udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA Y CIENCIAS

Proyecto Sistemas Distribuidos: Plataforma de Análisis de Tráfico en Región Metropolitana - Parte 1

Sección 3

Camilo Antonio Rios Tenderini—Camilo.rios1@mail.udp.cl

Renato Antonio Yáñez Riveros—Renato.yanez2@mail.udp.cl

Fecha: Abril - 2025

Profesor:
Nicolás Hidalgo

Índice

1. Introducción **1**

2. Arquitectura del sistema **2**

3. Implementación de módulos **3**
3.1. Uso de Docker 3
3.2. Web Scrapper 4
3.3. Almacenamiento 5
3.4. Visualización de almacenamiento 5
3.5. JSON Loader 6
3.6. Caché 6
3.7. Visualización de caché 6
3.8. Servidor Web 7
3.9. Generador de tráfico 8

4. Pruebas y resultados **9**
4.1. Distribución de frecuencias 9
4.1.1. Distribución Zipf 9
4.1.2. Distribución Normal 11
4.2. Distribución de frecuencias y tasa de arribo 12

5. Análisis y discusión. **14**
5.1. Generador de tráfico 14
5.2. Almacenamiento 14
5.3. Métricas 15
5.4. Pruebas de rendimiento 15

6. Conclusión **17**

1. Introducción

El presente informe describe una plataforma de análisis de tráfico desarrollada con el objetivo de estudiar el comportamiento de sistemas distribuidos, en particular la comparación de rendimientos al utilizar un sistema de caché frente a accesos directos a una base de datos en un entorno con un volumen considerable de solicitudes.

Se detalla todo el proceso de desarrollo de la plataforma, desde la recopilación de datos, el diseño de una arquitectura modular, hasta las fases de prueba y validación del sistema.

Todo el trabajo realizado está disponible en el repositorio de GitHub, incluyendo la documentación necesaria para desplegar el proyecto y reproducir las pruebas efectuadas.

2. Arquitectura del sistema

El sistema cuenta con cuatro módulos principales. A continuación, se describe el funcionamiento de cada uno junto a las tecnologías utilizadas para cada uno de ellos.

- **Módulo Scraper**: Responsable de extraer información en tiempo real desde la página de Waze (Link de la web), además de realizar un proceso de limpieza para dejar los datos que serán insertados al módulo de almacenamiento.
- **Módulo de Almacenamiento**: Responsable de persistir los datos obtenidos por el módulo scrapper para que estos sean consumidos por el módulo de generación de tráfico.
- **Módulo de Servidor Web**: Responsable de manejar los endpoints el cual el cliente (generador de trafico) consumirá para acceder a los datos.
- **Módulo Generador de Tráfico**: Responsable de simular el comportamiento de usuarios u otros sistemas que realizan consultas hacia el sistema diseñado. Genera tráfico en base a los registros del módulo de almacenamiento utilizando patrones de consultas realistas (como distribución normal y Zipf) accediendo mediante el servidor web.
- **Módulo de Cache**: Responsable de almacenar datos insertados por el modulo de generación de tráfico con el fin de reducir la carga sobre el módulo de almacenamiento y de aumentar el rendimiento del sistema gracias a la baja latencia que presenta por ser un sistema de almacenamiento en memoria.

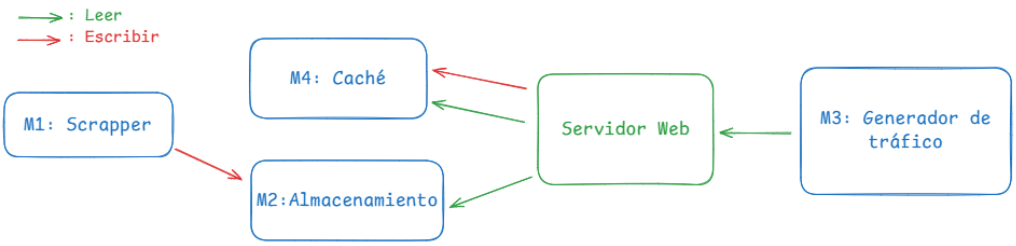

Figura 1: Diagrama de arquitectura realizado en Excalidraw.

3. Implementación de módulos

Se implementaron todos los módulos de manera dockerizada. A continuación se muestra el diagrama y se explican sus respectivas funcionalidades.

