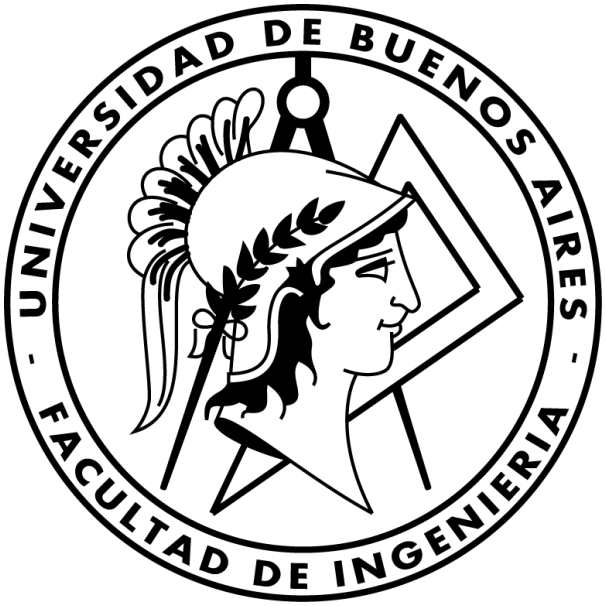
Sistemas Distribuidos II

(75.64)



Trabajo práctico final

Key-Value DB

Cuatrimestre: 2C-18

Padrón	Nombre y Apellido	Mail
92037	Germán Guzelj	german.guzelj@gmail.com

Motivación

Lograr confiabilidad en un sistema siempre es un desafío, y este problema se profundiza aún más cuando tratamos con un sistema distribuido. Sin embargo, este es un problema que no debemos ignorar, debido a la cantidad de beneficios que podemos obtener a cambio, como la resiliencia a fallas, escalabilidad, disponibilidad, por nombrar algunos ejemplos.

Un área que no escapa de esta problemática es el de las bases de datos. Cada día es más grande la cantidad sitios web que deben cumplir con ciertos requerimientos que solamente son alcanzables de manera distribuida (al menos con el hardware con el que se cuenta en el día de hoy), como por ejemplo la baja latencia y la alta disponibilidad. Por poner un número que refleja esta problemática, Amazon encontró que por cada 100 ms de latencia que se agrega al sitio se pierde un 1% de revenue.

Por otro lado, el uso de las bases de datos también cambió, existen múltiples situaciones que solamente demandan un acceso de clave valor. Algunos ejemplos que pueden resolverse con esta funcionalidad serían los carritos de compra, catálogo de productos, manejo de sesiones, y un largo etc. En sintonía con lo mencionado anteriormente, podemos agregar que este tipo de operaciones deben ser realizadas con una latencia muy baja, incluso con grandes cantidades de tráfico.

Esto, además, exige que no sea posible cumplir con las deseables propiedades ACID. En efecto, es conocido que las bases de datos que cumplen con estas propiedades tienden a sufrir problemas con la disponibilidad (CAP theorem). Es por este motivo que se prefiere sacrificar la consistencia, en pos de una mejor disponibilidad.

Por este motivo, en este trabajo práctico nos vamos a diseñar una base de datos clave valor distribuida.

Arquitectura

Para la definición de la arquitectura es conveniente destacar que la base de datos a realizar se puede caracterizar como una ***zero-hop* DHT** (distributed hash table), donde cada nodo mantiene suficiente información de ruteo para dirigir una request al nodo apropiado. En las siguientes subsecciones describiremos las distintas partes de la arquitectura de esta base de datos.

Interfaz

La manera de interactuar con la base de datos es a través de requests HTTP. Las dos operaciones posibles son:

Lectura

La lectura de un dato se realiza mediante una request a cualquier nodo con la siguiente información:

```
POST http://localhost:8080/
{
      "type" : "GET",
      "key" : "9"
}
```

Si el nodo que recibe la request contiene la información, la devuelve. Sino redirige la request al nodo que contiene el dato. Más adelante veremos de qué manera se puede determinar el nodo que contiene almacenado el valor. Las respuestas posibles son 2. Si el dato existe, la respuesta contiene un status 200 (OK), con el valor asociado a esa clave. Si el dato no existe, la respuesta tiene un status 404 (not found).

Escritura

La request que debe realizarse para guardar un dato es similar a la de lectura:

```
POST http://localhost:8080/
{
      "type" : "PUT",
      "key" : "1",
      "value" : "Valor para la key 1"
}
```

Esta operación no falla, ya que pueden darse dos situaciones posibles:

- No existía ningún valor para esa clave, por lo que se guarda el dato bajo esa clave.
- Ya existía un valor para esa clave, por lo que se pisa con el nuevo valor.

Notar que la interfaz no sigue los lineamientos de REST, donde una operación debe ser autodescriptiva. Es decir, uno podría esperar que la lectura sea realizada mediante un GET y una escritura mediante un PUT (en lugar de un POST para ambas). La respuesta es que un GET no permite enviar un body asociado a la request, por lo que la key que se envía debe enviarse a través de un query param (i.e. GET http://localhost?key=key1). El problema con este aproach es que la base de datos no exige ninguna restricción en cuanto a tamaño de key, mientras que en la práctica existen límites para el tamaño de la url.

Particionamiento

Otro de los problemas más importantes a definir es qué estrategia de particionamiento utilizar. Como mencionamos anteriormente, la idea es que no se agregue demasiada latencia, por lo que es necesario un mecanismo zero-hop. De esta forma, todos los nodos del cluster deben contener la información del resto de los nodos. Esta restricción es fuerte, en el sentido de que nos limita la cantidad de nodos que pueden formar parte del cluster (aunque para una base de datos no debería ser un problema).

