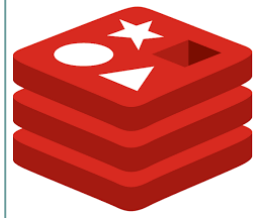
redis

Advanced Concepts

Ali KATKAR



Contents

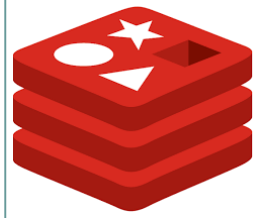
- Configuration
- Persistence
- Replication
- Partitioning
- Redis Cluster
- Redis High Availability
- LRU Caching
- Backup & Security
- Benchmarking

redis

Configuration

Ali KATKAR

Configuration



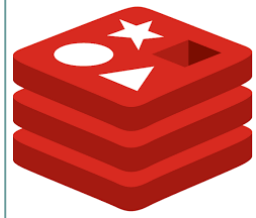
- In Redis, there is a configuration file available at the root directory of Redis.

```
$ sudo gedit /etc/redis/redis.conf
```

- Although you can get and set all Redis configurations by Redis **CONFIG** command.
- Syntax Following is the basic syntax of Redis **CONFIG** command.
  - CONFIG GET CONFIG_SETTING_NAME

```
redis 127.0.0.1:6379> CONFIG GET loglevel
1) "loglevel"
2) "notice"
redis 127.0.0.1:6379> CONFIG GET *
```

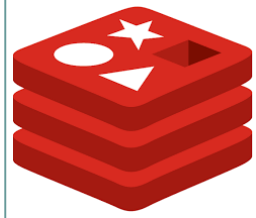
Configuration



- To update configuration:
  - you can edit redis.conf file directly
  - you can update configurations via CONFIG set command.
  - Note that modifying the configuration on the fly **has no effects on the redis.conf file**
- so at the next restart of Redis the old configuration will be used instead.

```
redis 127.0.0.1:6379> CONFIG SET loglevel "notice"
OK
redis 127.0.0.1:6379> CONFIG GET loglevel
1) "loglevel"
2) "notice"
```

Configuration



- Since Redis 2.6 it is possible to also pass Redis configuration parameters using the command line directly.
- This is very useful for testing purposes.
- The following is an example that starts a new Redis instance using port 6380 as a slave of the instance running at 127.0.0.1 port 6379.

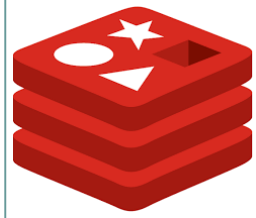
```
./redis-server --port 6380 --slaveof 127.0.0.1 6379
```

redis

Redis Persistence

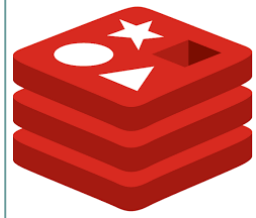
Ali KATKAR

Redis Persistence



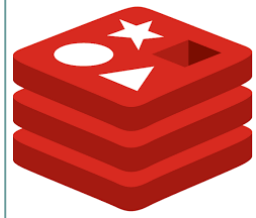
- Redis provides a different range of persistence options
  - The RDB persistence performs point-in-time snapshots of your dataset at specified intervals.
  - the AOF persistence logs every write operation received by the server, that will be played again at server startup, reconstructing the original dataset.
    - Commands are logged using the same format as the Redis protocol itself, in an append-only fashion.
    - Redis is able to rewrite the log on background when it gets too big.
  - If you wish, you can disable persistence at all
    - if you want your data to just exist as long as the server is running.
  - It is possible to combine both AOF and RDB in the same instance.
    - Notice that, in this case, when Redis restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.
- The most important thing to understand is the different trade-offs between the RDB and AOF persistence

Snapshotting



- By default Redis saves snapshots of the dataset on disk, in a binary file called **dump.rdb**.
  - You can configure Redis to have it save the dataset **every N seconds** if there are **at least M changes in** the dataset,
  - or you can manually call the **SAVE** or **BGSAVE** commands.
  - For example, this configuration will make Redis automatically dump the dataset to disk every 60 seconds if at least 1000 keys changed:
    - save 60 1000
- How it works?
- Whenever Redis needs to dump the dataset to disk, this is what happens:
  - Redis forks. We now have a child and a parent process.
  - The child starts to write the dataset to a temporary RDB file.
  - When the child is done writing the new RDB file, it replaces the old one.
  - This method allows Redis to benefit from copy-on-write semantics.

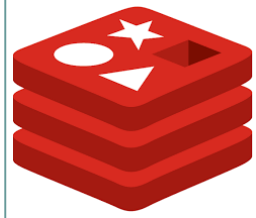
Append-Only File



- Snapshotting is not very durable.
  - If your computer running Redis stops, your power line fails,
  - or you accidentally kill -9 your instance, the latest data written on Redis will get lost.
  - While this may not be a big deal for some applications, there are use cases for full durability, and in these cases Redis was not a viable option.
- The append-only file is an alternative, fully-durable strategy for Redis.
  - It became available in version 1.1.

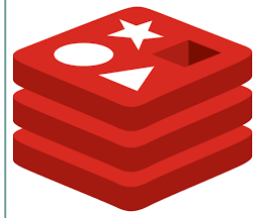
- You can turn on the AOF in your configuration file:
  - append-only yes
- From now on, every time Redis receives a command that changes the dataset (e.g. SET) it will append it to the AOF.
- When you restart Redis it will re-play the AOF to rebuild the state.

RDB Advantages



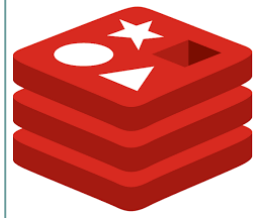
- RDB is a very compact single-file point-in-time representation of your Redis data.
  - RDB files are perfect for backups.
  - For instance you may want to archive your RDB files every hour for the latest 24 hours, and to save an RDB snapshot every day for 30 days.
  - This allows you to easily restore different versions of the data set in case of disasters.
- RDB is very good for disaster recovery
  - being a single compact file can be transferred to far data centers
  - or on Amazon S3 (possibly encrypted).
- RDB maximizes Redis performances since the only work the Redis parent process needs to do in order to persist is forking a child that will do all the rest.
  - The parent instance will never perform disk I/O or alike.
- RDB allows faster restarts with big datasets compared to AOF.

RDB Disadvantages



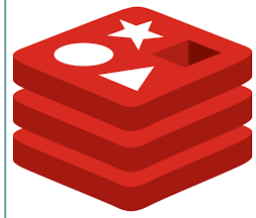
- RDB is NOT good if you need to minimize the chance of data loss in case Redis stops working (for example after a power outage).
  - You can configure different *save points* where an RDB is produced (for instance after at least five minutes and 100 writes against the data set, but you can have multiple save points).
  - However you'll usually create an RDB snapshot every five minutes or more, so in case of Redis stopping working without a correct shutdown for any reason you should be prepared to lose the latest minutes of data.
- RDB needs to fork() often in order to persist on disk using a child process.
  - Fork() can be time consuming if the dataset is big,
  - and may result in Redis to stop serving clients for some millisecond or even for one second if the dataset is very big and the CPU performance not great.
  - AOF also needs to fork() but you can tune how often you want to rewrite your logs without any trade-off on durability.

AOF Advantages



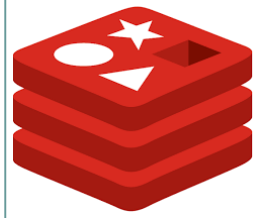
- Using AOF Redis is much more durable:
  - You can have different fsync policies:
    - no fsync at all, fsync every second, fsync at every query.
    - With the default policy of fsync every second write performances are still great but you can only lose one second worth of writes.
- The AOF log is an append only log, so there are no seeks, nor corruption problems if there is a power outage.
- Redis is able to automatically rewrite the AOF in background when it gets too big.
- AOF contains a log of all the operations one after the other in an easy to understand and parse format.

AOF Disadvantages



- AOF files are usually bigger than the equivalent RDB files for the same dataset.
- AOF can be slower than RDB depending on the exact fsync policy.
- In the past we experienced rare bugs in specific commands
  - for instance there was one involving blocking commands like BRPOPLPUSH causing the AOF produced to not reproduce exactly the same dataset on reloading.

What Should I Use?



- The general indication is that **you should use both** persistence methods
  - if you want a degree of data safety comparable to what PostgreSQL can provide you.
- If you care a lot about your data, but still **can live with a few minutes of data loss** in case of disasters, you can simply use RDB alone.
- There are many users using AOF alone, but we discourage it since to have an RDB snapshot from time to time is a great idea for doing database backups, for faster restarts, and in the event of bugs in the AOF engine.

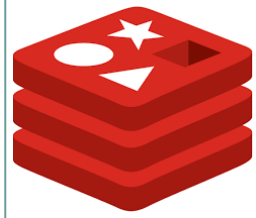
- **Note:** for all these reasons we'll likely end up unifying AOF and RDB into a single persistence model in the future (long term plan).

redis

Redis Replication

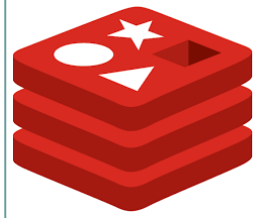
Ali KATKAR

Replication



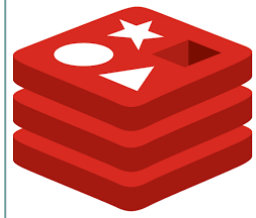
- There is a very simple to use and configure *leader follower* (master-slave) replication:
  - it allows slave Redis instances to be exact copies of master instances.
  - The slave will automatically reconnect to the master every time the link breaks, and will attempt to be an exact copy of it *regardless* of what happens to the master.
- 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 slaves 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 slaves, however it knows, if needed, what slave already processed what command.
  - This allows to have optional syncrhonous 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**
  - However with WAIT the probability of losign a write after a failure event is greatly reduced to certain hard to trigger failure modes.

Replication Mechanism



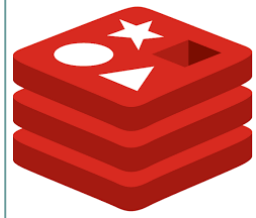
- This system works using three main mechanisms:
  - When a master and a slave instances are well-connected, **the master keeps the slave updated by sending a stream of commands to the slave**, in order to replicate the effects on the dataset happening in the master side due to:
    - client writes, keys expired or evicted, any other action changing the master dataset.
  - When the link between the master and the slave breaks, for network issues or because a timeout is sensed in the master or the slave, **the slave 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.
  - When a partial resynchronization is not possible**, the slave 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 slave, and then continue sending the stream of commands as the dataset changes.

Replication Key Points



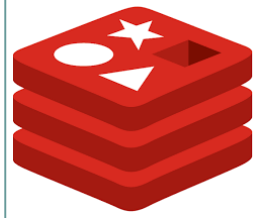
- Redis uses asynchronous replication, with asynchronous slave-to-master acknowledges of the amount of data processed.
- A master can have multiple slaves.
- Slaves are able to accept connections from other slaves.
- Redis replication is non-blocking on the master side.
  - This means that the master will continue to handle queries when one or more slaves perform the initial synchronization or a partial resynchronization.
- Replication is also largely non-blocking on the slave side.
- Replication can be used both for scalability and 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 slave configured to save from time to time, or with AOF enabled.

How It Works?



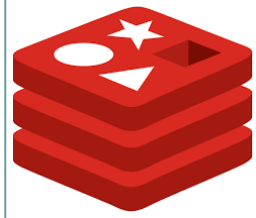
- If you set up a slave, upon connection it sends a SYNC command.
  - It doesn't matter if it's the first time it has connected or if it's a reconnection.
- The master then starts background saving to dump the dataset to a disk file (RDB)
- Starts to buffer all new commands received that will modify the dataset.
  - Any changes made to the data are copied to a replication buffer on the main process
- When the background saving is complete, the master transfers the database file to the slave, which saves it on disk, and then loads it into memory
- The master will then send to the slave all buffered commands

Partial Resynchronization



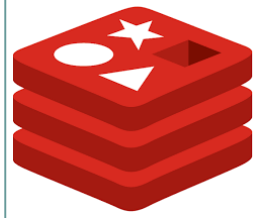
- Starting with Redis 2.8, master and slave are usually able to continue the replication process without requiring a full resynchronization after the replication link went down.
- This works by creating an in-memory backlog of the replication stream on the master side.
- The master and all the slaves agree on a replication offset and a master run id
  - so when the link goes down, the slave will reconnect and ask the master to continue the replication.
- Assuming the master run id is still the same, and that the offset specified is available in the replication backlog, replication will resume from the point where it left off.
- The new partial resynchronization feature uses the PSYNC command internally.

Diskless Replication



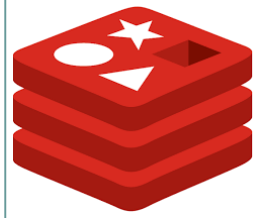
- Normally a full resynchronization requires to create an RDB file on disk
  - Then reload the same RDB from disk in order to feed the slaves with the data.
- With slow disks this can be a very stressing operation for the master.
- Version 2.8.18 was the first version to have experimental support for diskless replication.
- In this setup the child process directly sends the RDB over the wire to slaves, without using the disk as intermediate storage.

