Large-Scale and Multi-Structured Databases

Key – Value Databases: Redis

Part 2

Prof. Pietro Ducange

Eng. José Corcuera

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ DI PISA

CROSSLAB Innovation for industry 4.0

Objective of this Class

- To learn what the key **eviction** policies are and how they work.
- To learn how the **persistence** works.
- To learn how to activate the **High Availability (HA) - Replication** feature.
- To learn how **Horizontal Scaling** works and how to configure it.
- To review **Keyspace notifications**.
- To learn other features of Redis: **Publisher/Subscriber** scheme.
- **Jedis** part 2.
- Use cases scenarios.
- Exercises.

Eviction

- A configured eviction policy (***maxmemory-policy***) is activated when the Max. Memory limit (***maxmemory***) is reached.
- There exists different policies:
  - **noeviction**: New values aren't saved when memory limit is reached. When a database uses replication, this applies to the primary database.
  - **allkeys-lru**: Keeps most recently used keys; removes least recently used (LRU) keys.
  - **allkeys-lfu**: Keeps frequently used keys; removes least frequently used (LFU) keys.
  - **volatile-lru**: Removes least recently used keys with the expire field set to true.
  - **volatile-lfu**: Removes least frequently used keys with the expire field set to true.
  - **allkeys-random**: Randomly removes keys to make space for the new data added.
  - **volatile-random**: Randomly removes keys with expire field set to true.
  - **volatile-ttl**: Removes keys with expire field set to true and the shortest remaining time-to-live (TTL) value.
- Default values of these properties:

```
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "noeviction"
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379>
```

```
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "0"
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379>
```

Unlimited memory until the OS runs out of RAM and kills the process.

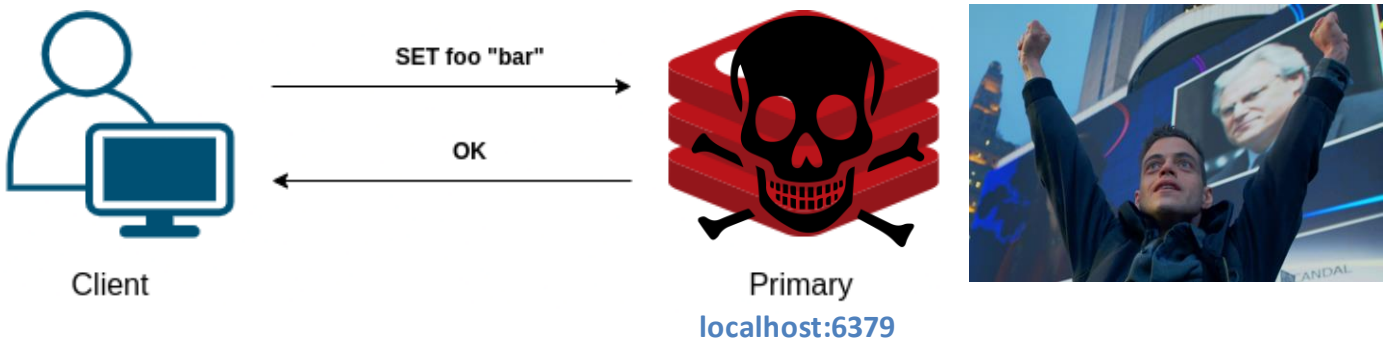
For more information: https://redis.io/docs/manual/eviction/

Persistence

- Redis database (RDB) writes data into a durable storage, such as a solid-state-disk (SSD).
- Persistence options are:
  - **RDB**: RDB performs point-in-time snapshots of your dataset at specific intervals.
  - **Append Only File (AOF)**: every write operation received by the server is written into a log file.
  - **No persistence**: your dataset exists if the server is running.
  - **RDB + AOF**: combination of the first two options. The AOF file is used to reconstruct the original dataset.

