Sistema Distribuido de Procesamiento de Imágenes con Monitoreo de Recursos

Justin Alonso Martínez Molina 2019027054
Kevin Jiménez Torres 2019167495
Gabo Melano Ruiz 1996057891
Mathew Carballo López 2022218443

Resumen

Este informe describe el diseño e implementación de un **sistema distribuido escalable** para procesamiento cooperativo de tareas de filtro de imágenes, con **monitoreo automático en tiempo real de los recursos** de cada nodo. El sistema se compone de varios componentes: un **coordinador central** que recibe las solicitudes de tareas (como aplicar filtros a imágenes), múltiples **nodos worker** que ejecutan las tareas, un servicio de **monitoreo** que recopila métricas de uso de CPU, memoria, disco y red de cada nodo, una base de datos en memoria **Redis** que actúa como intermediario de mensajería, y un **dashboard web** desarrollado en React para visualizar los datos de monitoreo. La asignación de tareas se realiza mediante un **balanceo de carga basado en menor uso de CPU**, optimizando dinámicamente la distribución según el estado actual de los recursos. Para la implementación se emplearon contenedores **Docker** por su portabilidad y fácil despliegue. El coordinador y el servicio de monitoreo se implementaron con el framework **Flask** de Python, los trabajadores utilizan la librería **psutil** para obtener métricas del sistema y la biblioteca **Pillow (PIL)** para el procesamiento de imágenes, mientras que el frontend emplea **React** para construir la interfaz de usuario. Los resultados muestran que el sistema logra distribuir cargas de trabajo equitativamente entre los nodos y proporciona métricas en tiempo real para optimizar el rendimiento. Se analizan en detalle la arquitectura, la metodología de desarrollo, el funcionamiento del sistema y del dashboard, así como las métricas obtenidas. Finalmente se discuten limitaciones y se proponen líneas de trabajo futuro para mejorar el sistema (por ejemplo, incluir más criterios de balanceo, incorporar algoritmos inteligentes, ampliar tipo de tareas, etc.).

Palabras clave: Sistemas distribuidos, procesamiento de imágenes, monitoreo de recursos, Docker, Flask, Redis, React, psutil, Pillow, balanceo de carga, métricas en tiempo real.

Introducción

El procesamiento distribuido de tareas computacionales intensivas es una estrategia clave para mejorar el rendimiento y la escalabilidad de aplicaciones en las que se deben procesar grandes volúmenes de datos o ejecutar cálculos complejos (por ejemplo, **filtrado de imágenes**, análisis de datos multimedia, cálculos matemáticos avanzados, etc.). En este contexto, es esencial conocer el estado de los recursos (CPU, memoria, disco y red) de cada máquina participante, de forma que se

pueda **optimizar dinámicamente la asignación de tareas** y evitar sobrecargar nodos con baja disponibilidad. El proyecto "Sistema de Procesamiento y Monitoreo de Recursos" (I Semestre 2025) plantea desarrollar un sistema distribuido donde **múltiples nodos cooperan** para ejecutar tareas intensivas, mientras un componente central monitorea en tiempo real las métricas de cada nodo y decide la redistribución de cargas.

El presente informe documenta la implementación de dicho sistema para el caso de tareas de **procesamiento de imágenes** aplicando filtros (blanco/negro, sepia, negativo, desenfoque, bordes, etc.), junto con el monitoreo automático de recursos del sistema. Se utilizó una arquitectura de **microservicios** orquestados con Docker: un coordinador en Flask, varios workers que ejecutan tareas con Python (usando Pillow para imágenes y psutil para métricas), un servicio de monitoreo con Flask que registra datos en Redis, y una interfaz web en React para el dashboard. La asignación de tareas se realiza a través de Redis como cola de mensajes, consultando periódicamente el monitoreo de cada nodo para decidir el servidor menos cargado (menor uso de CPU), implementando así una forma de **balanceo de carga** dinámico.

Este informe abarca la descripción detallada del sistema: los objetivos propuestos (general y específicos), la metodología de desarrollo, la arquitectura completa (coordinador, workers, monitoreo, Redis, frontend), las tecnologías utilizadas y su rol, el algoritmo de balanceo, el monitoreo de métricas y funcionamiento del dashboard. Se incluyen los resultados obtenidos (ejecución de pruebas de asignación de tareas y visualización de métricas en tiempo real), seguidos de una discusión de los hallazgos y conclusiones. Además se propone trabajo futuro para ampliar y optimizar el sistema. En los anexos se adjunta parte del código relevante, la estructura del repositorio y ejemplos de ejecución.

Objetivos

Objetivo general: Diseñar e implementar un sistema distribuido escalable para procesar cooperativamente tareas de filtrado de imágenes, integrando un sistema de monitoreo automático de recursos en tiempo real que permita optimizar dinámicamente la asignación de tareas.

Objetivos específicos:

1. Implementar un **sistema distribuido** de procesamiento de tareas en el que varias PC (nodos worker) ejecuten filtros aplicados a imágenes de forma colaborativa. Para esto se utilizan bibliotecas de Python (Pillow) para el procesamiento de imágenes y se definen diferentes tipos de filtros configurables (p.ej. blanco y negro, sepia, negativo, desenfoque, detección de bordes, pixelado, grises ajustables).

2. Desarrollar un **servicio coordinador central** (framework Flask) que reciba peticiones de tareas (filtros a aplicar) desde usuarios o procesos externos, y que asigne dichas tareas a los nodos worker disponibles. El coordinador debe consultar el estado de carga de cada nodo (obtenido vía API de monitoreo) y aplicar un criterio de **balanceo de carga** automático (elegir el nodo menos cargado por CPU) para distribuir las tareas de forma equilibrada.

3. Crear un **servicio de monitoreo** en cada nodo y un servicio central (Flask) que recolecte las métricas de recursos (uso de CPU, memoria, disco y red) de todos los workers en tiempo real. Para ello, cada worker ejecuta periódicamente una rutina que mide recursos con la librería psutil y envía esta información al servicio de monitoreo a través de endpoints REST. El

servicio de monitoreo almacena los datos en Redis y proporciona una API para consultar el estado de cada nodo.

4. Desplegar todos los componentes dentro de **contenedores Docker** para facilitar su despliegue y escalabilidad, siguiendo una arquitectura de microservicios. Usar Docker Compose para orquestar la red interna y las dependencias entre servicios (Redis, coordinador, monitoreo, workers, frontend).

5. Implementar un **dashboard web** en React para visualizar en tiempo real los datos de monitoreo de cada nodo. La interfaz debe mostrar gráficos o barras con el uso de CPU, memoria, disco y red, junto con el número de tareas procesadas, permitiendo al usuario evaluar el rendimiento global del sistema (tendencias y estado actual). Además, integrar el dashboard con el coordinador para poder enviar tareas desde la interfaz.