Por ejemplo, una implementación rudimentaria podría ser: **node = hash(key) mod_N,** donde N es la cantidad de nodos del cluster. Por ejemplo, si tenemos 3 nodos:

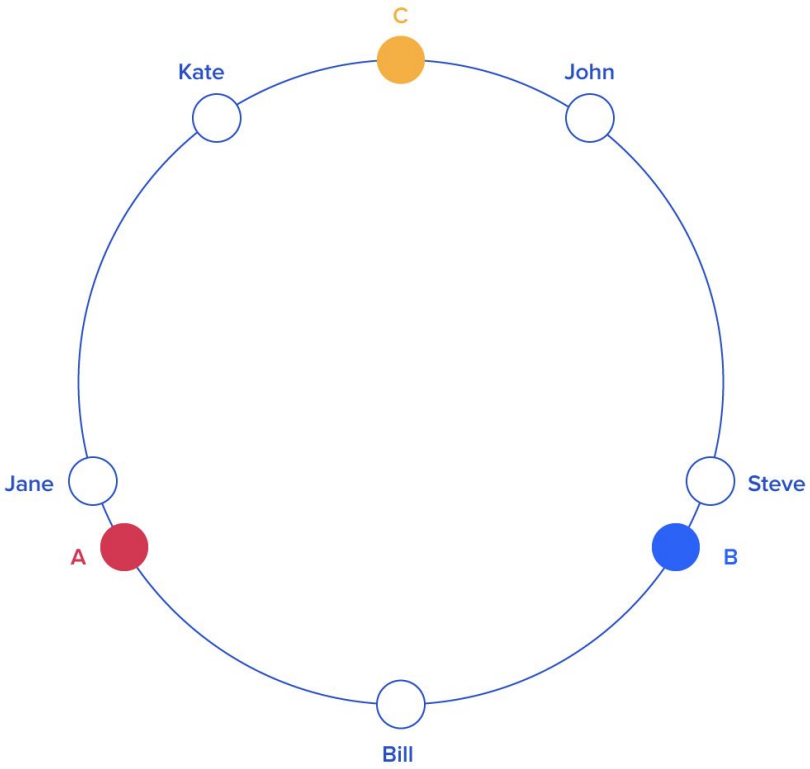
KEY	HASH	HASH mod 3
"john"	1633428562	2
"bill"	7594634739	0
"jane"	5000799124	1
"steve"	9787173343	0
"kate"	3421657995	2

El problema es que las claves son distribuidas en base a la **cantidad de nodos**, por lo que si se agrega o saca un nodo se tiene que re-hasear todo nuevamente. Siguiendo con el ejemplo anterior, si un nodo:

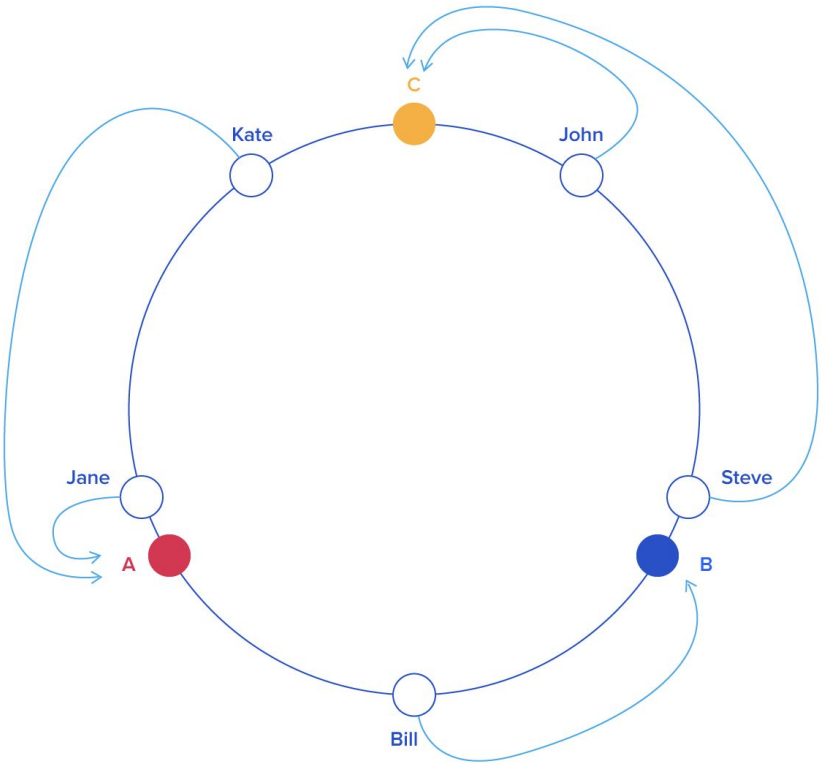
KEY	HASH	HASH mod 2
"john"	1633428562	0
"bill"	7594634739	1
"jane"	5000799124	0
"steve"	9787173343	1
"kate"	3421657995	1

Para resolver esto se utiliza **consistent hashing**, implementando un anillo lógico (el valor del hash más alto se envuelve alrededor del más chico). Se le asigna un valor aleatorio (token) a cada nodo, que representará su posición dentro del anillo. En consecuencia, para cada elemento que se quiera guardar se genera un hash de su clave, que nos dará como resultado una posición en el anillo, y luego se recorre el anillo en sentido horario (o antihorario) hasta encontrar un nodo. Por lo tanto, cada nodo se hace responsable de la región entre este y su nodo predecesor en el anillo.

De esta forma, si al ejemplo anterior le agregamos los nodos A, B y C en las siguientes posiciones:



Las claves serán asignadas de la siguiente manera:



La principal ventaja de este esquema es que agregar o sacar un nuevo nodo del sistema **solamente afecta a los nodos vecinos**, mientras el resto no percibe el cambio. Por otro lado, una de las desventajas es que la posición de los nodos es seleccionada de manera aleatoria, lo que puede resultar en una distribución **no** uniforme de las claves.

Otra mejora que se incluyó es el concepto de **nodo virtual.** La diferencia entre un nodo virtual y un nodo físico es que el nodo físico posee **varios** nodos virtuales. Internamente, lo que hace un nodo físico es elegir varios tokens, dando como resultado varios sectores del anillo. Estos sectores son los que termina manejando ese nodo.

La ventaja de agregar los nodos virtuales es que ahora cada vez que se agrega o saca un nodo el trabajo de migrar la información, a los otros nodos, se distribuye entre varios sectores.

Ring Membership

Como ya vimos, un aspecto importante del diseño de nuestro sistema es que los nodos puedan ingresar o salir del anillo sin mayores inconvenientes (por lo general debido a tareas de mantenimiento o fallas en los equipos). A diferencia de otros sistemas p2p, la cantidad de miembros no suele cambiar con frecuencia, por lo que se optó por un mecanismo **explícito** para agregar o sacar nodos.

Un administrador a cargo se conecta a algún nodo en particular, y agrega el cambio del anillo. El nodo que atiende el request persiste el cambio en un historial de eventos membership. Luego, el cambio es propagado a lo largo del cluster con un mecanismo que veremos más adelante.