For more information: https://redis.io/docs/manual/persistence/

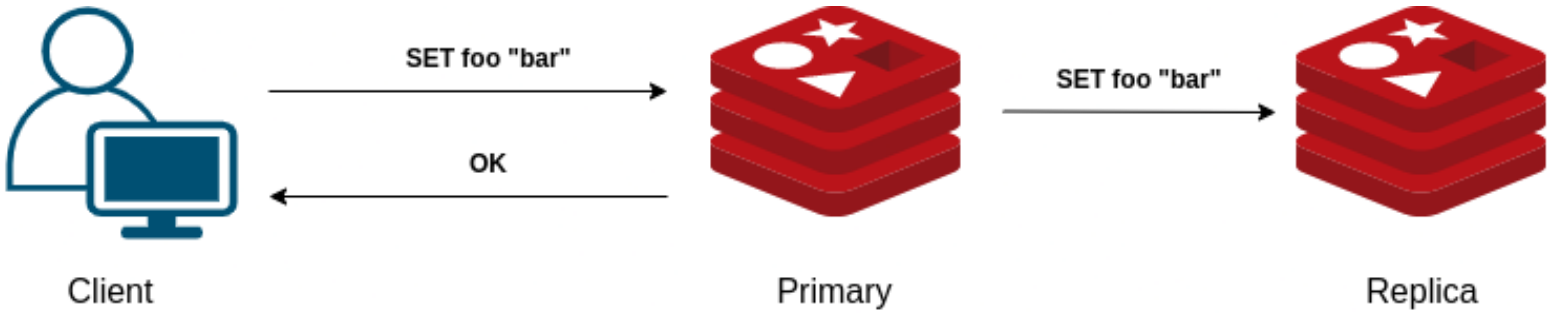
Simple Database



So far, we have worked with this configuration.

What happens if the primary server goes down?

High Availability (HA) - Replication



When the primary server goes down, the replica server takes over.

Redis uses by default asynchronous replication, but it is possible to perform synchronous replication of certain data.

What is the minimum number of replicas do I need to have to ensure HA?

It depends…

For more information: https://redis.io/docs/manual/replication/

Clustered Database (Scaling)

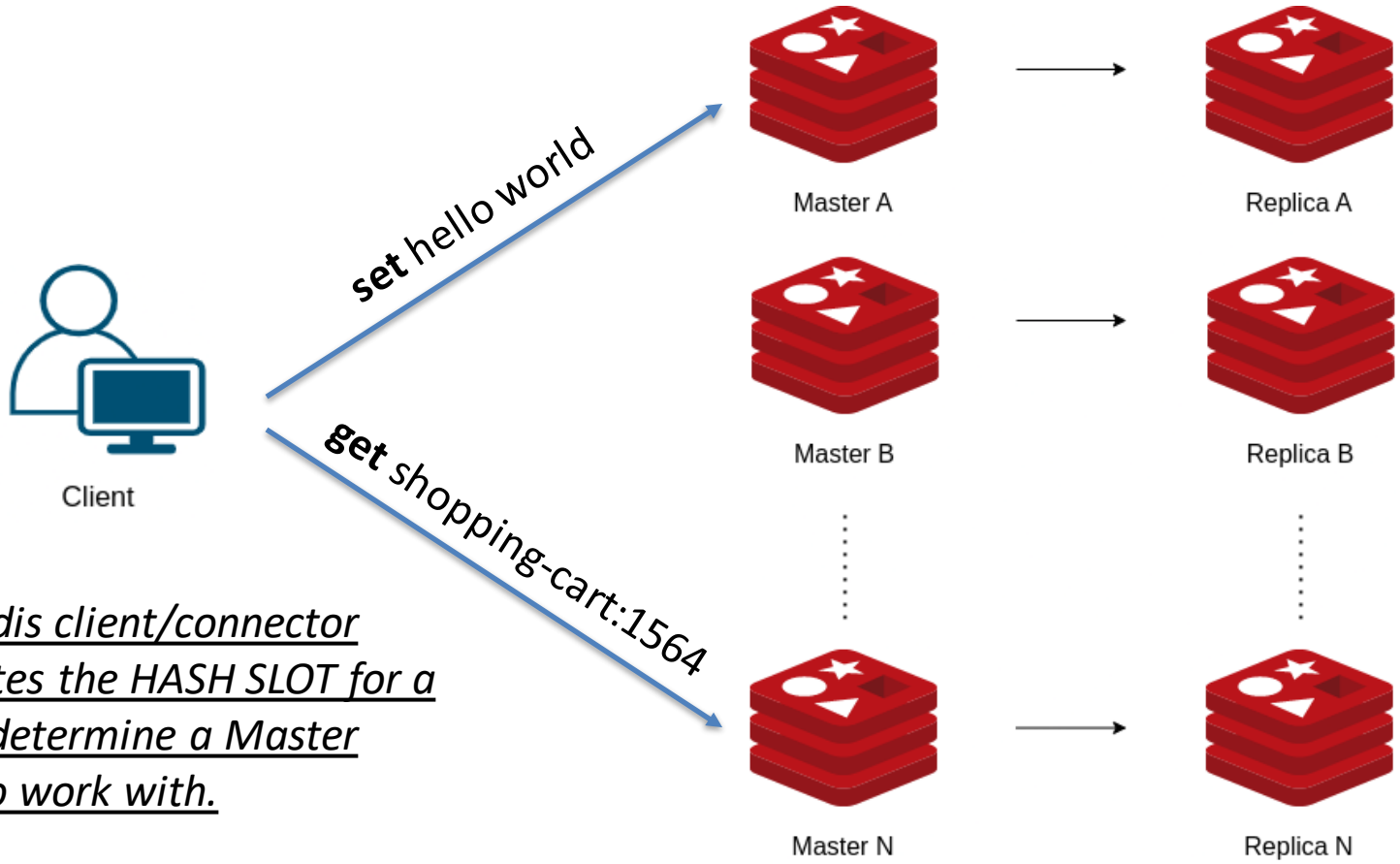
- Keys are distributed across cluster nodes.
- To determine in which node a key is stored a HASH_SLOT must be computed:
  HASH_SLOT = CRC16(**key**) mod 16384
- There are 16384 hash slots in Redis Cluster and each node of the cluster has assigned a range of them.
- In this configuration:
  - Node A contains hash slots from 0 to 5500.
  - Node B contains hash slots from 5501 to 11000.
  - Node C contains hash slots from 11001 to 16383.

