

Tarea 1 : Sistema de Caché Sistemas distribuidos

Profesor: Nicolas Hidalgo
Integrantes: Benjamin Alonso y Juan Muñoz

Índice

1. Resumen	2
2. Descripción del trabajo.	2
2.1. Selección de la API pública	2
2.2. Conexión del backend y uso de Redis.	2
2.3. Análisis del rendimiento del cache	2
2.3.1. Metodología	2
2.3.2. Análisis	3
2.4. Se explica el código utilizado	7
2.4.1. Backend	9
2.4.2. Client	10
2.4.3. cronjob	13
2.4.4. Descargar cambios	14
2.5. Debe realizar un análisis justificado del caché y agregar los análisis pertinentes de los siguientes módulos:	14
3. Anexo	14
3.1. Código Matlab	14

1 | Resumen

El trabajo consiste en implementar un sistema de cache distribuido utilizando contenedores y la tecnología Redis, con el objetivo de optimizar el rendimiento de una API pública seleccionada previamente. Los hitos del trabajo incluyen la implementación del cache distribuido en contenedores, la conexión del backend con los contenedores de cache, el análisis del rendimiento del cache implementado y la explicación de la selección de la política de remoción utilizada para la configuración interna del contenedor. También se debe explicar cómo se ha particionado el sistema de cache por ids, qué política de remoción se ha utilizado en el sistema de cache y cuál es el límite de memoria que se ha establecido para el sistema de cache.

Además, se debe explicar el código utilizado en el proyecto y realizar un análisis justificado del cache, incluyendo la configuración de contenedores y la identificación y método utilizado para hallar el punto crítico de bajo rendimiento en la API.

2 | Descripción del trabajo.

2.1. Selección de la API pública

La API seleccionada publica, para el beneficio de la actividad, extraída desde : <https://www.themealdb.com/>

2.2. Conexión del backend y uso de Redis.

La conexión del backend con Redis y el uso de un sistema de caché son dos conceptos relacionados con la optimización del rendimiento de una aplicación web.

En cuanto a la conexión del backend con Redis, Redis es una base de datos en memoria que puede utilizarse como un almacén de datos temporal para una aplicación. En una arquitectura típica de una aplicación web, el backend suele ser responsable de la lógica de negocio y el acceso a la base de datos principal, mientras que Redis se utiliza para almacenar datos temporalmente. Por ejemplo, se podría utilizar Redis para almacenar resultados de búsquedas recientes, perfiles de usuarios activos o información de sesión.

2.3. Análisis del rendimiento del cache

2.3.1. Metodología

Se realizaron dos pruebas en un buscador web. En la primera prueba se tomaron 100 , en la segunda prueba se tomaron 500 y en la tercera de tomaron 1000 capturas en tiempo en ms. En todas las pruebas, se registraron tres tablas diferentes: cacheTimes, randomTime y redis calls.

redisCalls	randomTimes	cacheTimes
0	559	286
0	482	257
0	477	278
0	264	253
0	276	269
0	448	274
0	238	260
0	265	272
1	256	3
0	263	260
0	265	270
0	262	257
0	275	267
0	286	265
0	271	273
0	257	273
0	267	271
0	264	251
0	262	270
0	255	245
0	247	274
0	271	251
1	257	263
0	271	7
0	281	273
0	267	253
0	245	254
0	249	277
0	283	261
0	265	298
0	264	248
0	257	264
0	286	270
0	265	261
0	261	262
0	261	266
0	271	265
0	244	272
0	279	265

Estos datos son obtenidos del contenedor de Docker creado por el cliente.js, encontrados en la ruta: `sistemas-distribuidos-t1-main-client/files/app/`.

File	Size
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	1
25	0
26	0
27	0
28	0
29	0
30	0
31	0
32	0
33	0
34	0
35	0
36	0
37	0
38	0

La tabla `cacheTimes` muestra el tiempo que tomó realizar una búsqueda cuando se utilizó el caché. La tabla `randomTime` muestra el tiempo que tomó realizar una búsqueda sin utilizar el caché. Por último, la tabla `redisCalls` muestra si la búsqueda se realizó utilizando el caché (1) o no (0).

