

# Primer proyecto de SO: Sistema Distribuido con Monitoreo Automático de Recursos y Procesamiento Cooperativo

María José Solís García, Minerva del Carmen Cárdenas Miranda

April 20, 2025

## Abstract

This academic project presents the development of a distributed system focused on real-time resource monitoring and cooperative execution of intensive computational tasks. The main objective is to simulate an environment in which multiple nodes (agents) collaborate to process complex mathematical operations, under the coordination of a central server that dynamically distributes workload based on the current resource usage of each node.

The system consists of Python-based agents that use the psutil library to collect CPU, memory, disk, and network statistics, and execute tasks when assigned. Communication between nodes and the coordinator is managed through Redis, which acts as a real-time messaging system. Task results are stored in a SQL Server relational database for further analysis.

A modern, interactive dashboard was also developed using React, allowing users to visualize the status of each node in real time, as well as a history of executed tasks. The backend, built with FastAPI, exposes a REST API that connects all components and facilitates frontend integration.

This project demonstrates core principles of distributed systems, including inter-process communication, automated decision-making, real-time monitoring, and parallel processing. The proposed solution meets the academic requirements and provides a solid foundation for more scalable distributed architectures.

## 1 Introducción

En el contexto actual de sistemas distribuidos, la eficiencia en la asignación de tareas y la utilización óptima de recursos computacionales se ha convertido en un objetivo prioritario. La creciente demanda de procesamiento intensivo en aplicaciones como visión por computador, inteligencia artificial y análisis multimedia ha impulsado el desarrollo de soluciones que permitan distribuir dinámicamente la carga de trabajo entre múltiples nodos interconectados.

El presente proyecto propone el diseño e implementación de un Sistema Distribuido con Monitoreo Automático de Recursos y Procesamiento Cooperativo, enfocado en el procesamiento de imágenes mediante técnicas de detección facial. Este sistema estará conformado por múltiples nodos (computadoras personales o servidores), que colaboran para procesar imágenes en paralelo, bajo la coordinación de un nodo central. Al mismo tiempo, se implementará un sistema de monitoreo en tiempo real de recursos críticos (CPU, memoria, almacenamiento y red), lo que permitirá redistribuir automáticamente las tareas cuando se detecte saturación o desequilibrio en la carga de trabajo.

Este proyecto no solo permitirá aplicar conocimientos teóricos sobre sistemas distribuidos, administración de recursos y comunicación entre procesos, sino que también representa una solución práctica y escalable que puede extenderse a múltiples aplicaciones del mundo real.

## 2 Primera parte

### 2.1 Investigación y selección tecnológica.

La implementación de un sistema distribuido requiere la integración de múltiples tecnologías que permitan la comunicación eficiente entre procesos, el monitoreo de recursos en tiempo real, la ejecución paralela de tareas, el almacenamiento de información y la presentación visual de los datos.

Para cumplir con los objetivos del proyecto, se realizó una investigación comparativa de distintas herramientas y lenguajes de programación, considerando criterios como: compatibilidad con sistemas Windows, facilidad de desarrollo, comunidad de soporte, rendimiento, escalabilidad y simplicidad de integración entre módulos.

A continuación, se describen las tecnologías seleccionadas y su rol dentro del sistema:

- **Lenguaje de programación: Python 3.10** Python fue elegido por su simplicidad sintáctica, amplio ecosistema de librerías y soporte para programación concurrente y comunicación entre procesos. Además, permite una integración sencilla con Redis y bases de datos como SQL Server mediante bibliotecas específicas.
- **Monitoreo de recursos: psutil** Psutil es una biblioteca de Python ampliamente utilizada para acceder a información del sistema, como el uso de CPU, memoria, disco y red. Cada nodo ejecuta un agente que utiliza psutil para recolectar métricas en tiempo real, que son luego enviadas al coordinador.
- **Sistema de mensajería en tiempo real: Redis** Redis se utiliza como sistema de intercambio rápido de mensajes entre nodos y el coordinador. Su arquitectura basada en memoria, sus estructuras de datos eficientes (listas, hashes, etc.) y su soporte para múltiples clientes simultáneos lo hacen ideal para la comunicación distribuida en este proyecto.
- **Base de datos histórica: SQL Server (SQL Management Studio)** Para el almacenamiento de resultados y análisis posterior, se utiliza Microsoft SQL Server, administrado mediante SQL Server Management Studio (SSMS). Esta elección se debe a su robustez, interfaz gráfica amigable y compatibilidad con sistemas Windows, donde se ejecuta todo el entorno de desarrollo.
- **Backend web: FastAPI** FastAPI fue elegido como framework para construir la API REST del sistema. Permite exponer endpoints de forma sencilla, soporta asincronía nativa, y es altamente eficiente para manejar solicitudes entre el frontend y la base de datos.
- **Frontend web: React (JSX clásico)** El dashboard fue construido con React, una de las bibliotecas más populares para interfaces web modernas. Se optó por React clásico (sin Tailwind) para garantizar la compatibilidad con el entorno de desarrollo. El frontend consulta la API periódicamente y muestra en tiempo real el estado de los nodos y el histórico de tareas.
- **Sistema operativo de desarrollo: Windows 11** Todo el sistema fue diseñado y probado en entorno Windows 11, utilizando Visual Studio Code como entorno de desarrollo, Redis ejecutado localmente como servicio, y SQL Server instalado en la misma máquina.
- **Automatización del entorno: Scripts .bat** Para facilitar la ejecución simultánea del backend, frontend, coordinador, Redis y nodos, se implementó un script .bat que abre cada componente en una terminal separada, iniciando el sistema completo con un solo clic.

## 2.2 Diseño de la arquitectura del sistema

La arquitectura del sistema propuesto se basa en un modelo distribuido maestro-esclavo, donde un componente central (coordinador) toma decisiones y delega tareas a múltiples nodos (agentes). La comunicación entre estos componentes se realiza en tiempo real a través de Redis, mientras que los resultados de las tareas se almacenan en una base de datos relacional para su posterior visualización y análisis.

### 2.2.1 Componentes principales

La arquitectura se compone de los siguientes módulos:

#### **Nodos Agentes**

- Cada nodo actúa como una unidad de procesamiento independiente.
- Ejecuta tareas matemáticas intensivas recibidas desde Redis.
- Reporta su estado de recursos periódicamente (CPU, RAM, Disco, Red).

- Envía los resultados de las tareas nuevamente a Redis para su procesamiento por el coordinador.

**Coordinador** Es el componente central encargado de:

- Monitorear el estado de los nodos.
- Asignar tareas al nodo menos cargado.
- Recibir resultados desde Redis.
- Guardar los resultados en SQL Server para análisis histórico.

**Redis (Broker de mensajes)** Se utiliza para:

- Encolar tareas pendientes por nodo (node:¡id¡:tasks).
- Almacenar resultados pendientes en results:queue.
- Guardar los datos de monitoreo de cada nodo (node:¡id¡:stats).
- Facilita una comunicación eficiente y desacoplada entre los componentes.

**Base de datos (SQL Server)**

- Contiene la tabla TaskHistory que almacena:ID del nodo, tipo de tarea, resultado, fecha y hora de ejecución.
- Permite generar reportes y mantener un historial de ejecución.

**Base de datos (SQL Server)**

- Contiene la tabla TaskHistory que almacena:ID del nodo, tipo de tarea, resultado, fecha y hora de ejecución.
- Permite generar reportes y mantener un historial de ejecución.

**Backend (FastAPI)** Expone una API REST que permite al frontend:

- Consultar los nodos activos y sus métricas.
- Ver el historial de tareas procesadas.
- Actúa como puente entre SQL Server, Redis y React.

**Frontend (React)** Dashboard moderno e intuitivo que muestra:

- Tarjetas con métricas de los nodos activos.
- Tabla con el historial de tareas ejecutadas.
- Consulta periódicamente la API para mantenerse actualizado en tiempo real.

## 2.3 Flujo de datos

1. Cada nodo monitorea su estado con psutil y lo envía a Redis.
2. El coordinador consulta esos datos y decide a qué nodo enviar la próxima tarea.
3. La tarea se coloca en una cola específica del nodo (node1:tasks, node2:tasks, etc.).
4. El nodo la recoge, la ejecuta, y coloca el resultado en results:queue.
5. El coordinador recoge el resultado y lo guarda en SQL Server.
6. El dashboard React consulta la API y muestra los datos actualizados al usuario.

