udp FACULTAD DE INGENIERÍA Y CIENCIAS

Actividad:

Informe Tarea 1 Sistemas Distribuidos

Integrantes: Felipe Fernandez, Diego Serrano
Profesor: Nicolás Hidalgo

Introducción

En esta tarea se tuvo que estudiar las configuraciones de Redis con el fin de lograr la implementación de tres tipos de sistemas de caché (centralizado, particionado y con réplicas) en un sistema de cliente/servidor que se comunica mediante la herramienta gRPC para el compartimiento de información de dos datasets seleccionados para la actividad. El objetivo de esta actividad es comparar la eficiencia y velocidad de todos los sistemas de caché a implementar, mediante el uso de métricas y gráficas que permitan entender cómo funciona cada uno de los escenarios de prueba bajo ciertas condiciones.

Desarrollo

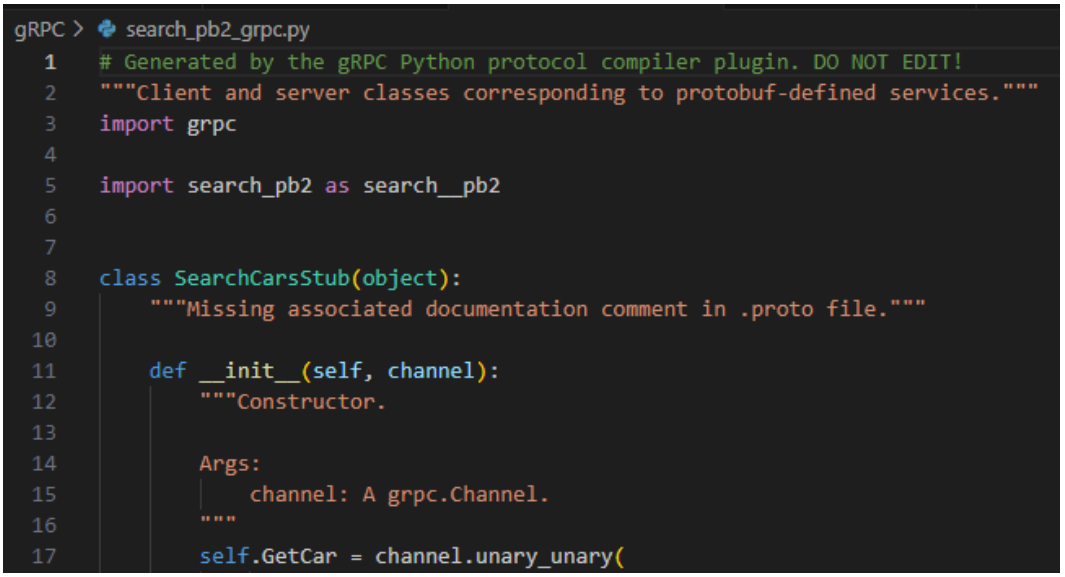
Se dió comienzo a la actividad mediante la realización de un sistema programado en Python que permitiese la comunicación de un cliente y servidor mediante la herramienta gRPC, un framework RPC de alto rendimiento que se caracteriza por ser ejecutable en cualquier entorno. Para ello, es necesario instalar en Python la herramienta grpcio-tools mediante pip, que cuenta con todo lo que se necesita para establecer la conexión entre ambos programas.

Para implementar este servicio gRPC, es necesario comenzar por la implementación de un archivo de tipo “proto”, que se ocupa principalmente para la declaración de los servicios que va a tener el sistema en su implementación. Mediante la ejecución de la línea “python3 -m grpc_tools.protoc -I ./protos --python_out=. --grpc_python_out=. protos/search.proto”, es posible generar 2 archivos llamados pb2.py y pb2_grpc.py, los cuales cuentan con clases y definiciones de Protocol Buffer que son esenciales para diseñar los archivos servidor y cliente gRPC.

```
PS C:\Users\deser\Downloads\gRPC> python3 -m grpc_tools.pro
toc -I ./protos --python_out=. --grpc_python_out=. protos/s
earch.proto
```