Además, se utilizó MATLAB para analizar los datos obtenidos en las pruebas y crear gráficos que ilustraran los resultados. Los gráficos creados permitieron visualizar de manera clara y concisa las diferencias en el tiempo de búsqueda entre el uso del caché y la búsqueda sin utilizar el caché.

2.3.2. Análisis

Después de realizar las tres pruebas (con 100, 500 y 1000 capturas), se realizó un análisis de los gráficos obtenidos. En general, los resultados fueron consistentes con los obtenidos en las pruebas anteriores.

En el primer gráfico, se puede observar una clara mejora en el tiempo de búsqueda cuando se utiliza el caché en comparación con la búsqueda sin utilizar el caché. Esto se observa en las tres pruebas, aunque la mejora es más evidente en la prueba con 100 capturas.

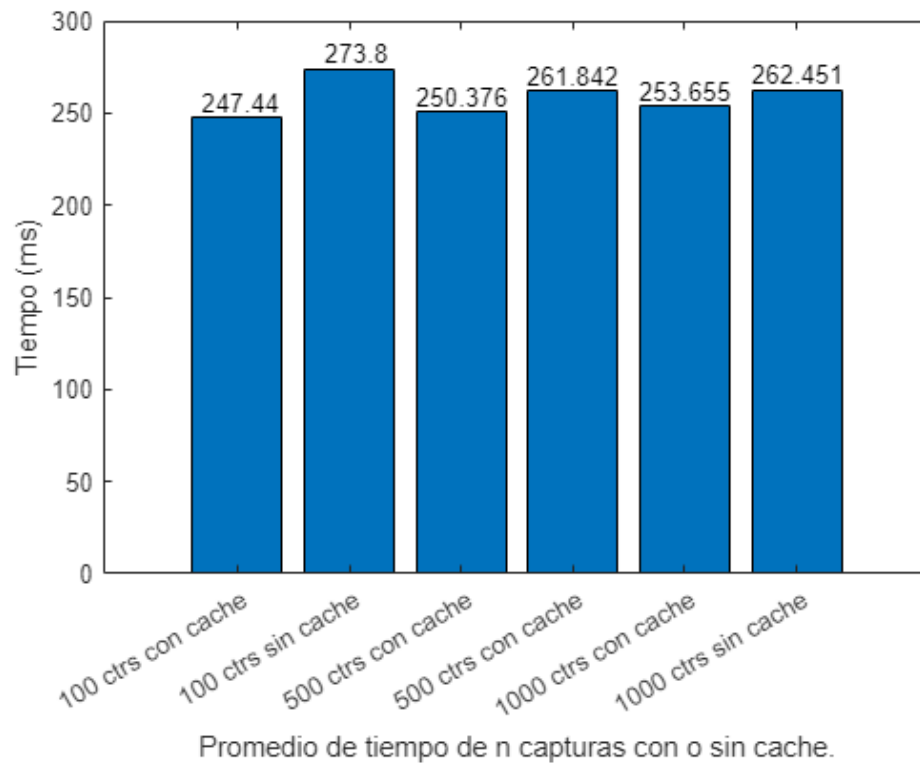


Figura 1: Gráfico 1

En el segundo gráfico, se muestra el porcentaje de veces que se entró al caché en cada una de las pruebas. Se puede observar que este porcentaje disminuye a medida que se aumenta el número de capturas. Esto sugiere y se infiere que, a medida que el número de búsquedas aumenta, el caché se vuelve menos efectivo y es menos probable que se utilice.