## 2.4 Modelo lógico general

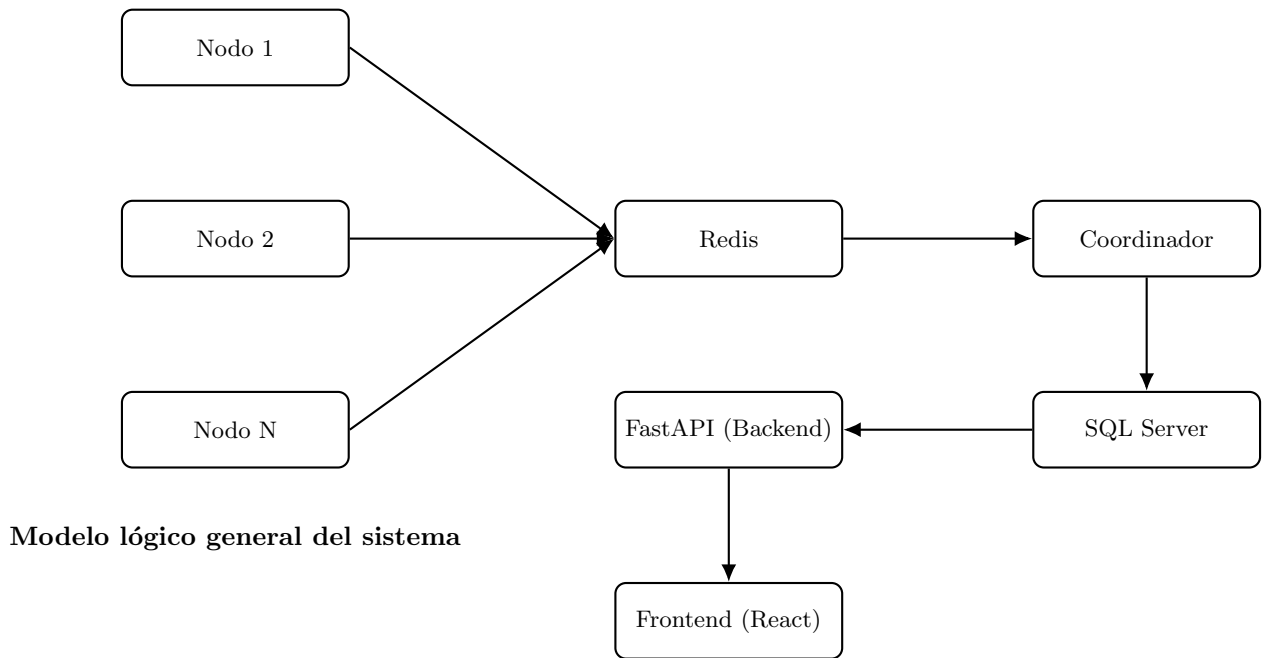


Figure 1: Diagrama lógico de interacción entre los componentes del sistema distribuido.

## 3 Segunda parte

### 3.1 Diseño e Implementación

El sistema fue diseñado de forma modular para garantizar claridad, mantenibilidad y escalabilidad. Se estructuró en tres capas funcionales: agentes de ejecución, coordinador central, y capa de monitoreo y visualización, las cuales interactúan entre sí mediante Redis y una API desarrollada con FastAPI.

A continuación, se describe cada componente, su diseño interno y la forma en que fueron implementados:

#### 3.1.1 Agentes (Nodos de Ejecución)

Cada agente representa un nodo del sistema distribuido. Su función principal es ejecutar tareas intensivas enviadas por el coordinador y monitorear sus propios recursos.

##### Diseño:

- Los agentes se ejecutan como scripts Python individuales.
- Cada agente utiliza psutil para obtener el estado actual de CPU, RAM, disco y red.
- Publica esos datos en Redis en una clave tipo `node:¡id!:stats`.
- Escucha una cola específica en Redis (`node:¡id!:tasks`) para recibir tareas.
- Una vez ejecutada la tarea, envía el resultado a `results:queue`.

##### Implementación:

- Lenguaje: Python
- Librerías: psutil, redis, json, threading
- Componente principal: `agent.py` + `taskexecutor.py`

### 3.1.2 Coordinador Central

El coordinador es el encargado de tomar decisiones. Asigna tareas automáticamente a los nodos menos cargados y gestiona el almacenamiento de resultados. **Diseño:**

- Consulta continuamente los datos de todos los nodos desde Redis.
- Usa una estrategia simple de balanceo de carga (menor CPU utilizada).
- Envía tareas al nodo elegido mediante Redis (`rpush node:idx:tasks`).
- Escucha la cola `results:queue` para recibir resultados procesados.
- Inserta cada resultado en la base de datos SQL Server.

