udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA

Facultad de Ingeniería
Escuela de Informática y Telecomunicaciones

Tarea 1: Sistema de Cache Distribuido
Sistemas Distribuidos

Profesor: Nicolás Hidalgo
Ayudante: Cristian Villavicencio, Joaquín Fernandez, Nicolás Núñez y Felipe Ulloa
Integrantes: Vicente Gonzales y Santiago Larraín

Índice

1. Introducción **2**

2. Desarrollo **2**
2.1. Cache Casero 2
2.2. Memcached 3
2.3. Sin Caché 3
2.4. Pruebas Comparativas 4
2.5. Investigación y Análisis de las Características Avanzadas de Memcached 6
2.6. Preguntas 7

3. Conclusión **8**

4. Documentación **8**

1 | Introducción

En la era actual de aplicaciones web dinámicas y sistemas distribuidos, la optimización del rendimiento y la eficiencia son fundamentales. Uno de los componentes clave para mejorar el rendimiento de las aplicaciones es la implementación de sistemas de caché. Estos sistemas permiten almacenar temporalmente datos en memoria para acelerar el acceso y reducir la carga en bases de datos u otras fuentes de información.

En este trabajo, se analiza y compara el rendimiento de un sistema de caché casero con la conocida solución de Memcached, un sistema de almacenamiento en caché en memoria distribuido de código abierto.

El objetivo de este trabajo es proporcionar una visión integral sobre las ventajas y desventajas de implementar un sistema de caché casero en comparación con una solución establecida como Memcached o incluso la opción de no implementar uno. Se explorará cómo estas decisiones pueden influir en el rendimiento y la eficiencia de una aplicación.

2 | Desarrollo

2.1. Cache Casero

Existen tres partes importantes que deben explicarse dentro del caché casero: los nodos maestros y esclavos, encargados de balancear las cargas de los nodos esclavos y almacenar los datos, y el buscador, que es la interfaz que interactúa directamente con el servidor de caché para recuperar, insertar o eliminar datos.

Como se mencionó previamente, se utilizan dos tipos de nodos, maestro y esclavo. Los primeros se encargan de balancear las cargas para que el trabajo se distribuya de manera uniforme. Se utiliza el algoritmo de Consistent Hashing de la librería uhashring. En el nodo esclavo se utiliza un algoritmo LRU para gestionar el almacenamiento en caché. Si un nodo alcanza su capacidad máxima, se elimina el elemento menos recientemente utilizado para dar espacio a nuevos elementos. Los nodos presentan cinco funciones con las cuales se trabajan. Estas son:

- Registro y Desregistro de Nodos: Los nodos esclavos pueden registrarse y desregistrarse en el nodo maestro.
- Put (Inserción): Inserta un elemento en el caché. Si el caché en un nodo esclavo alcanza su límite, se elimina el elemento menos recientemente utilizado.
- Get (Recuperación): Recupera un elemento del caché si está presente. Actualiza el estado LRU del elemento accedido.
- Remove (Eliminación): Elimina un elemento del caché si está presente.

El buscador es la interfaz que interactúa directamente con el servidor de caché para recuperar, insertar o eliminar datos. Utiliza gRPC para enviar solicitudes y recibir respuestas del servidor de caché. Las consultas se realizan sobre un archivo JSON.

En el archivo `search.py`, se realizaron los siguientes cambios. Se modificó la forma en que se miden los tiempos para que ambos se puedan medir de una manera más precisa, utilizando el tiempo transcurrido. Además, se agregó una opción para habilitar o deshabilitar el caché y se añadió una opción para cambiar la distribución de las claves relacionadas con una consulta para las simulaciones. El código resultante se encuentra disponible en GitHub y está comentado al final del informe.

2.2. Memcached

Memcached es un sistema de almacenamiento en caché de objetos en memoria distribuido, gratuito y de código abierto, de alto rendimiento y de naturaleza genérica, pero diseñado para su uso en la aceleración de aplicaciones web dinámicas aliviando la carga de la base de datos.

Memcached es un almacén de claves y valores en memoria para datos arbitrarios pequeños (strings, objects) generados a partir de resultados de consultas a la base de datos, llamadas a la API o renderización de páginas.

