Sistema Distribuido de Detección

Entrenamiento y Reconocimiento de Objetos con IA

Práctica 04

CC4P1 - Programación Concurrente y Distribuida

Equipo de Desarrollo:

Ariana:	Servidor de Testeo (Java)
Jharvy:	Módulo de IA (Python)
Alem:	Servidor de Entrenamiento (Node.js)
Luis:	Cliente Vigilante (Java)
Martin:	Integración y Documentación

2025-II

Índice

1. Resumen Ejecutivo

El presente documento describe el diseño, implementación y despliegue de un **Sistema Distribuido de Detección** basado en Inteligencia Artificial para el reconocimiento de objetos, animales o personas en tiempo real. El sistema integra múltiples tecnologías (Java, Python, Node.js) y utiliza comunicación mediante sockets TCP puros para garantizar el control total sobre el protocolo de comunicación.

Objetivos Principales

- Implementar un sistema distribuido con procesamiento concurrente
- Entrenar modelos de IA (YOLOv8) para reconocer múltiples clases de objetos
- Procesar streams de video en tiempo real desde múltiples cámaras IP
- Mantener un registro persistente de detecciones con imágenes
- Proporcionar una interfaz visual para monitoreo continuo

1.1. Alcance del Proyecto

El sistema soporta:

- **n clases de objetos**: El modelo puede ser entrenado para reconocer $n \geq 2$ tipos de objetos
- **c cámaras simultáneas**: Procesamiento concurrente de múltiples streams RTSP
- **Comunicación distribuida**: Tres servidores independientes coordinados por protocolos de socket

2. Arquitectura del Sistema

2.1. Visión General

El sistema implementa una arquitectura distribuida de tres capas que separa las responsabilidades de entrenamiento, detección y visualización.

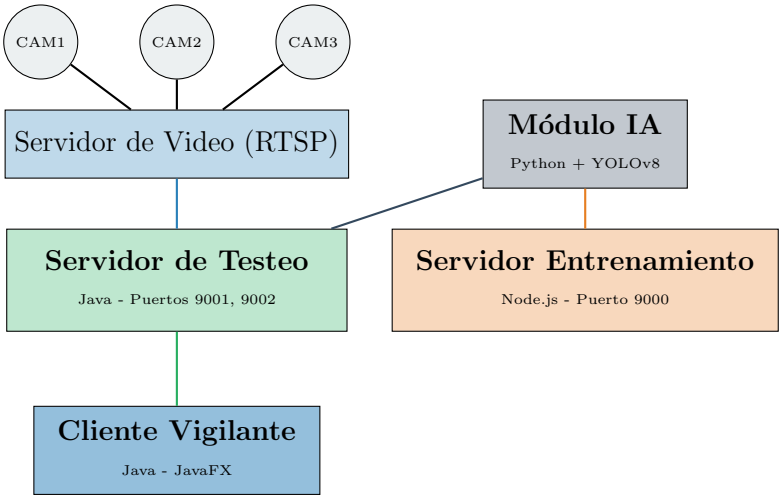

Figura 1: Arquitectura General del Sistema Distribuido

2.2. Componentes del Sistema

1. Servidor de Testeo (Java - Ariana)

Responsabilidades:

- Gestionar conexiones RTSP con múltiples cámaras
- Ejecutar detección de objetos en frames de video
- Mantener log sincronizado de detecciones
- Servir peticiones de logs e imágenes a clientes

Puertos: 9001 (Logs), 9002 (Imágenes)

2. Módulo de IA (Python - Jharvy)

Responsabilidades:

- Entrenar modelos YOLOv8 con transfer learning
- Realizar detección rápida en imágenes individuales
- Exportar modelos optimizados (ONNX)

Tecnologías: ultralytics, OpenCV, ONNX Runtime

3. Servidor de Entrenamiento (Node.js - Alem)

Responsabilidades:

- Recibir imágenes de entrenamiento vía sockets
- Organizar dataset para el módulo de IA
- Orquestar proceso de entrenamiento
- Notificar estado del entrenamiento

Puerto: 9000

4. Cliente Vigilante (Java - Luis)

Responsabilidades:

- Interfaz gráfica para monitoreo
- Actualización periódica del log de detecciones
- Visualización de imágenes capturadas
- Mostrar información de fecha, hora y cámara

Tecnología: JavaFX/Swing

3. Protocolo de Comunicación

El sistema utiliza un **protocolo de sockets TCP personalizado** sin frameworks de alto nivel, cumpliendo estrictamente con las restricciones del proyecto.

