.IMREAD_COLOR)

```

```
                        if frame is not None:
                            cv2.imshow(f"Video from Port {port}", frame)
                                # Display video in a window specific to
                                the port
                            if cv2.waitKey(1) & 0xFF == ord('q'):
                                break
                        else:
                            print(f"Error: Could not decode frame on port
                                {port}")
                    else:
                        print(f"Warning: Empty data received on port port
                            }")

                # Reset data for the next frame data = b''
except exception as e: print (f"Video receiving error on port {port}: {e}
    ")
        finally:
            print(f"Closing socket on port {port}")
            sock.close()

    def receive_video(self):
        # Create a thread for each socket to handle multiple ports
            concurrently
Threads = [] for idx, sock in enumerate(self.sockets):
            thread = threading.Thread(target=self.receive_video_from_port
                , args=(sock, self.ports[idx]))
            threads.append(thread)
            thread.start()

        # Wait for all threads to finish for thread in threads:
            thread.join()

        # Cleanup
        cv2.destroyAllWindows()

# Example usage
if __name__ == "__main__":
    receiver = VideoReceiver(listen_ip='0.0.0.0', ports=[5001,
        5002,5003])  # Listen on both ports
    receiver.receive_video()
```

The following code defines a class `VideoReceiver`, which is used to receive video data from multiple sources over UDP, process it, and display it using OpenCV. Here's a breakdown of the code:

1. **Imports:**
    - `cv2`: This is the OpenCV library, which is used for handling video frames and displaying them.
    - `socket`: The socket library is used for network communication. It allows the program to receive UDP packets containing video data.
    - `numpy`: This package is used to process large arrays of data. In this case, it's used to convert the received raw video data into image frames.

- `threading`: Although imported, it is not used in the provided code. However, it could be useful for handling concurrent tasks, such as receiving data from multiple ports.

2. **Class: VideoReceiver**
   - The `VideoReceiver` class is designed to handle receiving and displaying video streams.
3. **Constructor (`__init__` method):**
   - `listen_ip`: Specifies the IP address on which the server listens for incoming video data. The default is `0.0.0.0`, meaning it listens on all available interfaces.
   - `ports`: A list of ports (`5001`, `5002`, `5003`) that the server listens on for video data. This means we use 3 PCs as publishers and 1 PC as a subscriber. Of course, we can add as many publishers as we want; we just need to add the corresponding port for each one.
   - `chunk_size`: The size of the data chunks (8192 bytes) that will be received at a time.
   - `sockets`: A list to hold socket objects. Each socket listens on one of the specified ports.
4. **Creating a Socket:**
   - A socket is created using the `socket.socket` method with the `AF_INET` address family (IPv4) and the `SOCK_DGRAM` type (for UDP communication).
   - The socket is bound to the specified `listen_ip` and each port in the `ports` list.
   - The buffer size is set to 65536 bytes for efficient handling of incoming data.
5. **Method: `receive_video_from_port`:**
   - This method is responsible for receiving video data from a specific port.
   - Data is received in chunks and stored in the `data` variable.
   - When the `END` signal is received, the method stops receiving data and proceeds to decode the video frames.
   - The `np.frombuffer` function is used to convert the received byte data into a numpy array.
   - The numpy array is decoded into a video frame using `cv2.imdecode`.
   - If the frame is successfully decoded, it is displayed using `cv2.imshow`.
   - The display will continue until the user presses 'q' to quit.