Node A

Node B

Node C

HA Clustered Database



The Redis client/connector computes the HASH SLOT for a key to determine a Master node to work with.

Replication - Configuration

- Edit your **redis.conf** in each Redis instance:

Primary, IP: 173.18.15.2	**Replica**, IP: 172.18.15.3
bind **0.0.0.0** protected-mode **no** requirepass **custompassword**	bind **0.0.0.0** replicaof **173.18.15.2 6379** masterauth **custompassword**

- Restart your instances and test.
- protected-mode: *By default, protected mode is enabled. You should disable it only if you are sure, you want clients from other hosts to connect to Redis even if no authentication is configured, nor a specific set of interfaces are explicitly listed using the "bind" directive.*

HA Clustered Database – Configuration (1)

Let's create a cluster of 6 Redis instances in our local station.

1. Create 6 folders from 7000 to 7005. Each of these values represent the port number to be used in that Redis instance.

```
jose@uss-defiant:~/repository/UNIPI/LSMSD/redis-cluster$ pwd
/home/jose/repository/UNIPI/LSMSD/redis-cluster
jose@uss-defiant:~/repository/UNIPI/LSMSD/redis-cluster$ ls
7000  7001  7002  7003  7004  7005
jose@uss-defiant:~/repository/UNIPI/LSMSD/redis-cluster$
```

2. Inside each folder created in point (1), create the file **redis.conf** with the following content (*do not forget to update the port value*):

port **7000**
cluster-enabled **yes**
cluster-config-file **nodes.conf**
cluster-node-timeout **5000**
appendonly **yes**

HA Clustered Database – Configuration (2)

3. Start each Redis instance by running the following command in each folder:

redis-server ./redis.conf

4. To create a cluster, run the following command:

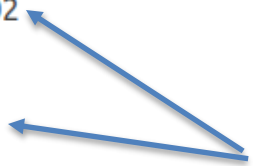
redis-cli --cluster create 127.0.0.1:7000 127.0.0.1:7001 127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 --cluster-replicas **1**

The option --cluster-replicas 1 means that we want a replica for every master created.

HA Clustered Database – Configuration (3)

5. Test your configuration by creation some keys:

```
jose@uss-defiant:~$ redis-cli -c -p 7000
127.0.0.1:7000> set foo bar
-> Redirected to slot [12182] located at 127.0.0.1:7002
OK
127.0.0.1:7002> set hello world
-> Redirected to slot [866] located at 127.0.0.1:7000
OK
127.0.0.1:7000> get foo
-> Redirected to slot [12182] located at 127.0.0.1:7002
"bar"
127.0.0.1:7002> get hello
-> Redirected to slot [866] located at 127.0.0.1:7000
"world"
127.0.0.1:7000>
```



Keys are distributed across master nodes.

6. To stop the cluster you can either stop each Redis instance or you can execute the script "create-cluster stop". Find the utils/create-cluster directory in the Redis distribution.

