UNIVERSIDAD DIEGO PORTALES

21 de Abril, 2024

Sistemas Distribuidos: Tarea 1

Integrantes:

Jose Martín Berrios Piña

Renato Óscar Benjamín Contreras Carvajal

Índice

1. Introducción

La gestión eficiente de datos en entornos distribuidos es crucial para garantizar un rendimiento óptimo y una alta disponibilidad de servicios. En este contexto, el uso de sistemas de caché distribuido juega un papel fundamental al mejorar el acceso a datos y reducir la carga en los sistemas de almacenamiento centralizados.

1.1. Descripción del Problema

El presente informe aborda el diseño e implementación de un sistema de caché distribuido para mejorar el rendimiento y la escalabilidad de aplicaciones distribuidas. Se enfrenta al desafío de gestionar eficientemente la caché de datos en un entorno distribuido, minimizando la latencia de acceso y optimizando el uso de recursos.

2. Objetivos

El propósito de esta tarea consiste en el diseño, implementación y evaluación de un sistema de caché distribuido utilizando Redis. Dicho sistema será sometido a pruebas en varios escenarios y con diferentes conjuntos de datos, con el objetivo de evaluar su rendimiento bajo diversas condiciones de carga. Esto permitirá comparar y contrastar las distintas configuraciones del sistema de caché distribuido, en términos de su eficiencia, robustez y capacidad para gestionar cargas de trabajo variables.

3. Redis

Como un estudio inicial de el software se puede decir que Redis se caracteriza por ser una base de datos en memoria. Esta proporciona soluciones tanto en la nube como en locales, destinadas al almacenamiento en caché, la búsqueda vectorial y las bases de datos NoSQL. Estas características permiten a los clientes digitales la creación, el escalado y el despliegue de aplicaciones de manera rápida y eficiente.

3.1. Configuraciones para tratar con particiones y replicas

En esta sección, se hace referencia a las fuentes 1 y 2 de la bibliografía, las cuales explican que el particionamiento y la replicación en Redis se logran a través de Redis Cluster. Esta es una topología de implementación que permite la división automática de los datos entre múltiples nodos de Redis, actuando como particionamiento.

Redis Cluster no utiliza hashing consistente, sino una forma de particionamiento en la que cada clave es parte de un “hash slot”. Existen un total de 16384 hash slots, distribuidos

entre los nodos del clúster. Esto facilita la adición y eliminación de nodos, ya que los hash slots pueden ser movidos entre nodos sin detener las operaciones.

En cuanto al método de replicación, se utiliza el modelo maestro-réplica que garantiza la disponibilidad al tener réplicas para cada nodo maestro. Si un nodo maestro falla, su réplica puede ser promovida como nuevo maestro. Sin embargo, si tanto el maestro como su réplica fallan simultáneamente, el clúster no podrá continuar operando.

3.2. Políticas de remoción

Como se puede observar en la fuente 3 de la bibliografía, se presentan las siguientes políticas de remoción para utilizar en Redis:

1. **Noeviction**: No se guardan nuevos valores cuando se alcanza el límite de memoria. Cuando una base de datos utiliza replicación, esto se aplica a la base de datos primaria.
2. **Allkeys-lru**: Mantiene las claves más recientemente utilizadas; elimina las claves menos recientemente utilizadas (LRU).
3. **Allkeys-lfu**: Mantiene las claves utilizadas con mayor frecuencia; elimina las claves utilizadas con menos frecuencia (LFU).
4. **Volatile-lru**: Elimina las claves menos recientemente utilizadas con el campo de expiración (expire) establecido en verdadero.
5. **Volatile-lfu**: Elimina las claves menos utilizadas con el campo de expiración (expire) establecido en verdadero.
6. **Allkeys-random**: Elimina claves al azar para hacer espacio para los nuevos datos agregados.
7. **Volatile-random**: Elimina claves al azar con el campo de expiración (expire) establecido en verdadero.
8. **Volatile-ttl**: Elimina claves con el campo de expiración (expire) establecido en verdadero y el tiempo de vida restante (TTL) más corto.