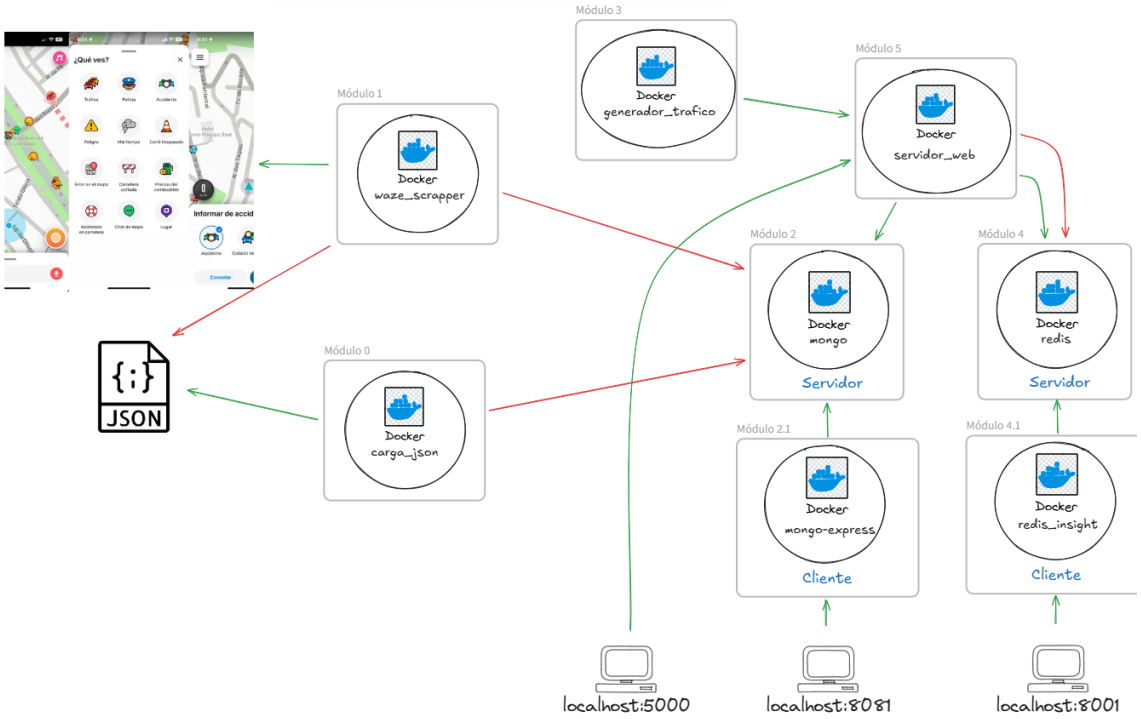

Figura 2: Diagrama de módulos realizado en Excalidraw.

3.1. Uso de Docker

Todos los módulos se ejecutan en contenedores Docker. Cada uno incluye:

- **Dockerfile**: Define cómo se construye el contenedor.
- **docker-compose.yml**: Orquesta y lanza los contenedores del sistema con las configuración asignada para cada uno como vólumenes de datos y la red a la que se conecta.

En el caso de los contenedores que utilizan imagenes de Python, se agrega un archivo **requirements.txt** que contiene las dependencias necesarias para el servicio.

3.2. Web Scrapper

Para extraer información en tiempo real desde la pagina de waze, utilizamos la URL de peticiones descubierta gracias a la herramienta para desarrolladores que utiliza la web.

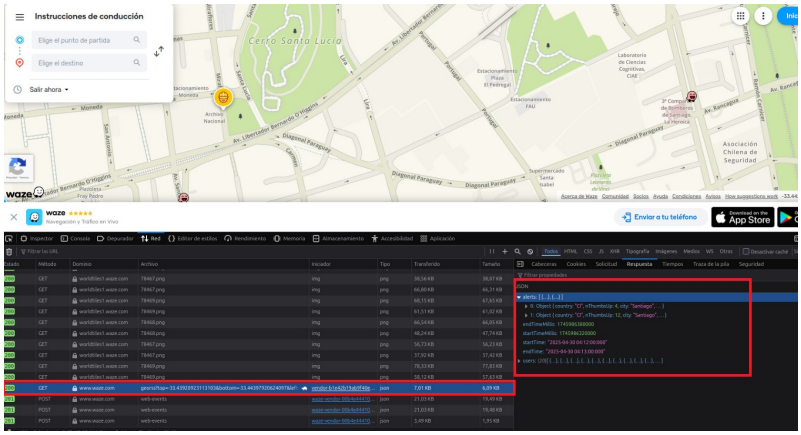

Figura 3: Herramienta para desarrolladores en WAZE.

En este apartado, observamos que Waze actualiza su mapa utilizando la información proveniente de la URL `https://www.waze.com/live-map/api/georss/coordenadas`, donde las *coordenadas* corresponden a una cuadrícula del mapa que el usuario está visualizando.

Dicha URL retorna un archivo en formato JSON como respuesta, el cual contiene datos organizados principalmente en dos categorías: alertas y atascos (*JAMS*). Estos datos incluyen toda la información que posteriormente es consumida por el frontend del mapa.

Al realizar pruebas nos dimos cuenta que cada petición devolvía un máximo de 200 eventos, por lo que se planteó una estrategia para consultar la información de tráfico de toda la Región Metropolitana bajo esta limitación. La estrategia consistió en lo siguiente:

1. Delimitar una cuadrícula que cubriera toda la Región Metropolitana.
2. Dividir dicha cuadrícula en 32 subcuadrículas, de forma que la suma de todas las subcuadrículas sea equivalente a la cuadricula principal.
3. Realizar peticiones para cada una de las 32 subcuadrículas con la librería **Requests**, almacenando los datos correspondientes a alertas y atascos obtenidos de cada una.

4. Insertar los eventos a la base de datos configurada para registros únicos.

Esta estrategia permitió extraer hasta 3000 eventos simultáneos de la región metropolitana durante una ejecución en hora punta.

3.3. Almacenamiento

Para el módulo de almacenamiento se eligió **MongoDB** como base de datos debido a que el scrapper extrae los datos en formato JSON, lo que permite una integración directa y sin necesidad de transformaciones, como requeriría una base de datos relacional. Su modelo basado en documentos se adapta de forma natural a este tipo de estructura.

Se definieron dos colecciones: **Atascos**, que almacena información sobre el estado del tráfico y calles congestionadas; y **Alertas**, que registra situaciones de peligro, presencia de policías, obras y otros eventos clasificados como alertas en los datos originales. Todos los eventos se indexan por UUID, por lo que solo se poblan eventos únicos.