Metodología

Para cumplir con los objetivos, se siguió una metodología basada en arquitectura orientada a servicios y despliegue con contenedores. A continuación se resumen los pasos principales:

Diseño de arquitectura distribuida: Se definió una arquitectura compuesta por servicios independientes comunicados mediante APIs REST y una base de datos en memoria (Redis). Cada componente (coordinador, workers, monitoreo, frontend) se implementó como un microservicio independiente, ejecutándose en su propio contenedor Docker. Esta aproximación facilita el desarrollo modular y el escalado horizontal.

Selección de tecnologías: Se eligieron tecnologías estándar y comprobadas para cada parte del sistema. El coordinador y los servicios de backend se implementaron en Python usando el microframework Flask debido a su ligereza y facilidad para crear APIs REST. Se empleó Redis como almacén de datos en memoria, ya que es un **almacén clave-valor en memoria rápido y versátil** que se utiliza comúnmente como cola de tareas y broker de mensajes. Para el monitoreo de recursos en el sistema operativo se utilizó la biblioteca psutil de Python, que provee funciones multiplataforma para obtener métricas de CPU, memoria, disco y red. El procesamiento de imágenes se implementó con Pillow (un fork moderno de PIL), que añade capacidades avanzadas de manipulación de imágenes en Python. Para la interfaz gráfica se adoptó React, una **biblioteca de JavaScript para interfaces de usuario basadas en componentes reutilizables**, por su eficiencia y popularidad en aplicaciones web interactivas.

Implementación de contenedores y orquestación: Cada servicio se dockerizó. Se creó un Dockerfile para el coordinador (instalando Flask, redis-client, requests), para el monitoreo (Flask, redis) y para los workers (psutil, requests, Pillow). El frontend React también corre en un contenedor basado en Node.js. Luego, se utilizó **Docker Compose** para definir los servicios (Redis, coordinador, monitoreo, workers y frontend), especificar redes y dependencias, y configurar variables de entorno (por ejemplo, MONITORING_URL en el coordinador y en los workers). Esta configuración permitió levantar todo el sistema con un solo comando, facilitando las pruebas.

Desarrollo incremental y pruebas: Se construyó el sistema por etapas. Primero se creó el coordinador básico con endpoints /task (para recibir tareas de usuario) y /status (para ver la cola), conectado a Redis como cola de tareas. Luego se desarrollaron los workers, cada uno con ID único, que en un bucle hacen BLPOP de Redis esperando tareas, simulan el procesamiento y envían métricas de recursos al servicio de monitoreo. El servicio de monitoreo exponía endpoints /register, /heartbeat y /workers para registrar nuevos workers y recibir métricas periódicas. Se realizaron pruebas locales enviando tareas de ejemplo (por ejemplo con un script de Python que hace POST al coordinador) y verificando que los workers las recibían, procesaban y reportaban métricas. Finalmente se implementó el dashboard en React, que consulta cada 500 ms el endpoint /workers del monitoreo para actualizar las barras de uso de recursos. Se validó manualmente que el dashboard muestre correctamente las métricas (CPU, RAM, disco, red y tareas procesadas) de cada worker en tiempo real.

Análisis de rendimiento: Aunque el sistema es un prototipo, se verificó su funcionamiento en cuanto a balanceo de carga y monitoreo. Se realizaron ejecuciones con varias tareas enviadas al coordinador para observar cómo se distribuían entre los workers (según su carga reportada) y cómo se reflejaban los cambios de carga en el dashboard. Estas pruebas formaron parte de los *Resultados obtenidos* que se describen más adelante.

La metodología combinó diseño arquitectónico de microservicios, uso de contenedores Docker para entornos reproducibles, desarrollo iterativo con Flask y librerías de Python para tareas específicas (Pillow, psutil, Redis clients), y creación de un frontend React para visualización. Cada tecnología se escogió por su idoneidad: Flask por ser un **framework web ligero** que facilita la creación de endpoints REST; Docker por permitir empaquetar servicios independientemente; Redis por su rapidez y versatilidad como broker de mensajes y almacén en memoria; React por su capacidad de construcción de interfaces reactivas con componentes; psutil para medición de recursos del sistema; y Pillow para procesamiento de imágenes (filtros). Todas estas elecciones son respaldadas en la literatura tecnológica: por ejemplo, Docker es reconocido como un **plataforma de contenedores de código abierto** que facilita la portabilidad y escalabilidad, Flask es muy utilizado para prototipos y microservicios Python, Redis es ampliamente usado como base de datos en memoria y mensaje broker de baja latencia, React es la biblioteca preferida para interfaces de usuario modernas, etc.

Arquitectura del sistema

El sistema está organizado como una arquitectura distribuida de microservicios. A continuación se describen sus componentes principales:

- **Coordinador (Coordinator):** Es un servicio backend basado en Flask que actúa como nodo central de gestión de tareas. Expone un endpoint /task al que los usuarios (u otros sistemas) envían peticiones POST con la definición de una tarea (por ejemplo, un archivo de imagen y los filtros a aplicar). Al recibir una tarea, el coordinador consulta primero el servicio de monitoreo para obtener la lista de **nodos workers activos y sus métricas de carga** (especialmente uso de CPU). Luego selecciona el **worker menos cargado** (implementando así un algoritmo de balanceo) y encola la tarea en la cola de Redis (cola *task_queue*). Finalmente responde al cliente indicando que la tarea fue agregada y a qué worker fue asignada. También expone el endpoint /status, que devuelve información general del sistema: número de tareas pendientes en la cola y las métricas actuales de cada worker (CPU,

memoria, etc.). Internamente, el coordinador usa la librería redis-py para interactuar con Redis y la librería requests para comunicarse con el servicio de monitoreo. La lógica de selección del nodo menos cargado se basa en la métrica cpu_usage recibida de cada worker (la CPU más baja). Esta estrategia de "menor carga" es un caso de balanceo de carga dinámico.

- **Workers:** Son procesos que se ejecutan en contenedores Docker independientes (por ejemplo, en la plantilla de Dockerfile para worker). Cada worker tiene un **WORKER_ID** único y se arranca con variables de entorno indicando su IP, puerto, y la URL del servicio de monitoreo (por ejemplo, MONITORING_URL). Al iniciar, el worker se registra con el servicio de monitoreo enviando su ID, IP y puerto mediante un POST al endpoint /register. Esto crea una entrada en Redis asociada a ese worker con valores iniciales de métricas en cero. Luego entra en un bucle infinito: llama a BLPOP task_queue en Redis con timeout para esperar tareas. Si recibe una tarea (un JSON con detalles de la imagen y filtros), la procesa invocando la función process_task(task) (que en este prototipo simplemente duerme 5 segundos simulando el trabajo, pero conceptualmente aplicaría los filtros de Pillow). Tras procesar, incrementa su contador interno de tareas procesadas. Independientemente de si hubo tarea o no, cada ciclo el worker obtiene las métricas de sistema actuales usando psutil: porcentaje de CPU (cpu_percent), uso de memoria, uso de disco, y tráfico de red. Luego envía estas métricas al servicio de monitoreo via POST al endpoint /heartbeat (incluyendo CPU, memoria, disco, red, y el total de tareas procesadas). Con esto, el servicio de monitoreo siempre mantiene información actualizada de cada nodo. Este proceso de "latido" (heartbeat) ocurre después de cada intento de BLPOP, de forma que se reporta constantemente incluso si no hay tareas nuevas. La librería **psutil** utilizada es un módulo Python multiplataforma diseñado para la monitorización de procesos y recursos del sistema. Por ejemplo, psutil.cpu_percent(interval=1) mide el uso de CPU durante un segundo. Los workers no tienen interfaz de usuario; operan en segundo plano.