4. Conjuntos de datos ocupados

En la elección de los conjuntos de datos a utilizar, se tuvo en cuenta que fueran de gran volumen y que contuvieran repeticiones. Por lo tanto, se utilizarán datos (ver Referencias 9) filtrados con Python con el archivo 'creador-bdd.py', tomando en cuenta las columnas que contengan números y palabras. Esto generará dos archivos: 'init.sql' e 'init2.sql'. El

primero contiene aproximadamente 10,000 filas y se utiliza para las pruebas de rendimiento, mientras que el segundo contiene 1,000 filas.

5. Implementación

Los códigos correspondientes al proyecto se hallan disponibles en GitHub (ver referencia 4 en la bibliografía). Para facilitar su comprensión, estos solo se mencionarán y se presentarán a través de ejemplos ilustrativos. Además, en la implementación de cada sistema se utilizan varias herramientas, como:

- **Redis**: Para la gestión de caché.
- **Docker**: Para la creación de contenedores y la gestión del despliegue del sistema.
- **PostgreSQL Alpine**: Para la base de datos, se utiliza la imagen de PostgreSQL optimizada para entornos con recursos limitados (PostgreSQL Alpine).
- **Python**: Para la implementación del backend y la lógica de la aplicación, se utiliza Python.

5.1. Red con un Contenedor de Redis Centralizado

Para el uso de las distintas pruebas, se utilizan los mismos archivos, pero varían los ajustes de conexión con Redis y la inicialización de sus contenedores. Para el caso exclusivo de Redis centralizado, se utiliza 1 contenedor en Docker Compose (los demás se comentan) y se establece su conexión con la API.

```
17  # Configuración de Redis, hosts, limpieza de cache
18  redis_hosts = ["myredis1"]
19  redis_instances = [redis.Redis(host=host, port=6379, db=0) for host in redis_hosts]
20  [redis_instance.flushall() for redis_instance in redis_instances]
21
```

Figura 1: Conexion de 1 solo contenedor de Redis.

Luego, para cada sistema, se aplican las siguientes opciones: se realiza la conexión con el servidor a través de gRPC.

```python
# Cliente para funcionalidad gRPC
class SearchClient(object):
    def __init__(self):

        #Stub que crea conexion entre el cliente y el backend(Servidor gRPC)
        self.stub = pb2.SearchStub(grpc.insecure_channel('{}:{}'.format('backend', '50051')))

    #Funcion utilizada para retornar la respuesta del servidor a un mensaje
    def get_url(self, message):
        message = pb2_grpc.Message(message=message)

        # Impresion del mensaje
        print(f'{message}')
        # Llamada al procedimiento remoto GetServerResponse y retorno del resultado
        return self.stub.GetServerResponse(message)

```

Figura 2: Conexión entre la API y el servidor.

Continuamente se crea una ruta con el método GET para poder realizar pruebas con la API.

```python
@app.route('/search', methods=['GET'])
def search():
    # Avisamos que trabajaremos con las variables globales
    global contador_cache
    global balanceador_entrecomillas

    # Captura el tiempo de inicio de la operación
    mystart = time.time()

    # Crea una instancia de SearchClient
    client = SearchClient()

    # Realiza impresiones múltiples de la instancia client (útil para propósitos de depuración)
    print(client)

    # Obtiene el parámetro de consulta 'search' de la solicitud HTTP
    myquery = request.args['id']

    #Arreglo utilizado para mostrar lo solicitado en la consulta
    cache_search = [redis_instance.get(myquery) for redis_instance in redis_instances]

```

Figura 3: La dirección /search será utilizada para realizar búsquedas.