3.4. Visualización de almacenamiento

Este módulo se implementó con una imagen de Docker llamada Mongo Express, se incorporó con el objetivo de facilitar la visualización de los datos almacenados en la base de datos con una interfaz web. Permite gestionar los índices y verificar que no existan problemas con los datos que se van a utilizar.

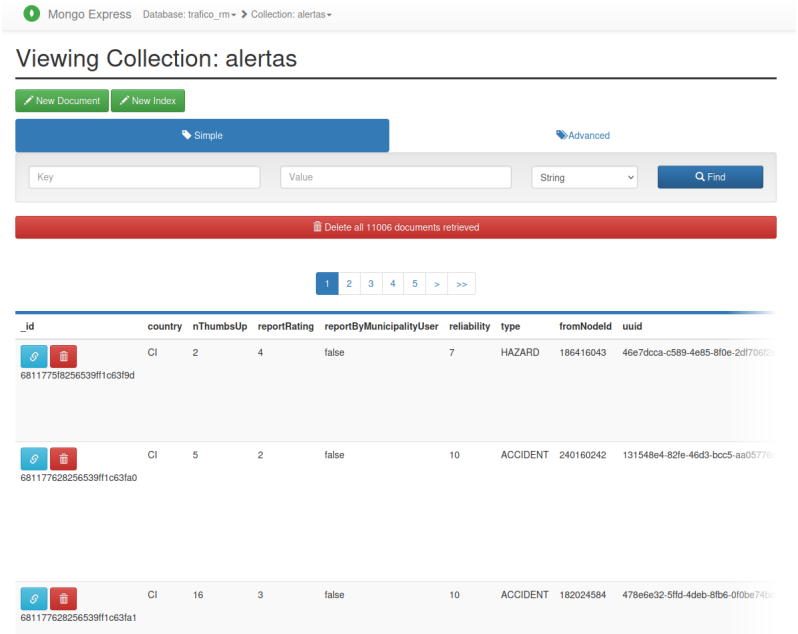

Figura 4: Visualización datos colección alertas en mongo-exrpress.

3.5. JSON Loader

Dado que uno de los requisitos era disponer de al menos 10.000 eventos, y el Scrapper solo obtenía un máximo de 3.000 en el mejor de los casos, se desarrolló este módulo con el objetivo de complementar dicha limitación. El módulo almacena archivos JSON que contienen eventos recopilados con el Web Scrapper durante aproximadamente una semana. Al iniciarse, el módulo ejecuta un script de python el cual carga (mediante la librería pymongo) estos archivos JSON, dejando la base de datos poblada con alrededor de 20.000 eventos únicos tras su ejecución.
Destacar que todos los datos obtenidos se recopilaron mediante el Web Scrapper, y esta es la manera de que las personas que prueben el proyecto tengan una cantidad de datos considerable a la hora de clonar el repositorio.

3.6. Caché

Para la implementación del sistema de caché, se optó por utilizar Redis debido a su simplicidad, eficiencia y amplia compatibilidad con distintos lenguajes, incluido Python. Redis permite establecer mecanismos de almacenamiento en memoria de forma muy ágil, lo que mejora significativamente el rendimiento en consultas repetitivas. Además, su configuración es sencilla y flexible mediante archivos `redis.config`, y fue posible integrarlo en el proyecto con pocas líneas de código gracias a las librerías disponibles para Python.

3.7. Visualización de caché

Este módulo se implementó utilizando Redis Insights, una herramienta de visualización para Redis. Su incorporación permite visualizar la base de datos de Redis y ver el caché en tiempo real mediante una interfaz web intuitiva. Permite inspeccionar las claves almacenadas, ver el uso de memoria y peticiones por segundo, ademas de mostrar el hitrate en tiempo real que tiene el sistema. A continuación, se muestra la vista general que ofrece Redis Insights:

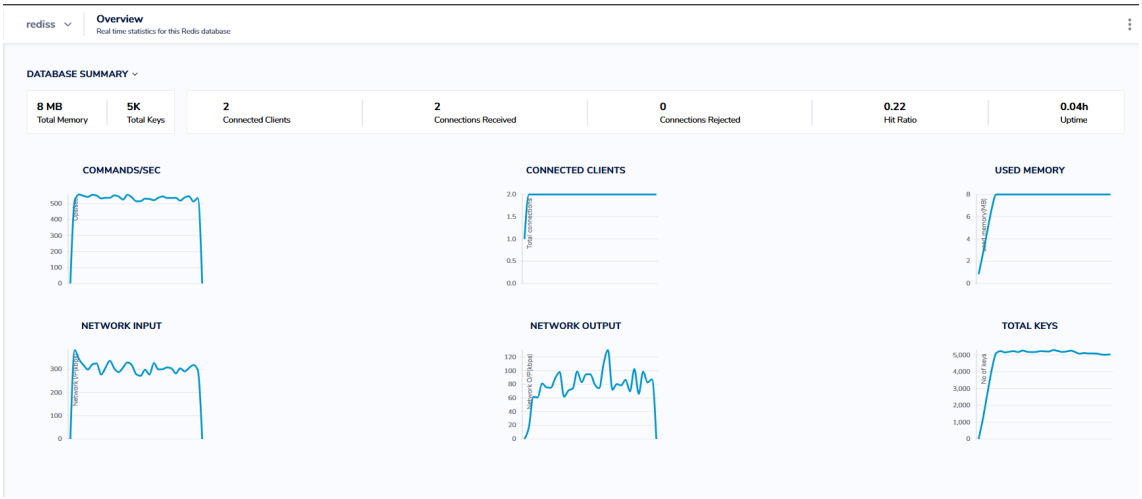

Figura 5: Overview de Redis Insights.

