Redis cluster Specification (work in progress)

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.
- No merge operations in order to play well with values size and semantics typical of the Redis data model.
- Write safety: the system tries to retain all the writes originating from clients connected with the majority of the nodes. However there are small windows where acknowledged writes can be lost.
- Availability: Redis Cluster is able to survive to partitions where the majority of the master nodes are reachable and there is at least a reachable slave for every master node that is no longer reachable.

What is described in this document is implemented in the `unstable` branch of the Github Redis repository. Redis Cluster has now entered the beta stage, so new betas are released every month and can be found in the download page of the Redis web site.

Implemented subset

Redis Cluster implements all the single keys 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 belong to the same node.

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

Redis Cluster does not support multiple databases like the stand alone version of Redis, there is just database 0, and SELECT 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 performing slave nodes election to master when needed.

To perform their tasks all the cluster nodes are connected using a TCP bus and a binary protocol (the **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.

Since cluster nodes are not able to proxy requests clients may be redirected to other nodes using redirections 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 take 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, so there are always windows 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 hard to retain all the writes that are performed by clients connected to the majority of masters, with two exceptions:

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 in case the master is unreachable for a long enough period that one of its slaves is promoted.

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 a not updated routing table may write to it before the master is converted to a slave (of the new master) by the cluster.

Practically this is very unlikely to happen because nodes not able to reach the majority of other masters for enough time to be failed over, 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.

Redis Cluster loses a non trivial amount 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 not reachable by the majority of masters for at least `NODE_TIMEOUT`, so if the partition is fixed before that time, no write is lost. When the partition lasts for more than `NODE_TIMEOUT`, the minority side of the cluster will start refusing writes as soon as `NODE_TIMEOUT` time has elapsed, so there is a maximum window after which the minority becomes no longer available, hence no write is accepted and 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 returns available after `NODE_TIMEOUT` plus some more second required for a slave to get elected and failover its master.

This means that Redis Cluster is designed to survive to failures of a few nodes in the cluster, but 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 soon 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) = 0.1111` probabilities that after two nodes are partitioned away from the majority, the cluster will no longer be available, that is about 11% of probabilities.

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).

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 does not wait for other nodes acknowledgment of writes (optional synchronous replication is a work in progress and will be likely added in future releases).

Also, because of the restriction to the subset of commands that don't perform operations on multiple keys, data is never moved between nodes if not in case of resharding.

So 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 allows to scale 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 stand alone Redis node case.

Very high performances and scalability while preserving weak (non CAP) but reasonable forms of consistency 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 since 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 a non trivial involvement of application-side logic.

Keys distribution model

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

All the master nodes will handle a percentage of the 16384 hash slots. When the cluster is **stable**, that means that there is no a cluster reconfiguration in progress (where hash slots are moved from one node to another) a single hash slot will be served exactly 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).

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):

```
HASH_SLOT = CRC16(key) mod 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 bit of the output of CRC16 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 kind 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 two 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 is computed in a different way. Basically 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.
- There is a } character on the right of {
- There are one or more characters between the first occurrence of { and the first occurrence of } after the first occurrence of {.

Then instead of hashing the key, only what is between the first occurrence of { and the first occurrence of } on its right are 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 hashes 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 an 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.

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

Every node has other associated information that all the other nodes know:

- The IP address and TCP port where the node is located.
- A set of flags.

- A set of hash slots served by the node.
- Last time we sent a ping packet using the cluster bus.
- Last time we received a pong packet in reply.
- The time at which we flagged the node as failing.
- The number of slaves of this node.
- The master node ID, if this node is a slave (or 0000000... if it is a master).

Some of this information is available using the `CLUSTER NODES` command that can be sent to all the nodes in the cluster, both master and slave nodes.

The following is an example of output of `CLUSTER NODES` sent to a master node in a small cluster of three nodes.

