



Redis Cluster Specification

Welcome to the **Redis Cluster Specification**. Here you'll find information about algorithms and design rationales of Redis Cluster. This document is a work in progress as it is continuously synchronized with the actual implementation of Redis.

Main properties and rationales of the design

Redis Cluster goals

Redis Cluster is a distributed implementation of Redis with the following goals, in order of importance in the design:

- High performance and linear scalability up to 1000 nodes. There are no proxies, asynchronous replication is used, and no merge operations are performed on values.
- Acceptable degree of write safety: the system tries (in a best-effort way) to retain all the writes originating from clients connected with the majority of the master nodes. Usually there are small windows where acknowledged writes can be lost. Windows to lose acknowledged writes are larger when clients are in a minority partition.
- Availability: Redis Cluster is able to survive partitions where the majority of the master nodes are reachable and there is at least one reachable slave for every master node that is no longer reachable. Moreover using *replicas migration*, masters no longer replicated by any slave will receive one from a master which is covered by multiple slaves.

What is described in this document is implemented in Redis 3.0 or greater.

Implemented subset

Redis Cluster implements all the single key commands available in the non-distributed version of Redis. Commands performing complex multi-key operations like Set type unions or intersections are implemented as well as long as the keys all hash to the same slot.

Redis Cluster implements a concept called **hash tags** that can be used in order to force certain keys to be stored in the same hash slot. However during manual resharding, multi-key operations may become unavailable for some time while single key operations are always available.

Redis Cluster does not support multiple databases like the stand alone version of Redis. There is just database 0 and the **SELECT** command is not allowed.

Clients and Servers roles in the Redis Cluster protocol

In Redis Cluster nodes are responsible for holding the data, and taking the state of the cluster, including mapping keys to the right nodes. Cluster nodes are also able to auto-discover other nodes, detect non-working nodes, and promote slave nodes to master when needed in order to continue to operate when a failure occurs.

To perform their tasks all the cluster nodes are connected using a TCP bus and a binary protocol, called the **Redis Cluster Bus**. Every node is connected to every other node in the cluster using the cluster bus. Nodes use a gossip protocol to propagate information about the cluster in order to discover new nodes, to send ping packets to make sure all the other nodes are working properly, and to send cluster messages needed to signal specific conditions. The cluster bus is also used in order to propagate Pub/Sub messages across the cluster and to orchestrate manual failovers when requested by users (manual failovers are failovers which are not initiated by the Redis Cluster failure detector, but by the system administrator directly).

Since cluster nodes are not able to proxy requests, clients may be redirected to other nodes using redirection errors `–MOVED` and `–ASK`. The client is in theory free to send requests to all the nodes in the cluster, getting redirected if needed, so the client is not required to hold the state of the cluster. However clients that are able to cache the map between keys and nodes can improve the performance in a sensible way.

Write safety

Redis Cluster uses asynchronous replication between nodes, and **last failover wins** implicit merge function. This means that the last elected master dataset eventually replaces all the other replicas. There is always a window of time when it is possible to lose writes during partitions. However these windows are very different in the case of a client that is connected to the majority of masters, and a client that is connected to the minority of masters.

Redis Cluster tries harder to retain writes that are performed by clients connected to the majority of masters, compared to writes performed in the minority side. The following are examples of scenarios that lead to loss of acknowledged writes received in the majority partitions during failures:

1. A write may reach a master, but while the master may be able to reply to the client, the write may not be propagated to slaves via the asynchronous replication used between master and slave nodes. If the master dies without the write reaching the slaves, the write is lost forever if the master is unreachable for a long enough period that one of its slaves is promoted. This is usually hard to observe in the case of a total, sudden failure of a master node since masters try to reply to clients (with the acknowledge of the write) and slaves (propagating the write) at about the same time. However it is a real world failure mode.
2. Another theoretically possible failure mode where writes are lost is the following:

- A master is unreachable because of a partition.
- It gets failed over by one of its slaves.
- After some time it may be reachable again.
- A client with an out-of-date routing table may write to the old master before it is converted into a slave (of the new master) by the cluster.

The second failure mode is unlikely to happen because master nodes unable to communicate with the majority of the other masters for enough time to be failed over will no longer accept writes, and when the partition is fixed writes are still refused for a small amount of time to allow other nodes to inform about configuration changes. This failure mode also requires that the client's routing table has not yet been updated.

Writes targeting the minority side of a partition have a larger window in which to get lost. For example, Redis Cluster loses a non-trivial number of writes on partitions where there is a minority of masters and at least one or more clients, since all the writes sent to the masters may potentially get lost if the masters are failed over in the majority side.

Specifically, for a master to be failed over it must be unreachable by the majority of masters for at least `NODE_TIMEOUT`, so if the partition is fixed before that time, no writes are lost. When the partition lasts for more than `NODE_TIMEOUT`, all the writes performed in the minority side up to that point may be lost. However the minority side of a Redis Cluster will start refusing writes as soon as `NODE_TIMEOUT` time has elapsed without contact with the majority, so there is a maximum window after which the minority becomes no longer available. Hence, no writes are accepted or lost after that time.

Availability

Redis Cluster is not available in the minority side of the partition. In the majority side of the partition assuming that there are at least the majority of masters and a slave for every unreachable master, the cluster becomes available again after `NODE_TIMEOUT` time plus a few more seconds required for a slave to get elected and failover its master (failovers are usually executed in a matter of 1 or 2 seconds).

This means that Redis Cluster is designed to survive failures of a few nodes in the cluster, but it is not a suitable solution for applications that require availability in the event of large net splits.

In the example of a cluster composed of N master nodes where every node has a single slave, the majority side of the cluster will remain available as long as a single node is partitioned away, and will remain available with a probability of $1 - (1 / (N * 2 - 1))$ when two nodes are partitioned away (after the first node fails we are left with $N * 2 - 1$ nodes in total, and the probability of the only master without a replica to fail is $1 / (N * 2 - 1)$).

For example, in a cluster with 5 nodes and a single slave per node, there is a $1 / (5 * 2 - 1) = 11.11\%$ probability that after two nodes are partitioned away from the majority, the cluster will no longer be available.

Thanks to a Redis Cluster feature called **replicas migration** the Cluster availability is improved in many real world scenarios by the fact that replicas migrate to orphaned masters (masters no longer having replicas). So at every successful failure event, the cluster may reconfigure the slaves layout in order to better resist the next failure.

Performance

In Redis Cluster nodes don't proxy commands to the right node in charge for a given key, but instead they redirect clients to the right nodes serving a given portion of the key space. Eventually clients obtain an up-to-date representation of the cluster and which node serves which subset of keys, so during normal operations clients directly contact the right nodes in order to send a given command.

Because of the use of asynchronous replication, nodes do not wait for other nodes' acknowledgment of writes (if not explicitly requested using the [WAIT](#) command).

Also, because multi-key commands are only limited to *near* keys, data is never moved between nodes except when resharding.

Normal operations are handled exactly as in the case of a single Redis instance. This means that in a Redis Cluster with N master nodes you can expect the same performance as a single Redis instance multiplied by N as the design scales linearly. At the same time the query is usually performed in a single round trip, since clients usually retain persistent connections with the nodes, so latency figures are also the same as the single standalone Redis node case.

Very high performance and scalability while preserving weak but reasonable forms of data safety and availability is the main goal of Redis Cluster.

Why merge operations are avoided

Redis Cluster design avoids conflicting versions of the same key-value pair in multiple nodes as in the case of the Redis data model this is not always desirable. Values in Redis are often very large; it is common to see lists or sorted sets with millions of elements. Also data types are semantically complex. Transferring and merging these kind of values can be a major bottleneck and/or may require the non-trivial involvement of application-side logic, additional memory to store meta-data, and so forth.

There are no strict technological limits here. CRDTs or synchronously replicated state machines can model complex data types similar to Redis. However, the actual run time behavior of such systems would not be similar to Redis Cluster. Redis Cluster was designed in order to cover the exact use cases of the non-clustered Redis version.

Overview of Redis Cluster main components

Keys distribution model

The key space is split into 16384 slots, effectively setting an upper limit for the cluster size of 16384 master nodes (however the suggested max size of nodes is in the order of ~ 1000 nodes).