Continuamente, en esta ruta, se verifica si los datos buscados están en caché. En caso de no ser así, se guardan en este y se buscan en la base de datos. Si se encuentran en Redis, se entregan desde la caché.

```python
    # Verifica si no hay datos en la caché
    if all(value is None for value in cache_search):
        # Le pedimos la data al servidor
        data = client.get_url(message=myquery)

        if balanceador_entrecomillas == 3:
            balanceador_entrecomillas = 0
        # Almacena los datos en la instancia de Redis
        redis_instances[balanceador_entrecomillas].set(myquery, str(data))
        redis_selected = "Almacenado en el redis " + str(balanceador_entrecomillas+1)
        balanceador_entrecomillas += 1
        # Renderizamos el html con los datos obtenidos de PostgreSQL
        return render_template('index.html', mydata=data, procedencia="Datos sacados de PostgreSQL en: " + str(int((time.time() - mystart) * 1000)) + "ms", redis_selected=r
    else:
        # Incrementa el contador de caché
        contador_cache += 1
        # Itera sobre los datos almacenados en la caché
        for datos in cache_search:
            # Verifica si hay datos en la entrada actual de la caché
            if datos != None:
                # Crea un diccionario para almacenar los datos
                dicc = dict()
                dicc['Resultado'] = datos.decode("utf-8")
                # Renderiza la plantilla 'index.html' con los datos obtenidos de Redis
                return render_template('index.html', mydata=dicc['Resultado'], procedencia="Datos sacados de Redis en: " + str(int((time.time() - mystart) * 1000)) + "ms",
```

Figura 4: Proceso de toma de decisiones para la entrega de datos.

Ahora, trabajando con el servidor, se realiza la conexión gRPC con la API y se establece la conexión con la base de datos.

```python
def serve():
    print("Server started")
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    pb2_grpc.add_SearchServicer_to_server(SearchService(), server)
    server.add_insecure_port('[::]:50051')
    server.start()
    server.wait_for_termination()

if __name__ == '__main__':
    #Esperamos la conexion para tener con el servidor o algo asi
    #Y que se carguen los datos
    sleep(18)
    conn = querys.init_db()
    cursor = conn.cursor()
    serve()
```

Figura 5: Funciones de conexión con la API y con la base de datos PostgreSQL.

Además, se configura la función que responderá a través de gRPC a la API en caso de que se le solicite.

```
26    postgres:
27      image: postgres:latest
28      container_name: postgres-tarea1
29      volumes:
30        - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql
31      ports:
32        - "5432:5432"
33      environment:
34        POSTGRES_PASSWORD: postgres
35        POSTGRES_USER: postgres
36        POSTGRES_DB: tarea1
```

Figura 6: Función que responde a la API.

La base de datos se inicializa gracias al archivo docker-compose.yaml y al init.db. El Docker Compose le asigna una imagen oficial de Docker, mientras que el archivo init.db crea la base de datos y la rellena con 9564 filas. También existe otro archivo de 1000 líneas que puede ser utilizado al cambiar las direcciones de enlace.

```
26    postgres:
27      image: postgres:latest
28      container_name: postgres-tarea1
29      volumes:
30        - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql
31      ports:
32        - "5432:5432"
33      environment:
34        POSTGRES_PASSWORD: postgres
35        POSTGRES_USER: postgres
36        POSTGRES_DB: tarea1
```

Figura 7: Creación del contenedor en docker-compose.yaml.

5.2. Red con Contenedores de Redis Particionados

Con la API en el sistema con Redis particionado, se siguen los pasos mencionados anteriormente, y para su correcto funcionamiento, se debe realizar la conexión con los distintos contenedores en Docker. Por defecto, actúan de forma particionada.

```
# Configuración de Redis, hosts, limpieza de cache
redis_hosts = ["myredis1", "myredis2", "myredis3"]
redis_instances = [redis.Redis(host=host, port=6379, db=0) for host in redis_hosts]
[redis_instance.flushall() for redis_instance in redis_instances]

```

Figura 8: Conexión de los 3 contenedores de Redis y su limpieza de caché.