Una propiedad interesante es que el estado del anillo puede reconstruirse a partir del historial de eventos. Cada evento tiene la información básica para cada nodo, como por ejemplo:

- Id: Un identificador único para cada nodo
- Timestamp: El momento exacto en que se creó ese nodo
- Tipo de evento: indica si se está creando un nuevo nodo o eliminando

- Nombre: Un nombre descriptivo para el nodo
- Url: La url donde podemos ubicarlo

Luego, solamente es cuestión de leer todos los eventos (ordenados cronológicamente), e ir aplicando cada cambio al anillo.

Todavía queda un problema a resolver, y son las particiones temporales que pueden generarse. Por ejemplo, si agregamos un nodo A al anillo y luego un nodo B, **tanto A como B desconocen la existencia del otro**, sin embargo ambos se consideran parte del anillo. Para evitar esto, algunos nodos juegan el rol de seed (**semilla**). Estos nodos son conocidos por todos los otros nodos del cluster.

Gossip protocol

Para este trabajo utilizaremos un protocolo distribuido para la detección de fallas, que le permitirá a cada nodo conocer sobre la partida o arribo de un nuevo nodo al cluster. El protocolo es del tipo gossip-based (basado en el chisme) y básicamente, cada nodo selecciona aleatoriamente otro nodo cada 1 segundo para reconciliar el estado del historial de membership.

En la medida que el tiempo transcurre el cambio es propagado a lo largo de todo el cluster, y es cada vez menos probable que un nodo desconozca lo que pasó en el anillo.

Este mecanismo sacrifica precisión a cambio de eficiencia ya que los métodos tradicionales de heartbeat pueden terminar produciendo una gran carga de tráfico en la red. Para más detalles sobre los distintos parámetros que se necesitan para configurar la frecuencia necesaria, es decir, los mensajes por unidad de tiempo pueden consultar el trabajo de Gupta y Goldszmidt.

Ejemplo práctico

El presente trabajo práctico fue desarrollado con java 8 y maven. En los siguientes pasos vamos a levantar un cluster. Primero clonamos el proyecto:

```
> git clone https://github.com/gguzelj/7563-DHT.git && cd 7563-DHT
```

```
> mvn clean install
```

```
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.18.1:test (default-test) @ controller ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ controller ---
[INFO] Building jar: /Users/german.guzelj/workspace/7563-DHT/controller/target/controlle
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.10.RELEASE:repackage (default) @ controller ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ controller ---
[INFO] Installing /Users/german.guzelj/workspace/7563-DHT/controller/target/controller-0
[INFO] Installing /Users/german.guzelj/workspace/7563-DHT/controller/pom.xml to /Users/g
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] dht ................................................ SUCCESS [  0.311 s]
[INFO] dto ................................................ SUCCESS [  1.375 s]
[INFO] node ............................................... SUCCESS [  2.229 s]
[INFO] controller ......................................... SUCCESS [  0.875 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.158 s
[INFO] Finished at: 2019-03-24T18:28:04-03:00
[INFO] Final Memory: 40M/284M
[INFO] ------------------------------------------------------------------------
~/workspace/7563-DHT  master
```

Como resultado, tendremos un jar con todas las dependencias empaquetadas. El jar se encuentra ubicado en el directorio `controller/target/controller-0.1-SNAPSHOT.jar`

Ahora levantamos un cluster con 2 nodos. Para levantar la aplicación ejecutamos el comando:

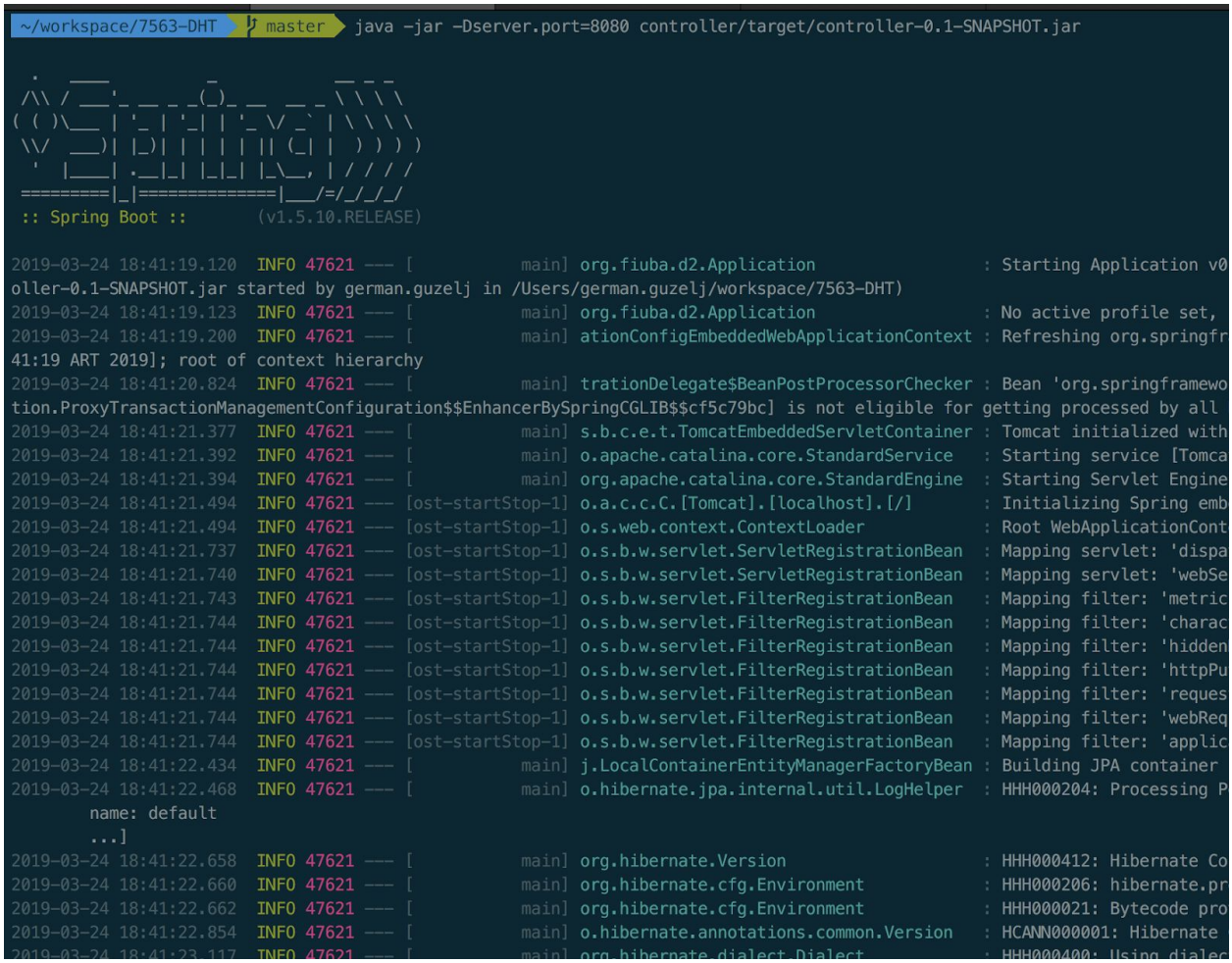
```
> java -jar controller/target/controller-0.1-SNAPSHOT.jar
```