Memcached es simple pero poderoso. Su diseño sencillo facilita la implementación rápida, la facilidad de desarrollo y resuelve muchos problemas relacionados con las cachés de datos grandes. Su API está disponible para la mayoría de los lenguajes de programación populares.

Para trabajar con Memcached, se hará a través de la imagen de Docker:

```
docker run --name my-memcache -d memcached
```

A Memcached se le configuró un sistema de liberación de caché mediante Python bajo el criterio de uso, se muestra la implementación en el siguiente código:

```python
import memcache
import time
import subprocess

ap = memcache.Client(['172.17.0.2:11211'])

def run_gather(access_counter, lock, min):
    while True:
        if access_counter.keys():
            break
    while True:
        with lock:
            for key, count in access_counter.items():
                if count < min:
                    ap.delete("id_"+key)
                    print("removed item: id_"+key)
                else:
                    print("Not removed: id_"+key)
            access_counter[key] = 0
        time.sleep(8)
```

Listing 1: Sistema de liberación de caché

2.3. Sin Caché

Para realizar las búsquedas sin el uso de caché, se restringe el acceso de lectura y escritura al caché, lo que obliga a que las consultas se realicen directamente a la base de datos (`cars.json`). Esto nos permite realizar las búsquedas pero mantiene el retraso de estas.

Para el desarrollo de las búsquedas sin caché, se trabajó en el mismo archivo `search.py`, donde se agregaron algunos condicionales con los que se restringieron las funcionalidades.

2.4. Pruebas Comparativas

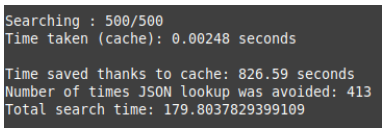

Figura 1: Resultados búsqueda con cache y una distribución normal.

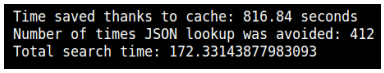

Figura 2: Resultados búsqueda con Memcached y una distribución normal.

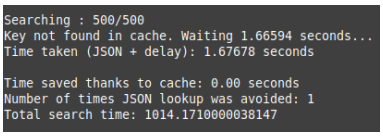

Figura 3: Resultados búsqueda sin cache y una distribución normal.

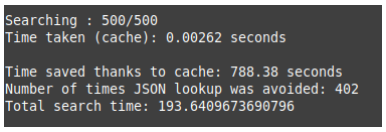

Figura 4: Resultados búsqueda con cache y una frecuencia 1.

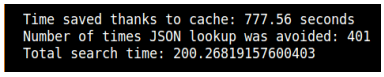

Figura 5: Resultados búsqueda con Memcached y una frecuencia 1.

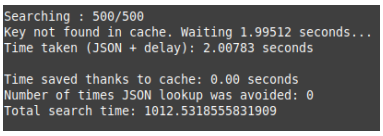

Figura 6: Resultados búsqueda sin cache y una frecuencia 1.

Los datos obtenidos en las capturas previamente mostradas se resumen y organizan en las siguientes tablas, las cuales se dividieron en 2 partes debido a problemas con el formato:

	Cantidad de búsquedas
Búsqueda con caché y distribución normal	500
Búsqueda con Memcached y distribución normal	500
Búsqueda sin caché y distribución normal	500
Búsqueda con caché y frecuencia 1	500
Búsqueda con Memcached y frecuencia 1	500
Búsqueda sin caché y frecuencia 1	500

Tabla 1: Tabla de resultados de las búsquedas con caché casero, Memcached y sin caché parte 1.

Tiempo ahorrado debido al caché	Número de vistas al JSON evitadas	Tiempo total de búsqueda
826.59 segundos	413	179.803 segundos
816.84 segundos	412	172.331 segundos
0 segundos	1	1014.171 segundos
788.38 segundos	402	193.640 segundos
777.56 segundos	401	200.268 segundos
0 segundos	0	1012.531 segundos

Tabla 2: Tabla de resultados de las búsquedas con caché casero, Memcached y sin caché parte 2.