```
$ redis-cli cluster nodes
d1861060fe6a534d42d8a19aeb36600e18785e04 :0 myself - 0 1318428930 connected 0-1364
3886e65cc906bfd9b1f7e7bde468726a052d1dae 127.0.0.1:6380 master - 1318428930 1318428931 cor
d289c575dcbc4bdd2931585fd4339089e461a27d 127.0.0.1:6381 master - 1318428931 1318428931 cor
```

In the above listing the different fields are in order: node id, address:port, flags, last ping sent, last pong received, link state, slots.

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.

Nodes handshake

Nodes always accept connection in the cluster bus port, and even reply to pings when received, even if the pinging node is not trusted. However all the other packets will be discarded by the node if the 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 will present 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 basically 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 that different Redis clusters will accidentally mix after change of IP addresses or other network related events.

All the nodes actively try to connect to all the other known nodes if the link is down.

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) it will see what node is responsible for the hash slot where the key belongs.

If the hash slot is served by the node, the query is simply processed, otherwise the node will check its internal hash slot -> node ID map and will reply to the client with a MOVED error.

A MOVED error is like the following:

```
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 need to reissue the query to the specified 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.

So while from the point of view of the cluster nodes are identified by IDs we try to simply 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 pick the right node with higher chances.

Note that when the Cluster is stable, 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 nor proxies or other single point of failure entities.

A client should be also able to handle -ASK redirections that are described later in this document.

Cluster live reconfiguration

Redis cluster supports the ability to add and remove nodes while the cluster is running. Actually adding or removing a node is abstracted into the same operation, that is, moving an hash slot from a node to another.

- To add a new node to the cluster an empty node is added to the cluster and some hash slot is 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.

So the core of the implementation is the ability to move slots around. Actually from a practical point of view an 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.

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:

- 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. After the hash slots are assigned they will propagate across all the cluster using the gossip protocol. The ADDSLOTS command is usually used when a new cluster is configured from scratch to assign slots to all the nodes in a fast way.

The SETSLOT subcommand is used to assign a slot to a specific node ID if the NODE form is used. Otherwise the slot can be set in the two special states MIGRATING and IMPORTING:

- When a slot is set as MIGRATING, the node will accept all the requests for 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 the requests for queries that are about this hash slot, but only if the request is preceded by an ASKING command. Otherwise if not ASKING command was given by the client, the query is redirected to the real hash slot owner via a –MOVED redirection error.

At first this may appear strange, but now we'll make it more clear. Assume that we have two Redis 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 the queries about already existing keys are processed by "A".
- All the queries about non existing keys in A are processed by "B".

This way we no longer create new keys in "A". In the meantime, a special client that is called redis-trib and is the Redis cluster configuration utility will make sure to migrate existing keys 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 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 will delete the old key from its own dataset. So from the point of view of an external client a key either exists in A or B in a given time.

In Redis cluster there is no need to specify a database other than 0, but MIGRATE can be used for other tasks as well not involving Redis cluster so it is a general enough command. MIGRATE is optimized to be as fast as possible even when moving complex keys such as long lists, but of course 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.

ASK redirection

In the previous section we briefly talked about ASK redirection, why we can't simply use the 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 only ask the next query to the specified node.

This is needed because the next query about hash slot 8 can be about the key that is still in A, so we always want that the client will 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.

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

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

So the full semantics of the ASK redirection is the following, from the point of view of the client.

- If ASK redirection is received send only the query in object to the specified node.
- Start the query with the ASKING command.
- Don't update local client tables to map hash slot 8 to B for now.

Once the 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:port pair. Note that however if a buggy client will perform the map earlier this is not a problem since it will not send the ASKING command before the query and B will redirect the client to A using a MOVED redirection error.

Clients first connection and handling of redirections.

While it is possible to have a Redis Cluster client implementation that does not takes the slots configuration (the map between slot numbers and addresses of nodes serving it) in memory, and only works 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 does not *require* to be updated, since contacting the wrong node will simply result in a redirection.

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

- 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 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 fetching the full map of slots - nodes from scratch.

In order to retrieve the slots configuration Redis Cluster offers (starting with 3.0.0 beta-7) 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 (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` does not guarantee to return ranges that will cover all the 16k 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 will try to execute commands about keys that belong to misconfigured (unassigned) slots.

However before to return an error to the caller, when a slot is found to be 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 multiple-keys operations. For example the following operation is valid:

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

However multi-key operations become unavailable when a resharding of the hash slot the keys are hashing to is being moved form a node to another (because of a manual resharding).

