Redis cluster tutorial

This document is a gentle introduction to Redis Cluster, that does not use complex to understand distributed systems concepts. It provides instructions about how to setup a cluster, test, and operate it, without going into the details that are covered in the Redis Cluster specification but just describing how the system behaves from the point of view of the user.

However this tutorial tries to provide information about the availability and consistency characteristics of Redis Cluster from the point of view of the final user, stated in a simple to understand way.

Note this tutorial requires Redis version 3.0 or higher.

If you plan to run a serious Redis Cluster deployment, the more formal specification is a suggested reading, even if not strictly required. However it is a good idea to start from this document, play with Redis Cluster some time, and only later read the specification.

Redis Cluster 101

Redis Cluster provides a way to run a Redis installation where data is **automatically sharded across multiple Redis nodes**.

Redis Cluster also provides **some degree of availability during partitions**, that is in practical terms the ability to continue the operations when some nodes fail or are not able to communicate. However the cluster stops to operate in the event of larger failures (for example when the majority of masters are unavailable).

So in practical terms, what do you get with Redis Cluster?

- The ability to **automatically split your dataset among multiple nodes**.
- The ability to **continue operations when a subset of the nodes are experiencing failures** or are unable to communicate with the rest of the cluster.

Redis Cluster TCP ports

Every Redis Cluster node requires two TCP connections open. The normal Redis TCP port used to serve clients, for example 6379, plus the port obtained by adding 10000 to the data port, so 16379 in the example.

This second *high* port is used for the Cluster bus, that is a node-to-node communication channel using a binary protocol. The Cluster bus is used by nodes for failure detection, configuration update, failover authorization and so forth. Clients should never try to communicate with the cluster bus port, but always with the normal Redis command port, however make sure you open both ports in your firewall, otherwise Redis cluster nodes will be not able to communicate.

The command port and cluster bus port offset is fixed and is always 10000.

Note that for a Redis Cluster to work properly you need, for each node:

1. The normal client communication port (usually 6379) used to communicate with clients to be open to all the clients that need to reach the cluster, plus all the other cluster nodes (that use the client port for keys migrations).
2. The cluster bus port (the client port + 10000) must be reachable from all the other cluster nodes.

If you don't open both TCP ports, your cluster will not work as expected.

The cluster bus uses a different, binary protocol, for node to node data exchange, which is more suited to exchange information between nodes using little bandwidth and processing time.

Redis Cluster and Docker

Currently Redis Cluster does not support NATted environments and in general environments where IP addresses or TCP ports are remapped.

Docker uses a technique called *port mapping*: programs running inside Docker containers may be exposed with a different port compared to the one the program believes to be using. This is useful in order to run multiple containers using the same ports, at the same time, in the same server.

In order to make Docker compatible with Redis Cluster you need to use the **host networking mode** of Docker. Please check the `--net=host` option in the Docker documentation for more information.

Redis Cluster data sharding

Redis Cluster does not use consistent hashing, but a different form of sharding where every key is conceptually part of what we call an **hash slot**.

There are 16384 hash slots in Redis Cluster, and to compute what is the hash slot of a given key, we simply take the CRC16 of the key modulo 16384.

Every node in a Redis Cluster is responsible for a subset of the hash slots, so for example you may have a cluster with 3 nodes, where:

- 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.

This allows to add and remove nodes in the cluster easily. For example if I want to add a new node D, I need to move some hash slot from nodes A, B, C to D. Similarly if I want to remove node A from the cluster I can just move the hash slots served by A to B and C. When the node A will be empty I can remove it from the cluster completely.

Because moving hash slots from a node to another does not require to stop operations, adding and removing nodes, or changing the percentage of hash slots hold by nodes, does not require any downtime.

Redis Cluster supports multiple key operations as long as all the keys involved into a single command execution (or whole transaction, or Lua script execution) all belong to the same hash slot. The user can force multiple keys to be part of the same hash slot by using a concept called *hash tags*.

Hash tags are documented in the Redis Cluster specification, but the gist is that if there is a substring between {} brackets in a key, only what is inside the string is hashed, so for example `this{foo}key` and `another{foo}key` are guaranteed to be in the same hash slot, and can be used together in a command with multiple keys as arguments.

Redis Cluster master-slave model

In order to remain available when a subset of master nodes are failing or are not able to communicate with the majority of nodes, Redis Cluster uses a master-slave model where every hash slot has from 1 (the master itself) to N replicas (N-1 additional slaves nodes).

In our example cluster with nodes A, B, C, if node B fails the cluster is not able to continue, since we no longer have a way to serve hash slots in the range 5501-11000.

However when the cluster is created (or at a later time) we add a slave node to every master, so that the final cluster is composed of A, B, C that are masters nodes, and A1, B1, C1 that are slaves nodes, the system is able to continue if node B fails.

