**Redis Sentinel Documentation**

Redis Sentinel provides high availability for Redis. In practical terms this means that using Sentinel you can create a Redis deployment that resists without human intervention to certain kind of failures.

Redis Sentinel also provides other collateral（并行的） tasks such as monitoring, notifications（通知） and acts as a configuration provider for clients.

This is the full list of Sentinel capabilities at a macroscopical（宏观的） level (i.e. the *big picture*):

* **Monitoring**. Sentinel constantly（不断地） 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 of Sentinel

Redis Sentinel is a distributed（分布式） 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 cooperating are the following:

1. 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（主观错误）.
2. 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（单点） of failure, after all.

The sum of Sentinels, Redis instances (masters and slaves) and clients connecting to Sentinel and Redis, are also a larger distributed system with specific properties（特征）. In this document concepts will be introduced gradually starting from basic information needed in order to understand the basic properties of Sentinel, to more complex information (that are optional) in order to understand how exactly Sentinel works.

**Quick Start**

Obtaining Sentinel

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（算法） (that are explained in this documentation).

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 (or if you have a symbolic link（软连接） with that name to the redis-server 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.

However **it is mandatory（强制的）** to use a configuration file when running Sentinel, as this file will be used by the system in order to save the current state（保存当前的状态） that will be reloaded in case of restarts. Sentinel will simply refuse to start if no configuration file is given or if the configuration file path is not writable.

Sentinels by default run **listening for connections to TCP port 26379**, so for Sentinels to work, port 26379 of your servers **must be open** to receive connections from the IP addresses of the other Sentinel instances. Otherwise Sentinels can't talk and can't agree about what to do, so failover will never be performed.

Fundamental things to know about Sentinel before deploying（部署）

1. You need at least three Sentinel instances for a robust deployment.
2. 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.
3. Sentinel + Redis distributed system does not guarantee that acknowledged writes are retained during failures, since Redis uses asynchronous（异步的） replication. However there are ways to deploy Sentinel that make the window to lose writes limited to certain moments, while there are other less secure ways to deploy it.
4. You need Sentinel support in your clients. Popular client libraries have Sentinel support, but not all.
5. There is no HA setup（HA设置） 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).
6. **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. Check the section about Sentinel and Docker later in this document for more information.

Configuring Sentinel

The Redis source distribution contains a file called sentinel.conf that is a self-documented example configuration file you can use to configure Sentinel, however a typical minimal configuration file looks like the following:

sentinel monitor mymaster 127.0.0.1 6379 2

该配置的语法为：sentinel monitor <master-name> <ip> <redis-port> <quorum>表示告诉sentinel监视redis的master，master端ip和port；2表示当两台sentinel都认为master down时才客观的认为master down了。

sentinel down-after-milliseconds mymaster 60000

语法：sentinel down-after-milliseconds <master-name> <milliseconds>，表示当sentinel 60000毫秒间隔都没有连接到master则主观的认为master已经down了

sentinel failover-timeout mymaster 180000

sentinel failover-timeout <master-name> <milliseconds>，表示当一个master down后，多久其他的slave被配置成新的master的slaves

sentinel parallel-syncs mymaster 1

sentinel parallel-syncs <master-name> <numslaves>，表示在failover阶段可以立即配置多少的slaves执行新的slave

sentinel monitor resque 192.168.1.3 6380 4

sentinel down-after-milliseconds resque 10000

sentinel failover-timeout resque 180000

sentinel parallel-syncs resque 5

You only need to specify the masters to monitor, giving to each separated master (that may have any number of slaves) a different name. There is no need to specify slaves, which are auto-discovered（自动被发现）. Sentinel will update the configuration automatically（自定更新） with additional information about slaves (in order to retain the information in case of restart). The configuration is also rewritten every time a slave is promoted to master during a failover and every time a new Sentinel is discovered.

The example configuration above, basically monitor two sets of Redis instances, each composed（组成了） of a master and an undefined number of slaves（未定义数量的slaves）. One set of instances is called mymaster, and the other resque.

The meaning of the arguments of sentinel monitor statements is the following:

sentinel monitor <master-group-name> <ip> <port> <quorum>

For the sake of clarity（为了清晰）, let's check line by line what the configuration options mean:

The first line is used to tell Redis to monitor a master called *mymaster*, that is at address 127.0.0.1 and port 6379, with a quorum of 2. Everything is pretty obvious but the **quorum** argument:

* 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**.

So for example if you have 5 Sentinel processes, and the quorum for a given master set to the value of 2, this is what happens:

* If two Sentinels agree at the same time about the master being unreachable, one of the two will try to start a failover.
* If there are at least a total of three Sentinels reachable, the failover will be authorized and will actually start.

In practical terms this means during failures **Sentinel never starts a failover if the majority of Sentinel processes are unable to talk** (aka no failover in the minority partition).

Other Sentinel options

The other options are almost always in the form:

sentinel <option\_name> <master\_name> <option\_value>

And are used for the following purposes:

* down-after-milliseconds is the time in milliseconds an instance should not be reachable (either does not reply to our PINGs or it is replying with an error) for a Sentinel starting to think it is down（就是说这个sentinel即监视主也监视从）.
* parallel-syncs sets the number of slaves that can be reconfigured to use the new master after a failover at the same time. The lower the number, the more time it will take for the failover process to complete, however if the slaves are configured to serve old data, you may not want all the slaves to re-synchronize with the master at the same time. While the replication process is mostly non blocking（非阻塞） for a slave, there is a moment when it stops to load the bulk data from the master. You may want to make sure only one slave at a time is not reachable by setting this option to the value of 1.

Additional options are described in the rest of this document and documented in the example sentinel.conf file shipped with the Redis distribution.

All the configuration parameters can be modified at runtime using the SENTINEL SET command. See the **Reconfiguring Sentinel at runtime** section for more information.

Example Sentinel deployments

Now that you know the basic information about Sentinel, you may wonder where you should place your Sentinel processes, how much Sentinel processes you need and so forth. This section shows a few example deployments.