De los datos observados es posible observar, en primera instancia, lo más esperado, la distribución de keys normal tiene tendencia a ser más rápida que la regular al aplicar cache, con datos más concentrados las chances de consultar una id en cache aumentan. Luego, al no utilizar cache ambas distribuciones tardan lo mismo, sin cache son esencialmente la misma simulación. Memcached mostró ser ligeramente más rápido para la distribución regular y poco más lento en el caso de la distribución normal. Consideramos que Memcached probablemente sería más rápido por defecto en ambos casos en sets más grandes, pues si bien tardó más en la distribución normal fue por un margen más pequeño que en la regular. Paralelamente a la idea de sets más grandes, la implementación de la funcionalidad de eliminación de registros en cache por regularidad de

consultas no se incluyó al concluir que, en este rango de sets que no llega a colapsar el sistema de Memcached, sólo alentaría el proceso de simulación.

2.5. Investigación y Análisis de las Características Avanzadas de Memcached

- **Balance de Carga:**

  Memcached proporciona un mecanismo de balance de carga incorporado. Utiliza un algoritmo de hash consistente para distribuir las claves y los datos entre los nodos del clúster. Esto garantiza que las solicitudes se distribuyan de manera uniforme entre los servidores de caché, evitando que un nodo se convierta en un cuello de botella. En el sistema casero, el balance de carga depende de la implementación específica y podría requerir más esfuerzo para lograr una distribución equitativa de la carga.

- **Distribución:**

  Memcached admite la distribución de datos en clústeres mediante la técnica de hash consistente. Esto facilita la escalabilidad horizontal al agregar o quitar nodos del clúster. Además, Memcached ofrece bibliotecas y clientes en varios lenguajes de programación para facilitar la implementación y el acceso a los datos distribuidos. En el sistema casero, la distribución de datos entre las máquinas puede requerir una implementación personalizada y podría ser menos escalable.

- **Tolerancia a Fallos:**

  Memcached tiene la capacidad de manejar la tolerancia a fallos mediante la adición de réplicas de datos en varios nodos. Si un nodo falla, los clientes pueden dirigir sus solicitudes a las réplicas disponibles, lo que garantiza que los datos sigan siendo accesibles. La tolerancia a fallos en el sistema casero dependerá de su implementación específica y podría requerir más esfuerzo para lograr un alto grado de disponibilidad.

- **Expiración de Datos**: Memcached permite establecer un tiempo de expiración para los datos en caché. Esto es útil para garantizar que los datos obsoletos se eliminen automáticamente. En un sistema casero, se puede implementar una lógica de expiración de datos similar.

- **Particionamiento de Datos**: Memcached admite el particionamiento de datos en clústeres para distribuir la carga de manera eficiente. Esto puede ser emulado en un sistema casero dividiendo los datos en múltiples máquinas.

- **Estadísticas y Monitoreo**: Memcached proporciona estadísticas y métricas que pueden ser monitoreadas para el análisis de rendimiento y la solución de problemas. Un sistema casero podría implementar un sistema de monitoreo similar.

- **Coeficiente de Gini para Distribución de Carga:**

  El coeficiente de Gini es una métrica que se utiliza para medir la desigualdad en la distribución de datos. En el contexto de Memcached, se puede aplicar el coeficiente de Gini para evaluar el nivel de distribución de carga entre las máquinas del clúster. Cuanto más cercano a cero sea el coeficiente de Gini, más equitativa será la distribución de carga entre las máquinas. Esto proporciona una métrica cuantitativa para evaluar la eficiencia del balanceo de carga en Memcached y en el sistema casero.

2.6. Preguntas

1. ¿Para todos los sistemas es recomendable utilizar cache?

   No, no es recomendable utilizar caché en todos los sistemas. La utilidad de la caché depende de varios factores, incluyendo el tipo de aplicación, los patrones de acceso a datos y los recursos disponibles. En sistemas donde la velocidad y la eficiencia son críticas, como aplicaciones web con alta demanda de lectura de datos, la caché puede ser altamente beneficiosa. Sin embargo, en sistemas donde la coherencia de los datos en tiempo real es esencial o donde el almacenamiento en caché puede introducir complejidad innecesaria, la caché puede no ser adecuada.