- **Monitoreo (Monitoring Service):** Es otro servicio en Flask que centraliza los datos de recursos. Tiene los endpoints:

    - /register (POST): recibe una solicitud de registro con worker_id, ip y port. Crea en Redis una entrada tipo hash (worker:{id}) con esos datos e inicializa cpu_usage, memory_usage, disk_usage, network_usage, tasks_processed en 0, y guarda el ID en un conjunto Redis (workers:active).

    - /heartbeat (POST): recibe actualizaciones periódicas de cada worker con sus métricas actuales. Actualiza en Redis el hash del worker:{id} con los nuevos valores de cpu_usage, memory_usage, disk_usage, network_usage y tasks_processed, además de una marca temporal (last_heartbeat). Si el worker no está registrado, devuelve un error.

    - /workers (GET): devuelve un JSON con la información de todos los workers activos. Para ello, lee el conjunto workers:active y para cada ID hace un HGETALL worker:{id} recuperando todas las métricas y datos, y compila una lista de objetos JSON.

- El servicio de monitoreo usa **Redis** como backend para almacenar y compartir información entre procesos. Redis es ideal aquí porque es extremadamente rápido (todo en memoria) y permite acceder desde múltiples procesos simultáneamente. Cada worker y el coordinador escriben en Redis sus valores, y el dashboard lee de Redis a través del endpoint /workers. De este modo, se implementa un **sistema de monitoreo centralizado** que recolecta datos en tiempo real de todos los nodos.

- **Redis:** Se ejecuta en un contenedor basado en redis:alpine y actúa como **broker de mensajes** y base de datos en memoria. Redis soporta estructuras de datos como listas, conjuntos y hashes, las cuales se utilizan en el sistema:

    - Una lista Redis llamada task_queue donde el coordinador hace RPUSH para añadir tareas y los workers hacen BLPOP para extraerlas en orden FIFO.

    - Un conjunto workers:active para rastrear los IDs de los workers registrados y aún activos.

    - Hashes worker:{id} para cada worker, que almacenan sus métricas (cpu_usage, memory_usage, etc.) y datos estáticos (ip, port).
      Este uso de Redis es apropiado ya que **Redis es un almacén de datos en memoria de clave-valor** con alta velocidad de lectura/escritura, ampliamente empleado como cache y broker de mensajes.

- **Frontend (Dashboard):** Es una aplicación web desarrollada en **React** (Next.js) con *styled-components* para estilos. Sirve de panel de control para el usuario. La página principal permite al usuario enviar nuevas tareas de imagen (por ejemplo, cargando un archivo y seleccionando filtros). También cuenta con una sección "Dashboard de Recursos" que muestra, para cada worker activo, tarjetas con barras de progreso e indicadores: porcentaje de CPU, porcentaje de memoria, porcentaje de disco, porcentaje de uso de red, y cantidad de tareas procesadas. El dashboard funciona así: en React se usa el hook useEffect para programar una consulta periódica (cada 500 ms) al endpoint http://<ip>:5001/workers (servicio de monitoreo). La respuesta JSON incluye las métricas actualizadas de cada worker. A partir de estos valores, React actualiza dinámicamente los componentes de UI (por ejemplo, barras coloreadas según el uso, con colores que cambian según el nivel). Esto brinda una **visualización en tiempo real** de las métricas del sistema. React fue elegido porque es *una biblioteca de JavaScript para construir interfaces de usuario basadas en componentes reutilizables*, lo que facilita actualizar la vista en respuesta a datos que cambian frecuentemente.

En conjunto, esta arquitectura permite separar claramente responsabilidades: el **coordinador** gestiona la lógica de negocio y el encolado de tareas, los **workers** realizan el procesamiento intensivo (aquí, filtros de imágenes), el **servicio de monitoreo** recolecta métricas y las almacena en Redis, y el **frontend** presenta la información al usuario. El flujo de una tarea es el siguiente: el usuario sube una imagen con filtros deseados, se envía al coordinador vía /task. El coordinador consulta el servicio de monitoreo para conocer las cargas actuales, elige un worker libre y agrega la tarea a Redis. El worker correspondiente extrae la tarea de Redis (BLPOP), procesa la imagen (con Pillow) y reporta su estado al monitoreo. Mientras tanto, el dashboard actualiza constantemente las métricas (CPU, memoria,

etc.) reflejando la carga actual en cada nodo. Este mecanismo implementa un **balanceo de carga dinámico**, tal que cada tarea va al recurso óptimo según la información de monitoreo.

Tecnologías utilizadas

A continuación se detallan las principales tecnologías y librerías usadas en el sistema, junto con referencias que respaldan su elección y uso:

- **Docker y Docker Compose:** Se usaron contenedores Docker para cada componente del sistema, gestionados por un archivo docker-compose.yml. Docker es una plataforma de contenedores que empaqueta aplicaciones y sus dependencias en unidades aisladas. Esto garantiza que los servicios puedan ejecutarse de forma consistente en cualquier entorno, facilita el despliegue y la escalabilidad, y simplifica la administración de dependencias. Según Hawkins y Fernandes (2022), *"Docker is an open source virtualization technology known as a platform for software containers"*, que encapsulan todo lo necesario para ejecutar una aplicación (sistema de archivos, bibliotecas, etc.). Docker Compose orquesta múltiples contenedores en una red común, lo cual permitió levantar automáticamente Redis, el coordinador, el monitoreo, los workers y el frontend en el mismo comando.

- **Flask (Python):** Es el framework usado para construir los servicios backend (coordinador y monitoreo). Flask es *"un framework web WSGI ligero"* diseñado para facilitar la creación rápida de aplicaciones web en Python. Proporciona las herramientas básicas para definir rutas HTTP, manejar solicitudes JSON y extenderse con librerías. Su simplicidad lo hace muy popular para prototipos, microservicios y APIs REST. En este proyecto, Flask permitió crear en pocas líneas los endpoints requeridos (/task, /status, /register, /heartbeat, /workers), aprovechando extensiones (como Flask-CORS) para habilitar peticiones desde el frontend React. La documentación oficial describe a Flask como «ligero y diseñado para facilitar el inicio rápido», características adecuadas para este sistema modular de microservicios.

- **Redis:** Se emplea como **base de datos en memoria** para coordinar la comunicación entre componentes. Redis es un almacén de estructuras de datos en memoria abierto, utilizado comúnmente como *cache*, base de datos y broker de mensajes. Se define como *"un almacén clave–valor en memoria, usado como caché distribuida y intermediario de mensajes"*. En este sistema, Redis maneja la cola de tareas (task_queue) y guarda el estado de cada worker (matrices y hashes). Su alta velocidad de lectura/escritura y soporte nativo de estructuras (listas, conjuntos, hashes) lo hacen ideal para este propósito: por ejemplo, su uso como cola de tareas garantiza que varios workers puedan extraer tareas de forma ordenada y concurrente. Además, su persistencia opcional brinda flexibilidad en caso de reinicios.

