

UNIVERSIDAD DIEGO PORTALES

FACULTAD DE INGENIERÍA Y CIENCIAS
ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES



Sistemas Distribuidos

Tarea 1

Profesor:
Nicolas Hidalgo

Ayudantes:
Cristian Villavicencio
Joaquin Fernandez
Nicolás Núñez

Estudiante:
Nicolás Moncada

Índice

1. Problema y solución	1
2. Módulos código	1
2.1. Postgres	1
2.1.1. Docker Compose	1
2.1.2. SQL	2
2.2. Api v1	2
2.3. Api v2	3
2.3.1. Server	3
2.3.2. Cliente	6
2.4. Redis	9
2.4.1. Idea 1	9
2.4.2. Idea 2	10
3. Funcionamiento caché	10
3.1. Explicación general	10
3.2. Redis	10
4. Resultados y análisis	11
5. Conclusión	11

1. Problema y solución

La empresa Xoogle necesita un sistema distribuido para poder usarlo como un web search engine. Pero si se hace un sistema tradicional (sin almacenamiento caché), en el momento que aumenten los usuarios activos el sistema backend se encontrará sobrecargado.

Debido a esto necesitamos una base de datos rápida (postgres), un sistema de almacenamiento caché (redis) y un sistema distribuido/escalable para así tener siempre disponibilidad en nuestro sistema.

2. Módulos código

2.1. Postgres

2.1.1. Docker Compose

```
version: '3.8'

services:
  database:
    image: postgres:alpine
    restart : always
    expose:
      - "4365"
    ports:
      - "4365:4365"
    environment:
      - DATABASE_HOST=${DATABASE_HOST}
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_DB=${POSTGRES_DB}
    volumes:
      - "./sql:/docker-entrypoint-initdb.d"
    command: -p 4365
```

Figura 1: Docker Compose

En el código de arriba podemos ver la configuración de puertos para postgres y en la sección de environment se configurarán las variables de entorno, las cuales son necesarias para la configuración inicial de postgres.

De igual forma definimos volumes, para entregar el archivo .sql (encargado de crear DB, tabla y llenar datos) y el archivo txt de logs que trabajaremos.

```
DATABASE_HOST=0.0.0.0
POSTGRES_USER=root
POSTGRES_PASSWORD=facil123
POSTGRES_DB=root
```

Figura 2: .env

Archivo .env para definir las variables de entorno.

2.1.2. SQL

```
SET timezone = 'America/Santiago';

DROP DATABASE IF EXISTS distri;
CREATE DATABASE distri;
\c distri;
CREATE TABLE logs [
    id int NOT NULL,
    title varchar NOT NULL,
    description timestamp NOT NULL,
    keywords int,
    URL varchar
];
\copy logs FROM '/docker-entrypoint-initdb.d/user-ct-test-collection-04.txt' DELIMITER E'\t' CSV HEADER;
```

Figura 3: SQL

Se verifica si es que existe la base de datos distri, en caso de existir se elimina y se crea nuevamente. Posterior a eso se crea una tabla logs con el mismo numero de columnas que trae nuestro txt. Por ultimo, hacemos un copy de la información del txt hacia nuestra tabla.

2.2. Api v1

```
const { Client } = require('pg');
const client = new Client({
    user: 'root',
    host: '0.0.0.0',
    database: 'distri',
    password: 'facil123',
    port: 4365
});
client.connect(function (error){
```

Figura 4: Conexión con Postgres

Gracias al uso de “`pg`” e ingresando los datos/variables de entorno de nuestra base de datos logramos que JavaScript se pueda conectar a postgres.

```
const getInfo = (request, response) => {
  const text = request.params.text

  client.query("SELECT * FROM logs WHERE title LIKE '%"+text+"%'", (error, results) => {
    if (error) {
      throw error
    }
    response.status(200).json(results.rows)
  })
}

const getMain = (request, response) => {
  response.json({ info: 'Node.js, Express, and Postgres API' })
}

app.listen(port, () => {
  console.log(`App running on port ${port}.`)
})

app.get('/info/:text', getInfo)
app.get('/', getMain);
```

Figura 5: Api

Api haciendo query a conexión con postgres para posteriormente entregarlo por medio de una consulta a “localhost:8080/info/:text”.