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

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

Fault Tolerance

Nodes heartbeat and gossip messages

Nodes in the cluster exchange ping / pong packets.

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

However every node makes sure to ping every other node that we don't either 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 amount of messages exchanged can be bigger than **O(N)** 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 we don't have fresh information for half the `NODE_TIMEOUT` time.

For example in a 100 nodes 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, that multiplied for 100 nodes is 330 pings per second in the total cluster.

There are ways to use the gossip information already exchanged by Redis Cluster to reduce the amount of messages exchanged in a significant way. For example we may ping within half `NODE_TIMEOUT` only nodes that are already reported to be in "possible failure" state (see later) by other nodes, and ping the other nodes that are reported as working only in a best-effort way within the limit of the few packets per second. However in real-world tests large clusters with very small `NODE_TIMEOUT` settings used to work reliably so this change will be considered in the future as actual deployments of large clusters will be tested.

Ping and Pong packets content

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

The common header has the following information:

- Node ID, that is 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` field, that are used in order to mount the distributed algorithms used by Redis Cluster (this is explained in details in the next sections). If the node is a slave the `configEpoch` is the last known `configEpoch` of the 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 a given node, or if the node is a slave, a bitmap of the slots served by its master.
- Port: the sender TCP base port (that is, the port used by Redis to accept client commands, add 10000 to this to obtain the cluster port).
- State: the state of the cluster from the point of view of the sender (down or ok).
- The master node ID, if this is a slave.

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

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 as a result of this event, either promote a slave to the role of master, of when this is not possible, put the cluster in an error state to stop receiving queries from clients.

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 in 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 still have to get a reply) pending for more than `NODE_TIMEOUT`, so 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 should usually not result into false failure reports between nodes.

FAIL flag:

The `PFAIL` flag alone is just some local information every node has about other nodes, but it is not used in order to act and is not sufficient to trigger a slave promotion. For a node to be really considered down the `PFAIL` condition needs to be promoted 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. So every node eventually receives the set of node flags for every other node. This way every node has a mechanism to signal other nodes about failure conditions they detected.

This mechanism is used in order to escalate a `PFAIL` condition 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 `PFAIL` condition within `NODE_TIMEOUT` $*$ `FAIL_REPORT_VALIDITY_MULT` 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.

Note that *the FAIL flag is mostly one way*, that is, a node can go from `PFAIL` to `FAIL`, but for the `FAIL` flag to be cleared there are only two possibilities:

- The node is already reachable, and it is a slave. In this case the `FAIL` flag can be cleared as slaves are not failed over.

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

While the PFAIL -> FAIL transition uses a form of agreement, the agreement used is weak:

1) Nodes collect views of other nodes during some time, 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 this state is stable.

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 liveness requirement: eventually all the nodes should agree about the state of a given node even in case of partitions, once the partitions heal. 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 an actual majority of masters flagged a node as FAIL, for the chain effect every other node will flag the master as FAIL eventually.

Case 2: When only a minority of masters flagged a node as FAIL, the slave promotion will not happen (as it uses a more formal algorithm that makes sure everybody will know about the promotion eventually) and every node will clear the FAIL state for the FAIL state clearing rules above (no promotion after some time > of N times the NODE_TIMEOUT).

Basically 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 the provide 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 to force the propagation of the state in the shortest amount of time in the reachable part of the cluster, have practical advantages. Because of this mechanisms usually all the nodes will stop accepting writes about at the same time if the cluster is in an error condition, that is a desirable feature from the point of view of applications using Redis Cluster. Also not needed election attempts, initiated by slaves that can't reach its master for local problems (that is otherwise reachable by the majority of the other master nodes), are avoided.

Cluster 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 an incremental version to events, so that when multiple nodes provide conflicting information, it is 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 ping or pong 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, then currentEpoch is updated to the sender epoch.

Because of this semantics eventually all the nodes will agree to the greater epoch in the cluster.

The way this information is used is when the state 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 whatever a given information wins over one with a smaller epoch.

Config 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 the authorization from a majority of masters. When a slave is authorized, a new unique `configEpoch` is created, the slave turns into a master using the new `configEpoch`.

As explained in the next sections the `configEpoch` helps to resolve conflicts due to different nodes claiming diverging 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 case of slaves the field represents the `configEpoch` of its master 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.