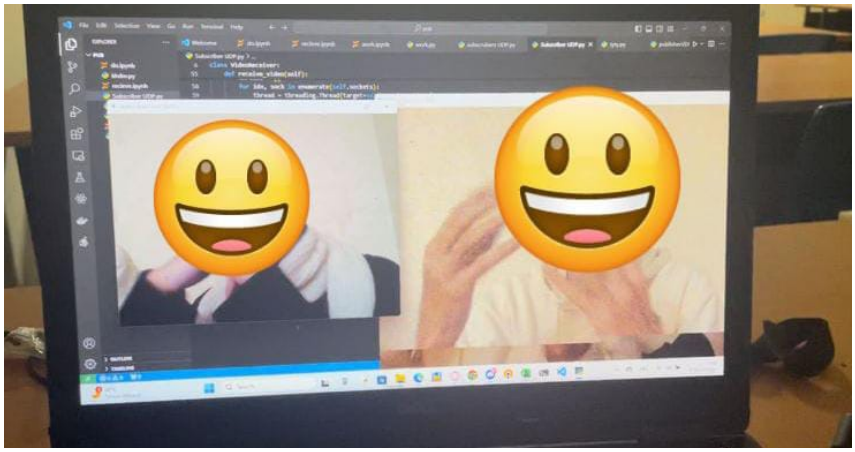

Figure 3.3: Subscriber recieved a video from 2 publishers

3.2 Second Implementation :

However, we aim to make the process more efficient and avoid any unnecessary delays. To achieve this, we restructured the server system.

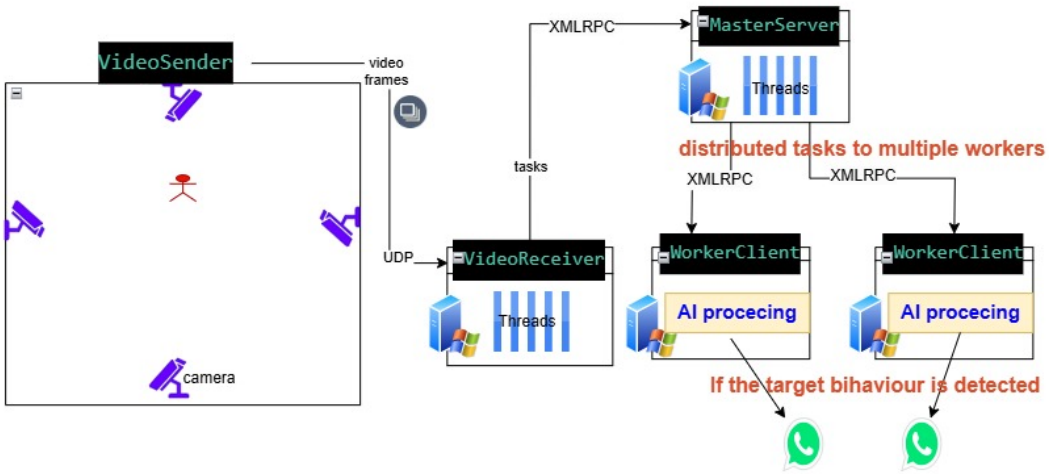

Figure 3.4: Conception of the second Implementation

- **VideoSender**: This component captures video frames from multiple cameras and transmits them to the `VideoReceiver` using UDP, a reliable and efficient protocol for real-time data transmission.
- **VideoReceiver**: The `VideoReceiver` acts as a central hub, receiving video frames from the `VideoSender`. It then distributes these frames to multiple worker clients using XML-RPC, a protocol suitable for remote procedure calls.

- **Worker Clients**: These clients process the received video frames using AI algorithms to detect specific target behaviors. If a worker client identifies the target behavior, it triggers a notification, such as a phone call.
- **Master Server**: Overseeing the entire system, the `Master Server` coordinates the communication and task distribution between the `VideoReceiver` and the worker clients. It ensures efficient resource utilization and system stability.

By distributing the video processing workload across multiple worker clients, the system achieves high performance and scalability. The use of UDP and XML-RPC enables reliable and efficient communication between components. The integration of AI algorithms empowers the system to detect complex patterns and behaviors within the video streams.