For more information: https://redis.io/docs/manual/scaling/#create-a-redis-cluster

Keyspace notifications

- Redis offers monitoring changes on keys and values in real time.
- Clients must subscribe to Pub/Sub channels to receives these events.
- Two types of events: Key-space and Key-event.
- Example of events:
  - Keys expiring in the database.
  - All the commands affecting a given key.

E: Keyevent events, published with __keyevent@<db>__ prefix.
x: Expired events (events generated every time a key expires)

```
jose@uss-defiant:~$ redis-cli
127.0.0.1:6379> subscribe __keyevent@0__:expired
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "__keyevent@0__:expired"
3) (integer) 1
```

```
jose@uss-defiant:~$ redis-cli
127.0.0.1:6379> config set notify-keyspace-events Ex
OK
127.0.0.1:6379> set test 10 ex 20
OK
127.0.0.1:6379>
```

After 20 seconds:

```
jose@uss-defiant:~$ redis-cli
127.0.0.1:6379> subscribe __keyevent@0__:expired
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "__keyevent@0__:expired"
3) (integer) 1
1) "message"
2) "__keyevent@0__:expired"
3) "test"
```

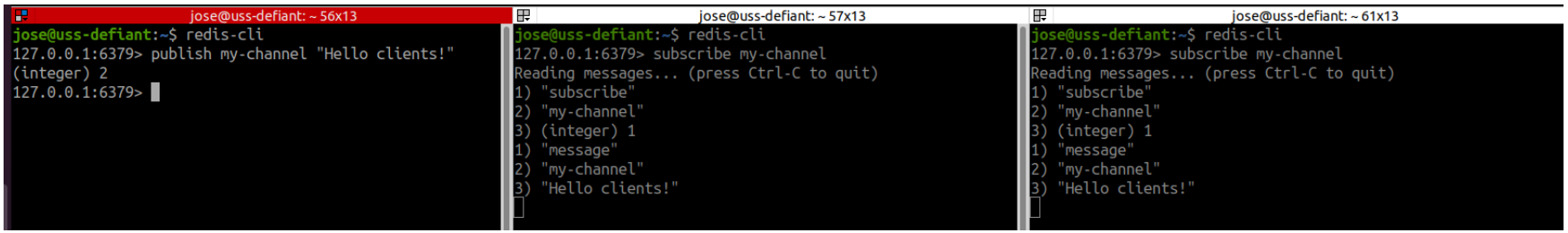
```
jose@uss-defiant:~$ redis-cli
127.0.0.1:6379> config set notify-keyspace-events Ex
OK
127.0.0.1:6379> set test 10 ex 20
OK
127.0.0.1:6379> get test
(nil)
127.0.0.1:6379>
```

By default, Redis support 16 databases (index from 0-15). Each database provides a distinct keyspace, independent from the others

For more information: https://redis.io/docs/manual/keyspace-notifications/

Publisher – Subscriber

- Redis implements the Publish/Subscribe messaging paradigm by offering the SUBSCRIBE, UNSUBSCRIBE and PUBLISH commands.
- A client must perform the SUBSCRIBE operation to specific channel(s).
- A publication on a channel will be notified to all the subscripted clients.
- To stop receiving notifications, the UNSUBSCRIBE command need to be executed.



Publisher Subscriber 1 Subscriber 2

Jedis in action (1)

- **Connecting to a Clustered Database:**

```
import redis.clients.jedis.*;
import java.util.HashSet;
import java.util.Set;

Set<HostAndPort> jedisClusterNodes = new HashSet<HostAndPort>();
jedisClusterNodes.add(new HostAndPort("127.0.0.1", 7001));
jedisClusterNodes.add(new HostAndPort("127.0.0.1", 7002));
jedisClusterNodes.add(new HostAndPort("127.0.0.1", 7003));

try (JedisCluster jedis = new JedisCluster(jedisClusterNodes)){
      /*
       do stuff here
    */
}
```

Jedis in action (2)

- **Receiving key space notifications (Expired keys):**

```
JedisPubSub jedisPubSub = new JedisPubSub() {
  @Override
  public void onPMessage(String  pattern, String channel, String message) {
   System.out.println("Pattern:  " + pattern + ", channel: " + channel + ", message: " + message);
  }
};
....
try (Jedis jedis = pool.getResource())  {
      jedis.psubscribe(jedisPubSub  , "__keyevent@0__:expired");
}
```

Do not forget to activate the event notification when a key expires. Run this command in Redis:

config set notify-keyspace-events Ex

Jedis in action (3)

