



Universidad Diego Portales

FACULTAD DE INGENIERÍA Y CIENCIAS

TAREA N°1 SISTEMAS DISTRIBUIDOS

Pablo Sáez

Abril de 2023

Índice

1. Introducción	2
2. Aspectos necesarios para la implementacion	2
2.1. API de datos	2
2.2. backend para consultas	2
2.3. Configuración de contenedores	2
3. Implementación	4
3.1. Cache distribuido con índices	4
4. Análisis de cache	8
4.1. Política de remoción: allkeys-lru	8
4.2. Política de remoción: random	8
5. Configuración de contenedores	8
6. Conclusiones	9
7. link a video	9

1. Introducción

En el siguiente informe se detallará como un sistema de cache aumenta el rendimiento a la hora de pedir información hacia una API o cualquier sistema de manejo de información online de una manera practica. Se explicará la implementación de una demostración de como el uso de un sistema de cache como redis optimiza el tiempo de fetching de los datos.

2. Aspectos necesarios para la implementacion

En las siguientes secciones se explicará las herramientas y configuraciones necesarias para que el sistema funcione

2.1. API de datos

Para obtener los datos con los que se trabajarán se utilizará la API <https://aves.ninjas.cl/>. La cual entrega información sobre las aves en el territorio chileno.

De esta api se utilizaran 2 endpoints, los cuales son los siguientes:

- <https://aves.ninjas.cl/api/birds>.
- <https://aves.ninjas.cl/api/birds/uid>

En el primer endpoint la API devuelve todos los datos posibles en formato json, en cambio para el segundo endpoint la API devuelve la información del ave en especifico con el uid entregado.

2.2. backend para consultas

Para crear el sistema de backend el cual buscará la información, se utilizo el lenguaje python. Para este backend se crearón 2 archivos principales, los cuales son **uid.py** y **cliente.py**.

En uid.py se leerán todos los uids de la API y se almacenarán en un archivo llamado **uids.txt**, el cual nos ayudará mas adelante para simular las peticiones al cache.

En cliente.py estará la lógica de búsqueda distribuida hacia la api. Es decir, que cada vez que se busque un valor que no este en cache, este se guarde de manera ordenada en una de las 3 instancias de cache (redis). Esto a traves de la elección de un índice entre 0 y 29 dado una función matemática simple. Lo anterior se explica de la siguiente manera:

1. Se busca un dato de la API y se guarda en una variable
2. Luego dado su uid se calcula con una función matemática un id que es tara en un rango, los cuales son tres y son de 0-9 , 10-19 , 20-29.
3. Luego si el valor calculado esta en el primer rango irá a la primera instancia de redis y asi sucesivamente

Lo anterior garantiza con una cierta probabilidad de que la información se distribuirá equitativamente en los distintos caches utilizados.

2.3. Configuración de contenedores

Para simular los caches su utlizo la imagen de redis/bitname. Para ello se creo un archivo **docker-compose.yml** el cual estará adjuntado en github. Pero se explicará a continuación con secciones de código.

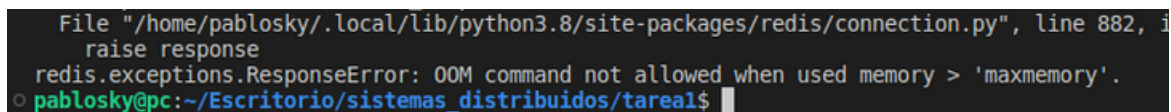
```
version: '3.9'
services:
  redis1:
    image: bitnami/redis:7.0.1
    ports:
```

```
- "6378:6379"
environment:
  - ALLOW_EMPTY_PASSWORD=yes
  - REDIS_MAXMEMORY_POLICY=allkeys-lru
command:
  [
    "redis-server",
    "--bind",
    "redis1",
    "--maxmemory",
    "1mb",
    "--maxmemory-policy",
    "allkeys-lru",
    "--protected-mode",
    "no"
  ]
```

En el archivo se ocupa la imagen de bitnami/redis en su versión 7.0.1. El código mostrado es solamente una instancia de redis llamada redis1, existen tres distintas las cuales ocuparan los puertos 6378,6380 y 6381 respectivamente. Para la conexión se configura para no necesitar credenciales de conexión, facilitando las peticiones (esto es con efectos de estudio y no es seguro para un ambiente de producción), luego también se le dice la política de remoción a utilizar por parte del servicio, en este caso se usará lru.