Node B1 replicates B, and B fails, the cluster will promote node B1 as the new master and will continue to operate correctly.

However note that if nodes B and B1 fail at the same time Redis Cluster is not able to continue to operate.

Redis Cluster consistency guarantees

Redis Cluster is not able to guarantee **strong consistency**. In practical terms this means that under certain conditions it is possible that Redis Cluster will lose writes that were acknowledged by the system to the client.

The first reason why Redis Cluster can lose writes is because it uses asynchronous replication. This means that during writes the following happens:

- Your client writes to the master B.
- The master B replies OK to your client.
- The master B propagates the write to its slaves B1, B2 and B3.

As you can see B does not wait for an acknowledge from B1, B2, B3 before replying to the client, since this would be a prohibitive latency penalty for Redis, so if your client writes something, B acknowledges the write, but crashes before being able to send the write to its slaves, one of the slaves (that did not receive the write) can be promoted to master, losing the write forever.

This is **very similar to what happens** with most databases that are configured to flush data to disk every second, so it is a scenario you are already able to reason about because of past experiences with traditional database systems not involving distributed systems. Similarly you can improve consistency by forcing the database to flush data on disk before replying to the client, but this usually results into prohibitively low performance. That would be the equivalent of synchronous replication in the case of Redis Cluster.

Basically there is a trade-off to take between performance and consistency.

Redis Cluster has support for synchronous writes when absolutely needed, implemented via the WAIT command, this makes losing writes a lot less likely, however note that Redis Cluster does not implement strong consistency even when synchronous replication is used: it is always possible under more complex failure scenarios that a slave that was not able to receive the write is elected as master.

There is another notable scenario where Redis Cluster will lose writes, that happens during a network partition where a client is isolated with a minority of instances including at least a master.

Take as an example our 6 nodes cluster composed of A, B, C, A1, B1, C1, with 3 masters and 3 slaves. There is also a client, that we will call Z1.

After a partition occurs, it is possible that in one side of the partition we have A, C, A1, B1, C1, and in the other side we have B and Z1.

Z1 is still able to write to B, that will accept its writes. If the partition heals in a very short time, the cluster will continue normally. However if the partition lasts enough time for B1 to be promoted to master in the majority side of the partition, the writes that Z1 is sending to B will be lost.

Note that there is a **maximum window** to the amount of writes Z1 will be able to send to B: if enough time has elapsed for the majority side of the partition to elect a slave as master, every master node in the minority side stops accepting writes.

This amount of time is a very important configuration directive of Redis Cluster, and is called the **node timeout**.

After node timeout has elapsed, a master node is considered to be failing, and can be replaced by one of its replicas. Similarly after node timeout has elapsed without a master node to be able to sense the majority of the other master nodes, it enters an error state and stops accepting writes.

Redis Cluster configuration parameters

We are about to create an example cluster deployment. Before we continue, let's introduce the configuration parameters that Redis Cluster introduces in the `redis.conf` file. Some will be obvious, others will be more clear as you continue reading.

- **cluster-enabled `<yes/no>`**: If yes enables Redis Cluster support in a specific Redis instance. Otherwise the instance starts as a stand alone instance as usual.
- **cluster-config-file `<filename>`**: Note that despite the name of this option, this is not an user editable configuration file, but the file where a Redis Cluster node automatically persists the cluster configuration (the state, basically) every time there is a change, in order to be able to re-read it at startup. The file lists things like the other nodes in the cluster, their state, persistent variables, and so forth. Often this file is rewritten and flushed on disk as a result of some message reception.
- **cluster-node-timeout `<milliseconds>`**: The maximum amount of time a Redis Cluster node can be unavailable, without it being considered as failing. If a master node is not reachable for more than the specified amount of time, it will be failed over by its slaves. This parameter controls other important things in Redis Cluster. Notably, every node that can't reach the majority of master nodes for the specified amount of time, will stop accepting queries.
- **cluster-slave-validity-factor `<factor>`**: If set to zero, a slave will always try to failover a master, regardless of the amount of time the link between the master and the slave remained disconnected. If the value is positive, a maximum disconnection time is calculated as the *node timeout* value multiplied by the factor provided with this option, and if the node is a slave, it will not try to start a failover if the master link was disconnected for more than the specified amount of time. For example if the node timeout is set to 5 seconds, and the validity factor is set to 10, a slave disconnected from the master for more than 50 seconds will not try to failover its master. Note that any value different than zero may result in Redis Cluster to be unavailable after a master failure if there is no slave able to failover it. In that case the cluster will return back available only when the original master rejoins the cluster.
- **cluster-migration-barrier `<count>`**: Minimum number of slaves a master will remain connected with, for another slave to migrate to a master which is no longer covered by any slave. See the appropriate section about replica migration in this tutorial for more information.
- **cluster-require-full-coverage `<yes/no>`**: If this is set to yes, as it is by default, the cluster stops accepting writes if some percentage of the key space is not covered by any node. If the option is set to no, the cluster will still serve queries even if only requests about a subset of keys can be processed.

