Sentinel client guidelines

aredis can be used together with redis sentinel to discover redis nodes. To use Sentinol support, you must have at least one Sentinel daemon running. Connecting aredis to an instance (instance) of the Sentinel is easy. You can use the Sentinel connection to discover the network addresses of the master and slaves: from redis.sentinel import Sentinel sentinel = Sentinel([(localhost', 26379)], stream_timeout=0.1) waiting sentinel.discover_master('mymaster') # ('127.0.0.1', 6379) waiting sentinel.discover_slaves('mymaster') # [('127.0.0.1', 6380)] You can also create a Redis client connection from the Sentinel instance. You can connect to either the master (for write operations) or the slave (for read-only operations). master=sentinel.master_for('mymaster', stream_timeout=0.1) slave=sentinel.slave_for('mymaster', stream_timeout=0.1) master.set('foo', 'bar') slave.get('foo') #'bar' Main and slave objects are normal StrictRedis cases with their connection pool bound to an instance of sentinel. When a Sentinel-backed client tries to establish a connection, it first queries sentinel servers to determine the appropriate host to connect to. If no server is found, MasterNotFoundError or SlaveNotFoundError will be increased. Both exceptions are connectionerror subclasses. When you try to connect to a slave client, the Sentinel connection pool will be erate through the slave list until it finds one that can be connected. If there are no slaves connected, he'll contact the master. For more information about redis Sentinel, see redis sentinel client guidelines. KeyDB has options for using Sentinel derived in redis, but with active-replication options provided, sentinel is not necessary and can be more complex to use. However, if you migrate through the Redis project, KeyDB will continue to work with the Sentinel instance settings. KeyDB Sentinel Documentation KeyDB Sentinel provides high availability for KeyDB. In practice, this means that with sentinel you can create a KeyDB deployment that resists without human intervention for certain types of failures. KeyDB Sentinel also provides other ancillary tasks such as monitoring, notifications, and acting as a configuration provider for clients. The following is a complete list of sentinel capabilities at macroscogical level (i.e. big picture): Monitoring. The Sentinel constantly checks to see if your master and slave instances are working as expected. Notification. Sentinel can notify system administrators, other computer programs, via the API that something is wrong with one of the monitored KeyDB instances. Automatic failure. If the master does not work as expected, the Sentinel can run a failure process where the slave is promoted to master, other other slaves are reconfigured to use the new master and applications that use the KeyDB server about the new address to be used when connecting. Configuration provider. Sentinel acts as a source of authority for client service discovery: clients connect to sentinels to request the address of the current master keydb responsible for the service. If a failure occurs, Sentinels reports a new address. The distributed nature of the Sentinel KeyDB Sentinel is a distributed system: The Sentinel itself is designed to run in a configuration where there are multiple Sentinel processes to work together. The advantage of collaborating multiple Sentinel processes is as follows: Failure detection is performed when multiple sentinels agree that the master is no longer available. This reduces the likelihood of false alarms. Sentinel works, although not all Sentinel processes work, so the system is fault-resistant. There is no fun in having a failure through a system that is itself the only point of failure, after all. The sum of Sentinels, KeyDB instances (masters and slaves) and clients connecting to Sentinel and KeyDB are also larger distributed systems with specific properties. This document gradually introduces concepts ranging from the basic information needed to understand the basic properties of Sentinol to more complex information (which is optional) to understand exactly how sentinel works. KeyDB Sentinel Spec KeyDB Sentinel is the name of the KeyDB high availability solution that is currently under development. It has nothing to do with the KeyDB cluster and is designed to be used by people who don't need a KeyDB cluster, but simply a way to perform automatic crashes when the main instance isn't working properly. The plan is to provide usable beta implementation of keydb sentinel in a short time, preferably in mid-July 2012. In short, this is what the KeyDB Sentinel will be able to do: Monitor master and slave instances to see if they are available. Encourage the slave to cope when the master fails. Adjust client configurations when you select a slave. Notify your system administrator of incidents using notifications. So three different keydb sentinel tasks can be summarized in the following three big aspects: Monitoring. Notification. Automatic failure. The following document explains what the KeyDB Sentinel design is to achieve these goals. KeyDB Sentinel idea The idea of a KeyDB Sentinel is to have multiple monitoring devices at different points of your network, monitoring keydb's main instance. However, these independent establishments cannot act without agreement with other sentinels. When a primary Instance of KeyDB is detected to fail to start the failure process, the indicator must verify that the agreement level exists. Number of indicators, their location on the network and the quorum configured, select the desired behavior between Options. KeyDB Sentinel does not use any proxy: Client reconfiguration is performed by running user-provided executable files (such as a shell script or python program) in a user-specific setting. In what form will be supplied KeyDB Sentinel is only a special mode keydb-server executable file. If a keydb-server is called with a keydb-sentinel as argv[0] (for example, by using a symbolic link or copying a file), or if the --sentinel option is passed, the KeyDB instance starts in sentinel mode and will only understand sentinel related commands. All other commands will be rejected. The entire implementation of sentinol will live in a separate sentinel.c file with minimal impact on the rest of the code base. However, this solution allows you to use all devices that have already been implemented inside KeyDB without the need to re-implement them or maintain a separate code base for the KeyDB Sentinel. Sentinels Network All sentinel with a permanent connection with: Monitored Masters. All its slaves that are discovered using the main INFO output. All other Sentinels attached to this master, discovered via Pub/Sub. Sentinels use the KeyDB protocol to talk to each other, and respond to external clients. KeyDB Sentinels export the SENTINEL command. SENTINEL sub-orders are used to perform various actions. For example, the sentinel masters command enumerates all monitored masters and their statuses. However, Sentinels may also respond to the PING command as a normal KeyDB instance, so it is possible to monitor the Sentinel considering this normal KeyDB instance. The list of network tasks performed by each sentinel is as follows: Sentinel publish your presence using the main Pub/Sub several times every five seconds. Sentinel accepts commands using a TCP port. By default, the port is 26379. The Sentinel constantly monitors masters, slaves, other sentinels sending PING commands. Sentinel sends INFO commands to masters and slaves every ten seconds to make a new list of connected slaves, master status, and so on. Sentinel monitors the Sentinel Pub/Sub hello channel in order to discover newly connected Sentinels, or to detect already connected Sentinels. The channel used __sentinel__:hello. Sentinels Discovering If you want to configure sentinels as simple as possible each sentinel sends out its presence using the KeyDB master Pub/Sub feature. Each sentinel is subscribed to the same channel and broadcasts information about its existence on the same channel, including the identification of the run sentinel, and the IP address and port where it listens to commands. Each sentinel maintains a list of other sentinels run ID, IP and port. Sentinel that has already announced its presence using Pub/Sub for too long will be removed from the list, provided that it seems to be working well. In this case, the notification is delivered to the system administrator. Detecting failing child the instance is not available from the KeyDB Sentinel perspective when it is no longer able to respond correctly to the PING command for more than the specified number of seconds in a row. Ping response considered valid, one of the following conditions should be payable: PING replied +PONG. PING replied with a -Loading error. Ping replied with a -MASTERDOWN error. What is not considered an acceptable answer: PING replied with a -BUSY error. PING replied with a -MISCONF error. The PING response was not received after more than a specified number of milliseconds. PING should never answer with a different error code than those mentioned above, but any other error code is considered an acceptable answer by the KeyDB Sentinel. Processing status -BUSY Error -BUSY is returned when the script is running longer than the configured script timeout. When this happens before you run a failure through the KeyDB Sentinel, it tries to send a SCRIPT KILL command that succeeds only if the script was read-only. Subjectively down and objectively down in terms of sentinel there are two different error conditions for the captain: Subjective down (aka S_DOWN) means that the master is down in terms of sentinel. Objectively down (aka O_DOWN) means that the master is subjectively down in terms of sufficient Sentinels to achieve a configured quorum for that master. As Sentinels agrees to mark the O_DOWN. When the Sentinel detects that the captain is in S_DOWN state, he begins sending a sentinel request to the master-down-by-addr every second. The answer is stored in the state that each Sentinel has in mind. Ten times per second the Sentinel scans the condition and checks to see if there are enough Sentinels thinking that the master is down (it is not specific to this operation, most state checks are done with this frequency). If this Sentinel already has a S_DOWN condition for this master, and there are enough other sentinels that have recently reported this condition (the validity period is currently set to 5 seconds), then the master is marked as O_DOWN (objectively down). Note that the O_DOWN is not propagated among sentinels. Each sentinel can achieve this condition independently. Sentinels command SENTINEL is-master-down-by-addr to ask other Sentinels about master status from their local point of view using SENTINEL's-master-down-by-addr command. This command corresponds to a boolean value (in the form of 0 or 1 integer response as the first element of the bulk response). However, to avoid