Replication Configuration



- To configure replication just add the following line to the slave configuration file:
  - slaveof 127.0.0.1 6379
- Diskless replication can be enabled using the **repl-diskless-sync** configuration parameter
- The delay to start the transfer in order to wait more slaves to arrive after the first one, is controlled by the **repl-diskless-sync-delay** parameter.

Replication Configuration



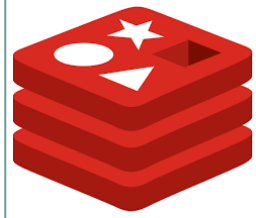
- Starting with Redis 2.8, it is possible to configure a Redis master to accept write queries only if at least N slaves are currently connected to the master.
- The user can configure a minimum number of slaves that have a lag not greater than a maximum number of seconds.
- If there are at least N slaves, with a lag less than M seconds, then the write will be accepted.
- There are two configuration parameters for this feature:
  - min-slaves-to-write <number of slaves>
  - min-slaves-max-lag <number of seconds>

redis

Partitioning

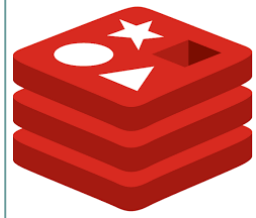
Ali KATKAR

Partitioning



- How to split data among multiple Redis instances?
- **Partitioning** is the process of splitting your data into multiple Redis instances
- Every instance will only contain a subset of your keys.

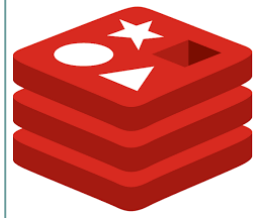
Why Partitioning is Useful?



Partitioning in Redis serves two main goals:

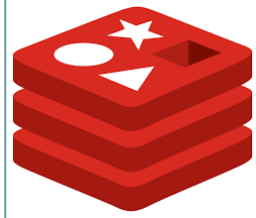
- It allows for much larger databases
  - Using the sum of the memory of many computers
  - Without partitioning you are limited to the amount of memory a single computer can support.
- It allows scaling
  - the computational power to multiple cores and multiple computers,
  - the network bandwidth to multiple computers and network adapters.

Partitioning Basics



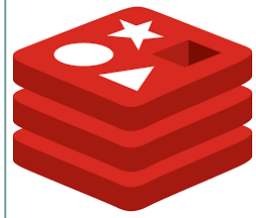
- There are different partitioning criteria
- Imagine we have four Redis instances
  - R0, R1, R2, R3,
- and many keys representing users like
  - user:1, user:2, ... and so forth,
- we can find different ways to select in which instance we store a given key.
- In other words there are different systems to map a given key to a given Redis server.

Range Partitioning



- One of the simplest ways to perform partitioning is **range partitioning**
- Mapped ranges of objects into specific Redis instances.
  - users from ID 0 to ID 10000 → instance **R0**,
  - users form ID 10001 to ID 20000 → instance **R1**
  - … so forth.
- This system works and is actually used in practice,
- However, it has the disadvantage of requiring a table that maps ranges to instances.
  - This table needs to be managed and a table is needed for every kind of object
  - Therefore range partitioning in Redis is often undesirable because **it is much more inefficient** than other alternative partitioning approaches.

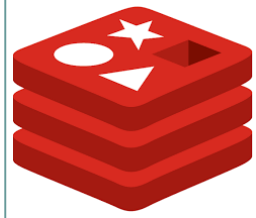
Hash Partitioning



- It works with any key,
  - without requiring a key in the form objectName:<id>

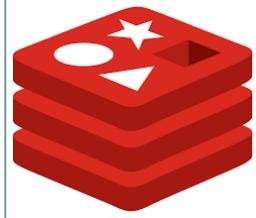
- Take the key name and use a hash function (e.g., the crc32 hash function) to turn it into a number.
  - For example, if the key is foobar, crc32(foobar) will output something like 93024922.
- Use a modulo operation with this number in order to turn it into a number between 0 and 3, so that this number can be mapped to one of my four Redis instances.
  - 93024922 modulo 4 equals 2, so I know my key foobar should be stored into the R2 instance.
  - Note: the modulo operation returns the remainder from a division operation, and is implemented with the % operator in many programming languages.

Partitioning: Advanced Form



- There are many other ways to perform partitioning
  - but with these two examples you should get the idea.
- One advanced form of hash partitioning is called **consistent hashing**
- It is implemented by a few Redis clients and proxies.

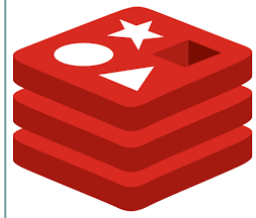
Different Implementations of Partitioning



Partitioning can be the responsibility of different parts of a software stack.

- **Client side partitioning** means that the clients directly select the right node where to write or read a given key.
  - Many Redis clients implement client side partitioning.
- **Proxy assisted partitioning** means that our clients send requests to a proxy that is able to speak the Redis protocol, instead of sending requests directly to the right Redis instance.
  - The proxy will make sure to forward our request to the right Redis instance accordingly to the configured partitioning schema, and will send the replies back to the client.
  - The Redis and Memcached proxy **Twemproxy** implements proxy assisted partitioning.
- **Query routing** means that you can send your query to a random instance, and the instance will make sure to forward your query to the right node.
  - Redis Cluster implements an hybrid form of query routing, with the help of the client (the request is not directly forwarded from a Redis instance to another, but the client gets redirected to the right node).

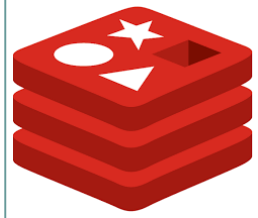
Disadvantages of Partitioning



Some features of Redis don't play very well with partitioning:

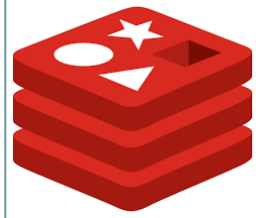
- Operations involving multiple keys are usually not supported.
  - For instance you can't perform the intersection between two sets if they are stored in keys that are mapped to different Redis instances
- Redis transactions involving multiple keys can not be used.
- The partitioning granularity is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set.
- When partitioning is used, data handling is more complex,
  - For instance you have to handle multiple RDB / AOF files,
  - and to make a backup of your data you need to aggregate the persistence files from multiple instances and hosts.
- Adding and removing capacity can be complex.
  - For instance Redis Cluster supports mostly transparent rebalancing of data with the ability to add and remove nodes at runtime, but other systems like client side partitioning and proxies don't support this feature. However a technique called Pre-sharding helps in this regard.

Data Store or Cache?



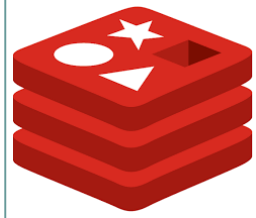
- Partitioning in Redis is conceptually the same for both data store and cache
- However, there is a significant limitation when using it as a data store.
  - For data store, a given key must always map to the same Redis instance.
  - For cahce, if a given node is unavailable it is not a big problem if a different node is used, altering the key-instance map as we wish to improve the *availability* of the system (that is, the ability of the system to reply to our queries).
- Consistent hashing implementations are often able to switch to other nodes if the preferred node for a given key is not available.
- Similarly if you add a new node, part of the new keys will start to be stored on the new node.
- The main concept here is the following:
  - If Redis is used as a cache **scaling up and down** using consistent hashing is easy.
  - If Redis is used as a store, **a fixed keys-to-nodes map is used, so the number of nodes must be fixed and cannot vary**.
    - Otherwise, a system is needed that is able to rebalance keys between nodes when nodes are added or removed, and currently only Redis Cluster is able to do this

Presharding



- We learned that a problem with partitioning is that,
  - unless we are using Redis as a cache, to add and remove nodes can be tricky, and it is much simpler to use a fixed keys-instances map.
- However the data storage needs may vary over the time.
  - Today I can live with 10 Redis nodes (instances), but tomorrow I may need 50 nodes.
- Since Redis is extremely small footprint and lightweight (a spare instance uses 1 MB of memory), a simple approach to this problem is to start with a lot of instances **since the start**.
  - Even if you start with just one server, you can decide to live in a distributed world since your first day, and run multiple Redis instances in your single server, using partitioning.
- And you can select this number of instances to be quite big since the start.
  - For example, 32 or 64 instances could do the trick for most users, and will provide enough room for growth.

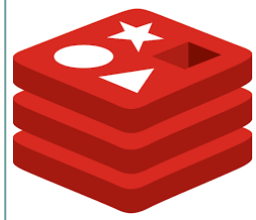
Presharding (cntd)



- In this way as your data storage needs increase and you need more Redis servers, what to do is to simply move instances from one server to another.
  - Once you add the first additional server, you will need to move half of the Redis instances from the first server to the second, and so forth.

- Using Redis replication you will likely be able to do the move with minimal or no downtime for your users:
  - Start empty instances in your new server.
  - Move data configuring these new instances as slaves for your source instances.
  - Stop your clients.
  - Update the configuration of the moved instances with the new server IP address.
  - Send the SLAVEOF NO ONE command to the slaves in the new server.
  - Restart your clients with the new updated configuration.
  - Finally shut down the no longer used instances in the old server.

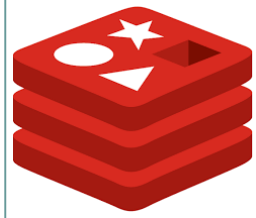
Implementations of Redis Partitioning



- So far we covered Redis partitioning in theory, but what about practice? What system should you use?

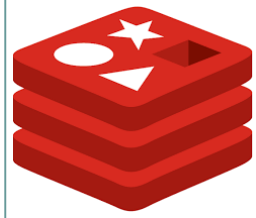
- Implementations of partioning:
  - Redis Cluster
  - Twemproxy
  - Clients supporting consistent hashing

Redis Cluster



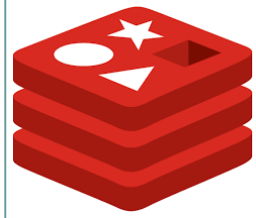
- Redis Cluster is the preferred way to get automatic sharding and high availability.
  - It is generally available and production-ready as of April 1st, 2015.
- Once Redis Cluster will be available, and if a Redis Cluster compliant client is available for your language Redis Cluster will be the de facto standard for Redis partitioning.
- Redis Cluster is a mix between ***query routing*** and ***client side partitioning***.

Twemproxy



- Twemproxy is a proxy developed at Twitter for the Memcached ASCII and the Redis protocol.
  - It is single threaded, it is written in C, and is extremely fast.
  - It is open source software released under the terms of the Apache 2.0 license.
- Twemproxy supports automatic partitioning among multiple Redis instances with optional node ejection if a node is not available
  - *this will change the keys-instances map, so you should use this feature only if you are using Redis as a cache*
- It is *not* a single point of failure since you can start multiple proxies and instruct your clients to connect to the first that accepts the connection.
- Basically Twemproxy is **an intermediate layer** between clients and Redis instances
  - that will reliably handle partitioning for us with minimal additional complexities.
- You can read more about Twemproxy in this antirez blog post.

Clients Supporting Consistent Hashing



- An alternative to Twemproxy is to use a client that implements client side partitioning via consistent hashing or other similar algorithms.
- There are multiple Redis clients with support for consistent hashing, notably Redis-rb and Predis.

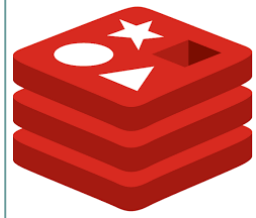
- Please check the full list of Redis clients to check if there is a mature client with consistent hashing implementation for your language.

redis

Redis Cluster

Ali KATKAR

Redis Cluster

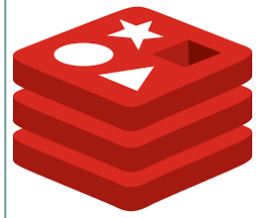


- Redis Cluster provides a way to run a Redis installation where data is **automatically sharded** across multiple Redis nodes.

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

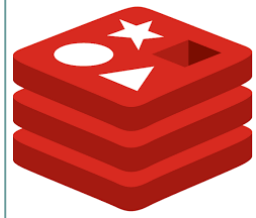
- So in practical terms, what do you get with Redis Cluster?
  - The ability to **automatically split your dataset** among multiple nodes.
  - The ability to **continue operations** when a subset of the nodes are experiencing failures or are unable to communicate with the rest of the cluster.

Redis Cluster TCP ports



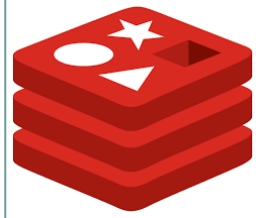
- Every Redis Cluster node requires two TCP connections open.
  - The normal Redis TCP port used to serve clients, for example **6379**,
  - plus the port obtained by adding 10000 to the data port, so **16379** in the example.
- This second high port is used for the **Cluster bus**,
  - This is a node-to-node communication channel using a binary protocol.
  - Is used by nodes for failure detection, configuration update, failover authorization etc.
  - Clients should never try to communicate with the cluster bus port
  - Make sure you open both ports in your firewall
- The command port and cluster bus port offset is fixed and is always 10000.
- Note that for a Redis Cluster to work properly you need, for each node:
  - The cluster bus port (the client port + 10000) must be reachable from all the other cluster nodes.
  - If you don't open both TCP ports, your cluster will not work as expected.
  - The cluster bus uses a different, binary protocol, for node to node data exchange, which is more suited to exchange information between nodes using little bandwidth and processing time.