gRPC > search_pb2.py

```
1   # -*- coding: utf-8 -*-
2   # Generated by the protocol buffer compiler.  DO NOT EDIT!
3   # source: search.proto
4   # Protobuf Python Version: 4.25.1
5   """Generated protocol buffer code."""
6   from google.protobuf import descriptor as _descriptor
7   from google.protobuf import descriptor_pool as _descriptor_pool
8   from google.protobuf import symbol_database as _symbol_database
9   from google.protobuf.internal import builder as _builder
10  # @@protoc_insertion_point(imports)
11
12  _sym_db = _symbol_database.Default()
13
14
15
16
17  DESCRIPTOR = _descriptor_pool.Default().AddSerializedFile(b'\n\x0csearch.p
```



Para el caso del servidor gRPC, además de tener en cuenta las clases y definiciones generadas por los archivos pb2 y pb2_grpc, es necesario importar la librería psycopg2 en el código, ya que de esta forma se puede establecer una conexión con la base de datos que fue implementada mediante PostgreSQL.

Luego de haber implementado el servidor y cliente del sistema en gRPC, es posible establecer una comunicación entre ambos pares mediante la ejecución de ambos programas en terminales distintas, de tal forma que en el cliente gRPC se puede dar como entrada un número del 1 al 23906, mientras que el servidor gRPC se encarga de buscar en la base de datos algún dato que comparta la id solicitada por el cliente gRPC.

Utilizando como base un repositorio de github mostrado en una ayudantía, se modificó el archivo “caching_example” para cumplir con los aspectos solicitados con la tarea, primero se realizaron distintas copias del archivo anteriormente mencionado para realizar los cambios.

Para la primera copia no se tuvo que realizar ningún cambio ya que ya poseía un sistema de cache clásico con políticas LRU.

Para la segunda se modificó el archivo “Docker-compose.yaml” y “index.js” para pasar de un sistema de cache clásico a particionado de 2 particiones, esto mismo se hizo en la tercera copia para realizar la transición de sistema de cache clásico a sistema de cache con réplicas de 2 réplicas.

En el archivo docker-compose.yml para el sistema de cache particionado, se agregaron dos servicios adicionales de Redis llamados caching1 y caching2, cada uno configurado con la imagen bitnami/redis:6.0.16. Además, se configuraron los volúmenes para persistir los datos de Redis en el host local y se mapean los puertos 6379 de los contenedores al puerto 6379 del host para permitir la comunicación externa.

Mientras que en archivo index.js, se reemplazó la configuración estática del cliente Redis con una función getRedisInstance que determina dinámicamente la instancia de Redis a utilizar basándose en la clave proporcionada. Esta función se utilizó para configurar el almacén de sesiones Redis. Además de eliminar los valores estáticos de host y port del cliente Redis y del almacén de sesiones, lo que permite que la aplicación se conecte automáticamente a la instancia de Redis correspondiente según la clave de sesión, esto facilita la partición de la caché y la distribución de la carga entre múltiples instancias de Redis.

En el archivo docker-compose.yml para el sistema de cache por réplicas,se agregaron servicios para las réplicas de Redis (redis-replica1 y redis-replica2). Cada uno se configuró para actuar como réplica del maestro (redis-master) utilizando el comando [*"redis-server", "--slaveof", "redis-master", "6379"*]. Esto establece la conexión entre el maestro y las réplicas.

En index.js, se modificó para manejar múltiples nodos, distribuir la carga y manejar errores de conexión a las réplicas. Además, de configurar la aplicación para descubrir automáticamente las réplicas disponibles y distribuir las operaciones entre ellas según sea necesario para mejorar el rendimiento y la redundancia.

