Spectrum Monitoring Platform

Arquitectura Distribuida para el Monitoreo de Espectro RF

Defensa de Proyecto Final

Departamento de Ingeniería Electrónica y Sistemas

10 de diciembre de 2025

Agenda

1. Descripción General
2. Arquitectura del Sistema
3. Módulo Backend
4. Módulo Sensor (Edge)
5. Conclusiones

Visión General del Sistema

El sistema actúa como un intermediario inteligente entre una red de sensores físicos y una interfaz de usuario (Dashboard ANE).

Características Backend (FastAPI)

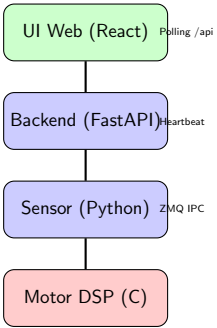
- **API Restful:** Versionado (‘/api/v1‘) y segregado por roles.
- **Seguridad:** Lista blanca de MACs (‘src/macs.json‘).
- **In-Memory:** Gestión de estado volátil para baja latencia.
- **RF Post-Processing:** Cálculo de SNR, Noise Floor y Peak Power *on-the-fly*.

Objetivo Principal

Orquestar la captura de señales RF, procesarlas matemáticamente y servirlas en tiempo real al Frontend.

Arquitectura basada en microservicios y Edge Computing.

Arquitectura de Alto Nivel



- **UI Web:**
  Consumo de datos y envío de configuraciones.
- **Backend:**
  Validación, almacenamiento en RAM y cálculo matemático.
- **Sensor:**
  Gateway Python para orquestación.
- **Motor C:**
  Interacción directa con HackRF y algoritmos DSP.

Lógica de Negocio: Backend

1. Ingesta (Sensor → Backend)

- El sensor consulta configuración (‘GET‘) y envía PSD cruda (‘POST‘).
- Almacenamiento del array P_{xx} en memoria (`device_state`).

2. Consumo y Cálculo (Backend → Frontend)

- Recuperación de la PSD desde RAM.
- Ejecución de `calculate_rf_metrics`:

$$SNR_{dB} = P_{peak} - P_{noise\_floor}$$

- Respuesta JSON lista para graficar.

API Reference: Respuesta al Frontend

Ejemplo de respuesta enriquecida tras el procesamiento matemático:

```
{
  "start_freq_hz": 88000000,
  "center_freq_hz": 98000000,
  "timestamp": 170584123.12,
  "Pxx": [-120.5, -119.0, -115.4],
  "metrics": {
      "noise_floor_dbm": -120.0,
      "peak_power_dbm": -85.2,
      "avg_power_dbm": -110.5,
      "snr_db": 34.8,
      "auto_threshold_dbm": -114.0
  }
}
```

Arquitectura del Sensor (Edge Computing)

El sensor se divide en dos procesos comunicados por **ZMQ (IPC)**:

1. rf_metrics (Motor C)

- **Headless:** Ejecución continua y robusta.
- **DSP:** Welch PSD, Ventaneo y FFT.
- **Watchdog:** Auto-recuperación de hardware USB (HackRF).
- **Logs:** Métricas de CPU, RAM y tiempo de adquisición.

2. Metrics Server (Python)

- **Orquestación:** Cliente HTTP y Gateway ZMQ.
- **Span Chopping:** Optimización de ancho de banda.
- **Gestión:** Rotación de logs y manejo de errores de red.

Optimización: Lógica de "Span Chopping"

Para reducir el consumo de datos (4G/IoT), el script de Python recorta la información antes de enviarla.

1. **Captura:** El HackRF captura un ancho de banda fijo basado en el `sample_rate`.
2. **Procesamiento:** Si el usuario solicita un `span` menor al capturado:
   - Se eliminan los bordes (bins) innecesarios del array P_{xx}.
   - Se reajustan los metadatos de frecuencia.
3. **Resultado:** Reducción drástica del tamaño del payload JSON.

Esto permite monitoreo remoto con bajo consumo de ancho de banda de subida.

Archivos de Log y Auditoría

El sistema genera evidencia local del rendimiento:

Módulo	**Métrica**	**Descripción**
Motor C	`Acq_Time_ms`	Tiempo llenando buffer RX (aire).
Motor C	`DSP_Time_ms`	Tiempo calculando FFT/Welch.
Python	`upload_duration_ms`	Latencia de subida al Backend.
Python	`outgoing_pkg_KB`	Tamaño real del JSON optimizado.

Conclusiones y Estado del Proyecto (v1.3.0)

- **Arquitectura Robusta:** Separación clara de responsabilidades entre C (Cómputo pesado) y Python/FastAPI (Gestión y Lógica).
- **Tolerancia a Fallos:** Mecanismos de reconexión USB automática y validación de timeouts en ZMQ.
- **Eficiencia:**
  - Backend: Baja latencia usando memoria RAM.
  - Sensor: Ahorro de datos mediante *Span Chopping*.
- **Escalabilidad:** Diseño API Restful preparado para múltiples sensores simultáneos.

¿Preguntas?