Note this tutorial requires Redis version 3.0 or higher.

The PCF Redis tile is just single instance Redis. We need use the Redis cluster broker to import the existing Redis cluster into the PCF.

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

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

The normal Redis TCP port used to serve clients, for example 6379, plus the port obtained by adding 10000 to the data port, so 16379 in the example. This second *high* port is used for the Cluster bus, that is a node-to-node communication channel using a binary protocol.  Clients should never try to communicate with the cluster bus port, but always with the normal Redis command port.

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

In order to make Docker compatible with Redis Cluster you need to use the **host networking mode** of Docker. Please check the --net=host option in the [Docker documentation](https://docs.docker.com/engine/userguide/networking/dockernetworks/) for more information.

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

There are 16384 hash slots in Redis Cluster,

* Node A contains hash slots from 0 to 5500.
* Node B contains hash slots from 5501 to 11000.
* Node C contains hash slots from 11001 to 16383.

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

Redis Cluster master-slave model

Redis Cluster uses a master-slave model where every hash slot has from 1 (the master itself) to N replicas (N-1 additional slaves nodes).

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

Redis Cluster consistency guarantees

Redis Cluster is not able to guarantee **strong consistency**.

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

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

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

Note that there is a **maximum window** to the amount of writes Z1 will be able to send to B:

# Redis Cluster configuration parameters

let's introduce the configuration parameters that Redis Cluster introduces in the redis.conf file.

# Creating and using a Redis Cluster

**Creating a Redis Cluster using the create-cluster script**.

To create a cluster, the first thing we need is to have a few empty Redis instances running in **cluster mode**.

Note that the **minimal cluster** that works as expected requires to contain at least three master nodes.

This ID will be used forever by this specific instance in order for the instance to have a unique name in the context of the cluster. Every node remembers every other node using this IDs, and not by IP or port.

## Creating the cluster

This is very easy to accomplish as we are helped by the Redis Cluster command line utility called redis-trib

## Creating a Redis Cluster using the create-cluster script

ust check utils/create-cluster directory in the Redis distribution. There is a script called create-clusterinside (same name as the directory it is contained into), it's a simple bash script.

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

the first node will start at port 30001 by default.

There is some framework we can use:

[redis-rb-cluster](http://github.com/antirez/redis-rb-cluster)

[redis-py-cluster](https://github.com/Grokzen/redis-py-cluster)

[StackExchange.Redis](https://github.com/StackExchange/StackExchange.Redis)

## Resharding the cluster

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

To start a resharding just type:

./redis-trib.rb reshard 127.0.0.1:7000

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

## Scripting a resharding operation

Reshardings can be performed automatically without the need to manually enter the parameters in an interactive way.

./redis-trib.rb reshard --from <node-id> --to <node-id> --slots <number of slots> --yes <host>:<port>

## Testing the failover

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

$ redis-cli -p 7000 cluster nodes | grep master

3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385482984082 0 connected 5960-10921

2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 master - 0 1385482983582 0 connected 11423-16383

97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5959 10922-11422

 Let's crash node 7002 with the **DEBUG SEGFAULT** command:

$ redis-cli -p 7002 debug segfault

Then we should see the error, the e can now check what is the cluster setup after the failover

$ redis-cli -p 7000 cluster nodes

The output of the [CLUSTER NODES](https://redis.io/commands/cluster-nodes) command may look intimidating, but it is actually pretty simple, and is composed of the following tokens:

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

## Manual failover

Manual failovers are supported by Redis Cluster using the [CLUSTER FAILOVER](https://redis.io/commands/cluster-failover) command, that must be executed in one of the **slaves** of the master you want to failover.

Manual failovers are special and are safer compared to failovers resulting from actual master failures, since they occur in a way that avoid data loss in the process,

## Adding a new node

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

At this point the server should be running.

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

./redis-trib.rb add-node 127.0.0.1:7006 127.0.0.1:7000

## Adding a new node as a replica

./redis-trib.rb add-node --slave 127.0.0.1:7006 127.0.0.1:7000

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

./redis-trib.rb add-node --slave --master-id 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7006 127.0.0.1:7000

## Removing a node

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

./redis-trib del-node 127.0.0.1:7000 `<node-id>`

## Replicas migration

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

CLUSTER REPLICATE <master-node-id>

## Upgrading nodes in a Redis Cluster

Upgrading slave nodes is easy since you just need to stop the node and restart it with an updated version of Redis.

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

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