Each master node in a cluster handles a subset of the 16384 hash slots. The cluster is **stable** when there is no cluster reconfiguration in progress (i.e. where hash slots are being moved from one node to another). When the cluster is stable, a single hash slot will be served by a single node (however the serving node can have one or more slaves that will replace it in the case of net splits or failures, and that can be used in order to scale read operations where reading stale data is acceptable).

The base algorithm used to map keys to hash slots is the following (read the next paragraph for the hash tag exception to this rule):

$$\text{HASH_SLOT} = \text{CRC16}(\text{key}) \bmod 16384$$

The CRC16 is specified as follows:

- Name: XMODEM (also known as ZMODEM or CRC-16/ACORN)
- Width: 16 bit
- Poly: 1021 (That is actually $x^{16} + x^{12} + x^5 + 1$)
- Initialization: 0000
- Reflect Input byte: False
- Reflect Output CRC: False
- Xor constant to output CRC: 0000
- Output for "123456789": 31C3

14 out of 16 CRC16 output bits are used (this is why there is a modulo 16384 operation in the formula above).

In our tests CRC16 behaved remarkably well in distributing different kinds of keys evenly across the 16384 slots.

Note: A reference implementation of the CRC16 algorithm used is available in the Appendix A of this document.

Keys hash tags

There is an exception for the computation of the hash slot that is used in order to implement **hash tags**. Hash tags are a way to ensure that multiple keys are allocated in the same hash slot. This is used in order to implement multi-key operations in Redis Cluster.

In order to implement hash tags, the hash slot for a key is computed in a slightly different way in certain conditions. If the key contains a "{...}" pattern only the substring between { and } is hashed in order to obtain the hash slot. However since it is possible that there are multiple occurrences of { or } the algorithm is well specified by the following rules:

- IF the key contains a { character.
- AND IF there is a } character to the right of {
- AND IF there are one or more characters between the first occurrence of { and the first occurrence of }.

Then instead of hashing the key, only what is between the first occurrence of { and the following first occurrence of } is hashed.

Examples:

- The two keys {user1000}.following and {user1000}.followers will hash to the same hash slot since only the substring user1000 will be hashed in order to compute the hash slot.
- For the key foo{}{bar} the whole key will be hashed as usually since the first occurrence of { is followed by } on the right without characters in the middle.
- For the key foo{{bar}}zap the substring {bar will be hashed, because it is the substring between the first occurrence of { and the first occurrence of } on its right.
- For the key foo{bar}{zap} the substring bar will be hashed, since the algorithm stops at the first valid or invalid (without bytes inside) match of { and }.
- What follows from the algorithm is that if the key starts with {}, it is guaranteed to be hashed as a whole. This is useful when using binary data as key names.

Adding the hash tags exception, the following is an implementation of the HASH_SLOT function in Ruby and C language.

Ruby example code:

```
def HASH_SLOT(key)
  s = key.index "{"
  if s
    e = key.index "}", s+1
    if e && e != s+1
      key = key[s+1..e-1]
    end
  end
  crc16(key) % 16384
end
```

C example code:

```

unsigned int HASH_SLOT(char *key, int keylen) {
    int s, e; /* start-end indexes of { and } */

    /* Search the first occurrence of '{'. */
    for (s = 0; s < keylen; s++)
        if (key[s] == '{') break;

    /* No '{' ? Hash the whole key. This is the base case. */
    if (s == keylen) return crc16(key,keylen) & 16383;

    /* '{' found? Check if we have the corresponding '}'. */
    for (e = s+1; e < keylen; e++)
        if (key[e] == '}') break;

    /* No '}' or nothing between {} ? Hash the whole key. */
    if (e == keylen || e == s+1) return crc16(key,keylen) & 16383;

    /* If we are here there is both a { and a } on its right. Hash
     * what is in the middle between { and }. */
    return crc16(key+s+1,e-s-1) & 16383;
}

```

Cluster nodes attributes

Every node has a unique name in the cluster. The node name is the hex representation of a 160 bit random number, obtained the first time a node is started (usually using `/dev/urandom`). The node will save its ID in the node configuration file, and will use the same ID forever, or at least as long as the node configuration file is not deleted by the system administrator, or a *hard reset* is requested via the [CLUSTER RESET](#) command.

The node ID is used to identify every node across the whole cluster. It is possible for a given node to change its IP address without any need to also change the node ID. The cluster is also able to detect the change in IP/port and reconfigure using the gossip protocol running over the cluster bus.

The node ID is not the only information associated with each node, but is the only one that is always globally consistent. Every node has also the following set of information associated. Some information is about the cluster configuration detail of this specific node, and is eventually consistent across the cluster. Some other information, like the last time a node was pinged, is instead local to each node.

Every node maintains the following information about other nodes that it is aware of in the cluster: The node ID, IP and port of the node, a set of flags, what is the master of the node if it is flagged as `slave`, last time the node was pinged and the last time the pong was received, the current *configuration epoch* of the node (explained later in this specification), the link state and finally the set of hash slots served.

A detailed [explanation of all the node fields](#) is described in the [CLUSTER NODES](#) documentation.

The [CLUSTER NODES](#) command can be sent to any node in the cluster and provides the state of the cluster and the information for each node according to the local view the queried node has of the cluster.

The following is sample output of the [CLUSTER NODES](#) command sent to a master node in a small cluster of three nodes.

```
$ redis-cli cluster nodes
d1861060fe6a534d42d8a19aeb36600e18785e04 127.0.0.1:6379 myself - 0 13
3886e65cc906bfd9b1f7e7bde468726a052d1dae 127.0.0.1:6380 master - 1318
d289c575dcbcb4bdd2931585fd4339089e461a27d 127.0.0.1:6381 master - 1318
```

In the above listing the different fields are in order: node id, address:port, flags, last ping sent, last pong received, configuration epoch, link state, slots. Details about the above fields will be covered as soon as we talk of specific parts of Redis Cluster.

The Cluster bus

Every Redis Cluster node has an additional TCP port for receiving incoming connections from other Redis Cluster nodes. This port is at a fixed offset from the normal TCP port used to receive incoming connections from clients. To obtain the Redis Cluster port, 10000 should be added to the normal commands port. For example, if a Redis node is listening for client connections on port 6379, the Cluster bus port 16379 will also be opened.

Node-to-node communication happens exclusively using the Cluster bus and the Cluster bus protocol: a binary protocol composed of frames of different types and sizes. The Cluster bus binary protocol is not publicly documented since it is not intended for external software devices to talk with Redis Cluster nodes using this protocol. However you can obtain more details about the Cluster bus protocol by reading the `cluster.h` and `cluster.c` files in the Redis Cluster source code.

Cluster topology

Redis Cluster is a full mesh where every node is connected with every other node using a TCP connection.

In a cluster of N nodes, every node has N-1 outgoing TCP connections, and N-1 incoming connections.

These TCP connections are kept alive all the time and are not created on demand. When a node expects a pong reply in response to a ping in the cluster bus, before waiting long enough to mark the node as unreachable, it will try to refresh the connection with the node by reconnecting from scratch.

While Redis Cluster nodes form a full mesh, **nodes use a gossip protocol and a configuration update mechanism in order to avoid exchanging too many messages between nodes during normal conditions**, so the number of messages exchanged is not exponential.

Nodes handshake

Nodes always accept connections on the cluster bus port, and even reply to pings when received, even if the pinging node is not trusted. However, all other packets will be discarded by the receiving node if the sending node is not considered part of the cluster.

A node will accept another node as part of the cluster only in two ways:

- If a node presents itself with a MEET message. A meet message is exactly like a [PING](#) message, but forces the receiver to accept the node as part of the cluster. Nodes will send MEET messages to other nodes **only if** the system administrator requests this via the following command:

```
CLUSTER MEET ip port
```

- A node will also register another node as part of the cluster if a node that is already trusted will gossip about this other node. So if A knows B, and B knows C, eventually B will send gossip messages to A about C. When this happens, A will register C as part of the network, and will try to connect with C.

This means that as long as we join nodes in any connected graph, they'll eventually form a fully connected graph automatically. This means that the cluster is able to auto-discover other nodes, but only if there is a trusted relationship that was forced by the system administrator.