Redis Cluster and Docker



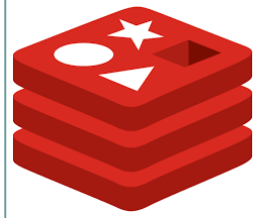
- Currently Redis Cluster does not support NATted environments and in general environments where IP addresses or TCP ports are remapped.
- Docker uses a technique called port mapping:
  - Programs running inside Docker containers may be exposed with a different port compared to the one the program believes to be using.
  - This is useful in order to run multiple containers using the same ports, at the same time, in the same server.
- In order to make Docker compatible with Redis Cluster you need to use the host networking mode of Docker.
  - Please check the --net=host option in the Docker documentation for more information.

Redis Cluster data Sharding



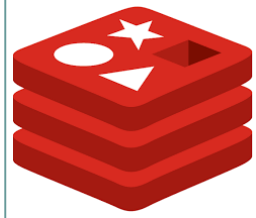
- Redis Cluster does not use consistent hashing, but a different form of sharding where every key is conceptually part of what we call an hash slot.
- There are **16384 hash slots** in Redis Cluster
  - To compute what is the hash slot of a given key, we simply take the CRC16 of the key modulo 16384.
- Every node in a Redis Cluster is responsible for a subset of the hash slots,
  - For example you may have a cluster with 3 nodes, where:
    - Node A contains hash slots from 0 to 5500.
    - Node B contains hash slots from 5501 to 11000.
    - Node C contains hash slots from 11001 to 16383.

Redis Cluster data Sharding



- This allows to add and remove nodes in the cluster easily.
  - For example if I want to add a new node D,
    - I need to move some hash slot from nodes A, B, C to D.
  - Similarly if I want to remove node A from the cluster
    - I can just move the hash slots served by A to B and C.
    - When the node A will be empty I can remove it from the cluster completely.
- Because moving hash slots from a node to another does not require to stop operations
  - adding and removing nodes, or changing the percentage of hash slots hold by nodes, does not require any downtime.

Redis Cluster
Master-Slave Model

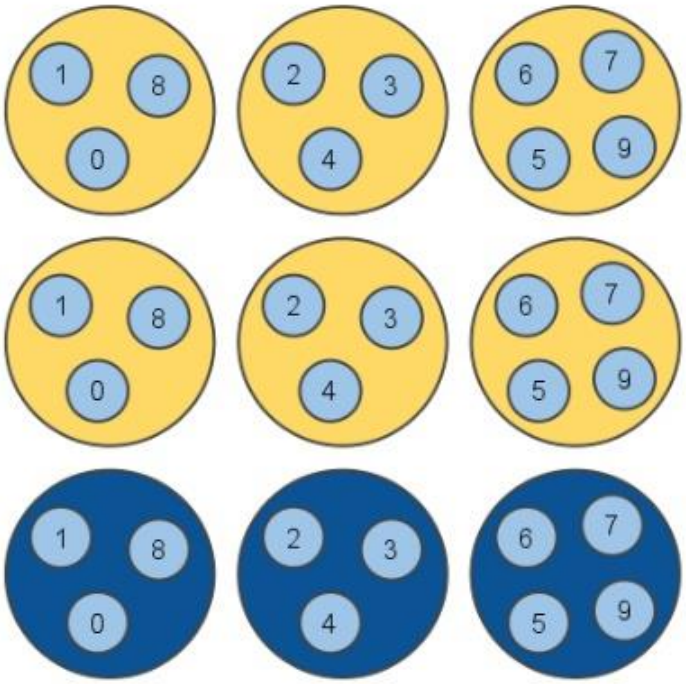


- In order to remain available when a failure happened, Redis Cluster uses a master-slave model
  - Every hash slot has from 1 (the master itself) to N replicas (N-1 additional slaves nodes).
- In our example cluster with nodes A, B, C
  - if node B fails the cluster is not able to continue, since we no longer have a way to serve hash slots in the range 5501-11000.
  - However when the cluster is created (or at a later time) we add a slave node to every master, so that the final cluster is composed of A, B, C that are masters nodes, and A1, B1, C1 that are slaves nodes, the system is able to continue if node B fails.
  - Node B1 replicates B, and B fails, the cluster will promote node B1 as the new master and will continue to operate correctly.
  - However note that if nodes B and B1 fail at the same time Redis Cluster is not able to continue to operate.

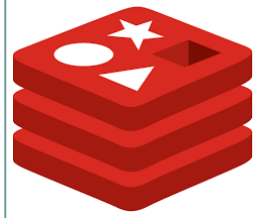
Redis Cluster
Master-Slave Model

Master and Slave nodes

Nodes are all connected and functionally equivalent, but actually there are two kind of nodes: slave and master nodes:



Redis Cluster Configuration Parameters



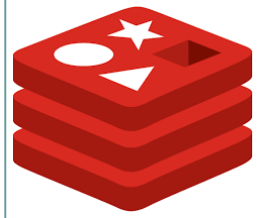
- **cluster-enabled <yes/no>:** If yes enables Redis Cluster support in a specific Redis instance. Otherwise the instance starts as a stand alone instance as usual.
- **cluster-config-file <filename>**
- **cluster-node-timeout <milliseconds>**
- **cluster-slave-validity-factor <factor>**
- **cluster-migration-barrier <count>**
- **cluster-require-full-coverage <yes/no>**

redis

Redis Cluster Demo

Ali KATKAR

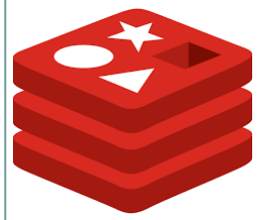
Redis Cluster
Step1: Create Instance



- Create directories:

```
$ cd /opt
$ sudo mkdir redis-cluster
$ cd redis-cluster
$ sudo mkdir scripts
$ sudo mkdir 7000 7001 7002 7003 7004 7005

$ cd 7000
$ sudo gedit redis.conf ➔ minimal cluster cnf
```

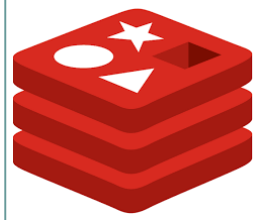
Redis Cluster Configuration Parameters



- A minimal Redis cluster configuration file:

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

Redis Cluster
Step 2: Create Scripts



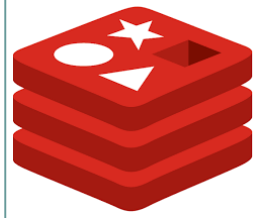
- Create an **instance_start.sh**

```
#!/bin/bash
cd /opt/redis-cluster/$1
/usr/bin/redis-server redis.conf
```

- Create **start_all.sh**

```
#!/bin/bash
for port in {7000..7005}; do
  x-terminal-emulator -e ./instance_start.sh $port
done
```

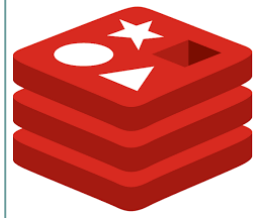
Redis Cluster
Step3: Create Cluster



- Create cluster

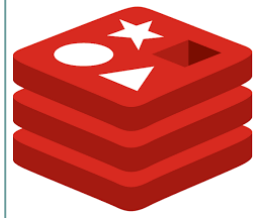
```
$ sudo ./start_all.sh

$ redis-cli --cluster create 127.0.0.1:7000 127.0.0.1:7001
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 --
cluster-replicas 1

can I set the above configuration? (type yes to accept): yes

[OK] All 16384 slots covered
```

Redis Cluster
Step4: Test the Cluster



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

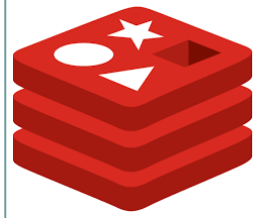
Resharding the Cluster



- Resharding basically means to move hash slots from a set of nodes to another set of nodes

```
$ redis-cli --cluster reshard 127.0.0.1:7000
How many slots do you want to move (from 1 to 16384)?
1000
```

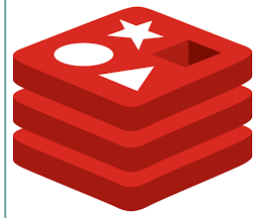
Testing the Failover



- Resharding basically means to move hash slots from a set of nodes to another set of nodes

```
$ redis-cli -p 7000 cluster nodes | grep master
3... 127.0.0.1:7001 master - 0 1385482984082 0 connected 5960-10921
2... 127.0.0.1:7002 master - 0 1385482983582 0 connected 11423-16383
9... :0 myself,master - 0 0 0 connected 0-5959 10922-11422

$ redis-cli -p 7002 debug segfault
```

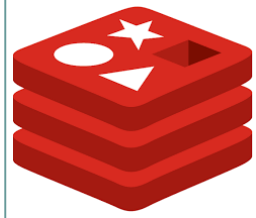
Adding a New Node



- Create a new instance
  - Goto redis-cluster 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
- Now we can use redis-cli in order to add the new node to the existing cluster

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

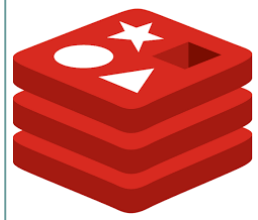
Slots for New Node



- It holds no data as it has no assigned hash slots.
- Because it is a master without assigned slots, it does not participate in the election process when a slave wants to become a master.
- Now it is possible to assign hash slots to this node using the **resharding** feature of redis-cli

```
$ redis-cli --cluster reshard 127.0.0.1:7006
```

Adding a New Node As a Replica



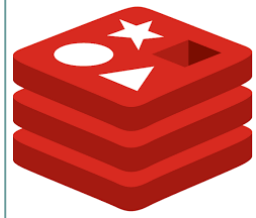
```
$ redis-cli --cluster add-node 127.0.0.1:7006 127.0.0.1:7000 --cluster-slave
```

- we are not specifying to which master we want to add the replica.
- redis-cli will add the new node as replica of **a random master** among the masters with less replicas.

- However you can specify exactly what master you want to target

```
$ redis-cli --cluster add-node 127.0.0.1:7006 127.0.0.1:7000 --cluster-slave
--cluster-master-id 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e
```

Removing a Node



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

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

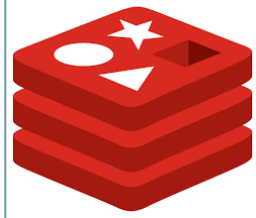
- You can remove a master node in the same way as well, **however in order to remove a master node it must be empty**.
- If the master is not empty you need to reshard data away from it to all the other master nodes before.

redis

Redis Sentinel

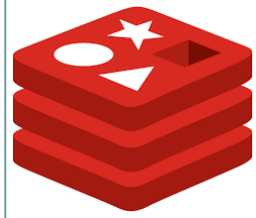
Ali KATKAR

Redis Sentinel



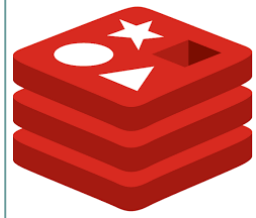
- Redis Sentinel provides high availability for Redis.
  - Using Sentinel you can create a Redis deployment that resists without human intervention to certain kind of failures.
- Redis Sentinel also provides other collateral tasks such as **monitoring**, **notifications** and acts as a **configuration provider** for clients.

Sentinel Capabilities



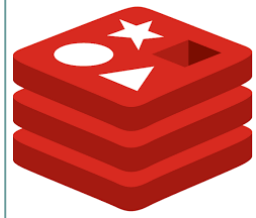
- **Monitoring:** Sentinel constantly checks if your master and slave instances are working as expected.
- **Notification:** Sentinel can notify the system administrator, another computer programs, via an API, that something is wrong with one of the monitored Redis instances.
- **Automatic Failover:** If a master is not working as expected, Sentinel can start a failover process where a slave is promoted to master
  - The other additional slaves are reconfigured to use the new master, and the applications using the Redis server informed about the new address to use when connecting.
- **Configuration Provider**. Sentinel acts as a source of authority for clients service discovery
  - Clients connect to Sentinels in order to ask for the address of the current Redis master responsible for a given service.
  - If a failover occurs, Sentinels will report the new address.

Distributed Nature



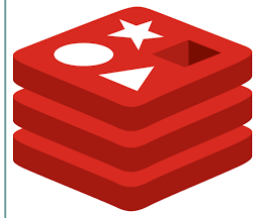
- Redis Sentinel is a distributed system:
- Sentinel itself is designed to run in a configuration where there are **multiple Sentinel processes** cooperating together.
- The advantage of having multiple Sentinel processes
  - Failure detection is performed when multiple Sentinels agree about the fact a given master is no longer available.
    - This lowers the probability of false positives.
  - Sentinel works even if not all the Sentinel processes are working, making the system robust against failures.
    - There is no fun in having a fail over system which is itself a single point of failure, after all.
- Components of a Sentinel
  - **Redis instances** (masters and slaves)
  - **Clients** connecting to Sentinel Redis,
- Sentinels are also a larger distributed system with specific properties.