Creating and using a Redis Cluster

Note: to deploy a Redis Cluster manually it is **very important to learn** certain operational aspects of it. However if you want to get a cluster up and running ASAP (As Soon As Possible) skip this section and the next one and go directly to **Creating a Redis Cluster using the create-cluster script**.

To create a cluster, the first thing we need is to have a few empty Redis instances running in **cluster mode**. This basically means that clusters are not created using normal Redis instances as a special mode needs to be configured so that the Redis instance will enable the Cluster specific features and commands.

The following is a minimal Redis cluster configuration file:

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

As you can see what enables the cluster mode is simply the `cluster-enabled` directive. Every instance also contains the path of a file where the configuration for this node is stored, which by default is `nodes.conf`. This file is never touched by humans; it is simply generated at startup by the Redis Cluster instances, and updated every time it is needed.

Note that the **minimal cluster** that works as expected requires to contain at least three master nodes. For your first tests it is strongly suggested to start a six nodes cluster with three masters and three slaves.

To do so, enter a new directory, and create the following directories named after the port number of the instance we'll run inside any given directory.

Something like:

```
mkdir cluster-test
cd cluster-test
mkdir 7000 7001 7002 7003 7004 7005
```

Create a `redis.conf` file inside each of the directories, from 7000 to 7005. As a template for your configuration file just use the small example above, but make sure to replace the port number `7000` with the right port number according to the directory name.

Now copy your redis-server executable, **compiled from the latest sources in the unstable branch at GitHub**, into the `cluster-test` directory, and finally open 6 terminal tabs in your favorite terminal application.

Start every instance like that, one every tab:

```
cd 7000
../redis-server ./redis.conf
```

As you can see from the logs of every instance, since no `nodes.conf` file existed, every node assigns itself a new ID.

```
[82462] 26 Nov 11:56:55.329 * No cluster configuration found, I'm 97a
```

This ID will be used forever by this specific instance in order for the instance to have a unique name in the context of the cluster. Every node remembers every other node using this IDs, and not by IP or port. IP addresses and ports may change, but the unique node identifier will never change for all the life of the node. We call this identifier simply **Node ID**.

Creating the cluster

Now that we have a number of instances running, we need to create our cluster by writing some meaningful configuration to the nodes.

If you are using Redis 5, this is very easy to accomplish as we are helped by the Redis Cluster command line utility embedded into `redis-cli`, that can be used to create new clusters, check or reshard an existing cluster, and so forth.

For Redis version 3 or 4, there is the older tool called `redis-trib.rb` which is very similar. You can find it in the `src` directory of the Redis source code distribution. You need to install `redis` gem to be able to run `redis-trib`.

```
gem install redis
```

The first example, that is, the cluster creation, will be shown using both `redis-cli` in Redis 5 and `redis-trib` in Redis 3 and 4. However all the next examples will only use `redis-cli`, since as you can see the syntax is very similar, and you can trivially change one command line into the other by using `redis-trib.rb help` to get info about the old syntax. **Important:** note that you can use Redis 5 `redis-cli` against Redis 4 clusters without issues if you wish.

To create your cluster for Redis 5 with `redis-cli` simply type:

```
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
```

Using `redis-trib.rb` for Redis 4 or 3 type:

```
./redis-trib.rb create --replicas 1 127.0.0.1:7000 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
```

The command used here is **create**, since we want to create a new cluster. The option `--cluster-replicas 1` means that we want a slave for every master created. The other arguments are the list of addresses of the instances I want to use to create the new cluster.

Obviously the only setup with our requirements is to create a cluster with 3 masters and 3 slaves.

Redis-cli will propose you a configuration. Accept the proposed configuration by typing **yes**. The cluster will be configured and *joined*, which means, instances will be bootstrapped into talking with each other. Finally, if everything went well, you'll see a message like that:

```
[OK] All 16384 slots covered
```

This means that there is at least a master instance serving each of the 16384 slots available.

Creating a Redis Cluster using the create-cluster script

If you don't want to create a Redis Cluster by configuring and executing individual instances manually as explained above, there is a much simpler system (but you'll not learn the same amount of operational details).

Just check `utils/create-cluster` directory in the Redis distribution. There is a script called `create-cluster` inside (same name as the directory it is contained into), it's a simple bash script. In order to start a 6 nodes cluster with 3 masters and 3 slaves just type the following commands:

1. `create-cluster start`
2. `create-cluster create`

Reply to `yes` in step 2 when the `redis-cli` utility wants you to accept the cluster layout.

You can now interact with the cluster, the first node will start at port 30001 by default. When you are done, stop the cluster with:

1. `create-cluster stop`.