#### Implementación:

- Lenguaje: Python
- Conexión a Redis: `redis-py`
- Conexión a SQL Server: `pyodbc`
- Estructura modular:
  - `coordinatorMain.py`: lógica principal
  - `taskScheduler.py`: asignación de tareas
  - `resultProcessor.py`: recepción e inserción de resultados
  - `database.py`: conexión e inserción a SQL Server

### 3.1.3 Backend (API REST)

El backend actúa como intermediario entre el sistema y el dashboard, sirviendo los datos desde Redis y SQL Server al frontend en formato JSON.

#### Diseño:

- Implementado con FastAPI, usando endpoints REST.
- Expone:
  - `/api/nodes`: lista de nodos activos y su estado.
  - `/api/tasks`: historial de tareas ejecutadas.
- Conexión directa con Redis y SQL Server.

#### Implementación:

- Lenguaje: Python
- Framework: FastAPI
- Entorno: entorno virtual con `uvicorn` para ejecución
- Archivo principal: `main.py`
- Módulo auxiliar: `sqlclient.py`

### 3.1.4 Frontend (Dashboard Web)

La interfaz gráfica del sistema permite visualizar el estado de los nodos en tiempo real y el historial de tareas.

#### Diseño:

- Construido en React usando JSX clásico.
- Interfaz moderna y responsiva.
- Tarjetas para mostrar información de cada nodo.
- Tabla para visualizar el historial con actualización automática.

#### Implementación

- Lenguaje: JavaScript
- Librerías: React, Fetch API
- Componentes:
  - App.jsx: lógica principal y estructura
  - NodeCard.jsx: tarjeta individual por nodo
  - TaskTable.jsx: tabla del historial
- Estilos definidos en App.css (personalizados)

### 3.1.5 Monitoreo en Tiempo Real con Redis

Redis permite la comunicación asincrónica entre componentes y garantiza baja latencia en el sistema.

#### Diseño

- Estructuras utilizadas:
  - node:idx:stats: estado del nodo en formato JSON
  - node:idx:tasks: cola de tareas asignadas
  - results:queue: cola de resultados completados
- Acceso concurrente por múltiples componentes.

#### Implementación

- Redis local ejecutado en Windows
- Ruta: Redis-x64-5.0.14.1/redis-server.exe
- Integración vía redis-py desde Python

### 3.1.6 Ejecución del sistema completo

Para facilitar la puesta en marcha, se creó un script .bat que inicia todos los componentes en terminales independientes:

- Redis
- Backend FastAPI
- Frontend React
- Coordinador
- Nodo 1
- Nodo 2

Esto permite ejecutar todo el sistema automáticamente con un doble clic.

## 4 Pruebas y resultados

Para validar el funcionamiento correcto del sistema distribuido, se diseñaron y ejecutaron una serie de pruebas que abarcan tanto la ejecución individual de cada componente como la interacción entre ellos. Estas pruebas se enfocaron en comprobar:

- La conectividad y comunicación entre los módulos (nodos, coordinador, Redis, backend, frontend).
- El correcto procesamiento de tareas por parte de los nodos.
- El monitoreo en tiempo real de recursos.
- La visualización de datos en el dashboard.
- El almacenamiento persistente en la base de datos.

A continuación, se describen las principales pruebas realizadas y los resultados obtenidos.

### 4.1 Prueba de monitoreo de recursos en tiempo real

**Objetivo:** Verificar que cada nodo capture y envíe correctamente sus métricas (CPU, RAM, Disco, Red) a Redis. **Procedimiento:**

- Se ejecutó un nodo con el script `agent.py`.
- Se monitoreó Redis para verificar la presencia de claves `node:nodo1:stats`.
- Se consultó manualmente la API: `http://localhost:8000/api/nodes`.

**Resultado obtenido:** Métricas actualizadas correctamente en Redis y visualizadas en tiempo real desde el dashboard

### 4.2 Prueba de asignación automática de tareas

**Objetivo:** Verificar que el coordinador asigne tareas al nodo con menor carga y que este la procese. **Procedimiento:**

- Se ejecutaron 2 nodos (nodo1 y nodo2).
- Se dejó correr el coordinador (`coordinatormain.py`) con tareas periódicas.
- Se observó la consola del coordinador y los agentes.