- **Publisher/Subscriber: defining a subscriber**

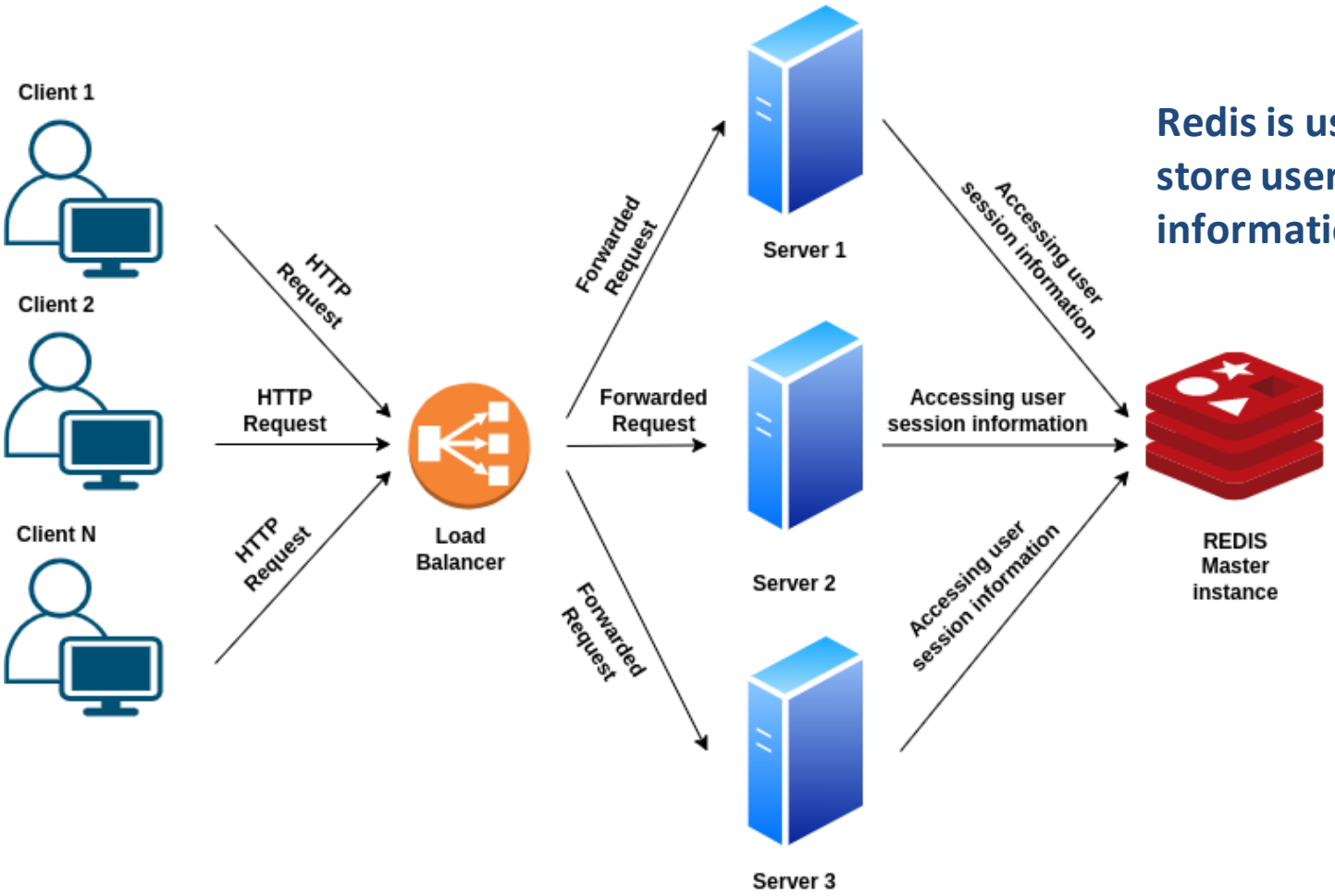
```
JedisPubSub jedisPubSub = new JedisPubSub() {

  @Override
  public void onMessage(String channel, String message) {
    System.out.println("Channel " + channel + " has sent a message : " + message );
  }

  @Override
  public void onSubscribe(String channel, int subscribedChannels) {
    System.out.println("Client is Subscribed to channel : "+ channel);
    System.out.println("Client is Subscribed to "+ subscribedChannels + " no. of channels");
  }

  @Override
  public void onUnsubscribe(String channel, int subscribedChannels) {
    System.out.println("Client is Unsubscribed from channel : "+ channel);
    System.out.println("Client is Subscribed to "+ subscribedChannels + " no. of channels");
  }

};
....
try (Jedis jedis = pool.getResource()) {
     jedis.subscribe(jedisPubSub, "Channel1", "Channel2");
}
```