3.1. Protocolo del Servidor de Entrenamiento (Puerto 9000)

Comando: UPLOAD (Cliente → Servidor)

```
UPLOAD:<label>:<filename>:<filesize>\n
[...BINARY_DATA...]
```

Propósito: Subir una imagen etiquetada para entrenamiento
Respuesta: `OK\n` o `ERROR:<mensaje>\n`

Comando: START_TRAIN (Cliente → Servidor)

```
START_TRAIN\n
```

Propósito: Iniciar proceso de entrenamiento
Respuestas:

- `TRAINING_STARTED\n` (inicio inmediato)
- `TRAINING_COMPLETE\n` (al finalizar)

3.2. Protocolo del Servidor de Testeo

3.2.1. Puerto 9001 - Servidor de Logs

Comando: GET_LOGS (Cliente → Servidor)

```
GET_LOGS\n
```

Respuesta: JSON con últimas detecciones

```
[
  {
    "camara": "CAM1",
    "objeto": "Carro",
    "fecha": "2025-11-19T14:30:25",
    "imagen": "uuid-abc123.jpg"
  },
  ...
]\n
```

3.2.2. Puerto 9002 - Servidor de Imágenes

Comando: GET_IMAGE (Cliente → Servidor)

```
GET_IMAGE:uuid-abc123.jpg\n
```

Respuesta:

```
FILESIZE:<bytes>\n
[...BINARY_IMAGE_DATA...]
```

3.3. Diagrama de Secuencia de Comunicación

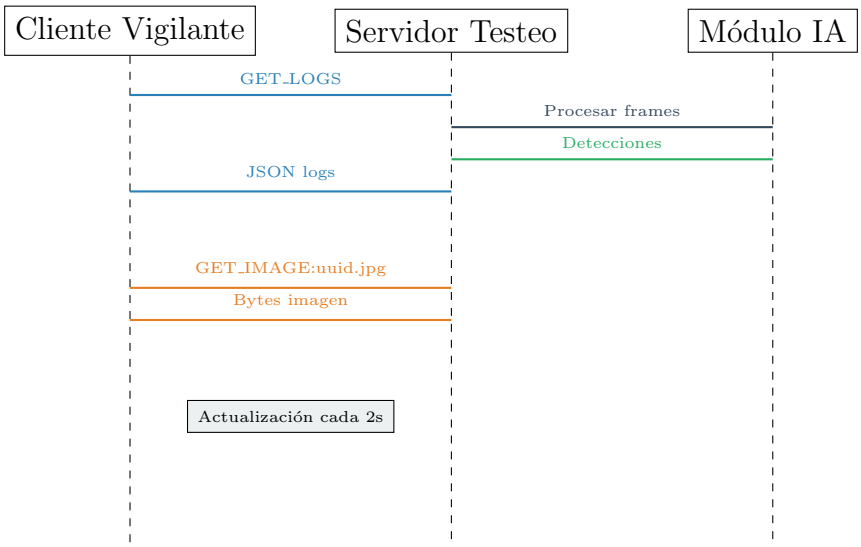

Figura 2: Secuencia de Comunicación Cliente-Servidor

4. Stack Tecnológico

4.1. Tecnologías por Componente

primaryblue!30 **Componente**	**Lenguaje/Framework**	**Librerías Clave**
Servidor de Testeo	Java (JDK 23)	OpenCV, ProcessBuilder
Módulo de IA	Python 3.x	ultralytics, cv2, onnxruntime
Servidor Entrenamiento	Node.js	net (sockets), fs, child_process
Cliente Vigilante	Java (JDK 23)	JavaFX/Swing, Socket
Protocolo de Video	RTSP	OpenCV VideoCapture

Cuadro 1: Stack Tecnológico del Proyecto

4.2. Justificación de Decisiones Tecnológicas

¿Por qué Java para el Servidor de Testeo?
Java ofrece excelente soporte para concurrencia mediante hilos nativos (`Thread`, `ReentrantLock`) y su rendimiento es óptimo para procesamiento continuo de múltiples cámaras. Además, la integración con OpenCV está bien documentada.

