Cámara de Vigilancia Inteligente con Arduino ESP32 y Alertas Móviles

Documentación del Software

WebApp

Descripción:

Se propone el diseño e implementación de un sistema de vigilancia autónomo e inteligente, construido sobre una plataforma de hardware de bajo costo. El sistema utiliza un módulo ESP32-CAM como unidad central, integrando una cámara para la grabación de eventos y un sensor PIR (Infrarrojo Pasivo) para una detección de movimiento precisa y eficiente en consumo energético.

El objetivo principal es monitorear un área designada y, mediante una lógica de análisis programada, identificar patrones de actividad "anormales". Un patrón anómalo se define como una alta frecuencia de movimientos detectados en un corto intervalo de tiempo (más de 5 detecciones en 30 segundos).

Esquema de conexión (Diagrama de Red):

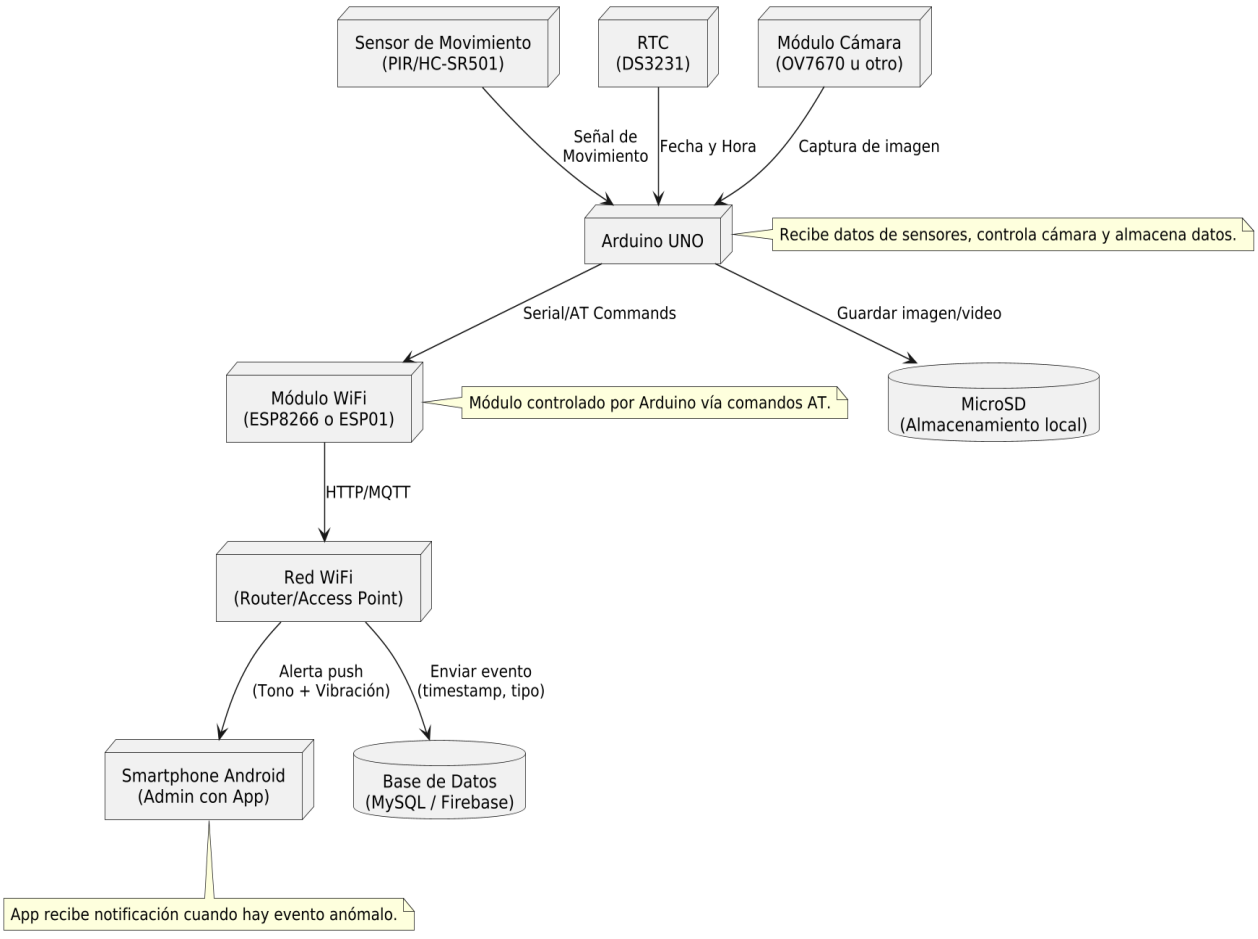
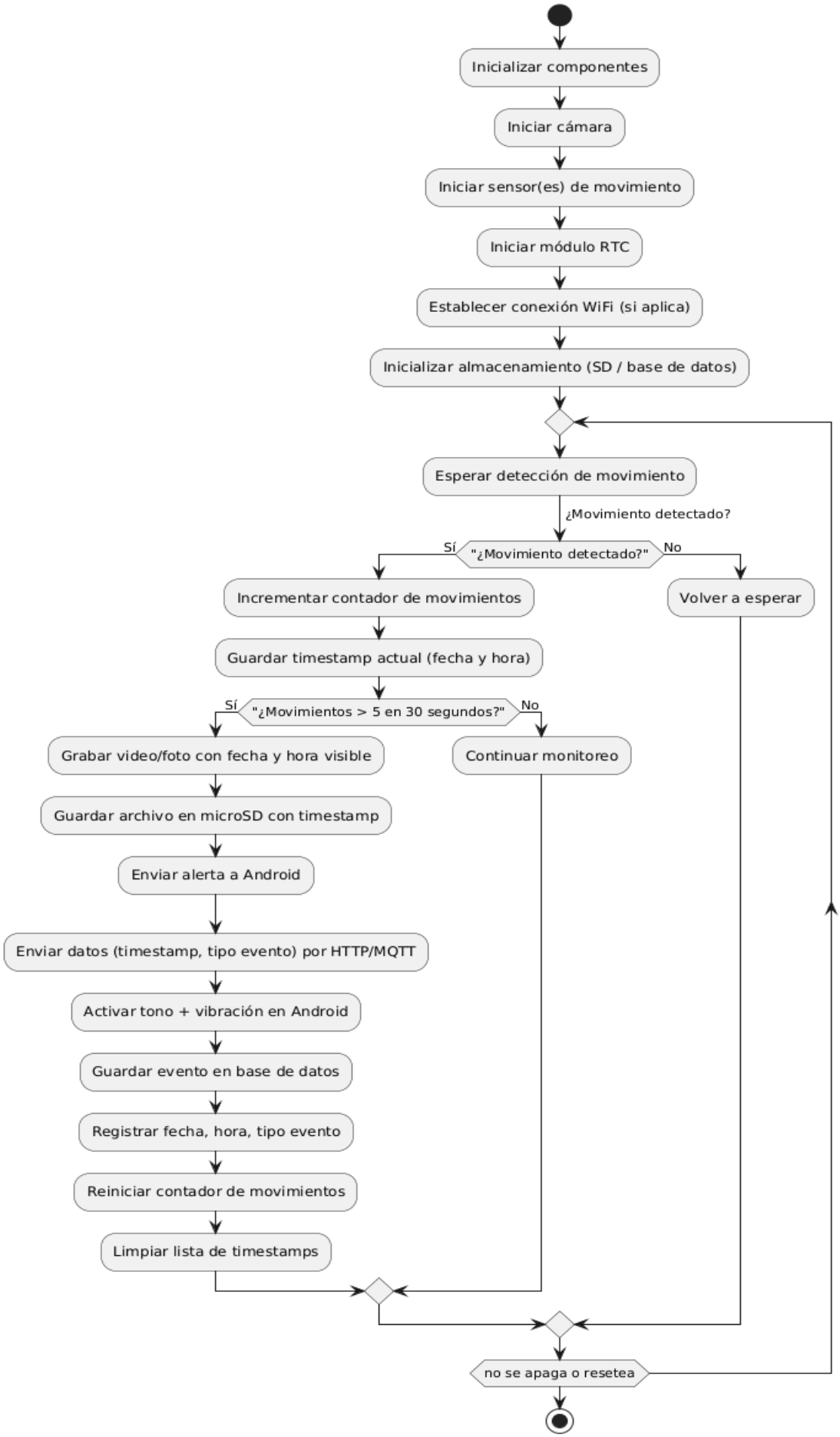