5.3. Red con Contenedores de Redis Replicados

Finalmente, para la implementación de Redis particionado, se debe realizar la conexión a los 3 contenedores de Redis y agregar los siguientes comandos.

```
  myredis1:
    image: bitnami/redis:latest
    container_name: mycontenedor-1-redis
    command: redis-server --appendonly yes
    ports:
      - "6379:6379"
    environment:
      - ALLOW_EMPTY_PASSWORD=yes
    restart: always

  myredis2:
    image: bitnami/redis:latest
    container_name: mycontenedor-2-redis
    command: redis-server --appendonly yes --slaveof myredis1 6379
    ports:
      - "7000:6379"
    environment:
      - ALLOW_EMPTY_PASSWORD=yes
    restart: always

  myredis3:
    image: bitnami/redis:latest
    container_name: mycontenedor-3-redis
    command: redis-server --appendonly yes --slaveof myredis1 6379
    ports:
      - "7001:6379"
    environment:
      - ALLOW_EMPTY_PASSWORD=yes
    restart: always
```

Figura 9: Conexión de los 3 contenedores de Redis como maestros y esclavos.

6. Preguntas y Escenarios

6.1. Preguntas

1. Explique y compare los beneficios de utilizar una comunicación gRPC vs una API REST HTTP clásica.
   R: gRPC y REST son dos formas de diseñar una API. gRPC, construido sobre HTTP/2, ofrece ventajas en comparación con las API HTTP REST. Los mensajes de gRPC se serializan mediante Protobuf, un formato de mensaje binario que genera cargas de mensajes pequeñas, lo cual es ayuda en escenarios de ancho de banda limitado. Por otro lado, REST utiliza HTTP. Las API RESTful gestionan las comunicaciones entre un cliente y un servidor a través de verbos HTTP, como POST, GET, PUT y DELETE. Entonces se puede decir que cada uno ofrece enfoques distintos para la comunicación entre sistemas. Mientras que gRPC se destaca por su eficiencia, serialización compacta y soporte para llamadas bidireccionales, REST es más ampliamente compatible y utiliza formatos más legibles por humanos.

2. ¿Para todos los sistemas es recomendable utilizar caché?
   R: No siempre es recomendable utilizar caché para todos los sistemas. La memoria caché es útil cuando los datos permanecen relativamente estáticos y el acceso a los datos de un servidor es más lento en comparación con la velocidad de la caché. Entonces, si los datos cambian con frecuencia o si el sistema requiere acceso en tiempo real a los datos más actualizados, el uso de caché puede resultar contraproducente.

3. ¿Qué ventajas aporta un caché replicado en comparación con un sistema clásico en términos de velocidad y eficiencia?
   R: Un caché replicado puede mejorar la eficiencia de un sistema informático al eliminar la necesidad de descargar datos y recursos innecesarios, lo cual es una de las ventajas de los caches. En términos de eficiencia, aporta una mayor consistencia de datos al haber más nodos con la misma información disponible. Sin embargo, en términos de velocidad, un sistema de caché replicado actuaría de manera similar a un sistema de caché clásico.

4. ¿Qué ventajas aporta un caché particionado en comparación con un sistema clásico en términos de velocidad y eficiencia?
   R: Un caché particionado puede distribuir la carga de manera más uniforme y permitir un acceso más rápido a los datos al evitar cuellos de botella en un único punto de acceso. En términos de velocidad, esto se traduce en tiempos de respuesta más rápidos para las solicitudes de datos, ya que las operaciones se distribuyen entre varios nodos en lugar de depender de un único servidor. En cuanto a la eficiencia, el caché particionado puede reducir la sobrecarga en el sistema al gestionar de manera

más efectiva la carga de trabajo y al proporcionar una mayor capacidad para escalar horizontalmente según sea necesario.

5. ¿Bajo qué condiciones o escenarios utilizaría un sistema de caché clásico, particionado o replicado?
   R: El caché clásico se utiliza en escenarios donde la carga de datos es manejable por un solo servidor y no se requiere una distribución de carga entre múltiples nodos. Por lo tanto, es recomendable para aplicaciones con un volumen de datos relativamente pequeño y una cantidad moderada de solicitudes, donde la complejidad adicional de implementar un caché distribuido no justifica los beneficios adicionales.

   Por otro lado, el caché particionado se emplea cuando la carga de datos es alta y se necesita escalar horizontalmente para manejar un gran volumen de solicitudes de manera eficiente. Este enfoque es adecuado para aplicaciones con una gran cantidad de datos o con un alto tráfico de lectura/escritura. Al particionar los datos entre múltiples nodos, se puede lograr una mejor distribución de la carga y una respuesta más rápida a las solicitudes. Además, cada nodo de caché se encarga de una parte específica de los datos, lo que permite manejar una mayor cantidad de solicitudes simultáneas.

