UNIVERSIDAD DIEGO PORTALES

FACULTAD DE INGENIERÍA Y CIENCIAS

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

Asignatura: Sistemas Distribuidos

Tarea 1

Profesor:

Nombres estudiantes: Alex Parada, Pablo Moraga, Sebastian Quintana

Santiago, Chile

Abril 2023

Índice

1. Introducción

En el presente informe se detalla, en primera instancia, la comparación al realizar una consulta a una API y realizar la misma consulta, pero guardada en caché. Para ejecutar esta comparación se dividirá en dos secciones, por una parte, análisis de escalabilidad (modificando parámetros que influyen en las consultas) y de rendimiento.

En la segunda parte de la tarea se realizará la comparación entre implementar un sistema RPC y una API REST, que tan conveniente es usar un sistema que el otro.

2. API utilizada

Para esta tarea se utiliza la API de PokeApi, esta página contiene hasta el Pokémon número 898 del anime y juego "Pokemon", proporcionando información de los Pokémon y de los objetos, tanto sus stats, ubicación, locación, etc.

3. Desarrollo actividad 1

3.1. Explicación del código

3.1.1. Instalaciones Previas

Antes de comenzar a explicar el código, se procedieron algunas instalaciones que permitieron facilitar la creación del código para la tarea. Algunas de las herramientas fueron:

Figura 1: Dependencias

Además de las dependencias anteriores, se emplearon la librería "fs " para poder ir guardando los tiempos en un archivo txt y, por otra parte, la librería de python matplotlib versión 3.5.1, esta se usó para poder graficar los resultados.

3.1.2. Consulta Api

Para realizar la consulta solo a la API (esto nos servirá para realizar la comparativa APIREST VS GRPC), se realiza el siguiente código:

En primera instancia se crean las variables que permitan proceder la captura de datos a la API y la que permitirá escribir las variables en un archivo txt.

Luego se genera la función RANDOM, esta función permitirá más adelante seleccionar los datos de un Pokémon al azar, el rango que funcionara esta función es entre el 1 al 898. Luego se actúa la función buscar que utilizara la variable axios para obtener los datos de la API.

```
1 const axios = require('axios');
2 const fs = require('fs');
3
4 function Random(min, max) {
5
6   return Math.floor(Math.random() * (max - min + 1)) + min;
7 }
8
```

```
async function buscar(num) {

  try {

    const respuesta = await axios.get(`https://pokeapi.co/api/v2/pokemon/${num}`);
    const pokemon = respuesta.data;
    return pokemon;
  } catch (error) {

    console.error('Error', error);
    throw error;
  }
}
```

Luego se procede la función "llamadas" que posee dos finalidades encenciales, por una parte, es traer desde la API la información del nombre, número de la pokedex y tipos asociado a los pokemones a los cuales se esté obteniendo de la llamada, esto se realiza para automatizar y simular las llamadas a la API. El tiempo de la consulta de cada llamada se va guardando en un arreglo con el fin de usar la variable "fs" para guardar tales datos en un archivo txt, con el fin de poder graficarlos posteriormente.

Para finalizar se hace el llamado de la función llamadas para simular a los clientes.

```
async function llamadas(n) {

  let tiempos = [];
  for (let i = 0; i < n; i++) {

    console.log(`Consulta ${i + 1}:`);
    const num = Random(1, 898);
    const inicio = new Date();
    const pokemon = await buscar(num);
    const fin = new Date();
    const tiempoConsulta = fin - inicio;
    tiempos.push(tiempoConsulta);
    console.log(`Nombre: ${pokemon.name}`);
    console.log(`N mero de la Pok dex: ${pokemon.id}`);
    console.log(`Tipos: ${pokemon.types.map(tipo => tipo.type.name).join(', ')}`);
    console.log(`Tiempo de consulta: ${tiempoConsulta} ms`);
    console.log('----------------');
  }

  fs.writeFile('tiempos.txt', tiempos.join('\n'), (err) => {  if (err) throw err;
    console.log('Los tiempos de consulta han sido guardados en el archivo tiempos.txt');
  });
}

const consultas = 100;
llamadas(consultas)
  .catch(error => {

    console.error('Error al realizar consultas', error);
  });
```

3.1.3. Consulta API-REDIS

Para este apartado, se sigue la misma lógica que el código de Consulta API, salvo lo siguiente.
A diferencia de la consulta de API, se emplea una nueva variable denominada Redis, esta variable nos permite efectuar la conexión con la base de datos Redis, para esto se crean 3 variables de Redis que permitirán asociar a cada instancia creada, abriendo los puertos asociados y especificando el host, cabe recalcar que estas instancias serán modificadas para el análisis de políticas de remoción.

```
const fs = require('fs');
const axios = require('axios');
const Redis = require('ioredis');

const redis1 = new Redis({

```

```
  port: 6379,
  host: 'localhost'
});

const redis2 = new Redis({

  port: 6380,
  host: 'localhost'
});

const redis3 = new Redis({

  port: 6381,
  host: 'localhost'
});
```

Otra diferencia que subyace es la siguiente.