Diagrama de Flujo - Cámara con Detección de Movimiento



Flujo de Alerta y Registro (al detectar una anomalía):

1. Detección de Anomalía: El ESP32 detecta una actividad anómala (motion_count >= 5).

2.Envío de Peticiones Paralelas: El código del ESP32 ejecuta 2 acciones casi simultáneamente:

1. **Conexión con IFTTT** (Para la alerta inmediata):

Realiza una petición HTTP GET o POST a tu URL de Webhook de IFTTT.

Propósito: Desencadenar la notificación push instantánea en tu teléfono a través de la app de IFTTT. Es la forma más rápida y fiable de recibir el "¡Atención!" con sonido y vibración.

2. **Conexión con Servidor Node.js** (Para el registro):

Realiza una petición HTTP POST a la URL de tu API, por ejemplo: http://tu-servidor.com/api/alerta.

La petición incluye en su cuerpo (body) un objeto JSON con el timestamp: { "timestamp": "2023-10-27 18:10:00" }.

3. Procesamiento del Servidor Node.js:

Tu servidor, que está escuchando en la ruta /api/alerta, recibe la petición del ESP32.

Extrae el timestamp del cuerpo de la petición.

4. Conexión del Servidor a la Base de Datos (SQLite):

El servidor ejecuta un comando SQL INSERT en el archivo de la base de datos (anomalias.db).

Propósito: Guardar el evento de forma permanente. Este es el registro que luego consultarás desde tu webapp de React o tu app de Android.

Estructura de archivos:

```
/PRACTICA5
│
├── client/             # React frontend
│   └── src/components/
│
├── server/             # Node.js + Express backend
│   ├── routes/         # rutas de servidor
│   ├── config/         # configuración de bd
│   ├── recordings/     # donde se guardan los clips
│   └── server.mjs      # inicia servidor backend
│
├── database/           # sqlite (base de datos)
├── data.sql            # archivo de datos tabla users
├── squema.sql          # archivo de esquema para SQLite3
├── security_system.db  # sqlite (base de datos)
└── .env                # configs (puerto, DB, JWT)
```

- **client** (carpeta que contiene el frontend web trabajado en **React.js**)
- **server** (carpeta que contiene el backend trabajado en **Node.js y Express**)
- **database** (carpeta contiene los scripts y el archivo .db (**sqlite)** de la base de datos)

Configuración:

Imagina que Node.js es un camarero muy eficiente que espera en tu servidor. Tu ESP32 es el cliente que le va a dar órdenes.

¿Qué es Node.js en este proyecto?

Es un pequeño programa (un servidor backend) que se ejecuta en la nube (en un servicio como Heroku, Vercel o un VPS).

Su único trabajo es:

Escuchar: Esperar a que el ESP32 le envíe un mensaje (una "petición").

Actuar: Cuando recibe un mensaje, lo guarda en una base de datos

2. ¿Cómo se comunican el ESP32 y Node.js?

Se comunican a través de una API REST. Es como si tuvieran una dirección web secreta.

ESP32 (el que habla): Cuando detecta una anomalía, hace una petición HTTP POST a una URL específica, por ejemplo: https://mi-servidor-node.com/api/alerta. En esa petición, envía los datos, como la fecha y la hora.

Node.js (el que escucha): Tu servidor Node.js tiene una "ruta" o "endpoint" que coincide con esa URL (/api/alerta). Cuando llega una petición a esa dirección, Node.js la captura.

Es un servidor Express: Usa import express from 'express'; y const app = express();. Esto es el corazón del servidor.

Está preparado para recibir datos: Con app.use(express.json());, le dices que entienda los datos en formato JSON que le enviará el ESP32.