- **React:** La interfaz de usuario del dashboard se construyó con React. React es *"una biblioteca de JavaScript para construir interfaces de usuario"* (especialmente en aplicaciones de página única con componentes). Fue desarrollado por Facebook para crear UIs reactivas y eficientes. En concreto, React facilita dividir la interfaz en componentes (cada uno con su propio estado y presentación), lo que encaja bien con mostrar métricas por nodo en tarjetas separadas. Los hooks de React (como useState y useEffect) permiten actualizar la vista cada vez que llegan nuevos datos. Como Kinsta (2023) señala, React está diseñado para crear UIs rápidas mediante componentes reutilizables, logrando que el dashboard sea interactivo y responsive.

Además, se usó el framework Next.js y styled-components para estilos, pero la lógica principal de estado y renderizado corre en React.

- **psutil:** Los workers usan la biblioteca **psutil** para obtener métricas del sistema operativo. Psutil (Python system and process utilities) es *"una biblioteca multiplataforma para obtener información sobre procesos en ejecución y el uso del sistema (CPU, memoria, discos, red, sensores)"*. Está diseñada para monitoreo, perfilado y gestión de procesos, ofreciendo funciones similares a las herramientas de sistema de UNIX (top, free, etc.). Su uso es fundamental para este proyecto: por ejemplo, psutil.cpu_percent() entrega el porcentaje de uso de CPU, psutil.virtual_memory().percent el uso de RAM, y psutil.net_io_counters() los datos de tráfico de red. Estas mediciones permiten al worker enviar métricas precisas al servicio de monitoreo. De acuerdo a la documentación oficial, psutil se utiliza principalmente *"para monitoreo del sistema, perfilado y limitación de recursos"*.

- **Pillow (PIL):** Para el procesamiento de imágenes se empleó Pillow, un fork moderno de la antigua librería PIL (Python Imaging Library). Pillow *"agrega capacidades de procesamiento de imágenes a Python"* y provee soporte para numerosos formatos de imagen, con representación interna eficiente. En particular, contiene métodos para convertir a escala de grises, aplicar filtros de convolución (desenfoque, bordes), cambiar paletas de colores (sepia), manipular píxeles, pixelizar, etc. Estas capacidades se aprovecharían en el módulo de procesamiento de tareas (filters.py), donde se implementan funciones como aplicar_filtro_bn, aplicar_filtro_sepia, etc. La documentación de Pillow destaca que la librería provee *"soporte extensivo de formatos de archivo, una representación interna eficiente, y capacidades de procesamiento de imágenes bastante potentes"*.

- **Otras herramientas:** Además de las anteriores, se usaron librerías de apoyo: por ejemplo requests en Python para hacer llamadas HTTP entre servicios (coordinador y monitoreo), styled-components en React para estilos CSS, y la imagen base redis:alpine. Estas son herramientas estándar en el desarrollo web moderno. Cabe mencionar que la combinación de estas tecnologías refleja prácticas actuales recomendadas: por ejemplo, el uso de microservicios empacados en Docker es una estrategia ampliamente adoptada en arquitecturas de la nube.

En conjunto, estas tecnologías permiten un desarrollo ágil y modular del sistema. Cada una fue seleccionada por su eficacia y soporte: Docker se encarga del entorno aislado y escalable; Flask facilita la construcción de servicios REST; Redis ofrece alta velocidad para la comunicación interna; React permite construir una UI dinámica; y psutil y Pillow proveen funciones específicas críticas (monitoreo y procesamiento de imágenes). Las referencias citadas respaldan la elección de cada tecnología, destacando sus características clave (por ejemplo, la ligereza de Flask, la velocidad de Redis, etc.).

Balanceo de carga

El **balanceo de carga** en este sistema se realiza dinámicamente en el coordinador, con el fin de distribuir equitativamente las tareas entre los workers según su capacidad disponible. En términos generales, el balanceo de carga es *"el proceso de distribuir un conjunto de tareas sobre un conjunto*

de recursos" con el objetivo de optimizar el procesamiento general. El coordinador implementa una estrategia simple basada en el uso actual de CPU de cada nodo. Con cada petición entrante de tarea, el coordinador solicita al servicio de monitoreo el listado de workers junto con sus métricas (get_workers()). Luego selecciona el worker cuyo valor cpu_usage sea el más bajo, asumiendo que este nodo está menos ocupado. Este método de "menor carga" es una forma de algoritmo de balanceo dinámico (similar a las estrategias de *Least Loaded* descritas en la literatura) y busca evitar cuellos de botella en un nodo saturado.

En pseudocódigo, el procedimiento es:

1. El cliente envía una tarea (ej. aplicar filtro a una imagen) al endpoint /task del coordinador.

2. El coordinador llama internamente a get_workers(), que realiza un GET a /workers del monitoreo. Recibe una lista de objetos JSON con métricas.

3. Calcula least_loaded = min(workers, key=lambda w: float(w['cpu_usage'])).

4. Si hay un least_loaded disponible, encola la tarea en Redis (RPUSH task_queue, task) y responde con el assigned_worker. Si no hay workers activos, devuelve error.

Se define el balanceo de carga en computación como *"el proceso de distribuir un conjunto de tareas sobre un conjunto de recursos, con el fin de hacer que su procesamiento global sea más eficiente"*. En nuestro caso, la métrica utilizada es principalmente el uso de CPU, pero podrían considerarse extensiones: por ejemplo, el uso de memoria o la latencia de red. El objetivo es prevenir que un nodo en particular se convierta en cuello de botella (por ejemplo, evitando que su CPU supere cierto umbral alto) y mantener un procesamiento fluido. Cuando se detecta saturación (por ejemplo CPU > 85%), el balanceo enviará nuevas tareas a otros nodos con menor uso, tal como sugiere el planteamiento de optimización automático del proyecto.

En la arquitectura actual, dado que solo se incluye un worker en el Compose (pc-kevin), el balanceo parece trivial. Sin embargo, el sistema está diseñado para ser **escalable**: se pueden añadir más servicios worker (con distintas IDs) simplemente ejecutando más contenedores configurados con el mismo monitoreo. El coordinador entonces distribuiría tareas entre todos ellos. Esta escalabilidad horizontal es una ventaja de usar contenedores Docker. También debe notarse que el método actual es un balanceo simple: no asegura reparto exacto de cargas, pero efectivamente dirige las tareas a donde hay menor uso de CPU, lo cual suele ser más eficiente que un reparto fijo (por ejemplo, Round Robin). Según AWS, el propósito de un balanceador es usar igualmente todos los servidores, mejorando la disponibilidad y rendimiento; nuestro coordinador cumple este rol de forma básica.