Dentro de la función buscar se verifica el rango del id y se asigna una instancia de Redis y un tiempo de vida (TTL) correspondiente a ese rango, este tiempo de vida se modificará en el apartado correspondiente.
Luego, la función intenta obtener el Pokémon con ese id de la caché de Redis utilizando el método get de la instancia de Redis. Si el Pokémon está presente en la caché, se devuelve el Pokémon en formato JSON parseado.
En caso contrario, la función actúa una llamada a la API de PokeAPI empleando Axios y recupera los detalles del Pokémon. Luego, se guarda el Pokémon en la caché usando el método set de la instancia de Redis con el tiempo de vida correspondiente.
Finalmente, la función devuelve el objeto Pokémon. Si ocurre un error, se lanza una excepción con un mensaje de error y se captura en el bloque catch.

```
async function buscar(id) {
  try {
    let pokemon;
    let redisInstance;
    let ttl;
    if (id <= 299) {
      redisInstance = redis1;
      ttl = 3600; // 1 hour TTL for pokemon <= 299
    } else if (id <= 598) {
      redisInstance = redis2;
      ttl = 1800; // 30 minutes TTL for pokemon <= 598
    } else {
      redisInstance = redis3;
      ttl = 900; // 15 minutes TTL for pokemon > 598
    }
    const cachedPokemon = await redisInstance.get(`pokemon:${id}`);
    if (cachedPokemon) {
      console.log(`Encontrado en cache redis ${id} en la instancia ${redisInstance.options.port}`)
    ;
      pokemon = JSON.parse(cachedPokemon);

    } else {
      console.log(`Cache miss for ${id}`);
      const respuesta = await axios.get(`https://pokeapi.co/api/v2/pokemon/${id}`);
      pokemon = {
        name: respuesta.data.name,
        id: respuesta.data.id,
        types: respuesta.data.types.map(tipo => tipo.type.name)
      };
      const pokemonString = JSON.stringify(pokemon);
      await redisInstance.set(`pokemon:${id}`, pokemonString, 'EX', ttl);

    }
    return pokemon;
  } catch (error) {
    console.error('Error', error);
    throw error;
  }
}
```

3.1.4. Docker-Compose

En esta sección se muestra el docker compose utilizado, en este apartado se muestra el comando donde se puede establecer el tamaño máximo que se utiliza en caché, la política de remoción que se usa y las particiones que se realizan.

```yaml
version: '3.8'

services:

  redis1:
    image: 'bitnami/redis:7.0.10'
    ports:
    - "6379:6379"
    environment:
      - ALLOW_EMPTY_PASSWORD=yes
    volumes:
    - ./red1s1:/bitnami/redis/data
    command: "/opt/bitnami/scripts/redis/run.sh --maxmemory 5mb --maxmemory-policy allkeys-lru"
```

Figura 2: Dependencias

```yaml
  redis2:
    image: 'bitnami/redis:7.0.10'
    ports:
    - "6380:6379"
    environment:
      - ALLOW_EMPTY_PASSWORD=yes
    volumes:
    - ./red1s2:/bitnami/redis/data
    command: "/opt/bitnami/scripts/redis/run.sh --maxmemory 5mb --maxmemory-policy allkeys-lru"
```

Figura 3: Dependencias

```yaml
redis3:
  image: 'bitnami/redis:7.0.10'
  ports:
  - "6381:6379"

  environment:
    - ALLOW_EMPTY_PASSWORD=yes
  volumes:
  - ./red1s3:/bitnami/redis/data
  command: "/opt/bitnami/scripts/redis/run.sh --maxmemory 5mb --maxmemory-policy allkeys-lru"
```

Figura 4: Dependencias

3.2. Analisis del Cache

3.2.1. Politica de remocion

Las políticas de remoción es la estrategia que se toma para eliminar datos de la caché cuando se alcanza el límite de la memoria de la base de datos.

- **LRU:** Esta es la política de remoción utilizada, el nombre por sus siglas es Least Recently Used, este elimina los elementos de uso menos **reciente**, o sea va eliminando los datos más antiguos en el caché.
  Esta política comúnmente es empleada en caso de redes sociales donde los últimos posts, comentarios o chats

son más relevantes, también en datos de búsqueda locales, dado que un usuario puede volver a buscar los datos más recientes.

- **LFU:** Esta política, por sus siglas Least Frequently Used, va eliminando los elementos con un uso menos **frecuente**, o sea en el caso de eliminar un dato lo hace en torno al dato menos frecuente consultado.
  Esta política comúnmente es empleada en el comercio electrónico, donde es importante guardar los datos de búsqueda de productos más vendidos y productos más buscados, o en buscadores como bing, google donde normalmente hay una tendencia de consulta de datos.