We use ASCII art（简单来说既是ASCII字符创作绘画） in order to show you configuration examples in a *graphical* format, this is what the different symbols means:

+--------------------+

| This is a computer |

| or VM that fails |

| independently. We|

| call it a "box" |

+---------------------------+

We write inside the boxes what they are running:

+-----------------------------+

| Redis master M1 |

| Redis Sentinel S1 |

+-----------------------------+

Different boxes are connected by lines, to show that they are able to talk:

+-------------+ +-------------+

| Sentinel S1 |----------------------| Sentinel S2 |

+-------------+ +-------------+

Network partitions are shown as interrupted lines using slashes:

+-------------+ +-------------+

| Sentinel S1 |--------- // ----------| Sentinel S2 |

+-------------+ +-------------+

Also note that:

* Masters are called M1, M2, M3, ..., Mn.
* Slaves are called R1, R2, R3, ..., Rn (R stands for *replica*).
* Sentinels are called S1, S2, S3, ..., Sn.
* Clients are called C1, C2, C3, ..., Cn.
* When an instance changes role because of Sentinel actions, we put it inside square brackets, so [M1] means an instance that is now a master because of Sentinel intervention（处理）.

Note that we will never show **setups where just two Sentinels are used**, since Sentinels always need **to talk with the majority** in order to start a failover.

Example 1: just two Sentinels, DON'T DO THIS

+----+ +----+

| M1 |-------------| R1 |

| S1 | | S2 |

+----+ +----+

Configuration: quorum = 1

* In this setup, if the master M1 fails, R1 will be promoted since the two Sentinels can reach agreement about the failure (obviously with quorum set to 1) and can also authorize a failover because the majority is two. So apparently it could superficially work, however check the next points to see why this setup is broken.
* If the box where M1 is running stops working, also S1 stops working. The Sentinel running in the other box S2 will not be able to authorize a failover, so the system will become not available.

Note that a majority is needed in order to order different failovers, and later propagate（繁殖） the latest configuration to all the Sentinels. Also note that the ability to failover in a single side of the above setup, without any agreement, would be very dangerous:

+----+ +------+

| M1 |----//-----| [M1] |

| S1 | | S2 |

+----+ +------+

In the above configuration we created two masters (assuming S2 could failover without authorization) in a perfectly symmetrical（对称的） way. Clients may write indefinitely to both sides, and there is no way to understand when the partition heals what configuration is the right one, in order to prevent a *permanent split brain condition*.

So please **deploy at least three Sentinels in three different boxes** always.

Example 2: basic setup with three boxes

This is a very simple setup（简单的设置）, that has the advantage to be simple to tune for additional safety. It is based on three boxes, each box running both a Redis process and a Sentinel process.

+----+

| M1 |

| S1 |

+----+

|

+----+ | +----+

| R2 |----+----| R3 |

| S2 | | S3 |

+----+ +----+

Configuration: quorum = 2

If the master M1 fails, S2 and S3 will agree about the failure and will be able to authorize a failover, making clients able to continue.

In every Sentinel setup, being Redis asynchronously replicated, there is always the risk of losing some write because a given acknowledged write may not be able to reach the slave which is promoted to master（就是在主从切换阶段会有数据不能写入，因为之前的master已经被重新配置为新的master的slave了，只读而不会写入）. However in the above setup there is an higher risk due to clients partitioned away with an old master, like in the following picture:

+----+

| M1 |

| S1 | <- C1 (writes will be lost)

+----+

|

/

/

+------+ | +----+

| [M2] |----+----| R3 |

| S2 | | S3 |

+------+ +----+

In this case a network partition isolated（隔离） the old master M1, so the slave R2 is promoted to master. However clients, like C1, that are in the same partition as the old master, may continue to write data to the old master. This data will be lost forever since when the partition will heal, the master will be reconfigured as a slave of the new master, discarding its data set.

This problem can be mitigated（减轻/缓和） using the following Redis replication feature, that allows to stop accepting writes（允许停止接受些命令） if a master detects that is no longer able to transfer its writes to the specified number of slaves.

min-slaves-to-write 1

min-slaves-max-lag 10

With the above configuration (please see the self-commented redis.conf example in the Redis distribution for more information) a Redis instance, when acting as a master, will stop accepting writes if it can't write to at least 1 slave. Since replication is asynchronous（异步） *not being able to write* actually means that the slave is either disconnected, or is not sending us asynchronous acknowledges（异步信号） for more than the specified max-lag number of seconds.

Using this configuration the old Redis master M1 in the above example, will become unavailable after 10 seconds. When the partition heals, the Sentinel configuration will converge to the new one, the client C1 will be able to fetch a valid configuration and will continue with the new master.

However there is no free lunch. With this refinement（设置）, if the two slaves are down, the master will stop accepting writes. It's a trade off.

Example 3: Sentinel in the client boxes

Sometimes we have only two Redis boxes available, one for the master and one for the slave. The configuration in the example 2 is not viable in that case, so we can resort to the following, where Sentinels are placed where clients are:

+----+ +----+

| M1 |----+----| R1 |

| S1 | | | S2 |

+----+ | +----+

|

+------------+------------+

| | |

| | |

+----+ +----+ +----+

| C1 | | C2 | | C3 |

| S1 | | S2 | | S3 |

+----+ +----+ +----+

Configuration: quorum = 2

In this setup, the point of view Sentinels is the same as the clients: if a master is reachable by the majority of the clients, it is fine. C1, C2, C3 here are generic clients, it does not mean that C1 identifies a single client connected to Redis. It is more likely something like an application server, a Rails app, or something like that.

If the box where M1 and S1 are running fails, the failover will happen without issues, however it is easy to see that different network partitions will result in different behaviors. For example Sentinel will not be able to setup if the network between the clients and the Redis servers will get disconnected, since the Redis master and slave will be both not available.