Está bien organizado: Esta es la clave. En lugar de poner toda la lógica en un solo archivo, tú la separas.

```
// server/server.mjs
import express from 'express';
import cors from 'cors';
import path from 'path';
import { fileURLToPath } from 'url';

import authRoutes from './routes/auth.mjs';

// Resolver __dirname en ES Modules
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Crear app
const app = express();
const PORT = process.env.PORT || 3000;

// Middlewares
app.use(cors());
app.use(express.json());
app.use(express.static(path.join(__dirname, '../public')));

// Rutas
app.use('/api', authRoutes);

// Ruta de estado
app.get('/api/status', (req, res) => {
  res.json({ status: 'online', message: 'Servidor funcionando con SQLite' });
});

// Iniciar servidor
app.listen(PORT, () => {
  console.log(`Servidor SQLite corriendo en http://localhost:${PORT}`);
});
```

server.mjs: Es el recepcionista. Prepara el servidor, pone los "middlewares" (como cors y express.json) y dice: "Todas las peticiones que empiecen con /api se las voy a pasar a mi compañero authRoutes".

routes/auth.mjs: Este es el especialista. Aquí es donde pondrás la lógica para manejar la petición específica de tu ESP32.

Interfaz de Usuario con Login

La aplicación React es una interfaz de usuario dinámica y reactiva. No contiene la lógica pesada, solo se encarga de pedir datos al servidor Node.js y presentarlos de forma amigable para que puedas ver y gestionar tu sistema de vigilancia cómodamente desde cualquier navegador.

La carpeta client/src contiene el código fuente de una Single Page Application (SPA) construida con React.js. Esta es la parte visual y con la que interactúas desde tu navegador (Chrome, Firefox, etc.).

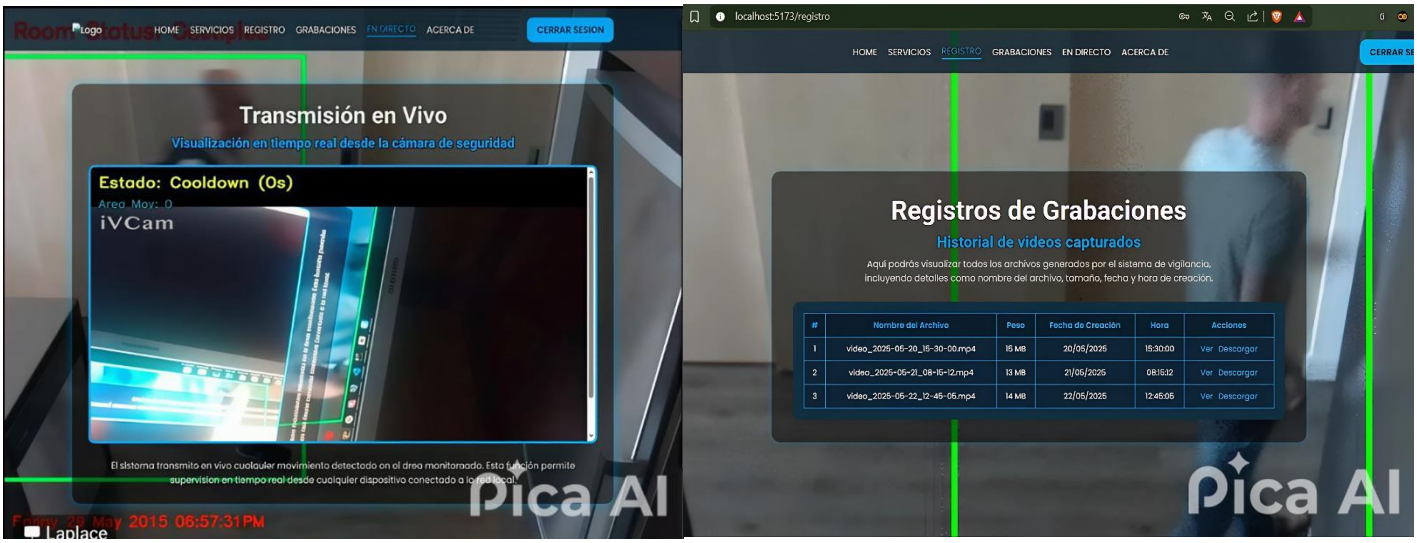
Su propósito principal es servir como un Dashboard de Monitoreo. Desde aquí, puedes:

-Ver el estado del sistema de vigilancia.