Jedis in action (4)

- **Publisher/Subscriber: publishing messages**

```
try (Jedis jedis = pool.getResource()) {

    /* Publishing message to channel Channel1*/
    jedis.publish("Channel1", "First message to channel Channel1");

    /* Publishing message to channel Channel2*/
    jedis.publish("Channel2", "First message to channel Channel2");

    /* Publishing message to channel Channel1*/
    jedis.publish("Channel1", "Second message to channel Channel1");

    /* Publishing message to channel Channel2*/
    jedis.publish("Channel2", "Second message to channel Channel2");

}
```

Use case scenario: Web Server Session Externalization



Redis is used to store user web information session.

Exercise 1: Setting up a HA Clustered Database

For this exercise you can work in groups. It is requested:

1. To create a cluster of 6 nodes with cluster-replica = 1.
2. To create a Maven application that connects to this cluster an defines 100 random keys with different values (use the namespace exercise1).
*3. To verify the number of keys in each master instance. To do so, you can connect via **redis-cli** to that instance and count the number of keys created (you can use the command KEYS app:*).*

Exercise 2: Design choices

You were hired as a Software Engineer in a well-known Company. This company implemented an E-commerce and you have asked to modelling how the information of shopping carts can be stored into a Redis database. Your design must take into consideration the following requirements:

1. *It must be easy to know, for each customer, which products are in their shopping cart.*
2. *It must be cheap the processing in answering what the potential income from sales could be.*
3. *Suppose a product is not more active (or deleted) and many users have that product in their shopping cart, it must be easy to remove it from them.*

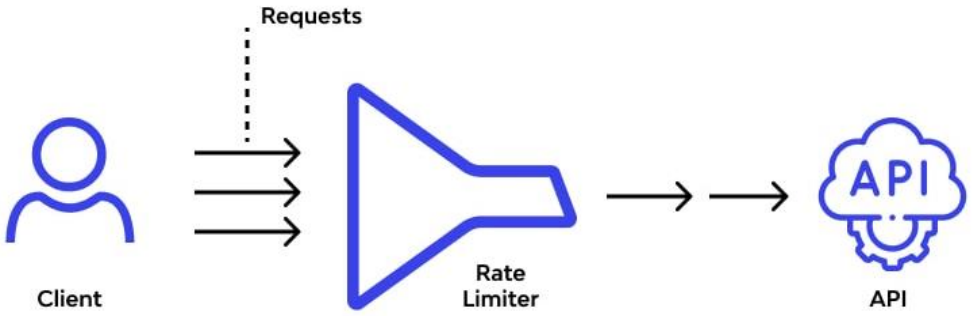
You must motivate your solution and mention possible problems/situations that could impact it.

Exercise 3: Rate limiting

Create a Java application that limit the number of request/invocations to a "method" made by threads. Your application must:

- *Create a class MyAPI which defines a method named "call" (it receives the name of a thread). This class defines a quota of 100 invocations every minute for that method. If there is quota left, this method prints the message "HTTP 200 <thread name here>" otherwise "HTTP 429 <thread name here>".*

- *Create N = 10 threads. Each of them is going to invoke the method defined in the previous call every second in 10 minutes interval.*



Exercise 4: Online auction/bid

Create a Java application that emulates the behavior of an auction. Your application must:

- *Create a class MyAuction which defines a method named "bid" (it receives the name of a thread and an amount of money). The auction starts with an initial price, and it ends after 60 seconds. If your bid is lower than the current bid then the message "Bid rejected <thread name here>" is returned otherwise, your bid is accepted and the message "Bid accepted <thread name here> <amount>".*
- *Create N = 10 threads. Each of them is going to invoke the method defined in the previous call every second in 1 minute interval. You can generate the amount of money to bid in the interval from 1 to 100.*
- *When the auction is over, the name of the winner is printed with his bid.*

References

- https://redis.io/docs/manual/
- https://redis.com/redis-best-practices/communication-patterns/pub-sub/
- https://redis.com/redis-best-practices/basic-rate-limiting/
- https://redis.com/blog/5-key-takeaways-for-developing-with-redis/