Please read the `README` inside this directory for more information on how to run the script.

Playing with the cluster

At this stage one of the problems with Redis Cluster is the lack of client libraries implementations.

I'm aware of the following implementations:

- redis-rb-cluster is a Ruby implementation written by me (@antirez) as a reference for other languages. It is a simple wrapper around the original redis-rb, implementing the minimal semantics to talk with the cluster efficiently.
- redis-py-cluster A port of redis-rb-cluster to Python. Supports majority of *redis-py* functionality. Is in active development.
- The popular Predis has support for Redis Cluster, the support was recently updated and is in active development.
- The most used Java client, Jedis recently added support for Redis Cluster, see the *Jedis Cluster* section in the project README.
- StackExchange.Redis offers support for C# (and should work fine with most .NET languages; VB, F#, etc)
- thunk-redis offers support for Node.js and io.js, it is a thunk/promise-based redis client with pipelining and cluster.
- redis-go-cluster is an implementation of Redis Cluster for the Go language using the Redigo library client as the base client. Implements MGET/MSET via result aggregation.
- The `redis-cli` utility in the unstable branch of the Redis repository at GitHub implements a very basic cluster support when started with the `-c` switch.

An easy way to test Redis Cluster is either to try any of the above clients or simply the `redis-cli` command line utility. The following is an example of interaction using the latter:

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

Note: if you created the cluster using the script your nodes may listen to different ports, starting from 30001 by default.

The redis-cli cluster support is very basic so it always uses the fact that Redis Cluster nodes are able to redirect a client to the right node. A serious client is able to do better than that, and cache the map between hash slots and nodes addresses, to directly use the right connection to the right node. The map is refreshed only when something changed in the cluster configuration, for example after a failover or after the system administrator changed the cluster layout by adding or removing nodes.

Writing an example app with redis-rb-cluster

Before going forward showing how to operate the Redis Cluster, doing things like a failover, or a resharding, we need to create some example application or at least to be able to understand the semantics of a simple Redis Cluster client interaction.

In this way we can run an example and at the same time try to make nodes failing, or start a resharding, to see how Redis Cluster behaves under real world conditions. It is not very helpful to see what happens while nobody is writing to the cluster.

This section explains some basic usage of redis-rb-cluster showing two examples. The first is the following, and is the `example.rb` file inside the redis-rb-cluster distribution:

```
require './cluster'

if ARGV.length != 2
    startup_nodes = [
        {:host => "127.0.0.1", :port => 7000},
        {:host => "127.0.0.1", :port => 7001}
    ]
else
    startup_nodes = [
        {:host => ARGV[0], :port => ARGV[1].to_i}
    ]
end

rc = RedisCluster.new(startup_nodes,32,:timeout => 0.1)

last = false

while not last
```

```
19      begin
20          last = rc.get("__last__")
21          last = 0 if !last
22      rescue => e
23          puts "error #{e.to_s}"
24          sleep 1
25      end
26  end
27
28  ((last.to_i+1)..1000000000).each{|x|
29      begin
30          rc.set("foo#{x}",x)
31          puts rc.get("foo#{x}")
32          rc.set("__last__",x)
33      rescue => e
34          puts "error #{e.to_s}"
35      end
36      sleep 0.1
37  }
```

The application does a very simple thing, it sets keys in the form `foo<number>` to `number`, one after the other. So if you run the program the result is the following stream of commands:

- SET foo0 0
- SET foo1 1
- SET foo2 2
- And so forth...

The program looks more complex than it should usually as it is designed to show errors on the screen instead of exiting with an exception, so every operation performed with the cluster is wrapped by `begin` `rescue` blocks.

The **line 14** is the first interesting line in the program. It creates the Redis Cluster object, using as argument a list of *startup nodes*, the maximum number of connections this object is allowed to take against different nodes, and finally the timeout after a given operation is considered to be failed.

The startup nodes don't need to be all the nodes of the cluster. The important thing is that at least one node is reachable. Also note that redis-rb-cluster updates this list of startup nodes as soon as it is able to connect with the first node. You should expect such a behavior with any other serious client.

Now that we have the Redis Cluster object instance stored in the **rc** variable we are ready to use the object like if it was a normal Redis object instance.

This is exactly what happens in **line 18 to 26**: when we restart the example we don't want to start again with `foo0`, so we store the counter inside Redis itself. The code above is designed to read this counter, or if the counter does not exist, to assign it the value of zero.

However note how it is a while loop, as we want to try again and again even if the cluster is down and is returning errors. Normal applications don't need to be so careful.

Lines between 28 and 37 start the main loop where the keys are set or an error is displayed.

Note the `sleep` call at the end of the loop. In your tests you can remove the sleep if you want to write to the cluster as fast as possible (relatively to the fact that this is a busy loop without real parallelism of course, so you'll get the usually 10k ops/second in the best of the conditions).