false alarms, the command acts as follows: If the specified ip and port is not known, 0 will be returned. If the specified ip and port are present but do not belong to the for example, 0 will be returned. If the sentinel is in TILT mode (see below), it will return on 0 December 2015. The value of 1 is returned only if the instance is known, the master is selected, the S_DOWN sentinel is in TILT mode. Duplicate sentinel deletion To achieve the configured quorum, we want to make sure that the quorum is reached by different physical instances of sentinel. Under no circumstances should we get approval from the same instance that for some reason there appears to be two or more different sentinel instances. This is forced by aggressive removal of duplicate Sentinels: every time the Sentinel sends a message in the Hello Pub/Sub channel with its address and runid, if we can't find the perfect match (same runid and address) inside the Sentinels table for this master, we remove all other Sentinel with the same runid or same address. And later add a new Sentinel. For example, if the Sentinel instance restarts, the run ID will vary, and the old Sentinel with the same IP address and port pair will be removed. Trigger failover: Leaders and observers fact that the master is labeled as a O_DOWN not enough to star in the failover process. What the Sentinel should start with spillover will also be decided. Also Sentinels can be configured in two ways: only as monitors that cannot perform a failure over, or as Sentinels that can trigger a failure. What is desirable is that only the Sentinel will begin the failure process, and this Sentinel should be selected among the Sentinels that are allowed to perform the failover. In the Sentinel there are two tasks during failure: Leader Sentinel is the one chosen to perform the failover. Sentinels observers are other sentinels just after the failover process without doing active operations. So the condition for running failover is: Master in O_DOWN state. Sentinel, who is elected leader. The Leader sentinel election process works as follows: Each Sentinel with a master in the state updates its internal state at a frequency of 10 HZ O_DOWN to restore what is a subjective leader from his point of view. Subjective Leader is selected in this way by each sentinel. Every Sentinel we know of a given master that is achievable (no S_DOWN state) that is allowed to perform a failover (this Sentinel-specific configuration is propagated through the Hello channel) is a possible candidate. Of all possible candidates, the one with a lexicographically smaller Run ID is selected. Each time the Sentinel answers with master's sentinel-down-by-addr command it also responds with the Run ID of its subjective Leader. Each Sentinel with a failing master (O_DOWN) checks its subjective leader and subjective leaders of all other Sentinels at a frequency of 10 HZ and is 10 HZ marked as Leader if the following conditions happen: subjective leader. At least N-1's other Sentinels, which see the master as down, and are achievable, also think he is a leader. Because N is a quorum configured for this master. At least 50% + 1 of all sentinels involved in the voting process (which are achievable and which also see the captain fail) should be agreed upon by the Leader. So, for example, if there are a total of three sentinels, a master fails, and all three sentinels are able to communicate (no Sentinel fails) and the configured quorum for that master is 2, the sentinel will feel itself an objective leader if at least it and the other Sentinel agree to be a subjective leader. Once the Sentinel discovers that it is an objective leader, it marks the master with flags FAILOVER_IN_PROGRESS and IM_THE_LEADER and starts the process of failure in SENTINEL_FAILOVER_DELAY (currently 5 seconds) plus a random additional time of between 0 milliseconds and 10,000 milliseconds. During this time we ask INFO for all slaves with an increased frequency of one time per second (usually the period is 10 seconds). If the slave, meanwhile, turns into a master, failure is suspended, and the leader IM_THE_LEADER to turn into an observer. Guarantees of the electoral process leader As you can see on the Sentinel to become leader a majority is not necessarily necessary. A user can force a majority to only need to set the main quorum, for example, to 5 if there are a total of 9 sentinels. However, it is also possible to set the quorum to 2 with 9 sentinels in order to improve resistance to netsplit or failing Sentinels or other error conditions. In this case, protection against racial conditions (multiple Sentinels begin to perform failures at the same time) is given by the random delay used to initiate the failure over, and a continuous monitor of slave instances to determine whether another Sentinel (or human) has started the process of failure. In addition, the slave in support is chosen through a deterministic process to minimize the chances that two different Sentinels with a full vision of working slaves can choose two different slaves to support. However, it is easy to imagine netsplits and specific configurations where two Sentinels can begin to act as leaders at the same time, electing two different slaves as masters, in two different parts of the network that cannot communicate. The KeyDB Sentinel user should evaluate the network topology and select the appropriate quorum in view of its objectives and various compromises. As observers understand, the failure started by the Observer is just the Sentinel, which does not believe it to be a leader but still sees a master O_DOWN state. The observer is still able to monitor and update the internal situation based on what is happening relies directly on Leader to communicate with him in order to be informed of progress. It simply observes the state of slaves to understand what is happening. Specifically, observers flag the master as FAILOVER_IN_PROGRESS if the slave attached to the master turns into a master (observers can see in the output info). The observer will also consider the failover complete once all the other achievable slaves appear to be slaves to this slave that has been converted into a master. If slave in FAILOVER_IN_PROGRESS and failover does not progress too much time, and at the same time other Sentinels begin to claim that this Sentinel is an objective leader (because, for example, the old leader is no longer achievable), the Sentinel will flag itself as a IM_THE_LEADER and will continue to fail. Note: the entire state of the Sentinel, including subjective and objective guidance, is a dynamic process that is constantly renewed with a period of 10 HZ. There is no step once in the Sentinel. Choosing Slave to support If the master has multiple slaves, the slave support master is selected by checking the priority of the slave (a new option to configure instances of KeyDB that spreads through the OUTPUT INFO) and selecting a number with a lower priority value (it is an integer similar to that in the MX field of DNS). All slaves that seem to be disconnected from the master for a long time are discarded (outdated data). If there are slaves with the same priority, the one with the lexicographically smaller Run ID is selected. If there is no Slave to choose because all salves failover fails it is not running at all. Instead, if there is no Slave to choose from because the master has never had slaves in the monitoring session, then failover is done however only by calling user scripts. However, for this to happen, a special configuration option (force-failover-without-slaves) must be set for this captain. This is useful because there are configurations where a new instance can be provisioned at the IP script level, but there are no connected slaves. Process failure The failure process consists of the following steps: Turns the selected slave into a master by using the SLAVEOF NO ONE command. Turn all remaining slaves, if any, into slaves of the new master. This is done gradually, one slave behind, waiting for the previous slave to complete the synchronization process before starting with another. Call the user script to inform clients that the configuration has changed. Completely remove the old failing master from the table and add a new master page with the same name. If steps 1 fail, the failure is interrupted. All other defects shall be considered non-fatal. Tilt Mode KeyDB Sentinel is heavily dependent on computer time: for example, in order to understand whether is available that remembers the time of the last successful response to ping, and compares it with the current time to understand how many years it is. However, if your computer's time changes unexpectedly, or if your computer is very busy or the process is blocked for some reason, the sentinel might start behaving in an unexpected way. TILT mode is a special protection mode that can enter the sentinel when something special is detected that can reduce system reliability. The Sentinel timer break is usually called 10 times per second, so we expect more or less 100 milliseconds to escape between two calls to interrupt the timer. What sentinel does is register the previous timer interrupt it was called, and compare it with the current call: if the time difference is negative or unexpectedly large (2 seconds or more) tilt mode is specified (or if the output from TILT mode has already been postponed). When it is in TILT mode, the Sentinel will continue to monitor everything, but: It will stop acting at all. It begins to respond negatively to sentinel's master-down-by-addr requests as the ability to detect failures is no longer trusted. If everything appears normal for 30 seconds, TILT mode will exit. Sentinels monitor other sentinels When sentinel no longer advertises itself using the Pub/Sub channel for too much time (30 minutes more configured timeout for the master), but at the same time the master appears to work correctly, the Sentinel is removed from the Sentinels table for this master and the notification is sent to the system administrator. User-provided Sentinels scripts can optionally call user-provided scripts to perform two tasks: Notify clients that the configuration has changed. Notify your system administrator of problems. The script to inform clients about the configuration change has the following parameters: ip:port call Sentinel. old master ip:port. new main ip:port. The notification script is called with the following parameters: ip:sentinel-called port. The message to be delivered to the system administrator shall be transmitted to the standard input. By using ip:port sentinel calls, scripts can call SENTINEL subcommittees to get more information if necessary. Specific implementation notification scripts will probably use a mail command or some other command to deliver SMS messages, emails, tweets. The implementation of the script to modify the configuration in Web applications is likely to use HTTP GET requirements to force clients to update the configuration, or