Currently when a node is restarted its `currentEpoch` is set to the greatest `configEpoch` of the known nodes. This is not safe in a crash-recovery system model, and the system will be modified in order to store the currentEpoch in the persistent configuration as well.

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 requires 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 to promote a slave with a reasonable data freshness.

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 replies to arrive for a maximum time of two times the `NODE_TIMEOUT`, but always for at least for 2 seconds.

Once a master 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 to avoid multiple slaves to get elected (even if with a different `configEpoch`) about at the same time.

A slave discards all the AUTH_ACK replies that are received having an epoch that is less than the currentEpoch at the time the vote request was sent, in order to never count as valid votes that are about 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).

A slave does not try to get elected as soon as the master is in FAIL state, but there is a little delay, that is computed as:

```
DELAY = 500 milliseconds + random delay between 0 and 500 milliseconds +
        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 when the masters are still not aware of the FAIL state, refusing to grant their vote.

The random delay is used to desynchronize slaves so they'll likely start an election in different moments.

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

Once a slave wins the election, it starts advertising itself as master in ping and pong packets, providing the set of served slots with a configEpoch set to the currentEpoch at which the election was started.

In order to speedup the reconfiguration of other nodes, a pong packet is broadcasted to all the nodes of the cluster (however nodes not currently reachable will eventually receive a ping or pong packet and will be reconfigured).

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 the configuration. Slaves of the old master, or the failed over master that rejoins the cluster, will not just upgrade the configuration but will also configure to replicate from the new master.

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 an vote request, the lastVoteEpoch is updated accordingly.
- 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 this rule:

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 tries to be elected again, with epoch 5 (4+1), the delayed reply reaches to slave with `currentEpoch` 5, and is accepted as valid.
- 4) Masters don't vote a slave of the same master before `NODE_TIMEOUT * 2` has elapsed since a slave of that master was already voted. This is not strictly required as it is not possible that two slaves win the election in the same epoch, but in practical terms it ensures that normally when a slave is elected it has plenty of time to inform the other slaves avoiding that another slave will win a new election.
- 5) Masters don't try to select the best slave in any way, simply if the slave's master is in `FAIL` state and the master did not voted in the current term, the positive vote is granted. However the best slave is the most likely to start the election and win it before the other slaves.
- 6) When a master refuses to vote for a given slave there is no negative response, the request is simply ignored.
- 7) Masters don't grant the vote to 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. What this means is basically 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.

Race conditions during slaves election

This section illustrates how the concept of epoch 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 as master.
- A 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 and will compete with C that will try to get elected in order to fail over B.

Both will eventually claim to be promoted slaves for the same set of hash slots, however the `configEpoch` they publish will be different, and the C epoch will be greater, so all the other nodes will upgrade their configuration to C.

A itself will detect pings from C serving the same slots with a greater epoch and will reconfigure as a slave of C.

Rules for server slots information 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.

Ping and Pong packets that instances continuously exchange contain an header that is used by the sender in order to advertise the hash slots it claims to be responsible for. This is the main mechanism used in order to propagate change, with the exception of a manual reconfiguration operated by the cluster administrator (for example a manual resharding via redis-trib in order to move hash slots among masters).

When a new Redis Cluster node is created, its local slot table, that maps a given hash slot with a given node ID, is initialized so that every hash slot is assigned to nil, that is, the hash slot is unassigned.

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

Rule 1: If an hash slot is unassigned, and a known node claims it, I'll modify my hash slot table to associate the hash slot to this node.

Because of this rule, when a new cluster is created, it is only needed to manually assign (using the `CLUSTER` command, usually via the redis-trib command line tool) the slots served by each master node to the node itself, and the information will rapidly propagate across the cluster.

However this rule is not enough when a configuration update happens because of a slave gets promoted to master after a master failure. The new master instance will advertise the slots previously served by the old slave, but those slots are not unassigned from the point of view of the other nodes, that will not upgrade the configuration if they just follow the first rule.

For this reason there is a second rule that is used in order to rebind an hash slot already assigned to a previous node to a new node claiming it. The rule is the following:

Rule 2: If an hash slot is already assigned, and a known node is advertising it using a `configEpoch` that is greater than the `configEpoch` advertised by the current owner of the slot, I'll rebind the hash slot to the new node.

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

UPDATE messages