3.2.1 VideoReveiver architecture

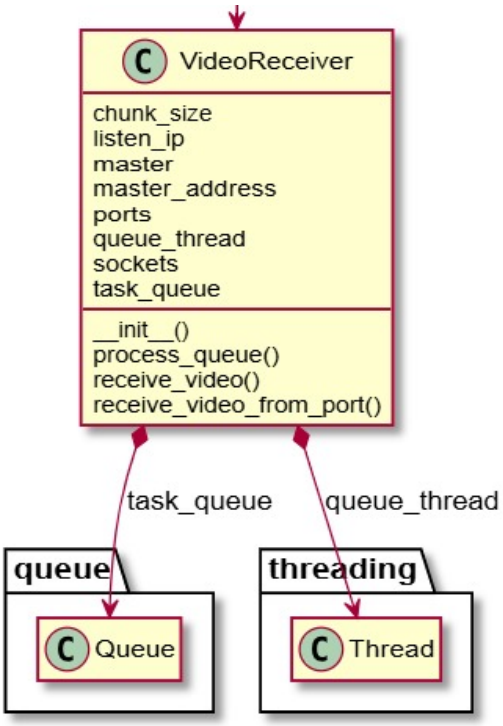

Figure 3.5: VideoReceiver UML diagram

The `VideoReceiver` class is responsible for receiving video data over a network. It has several attributes:

1. **Attributes:**
    - `chunk_size`: defines the size of data chunks to be received.
    - `listen_ip`: The IP address on which the receiver listens for incoming connections.
    - `master_ip`: The IP address of the master server (possibly for coordination or control).

- `master_address:` The address of the master server (a combination of IP and port).
- `ports:` A list of ports on which the receiver listens.
- `sockets:` A list of socket objects used for network communication.
- `queue_thread:` A reference to a thread responsible for managing a queue.
- `task_queue:` A reference to a queue used to store tasks or video frames to be processed.

2. **Methods:**

   - `init():` The constructor, initializes the attributes and sets up the receiver.
   - `process_queue():` Processes tasks or video frames from the task queue.
   - `receive_video():` Receives video data from the network.
   - `receive_video_from_port():` Receives video data from a specific port.

3. **Relationships:**

   - `task_queue:` A composition relationship with the `Queue` class. This indicates that the `VideoReceiver` owns the `Queue` object and is responsible for its creation and destruction.
   - `queue_thread:` A composition relationship with the `Thread` class. This indicates that the `VideoReceiver` owns the `Thread` object and is responsible for its creation and destruction.

4. **Interpretation:**

   - This class diagram suggests a video receiver that uses a multi-threaded approach to handle incoming video data. The `queue_thread` manages a queue of video frames to be processed. The `VideoReceiver` can receive video data from multiple sources and adds the received data to the queue. The `queue_thread` then processes the frames from the queue, possibly distributing the processing tasks to other threads or processes.

3.2.2 MasterServer architecture

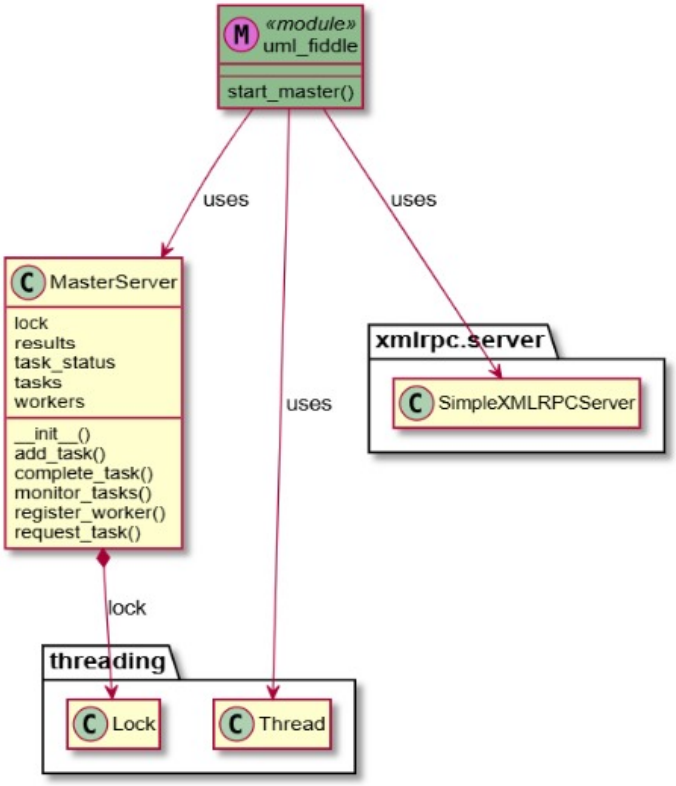

Figure 3.6: MasterServer UML diagram

The master server is responsible for managing tasks, assigning them to workers, and monitoring their progress.

