Try Free

Redis latency problems troubleshooting

This document will help you understand what the problem could be if you are experiencing latency problems with Redis.

In this context *latency* is the maximum delay between the time a client issues a command and the time the reply to the command is received by the client. Usually Redis processing time is extremely low, in the sub microsecond range, but there are certain conditions leading to higher latency figures.

I've little time, give me the checklist

The following documentation is very important in order to run Redis in a low latency fashion. However I understand that we are busy people, so let's start with a quick checklist. If you fail following these steps, please return here to read the full documentation.

1. Make sure you are not running slow commands that are blocking the server. Use the Redis Slow Log feature to check this.
2. For EC2 users, make sure you use HVM based modern EC2 instances, like m3.medium. Otherwise fork() is too slow.
3. Transparent huge pages must be disabled from your kernel. Use `echo never > /sys/kernel/mm/transparent_hugepage/enabled` to disable them, and restart your Redis process.
4. If you are using a virtual machine, it is possible that you have an intrinsic latency that has nothing to do with Redis. Check the minimum latency you can expect from your runtime environment using `./redis-cli --intrinsic-latency 100`. Note: you need to run this command in *the server* not in the client.
5. Enable and use the Latency monitor feature of Redis in order to get a human readable description of the latency events and causes in your Redis instance.

In general, use the following table for durability VS latency/performance tradeoffs, ordered from stronger safety to better latency.

1. AOF + fsync always: this is very slow, you should use it only if you know what you are doing.
2. AOF + fsync every second: this is a good compromise.
3. AOF + fsync every second + no-appendfsync-on-rewrite option set to yes: this is as the above, but avoids to fsync during rewrites to lower the disk pressure.
4. AOF + fsync never. Fsyncing is up to the kernel in this setup, even less disk pressure and risk of latency spikes.

5. RDB. Here you have a vast spectrum of tradeoffs depending on the save triggers you configure.

And now for people with 15 minutes to spend, the details...

Measuring latency

If you are experiencing latency problems, you probably know how to measure it in the context of your application, or maybe your latency problem is very evident even macroscopically. However redis-cli can be used to measure the latency of a Redis server in milliseconds, just try:

```
redis-cli --latency -h `host` -p `port`
```

Using the internal Redis latency monitoring subsystem

Since Redis 2.8.13, Redis provides latency monitoring capabilities that are able to sample different execution paths to understand where the server is blocking. This makes debugging of the problems illustrated in this documentation much simpler, so we suggest enabling latency monitoring ASAP. Please refer to the Latency monitor documentation.

While the latency monitoring sampling and reporting capabilities will make it simpler to understand the source of latency in your Redis system, it is still advised that you read this documentation extensively to better understand the topic of Redis and latency spikes.

Latency baseline

There is a kind of latency that is inherently part of the environment where you run Redis, that is the latency provided by your operating system kernel and, if you are using virtualization, by the hypervisor you are using.

While this latency can't be removed it is important to study it because it is the baseline, or in other words, you won't be able to achieve a Redis latency that is better than the latency that every process running in your environment will experience because of the kernel or hypervisor implementation or setup.

We call this kind of latency **intrinsic latency**, and `redis-cli` starting from Redis version 2.8.7 is able to measure it. This is an example run under Linux 3.11.0 running on an entry level server.

Note: the argument `100` is the number of seconds the test will be executed. The more time we run the test, the more likely we'll be able to spot latency spikes. 100 seconds is usually appropriate, however you may want to perform a few runs at different times. Please note that the test is CPU intensive and will likely saturate a single core in your system.

```
$ ./redis-cli --intrinsic-latency 100
Max latency so far: 1 microseconds.
Max latency so far: 16 microseconds.
Max latency so far: 50 microseconds.
Max latency so far: 53 microseconds.
Max latency so far: 83 microseconds.
Max latency so far: 115 microseconds.
```

Note: redis-cli in this special case needs to **run in the server** where you run or plan to run Redis, not in the client. In this special mode redis-cli does not connect to a Redis server at all: it will just try to measure the largest time the kernel does not provide CPU time to run to the redis-cli process itself.