The described system for the propagation of hash slots configurations only uses the normal ping and pong messages exchanged by nodes.

It also requires that there is a node that is either a slave or a master for a given hash slot and has the updated configuration, because nodes send their own configuration in pong and pong packets headers.

However sometimes a node may recover after a partition in a setup where it is the only node serving a given hash slot, but with an old configuration.

Example: a given hash slot is served by node A and B. A is the master, and at some point fails, so B is promoted as master. Later B fails as well, and the cluster has no way to recover since there are no more replicas for this hash slot.

However A may recover some time later, and rejoin the cluster with an old configuration in which it was writable as a master. There is no replica that can update its configuration. This is the goal of UPDATE messages: when a node detects that another node is advertising its hash slots with an old configuration, it sends the node an UPDATE message with the ID of the new node serving the slots and the set of hash slots (send as a bitmap) that it is serving.

NOTE: while currently configuration updates via ping / pong and UPDATE share the same code path, there is a functional overlap between the two in the way they update a configuration of a node with stale information. However the two mechanisms are both useful because ping / pong messages after some time are able to populate the hash slots routing table of a new node, while UPDATE messages are only sent when an old configuration is detected, and only cover the information needed to fix the wrong configuration.

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 there is limited availability 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 the operations as long 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 slave.
- Three hours later A1 fails in an independent manner (not related to the failure of A). No other slave is available for promotion since also 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 slaves layout in a Redis Cluster is not part of the cluster configuration that requires 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 vision of the failure state of other nodes.

The *acting slave* is the slave among the masters having 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 think to be the non-failing slave with the lower node ID (but it is an hard to trigger condition 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 left the ceding master without slaves, as soon as the cluster is stable again the algorithm will be re-executed again and will migrate the slave back to the original master.

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

The algorithm is controlled by an user-configurable parameter called `cluster-migration-barrier`, that is the number of good slaves a master will be left with for a slave to migrate. So for example if this parameter is set to 2, a slave will try to migrate only if its master remains with two working slaves.

configEpoch conflicts resolution algorithm

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

However during manual reshardings, when an 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 to require for an agreement from other nodes. This is needed so that the new slot configuration will win over the old one.

This process happens when the system administator performs a manual resharding, however it is possible that when the slot is closed after a resharding and the node assigns itself a new configuration epoch, at the same time a failure happens, just before the new `configEpoch` is propagated to the cluster. A slave may start a failover and obtain the authorization.

This scenario may lead to two nodes having the same `configEpoch`. There are other scenarios as well ending with two nodes having the same `configEpoch`:

- New cluster creation: all nodes start with the same `configEpoch` of 0.
- Possible software bugs.
- Manual editing of the configurations, filesystem corruptions.

When masters serving different hash slots have the same `configEpoch`, there are no issues, and we are more interested in making sure slaves failing over a master have a different and unique configuration epoch.

However manual interventions or more reshardings may change the cluster configuration in different ways. The Redis Cluster main liveness property is that the slot configuration always converges, so we really want under every condition that all the master nodes have a different `configEpoch`.

In order to enforce this, a conflicts resolution is used in the event that two nodes end 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 every node will pick an unique configEpoch regardless of what happened.

This mechanism also guarantees that after a fresh cluster is created all nodes start with a different `configEpoch`.

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 publish messages are forwarded as needed.

The current implementation will simply broadcast all the publish messages to all the other nodes, but at some point this will be optimized either using bloom filters or other algorithms.

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 are met:
 *
 *     * Redistributions of source code must retain the above copyright
 *       notice, this list of conditions and the following disclaimer.
 *     * Redistributions in binary form must reproduce the above copyright
 *       notice, this list of conditions and the following disclaimer in the
 *       documentation and/or other materials provided with the distribution.
 *     * Neither the name of the University of California, Berkeley nor the
 *       names of its contributors may be used to endorse or promote products
 *       derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE REGENTS AND CONTRIBUTORS BE LIABLE FOR ANY
 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * 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, using the
 * following parameters:
 *
 * Name                       : "XMODEM", also known as "ZMODEM", "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
 */

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,
    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;
}
```