1. **Initializes:**
    - Data structures to manage tasks, their statuses, worker addresses, and results.
    - A lock for thread safety to ensure concurrent access to shared resources.
2. **Task Management:**
    - `add_task:` Adds a new task to the task queue, assigns a unique ID, and marks its status as ”PENDING”.
    - `add_tasks:` Adds multiple tasks to the queue.
    - `request_task:` Retrieves a pending task from the queue and assigns it to a worker, marking its status as ”IN_PROGRESS”. If no pending tasks are available, it returns `None`.
    - `complete_task:` Marks a task as completed, stores the result, and prints a completion message.

- `monitor_tasks:` Periodically checks for tasks that have been in progress for too long and reassigns them as pending.

3. **Worker Management:**
   - `register_worker:` Registers the address of a worker client with the master server.
4. **Result Retrieval:**
   - `get_results:` Retrieves a list of completed tasks and their results for a specific worker port.
5. **Server Setup:**
   - `start_master:`
     - Creates an XML-RPC server on port 8000.
     - Initializes a `MasterServer` object.
     - Registers the `MasterServer`'s methods with the XML-RPC server, making them accessible to remote calls.
     - Starts a monitoring thread to periodically check the task queue.
     - Starts the server to listen for incoming RPC requests.
6. **Overall Functionality:**
   - The `MasterServer` acts as a central coordinator for a distributed system. It receives tasks, distributes them to worker clients, monitors their progress, and collects results.
   - The XML-RPC interface allows worker clients to communicate with the master server and request tasks.
   - The `monitor_tasks` function ensures that tasks are not stuck in the "IN_PROGRESS" state for too long, preventing potential deadlocks or resource leaks.
   - In essence, this code provides a framework for managing distributed tasks, ensuring efficient resource utilization and timely task completion.

3.2.3 WorkerClient architecture

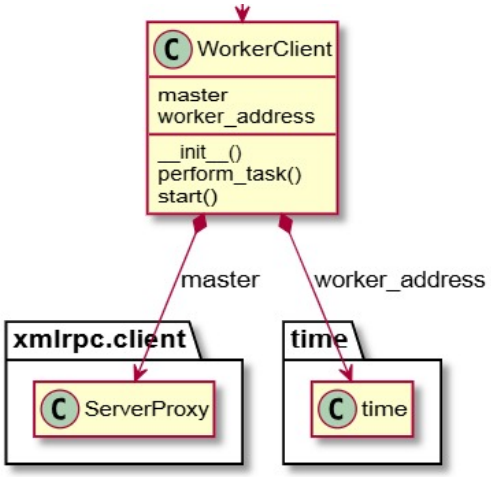

Figure 3.7: Workers UML diagram

The `WorkerClient` class is responsible for performing tasks assigned by a master server. It has several attributes:

1. **Attributes:**
   - `master:` A reference to the master server.
   - `worker_address:` The address of the worker client itself.
2. **Methods:**
   - `init():` The constructor, initializes the attributes and sets up the worker client.
   - `_perform_task():` Performs a task assigned by the master server.
   - `start():` Starts the worker client, possibly by creating a thread to handle tasks.
3. **Relationships:**
   - `master:` A usage relationship with the `xmlrpc.client.ServerProxy` class. This indicates that the `WorkerClient` uses the `ServerProxy` object to communicate with the master server via XML-RPC.