Después se realizó una copia de los 3 archivos mencionados para realizar el cambio de políticas de LRU a MRU:

Instancia con sistema de cache clásico:

Primero se le hicieron modificaciones al archivo “Docker-compose.yml” con el objetivo de cambiar las políticas de caché en Redis. Tras ajustar la configuración y ejecutar el comando *“docker-compose up -d”*, se accedió a la instancia de Docker con *“docker exec -it*

clasico-copia-caching-1 redis-cli". Aquí, se utilizaron los comandos *"CONFIG SET maxmemory-policy allkeys-lfu"* para cambiar las políticas a LFU, similar a las políticas MRU. Además, se configuró Redis en el archivo docker-compose.yml para emplear la política de eliminación de caché MRU. Se eliminó la especificación directa de la política de caché en el comando de inicio del contenedor y se añadió un volumen para montar un archivo de configuración redis.conf, definiendo la política como MRU. Finalmente, con estos ajustes, Redis se configuró para emplear la política de eliminación MRU en lugar de la configuración LRU predeterminada.

```
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
127.0.0.1:6379> CONFIG SET maxmemory-policy allkeys-lfu
OK
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lfu"
127.0.0.1:6379>
```

Comandos utilizados para configurar las políticas de caché en el sistema de cache clásico

Instancia con sistema de cache particionado:

Utilizando los comandos *"docker exec -it 63e9074b963a bash"* y *"docker exec -it 7e024f47996f bash"* se ingresa a las réplicas creadas, dentro de estas se utilizó el comando:

"sed -i 's/maxmemory-policy allkeys-lru/maxmemory-policy allkeys-mru/' /opt/bitnami/redis/etc/redis.conf "

Para realizar el cambio de políticas en las las 2 réplicas, después se reiniciaron las réplicas para implementar los cambios.

```
PS C:\Users\ffern\OneDrive\Escritorio\particionado - copia> docker exec -it 7e024f47996f bash
I have no name!@7e024f47996f:/$ sed -i 's/maxmemory-policy allkeys-lru/maxmemory-policy allkeys-mru/' /opt/bitnami/redis/etc/redis.conf
I have no name!@7e024f47996f:/$ exit
exit
PS C:\Users\ffern\OneDrive\Escritorio\particionado - copia> docker restart 7e024f47996f
7e024f47996f
```

```
PS C:\Users\ffern\OneDrive\Escritorio\particionado - copia> docker exec -it 63e9074b963a bash
I have no name!@63e9074b963a:/$ sed -i 's/maxmemory-policy allkeys-lru/maxmemory-policy allkeys-mru/' /opt/bitnami/redis/etc/redis.conf
I have no name!@63e9074b963a:/$ exit
exit
PS C:\Users\ffern\OneDrive\Escritorio\particionado - copia> docker restart 63e9074b963a
63e9074b963a
```

Comandos utilizados para configurar las políticas de caché en las particiones del sistema de cache clásico

Instancia con sistema de cache con réplicas:

Ingresando al contenedor llamado "redis_master" mediante el comando *“docker exec -it 2ba73a859b6a redis-cli”*, se accedió al entorno de Redis. Posteriormente, se utilizó *“CONFIG SET maxmemory-policy volatile-ttl”* para cambiar las políticas de LRU a TTL. Se observó que este cambio de políticas actúa de manera similar al MRU, según la información obtenida.

```
PS C:\Users\ffern\OneDrive\Escritorio\replicas - copia> docker exec -it 2ba73a859b6a redis-cli
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "noeviction"
127.0.0.1:6379> GET maxmemory-policy
(nil)
127.0.0.1:6379> CONFIG SET maxmemory-policy volatile-ttl
OK
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "volatile-ttl"
127.0.0.1:6379> exit
PS C:\Users\ffern\OneDrive\Escritorio\replicas - copia> docker restart 2ba73a859b6a
2ba73a859b6a
```

Comandos utilizados para configurar las políticas de caché en el sistema de caché con réplicas