Normalmente vamos a querer agregar algunos parámetros extra para configurar el cluster. Estos pueden ser:

1. **Server.port:** Puerto en el cual vamos a levantar la api
2. **Dht.node.amount-tokens:** Cantidad de tokens que vamos a utilizar para este nodo. Esto se traduce de manera directa en la cantidad de nodos virtuales que utilizaremos
3. **Dht.node.host:** La url donde está ubicado este nodo. Esta información es necesaria, ya que la aplicación no tiene forma de saber bajo qué ip la pueden ubicar externamente (ya que pueden existir varias interfaces de red). Esta ip viaja junto con los eventos de creación, de forma que los otros nodos puedan conocer a donde deben comunicarse.
4. **Dht.ring.seeds:** Un array con las urls de los nodos que son **semillas.** De esta forma, el nodo que se crea conoce con quien puede comunicarse para notificar del cambio en el anillo.

De esta forma, procedemos a levantar el primer nodo (que será semilla):

```
> java -jar -Dserver.port=8080
controller/target/controller-0.1-SNAPSHOT.jar
```



Una vez que la aplicación levanta podemos consultar el historial de eventos:

```
GET http://localhost:8080/events
```


```
[
    {
        "timestamp": 1553463685898,
        "type": "ADD",
        "nodeId": "26efe3ae-027d-43da-8171-d2aaad1f617c",
        "name": "Ted",
        "uri": "http://127.0.0.1:8080",
        "tokens": [
            {
                "value": "a0f1490a20d0211c997b44bc357e1972deab8ae3"
```

```
            },
            {
                "value": "adad2ca7ab313add6e955f704719e03d5229e4d0"
            }
        ]
    }
]
```

En este endpoint podemos consultar todos los eventos que se van propagando por el anillo. En esta caso solamente hay un evento, ya que tenemos un solo nodo en el anillo. A medida que vayamos agregando nodos vamos a encontrar esos eventos en esta respuesta. Como fue mencionado anteriormente, una propiedad interesante es que podemos reconstruir el estado actual del anillo aplicando el cambio correspondiente al anillo para cada evento (ordenado cronológicamente).

Luego, agregamos otro nodo. En este caso, lo vamos a levantar en el puerto 8081 y le indicaremos la dirección del nodo semilla, y además le indicaremos que genere 3 tokens, es decir, que se haga cargo de 3 secciones del anillo:

```
> java -jar -Dserver.port=8081
-Ddht.ring.seeds="http://127.0.0.1:8080" -Ddht.node.amount-tokens=3
controller/target/controller-0.1-SNAPSHOT.jar
```

Tras levantar la aplicación veremos en los logs del primer nodo (8080):

```
org.fiuba.d2.controller.EventController  : New event received:
MembershipEvent{timestamp=1553464232026, type=ADD, name='Khadijah',
uri='http://127.0.0.1:8081'}
```

Luego, si volvemos a consultar el historial de eventos, veremos que se agregó este nuevo nodo:

```
GET http://localhost:8080/events
```

```
[
    {
```

```
        "timestamp": 1553463685898,
        "type": "ADD",
        "nodeId": "26efe3ae-027d-43da-8171-d2aaad1f617c",
        "name": "Ted",
        "uri": "http://127.0.0.1:8080",
        "tokens": [
            {
                "value": "a0f1490a20d0211c997b44bc357e1972deab8ae3"
            },
            {
                "value": "adad2ca7ab313add6e955f704719e03d5229e4d0"
            }
        ]
    },
    {
        "timestamp": 1553464232026,
        "type": "ADD",
        "nodeId": "aa50245d-5023-4992-b33d-778a92394a03",
        "name": "Khadijah",
        "uri": "http://127.0.0.1:8081",
        "tokens": [
            {
                "value": "27f57cb359a8f86acf4af811c47a6380b4bb4209"
            },
            {
                "value": "60c79e75f9c2ea5f5aaf21ec2ad7d5b13d61f864"
            },
            {
                "value": "964992fde30239af2636655e58d714e73d8b5050"
            }
        ]
    }
]
```


Tras esto estamos listos para agregar nueva información. Esto lo hacemos con la siguiente request HTTP:

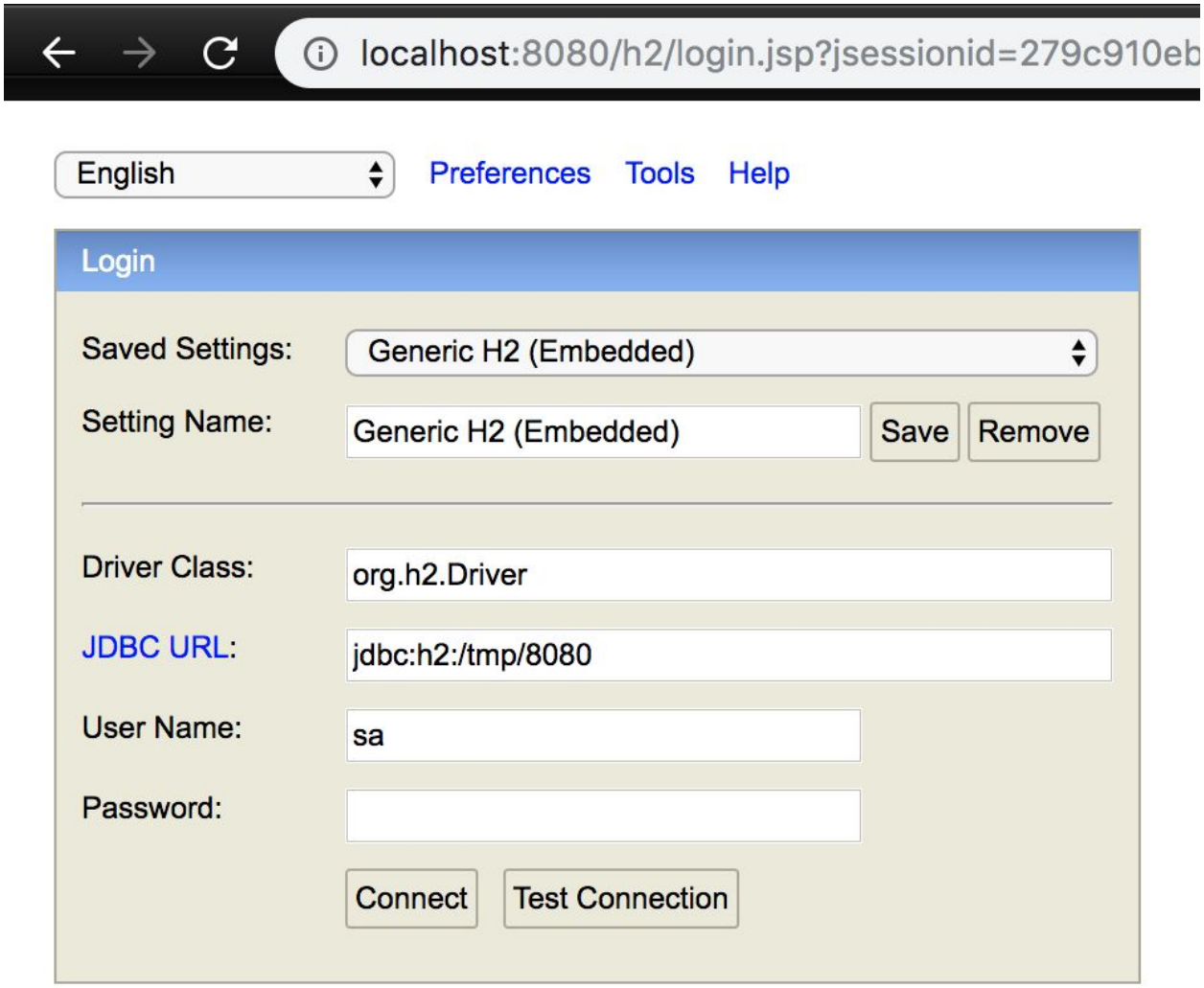
```
POST http://localhost:8080/events
```

```
{
      "type" : "PUT",
      "key" : "key_1",
      "value" : "value_1"
}
```

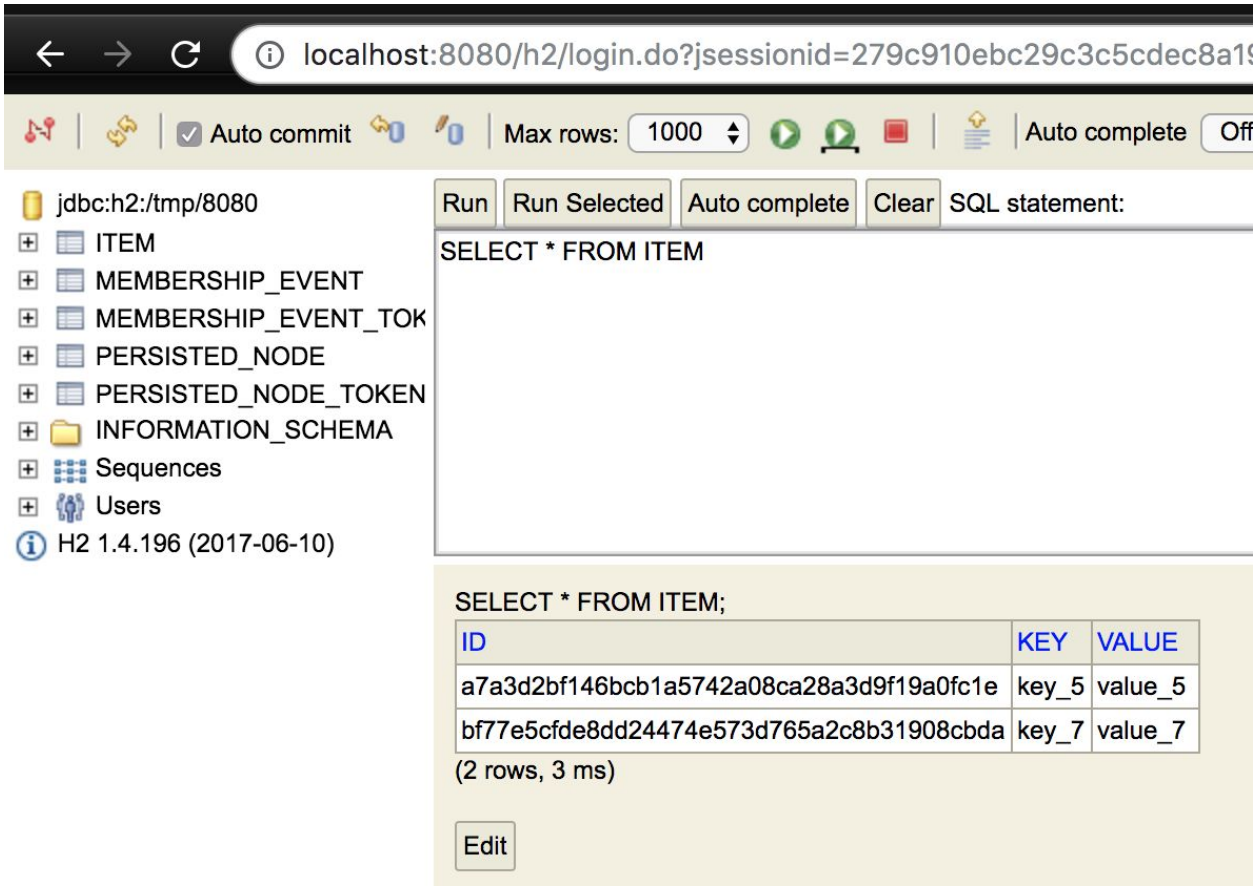
Lo que vamos a hacer se repetir este proceso otras diez veces más para generar entradas y luego ver cómo se distribuyen entre ambos nodos. Notar que todas las request las hicimos sobre el nodo alojado en el puerto 8080. Como este nodo conoce el anillo completo puede saber cada clave a qué nodo le corresponde almacenarla. Si la misma le corresponde a otro nodo vamos a poder ver en las líneas de logs la siguiente información:

```
Forwarding put request "key_2" to Khadijah[http://127.0.0.1:8081]
Forwarding put request "key_3" to Khadijah[http://127.0.0.1:8081]
Forwarding put request "key_3" to Khadijah[http://127.0.0.1:8081]
Forwarding put request "key_4" to Khadijah[http://127.0.0.1:8081]
Forwarding put request "key_6" to Khadijah[http://127.0.0.1:8081]
Forwarding put request "key_6" to Khadijah[http://127.0.0.1:8081]
Forwarding put request "key_8" to Khadijah[http://127.0.0.1:8081]
Forwarding put request "key_8" to Khadijah[http://127.0.0.1:8081]
Forwarding put request "key_9" to Khadijah[http://127.0.0.1:8081]
Forwarding put request "key_10" to Khadijah[http://127.0.0.1:8081]
```