In the above example, the intrinsic latency of the system is just 0.115 milliseconds (or 115 microseconds), which is a good news, however keep in mind that the intrinsic latency may change over time depending on the load of the system.

Virtualized environments will not show so good numbers, especially with high load or if there are noisy neighbors. The following is a run on a Linode 4096 instance running Redis and Apache:

```
$ ./redis-cli --intrinsic-latency 100
Max latency so far: 573 microseconds.
Max latency so far: 695 microseconds.
Max latency so far: 919 microseconds.
Max latency so far: 1606 microseconds.
Max latency so far: 3191 microseconds.
Max latency so far: 9243 microseconds.
Max latency so far: 9671 microseconds.
```

Here we have an intrinsic latency of 9.7 milliseconds: this means that we can't ask better than that to Redis. However other runs at different times in different virtualization environments with higher load or with noisy neighbors can easily show even worse values. We were able to measure up to 40 milliseconds in systems otherwise apparently running normally.

Latency induced by network and communication

Clients connect to Redis using a TCP/IP connection or a Unix domain connection. The typical latency of a 1 Gbit/s network is about 200 us, while the latency with a Unix domain socket can be as low as 30 us. It actually depends on your network and system hardware.

On top of the communication itself, the system adds some more latency (due to thread scheduling, CPU caches, NUMA placement, etc ...). System induced latencies are significantly higher on a virtualized environment than on a physical machine.

The consequence is even if Redis processes most commands in sub microsecond range, a client performing many roundtrips to the server will have to pay for these network and system related latencies.

An efficient client will therefore try to limit the number of roundtrips by pipelining several commands together. This is fully supported by the servers and most clients. Aggregated commands like MSET/MGET can be also used for that purpose. Starting with Redis 2.4, a number of commands also support variadic parameters for all data types.

Here are some guidelines:

- If you can afford it, prefer a physical machine over a VM to host the server.
- Do not systematically connect/disconnect to the server (especially true for web based applications). Keep your connections as long lived as possible.
- If your client is on the same host than the server, use Unix domain sockets.
- Prefer to use aggregated commands (MSET/MGET), or commands with variadic parameters (if possible) over pipelining.
- Prefer to use pipelining (if possible) over sequence of roundtrips.
- Redis supports Lua server-side scripting to cover cases that are not suitable for raw pipelining (for instance when the result of a command is an input for the following commands).

On Linux, some people can achieve better latencies by playing with process placement (taskset), cgroups, real-time priorities (chrt), NUMA configuration (numactl), or by using a low-latency kernel. Please note vanilla Redis is not really suitable to be bound on a **single** CPU core. Redis can fork background tasks that can be extremely CPU consuming like BGSAVE or BGREWRITEAOF. These tasks must **never** run on the same core as the main event loop.

In most situations, these kind of system level optimizations are not needed. Only do them if you require them, and if you are familiar with them.

Single threaded nature of Redis

Redis uses a *mostly* single threaded design. This means that a single process serves all the client requests, using a technique called **multiplexing**. This means that Redis can serve a single request in every given moment, so all the requests are served sequentially. This is very similar to how Node.js works as well. However, both products are not often perceived as being slow. This is caused in part by the small amount of time to complete a single request, but primarily because these products are designed to not block on system calls, such as reading data from or writing data to a socket.

I said that Redis is *mostly* single threaded since actually from Redis 2.4 we use threads in Redis in order to perform some slow I/O operations in the background, mainly related to disk I/O, but this does not change the fact that Redis serves all the requests using a single thread.

Latency generated by slow commands

A consequence of being single thread is that when a request is slow to serve all the other clients will wait for this request to be served. When executing normal commands, like GET or SET or LPUSH this is not a problem at all since these commands are executed in constant (and very small) time. However there are commands operating on many elements, like SORT, LREM, SUNION and others. For instance taking the intersection of two big sets can take a considerable amount of time.

The algorithmic complexity of all commands is documented. A good practice is to systematically check it when using commands you are not familiar with.

If you have latency concerns you should either not use slow commands against values composed of many elements, or you should run a replica using Redis replication where you run all your slow queries.

It is possible to monitor slow commands using the Redis Slow Log feature.