Sentinel Version



- The current version of Sentinel is called **Sentinel 2**.
  - It is a rewrite of the initial Sentinel implementation using stronger and simpler to predict algorithms
- A stable release of Redis Sentinel is shipped since Redis 2.8.
- New developments are performed in the *unstable* branch, and new features sometimes are back ported into the latest stable branch as soon as they are considered to be stable.
- Redis Sentinel version 1, shipped with Redis 2.6, **is deprecated** and **should not be used**.

Running Sentinel



- If you are using the **redis-sentinel** executable you can run Sentinel with the following command line:

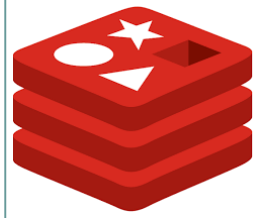
```
$ redis-sentinel /path/to/sentinel.conf
```

- Otherwise you can use directly the redis-server executable starting it in Sentinel mode:

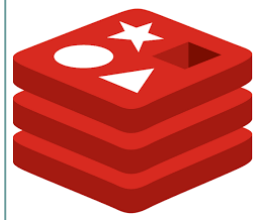
```
$ redis-server /path/to/sentinel.conf --sentinel
```

- Both ways work the same.
- But using a configuration file **it is mandatory**
- Sentinels by default run **listening for connections to TCP port 26379**

Key Points



- You need **at least three** Sentinel instances for a robust deployment.
- The three Sentinel instances **should be placed** into computers or virtual machines that are believed to **fail in an independent way.**
  - So for example different physical servers or Virtual Machines executed on different availability zones.
- Sentinel + Redis distributed system **does not guarantee** that acknowledged writes are retained during failures, since Redis uses **asynchronous replication**.
- You need Sentinel support in your clients.
- There is no HA setup which is safe if you don't test from time to time in development environments, or even better if you can, in production environments, if they work.
  - You may have a misconfiguration that will become apparent only when it's too late (at 3am when your master stops working).
- Sentinel, Docker, or other forms of Network Address Translation or Port Mapping should be mixed with care:
  - Docker performs port remapping, **breaking Sentinel auto discovery** of other Sentinel processes and the list of slaves for a master.

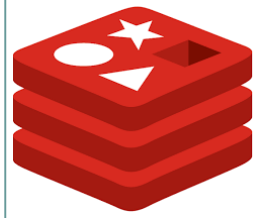


Configuring Sentinel

- The Redis source distribution contains a file called sentinel.conf
- Typical minimal configuration

```
sentinel monitor mymaster 127.0.0.1 6379 2
sentinel down-after-milliseconds mymaster 60000
sentinel failover-timeout mymaster 180000
sentinel parallel-syncs mymaster 1

sentinel monitor resque 192.168.1.3 6380 4
sentinel down-after-milliseconds resque 10000
sentinel failover-timeout resque 180000
sentinel parallel-syncs resque
```

Configuring Sentinel 2

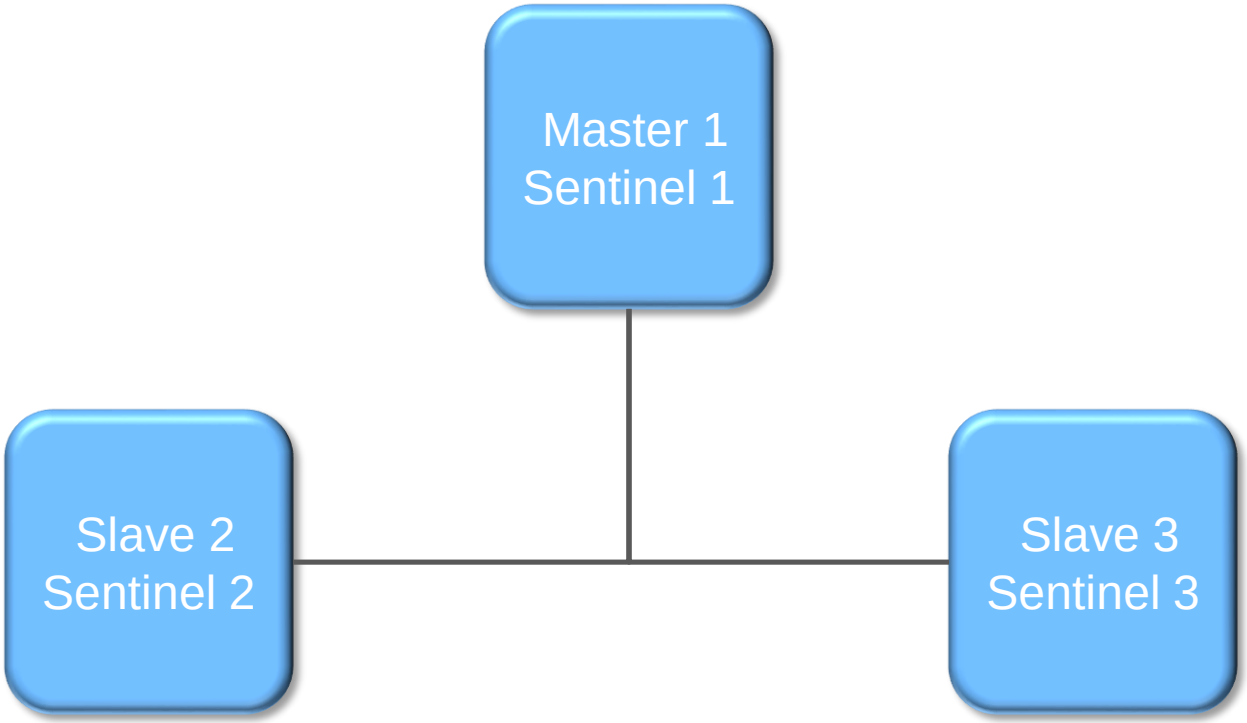


- You only need to specify the masters to monitor, giving to each separated master (that may have any number of slaves) a different name.

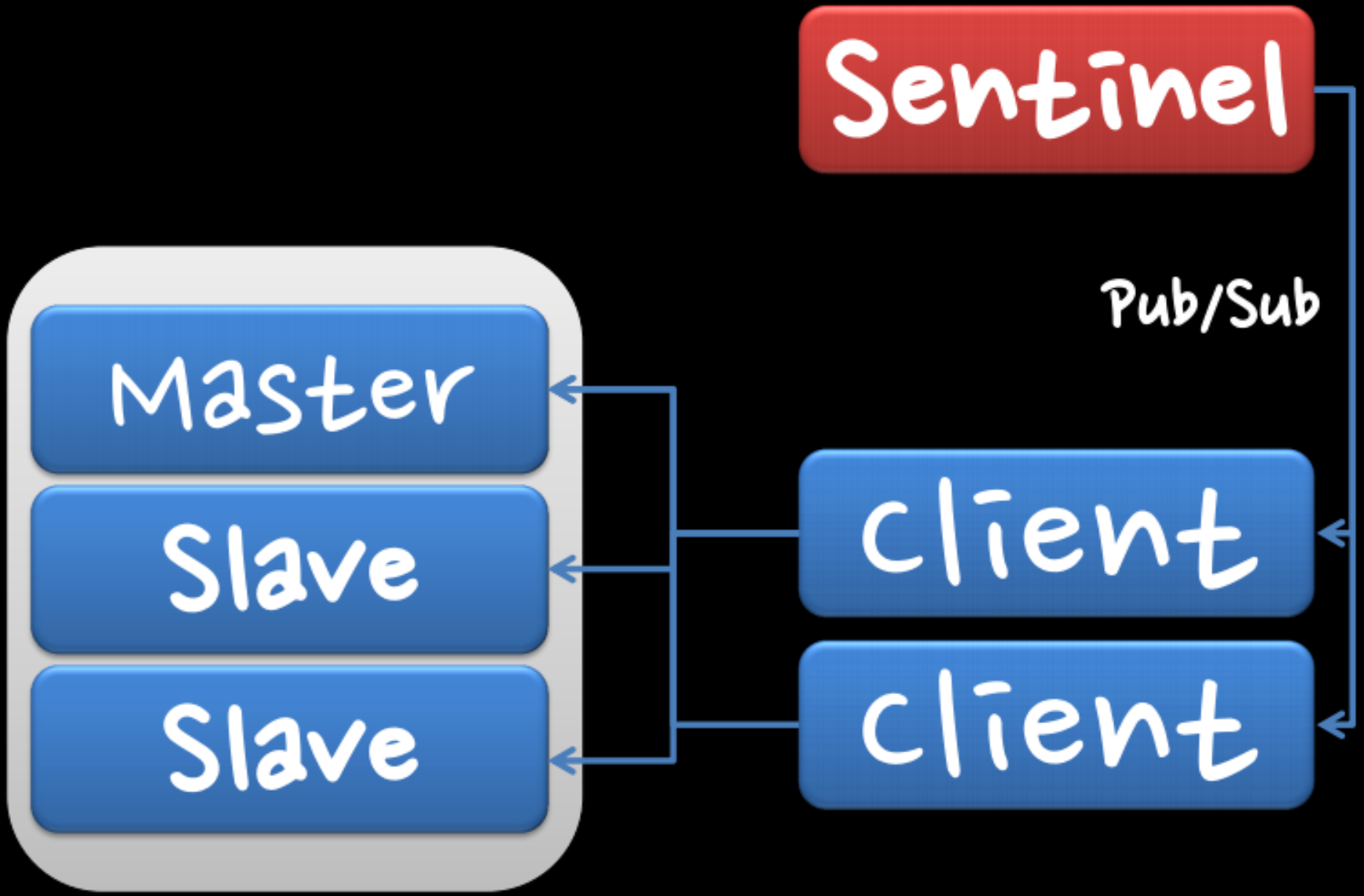
```
sentinel monitor <master-group-name> <ip> <port> <quorum>
sentinel monitor mymaster 127.0.0.1 6379 2
```

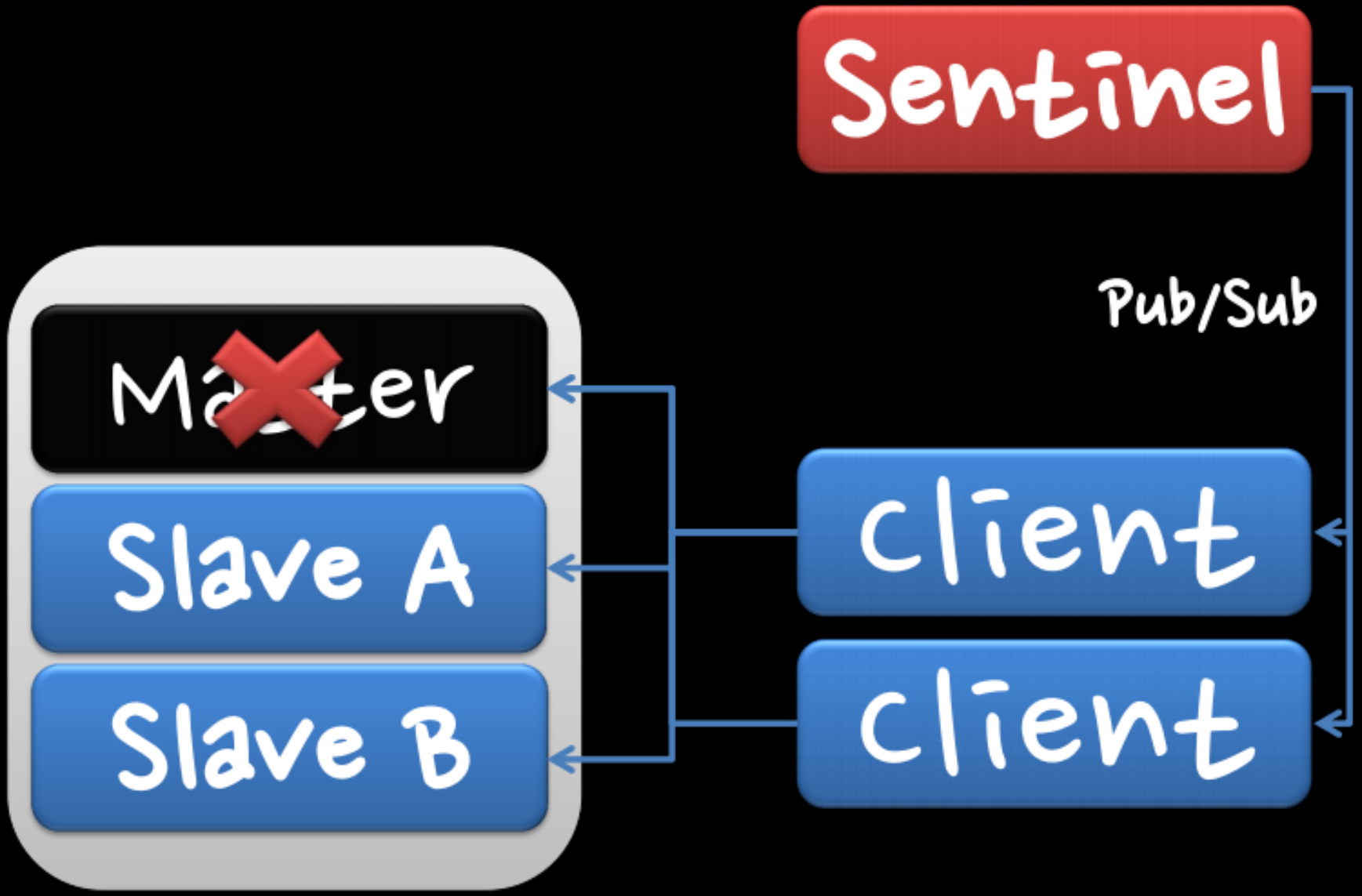
- The **quorum** is the number of Sentinels that need to agree about the fact the master is not reachable, in order for really mark the slave as failing, and eventually start a fail over procedure if possible.
- However **the quorum is only used to detect the failure**.
- In order to actually perform a failover, one of the Sentinels need to be elected leader for the failover and be authorized to proceed.
- This only happens with the vote of the **majority of the Sentinel processes**.