This mechanism makes the cluster more robust but prevents different Redis clusters from accidentally mixing after change of IP addresses or other network related events.

Redirection and resharding

MOVED Redirection

A Redis client is free to send queries to every node in the cluster, including slave nodes. The node will analyze the query, and if it is acceptable (that is, only a single key is mentioned in the query, or the multiple keys mentioned are all to the same hash slot) it will lookup what node is responsible for the hash slot where the key or keys belong.

If the hash slot is served by the node, the query is simply processed, otherwise the node will check its internal hash slot to node map, and will reply to the client with a MOVED error, like in the following example:

```
GET x
-MOVED 3999 127.0.0.1:6381
```

The error includes the hash slot of the key (3999) and the ip:port of the instance that can serve the query. The client needs to reissue the query to the specified node's IP address and port. Note that even if the client waits a long time before reissuing the query, and in the meantime the cluster configuration changed, the destination node will reply again with a MOVED error if the hash slot 3999 is now served by another node. The same happens if the contacted node had no updated information.

So while from the point of view of the cluster nodes are identified by IDs we try to simplify our interface with the client just exposing a map between hash slots and Redis nodes identified by IP:port pairs.

The client is not required to, but should try to memorize that hash slot 3999 is served by 127.0.0.1:6381. This way once a new command needs to be issued it can compute the hash slot of the target key and have a greater chance of choosing the right node.

An alternative is to just refresh the whole client-side cluster layout using the [CLUSTER NODES](#) or [CLUSTER SLOTS](#) commands when a MOVED redirection is received. When a redirection is encountered, it is likely multiple slots were reconfigured rather than just one, so updating the client configuration as soon as possible is often the best strategy.

Note that when the Cluster is stable (no ongoing changes in the configuration), eventually all the clients will obtain a map of hash slots -> nodes, making the cluster efficient, with clients directly addressing the right nodes without redirections, proxies or other single point of failure entities.

A client **must be also able to handle -ASK redirections** that are described later in this document, otherwise it is not a complete Redis Cluster client.

Cluster live reconfiguration

Redis Cluster supports the ability to add and remove nodes while the cluster is running. Adding or removing a node is abstracted into the same operation: moving a hash slot from one node to another. This means that the same basic mechanism can be used in order to rebalance the cluster, add or remove nodes, and so forth.

- To add a new node to the cluster an empty node is added to the cluster and some set of hash slots are moved from existing nodes to the new node.

- To remove a node from the cluster the hash slots assigned to that node are moved to other existing nodes.
- To rebalance the cluster a given set of hash slots are moved between nodes.

The core of the implementation is the ability to move hash slots around. From a practical point of view a hash slot is just a set of keys, so what Redis Cluster really does during *resharding* is to move keys from an instance to another instance. Moving a hash slot means moving all the keys that happen to hash into this hash slot.

To understand how this works we need to show the `CLUSTER` subcommands that are used to manipulate the slots translation table in a Redis Cluster node.

The following subcommands are available (among others not useful in this case):

- `CLUSTER ADDSLOTS` slot1 [slot2] ... [slotN]
- `CLUSTER DELSLOTS` slot1 [slot2] ... [slotN]
- `CLUSTER SETSLOT` slot NODE node
- `CLUSTER SETSLOT` slot MIGRATING node
- `CLUSTER SETSLOT` slot IMPORTING node

The first two commands, `ADDSLOTS` and `DELSLOTS`, are simply used to assign (or remove) slots to a Redis node. Assigning a slot means to tell a given master node that it will be in charge of storing and serving content for the specified hash slot.

After the hash slots are assigned they will propagate across the cluster using the gossip protocol, as specified later in the *configuration propagation* section.

The `ADDSLOTS` command is usually used when a new cluster is created from scratch to assign each master node a subset of all the 16384 hash slots available.

The `DELSLOTS` is mainly used for manual modification of a cluster configuration or for debugging tasks: in practice it is rarely used.

The `SETSLOT` subcommand is used to assign a slot to a specific node ID if the `SETSLOT <slot> NODE` form is used. Otherwise the slot can be set in the two special states `MIGRATING` and `IMPORTING`. Those two special states are used in order to migrate a hash slot from one node to another.

- When a slot is set as `MIGRATING`, the node will accept all queries that are about this hash slot, but only if the key in question exists, otherwise the query is forwarded using a `-ASK` redirection to the node that is target of the migration.
- When a slot is set as `IMPORTING`, the node will accept all queries that are about this hash slot, but only if the request is preceded by an `ASKING` command. If the `ASKING` command was not given by the client, the query is redirected to the real hash slot owner via a `-MOVED` redirection error, as would happen normally.

Let's make this clearer with an example of hash slot migration. Assume that we have two Redis master nodes, called A and B. We want to move hash slot 8 from A to B, so we issue commands like this:

- We send B: `CLUSTER SETSLOT 8 IMPORTING A`

- We send A: `CLUSTER SETSLOT 8 MIGRATING B`

All the other nodes will continue to point clients to node "A" every time they are queried with a key that belongs to hash slot 8, so what happens is that:

- All queries about existing keys are processed by "A".
- All queries about non-existing keys in A are processed by "B", because "A" will redirect clients to "B".

This way we no longer create new keys in "A". In the meantime, a special program called `redis-trib` used during reshardings and Redis Cluster configuration will migrate existing keys in hash slot 8 from A to B. This is performed using the following command:

```
CLUSTER GETKEYSINSLOT slot count
```

The above command will return `count` keys in the specified hash slot. For every key returned, `redis-trib` sends node "A" a `MIGRATE` command, that will migrate the specified key from A to B in an atomic way (both instances are locked for the time (usually very small time) needed to migrate a key so there are no race conditions). This is how `MIGRATE` works:

```
MIGRATE target_host target_port key target_database id timeout
```

`MIGRATE` will connect to the target instance, send a serialized version of the key, and once an OK code is received, the old key from its own dataset will be deleted. From the point of view of an external client a key exists either in A or B at any given time.

In Redis Cluster there is no need to specify a database other than 0, but `MIGRATE` is a general command that can be used for other tasks not involving Redis Cluster. `MIGRATE` is optimized to be as fast as possible even when moving complex keys such as long lists, but in Redis Cluster reconfiguring the cluster where big keys are present is not considered a wise procedure if there are latency constraints in the application using the database.

When the migration process is finally finished, the `SETSLOT <slot> NODE <node-id>` command is sent to the two nodes involved in the migration in order to set the slots to their normal state again. The same command is usually sent to all other nodes to avoid waiting for the natural propagation of the new configuration across the cluster.

ASK redirection

In the previous section we briefly talked about ASK redirection. Why can't we simply use `MOVED` redirection? Because while `MOVED` means that we think the hash slot is

permanently served by a different node and the next queries should be tried against the specified node, ASK means to send only the next query to the specified node.

This is needed because the next query about hash slot 8 can be about a key that is still in A, so we always want the client to try A and then B if needed. Since this happens only for one hash slot out of 16384 available, the performance hit on the cluster is acceptable.

We need to force that client behavior, so to make sure that clients will only try node B after A was tried, node B will only accept queries of a slot that is set as IMPORTING if the client sends the ASKING command before sending the query.

Basically the ASKING command sets a one-time flag on the client that forces a node to serve a query about an IMPORTING slot.

The full semantics of ASK redirection from the point of view of the client is as follows:

- If ASK redirection is received, send only the query that was redirected to the specified node but continue sending subsequent queries to the old node.
- Start the redirected query with the ASKING command.
- Don't yet update local client tables to map hash slot 8 to B.

Once hash slot 8 migration is completed, A will send a MOVED message and the client may permanently map hash slot 8 to the new IP and port pair. Note that if a buggy client performs the map earlier this is not a problem since it will not send the ASKING command before issuing the query, so B will redirect the client to A using a MOVED redirection error.

Slots migration is explained in similar terms but with different wording (for the sake of redundancy in the documentation) in the [CLUSTER SETSLOT](#) command documentation.

Clients first connection and handling of redirections

While it is possible to have a Redis Cluster client implementation that does not remember the slots configuration (the map between slot numbers and addresses of nodes serving it) in memory and only works by contacting random nodes waiting to be redirected, such a client would be very inefficient.