3.8. Servidor Web

Se implementó un servidor web con el objetivo de contar con un módulo encargado de gestionar todas las peticiones realizadas a los distintos servicios de la arquitectura. Este servidor fue desarrollado utilizando el framework **Flask** de Python y dispone de las siguientes funcionalidades:

Conexiones a servicios

- **MongoDB**: Conexión mediante la biblioteca *PyMongo* junto con *MongoClient*.
- **Redis**: Conexión mediante la biblioteca *redis*.

Endpoints principales

- `/uuids_alertas`: Devuelve la lista de todos los UUIDs de alertas almacenadas en la base de datos.
- `/uuids_atascos`: Devuelve la lista de todos los UUIDs de atascos registrados.
- `/alerta/{uuid}`: Busca una alerta, primero en la caché y, en caso de no encontrarla, en la base de datos MongoDB. Además, devuelve el tiempo en mili segundos que demoró en encontrar el evento ya sea en caché o en almacenamiento
- `/atasco/{uuid}`: Realiza el mismo proceso anterior, pero para los atascos.

Flujo esperado de ejecución

Los clientes que interactúan con este servidor deben seguir el siguiente flujo:

1. **Obtención de UUIDs:** El cliente solicita la lista de UUIDs disponibles mediante los endpoints correspondientes.
2. **Consulta de datos:** Una vez elegido un UUID, el cliente realiza una consulta inicial a la caché:
   - Si el dato se encuentra, se registra un *hit*.
   - Si no se encuentra, se registra un *miss* y se consulta la base de datos MongoDB.
3. **Medición de rendimiento:** La API registra los tiempos de acceso para comparar el rendimiento entre las consultas a la caché y a MongoDB, diferenciando entre *hits* y *misses*, y analizando los tiempos de respuesta obtenidos.

3.9. Generador de tráfico

El módulo **Generador de Tráfico** simula el comportamiento de usuarios, realizando peticiones al módulo servidor web, utilizando dos distribuciones de frecuencias de datos para las consultas: **Zipf** y **Normal**. Primero, utiliza los endpoints **GET /uuids_alertas** y **GET /uuids_atascos** para obtener todos los uuids que se encuentran en el modulo de almacenamiento. Luego, se combinan los uuids en un arreglo de objetos que guarda el tipo (alerta, atasco) y el valor del uuid, esto para luego aplicar el tipo de distribución al arreglo y realizar las consultas a los endpoints **GET /alerta/{alerta}** y **GET /atasco/{uuid}** según corresponda. Finalmente se guardan los datos obtenidos de la ejecución en un archivo JSON que contiene el numero total de consultas, el numero de hits, el numero de misses, el valor del hitrate, el tiempo en promedio que demoró la respuesta de caché (cuando era hit) y el tiempo promedio que demoró la respuesta de almacenamiento (cuando era miss). A continuación, se muestra un ejemplo del archivo JSON que guarda este módulo:

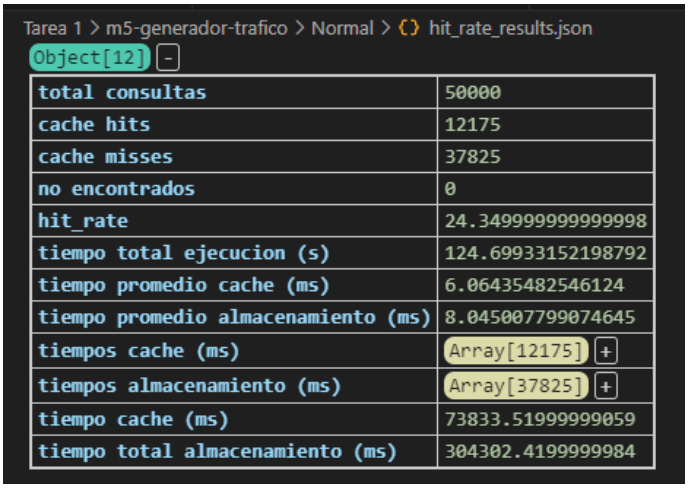

Figura 6: Archivo de resultados al ejecutar el generador de trafico con 50.000 peticiones siguiendo una distribución normal.

4. Pruebas y resultados

Se realizaron pruebas para distintos tamaños de cache, numero de consultas y políticas de remoción. A continuación se muestran los resultados obtenidos:

4.1. Distribución de frecuencias

Para comenzar, se realizaron pruebas sin considerar la tasa de arribo de las consultas, es decir, que cuando se levantar el contenedor de generación de tráfico, este envía las consultas sin un tiempo de espera entre ellas. Bajo este contexto se espera medir el hitrate evaluando las políticas de remoción LRU y LFU para las distribuciones de frecuencia Zipf y Normal.

4.1.1. Distribución Zipf

En esta distribución se cumple que existen datos mas populares que otros. Esto significa que algunos elementos aparecen con mas frecuencia que el resto, mientras que la mayoría son poco frecuentes. Gráficamente tiene forma de potencia inversa.