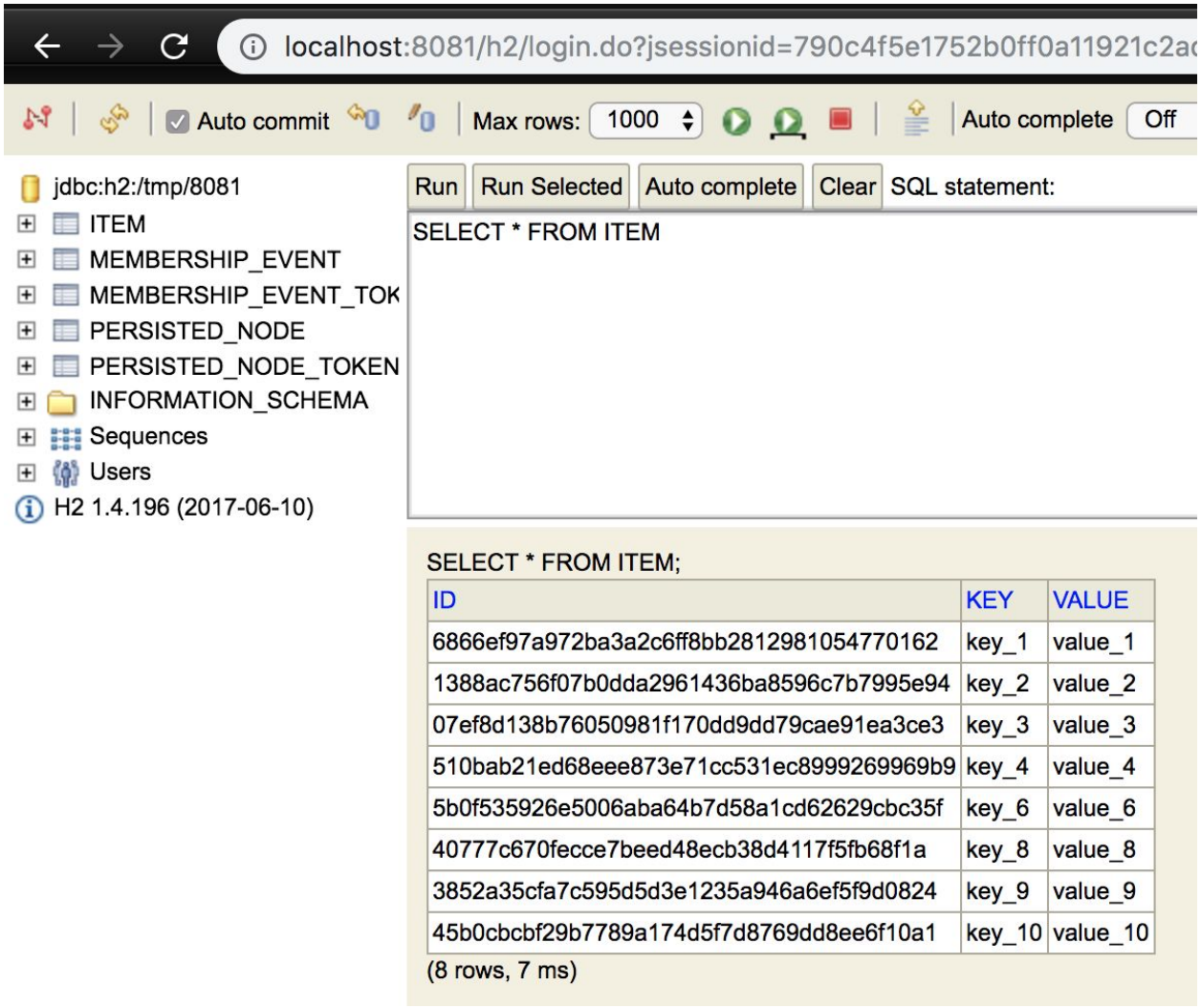
El proyecto además trae un cliente para poder ver la información almacenada en la base de datos de cada nodo. Lo único que hay que hacer es entrar en un browser a http://localhost:8080/h2/



El archivo con la información de la base de datos se guarda en /tmp/{puerto}. De esta forma, podemos conectarnos a cada uno de los nodos. Por ejemplo, para el del 8080 tenemos:



Mientras que para el nodo del 8081 tenemos:



Notar que al tener mayor cantidad de tokens para el nodo 2 es natural que el mismo almacene una mayor cantidad de información. El siguiente paso es agregar un nuevo nodo y ver cómo se distribuyen las claves. La aplicación detecta cuando se agrega un nuevo nodo, y migra la información automáticamente. Ejecutamos el siguiente comando:

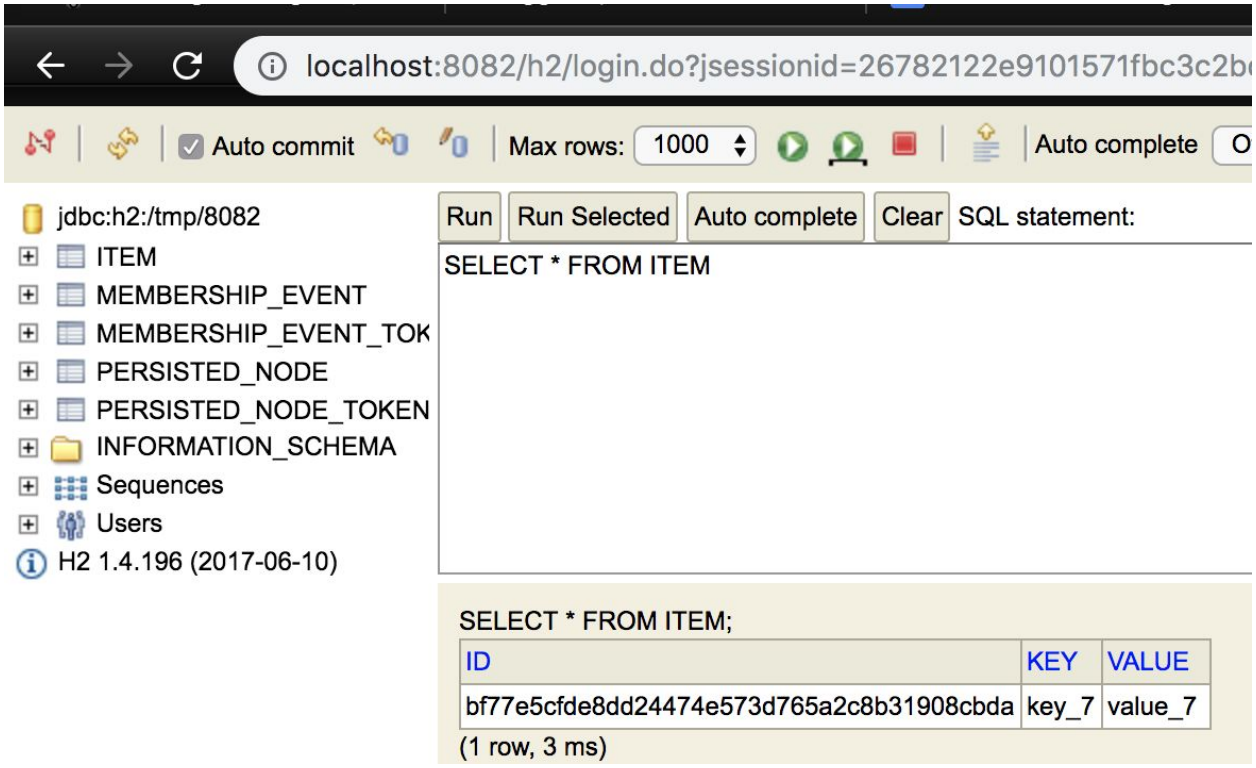
```
java -jar -Dserver.port=8082 -Ddht.ring.seeds="http://127.0.0.1:8080"
controller/target/controller-0.1-SNAPSHOT.jar
```

Luego, en el historial de eventos podremos ver:

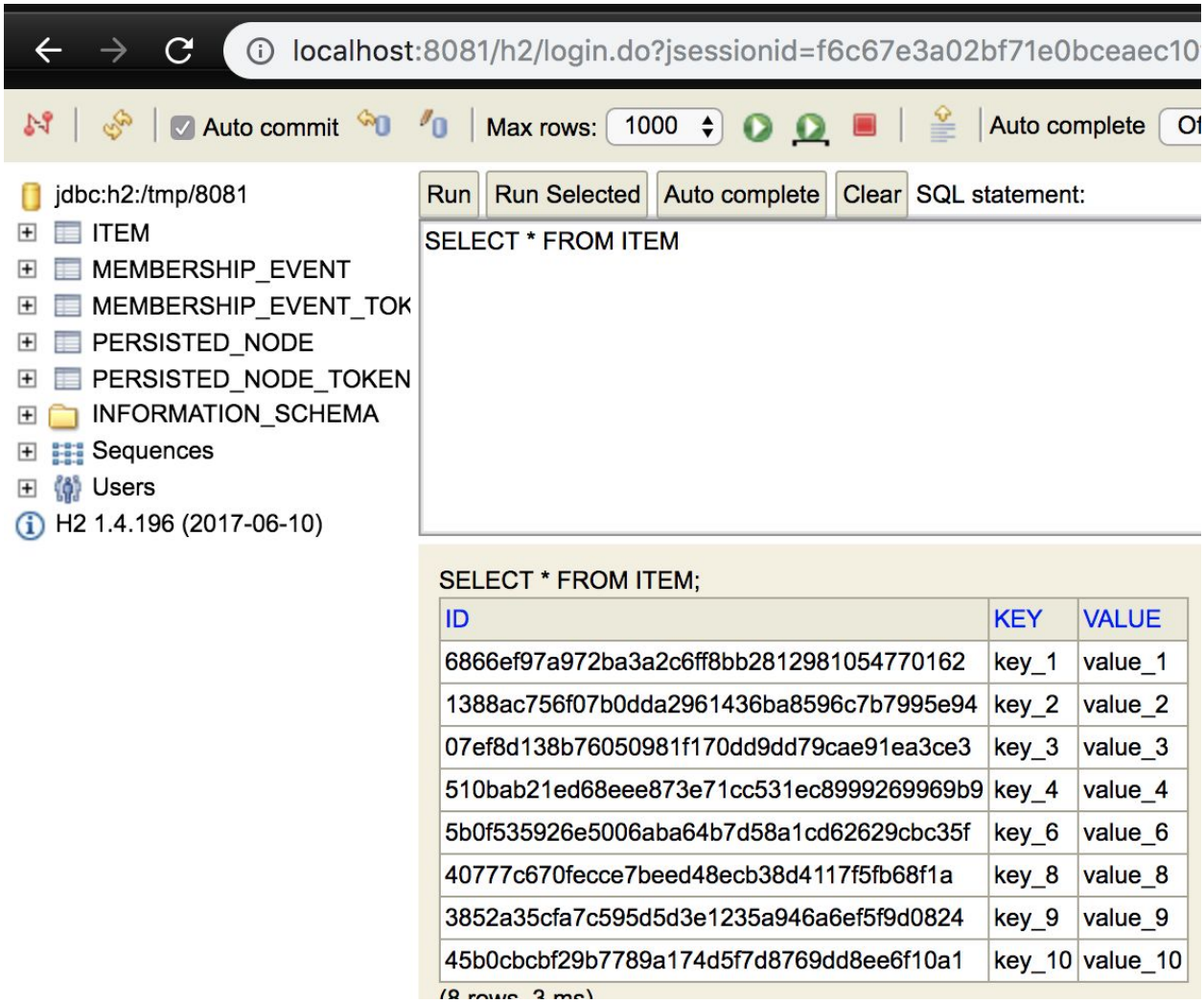
```
[
    {
        "timestamp": 1553463685898,
        "type": "ADD",
        "nodeId": "26efe3ae-027d-43da-8171-d2aaad1f617c",
        "name": "Ted",
        "uri": "http://127.0.0.1:8080",
        "tokens": [
            {
                "value": "a0f1490a20d0211c997b44bc357e1972deab8ae3"
            },
            {
                "value": "adad2ca7ab313add6e955f704719e03d5229e4d0"
            }
        ]
    },
    {
        "timestamp": 1553464232026,
        "type": "ADD",
        "nodeId": "aa50245d-5023-4992-b33d-778a92394a03",
        "name": "Khadijah",
        "uri": "http://127.0.0.1:8081",
        "tokens": [
            {
                "value": "27f57cb359a8f86acf4af811c47a6380b4bb4209"
            },
            {
                "value": "60c79e75f9c2ea5f5aaf21ec2ad7d5b13d61f864"
            },
            {
                "value": "964992fde30239af2636655e58d714e73d8b5050"
            }
        ]
    },
    {
        "timestamp": 1553466015562,
        "type": "ADD",
        "nodeId": "b6544739-cbd2-48de-9559-2a7367aaf31f",
```

```
        "name": "Larraine",
        "uri": "http://127.0.0.1:8082",
        "tokens": [
            {
                "value": "c4dd3c8cdd8d7c95603dd67f1cd873d5f9148b29"
            },
            {
                "value": "c78ebd3c85a39a596d9f5cfd2b8d240bc1b9c125"
            }
        ]
    }
]
```