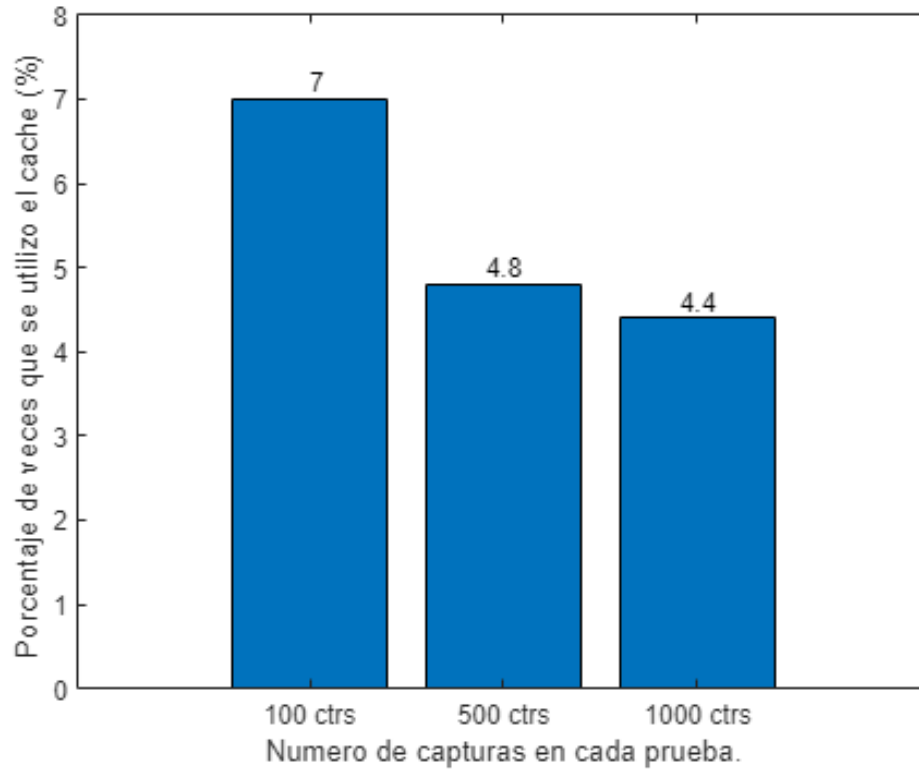


Figura 2: Gráfico 2

En el tercer gráfico, se muestra la diferencia entre los tiempos de búsqueda con y sin caché en cada una de las pruebas. Se puede observar que, en todas las pruebas, el caché mejora significativamente el tiempo de búsqueda. Además, la mejora en el tiempo de búsqueda es mayor en la prueba con 100 capturas. Combinado con la suposiciones anteriores, deberían hacerse más prueba para corroborar los resultados de que a medida que aumentan las capturas, disminuye la mejora. Un hipótesis, según la ley de los números grande que establece que a medida que se aumenta el tamaño de las muestra, la medida muestral se aproxima a la medida poblacional. Esto puede ser un ajuste a un verdadero valor de mejora promedio para n capturas.