Además se dará una memoria de 1 mb. Esto ya que la población de datos que se tiene es acotada, para ver la efectividad del cache se ocupará el mínimo de memoria disponible en el cache.

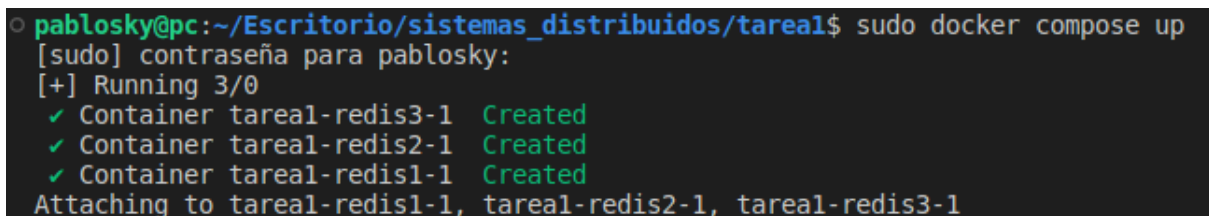
acotación: Se intentó usar un cache con memoria menor a 1mb sin embargo, la aplicación se cae y no deja almacenar ni siquiera 1 dato en el cache, arrojando el error mostrado en la figura 1 . Una vez configurado



```
File "/home/pablosky/.local/lib/python3.8/site-packages/redis/connection.py", line 882, i
raise response
redis.exceptions.ResponseError: OOM command not allowed when used memory > 'maxmemory'.
pablosky@pc:~/Escritorio/sistemas_distribuidos/tareal$
```

Figura 1: Error al ocupar menos de 1mb

el archivo para las tres instancias de redis se puede ejecutar con el comando: **sudo docker compose up** . Al ejecutar el comando se puede observar los contenedores corriendo con el comando **sudo docker ps**. Los resultados de ambos comandos se muestran a continuación en las figuras 2 y 3



```
pablosky@pc:~/Escritorio/sistemas_distribuidos/tareal$ sudo docker compose up
[sudo] contraseña para pablosky:
[+] Running 3/0
 ✓ Container tareal-redis3-1 Created
 ✓ Container tareal-redis2-1 Created
 ✓ Container tareal-redis1-1 Created
Attaching to tareal-redis1-1, tareal-redis2-1, tareal-redis3-1
```

Figura 2: comando: sudo docker compose up

Además con el comando se pueden observar los puertos que en el que están corriendo los servicios de redis en docker. Por default el puerto que utiliza redis es el 6379, sin embargo como se menciona antes se ocuparan los puertos 6378,6380 y 6381. Lo anterior se muestra en la figura 4

```
pablosky@pc:~$ sudo docker ps
CONTAINER ID        IMAGE
94945d0314eb       bitnami/redis:7.0.1
ab5c9a9276b7       bitnami/redis:7.0.1
01ef185bc339       bitnami/redis:7.0.1
pablosky@pc:~$
```

Figura 3: comando docker ps

PORTS	NAMES
0.0.0.0:6378->6379/tcp, :::6378->6379/tcp	tarea1-redis1-1
0.0.0.0:6380->6379/tcp, :::6380->6379/tcp	tarea1-redis2-1
0.0.0.0:6381->6379/tcp, :::6381->6379/tcp	tarea1-redis3-1

Figura 4: Puertos utilizados

3. Implementación

Una vez que ya tenemos todos los requisitos para ejecutar la demostración se procede a empezar el trabajo. Primeramente como se menciona en la sección de backend se creo el archivo uids.txt . Este archivo será importante para hacer las peticiones, ya que para simular el funcionamiento de un cache necesitamos buscar información, primeramente se preguntará si dicha información esta en cache, sino se pregunta directamente a la API.