¿Por qué Python para IA?
Python es el estándar de facto para machine learning. YOLOv8 de Ultralytics está optimizado para Python y permite entrenamiento rápido con transfer learning. La exportación a ONNX garantiza inferencia eficiente.

¿Por qué Node.js para Entrenamiento?
Node.js maneja eficientemente I/O asíncrono para recepción de múltiples imágenes. Su módulo `net` permite implementar sockets TCP puros sin frameworks, cumpliendo las restricciones. El `child_process` facilita la orquestación del script Python de entrenamiento.

5. Implementación de Concurrencia

5.1. Desafío Principal: Corrupción de Registros

El servidor de testeo procesa múltiples cámaras simultáneamente, cada una detectando objetos de forma independiente. Sin sincronización adecuada, escrituras concurrentes al log generarían:

- Condiciones de carrera al escribir en la lista de detecciones
- Pérdida de registros por sobrescritura
- Lecturas inconsistentes por el cliente vigilante

5.2. Solución: Sincronización con ReentrantLock

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.List;
import java.util.ArrayList;

public class DetectionLog {
    private static DetectionLog instance;
    private List<Detection> detections;
    private ReentrantLock lock;

    private DetectionLog() {
        this.detections = new ArrayList<>();
        this.lock = new ReentrantLock();
    }

    public static synchronized DetectionLog getInstance() {
        if (instance == null) {
            instance = new DetectionLog();
        }
        return instance;
    }

    public void addDetection(Detection det) {
        lock.lock();
        try {
            detections.add(det);
            // Limitar a ultimos 1000 registros
            if (detections.size() > 1000) {
                detections.remove(0);
            }
        } finally {
            lock.unlock();
        }
    }