   Por último, el caché replicado se utiliza en situaciones donde la alta disponibilidad y la tolerancia a fallos son prioritarias, como en escenarios donde la consistencia de los datos entre múltiples nodos es crucial. Además, la redundancia de datos puede proporcionar una capa adicional de seguridad contra la pérdida de datos o el mal funcionamiento del servicio.

6. Usando de referencia el contexto del problema y los escenarios clásico, particionado y replicado, realice un análisis respecto a las distintas políticas de remoción.
   R: Conociendo que las políticas de remoción en un sistema de caché se refieren a cómo se decide qué datos se eliminan de la caché cuando esta se llena y que estamos trabajando con redis, nos enfocaremos con las que este trabaja: Least Recently Used (LRU), Least Frequently Used (LFU), random y Volatile TTL. Partiendo con Least Recently Used (LRU) que elimina los elementos menos recientemente utilizados de la caché cuando se necesita espacio para nuevos elementos. Es útil en escenarios donde la temporalidad de acceso a los datos es importante y se desea maximizar la probabilidad de mantener en caché los datos más relevantes y recientes. Por ejemplo, en aplicaciones web donde ciertas páginas o recursos son solicitados con mayor frecuencia en un periodo medible, como la red social Twitter/X.

   La política Least Frequently Used (LFU) elimina los elementos menos frecuentemente utilizados de la caché. Es útil cuando la frecuencia de acceso a los datos es un mejor indicador de su relevancia que el tiempo transcurrido desde su última utilización.

Por ejemplo, en aplicaciones donde ciertos datos son accesados con una frecuencia constante pero irregular, por lo que ayuda a mantener en caché los elementos más relevantes y descartar aquellos que rara vez son solicitados.

Volatile-random elimina aleatoriamente claves de la caché cuando se necesita espacio y cuando el campo de expiración esta en verdadero. Aunque es simple de implementar, puede no ser la opción más eficiente ya que no toma en cuenta ni la frecuencia ni la recencia de uso de los datos. Esta política podría ser útil en escenarios donde no hay un patrón claro de acceso a los datos.

Finalmente, la política Volatile TTL (Time To Live) se refiere a la eliminación de elementos de la caché basada en un tiempo de vida predefinido. Los elementos en la caché tienen un tiempo de vida asignado y el campo de expiración activo, por lo que cuando se llena la cache, elimina el elemento con tiempo de vida restante (TTL) más corto. Esta política es útil en escenarios donde los datos tienen una relevancia que disminuye con el tiempo o cuando los datos se actualizan en intervalos regulares.

7. Análisis de Rendimiento de Configuraciones de Redis

En nuestras pruebas de rendimiento, evaluamos tres configuraciones de Redis: centralizado, replicado y particionado. Utilizamos dos métricas principales para evaluar el rendimiento: tiempo de respuesta de Redis y tasa de aciertos del caché.

7.1. Tiempo de Respuesta de Redis

En general, observamos que el tiempo de respuesta de Redis fue mejor en el sistema de caché particionado en comparación con las otras dos configuraciones. Esto se debe a la capacidad del sistema particionado para distribuir la carga entre múltiples nodos, lo que resultó en tiempos de respuesta más rápidos incluso en casos de carga con 1K de consultas.

A continuación, se presentan los gráficos que muestran el tiempo de respuesta de Redis para cada configuración en diferentes niveles de carga:

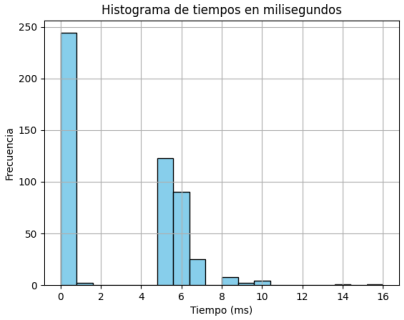

Figura 10: Sistema de caché centralizado.