Para el caso de la tarea se utilizó LRU dado que al hacer llamados por un random los datos no tendrán una tendencia real de datos más consultados, así que utilizar LRU o LFU no tendría una importancia significativa en este caso.
Cabe recalcar que al utilizar LFU se utiliza más memoria dado que tiene una tabla con los registros de frecuencias de las consultas.
En general, dependiendo de la naturaleza de los datos que se estén guardando en la caché y como se quiere optimizar el tiempo, se usa el uno o el otro.

3.2.2. Tamaño del cache

El tamaño en caché de redis predeterminado es de 1GB, este valor puede modificarse, lo ideal es no tener un tamaño de caché muy alta para que no haya una competencia de memoria con redis, sin embargo, si se usa un valor bajo es posible que las consultas no se puedan almacenar en caché y deban ser recuperadas de la fuente de datos original y esto haría que el proceso se ralentizara.

En general, el tamaño caché debe ser lo suficientemente grande para contener los datos más frecuentados e importantes, y al mismo tiempo lo suficientemente pequeño para no saturar la memoria del sistema.
Para la tarea se usa solamente 5mb, lo cual puede bajarse a 3 mb o más dado que se trabaja con datos los cuales no llegan a ser muy pesados, son solo información de pokemon y para ver como funciona la política de remoción lo ideal es tenerla en un tamaño justo que vaya borrando datos y sustituyendo como sería en una caché real, ademas se incluye más adelante en las pruebas de rendimiento un caso en donde se use una memoria significativamente más alta para ver sus efectos.

4. Pruebas de rendimiento

Primero que se realiza la distribución a los contenedores en 20 rangos, el contenedor 1 poseerá la información de los pokemones del 1 al 299, luego el contenedor 2 poseerá la información de los pokemones del 300 al 598, y el contenedor 3 los restantes. Esta distribución se hace para que cada contenedor posea equitativamente los pokemones presentes en la API.

Para esto se uso un tamaño de 50 mb en el contenedor, esto repercutirá ya que permite guardar mas datos en cache y no se aplicara directamente la politica Lru, la prueba con menos cache asociado a los contenedores se realizará en el video correspondiente, en donde la tasa de eficiencia disminuirá.

Para realizar la prueba de rendimiento se usarán 3 métricas,

- **Tiempo de Respuesta**: Tiempo que tarda en completarse una consulta a la fuente de datos original,
- **TTL o tiempo de Vida**:Tiempo en que los datos almacenados en caché están disponibles antes de ser descartados.
- **Tasa de eficiencia del caché**: Proporción de consultas que se resolvieron desde la caché en relación con el total de consultas realizadas, se usa la siguiente fórmula: $Tasa\ de\ eficiencia\ del\ caché\ = \frac{Consultas\ satisfechas\ por\ el\ caché}{Total\ de\ consultas}$

Primero se harán tres pruebas de tiempo de respuesta sin establecer un TTL previo, y, por otra parte, se hará lo mismo, pero aplicando TTL, además se analizará la tasa de eficiencia de caché en cada prueba.

4.1. Pruebas sin TTL

En esta prueba se realizaron 1000 llamadas a la API en cada iteración, esto para las demoras que posee la aplicación cuando ya están guardados los datos en caché y cuando no. Cabe destacar que no se reinició el contenido del docker compose para realizar las tres pruebas, ya que se busca ver como va repercutiendo las llamadas constantes al caché.
Los resultados fueron los siguientes.