A continuación se mostrarán secciones de código, el código completo lo puede encontrar en el github para esta tarea.

3.1. Cache distribuido con índices

Para lograr que el cache se distribuya se hicieron tres instancias de redis como se menciona en las secciones anteriores. para hacer la conexión desde el cliente simplemente se debe importar la librería redis para python.

```
import redis

# Conexión a las tres instancias de Redis
r1 = redis.Redis(host='localhost', port=6378, db=0)
r2 = redis.Redis(host='localhost', port=6380, db=0)
r3 = redis.Redis(host='localhost', port=6381, db=0)
```

En donde r1, r2 y r3 son las conexiones a las tres instancias distintas, cada una en su respectivo puerto

Luego se creo una función en la cual se pueda calcular el id del rango asociado al uid del dato. Para ello se implemento una función llamado encode_uid(uid), esta función recibe como parámetro un uid y retorna un valor entre 0 y 29.

```
def encode_uid(uid):
    # Dividir el UID en sus partes componentes
    parts = uid.split('-')
    number = int(parts[0])
    name1 = parts[1]

    # Calcular un valor único basado en las partes del UID
    value = (number * len(name1)) % 30
```

```
# Si el UID tiene una segunda parte de nombre, incluirla en el cálculo
if len(parts) == 3:
    name2 = parts[2]
    value = (value + len(name2)) % 30

# Devolver el valor calculado
return value
```

Con esta función se crean los índices para manejar el sistema de cache distribuido. Ahora se empezarán a preparar las consultas al cache. Para eso se abre el archivo txt y se eligen de manera random algunos uid para buscar en el cache.

```
with open('uids.txt', 'r') as f:
    uids = f.read().splitlines()

# Definir la cantidad de consultas que se desean hacer
n = 4000
sample_uids = random.choices(uids, k=n)
```

Para lo anterior se hicieron 4000 consultas seguidas. de las cuales se preguntará al cache si es que tiene la información solicitada.

Como se sabe a priori el uid de búsqueda, es decir se sabe el id y el rango en cache. Dado un uid se buscará en el cache en donde probablemente esté la información. esto simplemente se logra con condicionales simples como se muestra a continuación.

```
if (valor_rango > 0 and valor_rango <=9):
    start_time = time.time()
    result = r1.get(uid)
    if result is not None:
        print("cache hit in cache 1")
        end_time = time.time()
        contadorCache = contadorCache + 1
        AcumuladorCache = AcumuladorCache + end_time - start_time
    else:
        print("cache miss")
        try:
            valor_rango = encode_uid(uid)
            url = f'https://aves.ninjas.cl/api/birds/{uid}'
            response = requests.get(url)
            data = json.loads(response.text)
            size = data['size']
            species = data['species']
            nombre_comun = data['name']['spanish']
            r1.set(uid, f'{nombre_comun}, {size}, {species}')
            print("fetched from api")
            end_time = time.time()
            contadorAPI = contadorAPI + 1
            AcumuladorAPI = AcumuladorAPI + end_time - start_time

        except json.decoder.JSONDecodeError:
            print("Error decoding JSON. Continuing with next iteration.")
            continue

print(f"UID: {uid}, Resultado: {result}, Tiempo: {end_time - start_time}")
```

El código anterior muestra como funciona para en caso de que la información este en la primera instancia, es decir desde el id 0 hasta al 9. Pregunta en el cache si existe, en caso contrario busca en la API y guarda la información en el cache correspondiente. Mismo caso corre para las otras instancias solo que con diferente rango de ids.

También se usaron controladores de tiempo para obtener unidades de medidas experimentales. Como cuanto se demora en promedio el total de peticiones a cache y a la API. También es importante recalcar que los índices ayudan al rendimiento del cache, distribuyendo la información lo mas equitativamente posible. En pruebas anteriores se metieron todos los valores de la api al cache usando la reglas de los índices. En total habían 208 valores, pero con la política de cache junto con los índices, cada cache tenia 60, 62 y 80 datos respectivamente. logrando una distribución correcta de datos para optimizar tiempo de consultas.