2.3. Api v2

2.3.1. Server

```
const { Client } = require('pg');
const client = new Client({
  user: 'root',
  host: '0.0.0.0',
  database: 'distri',
  password: 'facil123',
  port: 4365
});
client.connect(function (error){
  if(error){
    console.log("Here we go again");
  }else{
    console.log("Conexion con base de datos fue exitosa");
  }
});
```

Figura 6: Conexión postgres

Se ocupó la misma base para la conexión a postgres de la api v1.

```
var PROTO_PATH = './proto/server.proto';

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 server_proto = grpc.loadPackageDefinition(packageDefinition).server;
```

Figura 7: Configuración Proto

Configuración default para que JavaScript reconozca las utilidades de gRPC.

```
const postgres = require('./postgres');

function ListarID(request, response) {
    const text = parseInt(request.params.text);
    const query = "SELECT * FROM logs where id='"+text+"'";

    postgres.query(query, function(err, rows, fields) {
        if (err) throw err;
        //console.log(rows.length)
        for(const data of rows){
            //console.log(data);
            response.write(data);
        }
        response.end();
    });
}

function Prueba(response) {
    const query = "SELECT count(*) as conteo FROM logs";

    postgres.query(query, function(err, rows, fields) {
        if (err) throw err;
        //console.log(rows.length)
        for(const data of rows){
            //console.log(data);
            response.write(data);
        }
        response.end();
    });
}
```

Figura 8: Funciones

En esta parte del código se definen las siguientes funciones:

- ListarID
- ListarTitle
- ListarDesc
- ListarKey
- ListarURL
- Prueba

La función “Prueba” se creó solo para poder solicitar un select count(*) y así probar de forma rápida la api.

```
function main() {
  var server = new grpc.Server();
  server.addService(server_proto.Casos.service, {
    ListarID: ListarID,
    ListarTitle: ListarTitle,
    ListarDesc: ListarDesc,
    ListarKey: ListarKey,
    ListarURL: ListarURL,
    Prueba: Prueba
  });
  server.bindAsync('0.0.0.0:8080', grpc.ServerCredentials.createInsecure(), () => {
    server.start();
    console.log('gRPC server on port 8080')
  });
}

main();
```

Figura 9: Casos/Servicios

Aquí le indicamos al servidor gRPC cuales serán las funciones/serivicios que deberá considerar.

2.3.2. Cliente

```
const express = require('express');
const app = express();
var morgan = require('morgan');
var cors = require('cors');

//Settings
const port = 3000;

//Middlewares
app.use(express.json());
app.use(morgan('dev'));
app.use(cors());

//Routes
//app.use('/',require('./routes/index'))
app.use('/caso',require('./routes/casos.js'));

app.listen(port, ()=>{
    console.log('Servidor en el puerto', port);
});
```

Figura 10: api-server.js

Configuración puerto gRPC cliente.

```
var PROTO_PATH = './proto/client.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 client_proto = grpc.loadPackageDefinition(packageDefinition).client;

var argv = parseArgs(process.argv.slice(2), {
  string: 'target'
});
var target;
if (argv.target) {
  target = argv.target;
} else {
  target = 'localhost:8080';
}
var client = new client_proto.Casos(target,grpc.credentials.createInsecure());

module.exports = client;
```

Figura 11: gRPC_client.js

Configuración para que exista comunicación entre servidor y cliente gRPC.

```
var express = require('express');
var router = express.Router();
const client = require('../gRPC_client')

router.get('/todos', function(req, res) {
  const rows = [];

  const call = client.Prueba();
  call.on('data', function(data) {
    rows.push(data);
  });
  call.on('end', function() {
    console.log('Data obtenida con exito');
    res.status(200).json({data:rows});
  });
  call.on('error', function(e) {
    console.log('Error en entrega',e);
  });
});

router.get('/title/:text', function(req, res) {

  const rows = [];
  const text = req.params.text;
  console.log(text);
  const call = client.ListarTitle(text);
  call.on('data', function(data) {
    rows.push(data);
  });
  call.on('end', function() {
    console.log('Data obtenida con exito');
    res.status(200).json({data:rows});
  });
  call.on('error', function(e) {
    console.log('Error en entrega',e);
  });
});

module.exports = router;
```

Figura 12: casos.js

En este paso es donde empezaron los problemas de código, dado a que en la ejecución y consulta vía url al cliente entregaba el mensaje de que no existían los casos solicitados. Pese a estar configurado en todos los archivos .proto.