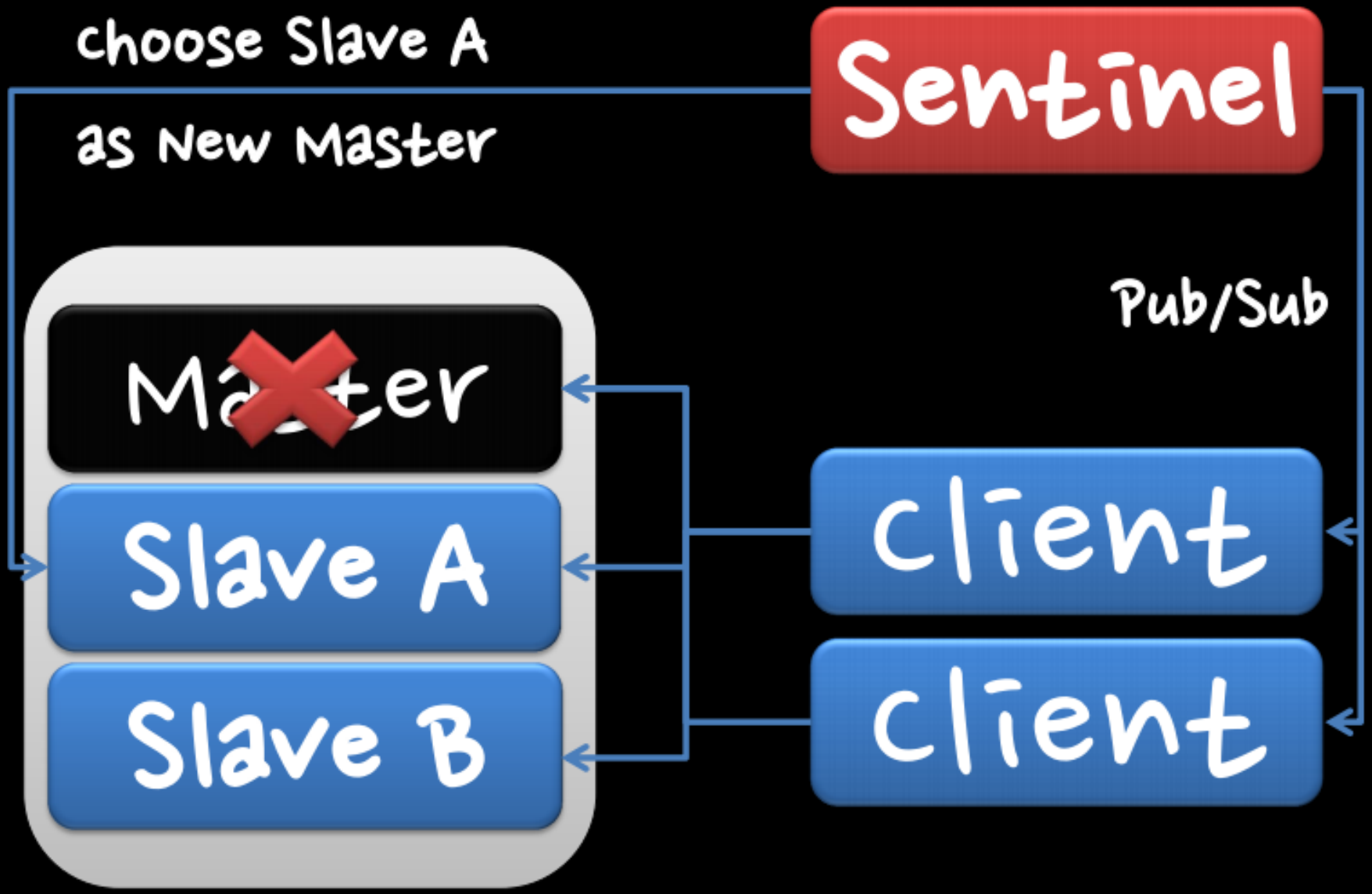
Basic Setup with Three Boxes

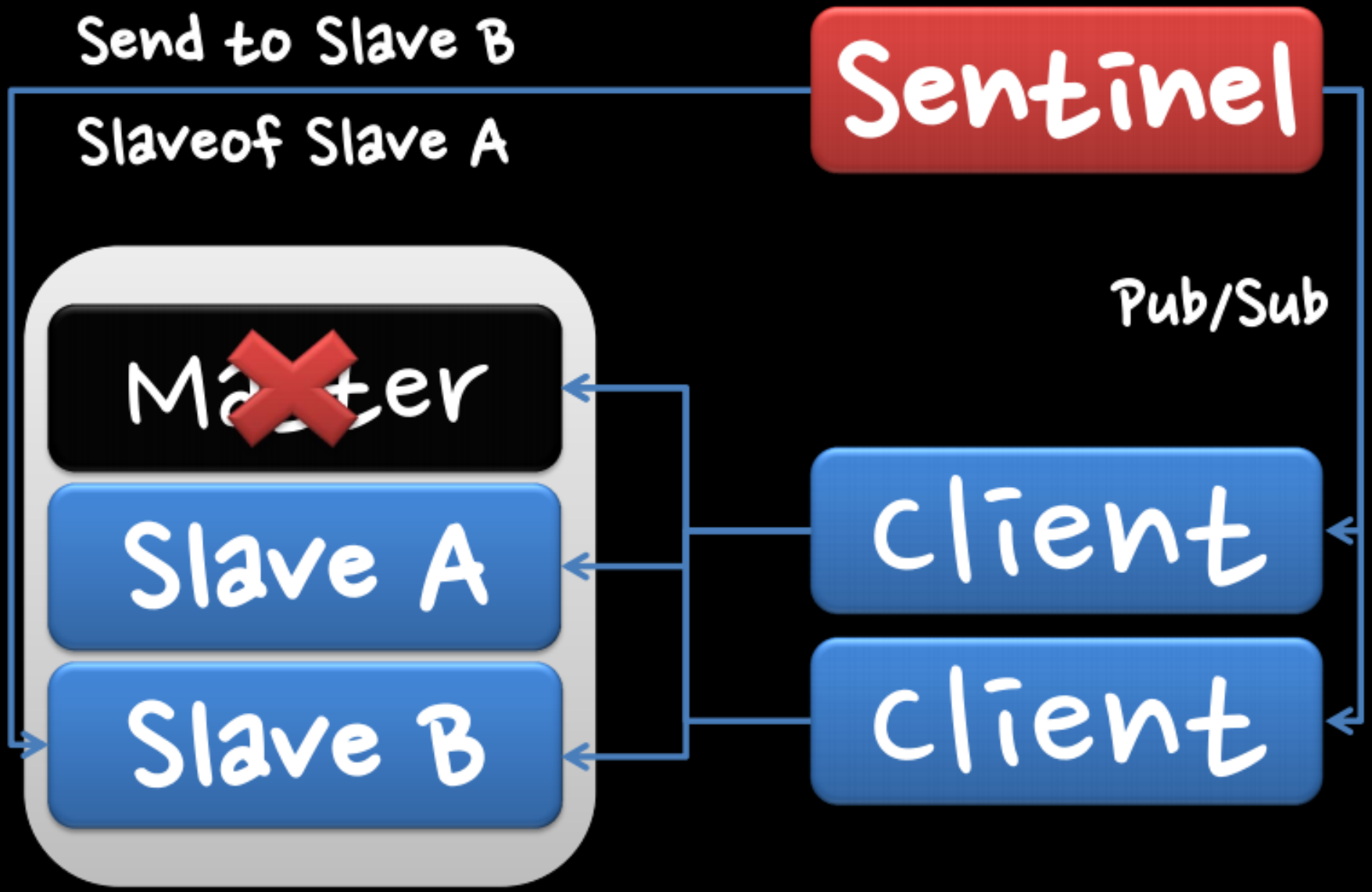


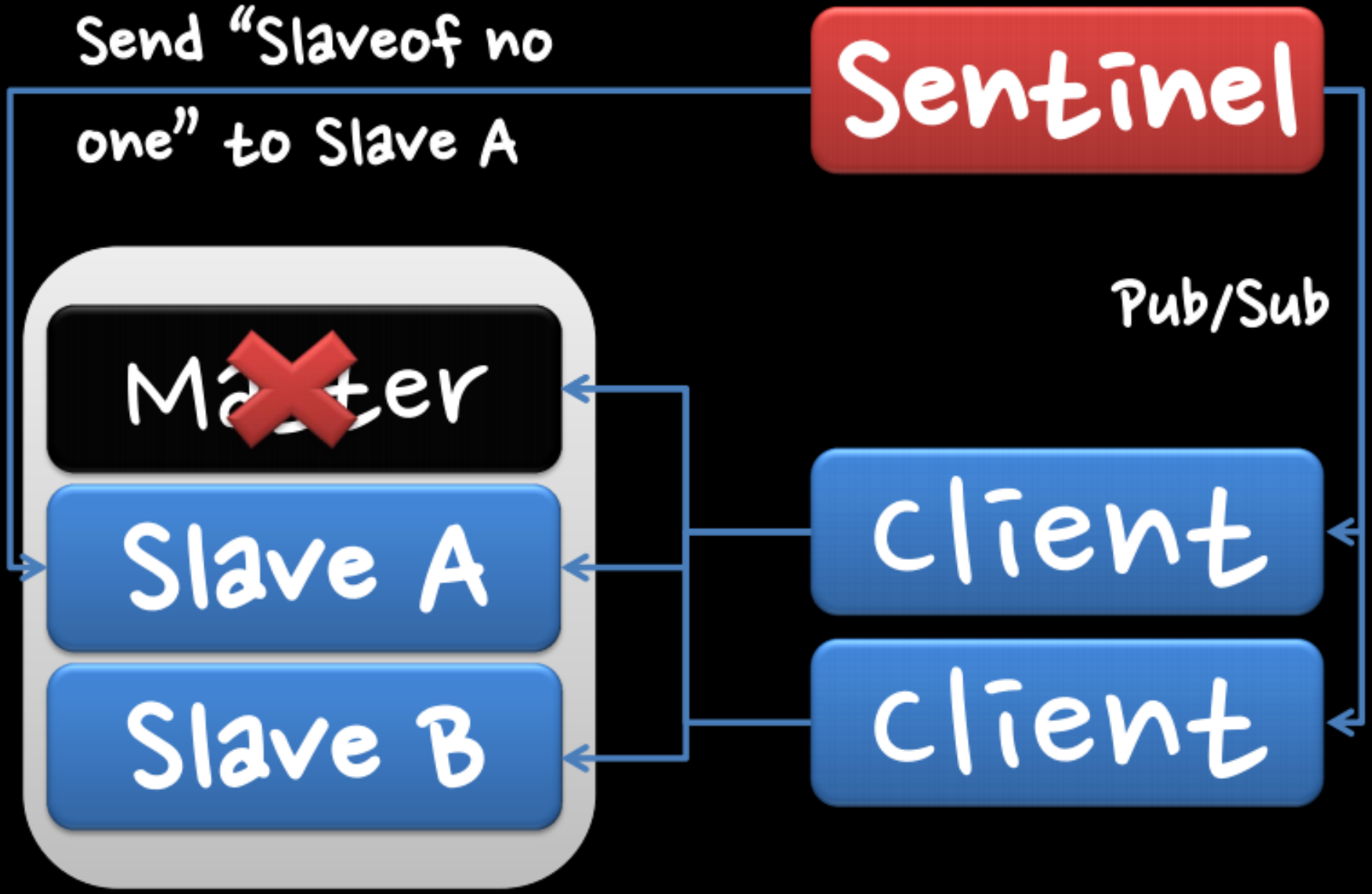
Quarum = 2

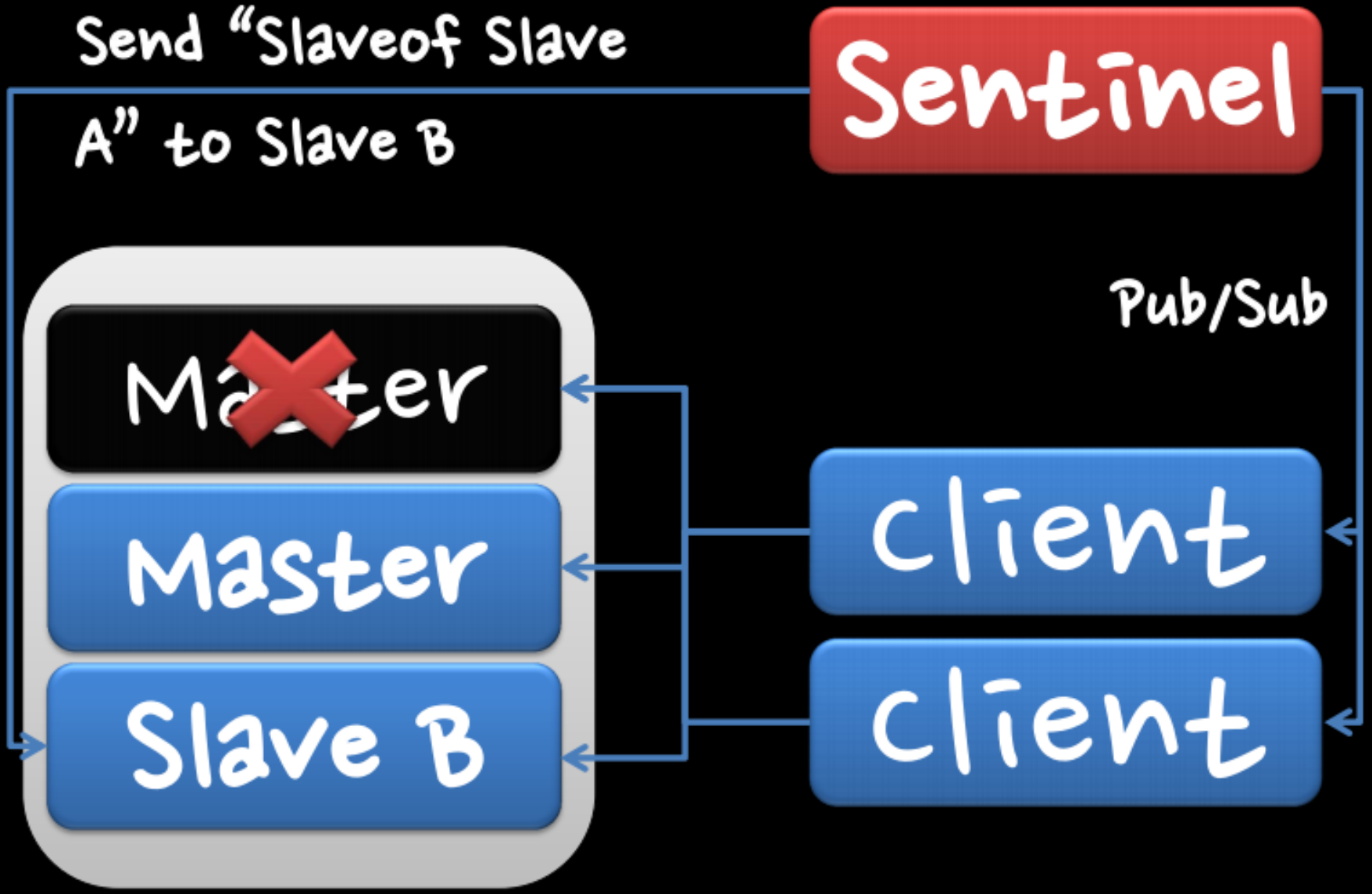

Sentinel
Pub/Sub
Master
Slave
Slave
client
client

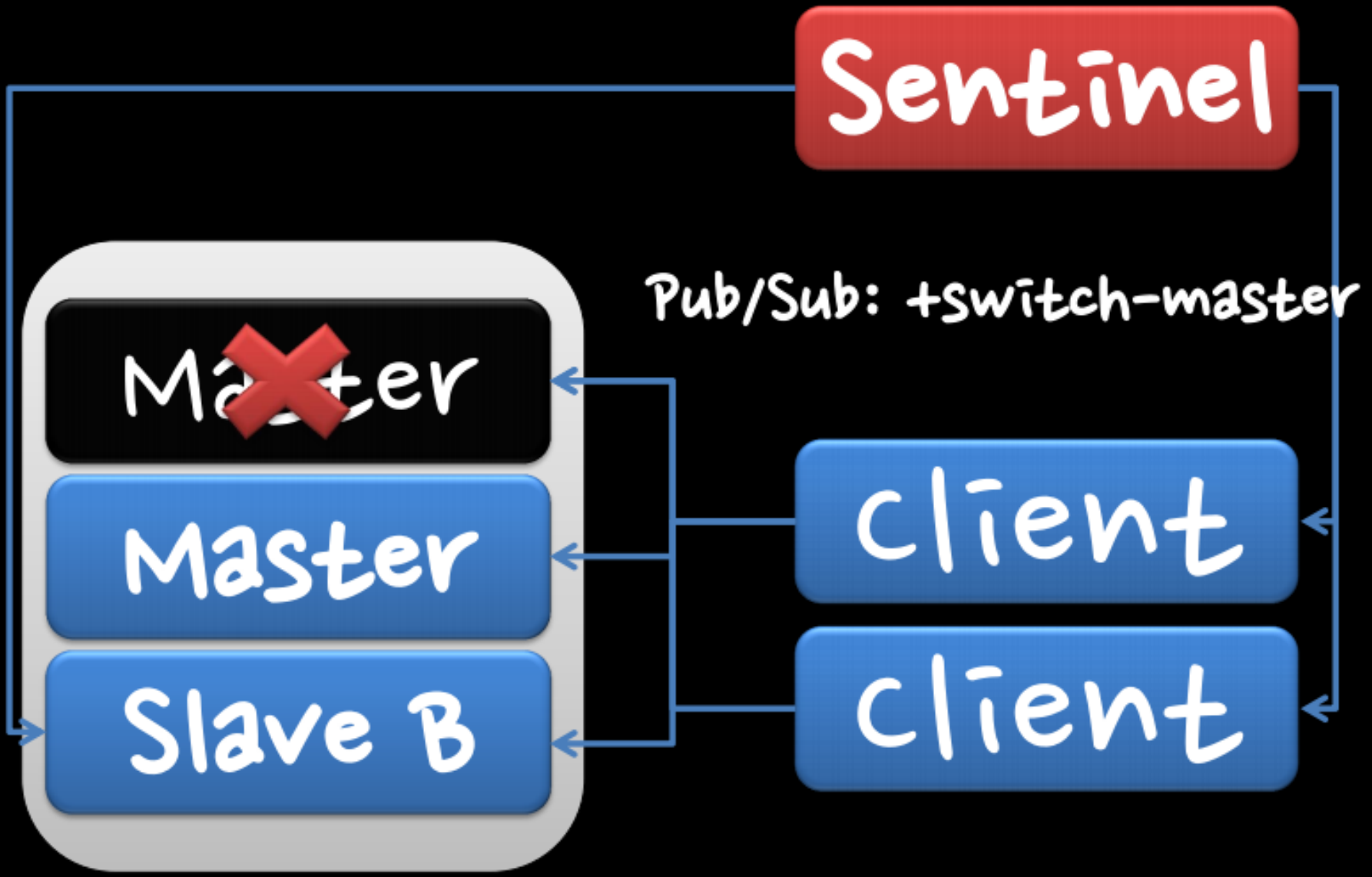

Sentinel
Pub/Sub
Master
Slave A
Slave B
client
client


choose Slave A
as New Master
Sentinel
Pub/Sub
Master
Slave A
Slave B
client
client


Send to Slave B
Slaveof Slave A
Sentinel
Pub/Sub
Master
Slave A
Slave B
client
client


Send "Slaveof no one" to Slave A
Sentinel
Pub/Sub
Master
Slave A
Slave B
client
client


Send "Slaveof Slave A" to Slave B
Sentinel
Pub/Sub
Master
Master
Slave B
client
client

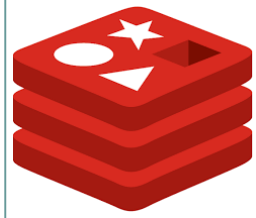

Sentinel
Pub/Sub: +switch-master
Master
Master
Slave B
client
client

redis

Redis Sentinel Demo

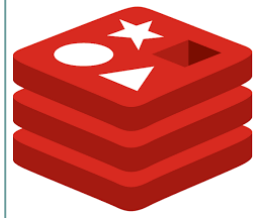
Ali KATKAR

Redis Sentinel
Demo Configuration



- Assume that the instances are executed at port 5000, 5001, 5002.
- Also assume that you have a running Redis master at port 6379 with a slave running at port 6380.

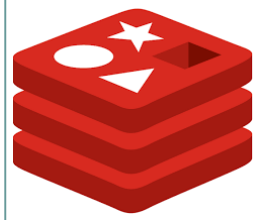
Redis Sentinel
Step1: Create Instance



- Create directories:

```
$ cd /opt
$ sudo mkdir redis-sentinel
$ cd redis-sentinel
$ sudo mkdir scripts
$ sudo mkdir 5000 5001 5002 6380

$ cd 6380
$ sudo cp /etc/redis/redis.conf .
$ sudo gedit redis.conf ➔ change port and add slave
$ cd 5000
$ sudo gedit sentinel.conf
```

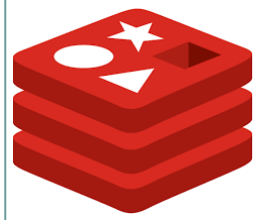
Redis Sentinel
Demo Configuration



- A minimal Redis sentinel configuration file:

```
port 5000
sentinel monitor mymaster 127.0.0.1 6379 2
sentinel down-after-milliseconds mymaster 5000
sentinel failover-timeout mymaster 60000
sentinel parallel-syncs mymaster 1
```

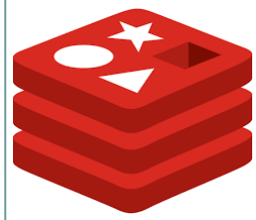
Redis Sentinel
Step 2: Create Scripts



- Create an **instance_start.sh**

```
#!/bin/bash
cd /opt/redis-sentinel/$1

if [ $1 -eq 6380 ]
then
/usr/bin/redis-server redis.conf
redis-cli –p $1
else
/usr/bin/redis-server sentinel.conf --sentinel
fi
```

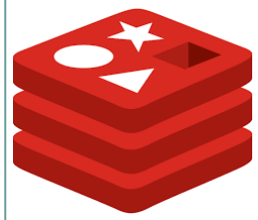
Redis Sentinel
Step 2: Create Scripts



- Create **start_all.sh**

```
#!/bin/bash

x-terminal-emulator –e ./instance_start.sh 6380

for port in {5000..5002}; do
  x-terminal-emulator –e ./instance_start.sh $port