Una vez explicado esto, simplemente se debe ejecutar los contenedores y llamar al script de python para que haga las consultas. Al finalizar entregará los tiempos utilizados para las consultas de cache y a la API. Con esto queda demostrada la implementación de la demostración para la utilización de cache distribuido.

```
UID: 11-spheniscus-humboldti, Resultado: None,
Tiempo: 0.4320950508117676
cache miss
fetched from api
UID: 93-charadrius-collaris, Resultado: None, Tiempo: 0.4538547992706299
cache miss
fetched from api
UID: 103-tringa-flavipes, Resultado: None,
Tiempo: 0.29033517837524414
cache hit in cache 2
UID: 4-rollandia-rolland, Resultado: b'Pimpollo, 24 - 36 cm., Nativa', Tiempo: 0.00044
cache hit in cache 3
UID: 79-milvago-chimango, Resultado: b'Tiuque, 32 - 43 cm., Nativa',
Tiempo: 0.0004956722259521484
```

Figura 5: Output para client.py

```

cache hit in cache 2
UID: 71-elanus-leucurus, Resultado: b'Bailar\xc3\xad, 35 - 43 cm., Nativa', Tiempo: 0.0004291534423828125
cache hit in cache 2
UID: 4-rollandia-rolland, Resultado: b'Pimpollo, 24 - 36 cm., Nativa', Tiempo: 0.0004436969757080078
cache hit in cache 3
UID: 41-chloephaga-picta, Resultado: b'Caiqu\xc3\xa9, 60 - 73 cm., Nativa',
Tiempo: 0.0005173683166503906
cache hit in cache 2
UID: 187-mimus-tenca, Resultado: b'Tenca, 28 - 29 cm., Nativa', Tiempo: 0.00040721893310546875
cache hit in cache 1
UID: 73-parabuteo-unicinctus, Resultado: b'Peuco, 45 - 59 cm., Nativa', Tiempo: 0.0003044605255126953
cache hit in cache 1
UID: 179-elaenia-albiceps, Resultado: b'F\xc3\xado-F\xc3\xado, 14 - 15 cm., Nativa', Tiempo: 0.00027108192443847656
cache hit in cache 3
UID: 103-tringa-flavipes, Resultado: b'Pitotoy Chico, 23 - 25 cm., Nativa',
Tiempo: 0.0003771781921386719
cache hit in cache 1
UID: 182-pygochelidon-cyanoleuca, Resultado: b'Golondrina de Dorso Negro, 12 - 13 cm., Nativa', Tiempo: 0.0002989768981933594
cache hit in cache 1
UID: 117-leucophaeus-scoresbii, Resultado: b'Gaviota Austral, 40 - 46 cm., Nativa', Tiempo: 0.0004086494445800781
cache hit in cache 3
UID: 166-scelorchilus-albicollis, Resultado: b'Tapaculo, 19 cm., Nativa',
Tiempo: 0.00048732757568359375
cache hit in cache 3
UID: 75-buteo-ventralis, Resultado: b'Aguilucho de Cola Rojiza, 45 - 60 cm., Endemica de Chile',
Tiempo: 0.0004291534423828125
cache hit in cache 3
UID: 11-spheniscus-humboldti, Resultado: b'Ping\xc3\xbcino de Humboldt, 65 - 70 cm., Nativa',
Tiempo: 0.00047135353088378906
cache hit in cache 2
UID: 197-phrygilus-fruticeti, Resultado: b'Yal Com\xc3\xban, 16 - 18 cm., Nativa', Tiempo: 0.0004723072052001953
el promedio de tiempo de cache es : 0.0004202002250667169
el promedio de tiempo de API es : 0.40327151192044747
pablosky@pc:~/Escritorio/sistemas_distribuidos/tareals$

```

Figura 6: Output para client.py

En la figura 5 se muestra que el inicio de ejecución, en el cual ya han sido guardados datos de la API en cache. Sin embargo cuando no encuentra los datos en el cache hace la consulta a la API. La figura 6 muestra el final de la ejecución mostrando los tiempos promedio de acceso.

