



Replication

At the base of Redis replication (excluding the high availability features provided as an additional layer by Redis Cluster or Redis Sentinel) there is a very simple to use and configure *leader follower* (master-slave) replication: it allows replica Redis instances to be exact copies of master instances. The replica will automatically reconnect to the master every time the link breaks, and will attempt to be an exact copy of it *regardless* of what happens to the master.

This system works using three main mechanisms:

1. When a master and a replica instances are well-connected, the master keeps the replica updated by sending a stream of commands to the replica, in order to replicate the effects on the dataset happening in the master side due to: client writes, keys expired or evicted, any other action changing the master dataset.
2. When the link between the master and the replica breaks, for network issues or because a timeout is sensed in the master or the replica, the replica reconnects and attempts to proceed with a partial resynchronization: it means that it will try to just obtain the part of the stream of commands it missed during the disconnection.
3. When a partial resynchronization is not possible, the replica will ask for a full resynchronization. This will involve a more complex process in which the master needs to create a snapshot of all its data, send it to the replica, and then continue sending the stream of commands as the dataset changes.

Redis uses by default asynchronous replication, which being low latency and high performance, is the natural replication mode for the vast majority of Redis use cases. However, Redis replicas asynchronously acknowledge the amount of data they received periodically with the master. So the master does not wait every time for a command to be processed by the replicas, however it knows, if needed, what replica already processed what command. This allows having optional synchronous replication.

Synchronous replication of certain data can be requested by the clients using the [WAIT](#) command. However [WAIT](#) is only able to ensure that there are the specified number of acknowledged copies in the other Redis instances, it does not turn a set of Redis instances into a CP system with strong consistency: acknowledged writes can still be lost during a failover, depending on the exact configuration of the Redis persistence. However with [WAIT](#) the probability of losing a write after a failure event is greatly reduced to certain hard to trigger failure modes.

You could check the Sentinel or Redis Cluster documentation for more information about high availability and failover. The rest of this document mainly describe the basic

characteristics of Redis basic replication.

The following are some very important facts about Redis replication:

- Redis uses asynchronous replication, with asynchronous replica-to-master acknowledges of the amount of data processed.
- A master can have multiple replicas.
- Replicas are able to accept connections from other replicas. Aside from connecting a number of replicas to the same master, replicas can also be connected to other replicas in a cascading-like structure. Since Redis 4.0, all the sub-replicas will receive exactly the same replication stream from the master.
- Redis replication is non-blocking on the master side. This means that the master will continue to handle queries when one or more replicas perform the initial synchronization or a partial resynchronization.
- Replication is also largely non-blocking on the replica side. While the replica is performing the initial synchronization, it can handle queries using the old version of the dataset, assuming you configured Redis to do so in `redis.conf`. Otherwise, you can configure Redis replicas to return an error to clients if the replication stream is down. However, after the initial sync, the old dataset must be deleted and the new one must be loaded. The replica will block incoming connections during this brief window (that can be as long as many seconds for very large datasets). Since Redis 4.0 it is possible to configure Redis so that the deletion of the old data set happens in a different thread, however loading the new initial dataset will still happen in the main thread and block the replica.
- Replication can be used both for scalability, in order to have multiple replicas for read-only queries (for example, slow $O(N)$ operations can be offloaded to replicas), or simply for improving data safety and high availability.
- It is possible to use replication to avoid the cost of having the master writing the full dataset to disk: a typical technique involves configuring your master `redis.conf` to avoid persisting to disk at all, then connect a replica configured to save from time to time, or with AOF enabled. However this setup must be handled with care, since a restarting master will start with an empty dataset: if the replica tries to synchronize with it, the replica will be emptied as well.

Safety of replication when master has persistence turned off

In setups where Redis replication is used, it is strongly advised to have persistence turned on in the master and in the replicas. When this is not possible, for example because of latency concerns due to very slow disks, instances should be configured to **avoid restarting automatically** after a reboot.