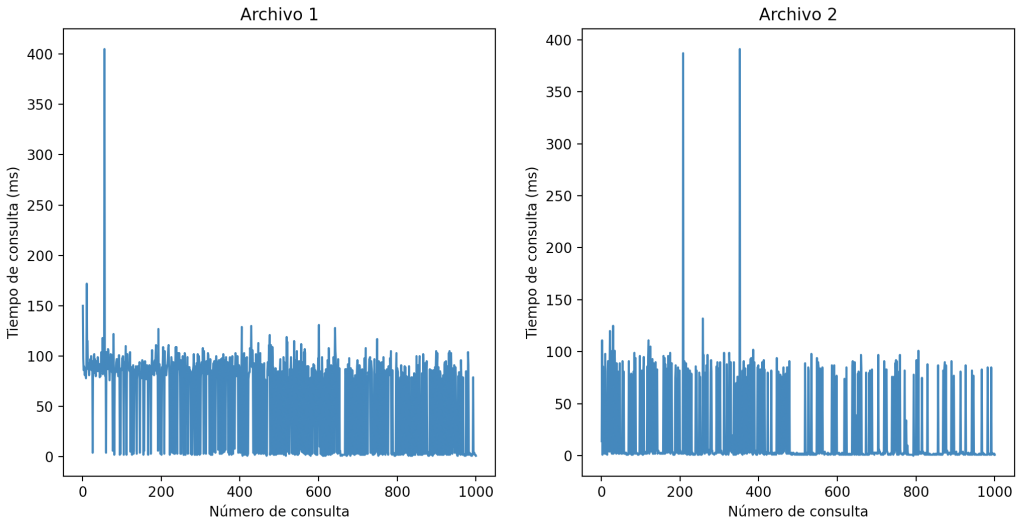

Figura 5: Consultas de iteracion 1 y 2

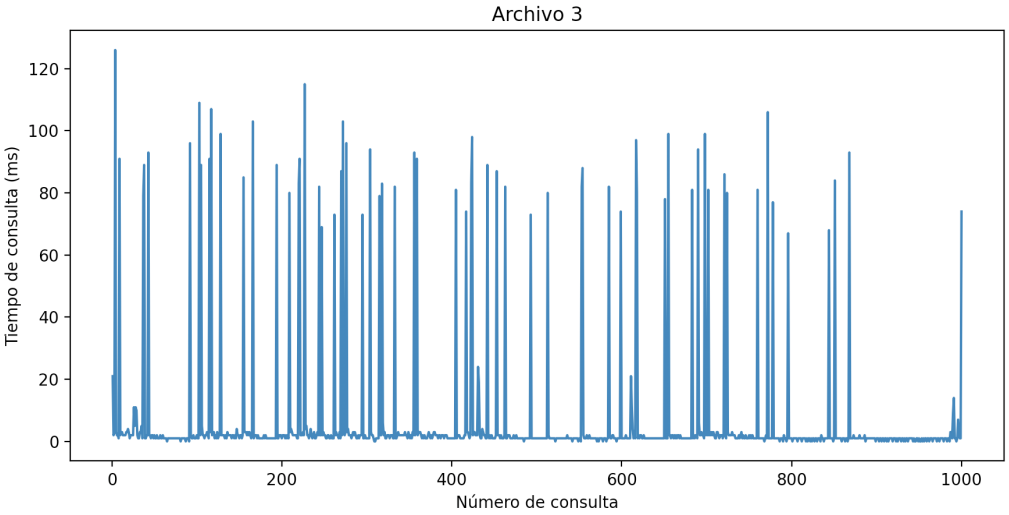

Figura 6: Iteracion 3

Se observa que, en la primera iteración, no había muchos datos guardados en caché porque los datos aún se estaban almacenando en la memoria caché. Esto condujo a que las consultas tuvieron que buscar la información en la API y esto aumentó el tiempo de consulta. Sin embargo, después de la primera iteración, una gran cantidad de datos se habían almacenado en caché y, por lo tanto, se redujo significativamente el tiempo de búsqueda en la base de datos. Esto es lo que explica el aumento en la velocidad de las consultas en las siguientes iteraciones. Otro factor importante es el uso de los índices, ya que se pueden buscar y recuperar datos de manera más eficiente, lo que puede reducir el tiempo de consulta. Además, al usar índices, se puede limitar el número de claves que se deben recorrer para buscar un valor específico, lo que también puede mejorar el rendimiento.

Además de lo anterior, como respaldo se obtuvo la tasa de eficiencia de caché para cada iteración obteniendo lo siguiente.

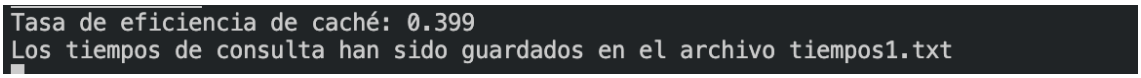

Figura 7: Eficiencia cache de la iteracion 1

Figura 8: Eficiencia cache de la iteracion 2

Figura 9: Eficiencia cache de la iteracion 3

Estos resultados vienen a complementar lo anterior dicho, mientas va avanzando el tiempo y el número de consultas, el porcentaje de consultas que se resuelven a través de la caché aumenta. Esto indica que el caché está funcionando de manera efectiva y está almacenando y proporcionando resultados más relevantes a medida que se hace un mayor utilización del sistema.

4.2. Pruebas con TTl

En este apartado, se agregará el parámetro del TTL, que como se especificó antes es el tiempo en que están almacenados los datos en caché antes de ser eliminados. Para esta prueba, se harán 3 iteraciones, una con 100 llamadas a la API, la iteración 2 con 500 llamadas a la API y la tercera con 1000 llamadas a la API. Para realizar esto se compara tanto para un TTL bajo de 2s y alto de 300s. Cabe destacar que cuando se modificó el TTL a uno alto, se vuelve a actuar el docker compose para que no haya ningún dato guardado en caché.
Se obtienen los siguientes resultados.