Redis Cluster clients should try to be smart enough to memorize the slots configuration. However this configuration is not *required* to be up to date. Since contacting the wrong node will simply result in a redirection, that should trigger an update of the client view.

Clients usually need to fetch a complete list of slots and mapped node addresses in two different situations:

- At startup in order to populate the initial slots configuration.
- When a MOVED redirection is received.

Note that a client may handle the MOVED redirection by updating just the moved slot in its table, however this is usually not efficient since often the configuration of multiple slots is modified at once (for example if a slave is promoted to master, all the slots served by the old master will be remapped). It is much simpler to react to a MOVED redirection by fetching the full map of slots to nodes from scratch.

In order to retrieve the slots configuration Redis Cluster offers an alternative to the **CLUSTER NODES** command that does not require parsing, and only provides the information strictly needed to clients.

The new command is called **CLUSTER SLOTS** and provides an array of slots ranges, and the associated master and slave nodes serving the specified range.

The following is an example of output of **CLUSTER SLOTS**:

```
127.0.0.1:7000> cluster slots
1) 1) (integer) 5461
   2) (integer) 10922
   3) 1) "127.0.0.1"
      2) (integer) 7001
   4) 1) "127.0.0.1"
      2) (integer) 7004
2) 1) (integer) 0
   2) (integer) 5460
   3) 1) "127.0.0.1"
      2) (integer) 7000
   4) 1) "127.0.0.1"
      2) (integer) 7003
3) 1) (integer) 10923
   2) (integer) 16383
   3) 1) "127.0.0.1"
      2) (integer) 7002
   4) 1) "127.0.0.1"
      2) (integer) 7005
```

The first two sub-elements of every element of the returned array are the start-end slots of the range. The additional elements represent address-port pairs. The first address-port pair is the master serving the slot, and the additional address-port pairs are all the slaves serving the same slot that are not in an error condition (i.e. the FAIL flag is not set).

For example the first element of the output says that slots from 5461 to 10922 (start and end included) are served by 127.0.0.1:7001, and it is possible to scale read-only load contacting the slave at 127.0.0.1:7004.

CLUSTER SLOTS is not guaranteed to return ranges that cover the full 16384 slots if the cluster is misconfigured, so clients should initialize the slots configuration map filling the target nodes with NULL objects, and report an error if the user tries to execute commands about keys that belong to unassigned slots.

Before returning an error to the caller when a slot is found to be unassigned, the client should try to fetch the slots configuration again to check if the cluster is now configured properly.

Multiple keys operations

Using hash tags, clients are free to use multi-key operations. For example the following operation is valid:

```
MSET {user:1000}.name Angela {user:1000}.surname White
```

Multi-key operations may become unavailable when a resharding of the hash slot the keys belong to is in progress.

More specifically, even during a resharding the multi-key operations targeting keys that all exist and all still hash to the same slot (either the source or destination node) are still available.

Operations on keys that don't exist or are - during the resharding - split between the source and destination nodes, will generate a `-TRYAGAIN` error. The client can try the operation after some time, or report back the error.

As soon as migration of the specified hash slot has terminated, all multi-key operations are available again for that hash slot.

Scaling reads using slave nodes

Normally slave nodes will redirect clients to the authoritative master for the hash slot involved in a given command, however clients can use slaves in order to scale reads using the `READONLY` command.

`READONLY` tells a Redis Cluster slave node that the client is ok reading possibly stale data and is not interested in running write queries.

When the connection is in readonly mode, the cluster will send a redirection to the client only if the operation involves keys not served by the slave's master node. This may happen because:

1. The client sent a command about hash slots never served by the master of this slave.
2. The cluster was reconfigured (for example resharded) and the slave is no longer able to serve commands for a given hash slot.

When this happens the client should update its hashslot map as explained in the previous sections.

The readonly state of the connection can be cleared using the `READWRITE` command.

Fault Tolerance

Heartbeat and gossip messages

Redis Cluster nodes continuously exchange ping and pong packets. Those two kind of packets have the same structure, and both carry important configuration information. The only actual difference is the message type field. We'll refer to the sum of ping and pong packets as *heartbeat packets*.

Usually nodes send ping packets that will trigger the receivers to reply with pong packets. However this is not necessarily true. It is possible for nodes to just send pong packets to send information to other nodes about their configuration, without triggering a reply. This is useful, for example, in order to broadcast a new configuration as soon as possible.

Usually a node will ping a few random nodes every second so that the total number of ping packets sent (and pong packets received) by each node is a constant amount regardless of the number of nodes in the cluster.

However every node makes sure to ping every other node that hasn't sent a ping or received a pong for longer than half the `NODE_TIMEOUT` time. Before `NODE_TIMEOUT` has elapsed, nodes also try to reconnect the TCP link with another node to make sure nodes are not believed to be unreachable only because there is a problem in the current TCP connection.

The number of messages globally exchanged can be sizable if `NODE_TIMEOUT` is set to a small figure and the number of nodes (N) is very large, since every node will try to ping every other node for which they don't have fresh information every half the `NODE_TIMEOUT` time.

For example in a 100 node cluster with a node timeout set to 60 seconds, every node will try to send 99 pings every 30 seconds, with a total amount of pings of 3.3 per second. Multiplied by 100 nodes, this is 330 pings per second in the total cluster.

There are ways to lower the number of messages, however there have been no reported issues with the bandwidth currently used by Redis Cluster failure detection, so for now the obvious and direct design is used. Note that even in the above example, the 330 packets per second exchanged are evenly divided among 100 different nodes, so the traffic each node receives is acceptable.

Heartbeat packet content

Ping and pong packets contain a header that is common to all types of packets (for instance packets to request a failover vote), and a special Gossip Section that is specific of Ping and Pong packets.

The common header has the following information:

- Node ID, a 160 bit pseudorandom string that is assigned the first time a node is created and remains the same for all the life of a Redis Cluster node.

- The `currentEpoch` and `configEpoch` fields of the sending node that are used to mount the distributed algorithms used by Redis Cluster (this is explained in detail in the next sections). If the node is a slave the `configEpoch` is the last known `configEpoch` of its master.
- The node flags, indicating if the node is a slave, a master, and other single-bit node information.
- A bitmap of the hash slots served by the sending node, or if the node is a slave, a bitmap of the slots served by its master.
- The sender TCP base port (that is, the port used by Redis to accept client commands; add 10000 to this to obtain the cluster bus port).
- The state of the cluster from the point of view of the sender (down or ok).
- The master node ID of the sending node, if it is a slave.

Ping and pong packets also contain a gossip section. This section offers to the receiver a view of what the sender node thinks about other nodes in the cluster. The gossip section only contains information about a few random nodes among the set of nodes known to the sender. The number of nodes mentioned in a gossip section is proportional to the cluster size.

For every node added in the gossip section the following fields are reported:

- Node ID.
- IP and port of the node.
- Node flags.

Gossip sections allow receiving nodes to get information about the state of other nodes from the point of view of the sender. This is useful both for failure detection and to discover other nodes in the cluster.

Failure detection

Redis Cluster failure detection is used to recognize when a master or slave node is no longer reachable by the majority of nodes and then respond by promoting a slave to the role of master. When slave promotion is not possible the cluster is put in an error state to stop receiving queries from clients.

As already mentioned, every node takes a list of flags associated with other known nodes. There are two flags that are used for failure detection that are called **PFAIL** and **FAIL**. **PFAIL** means *Possible failure*, and is a non-acknowledged failure type. **FAIL** means that a node is failing and that this condition was confirmed by a majority of masters within a fixed amount of time.

PFAIL flag:

A node flags another node with the **PFAIL** flag when the node is not reachable for more than `NODE_TIMEOUT` time. Both master and slave nodes can flag another node as **PFAIL**, regardless of its type.

The concept of non-reachability for a Redis Cluster node is that we have an **active ping** (a ping that we sent for which we have yet to get a reply) pending for longer than `NODE_TIMEOUT`. For this mechanism to work the `NODE_TIMEOUT` must be large compared to the network round trip time. In order to add reliability during normal operations, nodes will try to reconnect with other nodes in the cluster as soon as half of the `NODE_TIMEOUT` has elapsed without a reply to a ping. This mechanism ensures that connections are kept alive so broken connections usually won't result in false failure reports between nodes.