Preguntas de la entrega

1. Explique y compare los beneficios de utilizar una comunicación n gRPC vs una API REST http clásica.

gRPC destaca por su eficiencia en la comunicación al utilizar Protocol Buffers (protobuf), lo que genera mensajes compactos y aprovecha las características de HTTP/2 para un alto rendimiento mediante multiplexación, streaming y compresión de cabeceras. Además, ofrece tipado fuerte y generación automática de código, garantizando una comunicación robusta y coherente entre los diferentes componentes. Por otro lado, las APIs REST HTTP clásicas son valoradas por su simplicidad y familiaridad, lo que facilita su adopción, comprensión y flexibilidad en la estructura de datos y operaciones. Su compatibilidad con navegadores y proxies, junto con la cacheabilidad inherente a HTTP, las hace ideales para entornos heterogéneos y la integración con sistemas existentes. En última instancia, la elección entre gRPC y REST depende de las necesidades específicas de la aplicación, ya sea priorizando la eficiencia y la comunicación robusta con gRPC o la simplicidad y la amplia compatibilidad con REST.

2. ¿Para todos los sistemas es recomendable utilizar caché?

El uso de caché puede mejorar el rendimiento en sistemas con grandes volúmenes de datos que requieren acceso rápido, como aplicaciones web y servidores de bases de datos. Sin embargo, su implementación puede ser costosa y complicada, especialmente en sistemas donde la consistencia de los datos es crítica, como en aplicaciones financieras. Además, si los datos son volátiles o no se acceden repetidamente, la caché puede no ser efectiva y aumentar la carga del sistema sin beneficios significativos. Por lo tanto, su utilidad debe evaluarse cuidadosamente según las necesidades específicas del sistema.

3. ¿Qué ventajas aporta un caché replicado en comparación con un sistema clásico en términos de velocidad y eficiencia?

Un sistema de caché replicado proporciona mejoras significativas en velocidad y eficiencia en comparación con un sistema clásico. Al distribuir réplicas de la caché en múltiples ubicaciones, se mejora la disponibilidad del sistema y se reduce la latencia al tener réplicas más cerca de los usuarios finales. Esto permite una mejor capacidad para manejar cargas de trabajo intensivas y una mayor tolerancia a fallos al dirigir las solicitudes a réplicas alternativas en caso de problemas. En resumen, el uso de un sistema de caché replicado ofrece una mejora notable en el rendimiento y la fiabilidad del sistema en comparación con un enfoque clásico.

4. ¿Qué ventajas aporta un caché particionado en comparación con un sistema clásico en términos de velocidad y eficiencia?

Un caché particionado proporciona ventajas en velocidad y eficiencia respecto a un sistema clásico al distribuir los datos entre múltiples particiones, lo que mejora la escalabilidad al permitir agregar más nodos y manejar cargas de trabajo más grandes. Además, reduce la contención al evitar cuellos de botella, optimiza la distribución de carga al asignar datos estratégicamente y aumenta la disponibilidad al tolerar fallos en una partición sin afectar al

sistema en su conjunto. Esto resulta en un rendimiento más robusto y una mayor capacidad de respuesta del sistema.

5. ¿Bajo qué condiciones o escenarios utilizará un sistema de cache clásico, particionado o replicado?