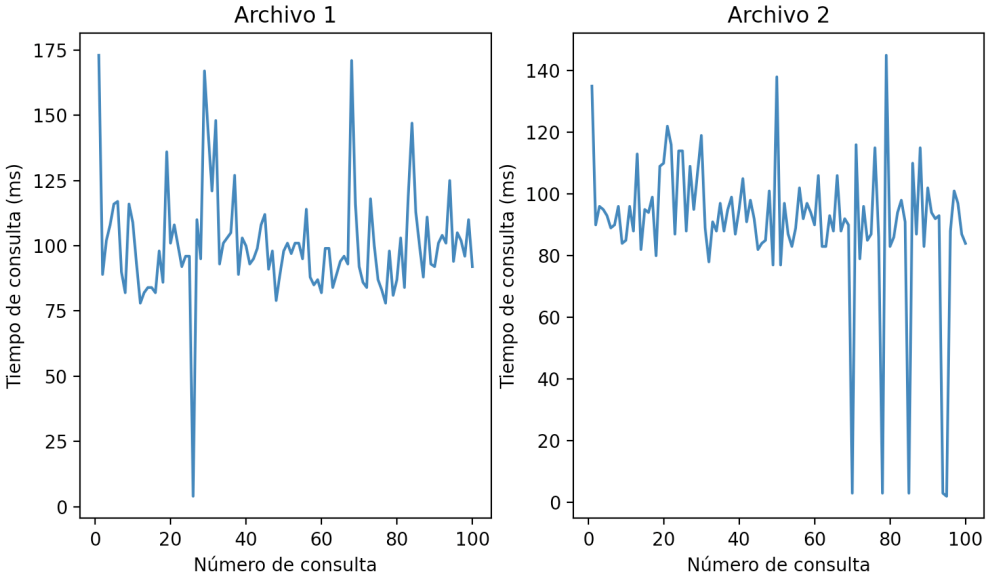

Figura 10: TTL Bajo vs Alto 100 consultas

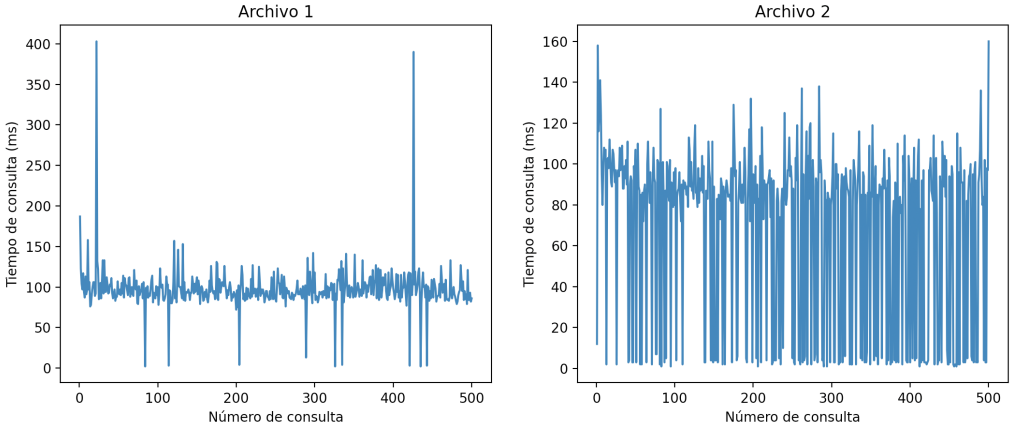

Figura 11: TTL Bajo vs Alto 500 consultas

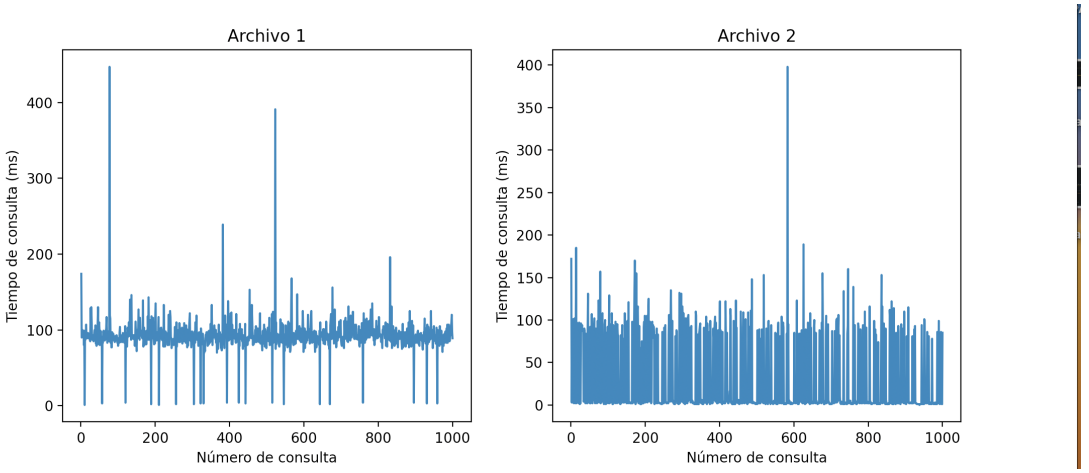

Figura 12: TTL Bajo vs Alto 1000 consultas

En las figuras presentadas, se observa que en la primera iteración ambos resultados se asemejan, sin embargo, el que presenta un bajo TTL posee menos consultas que requiere el cache y mas tiempo de consulta. Ya estos factores se ven drásticamente modificados, presentándose que las iteraciones con un alto TTL presentan tanto una mayor tasa exitosa de caché como también menor tiempo de consulta en la mayoría de iteraciones, en comparación a un TTL bajo que presenta un constante tiempo de consulta, pero una baja tasa de aceptación de caché. Sin embargo, presentar un TTL bajo o alto, puede traer beneficios como: Por una parte, un TTL bajo puede ser beneficioso en situaciones donde los datos de la caché cambian con frecuencia, como en aplicaciones con datos de tiempo real. Una clave con un TTL bajo se eliminará de la caché más rápido, lo que permitirá que se carguen los nuevos datos actualizados en la siguiente solicitud.
Por otro lado, un TTL alto puede ser beneficioso en situaciones donde los datos de la caché no cambian con frecuencia y no necesitan ser actualizados con frecuencia o por otra parte se obtenga una alta memoria asociada. Esto permite una mayor retención de datos en la caché, lo que puede mejorar el rendimiento de las aplicaciones al reducir la cantidad de accesos a la base de datos o al sistema de archivos.
En nuestro caso, al ser una función con llamadas random, repercute que necesitemos un TTL alto, ya que será difícil (debido al rango de pokemones que utilizamos) volver a encontrar al Pokémon, sin embargo, si la aplicación es recurrentemente llamada o posee una memoria asociada baja será necesario un TTL bajo.

5. Desarrollo actividad 2

5.1. Código gRPC

Para la implementación de gRPC se necesita un cliente y un servidor.
En primer lugar, se define una variable, la ruta de acceso al archivo de definición de protobuf y se importan las bibliotecas necesarias. Luego se carga el archivo de definición de protobuf y lo convierte en un objeto de definición de paquete, finalmente en la variable hollo_proto se carga la definición de protobuf en la biblioteca "'grpc"', lo que permite crear clientes gRPC y usar los mensajes definidos en el archivo de definición.

```
var PROTO_PATH = __dirname + '/../../protos/helloworld.proto';
var parseArgs = require('minimist');
var grpc = require('@grpc/grpc-js');
var protoLoader = require('@grpc/proto-loader');
var packageDefinition = protoLoader.loadSync(

    PROTO_PATH,
    {keepCase: true,
     longs: String,
     enums: String,
     defaults: true,
     oneofs: true
    });
```

```