Para la prueba se utilizan 50.000 datos distribuidos con Zipf y se parametriza el caché con 1MB de memoria (aproximadamente 90 entradas), asignando la política de remoción LRU en el archivo **redis.conf**, obteniendo el siguiente resultado:

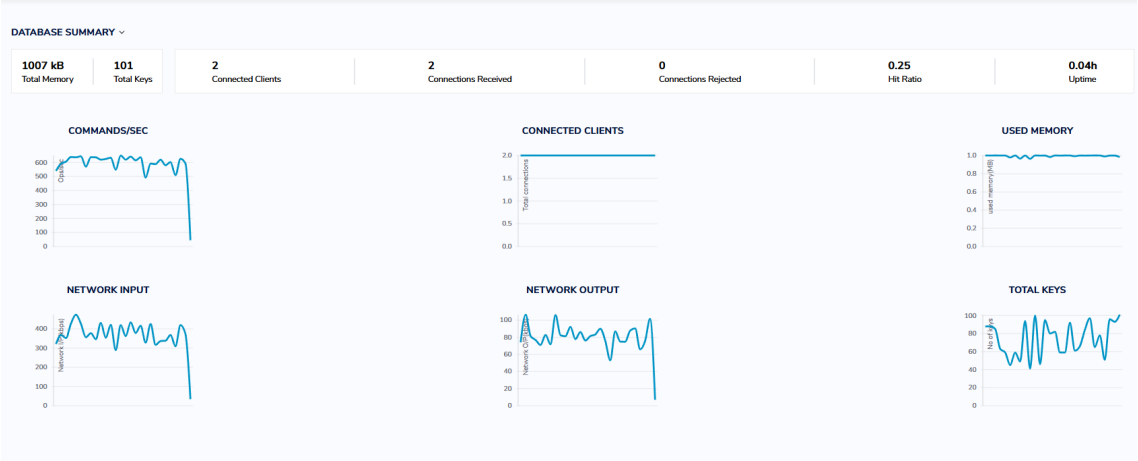

Figura 7: Resultado de consultar 50.000 datos al módulo servidor web con LRU en distribución Zipf de frecuencias.

Luego, se cambia la política de remoción a LFU y se obtiene el siguiente resultado:

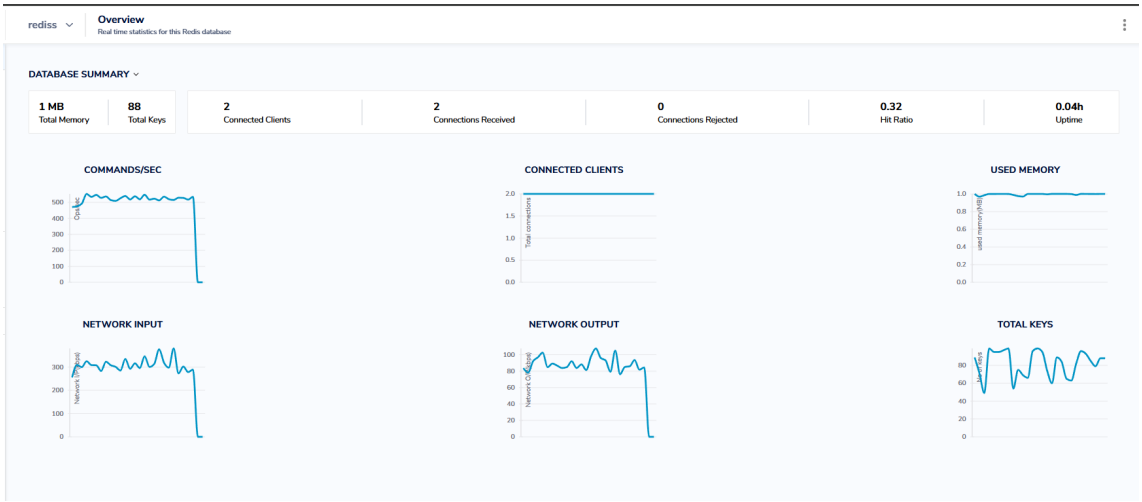

Figura 8: Resultado de consultar 50.000 datos al módulo servidor web con LFU en distribución Zipf de frecuencias.

Como se observa en las imágenes, cuando se utiliza la política LRU se obtiene un 25 % de hitrate, mientras que al utilizar la política LFU se obtiene un 32 % de hitrate. Esto se debe a que la política de LFU se encuentra mejor adaptada en escenarios donde la frecuencia de repeticiones es muy alta para algunos elementos (suficientes para que entren al caché). Además LRU se ve afectado por el bajo tamaño del caché que aumenta la competencia, provocando que la política elimine llaves que es probablemente va a necesitar en poco tiempo.

4.1.2. Distribución Normal

En esta distribución, se cumple que los valores tienden a agruparse en torno a un valor central, y mientras mas se alejan de este, menos frecuente son. En esta distribución, la media, la mediana y la moda coinciden en este punto central, y la probabilidad de observar valores extremos es baja. Gráficamente, tiene forma de campana.