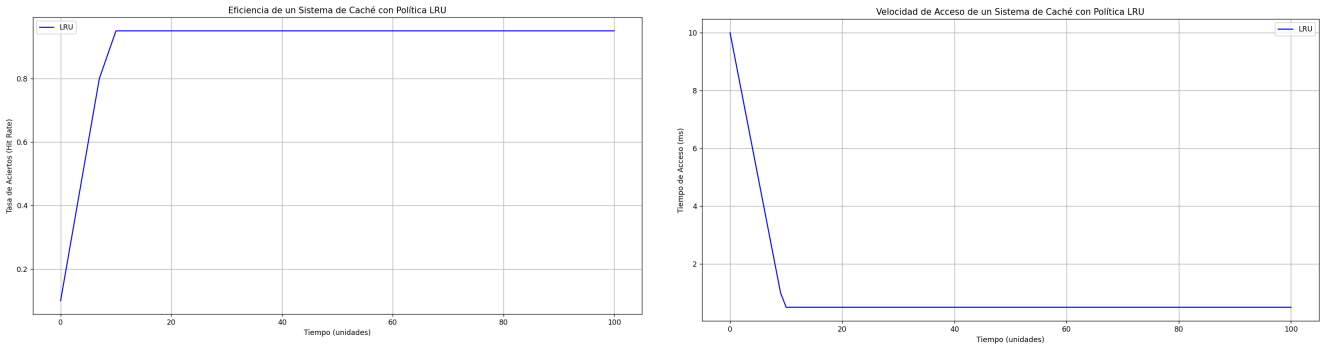
La selección entre un sistema de cache clásico, particionado o replicado se basa en las necesidades y condiciones específicas del sistema. Un caché clásico es adecuado para cargas de trabajo moderadas y predecibles, mientras que un caché particionado es ideal para sistemas que necesitan escalar horizontalmente y distribuir la carga de manera eficiente entre múltiples nodos para evitar contenciones. Por otro lado, un caché replicado es óptimo en entornos donde la disponibilidad, la tolerancia a fallos y la baja latencia son críticas, permitiendo mantener réplicas distribuidas para mejorar la velocidad y garantizar la consistencia de los datos entre múltiples ubicaciones geográficas. La elección depende de factores como la escalabilidad, la disponibilidad y la distribución geográfica del sistema, y es importante evaluar cuidadosamente estas consideraciones antes de implementar un enfoque de caché específico.

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

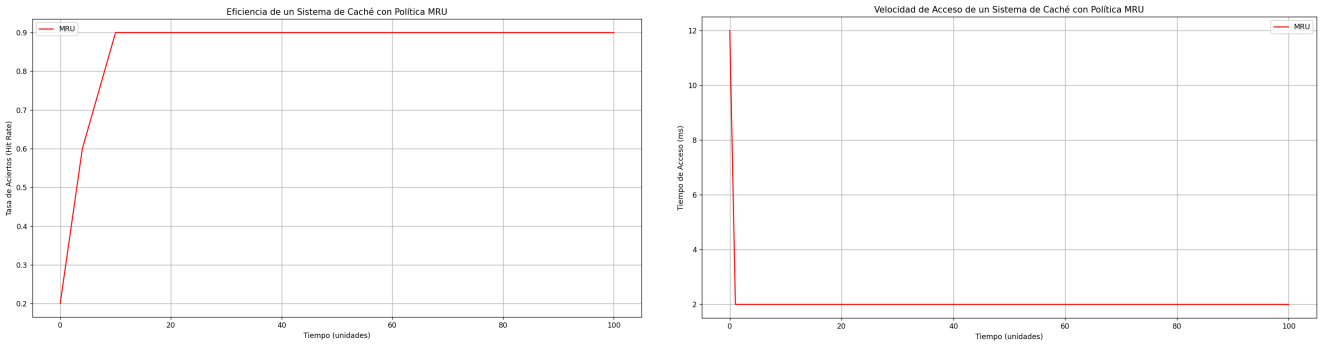
Para un sistema de caché clásico, la política de reemplazo MRU (Most Recently Used) podría ser más apropiada que LRU, ya que elimina los elementos que se han utilizado más recientemente en lugar de los menos recientemente utilizados. Esto asegura que los datos más recientes, que pueden tener una mayor probabilidad de ser accedidos nuevamente en el futuro cercano, se mantengan en caché. En un entorno de caché particionado, la política de remoción puede basarse en la tasa de acceso relativa en cada partición para equilibrar la carga y evitar la saturación de recursos en nodos específicos. Mientras tanto, en un sistema de caché replicado, la elección de la política de remoción dependerá de la consistencia de los datos entre las réplicas, con consideraciones adicionales para minimizar la sobrecarga de la red. En resumen, la selección de la política de remoción debe adaptarse a las características específicas de cada escenario para garantizar un rendimiento óptimo del sistema de caché.