FAIL flag:

The `PFAIL` flag alone is just local information every node has about other nodes, but it is not sufficient to trigger a slave promotion. For a node to be considered down the `PFAIL` condition needs to be escalated to a `FAIL` condition.

As outlined in the node heartbeats section of this document, every node sends gossip messages to every other node including the state of a few random known nodes. Every node eventually receives a set of node flags for every other node. This way every node has a mechanism to signal other nodes about failure conditions they have detected.

A `PFAIL` condition is escalated to a `FAIL` condition when the following set of conditions are met:

- Some node, that we'll call A, has another node B flagged as `PFAIL`.
- Node A collected, via gossip sections, information about the state of B from the point of view of the majority of masters in the cluster.
- The majority of masters signaled the `PFAIL` or `FAIL` condition within `NODE_TIMEOUT * FAIL_REPORT_VALIDITY_MULT` time. (The validity factor is set to 2 in the current implementation, so this is just two times the `NODE_TIMEOUT` time).

If all the above conditions are true, Node A will:

- Mark the node as `FAIL`.
- Send a `FAIL` message to all the reachable nodes.

The `FAIL` message will force every receiving node to mark the node in `FAIL` state, whether or not it already flagged the node in `PFAIL` state.

Note that *the FAIL flag is mostly one way*. That is, a node can go from `PFAIL` to `FAIL`, but a `FAIL` flag can only be cleared in the following situations:

- The node is already reachable and is a slave. In this case the `FAIL` flag can be cleared as slaves are not failed over.
- The node is already reachable and is a master not serving any slot. In this case the `FAIL` flag can be cleared as masters without slots do not really participate in the cluster and are waiting to be configured in order to join the cluster.
- The node is already reachable and is a master, but a long time (N times the `NODE_TIMEOUT`) has elapsed without any detectable slave promotion. It's better for it to rejoin the cluster and continue in this case.

It is useful to note that while the PFAIL -> FAIL transition uses a form of agreement, the agreement used is weak:

1. Nodes collect views of other nodes over some time period, so even if the majority of master nodes need to "agree", actually this is just state that we collected from different nodes at different times and we are not sure, nor we require, that at a given moment the majority of masters agreed. However we discard failure reports which are old, so the failure was signaled by the majority of masters within a window of time.
2. While every node detecting the FAIL condition will force that condition on other nodes in the cluster using the FAIL message, there is no way to ensure the message will reach all the nodes. For instance a node may detect the FAIL condition and because of a partition will not be able to reach any other node.

However the Redis Cluster failure detection has a liveness requirement: eventually all the nodes should agree about the state of a given node. There are two cases that can originate from split brain conditions. Either some minority of nodes believe the node is in FAIL state, or a minority of nodes believe the node is not in FAIL state. In both the cases eventually the cluster will have a single view of the state of a given node:

Case 1: If a majority of masters have flagged a node as FAIL, because of failure detection and the *chain effect* it generates, every other node will eventually flag the master as FAIL, since in the specified window of time enough failures will be reported.

Case 2: When only a minority of masters have flagged a node as FAIL, the slave promotion will not happen (as it uses a more formal algorithm that makes sure everybody knows about the promotion eventually) and every node will clear the FAIL state as per the FAIL state clearing rules above (i.e. no promotion after N times the NODE_TIMEOUT has elapsed).

The FAIL flag is only used as a trigger to run the safe part of the algorithm for the slave promotion. In theory a slave may act independently and start a slave promotion when its master is not reachable, and wait for the masters to refuse to provide the acknowledgment if the master is actually reachable by the majority. However the added complexity of the PFAIL -> FAIL state, the weak agreement, and the FAIL message forcing the propagation of the state in the shortest amount of time in the reachable part of the cluster, have practical advantages. Because of these mechanisms, usually all the nodes will stop accepting writes at about the same time if the cluster is in an error state. This is a desirable feature from the point of view of applications using Redis Cluster. Also erroneous election attempts initiated by slaves that can't reach its master due to local problems (the master is otherwise reachable by the majority of other master nodes) are avoided.

Configuration handling, propagation, and failovers

Cluster current epoch

Redis Cluster uses a concept similar to the Raft algorithm "term". In Redis Cluster the term is called epoch instead, and it is used in order to give incremental versioning to events. When multiple nodes provide conflicting information, it becomes possible for another node to understand which state is the most up to date.

The `currentEpoch` is a 64 bit unsigned number.

At node creation every Redis Cluster node, both slaves and master nodes, set the `currentEpoch` to 0.

Every time a packet is received from another node, if the epoch of the sender (part of the cluster bus messages header) is greater than the local node epoch, the `currentEpoch` is updated to the sender epoch.

Because of these semantics, eventually all the nodes will agree to the greatest `currentEpoch` in the cluster.

This information is used when the state of the cluster is changed and a node seeks agreement in order to perform some action.

Currently this happens only during slave promotion, as described in the next section.

Basically the epoch is a logical clock for the cluster and dictates that given information wins over one with a smaller epoch.

Configuration epoch

Every master always advertises its `configEpoch` in ping and pong packets along with a bitmap advertising the set of slots it serves.

The `configEpoch` is set to zero in masters when a new node is created.

A new `configEpoch` is created during slave election. Slaves trying to replace failing masters increment their epoch and try to get authorization from a majority of masters. When a slave is authorized, a new unique `configEpoch` is created and the slave turns into a master using the new `configEpoch`.

As explained in the next sections the `configEpoch` helps to resolve conflicts when different nodes claim divergent configurations (a condition that may happen because of network partitions and node failures).

Slave nodes also advertise the `configEpoch` field in ping and pong packets, but in the case of slaves the field represents the `configEpoch` of its master as of the last time they exchanged packets. This allows other instances to detect when a slave has an old configuration that needs to be updated (master nodes will not grant votes to slaves with an old configuration).

Every time the `configEpoch` changes for some known node, it is permanently stored in the `nodes.conf` file by all the nodes that receive this information. The same also happens for the `currentEpoch` value. These two variables are guaranteed to be saved and `fsync`-ed to disk when updated before a node continues its operations.

The `configEpoch` values generated using a simple algorithm during failovers are guaranteed to be new, incremental, and unique.

Slave election and promotion

Slave election and promotion is handled by slave nodes, with the help of master nodes that vote for the slave to promote. A slave election happens when a master is in `FAIL` state from the point of view of at least one of its slaves that has the prerequisites in order to become a master.

In order for a slave to promote itself to master, it needs to start an election and win it. All the slaves for a given master can start an election if the master is in `FAIL` state, however only one slave will win the election and promote itself to master.

A slave starts an election when the following conditions are met:

- The slave's master is in `FAIL` state.
- The master was serving a non-zero number of slots.
- The slave replication link was disconnected from the master for no longer than a given amount of time, in order to ensure the promoted slave's data is reasonably fresh. This time is user configurable.

In order to be elected, the first step for a slave is to increment its `currentEpoch` counter, and request votes from master instances.

Votes are requested by the slave by broadcasting a `FAILOVER_AUTH_REQUEST` packet to every master node of the cluster. Then it waits for a maximum time of two times the `NODE_TIMEOUT` for replies to arrive (but always for at least 2 seconds).

Once a master has voted for a given slave, replying positively with a `FAILOVER_AUTH_ACK`, it can no longer vote for another slave of the same master for a period of `NODE_TIMEOUT * 2`. In this period it will not be able to reply to other authorization requests for the same master. This is not needed to guarantee safety, but useful for preventing multiple slaves from getting elected (even if with a different `configEpoch`) at around the same time, which is usually not wanted.

A slave discards any `AUTH_ACK` replies with an epoch that is less than the `currentEpoch` at the time the vote request was sent. This ensures it doesn't count votes intended for a previous election.

Once the slave receives ACKs from the majority of masters, it wins the election. Otherwise if the majority is not reached within the period of two times `NODE_TIMEOUT` (but always at least 2 seconds), the election is aborted and a new one will be tried again after `NODE_TIMEOUT * 4` (and always at least 4 seconds).