any other reasonable mechanism for specific settings in use. Examples of Imaginary Settings: Computer A starts the KeyDB master. computer B starts KeyDB slave and client software. In this naïve configuration, one sentinel can be placed, with a minimum agreement set to one (without from other necessary sentinels), running on B. If A fails failure through the process begins, the slave will be chosen to handle it, and the client software will be reconfigured. Imaginary settings: Computer A starts KeyDB master computer B starts KeyDB slave computer C, D, E, F, G are Web servers acting as clients In this setting it is possible to run five sentinels located on C, D, E, F, G with the minimum agreement set to 3. The real production environment is to evaluate how individual computers are networked together, and to check what happens during clean distributions in order to choose where to place sentinels, and the level of minimum agreement, so that one hand of the network otherwise does not start to fail over. As a general rule, if a complex network topology is present, the minimum agreement should be set to the maximum number of indicators existing simultaneously in the same network arm plus one. SENTINEL SUBSUBSUCE SENTINEL masters, provides a list of configured masters. SENTINEL slaves <master name=>, provides a list of slaves for the master with the specified name. Sentinel sentinels <master name=>, provides a list of indicators for the master with the specified name. SENTINEL is-a-master-down-by-addr <ip> <port>, returns two elements of a more bulk response where the first element is:0 or :1, and the other is a subjective Leader for failover. A more detailed specification of how user script errors are handled, including what return codes might mean, such as 0: try again. 1: fatal error. 2: try again, and so on. A more detailed specification of what happens when the user script is not returned at that time. Add a push notification system for configuration changes. A document that specifies, for each master of the monitored configuration, a name for the master that is reported to all SENTINEL commands. It is clear that we are processing one Sentinel monitoring multiple masters. The current version of sentinol is called Sentinel 2. This is a rewrite of the initial implementation of sentinel using stronger and easier to predict algorithms (which are explained in this documentation). The stable edition of keydb sentinel comes from KeyDB 2.8. New development is carried out in an unstable branch, and new functions are sometimes ported back to the last stable branch once they are considered stable. KeyDB Sentinel version 1, shipped with KeyDB 2.6, is outdated and should not be used. Run sentinol If you are using a keydb-sentinel executable file (or if you have a symbolic link with this name to a keydb-server executable file), you can run the Sentinel with the following command command command: keydb-sentinel /path/to/sentinel.conf Otherwise, you can use the directly executable keydb-server file. which runs it in Sentinel mode: keydb-server /path/to/sentinel.conf --sentinel Both methods work the same way. However, it is required to use the configuration file at startup</port> </ip> </master> </master> because this file will be used by the system to save the current state, which will be reloaded in case of restart. Sentinel simply refuses to run if no configuration file is specified or if the path of the configuration file is not writable. Sentinels by default start listening to connect to TCP port 26379, so for Sentinels to work, port 26379 servers must be open to receive connections from the IP addresses of other Sentinel instances. Otherwise sentinels can not talk and can not agree with what to do, so failover will never be done. Basic things to know about sentinel before deploying You need at least three instances of Sentinel for robust deployment. Three instances of sentinel should be placed on computers or virtual machines that are expected to fail independently. So, for example, different physical servers or virtual machines are performed in different availability zones. Sentinel + KeyDB distributed system does not guarantee that confirmed writes are retained during failure because KeyDB uses asynchronous replication. However, there are ways to deploy sentinel that make window lose writes limited to certain moments, while there are other less secure ways to deploy. You need Sentinel support from your clients. Popular client libraries have Sentinel support, but not all. There is no HA setting that is safe if you do not test from time to time in development environments, or even better if you can, in a production environment if they work. You may have an incorrect configuration that only takes effect when it is 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 discovering other sentinel processes and listing slaves for master. For more information, see the Sentinel and Docker section later in this document. Configuring sentinel KeyDB source distribution contains a file named sentinel.conf, which is a self-documented example configuration file, which you can use to configure sentinel, however a typical minimum configuration file looks like this: sentinel monitor mymaster 127.0.0.1 63 79 2 sentinel down-after-milliseconds mymaster 60000 sentinel failover-timeout mymaster 180000 sentinel parallel synchronizes mymaster 1 sentinel monitor re 192.168.0 1.3 6380 4 sentinel down-after-milliseconds resque 10000 sentinel failover-timeout resque 180000 sentinel parallel-syncs resque 5 Just enter master pages to watch , giving each separated master (who can have any number of slaves) a different name. There is no need to specify the slaves that are auto-discovered. The Sentinel will update the configuration automatically with additional information about slaves (in order to information in case of restart). The configuration is Overwritten every time a slave is promoted to handle during a failover and every time a new Sentinel is discovered. An example of the configuration above, basically follow two sets of KeyDB instances, each consisting of a master and an undefined number of slaves. One set of instances is called mymaster, and the other resque. The meaning of the sentinel monitor statement arguments is as follows: sentinel monitor <master-group-name> <ip> <port> <quorum>For the sake of clarity, let's check line by line what the configuration options mean: The first line is used to make KeyDB track master named mymaster, it is at 127.0.0.1 and port 6379, with quorum 2. Everything is pretty obvious, but the quorum argument: Quorum is the number of Sentinels that must agree that the master is not achievable in order to really label the slave as a failure, and eventually begin to fail over the procedure if possible. However, the quorum is only used to detect a failure. In order to actually perform a failover, one of the Sentinels must be elected leader for failover and be entitled to continue. It only happens by voting for most Sentinel processes. So, for example, if you have 5 Sentinel processes, and the quorum for a given master file is 2, that's what happens: If two Sentinels agree at the same time about the master being unreachable, one of the two will try to run the failover. If at least three sentinels are achievable, the backup failure will be enabled and actually triggered. From a practical point of view, this means that during the failures of the Sentinel never begins to fail if most Sentinel processes are unable to speak (a.s. no failure in the minority section). Other Sentinel options Other options are almost always in the form: sentinel A <option_name> <master_name> <option_value>used for the following purposes: down-after-milliseconds is the time in milliseconds instance should not be achievable (either does not respond to our PINGS, or is answering with an error) for sentinel starting to think it is down. Parallel synchronizes sets the number of slaves that can be reconfigured to use the new master after failover at the same time. The lower the number, the more time it will take for the failover process to complete, but if the slaves are configured to serve the old data, you may not want all slaves to re-synchronize with the master at the same time. While the replication process is mostly not blocking for the slave, there is a moment when it stops to retrieve bulk data from the master page. You may want to make sure only one slave at a time is not reachable by setting this option to 1. Additional options are described in the rest of this document and documented in the example sentinel.conf file that comes with the KeyDB distribution. All configuration parameters can</option_value> </master_name> </option_name> </quorum> </port> </ip> modified at the time of the runtime using the SENTINEL SET command. For more information, see Reconfigure sentinel at runtime. Example Sentinel Deployment Now that you know the basic information about the Sentinel, you may wonder where you should place sentinel processes, how many Sentinel processes you need, and so on. Here are some examples of deployments. We use ASCII art to show you examples of configuration in graphical format, this is what different symbols mean: +--------------------+ | This is a computer | | or VM that fails | | Independently. We | | call it boxing | +--------------------+ We write in the boxes what runs: +-------------------+ | KeyDB master M1 | | KeyDB Sentinel S1 | +-------------------+ Various boxes are connected by lines to show that they are capable of speaking: +-------------+ +-------------+ | Sentinel S1 |---------------| Sentinel S2 | +-------------++-------------+ Network areas appear as broken lines with slashes: +-------------++-------------+ | Sentinel S1 |------ // ------| Sentinel S2 | +-------------+ +-------------+ Note also that: Masters are called M1, M2, M3, ..., Mn. Slaves are called R1, R2, R3, ..., Rn (R means replica). Sentinels are called S1, S2, S3, ..., Sn. Clients are called C1, C2, C3, ..., Cn. Note that we never show settings where only two Sentinels are used, because Sentinels always need to talk to most in order to start a failover. Example 1: only two Sentinels, NEÚs TO +----+ +----+ | M1 |---------| Article 10 02 — Sub-total | | S2 | +----+ +----+ Configuration: Quorum = 1 In this setting, if the main M1 fails, R1 will be supported because two Sentinels can reach an agreement to fail (of course with the quorum set to 1) and may also allow failure because most are two. So apparently it might work superficially, but check other points to see why this setting is broken. If the field in which m1 is running stops working, S1 will also stop working. Sentinel running in the second S2 field will not be able to enable failure, so the system will not be available. Note that most are needed to order various failovers and later distribute the latest configuration to all Sentinels. Note also that the ability to failover on one side of the above settings, without any agreement, would