**Resultado obtenido:** Tareas asignadas a nodo1 y nodo2 de forma balanceada, y correctamente procesadas.

### 4.3 Prueba de procesamiento de resultados y almacenamiento

**Objetivo:** Comprobar que los resultados generados por los nodos sean capturados por el coordinador y almacenados en la base de datos SQL Server.

**Procedimiento:**

- Se ejecutaron nodos + coordinador + backend.
- Se observaron los logs de consola del coordinador (Resultado recibido... y Resultado guardado...).
- Se consultó la tabla `TaskHistory` desde SQL Server Management Studio.

**Resultado obtenido:** Registros almacenados correctamente en la base de datos.

## 4.4 Prueba de visualización en el dashboard

**Objetivo:** Verificar que el frontend se actualice automáticamente y refleje el estado del sistema.

**Procedimiento:**

- Se accedió al dashboard en `http://localhost:3000`.
- Se observaron:
  - Tarjetas por nodo mostrando métricas.
  - Tabla con tareas históricas.

**Resultado obtenido:** Interfaz clara, actualizaciones automáticas confirmadas, historial visible.

## 4.5 Prueba de ejecución completa con script automatizado

**Objetivo:** Validar la ejecución integral del sistema con un solo clic.

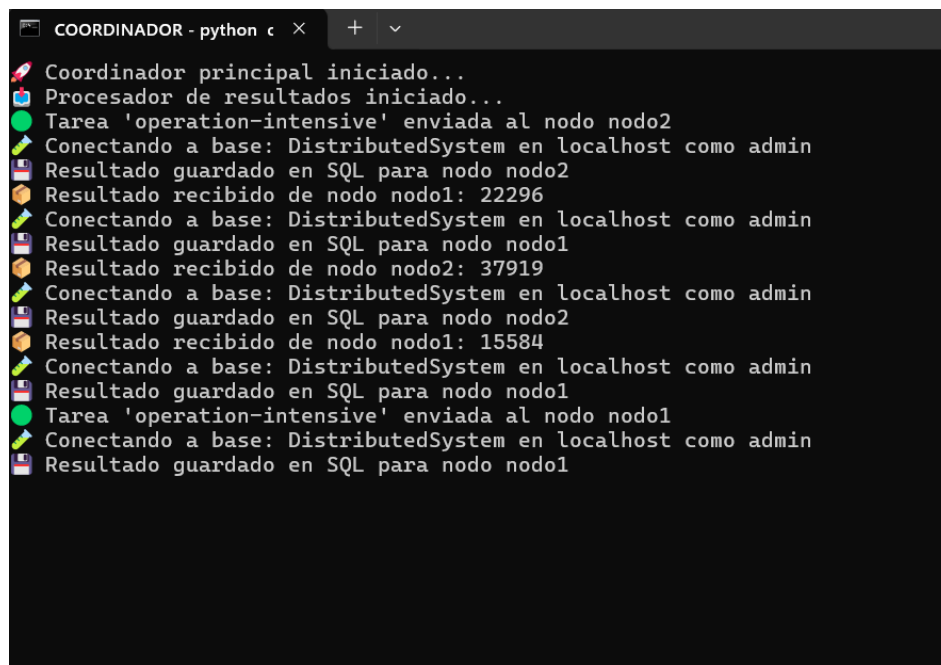
**Procedimiento:**

- Se ejecutó el archivo `iniciarsistemacompleto.bat`.
- Se verificó que se abran todas las ventanas necesarias:
  - Redis
  - Backend
  - Frontend
  - Coordinador
  - Nodo1 y Nodo2

**Resultado obtenido:** Sistema completo ejecutado correctamente con una sola acción.

## 4.6 Evidencias

- Terminal del Coordinador



```
COORDINADOR - python c x + v
Coordinador principal iniciado...
Procesador de resultados iniciado...
Tarea 'operation-intensive' enviada al nodo nodo2
Conectando a base: DistributedSystem en localhost como admin
Resultado guardado en SQL para nodo nodo2
Resultado recibido de nodo nodo1: 22296
Conectando a base: DistributedSystem en localhost como admin
Resultado guardado en SQL para nodo nodo1
Resultado recibido de nodo nodo2: 37919
Conectando a base: DistributedSystem en localhost como admin
Resultado guardado en SQL para nodo nodo2
Resultado recibido de nodo nodo1: 15584
Conectando a base: DistributedSystem en localhost como admin
Resultado guardado en SQL para nodo nodo1
Tarea 'operation-intensive' enviada al nodo nodo1
Conectando a base: DistributedSystem en localhost como admin
Resultado guardado en SQL para nodo nodo1
```