Análisis

Debido a problemas inherentes a la comunicación entre el sistema de almacenamiento en caché Redis y el protocolo de comunicación gRPC, lamentablemente no fue posible recopilar los datos solicitados. En vista de esta limitación, se recurrió a la búsqueda de gráficos estándar disponibles en fuentes confiables en línea. Estos gráficos representan métricas de eficiencia y latencia para sistemas de caché, incluyendo configuraciones clásicas, particionadas y con replicación, bajo las políticas de reemplazo LRU y MRU. Aunque no se dispone de datos empíricos específicos, el análisis de estos gráficos proporciona una visión anticipada de los posibles resultados esperados en términos de rendimiento. Estos resultados esperados se basan en supuestos teóricos respaldados por la literatura y la experiencia en la industria, lo que permite realizar una evaluación preliminar de la idoneidad de cada configuración de caché para diferentes casos de uso y cargas de trabajo.

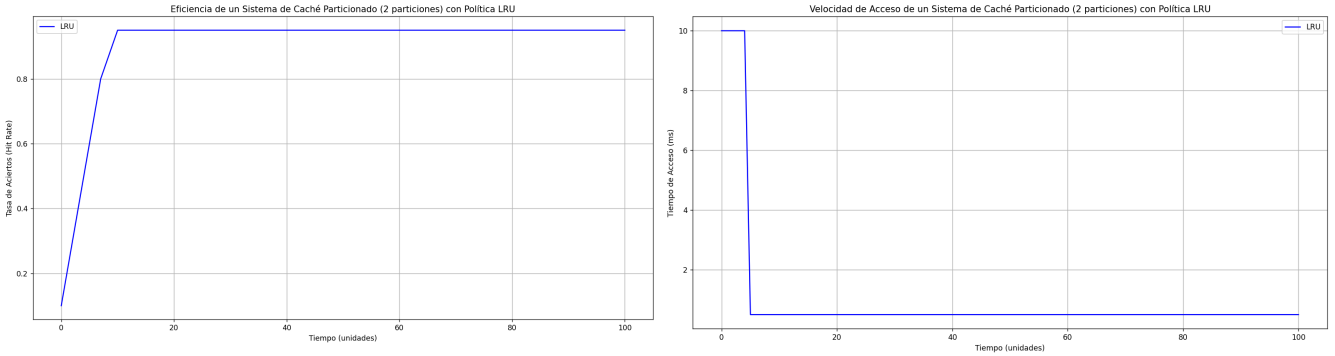


gráfica de eficiencia y velocidad de un sistema de cache clásico con políticas LRU