Note that if C3 gets partitioned（分段的） with M1 (hardly possible with the network described above, but more likely possible with different layouts, or because of failures at the software layer), we have a similar issue as described in Example 2, with the difference that here we have no way to break the symmetry（对称）, since there is just a slave and master, so the master can't stop accepting queries when it is disconnected from its slave, otherwise the master would never be available during slave failures.

So this is a valid setup but the setup in the Example 2 has advantages such as the HA system of Redis running in the same boxes as Redis itself which may be simpler to manage, and the ability to put a bound on the amount of time a master into the minority partition can receive writes.

Example 4: Sentinel client side with less than three clients

The setup described in the Example 3 cannot be used if there are not enough three boxes in the client side (for example three web servers). In this case we need to resort to a mixed setup like the following:

+----+ +----+

| M1 |----+----| R1 |

| S1 | | | S2 |

+----+ | +----+

|

+------+-----+

| |

| |

+----+ +----+

| C1 | | C2 |

| S3 | | S4 |

+----+ +----+

Configuration: quorum = 3

This is similar to the setup in Example 3, but here we run four Sentinels in the four boxes we have available. If the master M1 becomes not available the other three Sentinels will perform the failover.

In theory this setup works removing the box where C2 and S4 are running, and setting the quorum to 2. However it is unlikely that we want HA in the Redis side without having high availability in our application layer.

Sentinel, Docker, NAT, and possible issues

Docker uses a technique called port mapping（端口映射）: 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.

Docker is not the only software system where this happens, there are other Network Address Translation setups（网络地址解释设置） where ports may be remapped, and sometimes not ports but also IP addresses.

Remapping ports and addresses creates issues with Sentinel in two ways:

1. Sentinel auto-discovery of other Sentinels no longer works, since it is based on *hello* messages where each Sentinel announce at which port and IP address they are listening for connection. However Sentinels have no way to understand that an address or port is remapped, so it is announcing an information that is not correct for other Sentinels to connect.
2. Slaves are listed in the [INFO](https://redis.io/commands/info) output of a Redis master in a similar way: the address is detected by the master checking the remote peer of the TCP connection, while the port is advertised by the slave itself during the handshake, however the port may be wrong for the same reason as exposed in point 1.

Since Sentinels auto detect slaves using masters [INFO](https://redis.io/commands/info) output information, the detected slaves will not be reachable, and Sentinel will never be able to failover the master, since there are no good slaves from the point of view of the system, so there is currently no way to monitor with Sentinel a set of master and slave instances deployed with Docker, **unless you instruct Docker to map the port 1:1**.

For the first problem, in case you want to run a set of Sentinel instances using Docker with forwarded ports (or any other NAT setup where ports are remapped), you can use the following two Sentinel configuration directives in order to force Sentinel to announce a specific set of IP and port:

sentinel announce-ip <ip>

sentinel announce-port <port>

Note that Docker has the ability to run in *host networking mode* (check the --net=host option for more information). This should create no issues since ports are not remapped in this setup.

**A quick tutorial**

In the next sections of this document, all the details about Sentinel API, configuration and semantics（语义） will be covered incrementally. However for people that want to play with the system ASAP, this section is a tutorial that shows how to configure and interact with 3 Sentinel instances.

Here we assume that the instances are executed at port 5000, 5001, 5002. We also assume that you have a running Redis master at port 6379 with a slave running at port 6380. We will use the IPv4 loopback address 127.0.0.1 everywhere during the tutorial, assuming you are running the simulation on your personal computer.

The three Sentinel configuration files should look like the following:

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

The other two configuration files will be identical（相同的） but using 5001 and 5002 as port numbers（端口号）.

A few things to note about the above configuration:

* The master set is called  mymaster. It identifies（标识了） the master and its slaves. Since each *master set（主的设置）* has a different name, Sentinel can monitor different sets of masters and slaves at the same time.
* The quorum was set to the value of 2 (last argument of sentinel monitor configuration directive).
* The down-after-milliseconds value is 5000 milliseconds, that is 5 seconds, so masters will be detected as failing as soon as we don't receive any reply from our pings within this amount of time.

Once you start the three Sentinels, you'll see a few messages they log, like:

+monitor master mymaster 127.0.0.1 6379 quorum 2

This is a Sentinel event（sentinel事件）, and you can receive this kind of events via Pub/Sub（订阅） if you [SUBSCRIBE](https://redis.io/commands/subscribe) to the event name as specified later.

Sentinel generates and logs different events during failure detection and failover.

Asking Sentinel about the state of a master

The most obvious thing to do with Sentinel to get started, is check if the master it is monitoring is doing well:

$ redis-cli -p 5000（就是连接redis服务器的命令，其实redis的sentinel也是服务器以sentinel的模式开启运行）

127.0.0.1:5000> sentinel master mymaster（sentinel中配置的主的名字）

1) "name"

2) "mymaster"

3) "ip"

4) "127.0.0.1"

5) "port"

6) "6379"

7) "runid"

8) "953ae6a589449c13ddefaee3538d356d287f509b"

9) "flags"

10) "master"

11) "link-pending-commands"

12) "0"

13) "link-refcount"

14) "1"

15) "last-ping-sent"

16) "0"

17) "last-ok-ping-reply"

18) "735"

19) "last-ping-reply"

20) "735"

21) "down-after-milliseconds"

22) "5000"

23) "info-refresh"

24) "126"

25) "role-reported"

26) "master"

27) "role-reported-time"

28) "532439"

29) "config-epoch"

30) "1"

31) "num-slaves"（此master的slave个数）

32) "1"

33) "num-other-sentinels"（除了当前的sentinel外的sentinel的数量）

34) "2"

35) "quorum"

36) "2"

37) "failover-timeout"

38) "60000"

39) "parallel-syncs"

40) "1"

As you can see, it prints a number of information about the master. There are a few that are of particular interest for us:

1. num-other-sentinels is 2, so we know the Sentinel already detected two more Sentinels for this master. If you check the logs you'll see the +sentinel events generated（在开启sentinel时也会将当前可获得道德所有sentienl的简短信息输出））.
2. flags is just master. If the master was down we could expect to see s\_down or o\_down flag as well here.
3. num-slaves is correctly set to 1, so Sentinel also detected that there is an attached slave to our master.