be very dangerous: +----+ +------+ | M1 |----//-----| [M1] | | S1 | | S2 | +----+ +------+ In the above configuration, we have created two masters (assuming that S2 could fail without permission) in a perfectly symmetrical way. Clients can write endlessly on both sides, and there is no way to understand when a partition heals what the configuration is to prevent a permanent division of the brain state. So please deploy at least three Sentinels in three different boxes each time. Example 2: basic setup with three boxes This is a very simple setup that has the advantage of being easy to tune for additional security. It is based on three boxes, each running both the KeyDB process and the Sentinel process. +----+ | It is also ad point (a) of Article 1 S1 | +----+ | +----+ | +----+ | R2 |----+----| R3 | | S2 | | S3 | +----++----+ Configuration: Quorum = 2 If The Master M1 Fails, S2 and S3 will agree on a failure and will be able to authorize the failure so that clients can continue. In every Sentinel setting, KeyDB is asynchronously replicated, there is always a risk of losing some writing because given acknowledged writing may not be able to reach a slave who is promoted to master. However, in the above settings there is a higher risk due to clients split off with the old master, as in the following figure: +----+ | It is also ad point (a) of Article 1 S1 | <- C1 (writes lost) +----+ | / / +------+ | +----+ | [M2] |----+----| R3 | | S2 | | S3 | +------+ +----+ In this case, the network partition isolated the old Master M1, so slave R2 is promoted to master. However, clients like C1, which are in the same field as the old master, can continue to write data to the old master. This data will be lost forever, because when the partition is captured, the master will be reconfigured as a slave to the new master and will delete the dataset. This issue can be mitigated by using the following KeyDB replication feature, which allows you to stop receiving writes if the master detects that it is no longer able to pass its writes to a specified number of slaves. min-slaves-to-write 1 min-slaves-max-lag 10 With the configuration above (see self-commented redis.conf example in keydb distribution for more information) KeyDB instance, when acting as a master, stops accepting writes if it can not write at least 1 slave. Since replication is asynchronously not being able to write actually means that the slave is either disconnected or is not sending us asynchronously acknowledging more than the specified max-lag number of seconds. Using this configuration of the old KeyDB master M1 in the example above, it becomes unavailable after 10 seconds. When the partition is reassued, the Sentinel configuration converges to a new one, client C1 will be able to load a valid configuration and continue with the new master page. Free lunches, however, are not. With this clarification, if the two slaves are down, the master stops accepting writes. It's a compromise. Example 3: Sentinel in client boxes Sometimes we only have two KeyDB boxes, one for the master and one for the slave. The configuration in example 2 is not viable in this case, so we can take a look back at the following where sentinels are located, where the clients are: +----+ | M1 |----+ ----| R1 | | | | | | | +----+ | +----+ | +------------+------------+ | | | | | | +----+ +----+ +----+ | 1(1)(a)(i) S2 | | S3 | +----+ +----+ +----+ Configuration: Quorum = 2 In this setting, the Sentinels view is the same as for clients: if the main is reachable by most clients, it's fine. C1, C2, C3 here are generic clients, it does not mean that C1 identifies one client connected to KeyDB. It is more likely to be something like an application server, a Rails application, or something like that. If the field where the M1 and S1 are running fails, the failure happens without problems, but it is easy to see that different network partitions will result in different behaviors. For example, sentinel will not be able to set up if the network between clients and KeyDB servers is disconnected because keydb master and slave will be as unavailable. Note that if C3 gets split with the M1 (hardly possible with the network described above, but more likely possible with different layouts, or because of a failure on the software layer), we have a similar problem as described in example 2, with the difference that here we have no way to break the symmetry, because there is only a slave and master , so the master can not stop accepting questions when it is disconnected from its slave, otherwise the master would never have been available during the failure of the Slav. So this is a valid setting, but the setup in example 2 has advantages such as ha the KeyDB system runs in the same boxes as KeyDB itself, which may be easier to manage, and the ability to put tied to the amount of time a master in a minority area can receive writes. Example 4: Client-side sentinel with fewer than three clients The settings described in Example 3 cannot be used if there are not enough three client-side fields (for example, three Web servers). In this case, we must remit to a mixed setting such as the following: +----+ +----+ | M1 |----+ ----| Article 20 01 02 — Sub-total | | S2 | +----+ | +----+ | +------+-----+ | | | | +----+ +----+ | It is also the first time that a member of the public has been 1 S3 | | S4 | +----+ +----+ Configuration: Quorum = 3 This is similar to the setting in example 3, but here we run four Sentinels in the four fields available to us. If the main M1 is not available three more Sentinels will perform a failover. Theoretically, this setting works by deleting the field where C2 and S4 are running, and setting the quorum to 2. However, it is unlikely that we want HA in the KeyDB side without high availability in our application layer. Sentinel, Docker, NAT, and possible problems Docker uses a technique called port mapping: programs running inside Docker containers can be exposed with a different port compared to the one the program believes it is using. This is useful for running multiple containers using the same ports at the same time in the same Docker is not the only software system where this happens, there are other NETWORK ADDRESS TRANSLATION settings where ports can be remapped, and sometimes not ports, but also IP addresses. The execution of ports and addresses creates problems with sentinel in two ways: Sentinel auto-discovery of other Sentinels no longer works because it is based on hello messages where each Sentinel communicate on which port and IP address you are listening to to connect. Sentinels, however, have no way to understand that the address or port is remapped, so it communicates information that is not correct for other Sentinels to connect to. Slaves are listed in the INFO keydb master output in a similar way: the address is detected by the master checking remote peer TCP connection, while the port is advertised by the slave itself during the handshake, but the port may be wrong for the same reason as exposed in point 1. Since sentinels auto detect slaves using masters INFO output information, detected slaves will not be achievable, and sentinel will never be able to failover master, because there are no good slaves in terms of the system, so currently there is no way to track with sentinel file masters and slaves instances deployed with the Docker if you do not order the Docker to map port 1:1. For the first problem, in case you want to run a set of Sentinel instances using Docker with redirected ports (or any other NAT settings, where the ports are remapped), you can use the following two sentinel configuration directives in order to force sentinel to notify a specific set of IP and port: sentinel announce-ip <ip>sentinel announce-port <port>Note that Docker has the option to run in host mode (check --net = host option for more information). This should cause no problems because the ports are not remapped in this setting. Quick Guide In the next sections of this document, all details about the Sentinel API, configuration, and semantics will be gradually covered. However, for people who want to play with the system ASAP, this part is a tutorial that shows how to configure and interact with 3 instances of Sentinel. Here we assume that the instances are performed in port 5000, 5001, 5002. We also assume that you are running keydb master in port 6379 with slave running on port 6380. We will use IPv4 loopback address 127.0.0.1 everywhere during the tutorial, assuming you use the simulation on your personal computer. Three sentinel configuration files should look like this: port 5000 sentinel monitor mymaster 127.0.0.1 6379 2 sentinel down-after-milliseconds mymaster 5000 sentinel failover-timeout mymaster 60000 sentinel parallel synchronizes mymaster 1 The other two configuration files will be identical, but using 5001 and 5002 as port numbers. A few things to note about the above configuration: The main set is </port> </ip>mymaster. Identifies the master and his slaves. Since each master file has a different name, sentinel can track different sets of masters and slaves at the same time. The quorum has been set to 2 (the last argument of the Sentinel Monitor Configuration Directive). The value after milliseconds is 5000 milliseconds, which is 5 seconds, so the masters will be revealed as failing once we get no response from our ping at this time. Once you run three Sentinels, you will see several messages that log in, such as: + monitor master mymaster 127.0.0.1 6379 Quorum 2 This is a Sentinel event, and you can receive this kind of event via Pub/Sub if you log on to the event name as shown later. Sentinel generates and records various events during failure detection and failure. 