Para la prueba se utilizan 50.000 datos a los cuales se les aplico una distribución normal, se parametriza el caché con 8MB de memoria (aproximadamente 5000 entradas), asignando la política de remoción LRU en el archivo **redis.conf**, obteniendo el siguiente resultado:

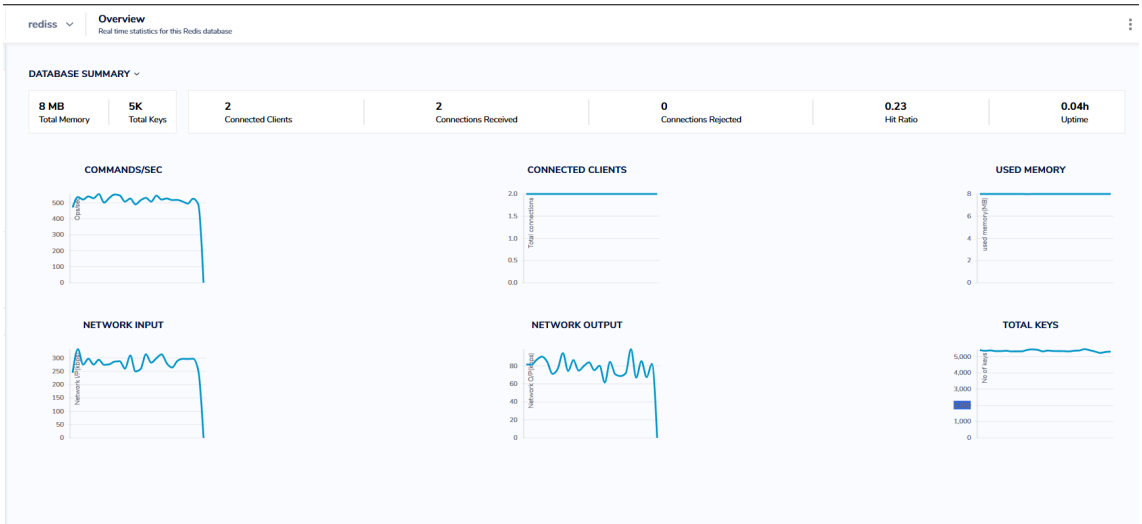

Figura 9: Resultado de consultar 50.000 datos al módulo servidor web con LRU en distribución normal de frecuencias.

Después, se cambia la política de remoción a LFU, obteniendo el siguiente resultado:

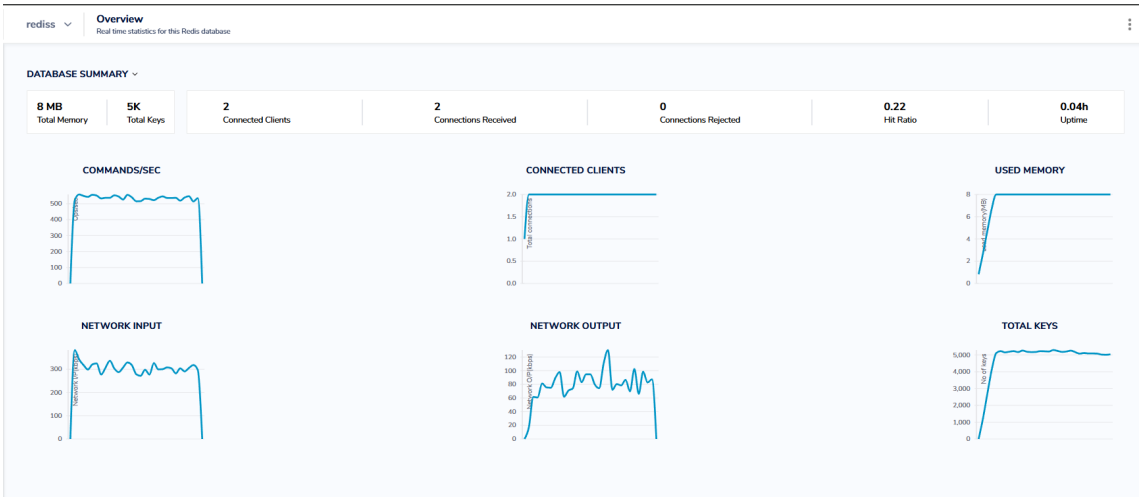

Figura 10: Resultado de consultar 50.000 datos al módulo servidor web con LFU en distribución normal de frecuencias.

En las imágenes se puede apreciar que para el caso de LRU se obtiene un 23 % de hitrate, mientras que para LFU se obtiene un 22 % de hitrate, siendo superior la política de remoción LRU debido a la distribución de frecuencias aplicada. Aunque LRU tiene mejor hitrate que LFU, este podría ser mucho mayor si se aplican iteraciones a la distribución, por ejemplo, realizando cambios a los uuids que más se repiten, ya que LRU se adapta mucho mas rápido a estos cambios. Por otro lado, LFU no se adapta bien a cambios de frecuencias ya que se basa en el comportamiento histórico de los datos, manteniendo registros que en algún momento fueron relevantes pero que podrían no serlo para un contexto actual.

4.2. Distribución de frecuencias y tasa de arribo

Sumado al apartado anterior, se implementan distribuciones para las tasas de arribo de las frecuencias, esto para ver como se comporta el sistema frente a picos de tráfico y como maneja las entradas a lo largo del tiempo mediante el valor del TTL.