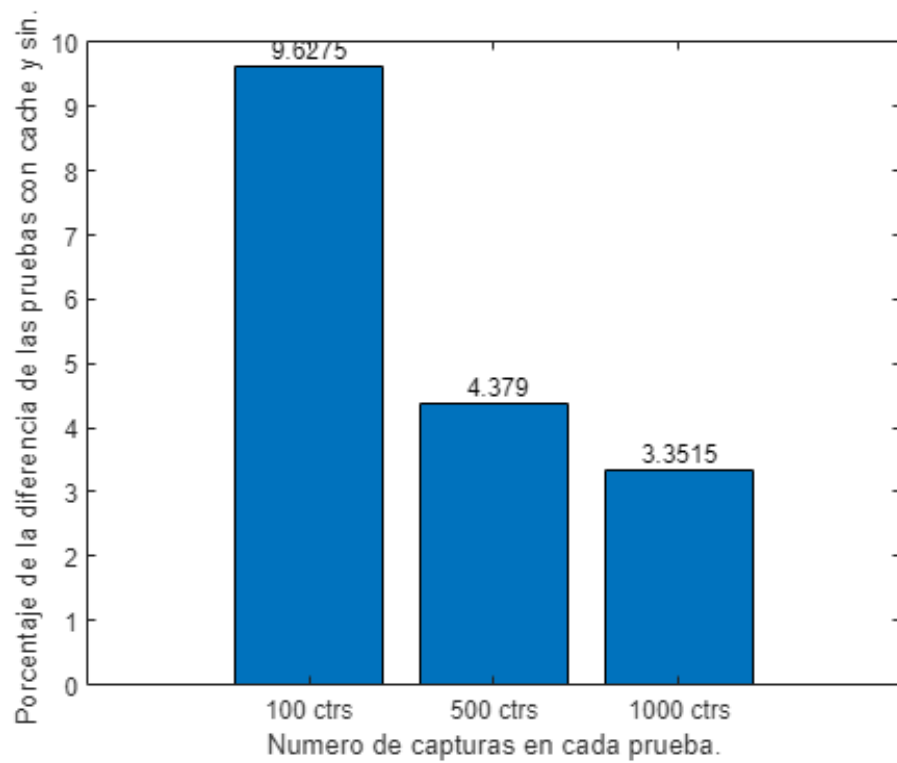


Figura 3: Gráfico 3

En el cuarto gráfico, se calculó el promedio de la mejora en el tiempo de búsqueda en las tres pruebas. El promedio indica que el uso del caché mejora el tiempo de búsqueda en un rango del 5.4 % al 5.78 %, lo cual es consistente con los resultados obtenidos en las pruebas anteriores.

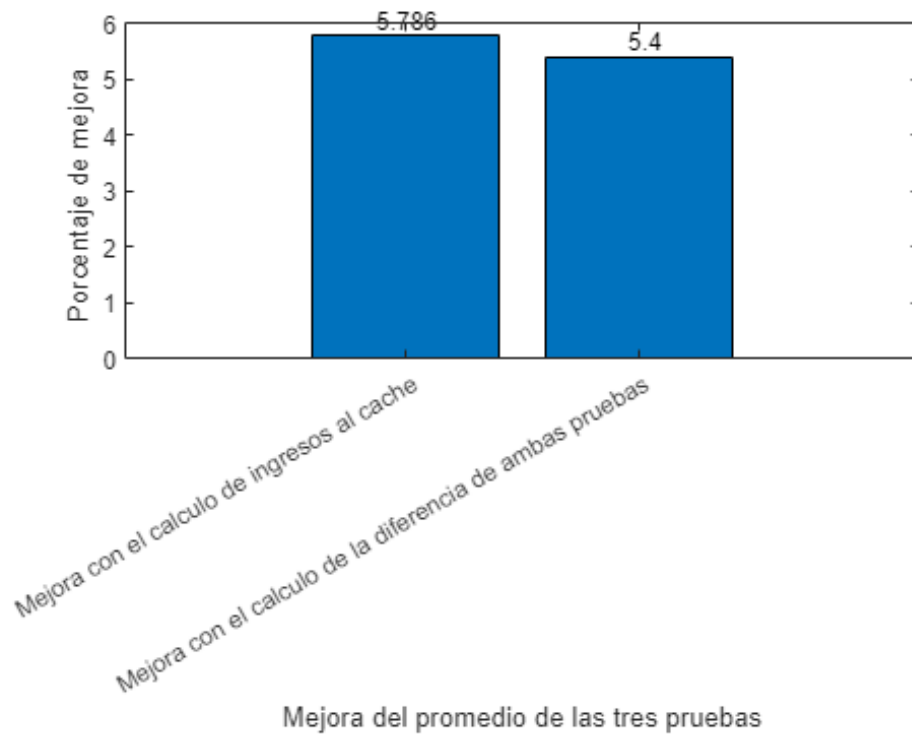


Figura 4: Gráfico 4

En conclusión, los resultados de las tres pruebas indican que el uso del caché mejora significativamente el tiempo de búsqueda en buscadores. Aunque el porcentaje de entradas al caché disminuye a medida que aumenta el número de búsquedas, la mejora en el tiempo de búsqueda es consistente en todas las pruebas. En promedio, se observa una mejora del 5.4 % al 5.78 % en el tiempo de búsqueda al utilizar el caché.