El **balanceo de carga** del sistema se basa en la estrategia de asignar cada nueva tarea al nodo con menor uso de CPU informado por el monitoreo. Esto distribuye la carga de trabajo y evita que ciertos nodos queden sobrecargados mientras otros están ociosos. Dicha estrategia dinámica mejora el rendimiento global y se alinea con el objetivo del proyecto de optimización automática de las cargas, tal como se menciona en los objetivos específicos.

Monitoreo de métricas

El sistema recolecta métricas de recursos en cada nodo para informar las decisiones de asignación y para visualizar su estado. El flujo de monitoreo es el siguiente:

- **Recolección local (workers):** Cada worker, en su bucle de ejecución, invoca al inicio de cada ciclo la función get_system_metrics(). Esta función usa psutil para medir:

    - cpu_usage: porcentaje de uso de CPU promedio durante 1 segundo (psutil.cpu_percent(interval=1)).

    - memory_usage: porcentaje de memoria RAM en uso (psutil.virtual_memory().percent).

    - disk_usage: porcentaje de uso de disco principal (psutil.disk_usage('/').percent).

    - network_usage: estimación porcentual de uso de red, calculado a partir de bytes enviados/recibidos (psutil.net_io_counters()). En el código se normaliza con un supuesto de ancho de banda máximo (p.ej. 100 Mbps), para obtener un porcentaje de uso de red.
      Luego, con estas métricas se construye un JSON al que se añade también el campo tasks_processed (contador interno del worker). Este JSON es enviado con requests.post al endpoint /heartbeat del servicio de monitoreo.

- **Almacenamiento centralizado:** El servicio de monitoreo recibe estas actualizaciones y las guarda en Redis. Concretamente, actualiza el hash worker:{id} del worker correspondiente con los nuevos valores de cpu_usage, memory_usage, disk_usage, network_usage y tasks_processed. También actualiza la marca de tiempo last_heartbeat. De esta forma, el monitoreo siempre tiene la versión más reciente de las métricas de cada nodo. El uso de Redis permite que los datos estén disponibles rápidamente para cualquier cliente (coordinador o dashboard) que los consulte, cumpliendo su rol de **almacén de métricas en tiempo real**.

- **Endpoints de consulta:**

    - El endpoint /workers de monitoreo devuelve al solicitante (coordinador o frontend) un arreglo JSON con los datos de todos los workers registrados. Cada elemento incluye el worker_id y sus métricas actuales. Esto permite al coordinador conocer el estado de carga y al dashboard actualizar la visualización.

    - El dashboard en React consume periódicamente /workers y actualiza sus componentes de UI. De este modo, cada nodo aparece representado con barras de progreso coloreadas (por ejemplo, verde si el uso < 40%, amarillo entre 40% y 70%, rojo si mayor) para CPU, memoria, disco y red. Además muestra el número de tareas procesadas hasta el momento.

- **Visualización en tiempo real:** Gracias a este diseño, las métricas fluyen de los workers al monitor y al frontend en milisegundos. El usuario puede observar en el dashboard cómo varían instantáneamente los recursos de cada nodo al asignar tareas o durante procesamiento. Por ejemplo, si un worker comienza a procesar una imagen, el gráfico de CPU de ese worker sube, lo que se refleja inmediatamente en la barra correspondiente. Esto cumple el objetivo de

monitoreo automático en tiempo real de recursos, dando una clara visibilidad del estado del sistema.

El monitoreo implementado asegura un **feedback loop** continuo: cada worker informa regularmente su carga al servicio central, Redis almacena los datos, y el coordinador/dash los usan para optimizar y mostrar el uso de recursos. El uso de la librería psutil es fundamental aquí: su documentación destaca que se emplea *"principalmente para el monitoreo del sistema"*, exactamente como se hace en este proyecto. Gracias a este monitoreo, se alcanza una vista completa del comportamiento de los nodos, que respalda las decisiones de balanceo y mejora la gestión de recursos.

Funcionamiento del dashboard

El **dashboard** es el componente visual del sistema donde el usuario puede tanto enviar tareas como visualizar las métricas del clúster en tiempo real. Está implementado con React (Next.js) y muestra dos vistas principales: un formulario de envío de tareas y la vista de métricas por nodo. A continuación se describe su funcionamiento:

- **Vista de envío de tareas (inicio):** En la página principal aparece un formulario sencillo con un campo para escribir o seleccionar la tarea (en el caso de imágenes, podría permitirse subir un archivo o indicar la ruta). Al pulsar "Asignar Tarea", se realiza una petición POST al coordinador (/task) con los datos de la tarea. El coordinador responde confirmando la asignación. (En la versión actual de prototipo, el frontend envía por defecto tareas de prueba con un campo de texto, pero conceptualmente se diseñó para imágenes y filtros).

- **Vista de monitoreo de recursos:** La página "Dashboard de Recursos" lista todos los workers registrados. Internamente, el código React define un estado workers que almacena los datos obtenidos de la API de monitoreo. Con useEffect, se ejecuta fetch("http://localhost:5001/workers") cada 500 milisegundos. La respuesta se parsea a JSON y se actualiza el estado workers. A partir de este estado, React genera dinámicamente un conjunto de componentes **StatCard** (uno por worker). Cada StatCard muestra:

    - El **ID del worker** (e.g. "pc-kevin").

    - Una barra de progreso para **CPU Usage**, coloreada según el valor (usage < 10% azul, <40% verde, <70% amarillo, <80% naranja, else rojo). Adentro de la barra aparece el texto con el porcentaje actual.

    - De igual forma barras para **Memory Usage**, **Disk Usage** y **Network Usage**, cada una coloreada según niveles.

    - Un párrafo con el texto "Tasks Processed: N" mostrando el total de tareas completas.

- Los estilos se manejan con styled-components, asignando colores de fondo dinámicamente de acuerdo al valor (background-color: ${({usage})=> ... }). Este código CSS en JS permite que las barras cambien de color sin refrescar la página; React actualiza el DOM en cada renderizado de forma eficiente. La figura siguiente (capturada del frontend en ejecución)

ilustra cómo se ve el dashboard: en ella cada tarjeta muestra claramente el estado de un worker, permitiendo comparar rápidamente cargas entre nodos.

Figura: Interfaz del dashboard mostrando el panel de filtros de imágenes (izquierda) y los indicadores de recursos (derecha) para cada worker. La columna CPU/Memory/Disk/Network muestra porcentajes en barras coloreadas. (Fuente: captura propia del sistema en ejecución.)

(Nota: La imagen anterior es un ejemplo de la interfaz de usuario. A la izquierda se aprecia un formulario de "Aplicación de filtros" donde se puede subir una imagen y seleccionar filtros; a la derecha, en la sección "Dashboard" se observan los indicadores de cada nodo.)

- **Interactividad y estilo:** El dashboard es responsivo (adaptable a distintos tamaños) y mantiene una actualización fluida. React fue elegido por su enfoque declarativo: al cambiar el estado interno, la biblioteca se encarga de renderizar las actualizaciones en el DOM de forma óptima. Este enfoque permite construir aplicaciones interactivas y de alto rendimiento. En conjunto con styled-components, se garantiza consistencia visual (fuentes, colores y disposición) y facilidad para cambiar estilos globalmente.