var hello_proto = grpc.loadPackageDefinition(packageDefinition).helloworld;
```

Luego se genera la función main que se empleara para crear y llamar al cliente gRPC después utiliza la biblioteca ‘minimist‘ para analizar los argumentos de línea de comandos pasados al script, esto establece el destino para la convección gRPC. Se establece el nombre del usuario para la llamada sayHello al servidor grpc, finalmente con client.sayHello se realiza la llamada al método sayHello de servidor gRPC y se espera una respuesta.

```
function main() {

  var argv = parseArgs(process.argv.slice(2), {

    string: 'target'
  });
  var target;
  if (argv.target) {

    target = argv.target;
  } else {

    target = 'localhost:50051';
  }
  var client = new hello_proto.Greeter(target,
                                       grpc.credentials.createInsecure());
  var user;
  if (argv._.length > 0) {

    user = argv._[0];
  } else {

    user = Random(1,898)/*'mundo'*/; // n mero que se genera para hacer la petici n al server
  }

  const startTime = performance.now(); // Tomar el tiempo de inicio antes de hacer la llamada al
    servidor
  client.sayHello({name: user}, function(err, response) {

    console.log('Greeting:', response.message);
  });
}
```

En esta ultima parte se hace los llamados al main con un pequeño delay para que las llamadas no colapsen el servidor y que los tiempos de respuesta sean los adecuados.

```
for(let i = 0; i<500;i++){

  setTimeout(() => {

    main();
  }, i * 10); // espera i segundos antes de llamar a main()
}
```

Con la función stayHello recsibve la petición del cliente, en este caso lo que envía el cliente es la pregunta que se hará a la API, para finalmente devolverle la información.

```

let tiempos= [];
async function sayHello(call, callback) {


  const num = call.request.name;
  const inicio = new Date();
  let respuesta = await
  fetch(`https://pokeapi.co/api/v2/pokemon/${num}`);
  let pokemon = await respuesta.json()
  const fin = new Date();
  const tiempoConsulta = fin - inicio;
  tiempos.push(tiempoConsulta)
```

```
  callback(null, {message:  pokemon.name+" "+pokemon.id+" "+pokemon.types.map(tipo => tipo.type.
    name).join(',')+" ms "+(tiempoConsulta)});

  fs.writeFile('tiempos.txt', tiempos.join('\n'), (err) =>{

    if(err) throw err;
    console.log('Los tiempos de consulta han sido guardados en el archivo tiempos.txt')
  });
}

function main() {

  var server = new grpc.Server();
  server.addService(hello_proto.Greeter.service, {sayHello: sayHello});
  server.bindAsync('0.0.0.0:50051', grpc.ServerCredentials.createInsecure(), () => {

    server.start();
  });
}

main();
```

5.2. Comparaciones de REST y gRPC:

Para la primera comparación se hizo con 500 consultas para cada una y además el delay que se le dio al cliente en todas las pruebas fue de 10 ms.