In order to explore more about this instance, you may want to try the following two commands:

SENTINEL slaves mymaster

SENTINEL sentinels mymaster

The first will provide similar information about the slaves connected to the master, and the second about the other Sentinels（当前的sentinel信息不会打印出）.

Obtaining the address of the current master

As we already specified, Sentinel also acts as a configuration provider for clients that want to connect to a set of master and slaves. Because of possible failovers or reconfigurations, clients have no idea about who is the currently active master for a given set of instances, so Sentinel exports an API to ask this question:

127.0.0.1:5000> SENTINEL get-master-addr-by-name mymaster

1) "127.0.0.1"

2) "6379"

Testing the failover

At this point our toy Sentinel deployment is ready to be tested. We can just kill our master and check if the configuration changes. To do so we can just do:

redis-cli -p 6379 DEBUG sleep 30

This command will make our master no longer reachable, sleeping for 30 seconds. It basically simulates（模拟） a master hanging for some reason.

If you check the Sentinel logs, you should be able to see a lot of action:

1. Each Sentinel detects the master is down with an +sdown（主观down） event.
2. This event is later escalated（逐步升级） to +odown（客观down）, which means that multiple Sentinels agree about the fact the master is not reachable.
3. Sentinels vote a Sentinel that will start the first failover attempt（sentinel选出一个leader来执行failover过程）.
4. The failover happens.

If you ask again what is the current master address for mymaster, eventually we should get a different reply this time:

127.0.0.1:5000> SENTINEL get-master-addr-by-name mymaster

1) "127.0.0.1"

2) "6380"

So far so good... At this point you may jump to create your Sentinel deployment or can read more to understand all the Sentinel commands and internals.

**Sentinel API**

Sentinel provides an API in order to inspect（检查） its state（状态）, check the health of monitored masters and slaves, subscribe in order to receive specific notifications, and change the Sentinel configuration at run time.

By default Sentinel runs using TCP port 26379 (note that 6379 is the normal Redis port). Sentinels accept commands using the Redis protocol, so you can use redis-cli or any other unmodified Redis client in order to talk with Sentinel.

It is possible to directly query a Sentinel to check what is the state of the monitored Redis instances from its point of view, to see what other Sentinels it knows, and so forth. Alternatively, using Pub/Sub, it is possible to receive *push style* notifications（发布类型的通知） from Sentinels, every time some event happens, like a failover, or an instance entering an error condition, and so forth.

Sentinel commands

The following is a list of accepted commands, not covering commands used in order to modify the Sentinel configuration, which are covered later.

* **PING** This command simply returns PONG.
* **SENTINEL masters** Show a list of monitored masters and their state.
* **SENTINEL master <master name>** Show the state and info of the specified master.
* **SENTINEL slaves <master name>** Show a list of slaves for this master, and their state.
* **SENTINEL sentinels <master name>** Show a list of sentinel instances for this master, and their state.
* **SENTINEL get-master-addr-by-name <master name>** Return the ip and port number of the master with that name. If a failover is in progress or terminated successfully for this master it returns the address and port of the promoted slave.
* **SENTINEL reset <pattern>** This command will reset all the masters with matching name. The pattern argument is a glob-style pattern. The reset process clears any previous state in a master (including a failover in progress), and removes every slave and sentinel already discovered and associated with the master.
* **SENTINEL failover <master name>** Force a failover as if the master was not reachable, and without asking for agreement to other Sentinels (however a new version of the configuration will be published so that the other Sentinels will update their configurations).
* **SENTINEL ckquorum <master name>** Check if the current Sentinel configuration is able to reach the quorum needed to failover a master, and the majority needed to authorize the failover. This command should be used in monitoring systems to check if a Sentinel deployment is ok.
* **SENTINEL flushconfig** Force Sentinel to rewrite its configuration on disk, including the current Sentinel state. Normally Sentinel rewrites the configuration every time something changes in its state (in the context of the subset of the state which is persisted on disk across restart). However sometimes it is possible that the configuration file is lost because of operation errors, disk failures, package upgrade scripts or configuration managers. In those cases a way to to force Sentinel to rewrite the configuration file is handy（手动）. This command works even if the previous configuration file is completely missing.

Reconfiguring Sentinel at Runtime

Starting with Redis version 2.8.4, Sentinel provides an API in order to add, remove, or change the configuration of a given master. Note that if you have multiple sentinels you should apply the changes to all to your instances for Redis Sentinel to work properly. This means that changing the configuration of a single Sentinel does not automatically propagates the changes to the other Sentinels in the network.

The following is a list of SENTINEL sub commands used in order to update the configuration of a Sentinel instance.

* **SENTINEL MONITOR <name> <ip> <port> <quorum>** This command tells the Sentinel to start monitoring a new master with the specified name, ip, port, and quorum. It is identical to the sentinel monitor configuration directive in sentinel.conf configuration file, with the difference that you can't use an hostname in as ip, but you need to provide an IPv4 or IPv6 address.
* **SENTINEL REMOVE <name>** is used in order to remove the specified master: the master will no longer be monitored, and will totally be removed from the internal state of the Sentinel, so it will no longer listed by SENTINEL masters and so forth.
* **SENTINEL SET <name> <option> <value>** The SET command is very similar to the [CONFIG SET](https://redis.io/commands/config-set) command of Redis, and is used in order to change configuration parameters of a specific master. Multiple option / value pairs can be specified (or none at all). All the configuration parameters that can be configured via sentinel.conf are also configurable using the SET command.