done
```

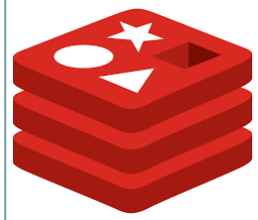
Redis Sentinel
Step3: Run Start Script



```
$ sudo ./start_all.sh
$
$ redis-cli -p 5000
127.0.0.1:5000> sentinel master mymaster
 1) "name"
 2) "mymaster"
 ...
```

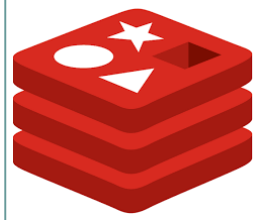
- As you can see, it prints a number of information about the master. There are a few that are of particular interest for us:
  - **num-other-sentinels** is 2, so we know the Sentinel already detected two more Sentinels for this master.
  - **flags** is just master. If the master was down we could expect to see s_down or o_down flag as well here.
  - **num-slaves** is correctly set to 1, so Sentinel also detected that there is an attached slave to our master.

Redis Sentinel
Step 4: Some Commands



```
127.0.0.1:5000> SENTINEL slaves mymaster

127.0.0.1:5000> SENTINEL sentinels mymaster

127.0.0.1:5000> SENTINEL get-master-addr-by-name mymaster
1) "127.0.0.1"
2) "6379"
```

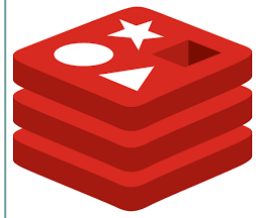
Redis Sentinel
Step 5: Test Failover



```
127.0.0.1:5000> SENTINEL get-master-addr-by-name mymaster
1) "127.0.0.1"
2) "6379"

127.0.0.1:5000> SENTINEL get-master-addr-by-name mymaster
1) "127.0.0.1"
2) "6380"
```

```
$ redis-cli -p 6379 DEBUG sleep 30
```

Redis Sentinel
Step 5: Test Failover



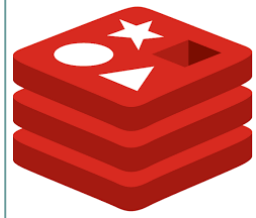
- If you check the Sentinel logs, you should be able to see a lot of action:
  - Each Sentinel detects the master is down with an **+sdown** event.
  - This event is later escalated to **+odown**, which means that multiple Sentinels agree about the fact the master is not reachable.
  - Sentinels vote a Sentinel that will start the first failover attempt.
  - The failover happens.

redis

LRU Caching

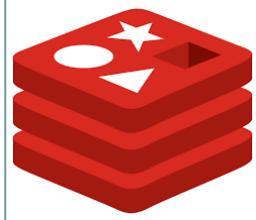
Ali KATKAR

Using Redis as an LRU Cache



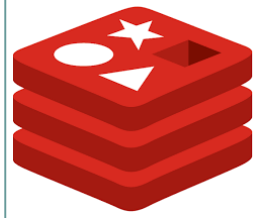
- When Redis is used as a cache, often it is handy to let it **automatically evict** old data as you add new one.
  - This behavior is very well known in the community of developers, since it is the default behavior of the popular memcached system.
- **LRU** (Least Recently Used) is actually only one of the supported eviction methods.
- Starting with Redis version 4.0, a new **LFU** (Least Frequently Used) eviction policy was introduced.