-Consultar el historial (registro) de alertas de "anomalías" que el ESP32 ha detectado y que el servidor Node.js ha guardado.

-Ver la transmisión de video en vivo desde el ESP32-CAM.

-Ver las grabaciones en videos .mp4 en donde se registraron anomalías



-Informarte con acerca de que resume la webapp.

Flujo de funcionamiento:

Carga de la página: Cuando abres la webapp en tu navegador, React se carga.

Petición de datos: Un componente de React (por ejemplo, AlertList.js) utiliza una función como fetch o una librería como axios para hacer una petición HTTP GET a una ruta del servidor Node.js. Por ejemplo, a http://tu-servidor.com/api/alertas.

Respuesta del servidor: El servidor Node.js recibe esa petición, consulta la base de datos (SELECT * FROM alertas), y devuelve la lista de todas las alertas guardadas en formato JSON.

Renderizado en pantalla: React recibe esta lista de datos y la utiliza para "pintar" en la pantalla una tabla o una lista con la fecha y hora de cada anomalía.

Aplicación Android

Descripción:

Esta es una aplicación nativa para Android, escrita en Kotlin. Su función principal no es ser un panel de control complejo como la webapp de React, sino actuar como un receptor de alertas prioritarias y un visor rápido de eventos.

La app tiene dos trabajos clave:

Recibir Notificaciones Push Inmediatas: Cuando el ESP32 detecta una anomalía y le avisa a IFTTT (o directamente a Firebase Cloud Messaging), esta app es la que recibe esa notificación push. Despierta el teléfono, reproduce un sonido de alarma personalizado (nokia_escape.mp3) y hace vibrar el dispositivo para captar tu atención de inmediato.

Mostrar el Historial de Anomalías: Una vez abierta, la app se conecta a tu servidor Node.js para obtener y mostrar la lista completa de todas las anomalías que han ocurrido, permitiéndote revisar el historial de eventos.

Funcionamiento y Conexiones:

La aplicación Android es un cliente inteligente que recibe alertas de forma pasiva a través de Firebase y solicita datos de forma activa a tu servidor Node.js cuando la estás usando. La estructura (ViewModel, Repository, Service) sigue las mejores prácticas recomendadas por Google, lo que la hace robusta y escalable.

La app tiene dos canales de comunicación distintos, y es crucial entenderlos:

Canal 1: Recepción de Alertas (Vía Firebase Cloud Messaging - FCM)

FirebaseAlarmService.kt: Este es un "servicio en segundo plano". Su única misión es estar a la escucha de mensajes de FCM. Cuando llega uno, usa el NotificationHelper.kt para construir y mostrar la notificación en la barra de estado del

teléfono, reproduciendo el sonido de la carpeta raw/. Este servicio es la razón por la que recibes alertas incluso si la app está cerrada.

Canal 2: Consulta del Historial (Vía tu API de Node.js)

Este canal se usa cuando tienes la aplicación abierta.

ApiService.kt: Este archivo, que usa la librería Retrofit, es el "mensajero" de la app. Define las URLs a las que la app puede llamar (ej. GET /api/alertas). Retrofit hace que sea muy fácil hablar con tu API de Node.js.

AnomalyRepository.kt: Es el "gerente de datos". Decide si los datos deben venir de la red (llamando al ApiService) o de una caché local (no implementado en tu estructura, pero posible).