2.4. Se explica el código utilizado

En un principio, se crean las imágenes de Docker con el archivo *docker-compose.yaml*, donde además de crear la imagen del cliente y del backend, se crea la imagen del caché directamente en este archivo como se puede ver en la siguiente imagen:


```
1 version: '3.8'
2 services:
3   client:
4     build: ./client
5     environment:
6       - PORT=3001
7       - BACKEND_PORT=3000
8       - BACKEND_HOST=backend
9     ports:
10      - 3001:3001
11     volumes:
12      - ./client/client.js:/app/client.js
13     depends_on:
14      - backend
15   backend:
16     build: ./backend
17     environment:
18       - REDIS_HOST=cache
19       - PORT=3000
20     ports:
21      - 3000:3000
22     volumes:
23      - ./backend/backend.js:/app/backend.js
24     depends_on:
25      - cache
26   cache:
27     image: bitnami/redis:6.0.16
28     restart: always
29     environment:
30       - ALLOW_EMPTY_PASSWORD=yes
31     volumes:
32      - ./cache/redis.conf:/opt/bitnami/redis/mounted-etc/overrides.conf
33     ports:
34      - "6379:6379"
```

Figura 5: Archivo docker-compose.yaml

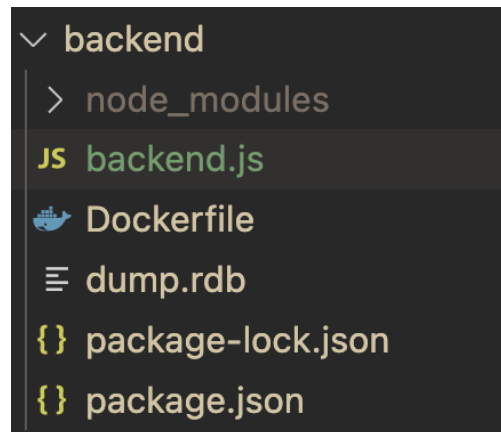
La imagen del caché es creada directamente en este archivo, como se puede ver entre las líneas 26 a 34. Mientras que las otras imágenes se crean desde los archivos Dockerfile dentro de cada carpeta. Debido a que tanto el lado del backend como el del cliente se realizan con javascript, la imagen es la misma para ambos, solo con la diferencia de la ejecución del comando *nodemon* junto al nombre del archivo.

```
1 FROM node:16-alpine
2 WORKDIR /app
3 COPY package.json package-lock.json /app/
4 RUN npm install
5 RUN npm install -g nodemon
6 CMD ["nodemon","backend"]
```

Figura 6: Dockerfile del backend

2.4.1. Backend

En general, esta carpeta se basa en su archivo dockerfile y el archivo backend.js que se encarga de recibir la consulta y verificar si debe consultarle a la API pública o a Redis



```
▼ backend
  > node_modules
  JS backend.js
  Dockerfile
  dump.rdb
  {} package-lock.json
  {} package.json
```

Figura 7: Carpeta backend

Las primeras líneas de este archivo consisten en la conexión con Redis y asignación de URL, por lo que no es necesario enfatizar en el detalle de este código.

```

1 app.get("/inventory/search", async (req, res) => {
2   const {query} = req;
3   const name = query.name
4   if (name == null){
5     res.status(400).json("Bad request")
6     return
7   }
8   let redisRes = await client.get(name)
9   if (redisRes){
10    console.log("Data obtained from Redis")
11    redisRes = JSON.parse(redisRes)
12    res.status(200).json({redisList: redisRes});
13  }else {
14    console.log("Data from public API")
15    const response = await axios.get(`${URL}search.php?s=${name}`).then(response => response.data)
16    if (response.meals && response.meals[0]){
17      for (let meal of response.meals){
18        await client.set(meal.strMeal, JSON.stringify(meal))
19      }
20      res.status(200).json({list : response.meals})
21    }else
22      res.status(500).json({response})
23  }
24 })
25

```

Figura 8: Endpoint /inventory/search

El endpoint anterior, consiste en recibir el nombre del producto. Es posible apreciar que después de la verificación del nombre en el endpoint se consulta en Redis si existe el nombre (línea 8), en caso de que no sea así, se le consulta a la API directamente. Cabe destacar que ambos formatos de respuesta son diferentes, esto se debe a que de esta manera es más simple poder diferenciar entre una consulta hecha hacia el caché o hacia la API pública.