Maxmemory Configuration Directive



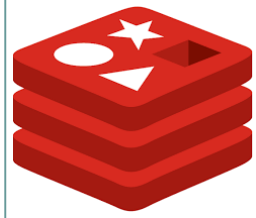
- The maxmemory configuration directive is used in order to configure Redis to use a specified amount of memory for the data set.
  - It is possible to set the configuration directive using the redis.conf file,
  - or later using the CONFIG SET command at runtime.
  - For example in order to configure a memory limit of 100 megabytes
    - **maxmemory 100mb**
- Setting maxmemory to zero results into no memory limits.
  - This is the default behavior for 64 bit systems, while 32 bit systems use an implicit memory limit of 3GB.
- When the specified amount of memory is reached, it is possible to select among different behaviors, called **policies**.
  - Redis can just **return errors** for commands that could result in more memory being used,
  - or it can **evict some old data** in order to return back to the specified limit every time new data is added.

Eviction Policies



- **noeviction:** return errors when the memory limit was reached
- **allkeys-lru:** evict keys by trying to remove LRU keys first
- **volatile-lru:** evict keys by trying to remove LRU keys first, but only among keys that have an expire set
- **allkeys-random:** evict keys randomly in order to make space for the new data added.
- **volatile-random:** evict keys randomly in order to make space for the new data added, but only evict keys with an expire set.
- **volatile-ttl:** evict keys with an expire set, and try to evict keys with a shorter time to live (TTL) first

LFU Mode



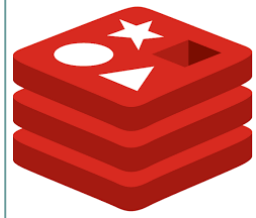
- Starting with Redis 4.0, a new Least Frequently Used eviction mode is available.
- This mode may work better in certain cases
  - LFU Redis will try to track the frequency of access of items
  - The ones used rarely are evicted
  - The ones used often have an higher chance of remaining in memory.
- If you think at LRU, an item that was recently accessed but is actually almost never requested, will not get expired, so the risk is to evict a key that has an higher chance to be requested in the future.
- LFU does not have this problem, and in general should adapt better to different access patterns.
- To configure the LFU mode, the following policies are available:
  - **volatile-lfu:** Evict using approximated LFU among the keys with an expire set.
  - **allkeys-lfu:** Evict any key using approximated LFU.

redis

Backup Security

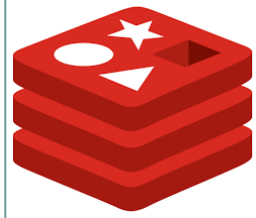
Ali KATKAR

Backup



- Redis **SAVE** command is used to create a backup of the current Redis database.
- Syntax
  - 127.0.0.1:6379> SAVE
  - This command will create a dump.rdb file in your Redis directory
- To restore Redis data,
  - Move Redis backup file (dump.rdb) into your Redis directory and start the server.
  - To get your Redis directory, use CONFIG command of Redis as shown below
    - 127.0.0.1:6379> config get dir
- To create Redis backup, an alternate command **BGSAVE** is also available.
  - This command will start the backup process and run this in the background.

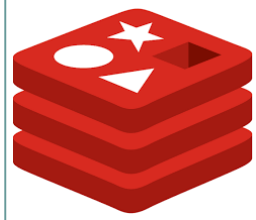
Security



- Redis database can be secured
  - Any client making a connection needs to authenticate before executing a command.
- To secure Redis, you need to set the password in the config file.

```
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) ""
127.0.0.1:6379> CONFIG set requirepass "redisPassword"
OK
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) "redisPassword"
```

Security:
Using Password



- After setting the password, if any client runs the command without authentication, then **(error) NOAUTH Authentication required**. error will return.
- The client needs to use AUTH command to authenticate himself.

```
127.0.0.1:6379> AUTH "redisPassword"
OK
127.0.0.1:6379> SET mykey "Test value"
OK
127.0.0.1:6379> GET mykey
"Test
```

redis

Benchmarking

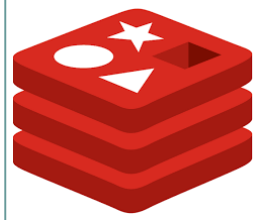
Ali KATKAR

Benchmark

- Redis benchmark is the utility to check the performance of Redis by running n commands simultaneously
- Syntax
  - `redis-benchmark [option] [option value]`
- For example by calling 100000 commands.

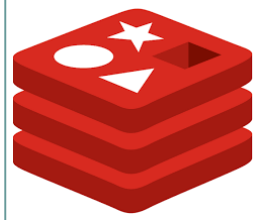
```
$ redis-benchmark -n 100000

PING_INLINE: 141043.72 requests per second
PING_BULK: 142857.14 requests per second
SET: 141442.72 requests per second
GET: 145348.83 requests per second
```

Benchmark Options



Option	Description	Default Value
-h	Specifies server host name	127.0.0.1
-p	Specifies server port	6379
-s	Specifies server socket	
-c	Specifies the number of parallel connections	50
-n	Specifies the total number of requests	10000
-d	Specifies data size of SET/GET value in bytes	2
-k	1=keep alive, 0=reconnect	1
-r	Use random keys for SET/GET/INCR, random values for SADD	
-p	Pipeline <numreq> requests	1
-h	Specifies server host name	
-q	Forces Quiet to Redis. Just shows query/sec values	
--csv	Output in CSV format	
-l	Generates loop, Run the tests forever	
-t	Only runs the comma-separated list of tests	
-I	Idle mode. Just opens N idle connections and wait	

Benchmark Example



- Multiple usage options in Redis benchmark utility.

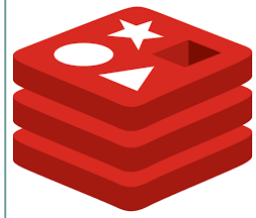
```
$ redis-benchmark -h 127.0.0.1 -p 6379 -t set,lpush -n 100000 -q

SET: 146198.83 requests per second
LPUSH: 145560.41 requests per second
```

redis

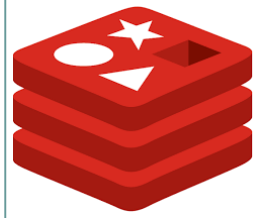
Monitoring

Ali KATKAR

Monitor Redis: metrics and alerts



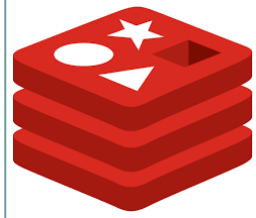
- You need to monitor some broad conditions:
  - Required processes are running as expected
  - System resources usage is within limits
  - Queries are executed successfully
  - Service is performing properly
  - Typical failure points



Redis process running

- These alerts will let us know if something basic is not in place,
  - like a daemon not running or respawning all the time.

Metric	Comments	Suggested Alert
redis process	Right binary daemon process running.	When process /usr/sbin/redis count != 1.
uptime	We want to make sure the service is not respawning all the time.	When uptime < 300s.

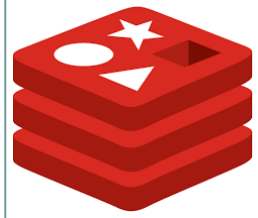


System Metrics

- The metrics listed below are the “usual suspects” behind most issues and bottlenecks.
- They also correspond to the top system resources you should monitor on pretty much any in-memory DB server.

Metric	Comments	Suggested Alert
Load	An all-in-one performance metric. A high load will lead to performance degradation.	When load is > factor x (number of cores). Our suggested factor is 4.
CPU usage	High CPU usage is not a bad thing as long as you don’t reach the limit.	None
Memory usage	RAM usage depends on how many keys and values we keep in memory. Redis should fit in memory with plenty of room to spare for the OS.	None
Swap usage	Swap is for emergencies only. Don’t swap. A bit of RAM is always in use, but if that grows, it’s an indicator for performance degradation.	When used swap is > 128MB.
Network bandwidth	Traffic is related to the number of connections and the size of those requests. Used for for troubleshooting but not for alerting.	None
Disk usage	Make sure you always have free space for new data, logs, temporary files, snapshot or backups.	When disk is > 85% usage.

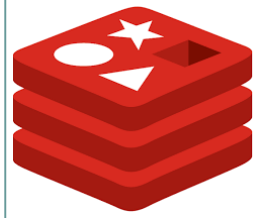
Monitoring Redis Availability and Queries



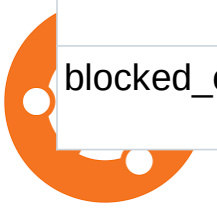
- These metrics will inform you if Redis is working as expected.

Metric	Comments	Suggested Alert
connected_clients	Number of clients connected to Redis. Typically your application nodes rather than final users.	When connected_clients < minimum number of application/consumers on your stack.
keyspace	Total number of keys in your database. Useful when compared to hit_rate in order to help troubleshoot any misses.	None
instantaneous_ops_per_sec	Number of commands processed per second.	None
hit rate (calculated)	keyspace_hits / (keyspace_hits + keyspace_misses)	None
rdb_last_save_time	Unix timestamp for last save to disk, when using persistence.	When rdb_last_save_time is > 3600 seconds (or your acceptable timeframe)
rdb_changes_since_last_save	Number of changes to the database since last dump. Data that you would lose upon restart.	None
connected_slaves	Number of slaves connected to this master instance	When connected_slaves != from the number of slaves in your cluster.
master_last_io_seconds_ago	Seconds since last interaction between slave and master	When master last_io_seconds_ago is > 30 seconds (or your acceptable timeframe)

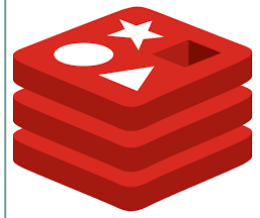
Monitoring Redis Performance



Metric	Comments	Suggested Alert
latency	Average time it takes Redis to respond to a query.	When latency is > 200ms (or your max acceptable).
used_memory	Memory used by the Redis server. If it exceeds physical memory, system will start swapping causing severe performance degradation. You can configure a limit with Redis maxmemory configuration setting for cache scenarios (you don't want to evict keys on database or queues scenarios!)	None
mem_fragmentation_ratio	Compares Redis memory usage to Linux virtual memory pages (mapped to physical memory chunks). A high ratio will lead to swapping and performance degradation.	When mem_fragmentation_ratio is > 1.5
evicted_keys	Number of keys removed (evicted) due to reaching maxmemory. Too many evicted keys means that new requests need to wait for an empty space before being stored in memory. When that happens, latency will increase.	None, but when using TTL for expiring keys and you don't expect evictions you could configure when evicted_keys is > 0.
blocked_clients	Number of clients waiting on a blocking call (BLPOP, BRPOP, BRPOPLPUSH).	None

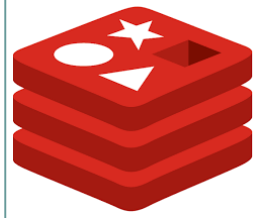


Monitoring Redis Errors



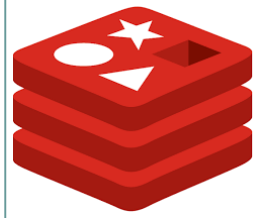
Metric	Comments	Suggested Alert
rejected_connections	Number of connections rejected due to hitting maxclient limit (remember to control max/used OS file descriptors)	When rejected_connections > XX, depending on the number of clients you might have.
keyspace_misses	Number of failed lookups of keys	Only when not using blocking calls (BRPOPLPUSH, BRPOP and BLPOP), when keyspace_misses > 0.
master_link_down_since_seconds	Time (in seconds) that the link between master and slave was down. When a new reconnect happens, the slave will send SYNC commands which will impact master performance.	When master_link_down_since_seconds is > 60 seconds.

Redis Monitoring Tools



- redis-cli info command
- redis-cli monitor command
- redis-stat
- redmon
- RedisLive
- redis-faina
- collectd
- Percona Monitoring Plugins
- Nagios
- Server Density

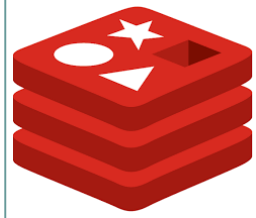
Redis-Cli info



- providing the most important information and statistics about the Redis server.
- As the output is pretty long, it has been divided in several sections:
  - **server**: General information about the Redis server
  - **clients**: Client connections information
  - **memory**: Memory usage information
  - **persistence**: Persistence (RDB and AOF) related information
  - **stats**: General statistics
  - **replication**: Master/slave replication information
  - **cpu**: CPU usage statistics
  - **commandstats**: Redis commands statistics
  - **cluster**: Redis Cluster information (if enabled)
  - **keyspace**: Database (key expiration) related statistics

```
127.0.0.1:6379> INFO
127.0.0.1:6379> INFO clients
```

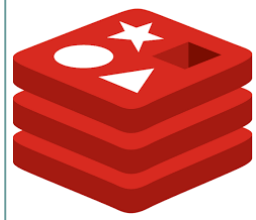
redis-cli monitor



- This one displays every command processed by the Redis server.
- It is used either for spotting bugs in an application, or for generic troubleshooting.
- It helps us understand what is happening to the database.
- It seriously affects performance, however, so it’s not something you want to run all the time.