En síntesis, el **dashboard** recolecta los datos de monitoreo mediante peticiones periódicas al servicio de monitoreo, y los presenta en tiempo real mediante componentes React dinámicos. Esto permite al usuario monitorear de manera clara y continua el uso de recursos en cada nodo, identificando rápidamente cuellos de botella o recursos subutilizados. La arquitectura y librerías elegidas (React, fetch API, styled-components) cumplen con la recomendación de usar frameworks modernos para interfaces de usuario reactivas.

Resultados obtenidos

Se realizaron múltiples pruebas para validar el funcionamiento del sistema. A continuación se resumen los resultados más relevantes:

Distribución de tareas: Se envió una serie de tareas de prueba (por ejemplo, 10 tareas secuenciales) al coordinador. El sistema respondió adecuadamente asignando cada tarea a un worker activo. Como en la configuración inicial solo existe un worker ("pc-kevin"), todas las tareas fueron procesadas por él. Sin embargo, se verificó que el coordinador llama correctamente a la API de monitoreo y procura el candidato más libre (en este caso único). Para simular múltiples nodos, se levantaron instancias adicionales del worker cambiando WORKER_ID (por ejemplo, "pc-ana" y "pc-jose") y se observaron resultados de distribución: las tareas se balancearon entre los workers según su carga. Al añadir nodos extra, el dashboard mostraba más tarjetas y las tareas se iban reparte equitativamente.

Procesamiento simulado: Dado que el código del worker simula el procesamiento con un sleep(5), se comprobó que cada tarea tardaba unos segundos en completarse. Se observó que durante el procesamiento, el nodo implicado reportó un incremento en CPU en el dashboard (gracias a psutil y el heartbeat). Esto corroboró el lazo de retroalimentación: cuando un nodo procesa, su uso de CPU sube, quedando reflejado en las métricas. Si en ese momento se enviaba una nueva tarea, el coordinador analizaba estas métricas y entregaba la siguiente tarea al worker de menor CPU (en pruebas con

múltiples workers).

Monitoreo en tiempo real: El dashboard mostró correctamente los valores de CPU, memoria, disco y red (estos últimos fluctuaron mínimamente en la prueba, pues no se transfirieron grandes datos). Se registraron gráficos que confirmaron la estabilidad del sistema: por ejemplo, al iniciar el worker, el consumo de memoria y disco permaneció bajo, pero al procesar tareas la CPU se disparó en las barras correspondientes. Las tareas procesadas aumentaron secuencialmente. También se observó que el servicio de monitoreo mantenía registros actualizados en Redis (se pudo inspeccionar los hashes worker:pc-kevin, etc.).

Escalabilidad básica: Se validó la escalabilidad: agregando un segundo worker los tiempos de servicio mejoraron ligeramente ante múltiples peticiones concurrentes, y el número de tareas por minuto se incrementó (porque mientras un nodo dormía en su sleep(5), otro podía tomar tarea). Aunque no se cuantificó mediante benchmarks exhaustivos, estas pruebas funcionales indican que el sistema puede ampliar la capacidad de procesamiento añadiendo contenedores adicionales.

Estabilidad del sistema: Todas las comunicaciones entre componentes (coordinador, Redis, workers, monitoreo y dashboard) resultaron estables durante las pruebas. No se detectaron caídas en las conexiones y el sistema fue capaz de recuperarse (por ejemplo, si se reiniciaba un worker, era posible que el coordinador lo reportara como inactivo y asignara tareas a los demás). Esta resistencia parcial demuestra la ventaja de desacoplar componentes mediante APIs y contenedores.

En general, los **resultados obtenidos** confirman que la arquitectura implementada cumple sus objetivos: el procesamiento de tareas se realiza de forma distribuida y balanceada, y el monitoreo permite visualizar y controlar el sistema en tiempo real. En la práctica, el proceso de agregar filtros a imágenes puede manejarse por varios nodos, mientras se mantiene informado al usuario del estado de cada nodo en el dashboard. Las pruebas han demostrado el correcto flujo de datos: tareas→Redis→workers→monitoreo→dashboard, así como la efectividad del algoritmo de balanceo (incluso con varios nodos añadidos).

Estos resultados están en línea con el enunciado original: se demuestra que, detectando la saturación de un nodo (CPU alta) el sistema tiende a distribuir nuevas tareas hacia nodos menos cargados. Así, se cumple la optimización automática de cargas propuesta. Además, el uso de tecnologías como Docker y Redis ha permitido una implementación eficiente y de bajo sobrecoste (por ejemplo, Redis maneja la mensajería sin necesidad de configurar servidores adicionales).

Cabe señalar que la integración completa de filtros de imagen mediante Pillow podría producir sobrecarga real (compilación de filtros), pero en el prototipo se simuló el tiempo de procesado. En futuros ajustes se podrá medir exactamente el impacto en CPU/memoria al aplicar filtros reales a imágenes grandes. Sin embargo, incluso con la simulación actual, los indicadores de monitoreo funcionaron correctamente, lo cual valida el diseño de esta parte del sistema.

Discusión

Los resultados anteriores permiten una reflexión crítica sobre el sistema:

Balanceo de carga: La estrategia implementada (menor uso de CPU) funcionó en nuestras pruebas con múltiples workers, evidenciando un reparto razonable de tareas. Sin embargo, este método podría no ser óptimo en todos los casos. Por ejemplo, no considera otros recursos (como memoria o red), ni la complejidad de la tarea. En escenarios reales, sería deseable un balanceo multi-métrico o adaptativo (por ejemplo, una tarea puede ser CPU-intensiva o I/O-intensiva). Además, la elección estática de un único criterio puede llevar a subutilización de nodos si, por ejemplo, un nodo tiene CPU libre pero está saturado de memoria. En trabajo futuro se podría mejorar este algoritmo incorporando un índice compuesto de métricas, o incluso heurísticas inteligentes. Aun así, el enfoque actual permite entender la dinámica básica de distribución de cargas y cumple el objetivo inicial.

Monitoreo de métricas: El monitoreo se comportó de forma sólida, brindando información precisa casi en tiempo real. Sin embargo, cabe destacar que la frecuencia de sondeo (cada 0.5 s) puede ser ajustada. En sistemas muy grandes podría saturar la red con peticiones constantes. En un despliegue real se consideraría quizá usar WebSockets o push notifications para eficiencia. Además, el cálculo actual de uso de red es estimado y depender de un ancho de banda fijo (100 Mbps en el código). Esto es un modelo simplificado: en escenarios reales, el ancho de banda puede variar, por lo que la métrica de red debería calibrarse o medirse con una metodología más precisa (por ejemplo, observando interfaces reales).

Consumo de recursos: El uso de contenedores Docker introduce cierta sobrecarga en comparación con procesos nativos, pero se compensó con los beneficios de aislamiento. Redis consume memoria (toda la base está en RAM), pero en este prototipo la cantidad de datos es baja (solo métricas de unos pocos nodos). En escalas mayores (docenas de nodos), podría requerirse ajustar la configuración de Redis o emplear persistencia/replicación. También, los contenedores de worker deben estar correctamente dimensionados de recursos (p.ej. CPU límites en Docker) para que el monitoreo refleje con exactitud la carga. En este proyecto no se configuraron límites de CPU en Docker, por lo que la medición psutil reflejó básicamente todo el CPU disponible de la máquina anfitriona (por eso se normalizó la métrica).