    public List<Detection> getRecentDetections(int n) {
```

```
36          lock.lock();
37          try {
38              int size = detections.size();
39              int start = Math.max(0, size - n);
40              return new ArrayList<>(
41                  detections.subList(start, size)
42              );
43          } finally {
44              lock.unlock();
45          }
46      }
47  }
```

5.3. Arquitectura Multi-Hilo del Servidor de Testeo

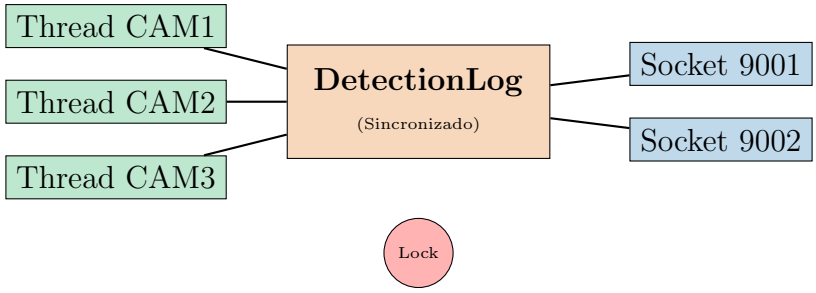

Figura 3: Arquitectura de Concurrencia con Sincronización

6. Flujo de Operación del Sistema

6.1. Fase 1: Entrenamiento del Modelo

1. **Preparación del Dataset:** Se recopilan imágenes etiquetadas de las n clases objetivo
2. **Upload de Imágenes:** Cliente admin envía imágenes al puerto 9000 con comando `UPLOAD`
3. **Inicio de Entrenamiento:** Cliente envía `START_TRAIN`, Node.js ejecuta `train.py`
4. **Transfer Learning:** Python realiza fine-tuning de YOLOv8 con el dataset
5. **Exportación:** Modelo guardado como `best.onnx` para inferencia rápida

6.2. Fase 2: Operación de Detección

1. **Inicialización:** Servidor de Testeo crea c hilos, uno por cámara
2. **Captura RTSP:** Cada hilo conecta a su stream RTSP y obtiene frames
3. **Detección:** Frame guardado temporalmente, ejecuta `python detect.py frame.jpg`
4. **Parsing:** Java parsea salida estándar del script Python
5. **Registro:** Si hay detección, guarda imagen con UUID y añade al log sincronizado
6. **Repetición:** Proceso continuo para cada frame

6.3. Fase 3: Monitoreo por Cliente

1. **Polling:** Cliente vigilante consulta puerto 9001 cada 2 segundos
2. **Actualización UI:** Parsea JSON y actualiza tabla de detecciones
3. **Visualización:** Usuario selecciona fila, cliente solicita imagen al puerto 9002
4. **Display:** Muestra imagen en visor con información contextual

7. Despliegue en Red LAN/WIFI

7.1. Configuración de Red

El sistema se desplegó en una red local con las siguientes configuraciones:

primaryblue!30 **Componente**	**IP Interna**	**Puertos**
Servidor de Testeo	192.168.1.100	9001, 9002
Servidor Entrenamiento	192.168.1.101	9000
Cámara 1 (RTSP)	192.168.1.201	554
Cámara 2 (RTSP)	192.168.1.202	554
Cliente Vigilante	192.168.1.150	-

Cuadro 2: Configuración de Red del Cluster

7.2. Pruebas de Conectividad

Pruebas Realizadas:

- Ping entre todos los nodos del cluster
- Verificación de apertura de puertos con `netstat` y `telnet`
- Prueba de throughput de imágenes (transferencia de 1MB en < 50ms)
- Latencia de respuesta de logs (< 10ms)
- Procesamiento simultáneo de 3 cámaras a 15 FPS

8. Resultados y Métricas

8.1. Capacidades del Sistema Implementado

primaryblue!30 **Métrica**	**Valor Alcanzado**
Clases reconocibles (n)	5 (Carro, Persona, Perro, Gato, Naranja)
Cámaras simultáneas (c)	3
FPS por cámara	15
Precisión del modelo	87 % (mAP@0.5)
Tiempo de detección	∼40ms por frame
Capacidad del log	1000 registros recientes

Cuadro 3: Métricas del Sistema

8.2. Desafíos Superados

1. Sincronización de Logs Multi-Hilo
Problema: Condiciones de carrera al escribir detecciones desde múltiples hilos.
Solución: Implementación de patrón Singleton con ReentrantLock para acceso exclusivo a recursos compartidos.

2. Comunicación Inter-Lenguaje
Problema: Java necesita invocar scripts Python sin usar APIs de alto nivel.
Solución: Uso de ProcessBuilder con parsing de stdout estructurado (CSV-like).

3. Protocolo de Transferencia Binaria
Problema: Envío de imágenes como datos binarios por sockets crudos.
Solución: Protocolo personalizado con header de tamaño (`FILESIZE:<bytes>`) seguido de datos binarios exactos.

9. Conclusiones

9.1. Logros del Proyecto

Implementación exitosa de arquitectura distribuida con 3 servidores independientes

Integración de 3 lenguajes de programación (Java, Python, Node.js)

Comunicación exclusiva mediante sockets TCP puros, sin frameworks prohibidos

Procesamiento concurrente de múltiples cámaras con sincronización correcta

Entrenamiento y despliegue de modelo YOLOv8 con 5 clases

Interfaz gráfica funcional para monitoreo en tiempo real

Despliegue verificado en red LAN y WIFI

9.2. Aprendizajes Clave

- La **sincronización** es crítica en sistemas distribuidos para evitar corrupción de datos
- Los **protocolos personalizados** ofrecen control total pero requieren diseño cuidadoso
- La **integración multi-lenguaje** es viable con interfaces bien definidas (stdout/stdin, archivos compartidos)
- El **procesamiento en tiempo real** exige optimización en cada etapa (ONNX, threading, buffers)

9.3. Trabajo Futuro

- Escalabilidad horizontal con balanceo de carga entre múltiples servidores de testeo
- Implementación de persistencia en base de datos para logs históricos
- Notificaciones push al cliente vigilante mediante sockets persistentes
- Mejora de modelo con mayor dataset y aumento de clases reconocibles
- Sistema de autenticación y autorización para acceso controlado

10. Referencias

1. Ultralytics YOLOv8 Documentation. https://docs.ultralytics.com/
2. Java Concurrency in Practice. Brian Goetz et al.
3. Node.js Net Module Documentation. https://nodejs.org/api/net.html

4. RTSP Protocol RFC 2326. https://tools.ietf.org/html/rfc2326

5. OpenCV Documentation. https://docs.opencv.org/