Slave rank

As soon as a master is in `FAIL` state, a slave waits a short period of time before trying to get elected. That delay is computed as follows:

```
DELAY = 500 milliseconds + random delay between 0 and 500 millisecond  
SLAVE_RANK * 1000 milliseconds.
```

The fixed delay ensures that we wait for the `FAIL` state to propagate across the cluster, otherwise the slave may try to get elected while the masters are still unaware of the `FAIL` state, refusing to grant their vote.

The random delay is used to desynchronize slaves so they're unlikely to start an election at the same time.

The `SLAVE_RANK` is the rank of this slave regarding the amount of replication data it has processed from the master. Slaves exchange messages when the master is failing in order to establish a (best effort) rank: the slave with the most updated replication offset is at rank 0, the second most updated at rank 1, and so forth. In this way the most updated slaves try to get elected before others.

Rank order is not strictly enforced; if a slave of higher rank fails to be elected, the others will try shortly.

Once a slave wins the election, it obtains a new unique and incremental `configEpoch` which is higher than that of any other existing master. It starts advertising itself as master in ping and pong packets, providing the set of served slots with a `configEpoch` that will win over the past ones.

In order to speedup the reconfiguration of other nodes, a pong packet is broadcast to all the nodes of the cluster. Currently unreachable nodes will eventually be reconfigured when they receive a ping or pong packet from another node or will receive an `UPDATE` packet from another node if the information it publishes via heartbeat packets are detected to be out of date.

The other nodes will detect that there is a new master serving the same slots served by the old master but with a greater `configEpoch`, and will upgrade their configuration. Slaves of the old master (or the failed over master if it rejoins the cluster) will not just upgrade the configuration but will also reconfigure to replicate from the new master. How nodes rejoining the cluster are configured is explained in the next sections.

Masters reply to slave vote request

In the previous section it was discussed how slaves try to get elected. This section explains what happens from the point of view of a master that is requested to vote for a given slave.

Masters receive requests for votes in form of `FAILOVER_AUTH_REQUEST` requests from slaves.

For a vote to be granted the following conditions need to be met:

1. A master only votes a single time for a given epoch, and refuses to vote for older epochs: every master has a `lastVoteEpoch` field and will refuse to vote again as long as the `currentEpoch` in the auth request packet is not greater than the `lastVoteEpoch`. When a master replies positively to a vote request, the `lastVoteEpoch` is updated accordingly, and safely stored on disk.
2. A master votes for a slave only if the slave's master is flagged as `FAIL`.
3. Auth requests with a `currentEpoch` that is less than the master `currentEpoch` are ignored. Because of this the master reply will always have the same `currentEpoch` as the auth request. If the same slave asks again to be voted, incrementing the `currentEpoch`, it is guaranteed that an old delayed reply from the master can not be accepted for the new vote.

Example of the issue caused by not using rule number 3:

Master `currentEpoch` is 5, `lastVoteEpoch` is 1 (this may happen after a few failed elections)

- Slave `currentEpoch` is 3.
 - Slave tries to be elected with epoch 4 (3+1), master replies with an ok with `currentEpoch` 5, however the reply is delayed.
 - Slave will try to be elected again, at a later time, with epoch 5 (4+1), the delayed reply reaches the slave with `currentEpoch` 5, and is accepted as valid.
1. Masters don't vote for a slave of the same master before `NODE_TIMEOUT * 2` has elapsed if a slave of that master was already voted for. This is not strictly required as it is not possible for two slaves to win the election in the same epoch. However, in practical terms it ensures that when a slave is elected it has plenty of time to inform the other slaves and avoid the possibility that another slave will win a new election, performing an unnecessary second failover.
 2. Masters make no effort to select the best slave in any way. If the slave's master is in `FAIL` state and the master did not vote in the current term, a positive vote is granted. The best slave is the most likely to start an election and win it before the other slaves, since it will usually be able to start the voting process earlier because of its *higher rank* as explained in the previous section.
 3. When a master refuses to vote for a given slave there is no negative response, the request is simply ignored.
 4. Masters don't vote for slaves sending a `configEpoch` that is less than any `configEpoch` in the master table for the slots claimed by the slave. Remember that the slave sends the `configEpoch` of its master, and the bitmap of the slots served by its master. This means that the slave requesting the vote must have a configuration

for the slots it wants to failover that is newer or equal the one of the master granting the vote.

Practical example of configuration epoch usefulness during partitions

This section illustrates how the epoch concept is used to make the slave promotion process more resistant to partitions.

- A master is no longer reachable indefinitely. The master has three slaves A, B, C.
- Slave A wins the election and is promoted to master.
- A network partition makes A not available for the majority of the cluster.
- Slave B wins the election and is promoted as master.
- A partition makes B not available for the majority of the cluster.
- The previous partition is fixed, and A is available again.

At this point B is down and A is available again with a role of master (actually UPDATE messages would reconfigure it promptly, but here we assume all UPDATE messages were lost). At the same time, slave C will try to get elected in order to fail over B. This is what happens:

1. C will try to get elected and will succeed, since for the majority of masters its master is actually down. It will obtain a new incremental `configEpoch`.
2. A will not be able to claim to be the master for its hash slots, because the other nodes already have the same hash slots associated with a higher configuration epoch (the one of B) compared to the one published by A.
3. So, all the nodes will upgrade their table to assign the hash slots to C, and the cluster will continue its operations.

As you'll see in the next sections, a stale node rejoining a cluster will usually get notified as soon as possible about the configuration change because as soon as it pings any other node, the receiver will detect it has stale information and will send an UPDATE message.

Hash slots configuration propagation

An important part of Redis Cluster is the mechanism used to propagate the information about which cluster node is serving a given set of hash slots. This is vital to both the startup of a fresh cluster and the ability to upgrade the configuration after a slave was promoted to serve the slots of its failing master.

The same mechanism allows nodes partitioned away for an indefinite amount of time to rejoin the cluster in a sensible way.

There are two ways hash slot configurations are propagated:

1. Heartbeat messages. The sender of a ping or pong packet always adds information about the set of hash slots it (or its master, if it is a slave) serves.

2. **UPDATE** messages. Since in every heartbeat packet there is information about the sender `configEpoch` and set of hash slots served, if a receiver of a heartbeat packet finds the sender information is stale, it will send a packet with new information, forcing the stale node to update its info.

The receiver of a heartbeat or **UPDATE** message uses certain simple rules in order to update its table mapping hash slots to nodes. When a new Redis Cluster node is created, its local hash slot table is simply initialized to **NULL** entries so that each hash slot is not bound or linked to any node. This looks similar to the following:

```
0 -> NULL
1 -> NULL
2 -> NULL
...
16383 -> NULL
```

The first rule followed by a node in order to update its hash slot table is the following:

Rule 1: If a hash slot is unassigned (set to **NULL**), and a known node claims it, I'll modify my hash slot table and associate the claimed hash slots to it.

So if we receive a heartbeat from node A claiming to serve hash slots 1 and 2 with a configuration epoch value of 3, the table will be modified to:

```
0 -> NULL
1 -> A [3]
2 -> A [3]
...
16383 -> NULL
```

When a new cluster is created, a system administrator needs to manually assign (using the **CLUSTER ADDSLOTS** command, via the `redis-trib` command line tool, or by any other means) the slots served by each master node only to the node itself, and the information will rapidly propagate across the cluster.

However this rule is not enough. We know that hash slot mapping can change during two events:

1. A slave replaces its master during a failover.
2. A slot is resharded from a node to a different one.

For now let's focus on failovers. When a slave fails over its master, it obtains a configuration epoch which is guaranteed to be greater than the one of its master (and more generally greater than any other configuration epoch generated previously). For example node B, which is a slave of A, may failover B with configuration epoch of 4. It will start to send heartbeat packets (the first time mass-broadcasting cluster-wide) and because of the following second rule, receivers will update their hash slot tables:

Rule 2: If a hash slot is already assigned, and a known node is advertising it using a `configEpoch` that is greater than the `configEpoch` of the master currently associated with the slot, I'll rebind the hash slot to the new node.