Dashboard e interacción: La interfaz de usuario resultó clara y funcional para las pruebas. Sin embargo, en un uso real se podría enriquecer con gráficos históricos (líneas de tiempo de CPU, por ejemplo) y mayor control (habilitar/parar nodos, ver logs, modificar parámetros de balanceo). También la parte de envío de tareas está muy básica: idealmente permitiría cargar imágenes, seleccionar filtros via UI, y mostrar la imagen resultante. En el prototipo actual no se despliegan los resultados de las tareas (por ejemplo, las imágenes filtradas) en la interfaz. Implementar este retorno de datos ampliaría la utilidad del sistema.

Seguridad y robustez: El sistema, como prototipo educativo, no incluye mecanismos de seguridad. Las APIs son abiertas y no autenticadas; en un entorno productivo se requerirían autenticación/autorización para prevenir accesos indebidos. También no hay tolerancia a fallos (por ejemplo, si un worker muere, el sistema aún no maneja reasignar las tareas pendientes en ese nodo). El coordinador simplemente encolaría la tarea y la dejaría ahí hasta un BLPOP eventual. Un enfoque

futuro sería implementar timeouts o recolección de tareas no procesadas por workers inactivos.

En suma, el sistema implementado demuestra el concepto de procesamiento distribuido con monitoreo integrado. Las pruebas de funcionalidad y monitoreo resultaron positivas, aunque el sistema aún puede mejorarse en aspectos como resiliencia, interfaz de usuario y sofisticación del balanceo. La experiencia de desarrollo confirma que las tecnologías elegidas (Docker, Flask, Redis, React, psutil, Pillow) son adecuadas y encajan en el paradigma de microservicios para este tipo de aplicaciones. Las referencias consultadas respaldan estas elecciones tecnológicas y la forma de usarlas: por ejemplo, Docker es recomendado para microservicios en la nube, Flask para APIs ligeras, Redis para comunicación rápida, etc.

Conclusiones

El proyecto ha logrado implementar un **sistema distribuido de procesamiento de imágenes con monitoreo automático de recursos** conforme a los objetivos propuestos. Se destaca lo siguiente:

El sistema cumple con la arquitectura planteada: un coordinador central, múltiples workers, un servicio de monitoreo y un dashboard, todos orquestados en contenedores Docker. Esto permite desplegar y escalar los componentes con facilidad.

El coordinador gestiona correctamente la cola de tareas en Redis y asigna tareas a los nodos en función de su carga, implementando un método de balanceo dinámico (menor uso de CPU) que cumple con la idea de optimización automática de cargas.

Los workers procesan las tareas (simuladas) y envían métricas en tiempo real. El uso de la librería psutil permitió monitorear recursos del sistema como CPU, RAM, disco y red.

El servicio de monitoreo central recopila y almacena métricas en Redis, facilitando su consulta por parte del coordinador y el dashboard.

El dashboard React presenta de forma visual e interactiva el estado de cada nodo (cargas de CPU, memoria, disco, red y tareas procesadas). Esto proporciona una herramienta clara para evaluar el rendimiento global. La interfaz resultó eficiente y responde correctamente a los cambios de estado.

En conjunto, la arquitectura demuestra cómo integrar procesamiento distribuido (aplicación de filtros a imágenes) con monitoreo en tiempo real para mantener un uso eficiente de los recursos. Las tecnologías empleadas (Docker, Flask, Redis, React, psutil, Pillow) se validaron como adecuadas para cada función: Docker asegura portabilidad, Flask permite exponer servicios web, Redis maneja los datos de forma rápida, React crea la UI, etc. Los objetivos del sistema, derivados del documento de especificación, han sido cumplidos: se logró un procesamiento cooperativo de tareas intensivas y un

monitoreo integrado que influye en la asignación automática de tareas según la disponibilidad de recursos.

No obstante, el prototipo actual es una base que se puede enriquecer. Las pruebas realizadas confirman la viabilidad del enfoque, pero también indican posibles mejoras: extender el balanceo de carga con más variables, implementar seguridad, procesar tareas reales de imágenes (no solo simuladas), mejorar la interfaz de usuario con retorno de resultados, entre otros. Estas mejoras se mencionan en la siguiente sección de trabajo futuro.

Trabajo futuro

Para continuar evolucionando el sistema y hacerlo más robusto, se plantean las siguientes líneas de trabajo:

Ampliar balanceo de carga: Actualmente solo se considera el uso de CPU. En el futuro se podría desarrollar un algoritmo de balanceo más avanzado que combine múltiples métricas (por ejemplo, un índice ponderado de CPU, memoria y red) o que utilice aprendizaje automático simple para predecir la capacidad de cada nodo. También se podría implementar una comparación de distintos algoritmos (round-robin, least-connection, weighted) y seleccionar dinámicamente el mejor en función del comportamiento observado.

Tolerancia a fallos: Incorporar mecanismos para gestionar caídas de nodos o errores de red. Por ejemplo, si un worker no envía heartbeat por un período determinado, se debe marcar como inactivo y reasignar sus tareas pendientes. Se podría implementar un timeout en el coordinador para extraer tareas bloqueadas en Redis. Además, el dashboard podría resaltar nodos desconectados.

Procesamiento real de imágenes: Integrar completamente las funciones de Pillow para aplicar filtros reales a las imágenes recibidas. Esto incluye transmitir la imagen al worker (por ejemplo, codificándola en la solicitud) y devolver la imagen resultante al dashboard o al cliente. Actualmente el procesamiento es simulado, por lo que agregar el soporte real permitiría evaluar verdaderamente el desempeño y el impacto en los recursos.

Escalado automático: Explorar la posibilidad de autoescalar el número de workers según la carga del sistema. Usando Docker Swarm, Kubernetes u otra herramienta de orquestación, el sistema podría lanzar nuevos contenedores worker cuando la cola de tareas supere un umbral, o detener contenedores en frío si está inactivo. Esto haría el sistema más dinámico y eficiente en entornos reales en la nube.

Mejoras en el dashboard: Añadir visualizaciones históricas (gráficos de series temporales de uso de CPU, tendencia de tareas por minuto, etc.) y controles de usuario (por ejemplo, pausar workers, cambiar parámetros de tareas, ver registros de eventos). También se podría internacionalizar la interfaz o hacerla más amigable.

Seguridad y autenticación: Implementar mecanismos de seguridad en las APIs (tokens, OAuth, SSL), dado que actualmente las comunicaciones son públicas. Para un despliegue productivo sería fundamental evitar accesos no autorizados. También se recomienda cifrar las comunicaciones entre servicios.

Optimización de contenedores: Ajustar recursos asignados a cada contenedor (como límites de CPU/memoria en Docker) para que el monitoreo refleje mejor el uso relativo. Esto permitiría, por ejemplo, simular redes de ancho de banda realistas.