2.4. Redis

2.4.1. Idea 1

```
version: '3.8'
services:
  redis1:
    image: redis
    container_name: redis1
    ports:
      - "6376:6376"
    volumes:
      - ./redis1/data:/data
      - ./redis1/config/redis.conf:/usr/local/etc/redis/redis.conf
    command: redis-server /usr/local/etc/redis/redis.conf
  redis2:
    image: redis
    container_name: redis2
    ports:
      - "6377:6377"
    volumes:
      - ./redis2/data:/data
      - ./redis2/config/redis.conf:/usr/local/etc/redis/redis.conf
    command: redis-server /usr/local/etc/redis/redis.conf
  redis3:
    image: redis
    container_name: redis3
    ports:
      - "6378:6378"
    volumes:
      - ./redis3/data:/data
      - ./redis3/config/redis.conf:/usr/local/etc/redis/redis.conf
    command: redis-server /usr/local/etc/redis/redis.conf
  redis4:
    image: redis
    container_name: redis4
    ports:
      - "6379:6379"
    volumes:
      - ./redis4/data:/data
      - ./redis4/config/redis.conf:/usr/local/etc/redis/redis.conf
    command: redis-server /usr/local/etc/redis/redis.conf
```

Figura 13: Intento 1 redis

Esta fue la idea que mejor funcionó, dado a que se levantaba un docker por cada imagen de redis. Pero falló el planteamiento dado a que pensaba vincularlo al node de la api v1. Pero la conexión realmente debía ir con el cliente gRPC correspondiente a mi api v2.

2.4.2. Idea 2

```
version: "3.8"
services:
  redis1:
    image: "redis:alpine"
    ports:
      - "8080:6376"
  redis2:
    image: "redis:alpine"
    ports:
      - "8080:6377"
  redis3:
    image: "redis:alpine"
    ports:
      - "8080:6378"
  redis4:
    image: "redis:alpine"
    ports:
      - "8080:6379"
  web:
    container_name: node-app
    build: ./node-app
    ports:
      - "3000:3000"
```

Figura 14: Intento 2 redis

Esta idea no funcionó por fallo en el planteamiento de los puertos.

3. Funcionamiento caché

3.1. Explicación general

El almacenamiento caché es una capa para almacenamiento de datos de alta velocidad, datos que se caracterizan por ser transitorios.

Estos datos se guardan para futuras solicitudes, debido a que de esta forma no se tendrá que hacer la solicitud hacia el almacenamiento tradicional (Disco Duro / HDD), dado a que tienen un tiempo de respuesta mucho más alto que el almacenamiento caché. Esta velocidad se logra gracias al uso de la memoria de acceso aleatorio (más conocida como RAM).

3.2. Redis

Redis es tomado mayoritariamente como una base de datos ya sea durable o persistente.
¿A qué nos referimos con durable o persistente?

- **Durable:** Estos datos se mantienen por un tiempo indefinido dentro de nuestra base de datos, ocupando un ejemplo dado en clases, si tomamos a Google como ejemplo, ellos saben que YouTube / Facebook / Instagram / etc. son servicios constantemente solicitados en su motor de búsqueda, por lo cual los pueden almacenar de forma durable para que de esta forma se desplieguen de forma más rápida para el usuario.
- **Persistente:** Estos datos son muy parecidos a los anteriores, pero con la pequeña diferencia que estos si pueden llegar a cambiar en algún momento. Podríamos pensar en una lista que ordene las 10 páginas más consultadas en tiempo real y que solo a estas les dé un tiempo de despliegue mucho menor al resto. Pero en el momento que dejen de recibir muchas visitas, simplemente bajarán en la lista cediendo su posición.

Es una base de datos pero con su planteamiento como un sistema de almacenamiento caché, dado a que la información la almacena en memoria ram.

4. Resultados y análisis

Lamentablemente no logré completar el 100% de esta tarea.

Pero me podré referir al resultado deseado tomando el output de la api v1.



Figura 15: Resultado api v1

La idea es que tomando estos resultados (pero la versión que si implementa gRPC) redis tomaría los resultados y los guardaría como formato llave valor. Es decir, tomando la consulta solicitada, guardaría su respuesta correspondiente y así el sistema podrá responder de forma mas rápida para la próxima ocasión.

5. Conclusión

Los almacenamientos caché nos ayudan a cargar menos nuestro sistema, reduciendo las consultas hacia nuestro backend y base de datos. Lo cual nos permite reducir el consumo de nuestros servidores, lo que hará ver como que nuestro servidor tiene mucha mas potencia de la real, dado a que podrá resolver mas solicitudes de las que podría sin el almacenamiento caché.