So after receiving messages from B that claim to serve hash slots 1 and 2 with configuration epoch of 4, the receivers will update their table in the following way:

```
0 -> NULL
1 -> B [4]
2 -> B [4]
...
16383 -> NULL
```

Liveness property: because of the second rule, eventually all nodes in the cluster will agree that the owner of a slot is the one with the greatest `configEpoch` among the nodes advertising it.

This mechanism in Redis Cluster is called **last failover wins**.

The same happens during resharding. When a node importing a hash slot completes the import operation, its configuration epoch is incremented to make sure the change will be propagated throughout the cluster.

UPDATE messages, a closer look

With the previous section in mind, it is easier to see how update messages work. Node A may rejoin the cluster after some time. It will send heartbeat packets where it claims it serves hash slots 1 and 2 with configuration epoch of 3. All the receivers with updated information will instead see that the same hash slots are associated with node B having a higher configuration epoch. Because of this they'll send an **UPDATE** message to A with the new configuration for the slots. A will update its configuration because of the **rule 2** above.

How nodes rejoin the cluster

The same basic mechanism is used when a node rejoins a cluster. Continuing with the example above, node A will be notified that hash slots 1 and 2 are now served by B. Assuming that these two were the only hash slots served by A, the count of hash slots served by A will drop to 0! So A will **reconfigure to be a slave of the new master**.

The actual rule followed is a bit more complex than this. In general it may happen that A rejoins after a lot of time, in the meantime it may happen that hash slots originally served by A are served by multiple nodes, for example hash slot 1 may be served by B, and hash slot 2 by C.

So the actual *Redis Cluster node role switch rule* is: **A master node will change its configuration to replicate (be a slave of) the node that stole its last hash slot.**

During reconfiguration, eventually the number of served hash slots will drop to zero, and the node will reconfigure accordingly. Note that in the base case this just means that the old master will be a slave of the slave that replaced it after a failover. However in the general form the rule covers all possible cases.

Slaves do exactly the same: they reconfigure to replicate the node that stole the last hash slot of its former master.

Replica migration

Redis Cluster implements a concept called *replica migration* in order to improve the availability of the system. The idea is that in a cluster with a master-slave setup, if the map between slaves and masters is fixed availability is limited over time if multiple independent failures of single nodes happen.

For example in a cluster where every master has a single slave, the cluster can continue operations as long as either the master or the slave fail, but not if both fail the same time. However there is a class of failures that are the independent failures of single nodes caused by hardware or software issues that can accumulate over time. For example:

- Master A has a single slave A1.
- Master A fails. A1 is promoted as new master.
- Three hours later A1 fails in an independent manner (unrelated to the failure of A). No other slave is available for promotion since node A is still down. The cluster cannot continue normal operations.

If the map between masters and slaves is fixed, the only way to make the cluster more resistant to the above scenario is to add slaves to every master, however this is costly as it requires more instances of Redis to be executed, more memory, and so forth.

An alternative is to create an asymmetry in the cluster, and let the cluster layout automatically change over time. For example the cluster may have three masters A, B, C. A and B have a single slave each, A1 and B1. However the master C is different and has two slaves: C1 and C2.

Replica migration is the process of automatic reconfiguration of a slave in order to *migrate* to a master that has no longer coverage (no working slaves). With replica migration the scenario mentioned above turns into the following:

- Master A fails. A1 is promoted.
- C2 migrates as slave of A1, that is otherwise not backed by any slave.

- Three hours later A1 fails as well.
- C2 is promoted as new master to replace A1.
- The cluster can continue the operations.

Replica migration algorithm

The migration algorithm does not use any form of agreement since the slave layout in a Redis Cluster is not part of the cluster configuration that needs to be consistent and/or versioned with config epochs. Instead it uses an algorithm to avoid mass-migration of slaves when a master is not backed. The algorithm guarantees that eventually (once the cluster configuration is stable) every master will be backed by at least one slave.

This is how the algorithm works. To start we need to define what is a *good slave* in this context: a good slave is a slave not in FAIL state from the point of view of a given node.

The execution of the algorithm is triggered in every slave that detects that there is at least a single master without good slaves. However among all the slaves detecting this condition, only a subset should act. This subset is actually often a single slave unless different slaves have in a given moment a slightly different view of the failure state of other nodes.

The *acting slave* is the slave among the masters with the maximum number of attached slaves, that is not in FAIL state and has the smallest node ID.

So for example if there are 10 masters with 1 slave each, and 2 masters with 5 slaves each, the slave that will try to migrate is - among the 2 masters having 5 slaves - the one with the lowest node ID. Given that no agreement is used, it is possible that when the cluster configuration is not stable, a race condition occurs where multiple slaves believe themselves to be the non-failing slave with the lower node ID (it is unlikely for this to happen in practice). If this happens, the result is multiple slaves migrating to the same master, which is harmless. If the race happens in a way that will leave the ceding master without slaves, as soon as the cluster is stable again the algorithm will be re-executed again and will migrate a slave back to the original master.

Eventually every master will be backed by at least one slave. However, the normal behavior is that a single slave migrates from a master with multiple slaves to an orphaned master.

The algorithm is controlled by a user-configurable parameter called `cluster-migration-barrier`: the number of good slaves a master must be left with before a slave can migrate away. For example, if this parameter is set to 2, a slave can try to migrate only if its master remains with two working slaves.

configEpoch conflicts resolution algorithm

When new `configEpoch` values are created via slave promotion during failovers, they are guaranteed to be unique.

However there are two distinct events where new `configEpoch` values are created in an unsafe way, just incrementing the local `currentEpoch` of the local node and hoping there

are no conflicts at the same time. Both the events are system-administrator triggered:

1. **CLUSTER FAILOVER** command with **TAKEOVER** option is able to manually promote a slave node into a master *without the majority of masters being available*. This is useful, for example, in multi data center setups.
2. Migration of slots for cluster rebalancing also generates new configuration epochs inside the local node without agreement for performance reasons.

Specifically, during manual resharding, when a hash slot is migrated from a node A to a node B, the resharding program will force B to upgrade its configuration to an epoch which is the greatest found in the cluster, plus 1 (unless the node is already the one with the greatest configuration epoch), without requiring agreement from other nodes. Usually a real world resharding involves moving several hundred hash slots (especially in small clusters). Requiring an agreement to generate new configuration epochs during resharding, for each hash slot moved, is inefficient. Moreover it requires an fsync in each of the cluster nodes every time in order to store the new configuration. Because of the way it is performed instead, we only need a new config epoch when the first hash slot is moved, making it much more efficient in production environments.

However because of the two cases above, it is possible (though unlikely) to end with multiple nodes having the same configuration epoch. A resharding operation performed by the system administrator, and a failover happening at the same time (plus a lot of bad luck) could cause `currentEpoch` collisions if they are not propagated fast enough.

Moreover, software bugs and filesystem corruptions can also contribute to multiple nodes having the same configuration epoch.

When masters serving different hash slots have the same `configEpoch`, there are no issues. It is more important that slaves failing over a master have unique configuration epochs.

That said, manual interventions or resharding may change the cluster configuration in different ways. The Redis Cluster main liveness property requires that slot configurations always converge, so under every circumstance we really want all the master nodes to have a different `configEpoch`.

In order to enforce this, a **conflict resolution algorithm** is used in the event that two nodes end up with the same `configEpoch`.

- IF a master node detects another master node is advertising itself with the same `configEpoch`.
- AND IF the node has a lexicographically smaller Node ID compared to the other node claiming the same `configEpoch`.
- THEN it increments its `currentEpoch` by 1, and uses it as the new `configEpoch`.

If there are any set of nodes with the same `configEpoch`, all the nodes but the one with the greatest Node ID will move forward, guaranteeing that, eventually, every node will pick

a unique `configEpoch` regardless of what happened.

This mechanism also guarantees that after a fresh cluster is created, all nodes start with a different `configEpoch` (even if this is not actually used) since `redis-trib` makes sure to use `CONFIG SET-CONFIG-EPOCH` at startup. However if for some reason a node is left misconfigured, it will update its configuration to a different configuration epoch automatically.