Normally writes are slowed down in order for the example application to be easier to follow by humans.

Starting the application produces the following output:

```
ruby ./example.rb
1
2
3
4
5
6
7
8
9
^C (I stopped the program here)
```

This is not a very interesting program and we'll use a better one in a moment but we can already see what happens during a resharding when the program is running.

Resharding the cluster

Now we are ready to try a cluster resharding. To do this please keep the example.rb program running, so that you can see if there is some impact on the program running. Also you may want to comment the `sleep` call in order to have some more serious write load during resharding.

Resharding basically means to move hash slots from a set of nodes to another set of nodes, and like cluster creation it is accomplished using the redis-cli utility.

To start a resharding just type:

```
redis-cli --cluster reshard 127.0.0.1:7000
```

You only need to specify a single node, redis-cli will find the other nodes automatically.

Currently redis-cli is only able to reshard with the administrator support, you can't just say move 5% of slots from this node to the other one (but this is pretty trivial to implement). So it starts with questions. The first is how much a big resharding do you want to do:

```
How many slots do you want to move (from 1 to 16384)?
```

We can try to reshard 1000 hash slots, that should already contain a non trivial amount of keys if the example is still running without the sleep call.

Then redis-cli needs to know what is the target of the resharding, that is, the node that will receive the hash slots. I'll use the first master node, that is, 127.0.0.1:7000, but I need to specify the Node ID of the instance. This was already printed in a list by redis-cli, but I can always find the ID of a node with the following command if I need:

```
$ redis-cli -p 7000 cluster nodes | grep myself
97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 con
```

Ok so my target node is 97a3a64667477371c4479320d683e4c8db5858b1.

Now you'll get asked from what nodes you want to take those keys. I'll just type `all` in order to take a bit of hash slots from all the other master nodes.

After the final confirmation you'll see a message for every slot that redis-cli is going to move from a node to another, and a dot will be printed for every actual key moved from one side to the other.

While the resharding is in progress you should be able to see your example program running unaffected. You can stop and restart it multiple times during the resharding if you want.

At the end of the resharding, you can test the health of the cluster with the following command:

```
redis-cli --cluster check 127.0.0.1:7000
```

All the slots will be covered as usual, but this time the master at 127.0.0.1:7000 will have more hash slots, something around 6461.

Scripting a resharding operation

Reshardings can be performed automatically without the need to manually enter the parameters in an interactive way. This is possible using a command line like the following:

```
redis-cli reshard <host>:<port> --cluster-from <node-id> --cluster-to
```

This allows to build some automatism if you are likely to reshard often, however currently there is no way for `redis-cli` to automatically rebalance the cluster checking the distribution of keys across the cluster nodes and intelligently moving slots as needed. This feature will be added in the future.

A more interesting example application

The example application we wrote early is not very good. It writes to the cluster in a simple way without even checking if what was written is the right thing.

From our point of view the cluster receiving the writes could just always write the key `foo` to 42 to every operation, and we would not notice at all.

So in the `redis-rb-cluster` repository, there is a more interesting application that is called `consistency-test.rb`. It uses a set of counters, by default 1000, and sends INCR commands in order to increment the counters.

However instead of just writing, the application does two additional things:

- When a counter is updated using INCR, the application remembers the write.
- It also reads a random counter before every write, and check if the value is what we expected it to be, comparing it with the value it has in memory.

What this means is that this application is a simple **consistency checker**, and is able to tell you if the cluster lost some write, or if it accepted a write that we did not receive acknowledgment for. In the first case we'll see a counter having a value that is smaller than the one we remember, while in the second case the value will be greater.

Running the consistency-test application produces a line of output every second:

```
$ ruby consistency-test.rb
925 R (0 err) | 925 W (0 err) |
5030 R (0 err) | 5030 W (0 err) |
9261 R (0 err) | 9261 W (0 err) |
13517 R (0 err) | 13517 W (0 err) |
17780 R (0 err) | 17780 W (0 err) |
22025 R (0 err) | 22025 W (0 err) |
25818 R (0 err) | 25818 W (0 err) |
```

The line shows the number of **R**eads and **W**rites performed, and the number of errors (query not accepted because of errors since the system was not available).

If some inconsistency is found, new lines are added to the output. This is what happens, for example, if I reset a counter manually while the program is running:

```
$ redis-cli -h 127.0.0.1 -p 7000 set key_217 0
OK

(in the other tab I see...)

94774 R (0 err) | 94774 W (0 err) |
98821 R (0 err) | 98821 W (0 err) |
102886 R (0 err) | 102886 W (0 err) | 114 lost |
107046 R (0 err) | 107046 W (0 err) | 114 lost |
```

When I set the counter to 0 the real value was 114, so the program reports 114 lost writes (INCR commands that are not remembered by the cluster).