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: $keydb-cli -p 5000 127.0.0.1:5000> sentinel master mymaster 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 3 0) 1 31) num-slaves 32) 1 33) num-other-sentinels 34) 2 35) quorum 36) 2 37) failover-timeout 38) 60000 39) parallel-syncs 40) 1 As you can see , prints some master information. There are a few that are particularly interesting to us: num-other-sentinels is 2, so we know the Sentinel has already detected two more Sentinels for this gentleman. If you check the logs you will see the +sentinel events generated. the flag is just a master. If the gentleman was down we could expect to see s_down or o_down flag as well here. num-slaves is correctly set to 1, so the Sentinel also found that it is attached slave to our master. In order to explore more about this case, you may want to try the following two commands: SENTINEL Slaves Mymaster SENTINEL Sentinels Mymaster First will provide similar information about slaves associated with the master, and the second about other Sentinels. Obtaining the address of the current master As mentioned above, sentinel also functions as a configuration provider for clients who want to connect to a set of masters and slaves. Because of possible failures or reconfiguncations clients have no idea who is currently the active master for a given set of instances, so sentinel export APIs ask this question: 127.0.0.1:5000> SENTINEL get-master-addr-by-name mymaster 1) 127.0.0.1 2) 6379 Testing failover this point our deployment toy sentinel is ready for testing. We can only kill our master and see if the configuration changes. To do this we can only do this: keydb-cli -p 6379 DEBUG sleep 30 This command will make our master already achievable, sleeping for 30 seconds. It basically simulates a master hanging for a reason. If you check the Sentinel logs, you should be able to see a lot of action: Each Sentinel detects that the master page is down with the +sdown event. This event later escalated to +odown, which means that more Sentinels agree that the master is not achievable. Sentinels are voting for the Sentinel, which will launch the first attempt at failure. Failover happens. If you ask again what is the current main address for mymaster, finally we should get another answer this time: 127.0.0.1:5000> SENTINEL get-master-addr-by-name mymaster 1) 127.0.0.1 2) 6380 Not yet so good ... At this point you can jump to create a sentinel deployment or you can read more to understand all sentinel commands and internal. The Sentinel API Sentinel provides an API to check its status, check the health of monitored masters and slaves, subscribe to specific notifications, and reconfigulate the Sentinel at the time of operation. By default, sentinel runs by using TCP port 26379 (note that 6379 is a normal KeyDB port). Sentinels receive commands using the KeyDB protocol so that you can use keydb-cli or any other unmodified KeyDB client in order to talk to sentinel. It is possible to directly query sentinel to check what is the status of monitored KeyDB instances from his perspective to see what other Sentinels he knows and so on. Alternatively, using Pub/Sub, it is possible to receive push style notifications from Sentinels, every time an event happens, such as a failover, or instances entering an error condition, and so on. Sentinel commands The following is a list of accepted commands that do not apply to commands used to modify sentinel configurations that are included later. PING This command simply returns PONG. Sentinel Masters View a list of monitored masters and their status. Sentinel master <master name=>View status and master information. Sentinel slaves <master name=>View a list of slaves for this master, and their status. Sentinel <master name=>indicators Show a list of sentinel instances for this master and their status. SENTINEL get-master-addr-by-name <master name=>Return ip and master port number with this name. If a failure is in progress or successfully completed for this master, it returns the address and port of the supported slave. Sentinel reset <pattern>This command restores all masters with the corresponding name. The pattern argument is a glob-style pattern. The reset process erases all previous states in the master (including the backup state) and removes each slave a</pattern> </master> </master> </master> </master> already appeared and associated with the master. Sentinel failover Force failover, as if the main was not achievable, and without asking for permission <master name=> with other Sentinels (however, the new version of the configuration will be published so that other Sentinels will update their configurations). SENTINEL ckquorum <master name=> Make sure that the current Sentinel configuration is capable of achieving the quorum required for master failure and most needed to enable failure. This command should be used in monitoring systems to check that sentinel deployment is in order. Sentinel flushconfig Force Sentinel override its configuration on the disk, including the current state of the Sentinel. Normally sentinel overrides the configuration every time something changes in its state (in the context of a subset of the state that persists on the disk through restart). However, sometimes the configuration file may be lost due to operation errors, disk failures, package upgrade scripts, or configuration administrators. In these cases, there is a handy way to force sentinel to overwrite the configuration file. This command works even if the previous configuration file is completely missing. Reconfiguring sentinel at Runtime Starting with KeyDB version 2.8.4, Sentinel provides an API to add, remove, or change the configuration of a given master page. Note that if you have multiple sentinels, you should apply the changes to all of your instances of the KeyDB Sentinel to work correctly. This means that reconfiguring one sentinel does not automatically spread changes to other sentinels on the network. The following is a list of SENTINEL sub-report used to update the Sentinel instance configuration. SENTINEL MONITOR <name> <ip> <port> <quorum> This command informs sentinel to start monitoring the new master with the specified name, ip, port, and quorum. It is identical to the sentinel monitor configuration directive in the sentinel.conf configuration file, with the difference that you cannot use the host name as ip, but you must provide an IPv4 or IPv6 address. SENTINEL REMOVE <name> is used to remove the designated master: the master will no longer be monitored and will be completely removed from the internal state of sentinel, so it will no longer be listed by SENTINEL masters and so on. SENTINEL SET <name><option><value>The SET command is very similar to the CONFIG SET KeyDB command and is used to change the configuration parameters of a particular master page. It is possible to specify multiple option/value pairs (or none at all). All configuration parameters that can be configured using sentinel.conf are also configurable by using the SET command. This is an example of a SENTINEL SET command to modify down-after-milliseconds of the master configuration with object-cachename:</value></name></name></quorum></port></ip></name></master></master>SET objects-cache-master down-after-milliseconds 1000 As mentioned earlier, SENTINEL SET can set all configuration parameters that are set in the startup configuration file. In addition, it is possible to change only the main quorum configuration without deleting 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 because SENTINEL MASTER provides all configuration parameters in a simple analyze format (such as array/value pairs array). Add or remove sentinels Adding a new sentinel to a deployment is a simple process because of the automatic discovery mechanism implemented by sentinel. All you have to do is run a new Sentinel configured to monitor the currently active master. Within 10 seconds the Sentinel will get a list of other Sentinels and a set of slaves attached to the master. If you need to add multiple Sentinels at once, it is suggested to add one by one, 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 most can only be achieved on one side of the partition, if the failure chances should happen in the process of adding new Sentinels. This can be easily achieved by adding each new Sentinel with a 30 second delay, and in the absence of network partitions. At the end of the process, you can use the SENTINEL MASTER mastername command to check that all sentinels match the total number of sentinels monitored by the master. Removing sentinels is a little tricky: Sentinels never forget to see sentinels anymore, even if they are not reachable for a long time because we don't want to dynamically change most needed to enable failover and create new configuration numbers. So in order to remove the Sentinel the following steps should be done in the absence of network partitions: Stop the Sentinel Sentinel process that you want to delete. Send sentinel RESET * command to all other sentinel instances (instead of * you can use the exact master name if you want to restore only one master). One by one, waiting at least 30 seconds between instances. Make sure that all sentinels match the number of sentinels currently active by checking the output of the SENTINEL MASTER master of each sentinel. The removal of the old master or the unattainable slaves sentinels will never forget the slaves of that master, even if they are unattainable for a long time. This is useful because Sentinels should be able to correctly reconfigure the returning slave after a network partition or event failure. Moreover, after failure, failed over the master is practically added as a slave to the new master, in this way it will be reconfigured so that with the new master as soon as it becomes available again. However, sometimes you want to remove a slave (it may be an old master) forever from the list of slaves monitored by sentinels. To do this, you must send the SENTINEL RESET principal command to all sentinels: within the next 10 seconds, they will refresh the slave list and add only those listed as correctly replicating from the current INFO main output. Pub/Sub Administration client can use Sentinel as it was a KeyDB compatible Pub/Sub server (but you can't use PUBLISH) in order to log in or PSUBSCRIBE on channels and get notification of specific events. The channel name is the same as the event name. For example, a channel named +sdown receives all notifications related to instances entering an SDOWN (SDOWN means that the instance is no longer reachable in terms of the Sentinel you are querying) condition. To get all messages simply log in using PSUBSCRIBE *. The following is a list of channels and message formats that you can receive using this API. The first word is the channel/event name, the rest is the data format. Note: if instance details are entered, it means that the following arguments are listed to identify the destination instance: @ The part identifying the master (from @až to the end) is optional and is specified only if that the instance is not alone <instance-type> <name> <ip> <port> <master-name> <master-ip> <master-port>se master. +reset-master -- Master has been reset. +slave -- A new slave has been detected and <instance details=> <instance details=>attached. +failover-state-reconf-slaves -- Failure status <instance details=>changed to reconf-slaves state. +failover-detected -- Failover was started by another Sentinel or any other external entity was detected (attached slave turned into <instance details=>master). +slave-reconf-sent -- The sentinel leader sent <instance details=>SLAVEOF command to this instance in order to reconfigure it for a new slave. +slave-reconf-inprog <instance details=>-- The slave is reconfigured to show that it is a slave to the new master ip:port pair, but the synchronization process is not yet complete. +slave-reconf-done -- Slave is now synchronized with the new master. -dup-sentinel -- One or more sentinels for the specified master have been removed as duplicates (this happens, for example, when you restart a Sentinel instance). +sentinel -- The new sentinel for this master has been detected and connected. +sdown -- The specified instance is now