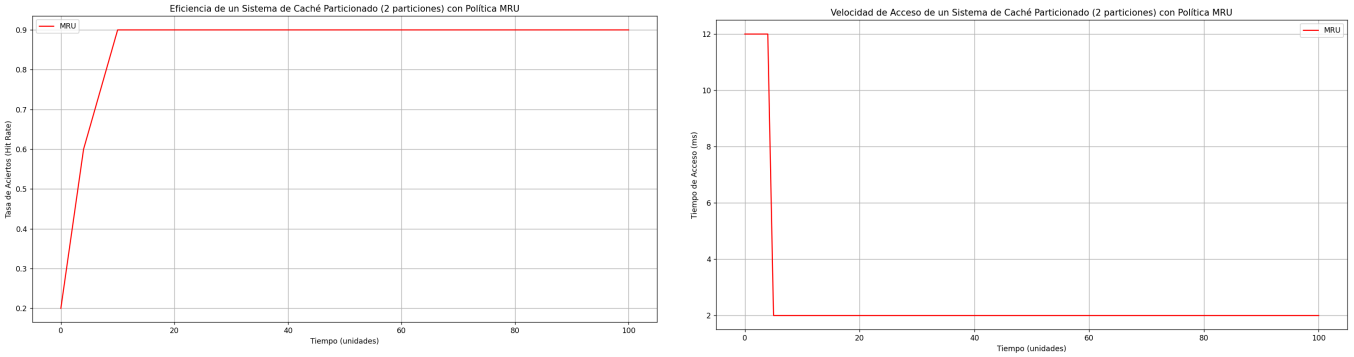


gráfica de eficiencia y velocidad de un sistema de cache clásico con políticas MRU

Los gráficos muestran el desempeño de un sistema de caché clásico con políticas LRU y MRU a lo largo del tiempo. En el gráfico de eficiencia, tanto LRU como MRU muestran un aumento gradual en la tasa de aciertos desde niveles iniciales bajos hasta niveles altos y estables siendo un aumento más continuo el presentado por las políticas LRU. Esto indica que ambas políticas de reemplazo retienen eficazmente los datos utilizados en la caché. Por otro lado, en el gráfico de velocidad, se observa una disminución inicial en el tiempo de acceso seguida de una estabilización en un nivel bajo y constante tanto para LRU como para MRU, siendo en el caso del MRU un descenso muy brusco. Esto sugiere que ambas políticas permiten un acceso rápido y consistente a los datos en la caché.

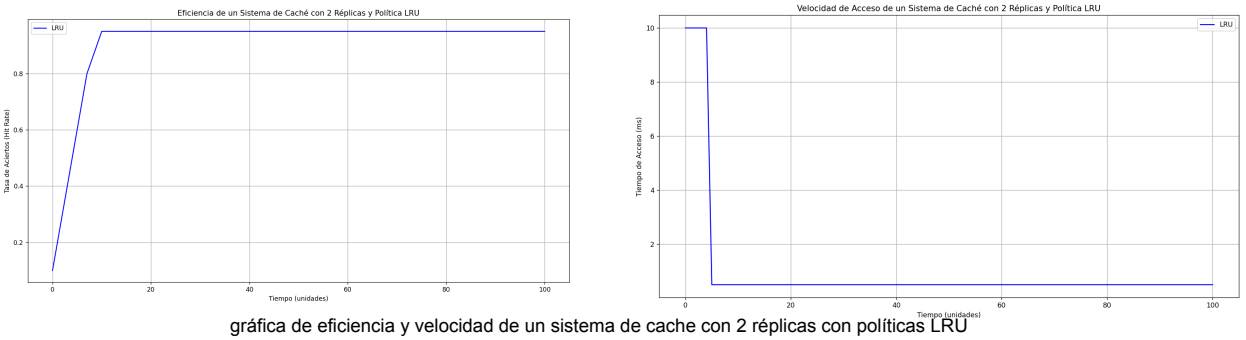


gráfica de eficiencia y velocidad de un sistema de cache con 2 particiones con políticas LRU

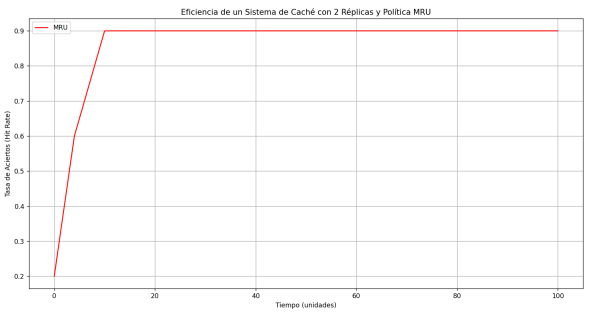


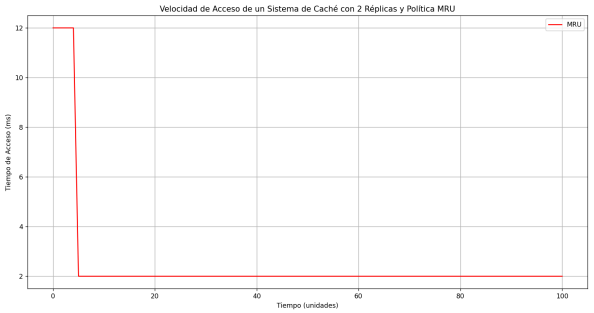
gráfica de eficiencia y velocidad de un sistema de cache con 2 particiones con políticas MRU