   - `worker_address:` A usage relationship with the `time` class. This suggests that the `WorkerClient` might use the `time` class to measure execution time or for other time-related operations.
4. **Interpretation:** This class diagram is a distributed system where multiple worker clients are managed by a master server. The `WorkerClient` class represents an individual worker that can receive tasks from the master server via XML-RPC. The `ServerProxy` object is used to communicate with the master server and send/receive data. The `time` class might be used for various purposes, such as measuring task execution time or synchronizing with the master server.

3.3 Final Implementation :

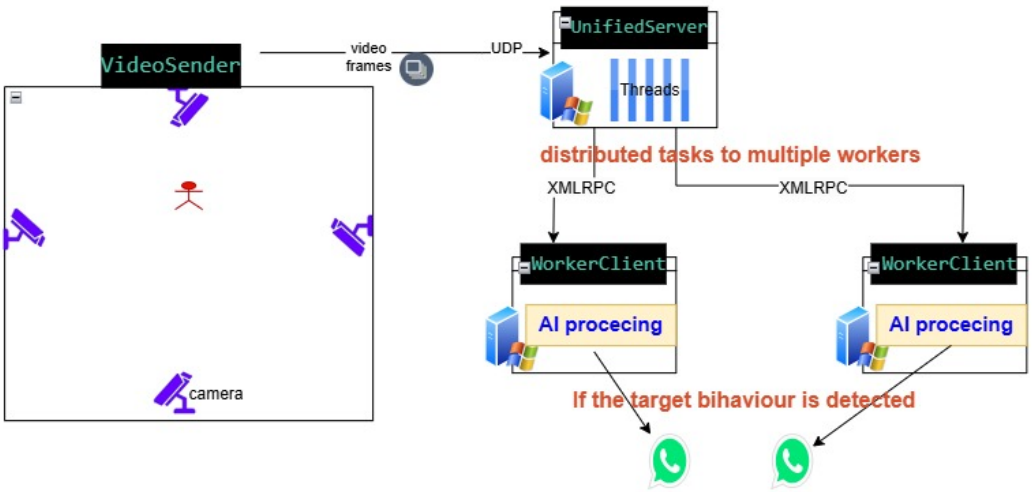

Figure 3.8: Conception of the final Implementation

The updated system merges the VideoReceiver and Master Server into a UnifiedServer to streamline the architecture and potentially improve performance. This consolidation allows for more efficient communication and task distribution between the VideoSender and worker clients. The UnifiedServer receives video frames from the VideoSender, distributes them to multiple worker clients, and coordinates the overall system. Worker clients process the received frames using AI algorithms to detect target behaviors and trigger notifications if necessary. While this merged approach offers potential benefits, it's essential to consider factors like load balancing, redundancy, performance monitoring, and scalability to ensure system reliability and efficiency.

3.3.1 UnifiedServer architecture

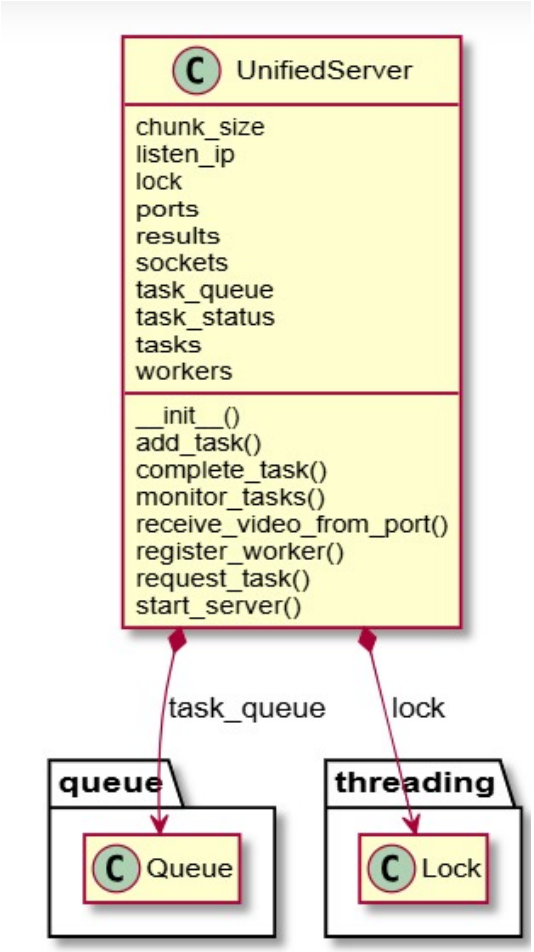

Figure 3.9: UnifiedServer UML Diagram

The `UnifiedServer` class is a central component in a distributed system, responsible for managing tasks and coordinating communication between different components. It has several attributes:

1. **Attributes:**
    - `chunk_size`: defines the size of data chunks to be processed.