subjectively down state. -sdown -- The specified instance is no longer <instance details=> <instance details=> <instance details=> <instance details=> <instance details=>subjective down state. +odown <instance=>-- The specified instance is now objectively down state. -odown <instance=>-- Specified nie je</instance> </instance> </instance> </instance> </instance> </instance> </instance> </instance> </instance> </instance> </instance> </instance> </master-port> </master-ip> </master-name> </port> </ip> </name> </instance-type> </instance-type> objectively down state. +new-epoch -- Current <instance details=>epoch has been updated. +try-failover -- New failure in progress pending election <instance details=>majority. +elected-leader <instance details=>-- Won the election for a specified epoch, can make a failover. +failover-state-select-slave <instance details=>-- The new backup state is a select-slave: we are trying to find a suitable slave to <instance details=>promotion. At the moment we will try after some time, but probably it will change and the state machine will interrupt the failover at all in this case. selected-slave -- We found a designated good slave on <instance details=>support. failover-state-send-slaveof-noone -- We are trying to reconfigure the promoted slave as master, waiting for <instance details=>to switch. failover-end-for-timeout <instance details=>-- Failover terminated for timeout, slaves will eventually be configured to replicate with the new master as well. failover-end <instance details=>-- Failover completed with success. It seems that all slaves are reconfigured to be replicated with the new master. switch-master <master name=> <oldip> <oldport> <newip> <newport>-- The main new IP and address is entered after reconfigume. This is a message that most external users are interested in. -tilt -- Tilt mode is over. State Processing -BUSY Error -BUSY is returned by an instance of KeyDB when the Lua script is running longer than the configured Lua script timeout. When this happens before you run a failure through the KeyDB Sentinel it tries to send a SCRIPT KILL command that succeeds only if the script was read-only. If the instance is still in an error state after this attempt, it will not disappoint in the end. Slaves priority KeyDB instances have a configuration parameter called slave-priority. This information is exposed to keydb slave instances in their INFO output, and the Sentinel is used in order to select a slave among those that can be used in order to failover the master: If the slave priority is set to 0, the slave is never promoted to master. Slavs with a lower priority number prefer the Sentinel. For example, if the slave S1 is in the same data center of the current master, and another slave S2 in another data center, it is possible to set the S1 with priority 10 and S2 with priority 100, so if the master fails and both S1 and S2 are available, S1 will be preferred. For more information about how to select slaves, see Choosing slaves and prioritize this documentation. Sentinel and KeyDB authentication When the master is configured so,</newport> </newip> </oldport> </oldip> </master> </instance> </instance> </instance> </instance> </instance> </instance> </instance> </instance> </instance> to require a password from clients as a security measure, they must also be aware of this password in order to verify with the master and and connection master-slave used for asynchronous replication protocol. This is achieved by using the following configuration guidelines: requirepass in the master page, in order to set a authentication password, and to make sure that the instance does not process requests to unautentiated clients. Masterauth in slaves to make slaves verify with the master in order to correctly replicate the data from it. When sentinel is used, there is not a single master, because after failover slaves can play the role of masters, and the old masters can be reconfigured to act as slaves, so what you want to do is set the above guidelines in all your cases, both masters and slaves. This is also usually a reasonable setting because you don't want to protect data only in the master page that have the same data accessible in slaves. However, in an unusual case, when you need a slave that is accessible without authentication, you can still do this by setting up slave priority zero to prevent this slave from being promoted to master, and configuring in this slave only masterauth directive, without using the requirepass directive, so the data will be readable by unverified clients. In order for sentinels to connect to KeyDB server instances when configured with requirepass, Sentinel configuration must contain sentinel auth-pass in the format: sentinel auth-pass Configuring Sentinel instances with authentication You can also configure the Sentinel instance itself to require client authentication using the AUTH command, but this feature is only available starting with <master-group-name> <pass>KeyDB 5.0.1. To do so, you just need to add the following configuration directive to all your Sentinel instances: requirepass your_password_here When configured in this way, sentinels will do two things: A password from clients will be required to send sentinel commands. This is obvious because this is how such a configuration directive generally works in keydb. Additionally, the same password configured to access the local Sentinel will be used in this Sentinel instance in order to authenticate to all other instances of sentinel it connects to. This means that you have to configure the same requirepass password in all instances of sentinel. This way any Sentinel can talk to any other Sentinel without the necessary configuration for each Sentinel password to access all other Sentinels, it would be very impractical. Before you use this configuration, make sure that your client library is able to send the AUTH sentinel command to the instance. Sentinel client implementation sentinel requires explicit client support if the system is configured to run a script that performs transparent redirection of all requests to a new root instance (virtual IP or other similar systems). Topic</pass> </master-group-name>library implementation is included in the Sentinel client guidance document. More advanced concepts In the following sections, we will cover a few details on how sentinel works, without re relling on the implementation details and algorithms that will be included in the final part of this document. The SDOWN and ODOWN failure states The KeyDB Sentinel has two different terms being down, one is called a subjective down condition (SDOWN) and is a condition down that is local to that Sentinel instance. Another is called objective down condition (ODOWN) and is achieved when enough Sentinels (at least a number configured as a quorum parameter monitored by the master) have an SDOWN condition, and get feedback from other Sentinels by using SENTINEL's master-down-by-addr command. From a Sentinel point of view, the SDOWN condition is reached when it does not receive a valid response to ping requests for the number of seconds specified in the configuration as an is-master-down-after-milliseconds parameter. The acceptable answer to PING is one of the following: PING replied with +PONG. PING replied with a -Loading error. Ping replied with a -MASTERDOWN error. Any other response (or no response) shall be considered invalid. Note, however, that a logical master who advertises himself as a slave in the INFO output is considered down. Note that SDOWN requires that no acceptable response is received for the entire interval configured, so for example, if the interval is 30,000 milliseconds (30 seconds) and we receive an acceptable ping response every 29 seconds, the instance is considered working. SDOWN is not enough to trigger a failure: this means only one Sentinel believes a KeyDB instance is not available. To trigger a failure, the ODOWN state must be reached. To switch from SDOWN to ODOWN no strong consensus algorithm is used, but only a form of gossip: if a given Sentinel receives a message that the master does not work from sufficient Sentinels within a given time range, SDOWN is promoted to ODOWN. If this confirmation is later missing, the flag shall be erased. A stricter permission that uses the actual majority is required in order to actually trigger a failure, but no failure can be triggered without reaching the ODOWN state. The ODOWN condition applies only to masters. For other kinds of cases the Sentinel does not require action, so the ODOWN state is never achieved for slaves and other sentinels, but only SDOWN's. However, SDOWN also has semantic implications. For example, a slave in an SDOWN state is not selected to support 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 exchange messages. However, you do not need to configure a list of other sentinel addresses in each instance of Sentinel to run how sentinel uses KeyDB Pub/Sub capabilities in order to discover other Sentinels that are monitoring the same masters and slaves. This feature is implemented by sending hello messages to a channel called __sentinel__:hello. Similarly, you do not need to configure what is the list of slaves attached to the master page, as the Sentinel will automatically discover this KeyDB query list. Each Sentinel publishes a message for each monitored master and slave pub / sub channel __sentinel__:hello, every two seconds, announces its presence with IP, port, runid. Each Sentinel is ordered on the Pub/Sub Channel __sentinel__: hello to every master and slave, looking for unknown sentinels. When new indicators are found, they are added as indicators of this master. Hello, the messages also contain the full current configuration of the master. If the receiving Sentinel has a configuration for that master that is older than received, it will be updated immediately to the new configuration. Before adding a new sentinel to master sentinel always checks whether there is a sentinel with the same runid or the same address (ip and port pair). In this case, all matching indicators are removed and a new one is added. Sentinel reconfiguration of instances outside the failover procedure Although no failure is in progress, Sentinels always tries 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 the evil master will be reconfigured to be replicated with the right master. Sentinels reconfigure slaves, incorrect configuration must be observed for some time, which is greater than the period used to broadcast new configurations. This prevents Sentinels with outdated configurations (for example, because they have just returned from the partition) from attempting to change the slave configuration before receiving the update. Note also how semantics always trying to save the current configuration makes failover more resistant to partitions: Masters failed are reconfigured as slaves when they return available. Slaves split during partition are reconfigured once achievable. An important lesson to remember about this section is: Sentinel is a system where each process will always try to save the last logical configuration to a set of monitored instances. Selection of slaves and priority When the Sentinel instance is ready to fail because the master is in an ODOWN state and the Sentinel has received permission to fail from most known Sentinel instances, it is necessary to select the appropriate slave. The selection process of Slovan evaluates the following information about slavs: Disconnect time from master. Priority of the Slav. Replication offset processed. Run id. Slave, which was found that disconnected from the main computer more than ten times the configured main timeout (down-after-milliseconds option) plus the time when the master server is not available even in terms of the sentinel that makes the failure is considered not suitable for the backup time and is skipped. In stricter terms, a slave whose output INFO indicates that he is disconnected from the master for more than: (down-after-milliseconds * 10) + milliseconds_since_master_is_in_SDOWN_state is considered unreliable and is completely ignored. The selection of the Slav takes into account only slaves who have passed the above test and sorts it on the basis of the above criteria, in the following order. Slaves are sorted by slave-priority as configured in the redis.conf file instance of KeyDB. Priority will be given to a lower priority. 