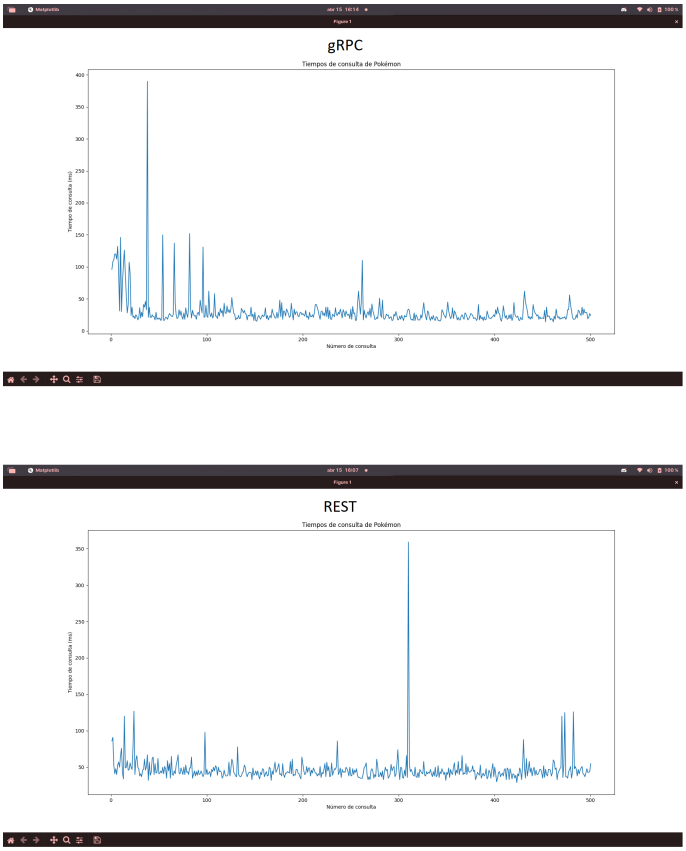

Figura 13: 500 consultas REST v/s gRPC

Como se puede observar en el gráfico anterior, para una cantidad de 500 peticiones, REST es más estable que gRPC por una mínima, ya que tiene más inestabilidad en sus tiempos de consulta y el punto más alto de gRPC sobre pasas el de REST, sin embargo, se puede decir que están más o menos a la par, ya que no existe un cambio

significativo.

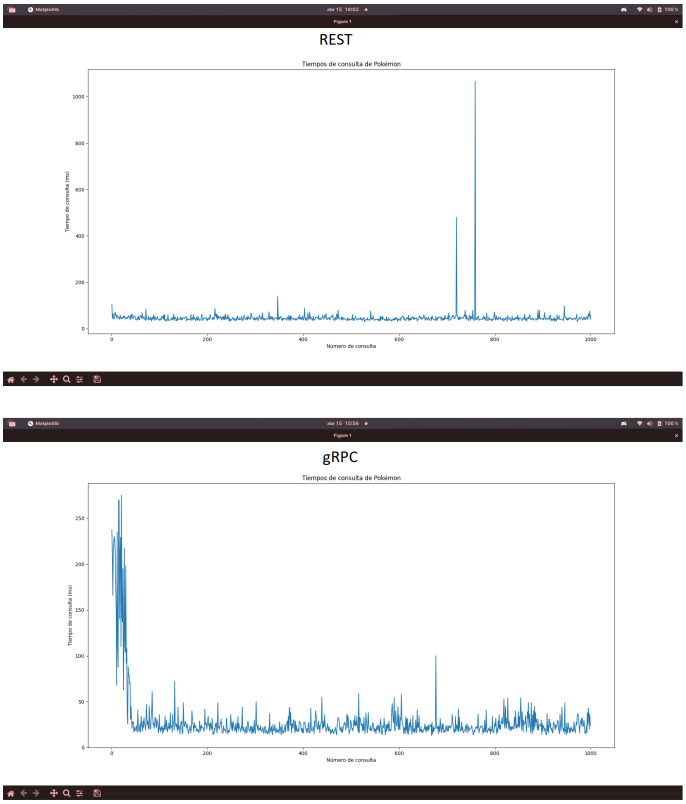

Figura 14: 1000 consultas REST v/s gRPC

En este gráfico se comienza a notar mucho más la diferencia entre usar gRPC y REST, ya que el tiempo de consulta más alto de gRPC no llega a los 300 ms, en cambio, en REST el más alto supera los 1000 ms, además la mayoría de consultas de gRPC no superan los 100 ms.