Figure 2: Coordinador mostrando tareas enviadas y resultados guardados



- Terminal de un nodo

```
AGENTE1 - python -m ager x + v
Agente 'nodo1' iniciado...
✓ Tarea ejecutada en nodo nodo1: 2787 → 16722
✓ Tarea ejecutada en nodo nodo1: 7792 → 70128
✓ Tarea ejecutada en nodo nodo1: 7951 → 55657
✓ Tarea ejecutada en nodo nodo1: 7303 → 51121
✓ Tarea ejecutada en nodo nodo1: 4458 → 35664
✓ Tarea ejecutada en nodo nodo1: 1584 → 11088
✓ Tarea ejecutada en nodo nodo1: 8122 → 48732
✓ Tarea ejecutada en nodo nodo1: 1732 → 8660
✓ Tarea ejecutada en nodo nodo1: 5050 → 25250
✓ Tarea ejecutada en nodo nodo1: 3173 → 25384
✓ Tarea ejecutada en nodo nodo1: 7383 → 22149
✓ Tarea ejecutada en nodo nodo1: 7713 → 30852
✓ Tarea ejecutada en nodo nodo1: 5459 → 32754
✓ Tarea ejecutada en nodo nodo1: 1622 → 14598
✓ Tarea ejecutada en nodo nodo1: 2877 → 20139
✓ Tarea ejecutada en nodo nodo1: 1095 → 6570
```

Figure 3: Nodo1 mostrando tareas ejecutadas

- Dashboard

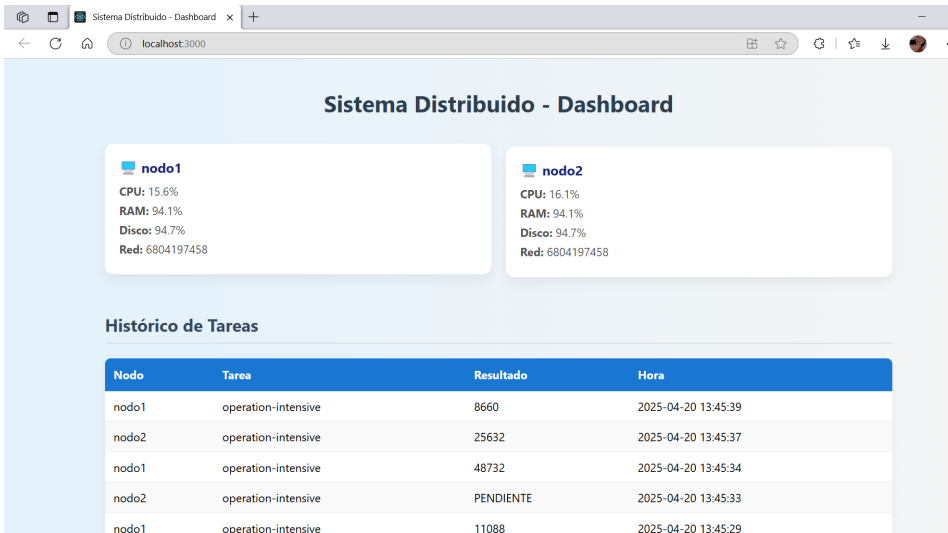


Figure 4: Dashboard con tarjetas y tabla de historial.