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. If multiple slaves have the same priority and processed the same data from the master, another check will be made, selecting a slave with lexicographically smaller run IDs. With a lower run ID is not a real advantage for the slave, but it is useful to make the process of selecting slaves more deterministic, instead of resorting to selecting a random slave. KeyDB masters (which can be converted into slaves after failover), and slaves, everything must be configured with slave-priority if there are machines to be strongly preferred. Otherwise, all instances can run with the default run ID (which is the suggested setting because it is much more interesting to select a slave replication offset). A KeyDB instance can be configured with a special slave zero priority so that Sentinels never selects it as the new master. However, a slave configured in this way will still reconfigure sentinels in order to replicate with the new master after failover, the only difference is that he will never become a master himself. Algorithms and internal in the following sections we will examine the details of sentinel behavior. It is not necessarily necessary for users to be aware of all the details, but a deep understanding of sentinol can help deploy and operate the Sentinel in a more efficient way. Quorum Previous sections have shown that each sentinel-monitored master is associated with the configured quorum. Specifies the number of Sentinel processes that must agree on the unavailability or error status of the master to trigger a failure. However, when you run a backup share, at least most sentinels must enable sentinel to fail in order for the failure to actually take place. The Sentinel will never fail in an area where there is a minority of Sentinels. Let's try to make things a little clearer: Quorum: the number of processes sentinel an error condition must be detected in order for the master to be marked as ODOWN. The failure is triggered by an ODOWN state. When a failure is started, a sentinel that attempts to fail is required to request permission for most sentinels (or more than most if the quorum is set to a number greater than most). The difference may seem subtle, but it is actually quite easy to understand and use. For example, if you have 5 instances of sentinel and the quorum is set to 2, the backup failure will start as soon as 2 Sentinels believe that the master is not achievable, but one of the two Sentinels will only be able to fail if it receives permission from at least 3 Sentinels. If instead the quorum is configured to 5, all Sentinels must agree to the status of the major error, and permission from all Sentinels is required to fail. This means that the quorum can be used to tune the sentinel in two ways: if the quorum is set to a value smaller than most sentinels we deploy, we are basically making the Sentinel more sensible to handle failures, triggering failure as soon as only a minority of sentinels are unable to speak to the master. If the quorum is set to higher than most sentinels, we make the Sentinel capable of failing only when there are a very large number (larger than most) of well-connected sentinels who agree that the captain is down. Configuring epoch sentinels require you to obtain permissions from most in order to run a failover for several important reasons: When sentinel is enabled, it gets a unique configuration epoch for the master to fail over. This is the number that will be used for the new configuration version after the failure is complete. Since most agreed that the version was assigned to the sentinel, no other Sentinel could be used. This means that each configuration of each failover is a version with a unique version. Let's see why it's so important. In addition, sentinels have a rule: if the Sentinel voted another Sentinel for the failure of a given master, it will wait some time to try to failover the same master again. This delay is a failover-timeout you configure in sentinel.conf. This means that sentinels will not try to failover the same master at the same time, the first to ask for permission will try if another tries after some time, and so on. The KeyDB Sentinel guarantees the vibrability of the property that if most Sentinels are able to speak, eventually one will be eligible for failover if the master is down. KeyDB Sentinel also guarantees the security features that each Sentinel will failover the same master using a different configuration of the epoch. Propagation configuration When sentinel successfully fails, it starts transmitting new configuration so that other Sentinels will update their master information. In order for a backup failure to be considered successful, the sentinel is required to be able to send slaveof NO ONE to the selected slave and that the transfer to the master is later observed in the master's information output. At this point, even if the reconfiguration of slaves is in progress, the failure is considered successful, and all Sentinels are required to start reporting the new configuration. The way the new configuration is spreading is why we need each sentinel failover to be enabled with a different version number (configuration epoch). Each Sentinel continuously broadcast their version of the master configuration using keydb pub/sub messages, both in master and all slaves. At the same time, all Sentinels wait for messages to see what is the configuration advertised by the other Sentinels. Configurations are broadcast in __sentinel__:hello Pub/Sub channel. Because each configuration has a different version number, a larger version always wins over smaller versions. So, for example, the configuration for master mymaster to start with all sentinels believing master is on 192.168.1.50:6379. This configuration has version 1. After some time sentinel is eligible for failover with version 2. If the failure is successful, a new configuration starts transmitting, say, 192.168.1.50:9000 with version 2. All other instances will see this configuration and update their configuration accordingly because the new configuration has a larger version. This means that the Sentinel guarantees a second liveness feature: a set of sentinels that are able to communicate all converge on the same configuration with a higher version number. Basically, if the network is split, each partition will converge on a higher local configuration. In the special case of no partitions, there is one partition and each Sentinel will agree to the configuration. Consistency within keydb sentinel configurations are ultimately consistent, so each partition converges on the higher configuration available. However, in a real-world system using sentinel there are three different players: KeyDB instances. Sentinel instances. Clients. In order to define the behaviour of the system, we need to consider all three. This is a simple network where there are 3 nodes, each running KeyDB instances, and sentinel instances: +-------------+ | Sentinel 1 |----- client A | KeyDB 1 (M) | +-------------+ | | +-------------+ | +------------+ | Sentinel 2 |-----+-- // ----| Sentinel 3 |----- client B | KeyDB 2 (S) | | KeyDB 3 (M)| +-------------+ +------------+ In this system the original state was that KeyDB 3 was a master, while KeyDB 1 and 2 were slaves. An area has occurred the old master. Sentinels 1 and 2 began the failover support of Sentinel 1 as the new master. Sentinel properties ensure that Sentinel 1 and 2 now have a new configuration for the master. However Sentinel 3 still has an old configuration because it lives in a different area. We know that Sentinel 3 will get its configuration updated when it will treat the network partition, but what happens during the partition if there are clients divided with the old master? Clients will still be able to write KeyDB 3, old master. When the partition is returned, KeyDB 3 is overdue on slave KeyDB 1, and all data written during the partition is lost. Depending on the configuration you may want or not, this scenario will happen: If you use KeyDB as a cache, it may be useful that Client B is still able to write to the old master, even if his data will be lost. If you are using KeyDB as a store, this is not good and you must configure the system in order to partially prevent this problem. Because KeyDB is replicated asynchronously, there is no way to how to completely prevent data loss in this scenario, but you can bound the differences between KeyDB 3 and KeyDB 1 using the following KeyDB configuration options: min-slaves-to-write 1 min-slaves-max-lag 10 With the above configuration (see self-commented redis.conf example in the distribution of KEYDB for more information) KeyDB instance when acting as master , stops receiving writes if you can not write at least 1 slave. Since replication is asynchronously not being able to write actually means that the slave is either disconnected or is not sending us asynchronously acknowledging more than the specified max-lag number of seconds. Using this KeyDB 3 configuration in the example above will become unavailable after 10 seconds. When the partition is reassued, sentinel 3 configuration will converg with the new one, and client B will be able to load a valid configuration and continue. Generally keydb + sentinel as a whole are ultimately a consistent system where the merge function is the last failover wins, and data from the old masters are discarded to replicate the data of the current master, so there is always a window for loss acknowledged writes. This is caused by KeyDB