To better understand why masters with persistence turned off configured to auto restart are dangerous, check the following failure mode where data is wiped from the master and all its replicas:

1. We have a setup with node A acting as master, with persistence turned down, and nodes B and C replicating from node A.
2. Node A crashes, however it has some auto-restart system, that restarts the process. However since persistence is turned off, the node restarts with an empty data set.
3. Nodes B and C will replicate from node A, which is empty, so they'll effectively destroy their copy of the data.

When Redis Sentinel is used for high availability, also turning off persistence on the master, together with auto restart of the process, is dangerous. For example the master can restart fast enough for Sentinel to not detect a failure, so that the failure mode described above happens.

Every time data safety is important, and replication is used with master configured without persistence, auto restart of instances should be disabled.

How Redis replication works

Every Redis master has a replication ID: it is a large pseudo random string that marks a given story of the dataset. Each master also takes an offset that increments for every byte of replication stream that it is produced to be sent to replicas, in order to update the state of the replicas with the new changes modifying the dataset. The replication offset is incremented even if no replica is actually connected, so basically every given pair of:

Replication ID, offset

Identifies an exact version of the dataset of a master.

When replicas connect to masters, they use the `PSYNC` command in order to send their old master replication ID and the offsets they processed so far. This way the master can send just the incremental part needed. However if there is not enough *backlog* in the master buffers, or if the replica is referring to an history (replication ID) which is no longer known, than a full resynchronization happens: in this case the replica will get a full copy of the dataset, from scratch.

This is how a full synchronization works in more details:

The master starts a background saving process in order to produce an RDB file. At the same time it starts to buffer all new write commands received from the clients. When the background saving is complete, the master transfers the database file to the replica, which saves it on disk, and then loads it into memory. The master will then send all buffered commands to the replica. This is done as a stream of commands and is in the same format of the Redis protocol itself.

You can try it yourself via telnet. Connect to the Redis port while the server is doing some work and issue the `SYNC` command. You'll see a bulk transfer and then every command

received by the master will be re-issued in the telnet session. Actually **SYNC** is an old protocol no longer used by newer Redis instances, but is still there for backward compatibility: it does not allow partial resynchronizations, so now **PSYNC** is used instead. As already said, replicas are able to automatically reconnect when the master-replica link goes down for some reason. If the master receives multiple concurrent replica synchronization requests, it performs a single background save in order to serve all of them.

Replication ID explained

In the previous section we said that if two instances have the same replication ID and replication offset, they have exactly the same data. However it is useful to understand what exactly is the replication ID, and why instances have actually two replication IDs the main ID and the secondary ID.

A replication ID basically marks a given *history* of the data set. Every time an instance restarts from scratch as a master, or a replica is promoted to master, a new replication ID is generated for this instance. The replicas connected to a master will inherit its replication ID after the handshake. So two instances with the same ID are related by the fact that they hold the same data, but potentially at a different time. It is the offset that works as a logical time to understand, for a given history (replication ID) who holds the most updated data set.

For instance, if two instances A and B have the same replication ID, but one with offset 1000 and one with offset 1023, it means that the first lacks certain commands applied to the data set. It also means that A, by applying just a few commands, may reach exactly the same state of B.

The reason why Redis instances have two replication IDs is because of replicas that are promoted to masters. After a failover, the promoted replica requires to still remember what was its past replication ID, because such replication ID was the one of the former master. In this way, when other replicas will synchronize with the new master, they will try to perform a partial resynchronization using the old master replication ID. This will work as expected, because when the replica is promoted to master it sets its secondary ID to its main ID, remembering what was the offset when this ID switch happened. Later it will select a new random replication ID, because a new history begins. When handling the new replicas connecting, the master will match their IDs and offsets both with the current ID and the secondary ID (up to a given offset, for safety). In short this means that after a failover, replicas connecting to the newly promoted master don't have to perform a full sync.

In case you wonder why a replica promoted to master needs to change its replication ID after a failover: it is possible that the old master is still working as a master because of some network partition: retaining the same replication ID would violate the fact that the same ID and same offset of any two random instances mean they have the same data set.

Diskless replication

Normally a full resynchronization requires creating an RDB file on disk, then reloading the same RDB from disk in order to feed the replicas with the data.