The following is an example of SENTINEL SET command in order to modify the down-after-milliseconds configuration of a master called objects-cache:

SENTINEL SET objects-cache-master down-after-milliseconds 1000

Localhost:6379>config get slaveof可得到当前的slave信息

As already stated, SENTINEL SET can be used to set all the configuration parameters that are settable in the startup configuration file. Moreover it is possible to change just the master quorum configuration without removing and re-adding the master with SENTINEL REMOVE followed by SENTINEL MONITOR, but simply using:

SENTINEL SET objects-cache-master quorum 5

Note that there is no equivalent GET command since SENTINEL MASTER provides all the configuration parameters in a simple to parse format (as a field/value pairs array).

Adding or removing Sentinels

Adding a new Sentinel to your deployment is a simple process because of the auto-discover mechanism implemented by Sentinel. All you need to do is to start the new Sentinel configured to monitor the currently active master. Within 10 seconds the Sentinel will acquire the list of other Sentinels and the set of slaves attached to the master.

If you need to add multiple Sentinels at once, it is suggested to add it one after the other, waiting for all the other Sentinels to already know about the first one before adding the next. This is useful in order to still guarantee that majority can be achieved only in one side of a partition, in the chance failures should happen in the process of adding new Sentinels.

This can be easily achieved by adding every new Sentinel with a 30 seconds delay, and during absence of network partitions.

At the end of the process it is possible to use the command SENTINEL MASTER mastername in order to check if all the Sentinels agree about the total number of Sentinels monitoring the master.

Removing a Sentinel is a bit more complex: **Sentinels never forget already seen Sentinels**, even if they are not reachable for a long time, since we don't want to dynamically change the majority needed to authorize a failover and the creation of a new configuration number. So in order to remove a Sentinel the following steps should be performed in absence of network partitions:

1. Stop the Sentinel process of the Sentinel you want to remove.
2. Send a SENTINEL RESET \* command to all the other Sentinel instances (instead of \* you can use the exact master name if you want to reset just a single master). One after the other, waiting at least 30 seconds between instances.
3. Check that all the Sentinels agree about the number of Sentinels currently active, by inspecting the output of SENTINEL MASTER mastername of every Sentinel.

Removing the old master or unreachable slaves

Sentinels never forget about slaves of a given master, even when they are unreachable for a long time. This is useful, because Sentinels should be able to correctly reconfigure a returning slave after a network partition or a failure event.

Moreover, after a failover, the failed over master is virtually added as a slave of the new master, this way it will be reconfigured to replicate with the new master as soon as it will be available again.

However sometimes you want to remove a slave (that may be the old master) forever from the list of slaves monitored by Sentinels.

In order to do this, you need to send a SENTINEL RESET mastername command to all the Sentinels: they'll refresh the list of slaves within the next 10 seconds, only adding the ones listed as correctly replicating from the current master [INFO](https://redis.io/commands/info) output.

Pub/Sub Messages

A client can use a Sentinel as it was a Redis compatible Pub/Sub server (but you can't use [PUBLISH](https://redis.io/commands/publish)) in order to [SUBSCRIBE](https://redis.io/commands/subscribe) or [PSUBSCRIBE](https://redis.io/commands/psubscribe) to channels and get notified about specific events.

The channel name is the same as the name of the event. For instance the channel named +sdown will receive all the notifications related to instances entering an SDOWN (SDOWN means the instance is no longer reachable from the point of view of the Sentinel you are querying) condition.

To get all the messages simply subscribe using PSUBSCRIBE \*.

The following is a list of channels and message formats you can receive using this API. The first word is the channel / event name, the rest is the format of the data.

Note: where *instance details* is specified it means that the following arguments are provided to identify the target instance:

<instance-type> <name> <ip> <port> @ <master-name> <master-ip> <master-port>

The part identifying the master (from the @ argument to the end) is optional and is only specified if the instance is not a master itself.

* **+reset-master** <instance details> -- The master was reset.
* **+slave** <instance details> -- A new slave was detected and attached.
* **+failover-state-reconf-slaves** <instance details> -- Failover state changed to reconf-slaves state.
* **+failover-detected** <instance details> -- A failover started by another Sentinel or any other external entity was detected (An attached slave turned into a master).
* **+slave-reconf-sent** <instance details> -- The leader sentinel sent the [SLAVEOF](https://redis.io/commands/slaveof) command to this instance in order to reconfigure it for the new slave.
* **+slave-reconf-inprog** <instance details> -- The slave being reconfigured showed to be a slave of the new master ip:port pair, but the synchronization process is not yet complete.
* **+slave-reconf-done** <instance details> -- The slave is now synchronized with the new master.
* **-dup-sentinel** <instance details> -- One or more sentinels for the specified master were removed as duplicated (this happens for instance when a Sentinel instance is restarted).
* **+sentinel** <instance details> -- A new sentinel for this master was detected and attached.
* **+sdown** <instance details> -- The specified instance is now in Subjectively Down state.
* **-sdown** <instance details> -- The specified instance is no longer in Subjectively Down state.
* **+odown** <instance details> -- The specified instance is now in Objectively Down state.
* **-odown** <instance details> -- The specified instance is no longer in Objectively Down state.
* **+new-epoch** <instance details> -- The current epoch was updated.
* **+try-failover** <instance details> -- New failover in progress, waiting to be elected by the majority.
* **+elected-leader** <instance details> -- Won the election for the specified epoch, can do the failover.
* **+failover-state-select-slave** <instance details> -- New failover state is select-slave: we are trying to find a suitable slave for promotion.
* **no-good-slave** <instance details> -- There is no good slave to promote. Currently we'll try after some time, but probably this will change and the state machine will abort the failover at all in this case.
* **selected-slave** <instance details> -- We found the specified good slave to promote.
* **failover-state-send-slaveof-noone** <instance details> -- We are trying to reconfigure the promoted slave as master, waiting for it to switch.
* **failover-end-for-timeout** <instance details> -- The failover terminated for timeout, slaves will eventually be configured to replicate with the new master anyway.
* **failover-end** <instance details> -- The failover terminated with success. All the slaves appears to be reconfigured to replicate with the new master.
* **switch-master** <master name> <oldip> <oldport> <newip> <newport> -- The master new IP and address is the specified one after a configuration change. This is **the message most external users are interested in**.
* **+tilt** -- Tilt mode entered.
* **-tilt** -- Tilt mode exited.