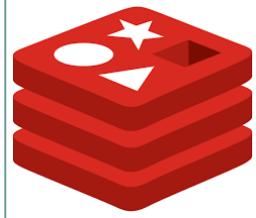
```
127.0.0.1:6379> monitor
Ok
```

Why Third Party Tool Needs?



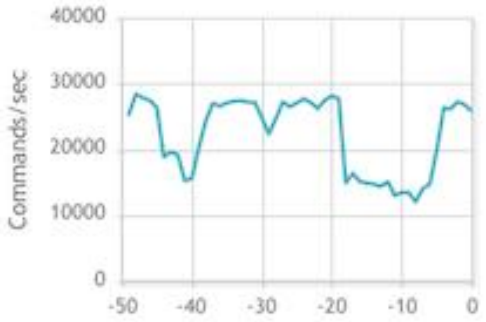
- While redis-cli info is a great interactive / realtime tool, you still need to set up some alerts so that you get notified when things go wrong.
- You may also want to record various metrics over time in order to identify trends.
- There is a list of available options for doing that

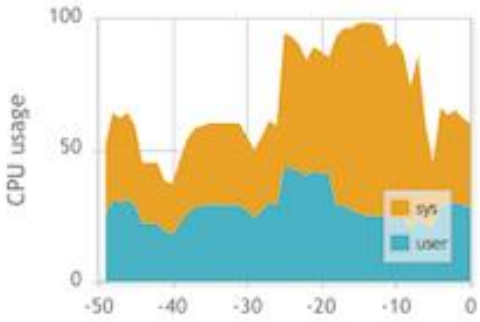
redis-stat

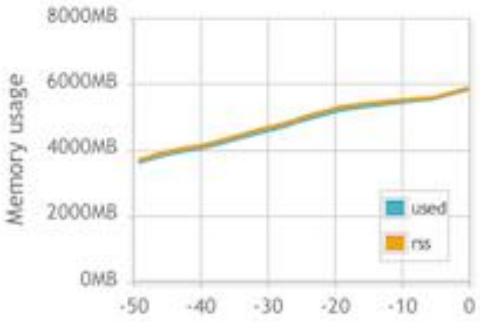


- redis-stat is a simple Redis monitoring tool written in Ruby.
- It tracks Redis performance in a terminal output in a vmstat like format or as a web based dashboard.
- It is based on the INFO command
  - which means **it shouldn't impact the performance** of the Redis server.
- redis-stat shows CPU and memory usage, commands, cache hits and misses, expires and evictions amongst other metrics.

Dashboard





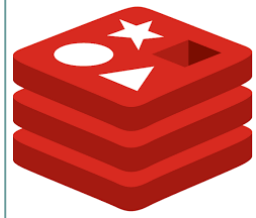


time	us	sy	cl	bcl	mem	rss	keys	cmd/s	exp/s	evt/s	hit/s	mis/s	aofcs	psch
07:16:33	28	32	81	0	5.69GB	5.73GB	2.57M	25.8k	0	0	2.83k	11.5k	0B	0
07:16:31	29	33	81	0	5.64GB	5.68GB	2.54M	26.8k	0	0	2.87k	11.9k	0B	0
07:16:29	30	35	81	0	5.59GB	5.62GB	2.52M	27.2k	0	0	2.90k	12.2k	0B	0
07:16:27	29	34	81	0	5.53GB	5.56GB	2.49M	26.2k	0	0	2.75k	11.7k	0B	0
07:16:25	29	37	81	0	5.48GB	5.51GB	2.47M	26.3k	0	0	2.80k	11.7k	0B	0
07:16:23	21	24	81	0	5.43GB	5.45GB	2.45M	19.9k	0	0	2.20k	8.83k	0B	0
07:16:21	21	39	81	0	5.39GB	5.42GB	2.43M	14.7k	0	0	1.68k	6.53k	0B	0
07:16:19	24	62	81	0	5.38GB	5.42GB	2.42M	14.1k	0	0	1.57k	6.25k	0B	0
07:16:17	20	54	81	0	5.35GB	5.40GB	2.40M	12.1k	0	0	1.51k	5.28k	0B	0
07:16:14	24	63	61	0	5.33GB	5.39GB	2.39M	13.4k	0	0	1.88k	5.76k	0B	0

Instance information

	127.0.0.1:6379
redis_version	2.4.17
process_id	22425
uptime_in_seconds	354
uptime_in_days	0
gcc_version	4.2.1
role	master
connected_slaves	0
aof_enabled	0
vm_enabled	0

redmon

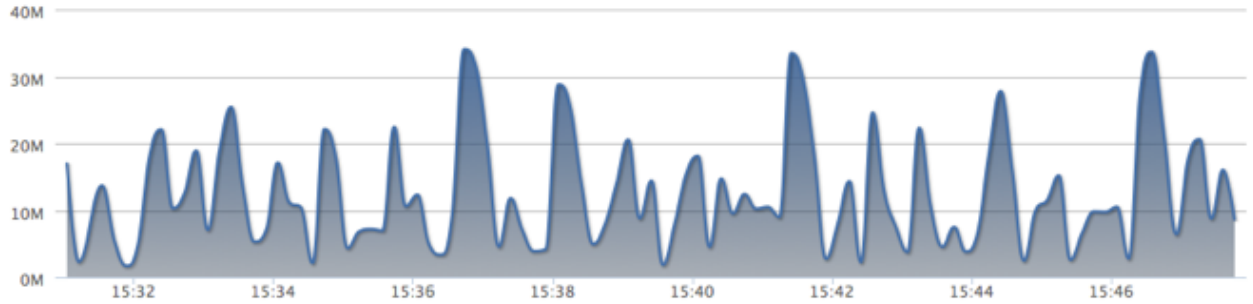


- Redmon is a simple Ruby dashboard based on the Sinatra framework.
- While its monitoring function is not as complete as redis-stats, Redmon comes with invaluable management features like server configuration and access to the CLI interface through the web UI.

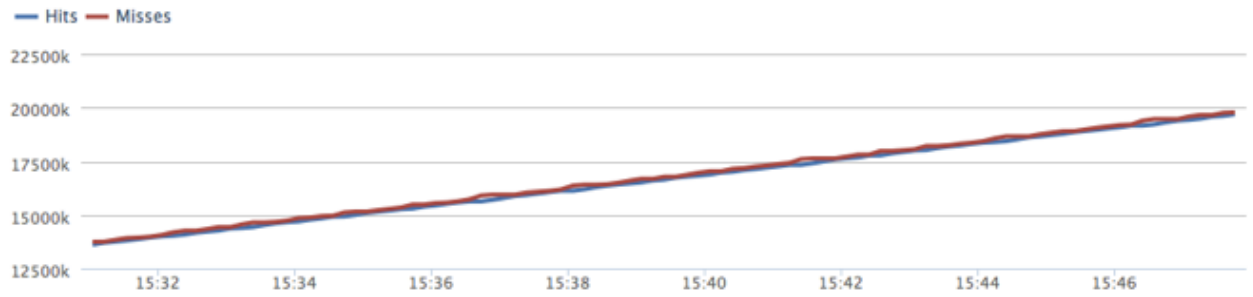
Redmon | Dashboard | Keys | CLI | Configuration | redis://127.0.0.1:6379 | Flush DB | Reset Statistics

Dashboard

Memory Usage



Keyspace



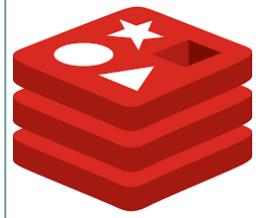
Slow Log

flushdb	97.088 ms	1/18 23:26:31
flushdb	72.648 ms	1/21 14:53:21
flushdb	60.193 ms	1/18 22:30:23
flushdb	28.113 ms	1/20 23:15:12
set key-338110 abcdedghijklmnopqrstuvwxyz	10.709 ms	1/21 14:53:32
set key-338196 abcdedghijklmnopqrstuvwxyz	10.624 ms	1/21 14:53:32

Info

Version	2.2.12
Role	master
Uptime Days	2
Memory Used	8.09M
Memory Peak	
Memory Fragmentation Ratio	8.28
DB Size (Keys)	44,171
Keyspace Hits	19,698,953
Keyspace Misses	19,783,444
Pub/Sub Channels	0
Pub/Sub Patterns	0
Total Connections Received	323
Total Commands Processed	59,248,945
Blocked Clients	0
Connected Slaves	0
Last Save	1/16 3:39:38
Changes Since Last Save	532,529
AOF Enabled	0
VM Enabled	0

RedisLive



- Redis Live is a monitoring dashboard written in Python and Tornado.
- It comes with a number of useful widgets, memory and various commands.
- It also shows the top used Redis commands and keys.
- Redis Live uses a database backend to store metrics over time, sqlite being the default choice.

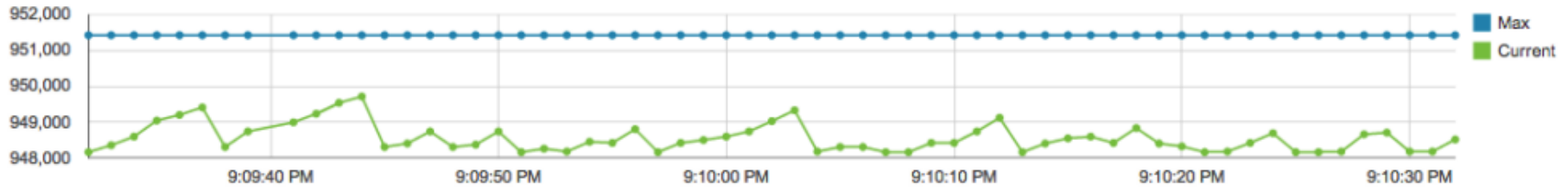
Redis Live

127.0.0.1:6379

USED MEMORY	TOTAL KEYS	CLIENTS	COMMANDS PROCESSED	UPTIME
926.17K	2	3	39.2K	3.7m

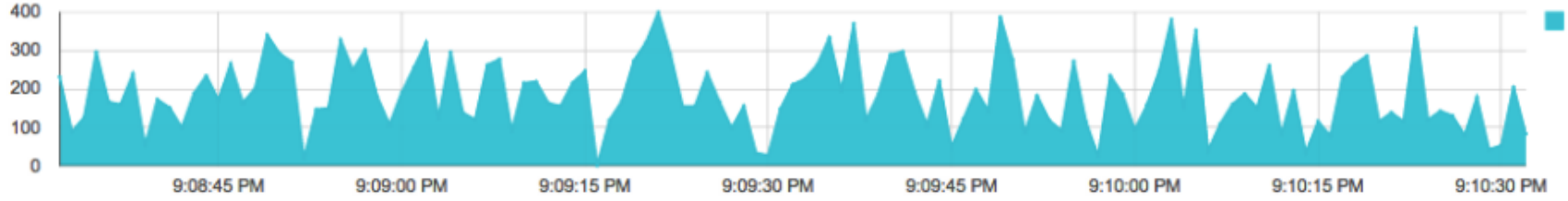
Memory Consumption

realtime



Commands Processed

realtime

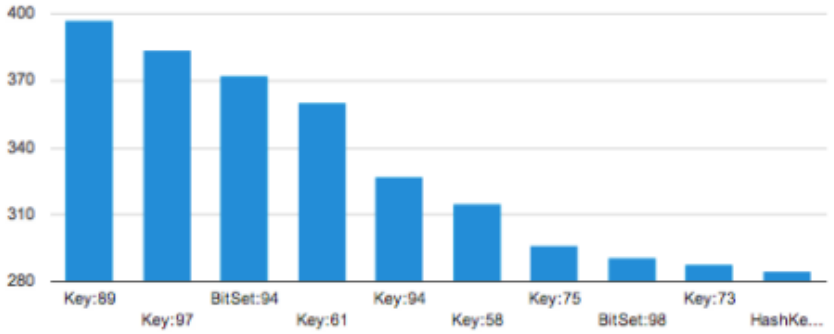


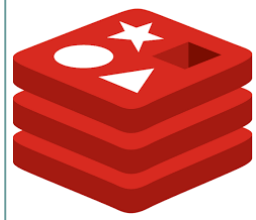
Top Commands

realtime

Top Keys

realtime





Resources

- https://redis.io/topics/quickstart
- https://redis.io/topics/config
- https://redis.io/topics/persistence
- https://redis.io/topics/replication
- https://redis.io/topics/cluster-tutorial
- https://redis.io/topics/sentinel
- https://blog.serverdensity.com/monitor-redis/