With slow disks this can be a very stressing operation for the master. Redis version 2.8.18 is the first version to have support for diskless replication. In this setup the child process directly sends the RDB over the wire to replicas, without using the disk as intermediate storage.

Configuration

To configure basic Redis replication is trivial: just add the following line to the replica configuration file:

```
replicaof 192.168.1.1 6379
```

Of course you need to replace 192.168.1.1 6379 with your master IP address (or hostname) and port. Alternatively, you can call the [REPLICAOF](#) command and the master host will start a sync with the replica.

There are also a few parameters for tuning the replication backlog taken in memory by the master to perform the partial resynchronization. See the example `redis.conf` shipped with the Redis distribution for more information.

Diskless replication can be enabled using the `repl-diskless-sync` configuration parameter. The delay to start the transfer in order to wait for more replicas to arrive after the first one is controlled by the `repl-diskless-sync-delay` parameter. Please refer to the example `redis.conf` file in the Redis distribution for more details.

Read-only replica

Since Redis 2.6, replicas support a read-only mode that is enabled by default. This behavior is controlled by the `replica-read-only` option in the `redis.conf` file, and can be enabled and disabled at runtime using [CONFIG SET](#).

Read-only replicas will reject all write commands, so that it is not possible to write to a replica because of a mistake. This does not mean that the feature is intended to expose a replica instance to the internet or more generally to a network where untrusted clients exist, because administrative commands like `DEBUG` or `CONFIG` are still enabled. However, security of read-only instances can be improved by disabling commands in `redis.conf` using the `rename-command` directive.

You may wonder why it is possible to revert the read-only setting and have replica instances that can be targeted by write operations. While those writes will be discarded if

the replica and the master resynchronize or if the replica is restarted, there are a few legitimate use case for storing ephemeral data in writable replicas.

For example computing slow Set or Sorted set operations and storing them into local keys is an use case for writable replicas that was observed multiple times.

However note that **writable replicas before version 4.0 were incapable of expiring keys with a time to live set**. This means that if you use [EXPIRE](#) or other commands that set a maximum TTL for a key, the key will leak, and while you may no longer see it while accessing it with read commands, you will see it in the count of keys and it will still use memory. So in general mixing writable replicas (previous version 4.0) and keys with TTL is going to create issues.

Redis 4.0 RC3 and greater versions totally solve this problem and now writable replicas are able to evict keys with TTL as masters do, with the exceptions of keys written in DB numbers greater than 63 (but by default Redis instances only have 16 databases).

Also note that since Redis 4.0 replica writes are only local, and are not propagated to sub-replicas attached to the instance. Sub-replicas instead will always receive the replication stream identical to the one sent by the top-level master to the intermediate replicas. So for example in the following setup:

```
A ----> B ----> C
```

Even if B is writable, C will not see B writes and will instead have identical dataset as the master instance A.

Setting a replica to authenticate to a master

If your master has a password via `requirepass`, it's trivial to configure the replica to use that password in all sync operations.

To do it on a running instance, use `redis-cli` and type:

```
config set masterauth <password>
```

To set it permanently, add this to your config file:

```
masterauth <password>
```

Allow writes only with N attached replicas

Starting with Redis 2.8, it is possible to configure a Redis master to accept write queries only if at least N replicas are currently connected to the master.

However, because Redis uses asynchronous replication it is not possible to ensure the replica actually received a given write, so there is always a window for data loss.

This is how the feature works:

- Redis replicas ping the master every second, acknowledging the amount of replication stream processed.
- Redis masters will remember the last time it received a ping from every replica.
- The user can configure a minimum number of replicas that have a lag not greater than a maximum number of seconds.

If there are at least N replicas, with a lag less than M seconds, then the write will be accepted.

You may think of it as a best effort data safety mechanism, where consistency is not ensured for a given write, but at least the time window for data loss is restricted to a given number of seconds. In general bound data loss is better than unbound one.

If the conditions are not met, the master will instead reply with an error and the write will not be accepted.