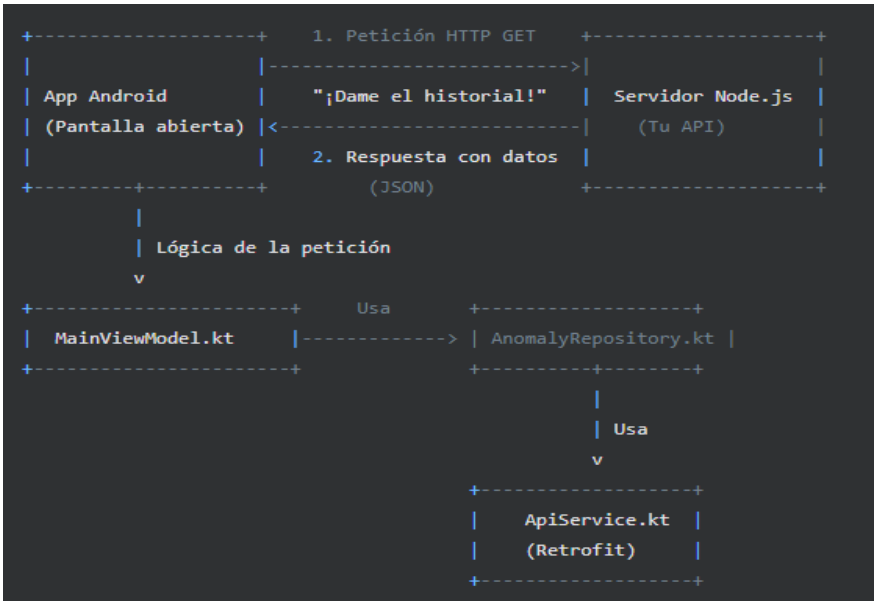
MainViewModel.kt: Es el "cerebro" de la pantalla principal. Cuando la pantalla se crea, le dice al Repository: "Oye, necesito la lista de anomalías". Luego, cuando el Repository le devuelve los datos, el ViewModel actualiza la interfaz de usuario para que veas la lista en pantalla.

Configuración:

build.gradle.kts: Aquí se han añadido las dependencias clave: Retrofit para las llamadas a la API y Gson para convertir los datos JSON del servidor en objetos Kotlin (Anomaly.kt) que la app pueda entender.

AndroidManifest.xml y xml/network_security_config.xml: Estos archivos son muy importantes. Le dicen al sistema operativo Android: "Confío en la conexión a esta dirección IP específica (la de tu PC o servidor). Permíteme conectarme a ella, aunque no sea una dirección HTTPS súper segura". Sin esto, Android bloquearía las conexiones de la app a tu servidor de desarrollo.

google-services.json: Este archivo contiene las claves que conectan tu app con tu proyecto de Firebase, permitiendo que FCM funcione.

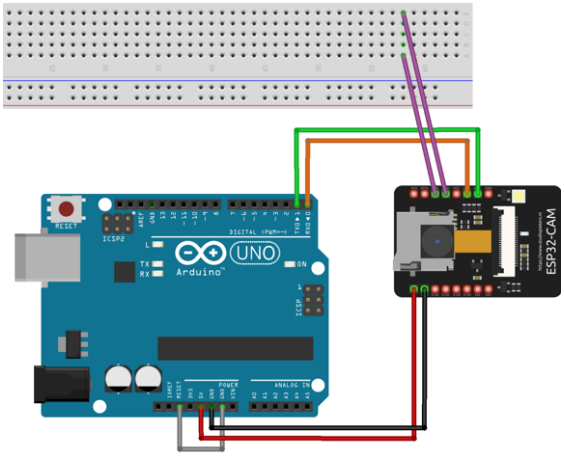


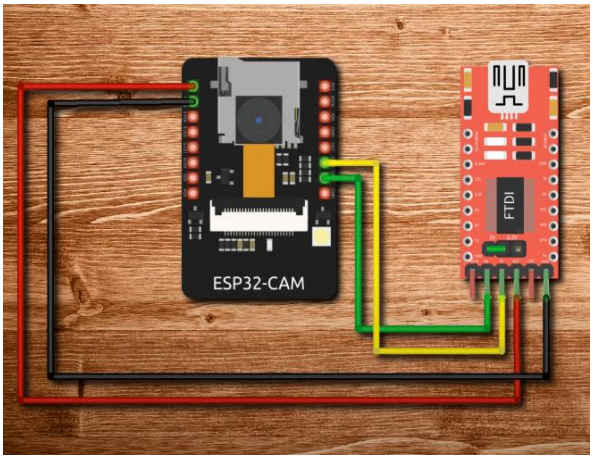
Documentación del Hardware

Descripción:
Se busca diseñar un sistema de videovigilancia de bajo costo utilizando un módulo ESP32-CAM para la transmisión de video en vivo. La detección de movimiento será gestionada mediante en el servidor, con posibilidad de mejorar la automatización de detección del sistema con un sensor PIR.