Los siguientes endpoints son solo para consultar datos a la API pública sin el uso de Redis, esto se hace para poder comparar valores entre el tiempo que tarda en hacer consultas sin el caché y con el caché

```

1 app.get("/inventory", async (req, res) => { // without cache, just for testing
2   const {query} = req;
3   const name = query.name
4   if (name == null){
5     res.status(400).json("Bad request")
6     return
7   }
8   const response = await axios.get(`${URL}search.php?s=${name}`).then(response => response.data)
9   if (response.meals && response.meals.length > 0)
10     res.status(200).json({list : response.meals})
11   else
12     res.status(500).json({response})
13 })
14
15 app.get("/random", async (req, res) => {
16   return await axios.get(`${URL}random.php`).then(response =>
17     res.status(200).json(response.data)).catch(err => res.status(500).json(err))
18 })
19

```

Figura 9: Otros endpoints

2.4.2. Client

Esta carpeta se basa en un archivo dockerfile y el archivo client.js, al igual que la carpeta del *backend*, con el mismo dockerfile explicado en la figura 7. Algunas funciones creadas solo para simplificar tareas, son

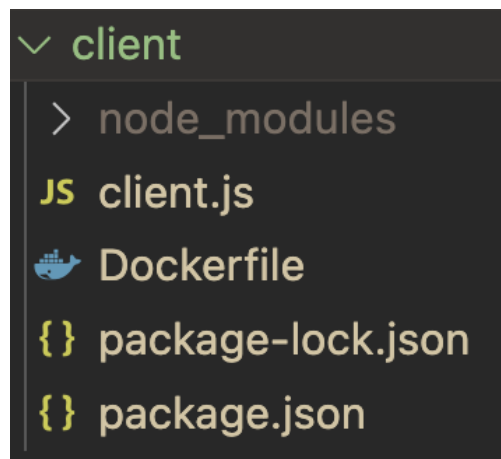


Figura 10: Carpeta del Cliente

sleep y *timeDiff*, donde *sleep* pausa el código por un tiempo X, mientras que *timeDiff* calcula la diferencia en milisegundos entre dos tiempos ingresados.

```
1 const sleep = async (ms) => {
2   return new Promise(resolve => setTimeout(resolve, ms));
3 }
4
5 const timeDiff = (time1, time2) => { // for some reason, time1 and time2 are strings
6   const time = moment(time1, "HH:mm:ss.SSS")
7   const secondTime = moment(time2, "HH:mm:ss.SSS")
8   let diff = time.diff(secondTime, 'milliseconds')
9   if (diff >= 0) return diff
10  return (diff*-1)
11 }
12
```

La función donde se obtienen los valores desde el backend, se puede apreciar en la siguiente imagen:

```

1 const getValues = async (URLAPI, redisTrack) => {
2   let timeDifference = 0;
3   let time = moment().format("HH:mm:ss.SSS")
4   let secondTime = moment().format("HH:mm:ss.SSS")
5   let value = await axios.get(URLAPI).then(response => {
6     secondTime = moment().format("HH:mm:ss.SSS")
7   return response.data
8   }).catch(err => err)
9
10  let meal
11  if (value.meals && value.meals[0]) meal = value.meals[0].strMeal
12  else if (value.list && value.list[0]) meal = value.list[0].strMeal
13
14  if (redisTrack){
15    if (value.redisList){
16      meal = value.redisList.strMeal
17      redisCalls.push(1)
18    }else redisCalls.push(0)
19  }
20  timeDifference= timeDiff(time, secondTime)
21
22  console.log("[client] Request time: ", timeDifference, "milliseconds")
23  console.log("[client] Name of the meal: ", meal)
24
25  if (!dictionary.has(meal.replace(/ /g, '%20')))
26    dictionary.add(meal.replace(/ /g, '%20'))
27  return timeDifference
28 }
29

```

Figura 11: función getValues

Las primeras tres líneas se encargan de obtener los tiempos, esto para poder calcular el tiempo que tarda entre consultar y obtener los datos. Entre la cuarta y octava línea se obtienen los datos y se agrega el tiempo en el momento que recibió los datos.