Additionally, you can use your favorite per-process monitoring program (top, htop, prstat, etc ...) to quickly check the CPU consumption of the main Redis process. If it is high while the traffic is not, it is usually a sign that slow commands are used.

IMPORTANT NOTE: a VERY common source of latency generated by the execution of slow commands is the use of the KEYS command in production environments. KEYS, as documented in the Redis documentation, should only be used for debugging purposes. Since Redis 2.8 a new commands were introduced in order to iterate the key space and other large collections incrementally, please check the SCAN, SSCAN, HSCAN and ZSCAN commands for more information.

Latency generated by fork

In order to generate the RDB file in background, or to rewrite the Append Only File if AOF persistence is enabled, Redis has to fork background processes. The fork operation (running in the main thread) can induce latency by itself.

Forking is an expensive operation on most Unix-like systems, since it involves copying a good number of objects linked to the process. This is especially true for the page table associated to the virtual memory mechanism.

For instance on a Linux/AMD64 system, the memory is divided in 4 kB pages. To convert virtual addresses to physical addresses, each process stores a page table (actually represented as a tree) containing at least a pointer per page of the address space of the process. So a large 24 GB Redis instance requires a page table of 24 GB / 4 kB * 8 = 48 MB.

When a background save is performed, this instance will have to be forked, which will involve allocating and copying 48 MB of memory. It takes time and CPU, especially on virtual machines where allocation and initialization of a large memory chunk can be expensive.

Fork time in different systems

Modern hardware is pretty fast at copying the page table, but Xen is not. The problem with Xen is not virtualization-specific, but Xen-specific. For instance using VMware or Virtual Box does not result into slow fork time. The following is a table that compares fork time for different Redis instance size. Data is obtained performing a BGSAVE and looking at the `latest_fork_usec` filed in the INFO command output.

However the good news is that **new types of EC2 HVM based instances are much better with fork times**, almost on par with physical servers, so for example using m3.medium (or better) instances will provide good results.

- **Linux beefy VM on VMware** 6.0GB RSS forked in 77 milliseconds (12.8 milliseconds per GB).
- **Linux running on physical machine (Unknown HW)** 6.1GB RSS forked in 80 milliseconds (13.1 milliseconds per GB)
- **Linux running on physical machine (Xeon @ 2.27Ghz)** 6.9GB RSS forked into 62 milliseconds (9 milliseconds per GB).
- **Linux VM on 6sync (KVM)** 360 MB RSS forked in 8.2 milliseconds (23.3 milliseconds per GB).
- **Linux VM on EC2, old instance types (Xen)** 6.1GB RSS forked in 1460 milliseconds (239.3 milliseconds per GB).
- **Linux VM on EC2, new instance types (Xen)** 1GB RSS forked in 10 milliseconds (10 milliseconds per GB).
- **Linux VM on Linode (Xen)** 0.9GBRSS forked into 382 milliseconds (424 milliseconds per GB).

As you can see certain VMs running on Xen have a performance hit that is between one order to two orders of magnitude. For EC2 users the suggestion is simple: use modern HVM based instances.

Latency induced by transparent huge pages

Unfortunately when a Linux kernel has transparent huge pages enabled, Redis incurs to a big latency penalty after the `fork` call is used in order to persist on disk. Huge pages are the cause of the following issue:

1. Fork is called, two processes with shared huge pages are created.
2. In a busy instance, a few event loops runs will cause commands to target a few thousand of pages, causing the copy on write of almost the whole process memory.
3. This will result in big latency and big memory usage.

Make sure to **disable transparent huge pages** using the following command:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

Latency induced by swapping (operating system paging)

Linux (and many other modern operating systems) is able to relocate memory pages from the memory to the disk, and vice versa, in order to use the system memory efficiently.

If a Redis page is moved by the kernel from the memory to the swap file, when the data stored in this memory page is used by Redis (for example accessing a key stored into this memory page) the kernel will stop the Redis process in order to move the page back into the main memory. This is a slow operation involving random I/Os (compared to accessing a page that is already in memory) and will result into anomalous latency experienced by Redis clients.