Componente	Descripción
ESP32-CAM	Microcontrolador con cámara OV2640, WiFi y capacidad de transmisión de video.
Módulo Programador	Adaptador USB a serie (TTL) utilizado para programar el ESP32-CAM desde el PC.
Cables Jumper	Utilizados para realizar las conexiones entre el ESP32-CAM y el módulo.
Fuente de alimentación	Fuente externa de 5V utilizada para el funcionamiento autónomo del ESP32-CAM.
Sensor PIR	Sensor de movimiento por infrarrojos pasivo, que detecta presencia humana.

Esquema de conexión:
Prototipo hecho en Fritzing usado para las pruebas y prototipo final del sistema





Configuración:
Usaremos el software Arduino IDE para configurar y programar el ESP32-CAM.

Parámetros:

- Placa : ESP32 Wrover Module
- Velocidad de carga: 115200 bits por segundo
- Partición de memoria: Huge APP (3MB No OTA)

Un fragmento del código para seleccionar el modelo de la camara y configurar las credenciales del wifi

```
#define CAMERA_MODEL_AI_THINKER

const char *ssid = "**********";
const char *password = "**********";
```

Una vez conectado el ESP32-CAM a la red WiFi, accede a la transmisión en vivo desde el navegador con la IP que responde el monitor de serie:

```
WiFi connecting......
WiFi connected
Camera Ready! Use 'http://192.168.1.117' to connect
```

Documentación del sistema de detección de movimiento

Descripción:
Este módulo implementa un sistema de detección de movimiento en tiempo real utilizando una cámara web (o una fuente de video compatible), la biblioteca OpenCV, y un servidor web construido con Flask. Los eventos de movimiento se registran en una base de datos SQLite3 y pueden visualizarse desde un navegador web a través de un endpoint de video en vivo (/video_feed). Este componente forma parte de un sistema mayor de videovigilancia con módulos adicionales de hardware, webapp, y notificaciones móvil.

Librerias: Instalar todo con pip install opencv-python flask numpy desde la terminal [Python3]

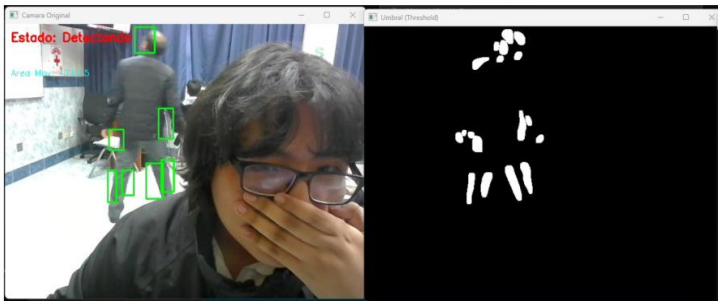
Flujo de Trabajo:

1. Se inicia la cámara.
2. Se calcula la diferencia entre el frames.
3. Si el área de movimiento supera el umbral, se considera un **movimiento brusco**.
4. Si no está en modo "cooldown", se registra el evento en la base de datos.
5. El frame procesado se envía en vivo al navegador.

Ejemplos:
Durante la transmisión, se muestra:

- El estado actual (Detectando, Cooldown, MOVIMIENTO BRUSCO).
- El área total detectada de movimiento.
- Rectángulos verdes sobre lo detectado.