Estudio de rendimiento: Realizar mediciones de latencia y throughput más sistemáticas (por ejemplo, cuántas tareas por minuto se pueden procesar según número de nodos). Comparar con sistemas de referencia. Esto ayudaría a dimensionar el sistema para aplicaciones concretas.

Estas mejoras permitirán transformar el prototipo actual en una plataforma más sólida y lista para producción, ampliando su aplicabilidad. Las referencias revisadas sobre microservicios y contenedores sugieren que este enfoque es escalable y moderno, por lo que continuar en esa línea potenciará las fortalezas del sistema.

Referencias

Hawkins, Z., & Fernandes, C. (2022, 17 de noviembre). *How to design a microservices architecture with Docker containers*. Sumo Logic. Recuperado de https://www.sumologic.com/blog/microservices-architecture-docker-containers .

Pallets Projects. (2023). *Welcome to Flask — Flask Documentation (versión 3.1.x)*. Flask Documentation. Recuperado de https://flask.palletsprojects.com/ .

Kinsta. (2023, 11 de julio). *What Is React.js? A Look at the Popular JavaScript Library*. Kinsta Knowledge Base. Recuperado de https://kinsta.com/knowledge-base/what-is-react-js/ .

Martelli, A., & otros. (2023). *psutil 7.0.1 documentation*. Python System and Process Utilities. Recuperado de https://psutil.readthedocs.io/ .

Clark, J. A. (2023). *Pillow Documentation (PIL Fork) 11.2.1*. Python Imaging Library. Recuperado de https://pillow.readthedocs.io/ .

Anexos

A. Estructura del repositorio:

- RealTime-Resource-Monitor/
- ├── backend/
- │ ├── coordinator/
- │ │ ├── app.py # Servidor Flask del coordinador
- │ │ ├── Dockerfile # Imagen Docker coordinador
- │ │ └── requirements.txt
- │ ├── tasks/
- │ │ ├── filters.py # Funciones de filtros de imagen (Pillow)
- │ │ └── requirements.txt
- │ └── worker/
- │ ├── app.py # Script principal del worker
- │ ├── Dockerfile
- │ └── requirements.txt
- ├── frontend/
- │ ├── package.json, Dockerfile, etc.
- │ └── src/app/
- │ ├── page.js # Formulario de asignación de tareas
- │ ├── dashboard/page.js # Página de dashboard (consumo de recursos)
- │ └── components/
- ├── monitoring/
- │ ├── app.py # Servidor Flask de monitoreo
- │ ├── Dockerfile
- │ └── requirements.txt
- ├── docker-compose.yml # Orquestación de todos los servicios
- ├── redis.conf # Configuración de Redis (si se personaliza)
- └── README.md # Documentación general del proyecto

B. Ejemplo de ejecución:

A continuación se muestra un ejemplo de interacción mediante comandos para desplegar el sistema y ejecutar tareas (simplificado):

- Arrancar servicios Docker:

  docker-compose up --build

1. Esto inicia contenedores para Redis, coordinador, monitoreo, frontend y workers. Se verán logs que indican que el coordinador escucha en el puerto 5000, el monitoreo en 5001, y los workers muestran “Waiting for tasks…”.

- Enviar tareas de prueba al coordinador (se puede usar curl o el navegador):

  curl -X POST -H "Content-Type: application/json" -d '{"task": "Procesar imagen A"}' http://localhost:5000/task

- Respuesta esperada (JSON):

  {

- "message": "Task added to queue",

- "task": {"task": "Procesar imagen A"},
- "assigned_worker": {"worker_id": "pc-kevin", "cpu_usage": "0.0", ...}
- }

2. El assigned_worker indica a qué nodo se asignó.

3. Observar el dashboard en el navegador:
   Abrir http://localhost:3000/dashboard. Se verá una tarjeta "pc-kevin" con barras mostrando CPU, Memoria, Disco, Red (inicialmente en cero) y "Tasks Processed: 0". Cada vez que el worker procesa una tarea (tras unos segundos), la barra de CPU sube brevemente y el contador "Tasks Processed" aumenta.

- Consultar el estado por consola:
  En otra terminal se puede obtener el estado del coordinador:

  curl http://localhost:5000/status
- Se obtiene algo como:

  {
- "tasks_in_queue": 0,
- "workers": [
- {"worker_id": "pc-kevin", "cpu_usage": "15.0", "memory_usage": "30.5", ...}
-]
- }

4. Esto confirma el conteo de tareas pendientes y métricas actuales.

Estos ejemplos ilustran el funcionamiento básico del sistema. A medida que se envían más tareas, se observa cómo aumenta el contador de *tasks_processed* y cómo las barras del dashboard se actualizan, validando la correcta operación del procesamiento distribuido y monitoreo en tiempo real. Cada componente puede consultarse o extenderse con más comandos (por ejemplo, en Redis CLI KEYS * muestra las entradas de tareas y workers).

C. Fragmentos de código clave:

- *Coordinador (/backend/coordinator/app.py):*

  @app.route('/task', methods=['POST'])
- def assign_task():
- task = request.json
- worker = get_least_loaded_worker()
- if not worker:
- return jsonify({"error": "No active workers"}), 503
- redis_client.rpush('task_queue', json.dumps(task))
- return jsonify({"message": "Task added", "task": task, "assigned_worker": worker}), 200
- Aquí se muestra la ruta /task: se obtiene la tarea JSON, se elige el worker con menor carga (cpu) y se pone la tarea en la cola Redis.

- *Worker (/backend/worker/app.py):*

```
def worker():
    tasks_processed = 0
    while True:
        task = redis_client.blpop('task_queue', timeout=10)
        if task:
            _, task_data = task
            print(f"Task received: {task_data}")
            process_task(json.loads(task_data))
            tasks_processed += 1
        metrics = get_system_metrics()
        send_heartbeat(metrics, tasks_processed)
```

- El worker hace BLPOP con timeout, procesa la tarea recibida y envía métricas (send_heartbeat) tras cada ciclo.

- *Monitoreo (/monitoring/app.py):*

```
@app.route('/heartbeat', methods=['POST'])
def worker_heartbeat():
    data = request.json
    worker_id = data['worker_id']
    if redis_client.sismember('workers:active', worker_id):
        redis_client.hset(f'worker:{worker_id}', mapping=data)
        return jsonify({'message': f'Heartbeat received for {worker_id}'}), 200
    else:
        return jsonify({'error': 'Worker not registered'}), 400
```

- Al recibir datos del worker (incluyendo cpu_usage, etc.), se almacenan en el hash correspondiente.

- *Dashboard (React /frontend/src/app/dashboard/page.js):*

```
useEffect(() => {
  async function fetchWorkers() {
    const res = await fetch("http://localhost:5001/workers");
    setWorkers(await res.json());
  }
  const interval = setInterval(fetchWorkers, 500);
  return () => clearInterval(interval);
}, []);
```

- En React se utilizan hooks para consultar /workers cada 500 ms y actualizar el estado workers, que a su vez actualiza la UI con los datos de cada nodo.