The kernel relocates Redis memory pages on disk mainly because of three reasons:

- The system is under memory pressure since the running processes are demanding more physical memory than the amount that is available. The simplest instance of this problem is simply Redis using more memory than is available.
- The Redis instance data set, or part of the data set, is mostly completely idle (never accessed by clients), so the kernel could swap idle memory pages on disk. This problem is very rare since even a moderately slow instance will touch all the memory pages often, forcing the kernel to retain all the pages in memory.
- Some processes are generating massive read or write I/Os on the system. Because files are generally cached, it tends to put pressure on the kernel to increase the filesystem cache, and therefore generate swapping activity. Please note it includes Redis RDB and/or AOF background threads which can produce large files.

Fortunately Linux offers good tools to investigate the problem, so the simplest thing to do is when latency due to swapping is suspected is just to check if this is the case.

The first thing to do is to checking the amount of Redis memory that is swapped on disk. In order to do so you need to obtain the Redis instance pid:

```
$ redis-cli info | grep process_id
process_id:5454
```

Now enter the /proc file system directory for this process:

```
$ cd /proc/5454
```

Here you'll find a file called **smaps** that describes the memory layout of the Redis process (assuming you are using Linux 2.6.16 or newer). This file contains very detailed information about our process memory maps, and one field called **Swap** is exactly what we are looking for. However there is not just a single swap field since the smaps file contains the different memory maps of our Redis process (The memory layout of a process is more complex than a simple linear array of pages).

Since we are interested in all the memory swapped by our process the first thing to do is to grep for the Swap field across all the file:

```
$ cat smaps | grep 'Swap:'
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                 12 kB
Swap:                156 kB
Swap:                  8 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  4 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  4 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  4 kB
Swap:                  4 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
Swap:                  0 kB
```

If everything is 0 kB, or if there are sporadic 4k entries, everything is perfectly normal. Actually in our example instance (the one of a real web site running Redis and serving hundreds of users every second) there are a few entries that show more swapped pages. To investigate if this is a serious problem or not we change our command in order to also print the size of the memory map:

```
$ cat smaps | egrep '^(Swap|Size)'
Size:                316 kB
```

```
Size:                316 kB
Swap:                  0 kB
Size:                  4 kB
Swap:                  0 kB
Size:                  8 kB
Swap:                  0 kB
Size:                 40 kB
Swap:                  0 kB
Size:                132 kB
Swap:                  0 kB
Size:             720896 kB
Swap:                 12 kB
Size:               4096 kB
Swap:                156 kB
Size:               4096 kB
Swap:                  8 kB
Size:               4096 kB
Swap:                  0 kB
Size:                  4 kB
Swap:                  0 kB
Size:               1272 kB
Swap:                  0 kB
Size:                  8 kB
Swap:                  0 kB
Size:                  4 kB
Swap:                  0 kB
Size:                 16 kB
Swap:                  0 kB
Size:                 84 kB
Swap:                  0 kB
Size:                  4 kB
Swap:                  0 kB
Size:                  4 kB
Swap:                  0 kB
Size:                  8 kB
Swap:                  4 kB
Size:                  8 kB
Swap:                  0 kB
Size:                  4 kB
Swap:                  0 kB
Size:                  4 kB
Swap:                  4 kB
Size:                144 kB
```

```
Size:                 144 kB
Swap:                   0 kB
Size:                   4 kB
Swap:                   0 kB
Size:                   4 kB
Swap:                   4 kB
Size:                  12 kB
Swap:                   4 kB
Size:                 108 kB
Swap:                   0 kB
Size:                   4 kB
Swap:                   0 kB
Size:                   4 kB
Swap:                   0 kB
Size:                 272 kB
Swap:                   0 kB
Size:                   4 kB
Swap:                   0 kB
```

As you can see from the output, there is a map of 720896 kB (with just 12 kB swapped) and 156 kB more swapped in another map: basically a very small amount of our memory is swapped so this is not going to create any problem at all.