Y al consultar la información de este nuevo nodo veremos:



Como vemos, de los diez datos agregados solamente uno cae en el rango que maneja este nodo. Naturalmente, si consultamos la información almacenada del nodo 8081 (que anteriormente tenía la clave “key_7”) ya no debería tener este dato:



Ahora podemos agregar más información, y eliminar los nodos ubicados en el 8081 y 8082 (es decir, solamente dejar el nodo semilla), y ver como toda la información es migrada a este nodo.
Para esto, tenemos que invocar al endpoint:

```
POST http://localhost:8081/shutdown
```

```
POST http://localhost:8082/shutdown
```

De esta forma, se generan los eventos de que estos nodos fueron destruidos:

```
[
    {
        "timestamp": 1553463685898,
        "type": "ADD",
        "nodeId": "26efe3ae-027d-43da-8171-d2aaad1f617c",
        "name": "Ted",
        "uri": "http://127.0.0.1:8080",
        "tokens": ...
    },
    {
        "timestamp": 1553464232026,
        "type": "ADD",
        "nodeId": "aa50245d-5023-4992-b33d-778a92394a03",
        "name": "Khadijah",
        "uri": "http://127.0.0.1:8081",
        "tokens": …
    },
    {
        "timestamp": 1553466015562,
        "type": "ADD",
        "nodeId": "b6544739-cbd2-48de-9559-2a7367aaf31f",
        "name": "Larraine",
        "uri": "http://127.0.0.1:8082",
        "tokens": ...
    },
    {
        "timestamp": 1553466559369,
        "type": "REMOVE",
        "nodeId": "b6544739-cbd2-48de-9559-2a7367aaf31f",
        "name": "Larraine",
        "uri": "http://127.0.0.1:8082",
        "tokens": ...
    },
    {
        "timestamp": 1553466617901,
        "type": "REMOVE",
        "nodeId": "aa50245d-5023-4992-b33d-778a92394a03",
        "name": "Khadijah",
```

```
        "uri": "http://127.0.0.1:8081",
        "tokens": ...
    }
]
```

Y al consultar la información almacenada por el nodo 8080:

SELECT * FROM ITEM;

ID	KEY ▼	VALUE
6866ef97a972ba3a2c6ff8bb28129810547701 62	key_1	value_1
45b0cbcbf29b7789a174d5f7d8769dd8ee6f10a1	key_10	value_10
09f15a5f3ea1549a6b976fa493675edcc1ab4bc6	key_11	value_11
b8d1ecc014e691fcb6199dd2bcf8875da91e606f	key_12	value_12
cc8e21b3b375b352cf9fc0224f7ac17c47d0da3e	key_13	value_13
df522bb2eb4b75025ae5a0a2097ff6a74be5b579	key_14	value_14
4379a2670a2708a3e2bd69e95d09a90f922be38b	key_15	value_15
463dfc8d025f50452d034d1eab12d2b12fdab2e4	key_16	value_16
83ca8bd79a0d94840d2dc58340570db7ca127f56	key_17	value_17
f5a8236b179fbe6b47c30aa93c62d7e14545ff1c	key_18	value_18
2879e92e6cfbf0dd722b2355d3e3898e281b8923	key_19	value_19
1388ac756f07b0dda2961436ba8596c7b7995e94	key_2	value_2
ba37975d837c55a30658a6b026c5f7937b85262e	key_20	value_20
07ef8d138b76050981f170dd9dd79cae91ea3ce3	key_3	value_3
510bab21ed68eee873e71cc531ec8999269969b9	key_4	value_4
a7a3d2bf146bcb1a5742a08ca28a3d9f19a0fc1e	key_5	value_5
5b0f535926e5006aba64b7d58a1cd62629cbc35f	key_6	value_6
bf77e5cfde8dd24474e573d765a2c8b31908cbda	key_7	value_7
40777c670fecce7beed48ecb38d4117f5fb68f1a	key_8	value_8
3852a35cfa7c595d5d3e1235a946a6ef5f9d0824	key_9	value_9

(20 rows, 4 ms)

Trabajo futuro

Versionado

Cada modificación realizada sobre la información del sistema es tratada como una nueva versión inmutable. Al no haber una consistencia fuerte, puede haber distintas versiones de un mismo objeto. Estas diferencias deben reconciliarse, y por lo general es utilizado algún mecanismo como **vector clocks**.

Replica Synchronization

El tiempo durante el cual una réplica esta caída es posible que sucedan cambios que necesitan ser reconciliados. La idea es que cuando una réplica vuelva a estar disponible se pueda volver a sincronizar, posiblemente mediante alguna estructura de datos como los **merkle trees**.

Referencias

https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.147.1818&rep=rep1&type=pdf