Handling of -BUSY state

The -BUSY error is returned by a Redis instance when a Lua script is running for more time than the configured Lua script time limit. When this happens before triggering a fail over Redis Sentinel will try to send a [SCRIPT KILL](https://redis.io/commands/script-kill)command, that will only succeed if the script was read-only.

If the instance will still be in an error condition after this try, it will eventually be failed over.

Slaves priority

Redis instances have a configuration parameter called slave-priority. This information is exposed by Redis slave instances in their [INFO](https://redis.io/commands/info) output, and Sentinel uses it in order to pick a slave among the ones that can be used in order to failover a master:

1. If the slave priority is set to 0, the slave is never promoted to master.
2. Slaves with a *lower* priority number are preferred by Sentinel.

For example if there is a slave S1 in the same data center of the current master, and another slave S2 in another data center, it is possible to set S1 with a priority of 10 and S2 with a priority of 100, so that if the master fails and both S1 and S2 are available, S1 will be preferred.

For more information about the way slaves are selected, please check the **slave selection and priority** section of this documentation.

Sentinel and Redis authentication

When the master is configured to require a password from clients, as a security measure, slaves need to also be aware of this password in order to authenticate with the master and create the master-slave connection used for the asynchronous replication protocol.

This is achieved using the following configuration directives:

* requirepass in the master, in order to set the authentication password, and to make sure the instance will not process requests for non authenticated clients.
* masterauth in the slaves in order for the slaves to authenticate with the master in order to correctly replicate data from it.

When Sentinel is used, there is not a single master, since after a failover slaves may play the role of masters, and old masters can be reconfigured in order to act as slaves, so what you want to do is to set the above directives in all your instances, both masters and slaves.

This is also usually a sane setup since you don't want to protect data only in the master, having the same data accessible in the slaves.

However, in the uncommon case where you need a slave that is accessible without authentication, you can still do it by setting up **a slave priority of zero**, to prevent this slave from being promoted to master, and configuring in this slave only the masterauth directive, without using the requirepass directive, so that data will be readable by unauthenticated clients.

Sentinel clients implementation

Sentinel requires explicit client support, unless the system is configured to execute a script that performs a transparent redirection of all the requests to the new master instance (virtual IP or other similar systems). The topic of client libraries implementation is covered in the document [Sentinel clients guidelines](https://redis.io/topics/sentinel-clients).

**More advanced concepts**

In the following sections we'll cover a few details about how Sentinel work, without to resorting to implementation details and algorithms that will be covered in the final part of this document.

SDOWN and ODOWN failure state

Redis Sentinel has two different concepts of *being down*, one is called a *Subjectively Down* condition (SDOWN) and is a down condition that is local to a given Sentinel instance. Another is called *Objectively Down* condition (ODOWN) and is reached when enough Sentinels (at least the number configured as the quorum parameter of the monitored master) have an SDOWN condition, and get feedback from other Sentinels using the SENTINEL is-master-down-by-addr command.

From the point of view of a Sentinel an SDOWN condition is reached when it does not receive a valid reply to PING requests for the number of seconds specified in the configuration as is-master-down-after-millisecondsparameter.

An acceptable reply to PING is one of the following:

* PING replied with +PONG.
* PING replied with -LOADING error.
* PING replied with -MASTERDOWN error.

Any other reply (or no reply at all) is considered non valid. However note that **a logical master that advertises itself as a slave in the INFO output is considered to be down**.

Note that SDOWN requires that no acceptable reply is received for the whole interval configured, so for instance if the interval is 30000 milliseconds (30 seconds) and we receive an acceptable ping reply every 29 seconds, the instance is considered to be working.

SDOWN is not enough to trigger a failover: it only means a single Sentinel believes a Redis instance is not available. To trigger a failover, the ODOWN state must be reached.

To switch from SDOWN to ODOWN no strong consensus algorithm is used, but just a form of gossip: if a given Sentinel gets reports that a master is not working from enough Sentinels **in a given time range**, the SDOWN is promoted to ODOWN. If this acknowledge is later missing, the flag is cleared.

A more strict authorization that uses an actual majority is required in order to really start the failover, but no failover can be triggered without reaching the ODOWN state.

The ODOWN condition **only applies to masters**. For other kind of instances Sentinel doesn't require to act, so the ODOWN state is never reached for slaves and other sentinels, but only SDOWN is.

However SDOWN has also semantic implications. For example a slave in SDOWN state is not selected to be promoted by a Sentinel performing a failover.

Sentinels and Slaves auto discovery

Sentinels stay connected with other Sentinels in order to reciprocally check the availability of each other, and to exchange messages. However you don't need to configure a list of other Sentinel addresses in every Sentinel instance you run, as Sentinel uses the Redis instances Pub/Sub capabilities in order to discover the other Sentinels that are monitoring the same masters and slaves.

This feature is implemented by sending *hello messages* into the channel named \_\_sentinel\_\_:hello.

Similarly you don't need to configure what is the list of the slaves attached to a master, as Sentinel will auto discover this list querying Redis.

* Every Sentinel publishes a message to every monitored master and slave Pub/Sub channel \_\_sentinel\_\_:hello, every two seconds, announcing its presence with ip, port, runid.
* Every Sentinel is subscribed to the Pub/Sub channel \_\_sentinel\_\_:hello of every master and slave, looking for unknown sentinels. When new sentinels are detected, they are added as sentinels of this master.
* Hello messages also include the full current configuration of the master. If the receiving Sentinel has a configuration for a given master which is older than the one received, it updates to the new configuration immediately.
* Before adding a new sentinel to a master a Sentinel always checks if there is already a sentinel with the same runid or the same address (ip and port pair). In that case all the matching sentinels are removed, and the new added.