If instead a non trivial amount of the process memory is swapped on disk your latency problems are likely related to swapping. If this is the case with your Redis instance you can further verify it using the **vmstat** command:

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ---
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us
 0  0   3980 697932 147180 1406456    0    0     2     2    2    0  4
 0  0   3980 697428 147180 1406580    0    0     0     0 19088 16104
 0  0   3980 697296 147180 1406616    0    0     0    28 18936 16193
 0  0   3980 697048 147180 1406640    0    0     0     0 18613 15987
 2  0   3980 696924 147180 1406656    0    0     0     0 18744 16299
 0  0   3980 697048 147180 1406688    0    0     0     4 18520 15974
^C
```

The interesting part of the output for our needs are the two columns **si** and **so**, that counts the amount of memory swapped from/to the swap file. If you see non zero counts in those two columns then there is swapping activity in your system.

Finally, the **iostat** command can be used to check the global I/O activity of the system.

```
$ iostat -xk 1
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          13.55    0.04    2.92    0.53    0.00   82.95

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avg
sda               0.77     0.00    0.01    0.00     0.40     0.00
sdb               1.27     4.75    0.82    3.54    38.00    32.32
```

If your latency problem is due to Redis memory being swapped on disk you need to lower the memory pressure in your system, either adding more RAM if Redis is using more memory than the available, or avoiding running other memory hungry processes in the same system.

Latency due to AOF and disk I/O

Another source of latency is due to the Append Only File support on Redis. The AOF basically uses two system calls to accomplish its work. One is write(2) that is used in order to write data to the append only file, and the other one is fdatasync(2) that is used in order to flush the kernel file buffer on disk in order to ensure the durability level specified by the user.

Both the write(2) and fdatasync(2) calls can be source of latency. For instance write(2) can block both when there is a system wide sync in progress, or when the output buffers are full and the kernel requires to flush on disk in order to accept new writes.

The fdatasync(2) call is a worse source of latency as with many combinations of kernels and file systems used it can take from a few milliseconds to a few seconds to complete, especially in the case of some other process doing I/O. For this reason when possible Redis does the fdatasync(2) call in a different thread since Redis 2.4.

We'll see how configuration can affect the amount and source of latency when using the AOF file.

The AOF can be configured to perform an fsync on disk in three different ways using the **appendfsync** configuration option (this setting can be modified at runtime using the **CONFIG SET** command).

- When appendfsync is set to the value of **no** Redis performs no fsync. In this configuration the only source of latency can be write(2). When this happens usually there is no solution since simply the disk can't cope with the speed at which Redis is receiving data, however this is uncommon if the disk is not seriously slowed down by other processes doing I/O.

- When appendfsync is set to the value of **everysec** Redis performs an fsync every second. It uses a different thread, and if the fsync is still in progress Redis uses a buffer to delay the write(2) call up to two seconds (since write would block on Linux if an fsync is in progress against the same file). However if the fsync is taking too long Redis will eventually perform the write(2) call even if the fsync is still in progress, and this can be a source of latency.
- When appendfsync is set to the value of **always** an fsync is performed at every write operation before replying back to the client with an OK code (actually Redis will try to cluster many commands executed at the same time into a single fsync). In this mode performances are very low in general and it is strongly recommended to use a fast disk and a file system implementation that can perform the fsync in short time.

Most Redis users will use either the **no** or **everysec** setting for the appendfsync configuration directive. The suggestion for minimum latency is to avoid other processes doing I/O in the same system. Using an SSD disk can help as well, but usually even non SSD disks perform well with the append only file if the disk is spare as Redis writes to the append only file without performing any seek.

If you want to investigate your latency issues related to the append only file you can use the strace command under Linux:

```
sudo strace -p $(pidof redis-server) -T -e trace=fdatasync
```

The above command will show all the fdatasync(2) system calls performed by Redis in the main thread. With the above command you'll not see the fdatasync system calls performed by the background thread when the appendfsync config option is set to **everysec**. In order to do so just add the -f switch to strace.

If you wish you can also see both fdatasync and write system calls with the following command:

```
sudo strace -p $(pidof redis-server) -T -e trace=fdatasync,write
```

However since write(2) is also used in order to write data to the client sockets this will likely show too many things unrelated to disk I/O. Apparently there is no way to tell strace to just show slow system calls so I use the following command:

```
sudo strace -f -p $(pidof redis-server) -T -e trace=fdatasync,write 2
```

Latency generated by expires

Redis evict expired keys in two ways:

- One *lazy* way expires a key when it is requested by a command, but it is found to be already expired.
- One *active* way expires a few keys every 100 milliseconds.

The active expiring is designed to be adaptive. An expire cycle is started every 100 milliseconds (10 times per second), and will do the following:

- Sample `ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP` keys, evicting all the keys already expired.
- If the more than 25% of the keys were found expired, repeat.

Given that `ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP` is set to 20 by default, and the process is performed ten times per second, usually just 200 keys per second are actively expired. This is enough to clean the DB fast enough even when already expired keys are not accessed for a long time, so that the *lazy* algorithm does not help. At the same time expiring just 200 keys per second has no effects in the latency a Redis instance.

However the algorithm is adaptive and will loop if it finds more than 25% of keys already expired in the set of sampled keys. But given that we run the algorithm ten times per second, this means that the unlucky event of more than 25% of the keys in our random sample are expiring at least *in the same second*.

Basically this means that **if the database has many many keys expiring in the same second, and these make up at least 25% of the current population of keys with an expire set**, Redis can block in order to get the percentage of keys already expired below 25%.

This approach is needed in order to avoid using too much memory for keys that are already expired, and usually is absolutely harmless since it's strange that a big number of keys are going to expire in the same exact second, but it is not impossible that the user used EXPIREAT extensively with the same Unix time.

In short: be aware that many keys expiring at the same moment can be a source of latency.

Redis software watchdog

Redis 2.6 introduces the *Redis Software Watchdog* that is a debugging tool designed to track those latency problems that for one reason or the other escaped an analysis using normal tools.

The software watchdog is an experimental feature. While it is designed to be used in production environments care should be taken to backup the database before proceeding as it could possibly have unexpected interactions with the normal execution of the Redis server.

It is important to use it only as *last resort* when there is no way to track the issue by other means.

This is how this feature works:

- The user enables the software watchdog using the CONFIG SET command.
- Redis starts monitoring itself constantly.
- If Redis detects that the server is blocked into some operation that is not returning fast enough, and that may be the source of the latency issue, a low level report about where the server is blocked is dumped on the log file.
- The user contacts the developers writing a message in the Redis Google Group, including the watchdog report in the message.

Note that this feature cannot be enabled using the redis.conf file, because it is designed to be enabled only in already running instances and only for debugging purposes.

To enable the feature just use the following:

```
CONFIG SET watchdog-period 500
```

The period is specified in milliseconds. In the above example I specified to log latency issues only if the server detects a delay of 500 milliseconds or greater. The minimum configurable period is 200 milliseconds.

When you are done with the software watchdog you can turn it off setting the `watchdog-period` parameter to 0. **Important:** remember to do this because keeping the instance with the watchdog turned on for a longer time than needed is generally not a good idea.

The following is an example of what you'll see printed in the log file once the software watchdog detects a delay longer than the configured one:

```
[8547 | signal handler] (1333114359)
--- WATCHDOG TIMER EXPIRED ---
/lib/libc.so.6(nanosleep+0x2d) [0x7f16b5c2d39d]
/lib/libpthread.so.0(+0xf8f0) [0x7f16b5f158f0]
/lib/libc.so.6(nanosleep+0x2d) [0x7f16b5c2d39d]
/lib/libc.so.6(usleep+0x34) [0x7f16b5c62844]
./redis-server(debugCommand+0x3e1) [0x43ab41]
./redis-server(call+0x5d) [0x415a9d]
./redis-server(processCommand+0x375) [0x415fc5]
./redis-server(processInputBuffer+0x4f) [0x4203cf]
./redis-server(readQueryFromClient+0xa0) [0x4204e0]
./redis-server(aeProcessEvents+0x128) [0x411b48]
./redis-server(aeMain+0x2b) [0x411dbb]
./redis-server(main+0x2b6) [0x418556]
/lib/libc.so.6(__libc_start_main+0xfd) [0x7f16b5ba1c4d]
./redis-server() [0x411099]
------
```

Note: in the example the **DEBUG SLEEP** command was used in order to block the server. The stack trace is different if the server blocks in a different context.

If you happen to collect multiple watchdog stack traces you are encouraged to send everything to the Redis Google Group: the more traces we obtain, the simpler it will be to understand what the problem with your instance is.