4. Análisis de cache

En esta sección se analizara el rendimiento que tiene la implementación del cache y como afecta los tiempos de búsqueda, se analizará en función de las políticas usadas, en este caso, se usó lru y random. También se observará y comparará el rendimiento del cache vs las consultas directas a la API.

4.1. Política de remoción: allkeys-lru

En esta política se eliminan los elementos menos utilizados de la caché, para esta sección se hicieron distintos niveles de peticiones. con 500 peticiones, 2000 y 4000 respectivamente. en las cuales se recogieron los siguientes datos.

Politica LRU	500 consultas	2000 consultas	4000 consultas
Promedio de tiempo para cache	0.0004990345176454147	0.0004072658901058076	0.00039678494649215
Promedio de tiempo para API	0.42073752710728046	0.37511989098150755	0.3459843788432

De la tabla se puede observar que el tiempo de consulta para cache es considerablemente mas bajo, cabe recalcar que esto es un promedio de acceso. Contrariamente el proceso para obtener datos desde la API es significativo, pensando en el contexto de tener que hacer una cierta cantidad de consultas a la API, puede resultar en perdida de tiempo y recursos informáticos.

También se puede observar que el tiempo de búsqueda mejora o converge a un tiempo menor dada una cierta cantidad de peticiones, LRU ofrece una considerable mejora para casos donde los datos buscados se repiten, esto puede ser el contexto de un navegador. Considerar que en este caso se piden datos al azar, entonces la eficacia del algoritmo decae un poco. Sin embargo para contextos de producción es el algoritmo generalmente mas eficaz.

4.2. Política de remoción: random

En esta política se eliminan datos random del cache cuando este se llena, esta política funciona bien para contextos en los cuales la búsqueda de datos no sigue una distribución fija, como en esta caso buscamos datos de manera aleatoria, el algoritmo de remoción debería disminuir en algún grado el tiempo de búsqueda para cache.

El resultado que se obtuvo de medir con la cantidad de peticiones es el siguiente:

Politica LRU	500 consultas	2000 consultas	4000 consultas
Promedio de tiempo para cache	0.0004944931970883722	0.00040937368612052144	0.00037678694619774553
Promedio de tiempo para API	0.41993575300124875	0.4009164498981674	0.31434789238902

Como se puede observar de los datos obtenidos al hacer las consultas, los números respecto a la primera política son un poco mejores en algunos casos, sin embargo al ser random no necesariamente se cumplirá para todos los casos.

Sin embargo Para ambas políticas se puede observar un tiempo considerable en los tiempos de acceso a cache vs la obtención de los datos de la API, siendo el cache una herramienta clara de optimización de obtención de datos.

5. Configuración de contenedores

Como se menciona anteriormente se utilizaron dos políticas de remoción. LRU y random. De las cuales dadas las características del contexto y el tipo de api escogida se ajusto mejor en la mayoría de los casos la politica random, obteniendo tiempos de acceso a cache y éxitos de cache mayores. Los demas aspectos solicitados fueron explicados en las secciones anteriores.

6. Conclusiones

En términos de tiempo, se demostró que la implementación de un cache distribuido puede reducir significativamente el tiempo de consulta, asociado a una política de remoción según el contexto al cual se enfrenta. Además se debe observar el modelo de negocios para saber si conviene tener un cache de tamaño grande o no. Si se tiene un sistema el cual no se tienen consultas recurrentemente o siempre se pregunta lo mismo, el tamaño puede ser menor. Así se ahorra en costo de hardware y procesamiento. Sin embargo si la lógica de negocios esta en el traspaso de información (a través de datos o peticiones) es indispensable la implementación de un sistema de cache distribuido para poder optimizar y manejar las consultas de una manera rápida.

7. link a video

<https://youtu.be/tZxbT8kN8-8>