2. ¿Qué ventajas aporta Memcached en comparación con nuestro sistema casero en términos de velocidad y eficiencia?

   Memcached ofrece varias ventajas en términos de velocidad y eficiencia en comparación con un sistema casero:

   - Memcached es altamente optimizado y está diseñado específicamente para almacenar en caché en memoria, lo que lo hace extremadamente rápido en la recuperación de datos.
   - Memcached es distribuido, lo que permite la escalabilidad horizontal para manejar grandes cargas de trabajo.
   - Memcached ofrece características avanzadas como el balance de carga y la tolerancia a fallos para garantizar la disponibilidad y el rendimiento.

3. Enumera y describe tres características avanzadas que Memcached ofrece que nuestro sistema casero no posee.

   a) **Balance de Carga**: Memcached utiliza un algoritmo de hash consistente para distribuir las solicitudes de manera uniforme entre los nodos, lo que garantiza un balance de carga eficiente. Nuestro sistema casero puede requerir más esfuerzo para lograr un balanceo de carga similar.

   b) **Distribución de Datos**: Memcached permite la distribución de datos en clústeres, lo que facilita la escalabilidad horizontal al agregar o quitar nodos. En nuestro sistema casero, la distribución de datos entre máquinas puede requerir una implementación personalizada.

   c) **Tolerancia a Fallos**: Memcached admite la tolerancia a fallos mediante la adición de réplicas de datos en varios nodos. Si un nodo falla, los clientes pueden dirigir sus solicitudes a las réplicas disponibles, lo que garantiza que los datos sigan siendo accesibles. Esta característica puede ser más compleja de implementar en un sistema casero.

4. ¿Cómo se podría mejorar nuestro sistema casero para que se acerque más a la robustez y funcionalidad de Memcached?

   Para mejorar nuestro sistema casero y hacerlo más robusto y funcional, podríamos considerar las siguientes mejoras:

   - Implementar técnicas de balance de carga para distribuir las solicitudes de manera uniforme entre las máquinas.
   - Agregar mecanismos de tolerancia a fallos, como la replicación de datos o la conmutación por error, para garantizar la disponibilidad de datos en caso de fallos.
   - Desarrollar un sistema de monitoreo y registro para realizar un seguimiento del rendimiento y solucionar problemas de manera proactiva.

5. ¿En qué situaciones considerarías apropiado utilizar nuestro sistema de caché casero en lugar de una solución como Memcached?

   La implementación de un sistema de caché casero podría ser apropiado en las siguientes situaciones:

   - Cuando se requiere una solución altamente personalizada para satisfacer necesidades específicas que Memcached no puede cubrir.

- En entornos de desarrollo o pruebas donde se necesita una solución rápida y de bajo costo para cachear datos.
- En casos donde los recursos son limitados.

3 | Conclusión

Se observó que Memcached ofrece ventajas ligeramente significativas en términos de velocidad y eficiencia en comparación con el sistema de caché casero. Esto podría deberse al hecho de que, debido a la cantidad de datos, las funciones de Memcached, como el balance de carga, la distribución de datos y la tolerancia a fallos, no se destacaron de manera evidente. Por lo tanto, en el caso de bases de datos pequeñas y no muy complejas, la implementación de un sistema de caché casero personalizado o Memcached puede ser igualmente viable en términos de velocidad y eficiencia.

Si comparamos los resultados de la aplicación de un caché casero y Memcached con un sistema de búsqueda sin la implementación de un caché, podemos observar que los primeros son más eficientes y veloces. Este resultado podría deberse a la persistencia del retraso presente en el sistema de búsqueda del caché casero (se utilizó el mismo código pero se modificó para restringir el acceso al caché) al momento de realizar la búsqueda sin cache.

En resumen, la elección entre un sistema de caché casero y Memcached debe basarse en las necesidades específicas del proyecto y las metas de rendimiento, considerando factores como la complejidad de implementación y la escalabilidad requerida.

4 | Documentación

- Repositorio Github con la tarea
- Imagen de Memcached de docker hub
- Pagina Oficial Memcached
- Wiki Oficial de Memcached
- Vídeo