Desde la línea 10 a la línea 19 se encarga de recibir los datos, para poder diferenciar desde qué lugar llegan los datos existen tres tipos de response. Si bien esto no es una buena práctica debido a que una API inconsistente no es efectiva, en estos casos donde el enfoque es mas educacional, la API diferencia los datos para poder diferenciar cuando se llegan los datos desde redis o no.

En las últimas líneas se analiza si el dato ha sido agregado en el diccionario, este diccionario se utiliza para que cuando se realicen los llamados random (sin utilizar caché) después utilice estos nombres random llamados anteriormente, eso ocurre debido a que al existir una alta cantidad de datos en la API, si se llegaran a utilizar datos 100 % al azar practicamente todos los llamados serían únicos, lo que es justo lo que se desea evitar (para utilizar el cache).

La siguiente figura muestra las funciones que fueron utilizadas para llamar a la función *getValues* para

conseguir los datos.

```

1 const randomRequests = async() => { // this is just for add values to the dictionary
2   console.log("\n[client] Getting 200 random values\n")
3   for (let i = 0; i < 30; i++){
4     let timeDiff= await getValues(URL+"random")
5     randomTimes.push(timeDiff)
6   }
7 }
8 const getValuesWithCache = async () => {
9   console.log("\n[client] Getting 200 values with cache\n")
10  const array = Array.from(dictionary)
11  for (let i = 0; i < 30; i++){
12    let index = Math.floor(Math.random() * array.length)
13    console.log("index", array)
14    let timeDiff = await getValues(URL+"inventory/search?name="+array[index], true)
15    cacheTimes.push(timeDiff)
16  }
17 }

```

Figura 12: Funciones de llamados a la API con 30 intentos

La función *randomRequests* se ejecuta antes que *getValuesWithCache* debido a que *randomRequest* agrega los valores al diccionario para que luego pueda hacer las consultas con la estructura mencionada.

2.4.3. cronjob

La función *keepGoing* se encarga de realizar consultas al azar y calcular el rendimiento de realizar tales consultas. Debido a que la función *randomRequest* rellena todos los tiempos realizados en cada consulta, al sumar cada uno de estos valores nos da el tiempo final que tardó en realizar todas las consultas. En el caso que este tiempo demore más de un 40% se dejarán de realizar consultas.

```

1 const keepGoing = async () => {
2   console.log("[client] Getting values again without cache")
3   const totalTime = randomTimes.reduce((a, b) => a + b, 0)
4   randomTimes = []
5   await randomRequests()
6   const newTotalTime = randomTimes.reduce((a, b) => a + b, 0)
7   if (totalTime + totalTime*0.4 < newTotalTime){
8     console.log("[client] API performance decreased in a 40%")
9     console.log("[client] Stopping requests")
10  }else {
11    console.log("[client] Waiting 10 seconds to get values again")
12    sleep(10000)
13    await keepGoing()
14  }
15 }
16
17 const startRequests = async () => {
18   await sleep(5000);
19   await randomRequests()
20   await getValuesWithCache()
21 }
22
23 await startRequests()