Para esta prueba se utiliza una distribución de Poisson / Exponencial para que tanto el numero de consultas como el tiempo entre consultas sea variable a lo largo de la ejecución y se comparan ambas políticas de remoción. Además, para la distribución de frecuencias se utiliza zipf con un mayor sesgo (1.3) de modo que se repitan aun más entradas, esperando aumentar el hitrate en esta prueba. Primero se realiza la prueba configurando Redis con LRU como política de remoción, obteniendo el siguiente resultado:

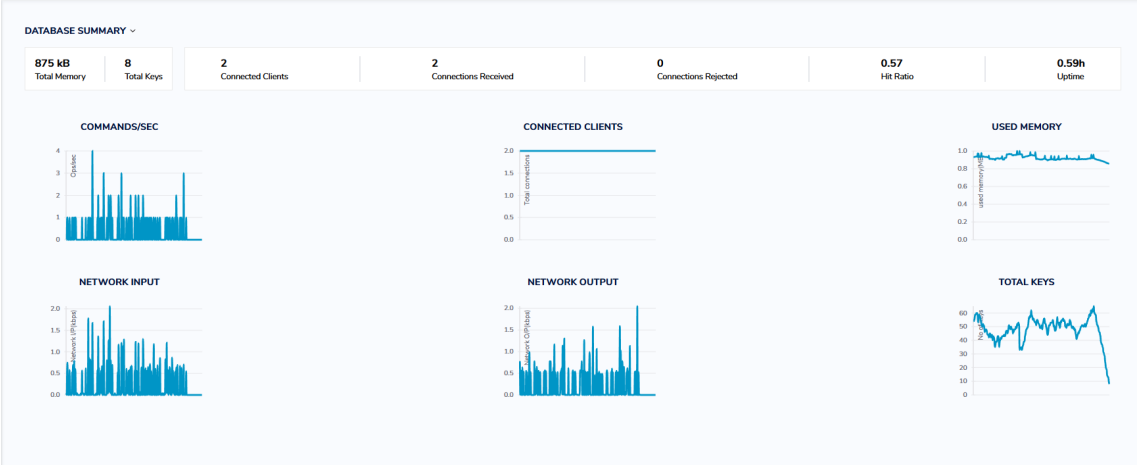

Figura 11: Resultado de consultar 1000 datos al módulo servidor web con LRU en distribución de Poisson.

Después, se cambia la política de remoción a LFU, obteniendo el siguiente resultado:

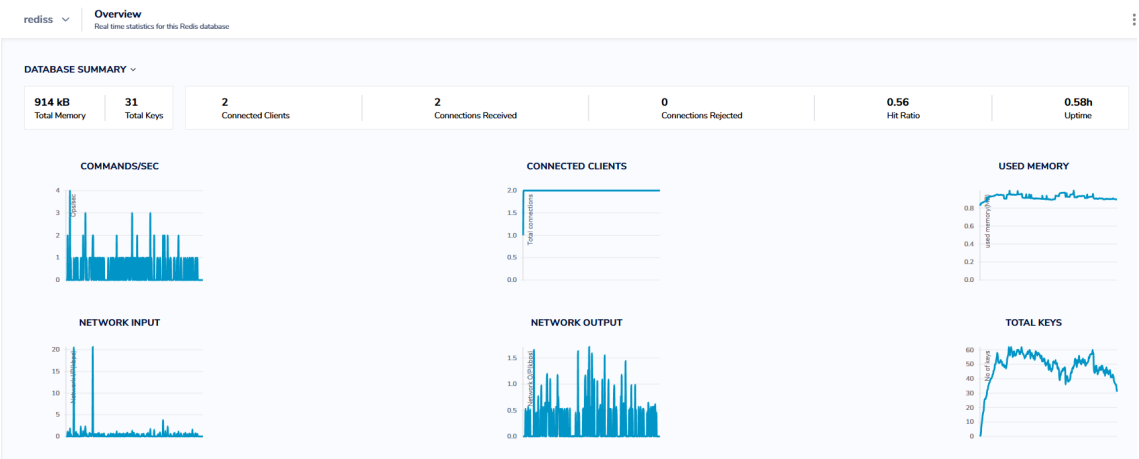

Figura 12: Resultado de consultar 1000 datos al módulo servidor web con LFU en distribución de Poisson.

Como se observa en las imágenes, las pruebas duran cerca de 40 minutos y muestran la respuesta al sistema a los picos de tráfico y repetición de consultas, siendo positiva y estable como se esperaba luego de realizar las pruebas anteriores que estresaban aún más el sistema. En esta prueba se puede ver un aumento en el hitrate debido al aumento del sesgo en la distribución de frecuencias.

5. Análisis y discusión.

Bajo el contexto del proyecto, se realiza un análisis de las elecciones en cuanto a tecnologías y técnicas utilizadas.

5.1. Generador de tráfico

¿Por qué utiliza la distribución seleccionada en el generador de tráfico? ¿En qué se basa dicha decisión? ¿Es realista la distribución utilizada? ¿En qué escenarios vemos este tipo de comportamientos?

Se optó inicialmente por una distribución de Zipf, ya que es la que mejor se ajusta al comportamiento esperado en sistemas con caché. Esta elección se basa en que la frecuencia de acceso a los datos sigue una tendencia exponencial: un pequeño grupo de elementos es consultado con mucha mayor frecuencia que el resto.

Este tipo de distribución es representativo de escenarios reales, como los motores de búsqueda, donde ciertas consultas se repiten constantemente. En un contexto como el de Waze, se puede inferir que durante determinadas condiciones (por ejemplo, horas punta), la mayoría de las consultas se concentran en zonas específicas con alta densidad de tráfico, como el centro de Santiago, siguiendo así una distribución similar a Zipf.

5.2. Almacenamiento

¿Por qué utiliza este sistema de almacenamiento y no otro? ¿Qué comportamiento o funcionalidades son las que justifican su decisión? ¿Qué limitantes tiene el sistema utilizado?