This program is much more interesting as a test case, so we'll use it to test the Redis Cluster failover.

Testing the failover

Note: during this test, you should take a tab open with the consistency test application running.

In order to trigger the failover, the simplest thing we can do (that is also the semantically simplest failure that can occur in a distributed system) is to crash a single process, in our case a single master.

We can identify a cluster and crash it with the following command:

```
$ redis-cli -p 7000 cluster nodes | grep master
3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 13
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 master - 0 13
97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 con
```

Ok, so 7000, 7001, and 7002 are masters. Let's crash node 7002 with the **DEBUG SEGFAULT** command:

```
$ redis-cli -p 7002 debug segfault
Error: Server closed the connection
```

Now we can look at the output of the consistency test to see what it reported.

```
18849 R (0 err) | 18849 W (0 err) |
23151 R (0 err) | 23151 W (0 err) |
27302 R (0 err) | 27302 W (0 err) |

... many error warnings here ...

29659 R (578 err) | 29660 W (577 err) |
33749 R (578 err) | 33750 W (577 err) |
37918 R (578 err) | 37919 W (577 err) |
42077 R (578 err) | 42078 W (577 err) |
```

As you can see during the failover the system was not able to accept 578 reads and 577 writes, however no inconsistency was created in the database. This may sound unexpected as in the first part of this tutorial we stated that Redis Cluster can lose writes during the failover because it uses asynchronous replication. What we did not say is that this is not very likely to happen because Redis sends the reply to the client, and the commands to replicate to the slaves, about at the same time, so there is a very small window to lose data. However the fact that it is hard to trigger does not mean that it is impossible, so this does not change the consistency guarantees provided by Redis cluster.

We can now check what is the cluster setup after the failover (note that in the meantime I restarted the crashed instance so that it rejoins the cluster as a slave):

```
$ redis-cli -p 7000 cluster nodes
3fc783611028b1707fd65345e763befb36454d73 127.0.0.1:7004 slave 3e3a6cb
a211e242fc6b22a9427fed61285e85892fa04e08 127.0.0.1:7003 slave 97a3a64
97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 con
3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7005 master - 0 13
3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 13
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c7
```

Now the masters are running on ports 7000, 7001 and 7005. What was previously a master, that is the Redis instance running on port 7002, is now a slave of 7005.

The output of the CLUSTER NODES command may look intimidating, but it is actually pretty simple, and is composed of the following tokens:

- Node ID
- ip:port
- flags: master, slave, myself, fail, ...
- if it is a slave, the Node ID of the master
- Time of the last pending PING still waiting for a reply.
- Time of the last PONG received.
- Configuration epoch for this node (see the Cluster specification).
- Status of the link to this node.
- Slots served...

Manual failover

Sometimes it is useful to force a failover without actually causing any problem on a master. For example in order to upgrade the Redis process of one of the master nodes it is a good idea to failover it in order to turn it into a slave with minimal impact on availability.

Manual failovers are supported by Redis Cluster using the CLUSTER FAILOVER command, that must be executed in one of the **slaves** of the master you want to failover.

Manual failovers are special and are safer compared to failovers resulting from actual master failures, since they occur in a way that avoid data loss in the process, by switching clients from the original master to the new master only when the system is sure that the new master processed all the replication stream from the old one.

This is what you see in the slave log when you perform a manual failover:

```
# Manual failover user request accepted.
# Received replication offset for paused master manual failover: 3475
# All master replication stream processed, manual failover can start.
# Start of election delayed for 0 milliseconds (rank #0, offset 34754
# Starting a failover election for epoch 7545.
# Failover election won: I'm the new master.
```

Basically clients connected to the master we are failing over are stopped. At the same time the master sends its replication offset to the slave, that waits to reach the offset on its side. When the replication offset is reached, the failover starts, and the old master is informed about the configuration switch. When the clients are unblocked on the old master, they are redirected to the new master.

Adding a new node

Adding a new node is basically the process of adding an empty node and then moving some data into it, in case it is a new master, or telling it to setup as a replica of a known node, in case it is a slave.

We'll show both, starting with the addition of a new master instance.

In both cases the first step to perform is **adding an empty node**.

This is as simple as to start a new node in port 7006 (we already used from 7000 to 7005 for our existing 6 nodes) with the same configuration used for the other nodes, except for the port number, so what you should do in order to conform with the setup we used for the previous nodes:

- Create a new tab in your terminal application.
- Enter the `cluster-test` directory.
- Create a directory named `7006`.
- Create a redis.conf file inside, similar to the one used for the other nodes but using 7006 as port number.
- Finally start the server with `../redis-server ./redis.conf`

At this point the server should be running.