Sentinel reconfiguration of instances outside the failover procedure

Even when no failover is in progress, Sentinels will always try to set the current configuration on monitored instances. Specifically:

* Slaves (according to the current configuration) that claim to be masters, will be configured as slaves to replicate with the current master.
* Slaves connected to a wrong master, will be reconfigured to replicate with the right master.

For Sentinels to reconfigure slaves, the wrong configuration must be observed for some time, that is greater than the period used to broadcast new configurations.

This prevents Sentinels with a stale configuration (for example because they just rejoined from a partition) will try to change the slaves configuration before receiving an update.

Also note how the semantics of always trying to impose the current configuration makes the failover more resistant to partitions:

* Masters failed over are reconfigured as slaves when they return available.
* Slaves partitioned away during a partition are reconfigured once reachable.

The important lesson to remember about this section is: **Sentinel is a system where each process will always try to impose the last logical configuration to the set of monitored instances**.

Slave selection and priority

When a Sentinel instance is ready to perform a failover, since the master is in ODOWN state and the Sentinel received the authorization to failover from the majority of the Sentinel instances known, a suitable slave needs to be selected.

The slave selection process evaluates the following information about slaves:

1. Disconnection time from the master.
2. Slave priority.
3. Replication offset processed.
4. Run ID.

A slave that is found to be disconnected from the master for more than ten times the configured master timeout (down-after-milliseconds option), plus the time the master is also not available from the point of view of the Sentinel doing the failover, is considered to be not suitable for the failover and is skipped.

In more rigorous terms, a slave whose the [INFO](https://redis.io/commands/info) output suggests to be disconnected from the master for more than:

(down-after-milliseconds \* 10) + milliseconds\_since\_master\_is\_in\_SDOWN\_state

Is considered to be unreliable and is disregarded entirely.

The slave selection only considers the slaves that passed the above test, and sorts it based on the above criteria, in the following order.

1. The slaves are sorted by slave-priority as configured in the redis.conf file of the Redis instance. A lower priority will be preferred.
2. If the priority is the same, the replication offset processed by the slave is checked, and the slave that received more data from the master is selected.
3. If multiple slaves have the same priority and processed the same data from the master, a further check is performed, selecting the slave with the lexicographically smaller run ID. Having a lower run ID is not a real advantage for a slave, but is useful in order to make the process of slave selection more deterministic, instead of resorting to select a random slave.

Redis masters (that may be turned into slaves after a failover), and slaves, all must be configured with a slave-priority if there are machines to be strongly preferred. Otherwise all the instances can run with the default run ID (which is the suggested setup, since it is far more interesting to select the slave by replication offset).

A Redis instance can be configured with a special slave-priority of zero in order to be **never selected** by Sentinels as the new master. However a slave configured in this way will still be reconfigured by Sentinels in order to replicate with the new master after a failover, the only difference is that it will never become a master itself.

**Algorithms and internals**

In the following sections we will explore the details of Sentinel behavior. It is not strictly needed for users to be aware of all the details, but a deep understanding of Sentinel may help to deploy and operate Sentinel in a more effective way.

Quorum

The previous sections showed that every master monitored by Sentinel is associated to a configured **quorum**. It specifies the number of Sentinel processes that need to agree about the unreachability or error condition of the master in order to trigger a failover.

However, after the failover is triggered, in order for the failover to actually be performed, **at least a majority of Sentinels must authorize the Sentinel to failover**. Sentinel never performs a failover in the partition where a minority of Sentinels exist.

Let's try to make things a bit more clear:

* Quorum: the number of Sentinel processes that need to detect an error condition in order for a master to be flagged as **ODOWN**.
* The failover is triggered by the **ODOWN** state.
* Once the failover is triggered, the Sentinel trying to failover is required to ask for authorization to a majority of Sentinels (or more than the majority if the quorum is set to a number greater than the majority).

The difference may seem subtle but is actually quite simple to understand and use. For example if you have 5 Sentinel instances, and the quorum is set to 2, a failover will be triggered as soon as 2 Sentinels believe that the master is not reachable, however one of the two Sentinels will be able to failover only if it gets authorization at least from 3 Sentinels.

If instead the quorum is configured to 5, all the Sentinels must agree about the master error condition, and the authorization from all Sentinels is required in order to failover.

This means that the quorum can be used to tune Sentinel in two ways:

1. If a the quorum is set to a value smaller than the majority of Sentinels we deploy, we are basically making Sentinel more sensible to master failures, triggering a failover as soon as even just a minority of Sentinels is no longer able to talk with the master.
2. If a quorum is set to a value greater than the majority of Sentinels, we are making Sentinel able to failover only when there are a very large number (larger than majority) of well connected Sentinels which agree about the master being down.

Configuration epochs

Sentinels require to get authorizations from a majority in order to start a failover for a few important reasons:

When a Sentinel is authorized, it gets a unique **configuration epoch** for the master it is failing over. This is a number that will be used to version the new configuration after the failover is completed. Because a majority agreed that a given version was assigned to a given Sentinel, no other Sentinel will be able to use it. This means that every configuration of every failover is versioned with a unique version. We'll see why this is so important.

Moreover Sentinels have a rule: if a Sentinel voted another Sentinel for the failover of a given master, it will wait some time to try to failover the same master again. This delay is the failover-timeout you can configure in sentinel.conf. This means that Sentinels will not try to failover the same master at the same time, the first to ask to be authorized will try, if it fails another will try after some time, and so forth.

Redis Sentinel guarantees the *liveness* property that if a majority of Sentinels are able to talk, eventually one will be authorized to failover if the master is down.

Redis Sentinel also guarantees the *safety* property that every Sentinel will failover the same master using a different *configuration epoch*.

Configuration propagation

Once a Sentinel is able to failover a master successfully, it will start to broadcast the new configuration so that the other Sentinels will update their information about a given master.

For a failover to be considered successful, it requires that the Sentinel was able to send the SLAVEOF NO ONEcommand to the selected slave, and that the switch to master was later observed in the [INFO](https://redis.io/commands/info) output of the master.

At this point, even if the reconfiguration of the slaves is in progress, the failover is considered to be successful, and all the Sentinels are required to start reporting the new configuration.