    - `listen_ip`: The IP address on which the server listens for incoming connections.
    - `lock`: A synchronization object to protect shared resources.
    - `ports`: A list of ports on which the server listens.
    - `results`: A data structure to store results of processed tasks.
    - `sockets`: A list of socket objects used for network communication.
    - `task_queue`: A queue to store tasks to be processed.

- `task_status:` A data structure to track the status of tasks.
- `tasks:` A list of tasks to be processed.
- `workers:` A list of registered worker clients.

2. **Methods:**
   - `init():` The constructor, initializes the attributes and sets up the server.
   - `add_task():` Adds a new task to the task queue.
   - `complete_task():` Marks a task as completed and potentially removes it from the queue.
   - `monitor_tasks():` Monitors the status of tasks and triggers actions based on their completion.
   - `receive_video_from_port():` Receives video data from a specific port.
   - `register_worker():` Registers a new worker client with the server.
   - `request_task():` Requests a task from the server.
   - `start_server():` Starts the server and begins listening for connections.
3. **Relationships:**
   - `task_queue:` A composition relationship with the `Queue` class. This indicates that the `UnifiedServer` owns the `Queue` object and is responsible for its creation and destruction.
   - `lock:` A usage relationship with the `Lock` class. This indicates that the `UnifiedServer` uses the `Lock` object for synchronization purposes, to protect shared resources like the task queue.
4. **Interpretation:** This class diagram is a server-based system that manages tasks and coordinates communication between workers and other components in a distributed system.

Chapter 4

AI Algorithms

In this chapter, we focus on the use of AI algorithms for real-time video processing. One key model utilized in the system is the pre-trained `facebook/detr-resnet-50`, an object detection model based on the DETR (DEtection TRansformers) architecture. The model's main purpose is to detect and classify objects within video frames.

4.1 facebook/detr-resnet-50

`facebook/detr-resnet-50` is a powerful object detection model that employs transformers for image processing. Unlike traditional CNN-based object detection methods, DETR uses the transformer architecture, which has proven highly effective in natural language processing tasks, for image analysis.

The model typically comes with a `json` file that defines the classes the model is trained to detect. Below is an explanation of the format of this `json` file and the meaning of the classes.

4.2 Classes in the `json` File

The `json` file associated with the `facebook/detr-resnet-50` model contains the labels (class names) that the model is capable of detecting. Each label corresponds to a specific object or entity that the model can identify. Below is an example structure of a `json` file and an explanation of some common class labels.

- The `labels` array in the `json` file holds the names of the object categories that the model can recognize.
- Each object class is represented by a string label, and the model outputs predictions in terms of these classes.

Here is an example snippet of a `json` file for the `facebook/detr-resnet-50` model:

```
{
  "labels": [
    "person",
    "knife",
    "scissors",
```

```
    "bottle",
    "car",
    ...
  ]
}
```

It is commonly useful to fine-tune the model on a large-scale dataset for specific classes to detect them efficiently, or to retrain the model with the same number of classes but on different classes.