Now we can use **redis-cli** as usual in order to add the node to the existing cluster.

```
redis-cli --cluster add-node 127.0.0.1:7006 127.0.0.1:7000
```

As you can see I used the **add-node** command specifying the address of the new node as first argument, and the address of a random existing node in the cluster as second argument.

In practical terms redis-cli here did very little to help us, it just sent a CLUSTER MEET message to the node, something that is also possible to accomplish manually. However redis-cli also checks the state of the cluster before to operate, so it is a good idea to perform cluster operations always via redis-cli even when you know how the internals work.

Now we can connect to the new node to see if it really joined the cluster:

```
redis 127.0.0.1:7006> cluster nodes
3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 13
3fc783611028b1707fd65345e763befb36454d73 127.0.0.1:7004 slave 3e3a6cb
f093c80dde814da99c5cf72a7dd01590792b783b :0 myself,master - 0 0 0 con
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c7
a211e242fc6b22a9427fed61285e85892fa04e08 127.0.0.1:7003 slave 97a3a64
97a3a64667477371c4479320d683e4c8db5858b1 127.0.0.1:7000 master - 0 13
3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7005 master - 0 13
```

Note that since this node is already connected to the cluster it is already able to redirect client queries correctly and is generally speaking part of the cluster. However it has two peculiarities compared to the other masters:

- It holds no data as it has no assigned hash slots.
- Because it is a master without assigned slots, it does not participate in the election process when a slave wants to become a master.

Now it is possible to assign hash slots to this node using the resharding feature of `redis-cli`. It is basically useless to show this as we already did in a previous section, there is no difference, it is just a resharding having as a target the empty node.

Adding a new node as a replica

Adding a new Replica can be performed in two ways. The obvious one is to use redis-cli again, but with the --cluster-slave option, like this:

```
redis-cli --cluster add-node 127.0.0.1:7006 127.0.0.1:7000 --cluster-
```

Note that the command line here is exactly like the one we used to add a new master, so we are not specifying to which master we want to add the replica. In this case what happens is that redis-cli will add the new node as replica of a random master among the masters with less replicas.

However you can specify exactly what master you want to target with your new replica with the following command line:

```
redis-cli --cluster add-node 127.0.0.1:7006 127.0.0.1:7000 --cluster-
```

This way we assign the new replica to a specific master.

A more manual way to add a replica to a specific master is to add the new node as an empty master, and then turn it into a replica using the CLUSTER REPLICATE command. This also works if the node was added as a slave but you want to move it as a replica of a different master.

For example in order to add a replica for the node 127.0.0.1:7005 that is currently serving hash slots in the range 11423-16383, that has a Node ID 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e, all I need to do is to connect with the new node (already added as empty master) and send the command:

```
redis 127.0.0.1:7006> cluster replicate 3c3a0c74aae0b56170ccb03a76b60
```

That's it. Now we have a new replica for this set of hash slots, and all the other nodes in the cluster already know (after a few seconds needed to update their config). We can verify with the following command:

```
$ redis-cli -p 7000 cluster nodes | grep slave | grep 3c3a0c74aae0b56
f093c80dde814da99c5cf72a7dd01590792b783b 127.0.0.1:7006 slave 3c3a0c7
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c7
```

The node 3c3a0c... now has two slaves, running on ports 7002 (the existing one) and 7006 (the new one).

Removing a node

To remove a slave node just use the `del-node` command of redis-cli:

```
redis-cli --cluster del-node 127.0.0.1:7000 `<node-id>`
```

The first argument is just a random node in the cluster, the second argument is the ID of the node you want to remove.

You can remove a master node in the same way as well, **however in order to remove a master node it must be empty**. If the master is not empty you need to reshard data away from it to all the other master nodes before.

An alternative to remove a master node is to perform a manual failover of it over one of its slaves and remove the node after it turned into a slave of the new master. Obviously this does not help when you want to reduce the actual number of masters in your cluster, in that case, a resharding is needed.

Replicas migration

In Redis Cluster it is possible to reconfigure a slave to replicate with a different master at any time just using the following command:

```
CLUSTER REPLICATE <master-node-id>
```

However there is a special scenario where you want replicas to move from one master to another one automatically, without the help of the system administrator. The automatic reconfiguration of replicas is called *replicas migration* and is able to improve the reliability of a Redis Cluster.

Note: you can read the details of replicas migration in the Redis Cluster Specification, here we'll only provide some information about the general idea and what you should do in order to benefit from it.

The reason why you may want to let your cluster replicas to move from one master to another under certain condition, is that usually the Redis Cluster is as resistant to failures as the number of replicas attached to a given master.

For example a cluster where every master has a single replica can't continue operations if the master and its replica fail at the same time, simply because there is no other instance to have a copy of the hash slots the master was serving. However while netsplits are likely to isolate a number of nodes at the same time, many other kind of failures, like hardware or software failures local to a single node, are a very notable class of failures that are unlikely to happen at the same time, so it is possible that in your cluster where every master has a slave, the slave is killed at 4am, and the master is killed at 6am. This still will result in a cluster that can no longer operate.