There are two configuration parameters for this feature:

- `min-replicas-to-write <number of replicas>`
- `min-replicas-max-lag <number of seconds>`

For more information, please check the example `redis.conf` file shipped with the Redis source distribution.

How Redis replication deals with expires on keys

Redis expires allow keys to have a limited time to live (TTL). Such a feature depends on the ability of an instance to count the time, however Redis replicas correctly replicate keys with expires, even when such keys are altered using Lua scripts.

To implement such a feature Redis cannot rely on the ability of the master and replica to have synchronized clocks, since this is a problem that cannot be solved and would result in race conditions and diverging data sets, so Redis uses three main techniques in order to make the replication of expired keys able to work:

1. Replicas don't expire keys, instead they wait for masters to expire the keys. When a master expires a key (or evict it because of LRU), it synthesizes a `DEL` command which is transmitted to all the replicas.
2. However because of master-driven expire, sometimes replicas may still have in memory keys that are already logically expired, since the master was not able to provide the `DEL` command in time. In order to deal with that the replica uses its logical clock in order to report that a key does not exist **only for read operations** that

don't violate the consistency of the data set (as new commands from the master will arrive). In this way replicas avoid reporting logically expired keys are still existing. In practical terms, an HTML fragments cache that uses replicas to scale will avoid returning items that are already older than the desired time to live.

3. During Lua scripts executions no key expiries are performed. As a Lua script runs, conceptually the time in the master is frozen, so that a given key will either exist or not for all the time the script runs. This prevents keys expiring in the middle of a script, and is needed in order to send the same script to the replica in a way that is guaranteed to have the same effects in the data set.

Once a replica is promoted to a master it will start to expire keys independently, and will not require any help from its old master.

Configuring replication in Docker and NAT

When Docker, or other types of containers using port forwarding, or Network Address Translation is used, Redis replication needs some extra care, especially when using Redis Sentinel or other systems where the master `INFO` or `ROLE` commands output is scanned in order to discover replicas' addresses.

The problem is that the `ROLE` command, and the replication section of the `INFO` output, when issued into a master instance, will show replicas as having the IP address they use to connect to the master, which, in environments using NAT may be different compared to the logical address of the replica instance (the one that clients should use to connect to replicas).

Similarly the replicas will be listed with the listening port configured into `redis.conf`, that may be different from the forwarded port in case the port is remapped.

In order to fix both issues, it is possible, since Redis 3.2.2, to force a replica to announce an arbitrary pair of IP and port to the master. The two configurations directives to use are:

```
replica-announce-ip 5.5.5.5
replica-announce-port 1234
```

And are documented in the example `redis.conf` of recent Redis distributions.

The INFO and ROLE command

There are two Redis commands that provide a lot of information on the current replication parameters of master and replica instances. One is `INFO`. If the command is called with the `replication` argument as `INFO replication` only information relevant to the replication are displayed. Another more computer-friendly command is `ROLE`, that

provides the replication status of masters and replicas together with their replication offsets, list of connected replicas and so forth.

Partial resynchronizations after restarts and failovers


Since Redis 4.0, when an instance is promoted to master after a failover, it will be still able to perform a partial resynchronization with the replicas of the old master. To do so, the replica remembers the old replication ID and offset of its former master, so can provide part of the backlog to the connecting replicas even if they ask for the old replication ID.

However the new replication ID of the promoted replica will be different, since it constitutes a different history of the data set. For example, the master can return available and can continue accepting writes for some time, so using the same replication ID in the promoted replica would violate the rule that a of replication ID and offset pair identifies only a single data set.

Moreover, replicas - when powered off gently and restarted - are able to store in the RDB file the information needed in order to resynchronize with their master. This is useful in case of upgrades. When this is needed, it is better to use the [SHUTDOWN](#) command in order to perform a `save & quit` operation on the replica.

It is not possible to partially resynchronize a replica that restarted via the AOF file. However the instance may be turned to RDB persistence before shutting down it, than can be restarted, and finally AOF can be enabled again.

This website is open source software. See all credits.

Sponsored by  **redislabs**
HOME OF REDIS