The way a new configuration is propagated is the reason why we need that every Sentinel failover is authorized with a different version number (configuration epoch).

Every Sentinel continuously broadcast its version of the configuration of a master using Redis Pub/Sub messages, both in the master and all the slaves. At the same time all the Sentinels wait for messages to see what is the configuration advertised by the other Sentinels.

Configurations are broadcast in the \_\_sentinel\_\_:hello Pub/Sub channel.

Because every configuration has a different version number, the greater version always wins over smaller versions.

So for example the configuration for the master mymaster start with all the Sentinels believing the master is at 192.168.1.50:6379. This configuration has version 1. After some time a Sentinel is authorized to failover with version 2. If the failover is successful, it will start to broadcast a new configuration, let's say 192.168.1.50:9000, with version 2. All the other instances will see this configuration and will update their configuration accordingly, since the new configuration has a greater version.

This means that Sentinel guarantees a second liveness property: a set of Sentinels that are able to communicate will all converge to the same configuration with the higher version number.

Basically if the net is partitioned, every partition will converge to the higher local configuration. In the special case of no partitions, there is a single partition and every Sentinel will agree about the configuration.

Consistency under partitions

Redis Sentinel configurations are eventually consistent, so every partition will converge to the higher configuration available. However in a real-world system using Sentinel there are three different players:

* Redis instances.
* Sentinel instances.
* Clients.

In order to define the behavior of the system we have to consider all three.

The following is a simple network where there are 3 nodes, each running a Redis instance, and a Sentinel instance:

+-------------+

| Sentinel 1 |----- Client A

| Redis 1 (M) |

+-------------+

|

|

+-------------+ | +------------+

| Sentinel 2 |-----+-- // ----| Sentinel 3 |----- Client B

| Redis 2 (S) | | Redis 3 (M)|

+-------------+ +------------+

In this system the original state was that Redis 3 was the master, while Redis 1 and 2 were slaves. A partition occurred isolating the old master. Sentinels 1 and 2 started a failover promoting Sentinel 1 as the new master.

The Sentinel properties guarantee that Sentinel 1 and 2 now have the new configuration for the master. However Sentinel 3 has still the old configuration since it lives in a different partition.

We know that Sentinel 3 will get its configuration updated when the network partition will heal, however what happens during the partition if there are clients partitioned with the old master?

Clients will be still able to write to Redis 3, the old master. When the partition will rejoin, Redis 3 will be turned into a slave of Redis 1, and all the data written during the partition will be lost.

Depending on your configuration you may want or not that this scenario happens:

* If you are using Redis as a cache, it could be handy that Client B is still able to write to the old master, even if its data will be lost.
* If you are using Redis as a store, this is not good and you need to configure the system in order to partially prevent this problem.

Since Redis is asynchronously replicated, there is no way to totally prevent data loss in this scenario, however you can bound the divergence between Redis 3 and Redis 1 using the following Redis configuration option:

min-slaves-to-write 1

min-slaves-max-lag 10

With the above configuration (please see the self-commented redis.conf example in the Redis distribution for more information) a Redis instance, when acting as a master, will stop accepting writes if it can't write to at least 1 slave. Since replication is asynchronous *not being able to write* actually means that the slave is either disconnected, or is not sending us asynchronous acknowledges for more than the specified max-lag number of seconds.

Using this configuration the Redis 3 in the above example will become unavailable after 10 seconds. When the partition heals, the Sentinel 3 configuration will converge to the new one, and Client B will be able to fetch a valid configuration and continue.

In general Redis + Sentinel as a whole are a an **eventually consistent system** where the merge function is **last failover wins**, and the data from old masters are discarded to replicate the data of the current master, so there is always a window for losing acknowledged writes. This is due to Redis asynchronous replication and the discarding nature of the "virtual" merge function of the system. Note that this is not a limitation of Sentinel itself, and if you orchestrate the failover with a strongly consistent replicated state machine, the same properties will still apply. There are only two ways to avoid losing acknowledged writes:

1. Use synchronous replication (and a proper consensus algorithm to run a replicated state machine).
2. Use an eventually consistent system where different versions of the same object can be merged.

Redis currently is not able to use any of the above systems, and is currently outside the development goals. However there are proxies implementing solution "2" on top of Redis stores such as SoundCloud [Roshi](https://github.com/soundcloud/roshi), or Netflix [Dynomite](https://github.com/Netflix/dynomite).

Sentinel persistent state

Sentinel state is persisted in the sentinel configuration file. For example every time a new configuration is received, or created (leader Sentinels), for a master, the configuration is persisted on disk together with the configuration epoch. This means that it is safe to stop and restart Sentinel processes.

TILT mode

Redis Sentinel is heavily dependent on the computer time: for instance in order to understand if an instance is available it remembers the time of the latest successful reply to the PING command, and compares it with the current time to understand how old it is.

However if the computer time changes in an unexpected way, or if the computer is very busy, or the process blocked for some reason, Sentinel may start to behave in an unexpected way.

The TILT mode is a special "protection" mode that a Sentinel can enter when something odd is detected that can lower the reliability of the system. The Sentinel timer interrupt is normally called 10 times per second, so we expect that more or less 100 milliseconds will elapse between two calls to the timer interrupt.

What a Sentinel does is to register the previous time the timer interrupt was called, and compare it with the current call: if the time difference is negative or unexpectedly big (2 seconds or more) the TILT mode is entered (or if it was already entered the exit from the TILT mode postponed).

When in TILT mode the Sentinel will continue to monitor everything, but:

* It stops acting at all.
* It starts to reply negatively to SENTINEL is-master-down-by-addr requests as the ability to detect a failure is no longer trusted.

If everything appears to be normal for 30 second, the TILT mode is exited.

Note that in some way TILT mode could be replaced using the monotonic clock API that many kernels offer. However it is not still clear if this is a good solution since the current system avoids issues in case the process is just suspended or not executed by the scheduler for a long time.