To improve reliability of the system we have the option to add additional replicas to every master, but this is expensive. Replica migration allows to add more slaves to just a few masters. So you have 10 masters with 1 slave each, for a total of 20 instances. However you add, for example, 3 instances more as slaves of some of your masters, so certain masters will have more than a single slave.

With replicas migration what happens is that if a master is left without slaves, a replica from a master that has multiple slaves will migrate to the *orphaned* master. So after your slave goes down at 4am as in the example we made above, another slave will take its place, and when the master will fail as well at 5am, there is still a slave that can be elected so that the cluster can continue to operate.

So what you should know about replicas migration in short?

- The cluster will try to migrate a replica from the master that has the greatest number of replicas in a given moment.
- To benefit from replica migration you have just to add a few more replicas to a single master in your cluster, it does not matter what master.
- There is a configuration parameter that controls the replica migration feature that is called `cluster-migration-barrier`: you can read more about it in the example `redis.conf` file provided with Redis Cluster.

Upgrading nodes in a Redis Cluster

Upgrading slave nodes is easy since you just need to stop the node and restart it with an updated version of Redis. If there are clients scaling reads using slave nodes, they should be able to reconnect to a different slave if a given one is not available.

Upgrading masters is a bit more complex, and the suggested procedure is:

1. Use CLUSTER FAILOVER to trigger a manual failover of the master to one of its slaves (see the "Manual failover" section of this documentation).
2. Wait for the master to turn into a slave.
3. Finally upgrade the node as you do for slaves.
4. If you want the master to be the node you just upgraded, trigger a new manual failover in order to turn back the upgraded node into a master.

Following this procedure you should upgrade one node after the other until all the nodes are upgraded.

Migrating to Redis Cluster

Users willing to migrate to Redis Cluster may have just a single master, or may already using a preexisting sharding setup, where keys are split among N nodes, using some in-house algorithm or a sharding algorithm implemented by their client library or Redis proxy.

In both cases it is possible to migrate to Redis Cluster easily, however what is the most important detail is if multiple-keys operations are used by the application, and how. There are three different cases:

1. Multiple keys operations, or transactions, or Lua scripts involving multiple keys, are not used. Keys are accessed independently (even if accessed via transactions or Lua scripts grouping multiple commands, about the same key, together).

2. Multiple keys operations, transactions, or Lua scripts involving multiple keys are used but only with keys having the same **hash tag**, which means that the keys used together all have a `{...}` sub-string that happens to be identical. For example the following multiple keys operation is defined in the context of the same hash tag: `SUNION {user:1000}.foo {user:1000}.bar`.
3. Multiple keys operations, transactions, or Lua scripts involving multiple keys are used with key names not having an explicit, or the same, hash tag.

The third case is not handled by Redis Cluster: the application requires to be modified in order to don't use multi keys operations or only use them in the context of the same hash tag.

Case 1 and 2 are covered, so we'll focus on those two cases, that are handled in the same way, so no distinction will be made in the documentation.

Assuming you have your preexisting data set split into N masters, where N=1 if you have no preexisting sharding, the following steps are needed in order to migrate your data set to Redis Cluster:

1. Stop your clients. No automatic live-migration to Redis Cluster is currently possible. You may be able to do it orchestrating a live migration in the context of your application / environment.
2. Generate an append only file for all of your N masters using the BGREWRITEAOF command, and waiting for the AOF file to be completely generated.
3. Save your AOF files from aof-1 to aof-N somewhere. At this point you can stop your old instances if you wish (this is useful since in non-virtualized deployments you often need to reuse the same computers).
4. Create a Redis Cluster composed of N masters and zero slaves. You'll add slaves later. Make sure all your nodes are using the append only file for persistence.
5. Stop all the cluster nodes, substitute their append only file with your pre-existing append only files, aof-1 for the first node, aof-2 for the second node, up to aof-N.
6. Restart your Redis Cluster nodes with the new AOF files. They'll complain that there are keys that should not be there according to their configuration.
7. Use `redis-cli --cluster fix` command in order to fix the cluster so that keys will be migrated according to the hash slots each node is authoritative or not.
8. Use `redis-cli --cluster check` at the end to make sure your cluster is ok.
9. Restart your clients modified to use a Redis Cluster aware client library.

There is an alternative way to import data from external instances to a Redis Cluster, which is to use the `redis-cli --cluster import` command.

The command moves all the keys of a running instance (deleting the keys from the source instance) to the specified pre-existing Redis Cluster. However note that if you use a Redis 2.8 instance as source instance the operation may be slow since 2.8 does not implement migrate connection caching, so you may want to restart your source instance with a Redis 3.x version before to perform such operation.