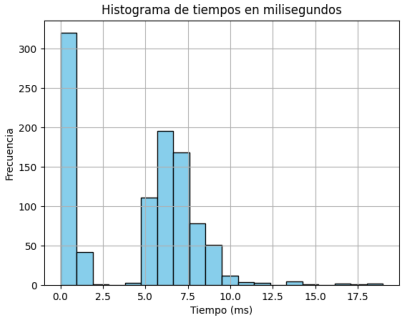

Figura 11: Sistema de caché particionado.

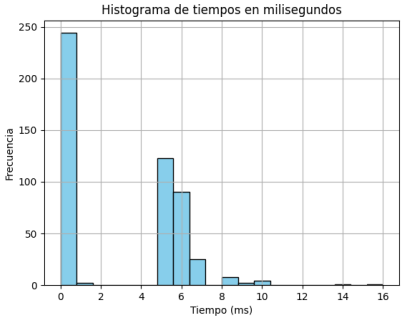

Figura 12: Sistema de caché replicado.

Como se puede observar en los gráficos, el sistema de caché particionado mostró consistentemente tiempos de respuesta más bajos en comparación con el centralizado y el replicado. Esta diferencia significativa se debe a la naturaleza de cada configuración.

El sistema de caché centralizado y el replicado funcionan de manera similar en el sentido de que ambos actúan como un solo contenedor para gestionar todas las solicitudes. Esto significa que, independientemente de la carga, todo el trabajo recae en un único nodo,

lo que puede resultar en cuellos de botella y tiempos de respuesta más lentos cuando la demanda es alta.

Por otro lado, el sistema de caché particionado divide la carga entre múltiples nodos, lo que le permite distribuir eficazmente las solicitudes y evitar la acumulación de trabajo en un solo punto.

7.2. Tasa de Aciertos del Caché

También evaluamos la tasa de aciertos del caché para cada configuración de Redis. Esta métrica nos indica qué tan efectivamente el sistema de caché está sirviendo las solicitudes desde la memoria en lugar de acceder a la base de datos.

A continuación, se presentan los gráficos que muestran la tasa de aciertos del caché para cada configuración en diferentes niveles de carga:

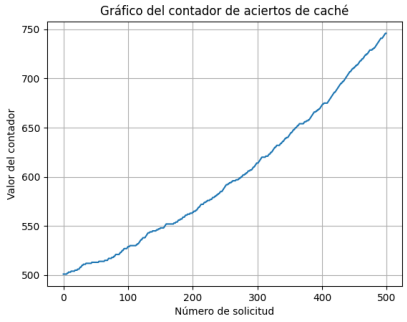

Figura 13: Sistema de caché centralizado.

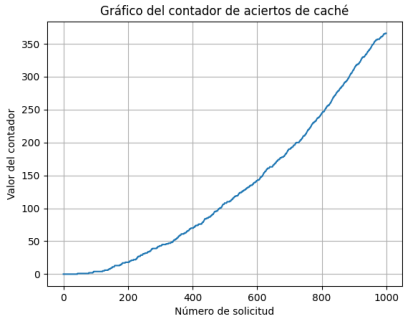

Figura 14: Sistema de caché particionado.

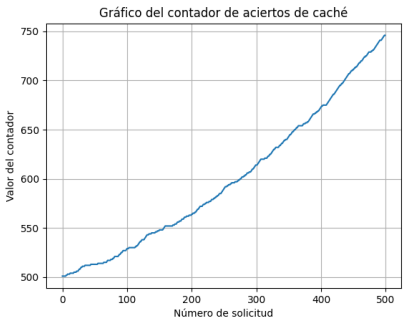

Figura 15: Sistema de caché replicado.

Como se puede ver en los gráficos, el sistema de caché particionado también mostró una tasa de aciertos del caché más alta en comparación con las otras dos configuraciones. Esto indica que el sistema particionado fue más efectivo en servir las solicitudes desde la caché en lugar de acceder a la base de datos, incluso en condiciones de carga pesada.