asynchronous replication and the tossing nature of the virtual merge system function. Note that this is not a restriction of the Sentinel itself, and if you organized a failover with a strongly consistent replicated state of the machine, the same features will still apply. There are only two ways to avoid losing acknowledged writes: Use synchronous replication (and the correct consensus algorithm to run the replicated state of the machine). Finally, use a consistent system where you can merge different versions of the same object. KeyDB is currently unable to use any of the above systems, and currently outside the development goals. However there are proxies to implement Solution 2 on top of KeyDB stores such as SoundCloud Roshi, or Netflix Dynomite. The Sentinel permanent status of the Sentinel state persists in the sentinel configuration file. For example, every time a new configuration is received or created (leader Sentinels) for a master page, the configuration persists on the disk along with the epoch configuration. This means that it is safe to stop and restart Sentinel processes. Tilt mode KeyDB Sentinel is heavily dependent on computer time: for example, in order to understand whether an instance is available remembers the time of the last successful response to the PING command, and compares it with the current time to understand how old it is. However, if your computer's time changes unexpectedly, or if your computer is very busy or the process is blocked for some reason, the sentinel might start behaving in an unexpected way. TILT mode is a special protection mode that can enter the sentinel when something special is detected that can reduce system reliability. The Sentinel timer break is usually called 10 times per second, so we expect more or less 100 milliseconds to escape between two calls to interrupt the timer. What sentinel does is register the previous timer interrupt it was called, and compare it with the current call: if the time difference is negative or unexpectedly large (2 seconds or more) tilt mode is specified (or if the output from TILT mode has already been postponed). When it is in TILT mode, the Sentinel will continue to monitor everything, but: It will stop acting at all. It begins to respond negatively to sentinel's master-down-by-addr requests as the ability to detect failures is no longer trusted. If everything appears normal for 30 seconds, TILT mode will exit. Note that in some way TILT mode could be replaced using the monotonous clock API that many cores offer. However, it is still unclear whether this is a good solution, as the current system avoids problems if the process is only suspended or not carried out by the planner for a long time. PLEASE NOTE: This document is a draft and the guidelines it contains may change in the future as the Sentinel project develops. Instructions for keydb clients with KeyDB Sentinel Sentinel Sentinel support is a monitoring solution for keydb instances that handles automatic failure of KeyDB masters and discovery services (who is the current master for that group of instances?). Since sentinel is responsible for reconfiguring instances during failovers, and providing configurations to clients connecting keydb masters or slaves, clients require to have explicit support for keydb sentinel. This document focuses on KeyDB clients of developers who want to support Sentinel in implementing their clients with the following Automatic configuration of clients via Sentinel. Improve security keydb Sentinel automatic failure. For details on how the KeyDB Sentinel works, see the KeyDB documentation because this document contains only the information needed for KeyDB client developers, and readers are expected to be familiar with the way the KeyDB Sentinel works. KeyDB discovery services via the Sentinel KeyDB Sentinel identify each master named as statistics or cache. Each name actually identifies a group of instances, composed of a master and a variable number of slaves. The keydb master address that is used for a specific purpose inside the network may change after events such as automatic failure, manually triggered failover (for example, in order to upgrade KeyDB instances) and other reasons. Normally KeyDB clients have some hard-coded configuration that specifies the KeyDB address of the main instance on the network as the IP address and port number. However, if the main address changes, manual intervention is required in each client. A Sentinel-supporting KeyDB client can automatically discover the keydb master address from the primary name by using the KeyDB Sentinel. So instead of firmly behind the encoded IP address and port, the Sentinel-supporting client should optionally be able to accept it as input: a list of ip:port pairs pointing to known instances of Sentinel. The name of the service, such as cache or timeline. This is the procedure that the client should follow to obtain the main address starting with the Sentinels list and the service name. Step 1: Connecting to the first Sentinel Client should iterate the sentinel address list. For each address should try to connect to the Sentinel, using a short time limit (in order of several hundred milliseconds). For errors or timeouts, the next Sentinel address should be tested. If all Sentinel addresses have been tried without success, the error should be returned to the client. The first sentinel response to a client request should be at the top of the list, so the next time we reconnect, we'll first try the sentinel that was achievable on the previous connection attempt, minimizing latency. Step 2: Request the main address Once a link to sentinel is established, the client should run the following command again on sentinel: SENTINEL get-master-addr-by-name master-name Where the master name should be replaced with the actual service name specified by the user. This call can result in one of the following two responses: ip:port pair. Null answer. That means the Sentinel doesn't know this gentleman. If ip:port is accepted, this address should be used to connect to the KeyDB master server. Otherwise, if a zero response is received, the client should try the next Sentinel in the list. Step 3: Call the ROLE command in the target instance as soon as the client has discovered instance, you should try to connect to the master and call the ROLE command to verify that the instance role is in fact a master. If role commands are not available (this was introduced in KeyDB 2.8.12), the client may reseal the info replication command to analyze the task: The output field. If the instance is not a master as expected, the client should wait a short amount of time (several hundred milliseconds) and should try again from step 1. Reconnection processing After you resolve the service name to the main address and the connection is created with the main instance keydb, each time a reconnection is required, the client should resolve the retry address by using sentinels restarting from step 1. For example, the Sentinel should contact the following cases again: If the client connects after a timeout or socket error. If the client reconnects because it has been explicitly closed or reconnected by the user. In the above cases, and in any other case, when the client has lost contact with the KeyDB server, the client should resolve the root address again. Sentinel failover disconnection Starting with KeyDB 2.8.12, when the KeyDB Sentinel changes the configuration of an instance, such as slave support to master, demote master replicate to new master after failure, or simply change the root address of an outdated slave instance, it sends the client kill type to a normal command instance to make sure all clients are disconnected from the reconfigure instance. This forces clients to resolve the root address again. If the client will contact sentinel with information that is not yet updated, the authentication keydb instance of the task through the ROLE command fails, allowing the client to determine that the contacted Sentinel has outdated information, and will try again. Note It is possible that the outdated master will be returned online at the same time the client contacts the outdated Sentinel instance so that the client can connect with the outdated master, and yet the role output will match. However, when the master is back the Sentinel attempts to degrade to a slave, triggering a new disconnection. The same reasoning applies to the connection to musty slaves who reconfigure themselves to replicate with another master. Connecting to slaves Sometimes clients are interested in joining slaves, for example, in order to scale reading requests. This protocol supports connecting to slaves by modifying step 2 slightly. Instead of calling the following command: SENTINEL get-master-addr-by-name master-name Clients should call instead: SENTINEL slaves master-name In order to obtain a list of instances of slaves. Symmetrically, the client should verify by using the ROLE command that the instance is actually a slave to avoid scaling read queries with the master. Connection funds for clients when you reconnect a single connection, the sentinel should be contacted again, and if the main address is changing, all existing connections should be closed and connected to the new address. Error reporting The client should correctly return the information to the user in case of errors. Specifically: If no sentinel can be contacted (so that the client has never been able to get a response to sentinel get-master-addr-by-name), an error that clearly states that the KeyDB Sentinel is unreachable should be returned. If all sentinels in the pool responded with a null response, the user should be informed with an error that sentinels do not know this principal name. Sentinels list auto-renew Optionally after a successful response to get-master-addr-by-name, the client can update its internal list of sentinel nodes by following these steps: Get a list of additional Sentinels for this master by using the SENTINEL sentinels command <master-name>;. Add each ip:port pair is not yet in our list at the end of the list. There is no need for the client to be able to create a list of persistent custom configuration updates. The ability to upgrade in-memory representations of the Sentinels list may already be useful for improved reliability. Subscribe to Sentinel events to improve sentinel response documentation showing how clients can connect to sentinel instances by using Pub/Sub in order to subscribe to changes in keydb instance configurations. This mechanism can be used to speed up client reconfiguration, that is, clients can listen to Pub/Sub to know when a configuration change has occurred in order to run the three steps explained in this document in order to resolve the new keydb master (or slave) address. However, updating messages received through Pub/Sub should not replace the above procedure because there is no guarantee that the client is able to receive all update messages. Messages. </master-name>