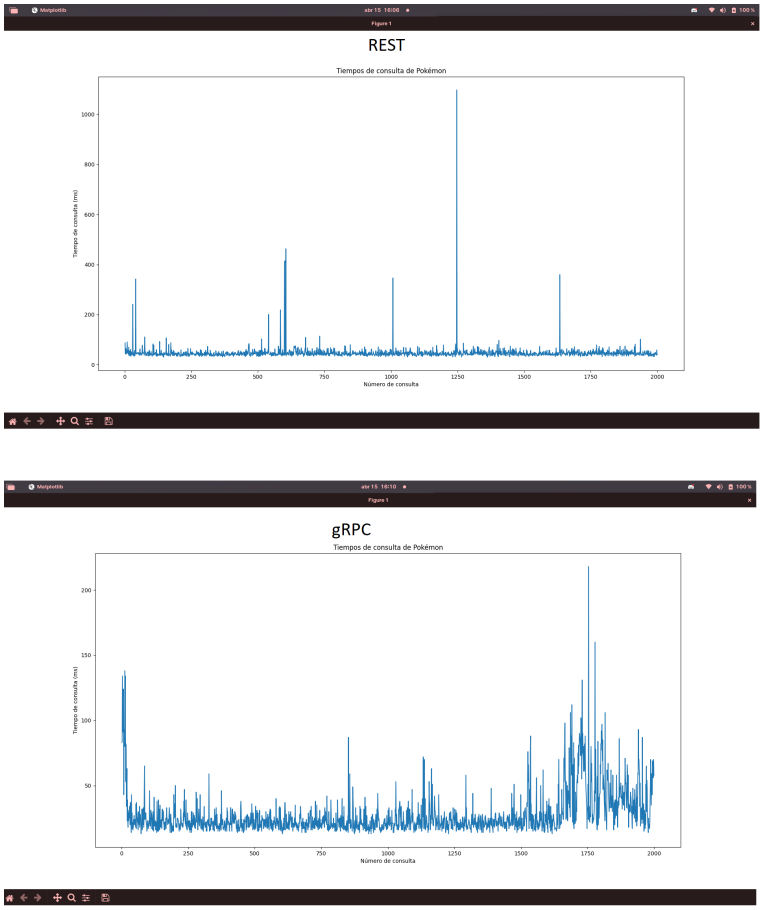

Figura 15: 2000 consultas REST v/s gRPC

Finalmente, notamos que REST es inferior a gRPC, ya que este último demostró tiempos más estables y mucho mas bajos.

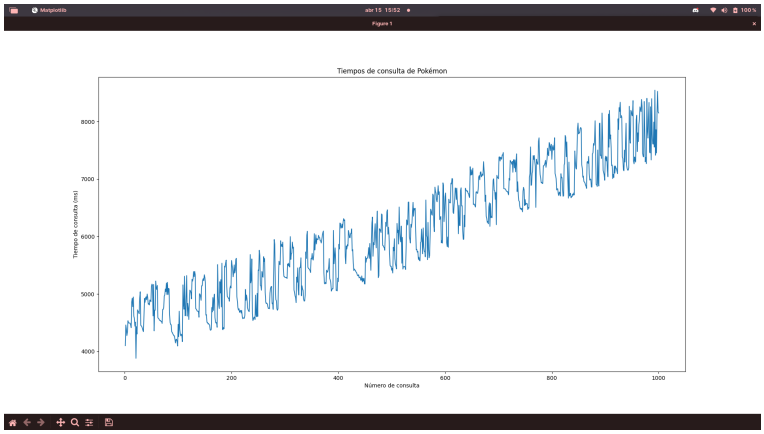

Figura 16: Gráfico gRPC si no se le aplica delay a las consultas del cliente

Como se dijo anteriormente, a las consultas del cliente se le aplicó un pequeño delay, pero en el caso de que este delay no se aplicara los valores de tiempo de consulta crecerían de una manera desproporcional y gRPC dejaría de ser el mas eficiente.

Para concluir con esta comparación, si bien gRPC supero por una amplia mayoría a REST y demostró ser estable. Sin embargo, si gRPC se quisiera ocupar en un sistema que reciba querys continuamente sin ningún tipo

de delay no sería una buena opción y en ese caso sería más óptimo usar REST.

Para concluir, es cierto que aunque gRPC puede tener un mejor rendimiento en términos de velocidad y eficiencia de red en comparación con REST, no siempre es la mejor opción para todos los casos de uso.

En el caso de una aplicación en la que se reciban consultas continuas sin ningún tipo de retardo, como por ejemplo una aplicación que reciba querys continuamente sin ningún tipo de delay no sería una buena opción y en ese caso sería más óptimo emplear REST.

6. Link al github

Click aqui

7. Link al Vídeo

Video de Tarea