4.3 Explanation of Common Classes

Each class is assigned a unique label, and when the model makes a prediction, it returns the class label for the detected object along with the associated bounding box coordinates, which help in locating the object within the image or video frame.

By leveraging this `json` file, the model can efficiently identify a wide range of objects, making it a versatile tool for real-time video processing and object detection.

4.4 showcasing an example of result

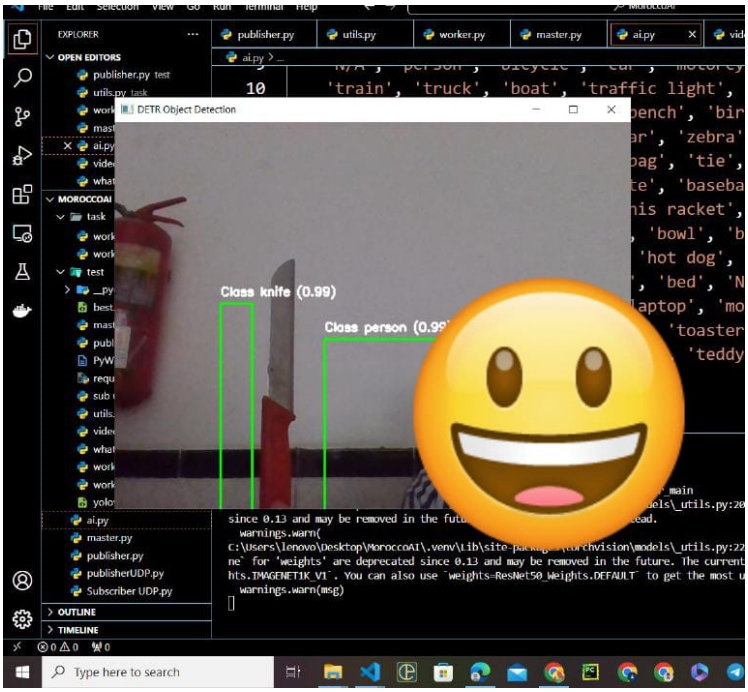

Figure 4.1: Test to detect a knife in real time

As shown, this system detects the knife in real-time. Through our distributed system and IoT approach, it will automatically send an alert to a specific number via WhatsApp.

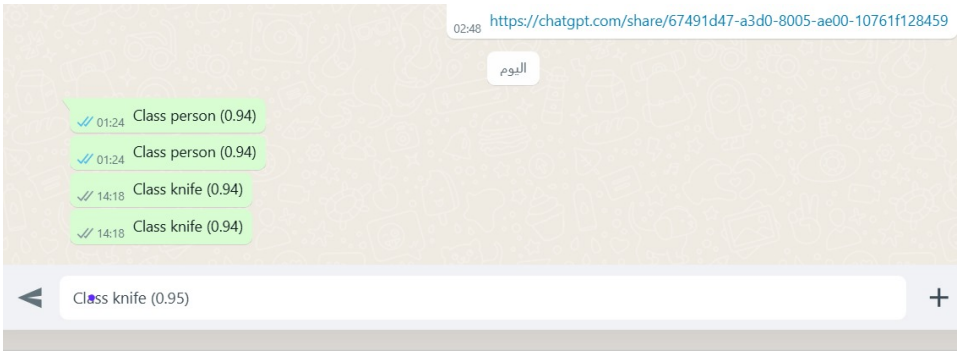

Figure 4.2: Exemple of message recepted in whatsapp after detection

For other social media platforms (such as Gmail), adding also a fog to blure the vison, it only requires adaptation to the specific task. We can easily modify the system to send alerts to a specific location, such as the police. This means that if the owner or any person in the building is in danger and unable to take action, the system will automatically notify the police, ensuring that help arrives without delay.