Este es un ejemplo funcional probado en clase y cumpliendo el propósito, el movimiento detectado resalta en la imagen procesada de blanco y negro (el umbral) a la derecha, y en la izquierda la imagen original



Esto sirve para resaltar diferencias en la imagen y es util para motion detection

Un poco de código del procesador de movimiento:

Los eventos se registran en la database con:

- id (entero)
- timestamp (fecha y hora del evento)
- motion_detection (valor numérico del área del movimiento)

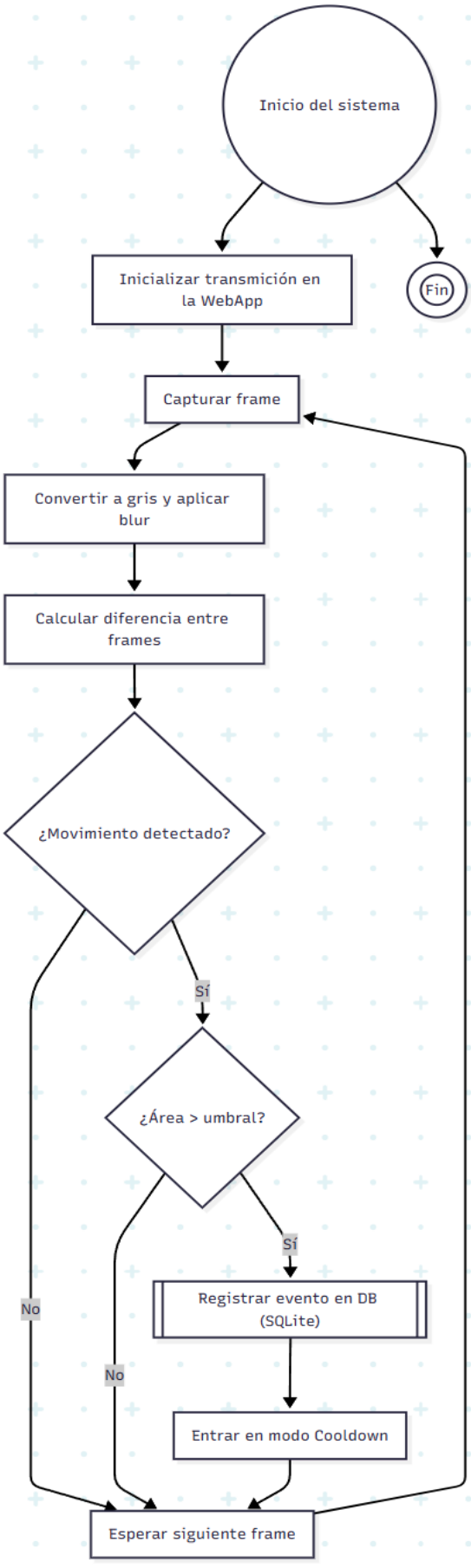
255 pixeles en blanco
0 negro resto del umbral

```
if is_in_cooldown and current_time > cooldown_end_time:
    is_in_cooldown = False

if motion_detected_this_frame and total_motion_area > MOTION_THRESHOLD_AREA and not is_in_cooldown:
    threading.Thread(target=record_motion_event, args=(DEVICE_ID,)).start()
    is_in_cooldown = True
    cooldown_end_time = current_time + COOLDOWN_SECONDS

status_text = "Detectando"
color = (0, 255, 0)
if is_in_cooldown:
    status_text = f"Cooldown ({int(cooldown_end_time - current_time)}s)"
    color = (0, 255, 255)
if motion_detected_this_frame and total_motion_area > MOTION_THRESHOLD_AREA:
    status_text = "MOVIMIENTO BRUSCO"
    color = (0, 0, 255)

cv2.putText(frame, f"Estado: {status_text}", (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7, color, 2)
cv2.putText(frame, f"Area Mov: {int(total_motion_area)}", (10, 60), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
```


Inicio del sistema
Fin
Inicializar transmición en la WebApp
Capturar frame
Convertir a gris y aplicar blur
Calcular diferencia entre frames
¿Movimiento detectado?
Sí
No
¿Área > umbral?
Sí
No
Registrar evento en DB (SQLite)
Entrar en modo Cooldown
Esperar siguiente frame