El sistema de almacenamiento seleccionado se basa exclusivamente en el formato de respuesta de los archivos extraídos desde la web de Waze. Como se menciona en la implementación, los archivos extraídos desde la URL que alimenta el frontend están en formato JSON, el cual contiene toda la información del tráfico de manera estructurada, con numerosos campos anidados. Estos campos anidados resultan complicados de manejar en SQL, ya que implicarían la creación de múltiples tablas intermedias. Por esta razón, se decidió almacenar los JSON en MongoDB para mantener el formato original.

Aunque esta elección fue relativamente sencilla de implementar, las posibles limitaciones incluyen la escalabilidad y el rendimiento en consultas complejas, dado que MongoDB, al ser una base de datos NoSQL, podría enfrentar dificultades al manejar grandes volúmenes de datos o realizar operaciones de búsqueda y filtrado más avanzadas, pero para efectos de la actividad, fue mas que suficiente.

5.3. Métricas

¿Qué métricas considera esenciales para evaluar la eficiencia de su sistema de caché? ¿Qué política de remoción le resulta más eficiente para este sistema y sus eventos? ¿Tiene un efecto relevante sobre el rendimiento del caché la distribución de tráfico que emplea en el generador de tráfico implementado?

Las métricas esenciales seleccionadas para evaluar la eficiencia del caché fueron las siguientes:

- **Hit rate:** Proporción de búsquedas exitosas en el caché respecto al total de búsquedas realizadas.
- **Miss rate:** Proporción de búsquedas no encontradas en el caché respecto al total de búsquedas realizadas.
- **Tiempo promedio caché:** Tiempo promedio que tarda una respuesta en devolverse cuando se trata de un *hit*.
- **Tiempo promedio almacenamiento:** Tiempo promedio que tarda una respuesta en devolverse cuando se trata de un *miss*.

Estas métricas permiten evaluar que tanto optimiza el caché las respuestas que puede devolver a los usuarios en cuanto a tiempo, y computo en el backend.

En cuanto a las políticas de remoción, estas tienen un impacto directo en el rendimiento del caché. Asumiendo que el tráfico en Waze sigue una distribución Zipf durante las horas punta, la política LFU (*Least Frequently Used*) podría ser más eficaz para mejorar el *hit rate*, ya que prioriza las entradas de caché que son solicitadas con mayor frecuencia. Dado que en una distribución Zipf, las solicitudes más frecuentes tienen una probabilidad mucho mayor de repetirse, LFU asegura que los elementos más solicitados permanezcan en caché, optimizando el acceso a estos. Asumiendo otras distribuciones, otras políticas se adaptarían mejor para tener mejor eficiencia.

5.4. Pruebas de rendimiento

¿Cuál es el objetivo de los escenarios de prueba generados? ¿El escenario implementado refleja un comportamiento realista? ¿El diseño propuesto garantiza alta disponibilidad y escalabilidad?

Los escenarios de prueba fueron diseñados específicamente para evaluar el rendimiento del sistema de caché, con el objetivo de observar cómo el precomputo de respuestas frecuentes puede reducir significativamente la carga sobre el backend y, como resultado, mejorar los tiempos de respuesta.

El escenario evaluado es realista desde el punto de vista del comportamiento del caché, ya que en aplicaciones reales las consultas suelen seguir una distribución específica. Por ello, las configuraciones del caché deben adaptarse a dicho patrón. En este caso, se asumió que las consultas siguen una distribución Zipf (común en tráfico real como el de Waze), y se ajustó la estrategia del caché para optimizar el rendimiento bajo esa condición.

Esta solución provee alta disponibilidad, primero por su diseño modular, y también por su capacidad de escalar horizontalmente, permitiendo múltiples instancias del sistema en paralelo para atender grandes volúmenes de solicitudes.

6. Conclusión

Una vez completada la actividad y realizar los respectivos análisis de esta, se concluye lo siguiente:

Se logró aplicar los conocimientos de sistemas distribuidos adquiridos durante la primera parte del curso. Se logró comprender, de forma básica, cómo diseñar arquitecturas modulares y cómo desplegar servicios de manera rápida utilizando Docker, además de explorar técnicas para poblar este tipo de proyectos con datos reales.

Se evidenció el impacto del uso de caché mediante pruebas experimentales y la recolección de métricas, demostrando que la implementación de un servicio de caché mejora significativamente los tiempos de respuesta y el rendimiento general del sistema.

Referencias

[1] Docker, Inc. **Docker Hub - Python Official Image**. Disponible en: `https://hub.docker.com/_/python` [Último acceso: abril 2025].

[2] SelfTuts. **Instalar Redis usando Docker Compose**. Disponible en: `https://selftuts.in/install-redis-using-docker-compose/` [Último acceso: abril 2025].

[3] Docker, Inc. **Docker Hub - Mongo Express Official Image**. Disponible en: `https://hub.docker.com/_/mongo-express` [Último acceso: abril 2025].

[4] Redis Ltd. **redis-py – Python Client for Redis**. Disponible en: `https://redis.io/docs/latest/develop/clients/redis-py/` [Último acceso: abril 2025].

[5] MongoDB, Inc. **PyMongo - MongoDB Python Driver**. Disponible en: `https://pymongo.readthedocs.io/en/stable/tutorial.html` [Último acceso: abril 2025].

[6] Docker, Inc. **Docker Compose Documentation**. Disponible en: `https://docs.docker.com/compose/` [Último acceso: abril 2025].