8. Conclusiones

En conclusión, los objetivos planteados al inicio del laboratorio se lograron con éxito. Se diseñó, implementó y evaluó un sistema de caché distribuido utilizando REDIS, que demostró ser una herramienta eficaz para mejorar el rendimiento y la escalabilidad de las aplicaciones distribuidas.

En cuanto al desempeño de los diferentes escenarios de caché, se observaron diferencias significativas. El caché centralizado presentó un rendimiento sólido y confiable en las pruebas sin tantos datos, pero carecía de la eficiencia y escalabilidad de los otros dos sistemas. El caché particionado, por otro lado, demostró ser el más eficiente en términos de rendimiento y escalabilidad, permitiendo una distribución equitativa de la carga. El caché replicado, aunque ofrecía una alta disponibilidad de datos, se comportaba de manera similar a un centralizado.

Dado el problema planteado por la Universidad Digna Pública (UDP) de gestionar eficientemente la distribución de salas para los diferentes cursos, se decidió utilizar el sistema de caché particionado. Esta elección se basó en su capacidad para manejar eficientemente una gran cantidad de solicitudes y distribuir la carga de manera equitativa entre los nodos. Además, se decidió utilizar la política de remoción LFU (Least Frequently Used) debido a su capacidad para manejar eficientemente las consultas que ocurren en momentos críticos. Un ejemplo de esto en el caso de la UDP sería si un al comienzo de mes se juntan todos los campos en un auditorio y durante el resto del mes se hacen una cantidad mayor de peticiones pero distribuidas en todas las facultades, el cache a momento de llenarse eliminaria las salas menos ocupadas y guardaria las mas ocupadas como el auditorio, lo cual ayudaria

a momento de comienzo de mes a tenerlo en cache.

Por otro lado, en el laboratorio solo se hace uso de la política de remoción LRU, ya que es la política por defecto que utiliza Redis (ver Referencia 3).

Durante la implementación y las pruebas, nos enfrentamos a diversos desafíos. Uno de los principales fue establecer la conexión adecuada entre los contenedores de Docker y manejar las diferentes formas de comunicación, como las conexiones API-REDIS, API-SERVER y SERVER-POSTGRESQL. Afortunadamente, pudimos superar estos obstáculos gracias a los recursos proporcionados en los videos que se encuentran en las referencias y los repositorios de cada software utilizado.

En resumen, este laboratorio demostró la importancia de un sistema de caché distribuido bien diseñado e implementado para mejorar el rendimiento y la escalabilidad de las aplicaciones distribuidas. A través de la experimentación y la evaluación, se pudo determinar la configuración óptima para el sistema de caché de la UDP, proporcionando una solución eficaz a su problema de distribución de salas.

9. Bibliografía

1. Redis. (2024). Scale with Redis Cluster. Recuperado el 17 de abril de 2024, de `https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/`

2. Redis. (2024). Redis replication. Recuperado el 17 de abril de 2024, de `https://redis.io/docs/latest/operate/oss_and_stack/management/replication/`

3. Redis. (2024). Key eviction. Recuperado el 17 de abril de 2024, de `https://redis.io/docs/latest/develop/reference/eviction/`

4. Real Python. (s. f.). How to Use Redis With Python. Real Python. Recuperado de `https://realpython.com/python-redis/`

5. gRPC. (2023). Quick start — Python — gRPC. gRPC Documentation. Recuperado el 18 de abril de 2024, de `https://grpc.io/docs/languages/python/quickstart/`

6. PostgreSQL Tutorial. (s. f.). PostgreSQL Python. PostgreSQL Tutorial. Recuperado el 18 de abril de 2024, de `https://www.postgresqltutorial.com/postgresql-python/`

7. PartTimeLarry. (s. f.). Python and Redis Tutorial - Caching API Responses. YouTube. Recuperado de `https://www.youtube.com/watch?v=_8lJ5lp8P0U&ab_channel=PartTimeLarry`

8. OpenAI. (2022). ChatGPT 3.5 [Modelo de lenguaje]. `https://openai.com/chatgpt`

9. NASA. (s. f.). Kepler Exoplanet Search Results. Recuperado el 17 de abril de 2024, de Kaggle.