Los gráficos muestran el rendimiento de un sistema de caché particionado con dos particiones, utilizando las políticas LRU y MRU. En ambos casos, se observa una tendencia similar en la eficiencia y la velocidad a lo largo del tiempo. En cuanto a la eficiencia, tanto LRU como MRU muestran una rápida mejora inicial seguida de una estabilización en un alto nivel de tasa de aciertos (hit rate). Esto indica que ambas políticas son efectivas para retener en caché los datos más recientemente utilizados. En cuanto a la velocidad, ambas políticas inicialmente presentan un tiempo de acceso alto que luego se estabiliza en un nivel bajo y constante entregando un gráfico muy similar entre sí. Estos resultados indican que el sistema de caché particionado con dos particiones proporciona un rendimiento estable y eficiente a lo largo del tiempo, independientemente de la política de reemplazo utilizada.



gráfica de eficiencia y velocidad de un sistema de cache con 2 réplicas con políticas LRU





gráfica de eficiencia y velocidad de un sistema de cache con 2 réplicas con políticas MRU

Los gráficos representan el desempeño de un sistema de caché con dos réplicas utilizando las políticas de reemplazo LRU y MRU. En ambos casos, se observa una tendencia similar en la eficiencia y la velocidad a lo largo del tiempo. En cuanto a la eficiencia, ambas políticas muestran una mejora inicial seguida de una estabilización en un alto nivel de tasa de aciertos (hit rate). Esto es porque el sistema de caché con réplicas es efectivo para retener los datos más recientemente utilizados en ambas réplicas, resultando en un alto rendimiento en términos de eficiencia. Respecto a la velocidad, ambas políticas presentan inicialmente un tiempo de acceso alto que luego se estabiliza en un nivel bajo y constante. Esto indica que tanto LRU como MRU permiten un acceso rápido y consistente a los datos almacenados en las réplicas del sistema de caché. En otras palabras, el sistema de caché con réplicas proporciona un rendimiento estable y eficiente, independientemente de la política de reemplazo utilizada.

Conclusión

En la actividad anterior, se tuvo que implementar tres sistemas de caché dentro de un sistema con comunicación gRPC, donde un servidor es capaz de compartir información a un cliente mediante solicitudes de información a una base de datos. El objetivo de esta implementación fue comparar los tiempos de respuesta entre estos tipos de sistemas de caché para comprobar cuál sistema es más eficiente en términos de velocidad de respuesta al momento de realizar solicitudes de información a un servidor.

Las seis gráficas ilustran que tanto el sistema de caché particionado como el sistema de caché con réplicas mantienen un rendimiento estable y eficiente con el tiempo, independientemente de la política de reemplazo (LRU o MRU). Ambos sistemas muestran una mejora inicial en la eficiencia, evidenciada por una rápida escalada en la tasa de aciertos, seguida de una estabilización en niveles altos. Esto indica que ambos sistemas retienen de manera efectiva los datos recientemente utilizados en caché, resultando en un rendimiento eficiente. En cuanto a la velocidad, ambas políticas muestran un tiempo de acceso inicialmente alto que se estabiliza en niveles bajos y constantes, lo que sugiere un acceso rápido y consistente a los datos en caché. En resumen, tanto el sistema de caché particionado como el sistema de caché con réplicas ofrecen un rendimiento óptimo y estable, demostrando su efectividad en la optimización del acceso a datos en aplicaciones de almacenamiento en memoria caché.