4.5 Other possible approches:

Our system can be adapted on any other task, it only depends on the model wish classes it detects, for exemple if we want to detect fire and smoke we can fine tune yolov8 model on this specific dataset and load the model in our system :

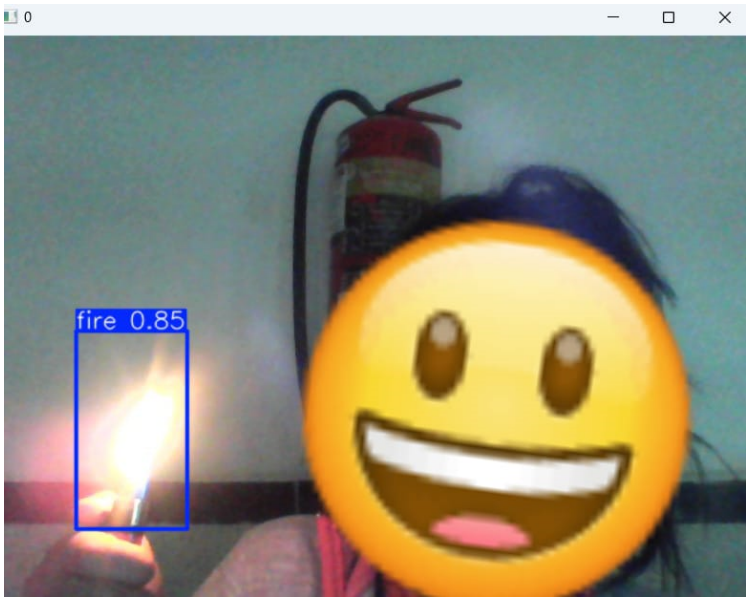

Figure 4.3: Test to detect fire in real time

The fire or smoke for exemple in this use case will be detected once shown and will send an alert

to pompiers in this instance ect...

If for exemple it may requires suspecious behaviour detection in real time it may requires model trained on this specific task .....

4.6 Limitations and further task

We need to adapt this approch in general and train a model from scratch that will handle all usecases possibles ,that will not be possible without adding experience people with us for exemple adding psychologist that will study the specific behaviours of humans and try to adopt in large scaled dataset

Conclusion

In this document, we explored the design, implementation, and application of an AI-driven system for real-time video processing and object detection. We utilized cutting-edge machine learning algorithms, such as the `facebook/detr-resnet-50` model, to create a versatile and scalable solution for detecting a wide range of objects and behaviors in video streams.

The system's architecture is based on a distributed approach, leveraging IoT technologies and real-time data processing. This enables the detection of various objects, such as knives, fire, smoke, and suspicious behavior, with high accuracy. Through the integration of AI models, the system can make autonomous decisions, such as triggering alerts via WhatsApp or other communication platforms, in case of potential danger or emergencies.

We also discussed the flexibility and adaptability of the system. By fine-tuning existing models or incorporating new datasets, the system can be modified to address specific needs, such as detecting fire and smoke using YOLOv8, or monitoring for specific suspicious behaviors. This level of adaptability ensures that the system can be used in various domains, from security surveillance to environmental monitoring and more.

However, the system is not without its challenges. Handling complex scenarios, ensuring real-time performance, and expanding the range of detected objects require continuous refinement and optimization. Moreover, further adaptation is needed to address specific tasks, such as human behavior detection, which may benefit from the inclusion of domain experts, such as psychologists, to better understand and model specific human actions.

Despite these challenges, the system demonstrates considerable promise. With ongoing advancements in AI, computer vision, and IoT, the potential for this technology to provide real-time insights, improve safety, and optimize automation is vast. The ability to detect, process, and respond to dynamic events in real-time holds significant value across a variety of industries, including public safety, transportation, healthcare, and smart cities.

Looking forward, there is immense potential for further enhancement. As AI models continue to evolve, the system can be trained to handle even more complex tasks, making it increasingly effective and versatile. By integrating more specialized expertise and further improving the system's performance, the technology can be deployed in broader applications, ensuring a safer and more efficient environment for everyone.