- SQL (TaskHistory)

Results		Messages			
	ID	NodeID	TaskType	Result	Timestamp
1	1	nodo1	operation-intensive	PENDIENTE	2025-04-19 11:25:45.000
2	2	nodo1	operation-intensive	PENDIENTE	2025-04-19 11:25:55.000
3	3	nodo1	operation-intensive	PENDIENTE	2025-04-19 11:26:05.000
4	4	nodo1	operation-intensive	PENDIENTE	2025-04-19 11:26:23.000
5	5	nodo1	operation-intensive	43386	2025-04-19 11:26:26.000
6	6	nodo1	operation-intensive	36722	2025-04-19 11:26:31.000
7	7	nodo1	operation-intensive	PENDIENTE	2025-04-19 11:26:33.000
8	8	nodo1	operation-intensive	85570	2025-04-19 11:26:36.000
9	9	nodo1	operation-intensive	12876	2025-04-19 11:26:41.000
10	10	nodo1	operation-intensive	PENDIENTE	2025-04-19 11:26:43.000
11	11	nodo1	operation-intensive	22296	2025-04-19 11:26:46.000
12	12	nodo1	operation-intensive	15584	2025-04-19 11:26:51.000
13	13	nodo1	operation-intensive	PENDIENTE	2025-04-19 11:26:53.000
14	14	nodo1	operation-intensive	63608	2025-04-19 11:26:56.000
15	15	nodo1	operation-intensive	58424	2025-04-19 11:27:01.000
16	16	nodo2	operation-intensive	PENDIENTE	2025-04-19 11:27:03.000

Figure 5: Resultado en SQL Server (SELECT \* FROM TaskHistory).

Todas las pruebas funcionales clave fueron exitosas. El sistema cumple con sus objetivos académicos y técnicos, demostrando su funcionamiento correcto tanto en procesamiento distribuido como en monitoreo en tiempo real y persistencia de resultados.

## 5 Conclusiones

El desarrollo del proyecto "Sistema Distribuido con Monitoreo Automático de Recursos y Procesamiento Cooperativo" permitió poner en práctica de forma integral los conceptos clave abordados durante el curso de Sistemas Distribuidos, demostrando cómo múltiples tecnologías pueden integrarse armónicamente para resolver un problema real mediante arquitectura distribuida.

Durante la implementación se logró:

- Diseñar e implementar un sistema completamente modular y escalable, que separa claramente los roles de monitoreo, coordinación, procesamiento y visualización.
- Integrar herramientas de software como Python, Redis, SQL Server, FastAPI y React, logrando una solución funcional de punta a punta.
- Simular un entorno de ejecución distribuido con múltiples nodos trabajando de forma paralela y cooperativa.
- Implementar lógica de asignación automática de tareas en función del estado actual de los recursos, mejorando la eficiencia del procesamiento.
- Monitorear en tiempo real el estado de los nodos y visualizarlo mediante un dashboard moderno e intuitivo.
- Registrar los resultados de las tareas de forma persistente para análisis y trazabilidad futura.

Además de los objetivos técnicos alcanzados, el proyecto fortaleció habilidades en trabajo modular, depuración en entornos concurrentes, manejo de eventos en tiempo real, configuración de servicios en sistemas Windows y desarrollo de interfaces web.

Como posibilidad de mejora futura, se plantea:

- Incluir autenticación de usuarios y seguridad en el acceso al dashboard.
- Agregar un módulo de alertas o eventos críticos basado en umbrales de uso de recursos.

- Permitir la ejecución remota real entre diferentes dispositivos conectados en red.
- Incluir una interfaz de configuración para agregar o quitar nodos de forma dinámica.

El sistema entregado no solo cumple con los requerimientos académicos del proyecto, sino que representa una base sólida para el desarrollo de soluciones distribuidas más complejas en entornos reales o de producción.

## References

- [1] Redis Ltd. *Redis Documentation*. <https://redis.io/docs/>. Accedido: 20 abril 2025. 2025.
- [2] Meta Platforms Inc. *React Official Documentation*. <https://reactjs.org>. Accedido: 20 abril 2025. 2025.
- [3] Microsoft. *SQL Server Documentation*. <https://learn.microsoft.com/sql>. Accedido: 20 abril 2025. 2025.
- [4] Overleaf. *Overleaf LaTeX Documentation*. <https://www.overleaf.com/learn>. Accedido: 20 abril 2025. 2025.
- [5] Python Software Foundation. *Python 3.10 Documentation*. <https://docs.python.org/3.10>. Accedido: 20 abril 2025. 2025.
- [6] Sebastián Ramírez. *FastAPI Documentation*. <https://fastapi.tiangolo.com>. Accedido: 20 abril 2025. 2025.
- [7] Giampaolo Rodola. *psutil - Process and System Utilities*. <https://psutil.readthedocs.io>. Accedido: 20 abril 2025. 2025.
- [8] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. 2nd. Pearson Education, 2007.