```

2.5 Debe realizar un análisis justificado del caché y agregar los análisis pertinentes de los siguientes módulos:

2.4.4. Descargar cambios

Finalmente, para poder graficar los datos obtenidos, estos se descargan para poder utilizarlos con otro software que facilite la creación de gráficos.

```
1 fs.writeFile("randomTimes.txt", randomTimes.join('\n'), function(err) {
2     if(err) {
3         return console.log(err);
4     }
5     console.log("The file randomTimes was saved!");
6 }
7 )
8 fs.writeFile("cacheTimes.txt", cacheTimes.join('\n'), function(err) {
9     if(err) {
10        return console.log(err);
11    }
12    console.log("The file cacheTimes was saved!");
13 }
14 )
15 fs.writeFile("redisCalls.txt", redisCalls.join('\n'), function(err) {
16     if(err) {
17         return console.log(err);
18     }
19     console.log("The file redisCalls was saved!");
20 }
21 )
```

2.5. Debe realizar un análisis justificado del caché y agregar los análisis pertinentes de los siguientes módulos:

3 | Anexo

3.1. Código Matlab

```
sr500 = sum(randomTimes500{:, :});
sc500 = sum(cacheTimes500{:, :});
sr100 = sum(randomTimes100{:, :});
sc100 = sum(cacheTimes100{:, :});
sr1000 = sum(randomTimes1000{:, :});
sc1000 = sum(cacheTimes1000{:, :});

pr500 = mean(randomTimes500{:, :});
pc500 = mean(cacheTimes500{:, :});
pr100 = mean(randomTimes100{:, :});
pc100 = mean(cacheTimes100{:, :});
pr1000 = mean(randomTimes1000{:, :});
pc1000 = mean(cacheTimes1000{:, :});

c500 = sum(redisCalls500{:, :} == 1);
c100 = sum(redisCalls100{:, :} == 1);
c1000 = sum(redisCalls1000{:, :} == 1);

tabla = table(pc100, pr100, pc500, pr500, pc1000, pr1000);
bar(table2array(tabla));

text(1:length(table2array(tabla)), table2array(tabla), num2str(table2array(tabla)'), 'HorizontalAlignment', 'center', 'VerticalAlignment', 'bottom');
xticklabels({'100 ctrs con cache', '100 ctrs sin cache', '500 ctrs con cache', '500 ctrs sin cache', '1000 ctrs con cache', '1000 ctrs sin cache'});
ylabel('Tiempo (ms)');
xlabel('Promedio de tiempo de n capturas con o sin cache.');
```

```

error100 = (c100/100) *100;
error500 = (c500/500) *100;
error1000 = (c1000/1000)*100;

tabla1 = table(error100, error500,error1000);
bar(table2array(tabla1));

text(1:length(table2array(tabla1)), table2array(tabla1), num2str(table2array(tabla1)'), 'HorizontalAlignment', 'center', 'VerticalAlignment', 'bottom');
xticklabels({'100 ctrs', '500 ctrs', '1000 ctrs'});
ylabel('Porcentaje de veces que se utilizo el cache (%) ');
xlabel('Numero de capturas en cada prueba.');
```

```

errorp100 = ((pr100-pc100)/pr100) *100;
errorp500 = ((pr500-pc500)/pr500) *100;
errorp1000 = ((pr1000-pc1000)/pr1000) *100;

tabla2 = table(errorp100, errorp500,errorp1000);
bar(table2array(tabla2));

text(1:length(table2array(tabla2)), table2array(tabla2), num2str(table2array(tabla2)'), 'HorizontalAlignment', 'center', 'VerticalAlignment', 'bottom');
xticklabels({'100 ctrs', '500 ctrs', '1000 ctrs'});
ylabel('Porcentaje de la diferencia de las pruebas con cache y sin. ');
xlabel('Numero de capturas en cada prueba.');
```

```

numprom = (errorp100 + errorp1000 + errorp500) / 3;
difprom = (error500 + error100 + error1000) / 3;

tabla3 = table(numprom,difprom);
bar(table2array(tabla3));

text(1:length(table2array(tabla3)), table2array(tabla3), num2str(table2array(tabla3)'), 'HorizontalAlignment', 'center', 'VerticalAlignment', 'bottom');
xticklabels({'Mejora con el calculo de ingresos al cache', 'Mejora con el calculo de la diferencia de ambas pruebas'});
ylabel('Porcentaje de mejora ');
xlabel('Mejora del promedio de las tres pruebas');
```