Node resets

Nodes can be software reset (without restarting them) in order to be reused in a different role or in a different cluster. This is useful in normal operations, in testing, and in cloud environments where a given node can be reprovisioned to join a different set of nodes to enlarge or create a new cluster.

In Redis Cluster nodes are reset using the [CLUSTER RESET](#) command. The command is provided in two variants:

- `CLUSTER RESET SOFT`
- `CLUSTER RESET HARD`

The command must be sent directly to the node to reset. If no reset type is provided, a soft reset is performed.

The following is a list of operations performed by a reset:

1. Soft and hard reset: If the node is a slave, it is turned into a master, and its dataset is discarded. If the node is a master and contains keys the reset operation is aborted.
2. Soft and hard reset: All the slots are released, and the manual failover state is reset.
3. Soft and hard reset: All the other nodes in the nodes table are removed, so the node no longer knows any other node.
4. Hard reset only: `currentEpoch`, `configEpoch`, and `lastVoteEpoch` are set to 0.
5. Hard reset only: the Node ID is changed to a new random ID.

Master nodes with non-empty data sets can't be reset (since normally you want to reshard data to the other nodes). However, under special conditions when this is appropriate (e.g. when a cluster is totally destroyed with the intent of creating a new one), [FLUSHALL](#) must be executed before proceeding with the reset.

Removing nodes from a cluster

It is possible to practically remove a node from an existing cluster by resharding all its data to other nodes (if it is a master node) and shutting it down. However, the other nodes will still remember its node ID and address, and will attempt to connect with it.

For this reason, when a node is removed we want to also remove its entry from all the other nodes tables. This is accomplished by using the `CLUSTER FORGET <node-id>` command.

The command does two things:

1. It removes the node with the specified node ID from the nodes table.
2. It sets a 60 second ban which prevents a node with the same node ID from being re-added.

The second operation is needed because Redis Cluster uses gossip in order to auto-discover nodes, so removing the node X from node A, could result in node B gossiping about node X to A again. Because of the 60 second ban, the Redis Cluster administration tools have 60 seconds in order to remove the node from all the nodes, preventing the re-addition of the node due to auto discovery.

Further information is available in the [CLUSTER FORGET](#) documentation.

Publish/Subscribe

In a Redis Cluster clients can subscribe to every node, and can also publish to every other node. The cluster will make sure that published messages are forwarded as needed.

The current implementation will simply broadcast each published message to all other nodes, but at some point this will be optimized either using Bloom filters or other algorithms.

Appendix

Appendix A: CRC16 reference implementation in ANSI C

```
/*
 * Copyright 2001–2010 Georges Menie (www.menie.org)
 * Copyright 2010 Salvatore Sanfilippo (adapted to Redis coding style
 * All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 *
 * * Redistributions of source code must retain the above copyrig
 * notice, this list of conditions and the following disclaimer
 * * Redistributions in binary form must reproduce the above copy
 * notice, this list of conditions and the following disclaimer
 * documentation and/or other materials provided with the distr
 * * Neither the name of the University of California, Berkeley n
 * names of its contributors may be used to endorse or promote
 * derived from this software without specific prior written pe
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'
```

```

* EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* DISCLAIMED. IN NO EVENT SHALL THE REGENTS AND CONTRIBUTORS BE LIAB
* DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER C
* ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
* SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

```

```

/* CRC16 implementation according to CCITT standards.

```

```

*
* Note by @antirez: this is actually the XMODEM CRC 16 algorithm, us
* following parameters:
*
* Name                : "XMODEM", also known as "ZMODEM", "CR
* Width                : 16 bit
* Poly                : 1021 (That is actually  $x^{16} + x^{12} +$ 
* Initialization      : 0000
* Reflect Input byte   : False
* Reflect Output CRC   : False
* Xor constant to output CRC : 0000
* Output for "123456789" : 31C3
*/

```

```

static const uint16_t crc16tab[256]= {
    0x0000,0x1021,0x2042,0x3063,0x4084,0x50a5,0x60c6,0x70e7,
    0x8108,0x9129,0xa14a,0xb16b,0xc18c,0xd1ad,0xe1ce,0xf1ef,
    0x1231,0x0210,0x3273,0x2252,0x52b5,0x4294,0x72f7,0x62d6,
    0x9339,0x8318,0xb37b,0xa35a,0xd3bd,0xc39c,0xf3ff,0xe3de,
    0x2462,0x3443,0x0420,0x1401,0x64e6,0x74c7,0x44a4,0x5485,
    0xa56a,0xb54b,0x8528,0x9509,0xe5ee,0xf5cf,0xc5ac,0xd58d,
    0x3653,0x2672,0x1611,0x0630,0x76d7,0x66f6,0x5695,0x46b4,
    0xb75b,0xa77a,0x9719,0x8738,0xf7df,0xe7fe,0xd79d,0xc7bc,
    0x48c4,0x58e5,0x6886,0x78a7,0x0840,0x1861,0x2802,0x3823,
    0xc9cc,0xd9ed,0xe98e,0xf9af,0x8948,0x9969,0xa90a,0xb92b,
    0x5af5,0x4ad4,0x7ab7,0x6a96,0x1a71,0x0a50,0x3a33,0x2a12,
    0xdbfd,0xcbdc,0xfbbf,0xeb9e,0x9b79,0x8b58,0xbb3b,0xab1a,
    0x6ca6,0x7c87,0x4ce4,0x5cc5,0x2c22,0x3c03,0x0c60,0x1c41,
    0xedae,0xfd8f,0xcdec,0xddcd,0xad2a,0xbd0b,0x8d68,0x9d49,
    0x7e97,0x6eb6,0x5ed5,0x4ef4,0x3e13,0x2e32,0x1e51,0x0e70,

```




```

0x7c57,0x8e28,0x9c45,0x1c11,0x3c15,0x2c32,0x1c31,0x8c7c,
0xff9f,0xefbe,0xdfdd,0xcffc,0xbf1b,0xaf3a,0x9f59,0x8f78,
0x9188,0x81a9,0xb1ca,0xa1eb,0xd10c,0xc12d,0xf14e,0xe16f,
0x1080,0x00a1,0x30c2,0x20e3,0x5004,0x4025,0x7046,0x6067,
0x83b9,0x9398,0xa3fb,0xb3da,0xc33d,0xd31c,0xe37f,0xf35e,
0x02b1,0x1290,0x22f3,0x32d2,0x4235,0x5214,0x6277,0x7256,
0xb5ea,0xa5cb,0x95a8,0x8589,0xf56e,0xe54f,0xd52c,0xc50d,
0x34e2,0x24c3,0x14a0,0x0481,0x7466,0x6447,0x5424,0x4405,
0xa7db,0xb7fa,0x8799,0x97b8,0xe75f,0xf77e,0xc71d,0xd73c,
0x26d3,0x36f2,0x0691,0x16b0,0x6657,0x7676,0x4615,0x5634,
0xd94c,0xc96d,0xf90e,0xe92f,0x99c8,0x89e9,0xb98a,0xa9ab,
0x5844,0x4865,0x7806,0x6827,0x18c0,0x08e1,0x3882,0x28a3,
0xcb7d,0xdb5c,0xeb3f,0xfb1e,0x8bf9,0x9bd8,0xabbb,0xbb9a,
0x4a75,0x5a54,0x6a37,0x7a16,0x0af1,0x1ad0,0x2ab3,0x3a92,
0xfd2e,0xed0f,0xdd6c,0xcd4d,0xbdaa,0xad8b,0x9de8,0x8dc9,
0x7c26,0x6c07,0x5c64,0x4c45,0x3ca2,0x2c83,0x1ce0,0x0cc1,
0xef1f,0xff3e,0xcf5d,0xdf7c,0xaf9b,0xbfba,0x8fd9,0x9ff8,
0x6e17,0x7e36,0x4e55,0x5e74,0x2e93,0x3eb2,0x0ed1,0x1ef0
};

uint16_t crc16(const char *buf, int len) {
    int counter;
    uint16_t crc = 0;
    for (counter = 0; counter < len; counter++)
        crc = (crc<<8) ^ crc16tab[((crc>>8) ^ *buf++)&0x00FF];
    return crc;
}

```

This website is open source software. See all credits.

Sponsored by  **redislabs**
HOME OF REDIS