Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

HAPROXY COMMUNITY EDITION (http://www.haprox

Management Guide

version 2.0.35

```
Note to documentation contributors :
    This document is formatted with 80 columns per line, with even number of
    spaces for indentation and without tabs. Please follow these rules strictly
    so that it remains easily printable everywhere. If you add sections, please
    update the summary below for easier searching.
```

Summary

1. Prerequisites

```
In this document it is assumed that the reader has sufficient administration
skills on a UNIX-like operating system, uses the shell on a daily basis and is
familiar with troubleshooting utilities such as strace and tcpdump.
```

Summary Keywords

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

2. Quick reminder about HAProxy's architecture

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

HAProxy is a multi-threaded, event-driven, non-blocking daemon. This means is uses event multiplexing to schedule all of its activities instead of relying on the system to schedule between multiple activities. Most of the time it runs as a single process, so the output of "ps aux" on a system will report only one "haproxy" process, unless a soft reload is in progress and an older process is finishing its job in parallel to the new one. It is thus always easy to trace its activity using the strace utility. In order to scale with the number of available processors, by default haproxy will start one worker thread per processor it is allowed to run on. Unless explicitly configured differently, the incoming traffic is spread over all these threads, all running the same event loop. A great care is taken to limit inter-thread dependencies to the strict minimum, so as to try to achieve near-linear scalability. This has some impacts such as the fact that a given connection is served by a single thread. Thus in order to use all available processing capacity, it is needed to have at least as many connections as there are threads, which is almost always granted.

HAProxy is designed to isolate itself into a chroot jail during startup, where it cannot perform any file-system access at all. This is also true for the libraries it depends on (eg: libc, libssl, etc). The immediate effect is that a running process will not be able to reload a configuration file to apply changes, instead a new process will be started using the updated configuration file. Some other less obvious effects are that some timezone files or resolver files the libc might attempt to access at run time will not be found, though this should generally not happen as they're not needed after startup. A nice consequence of this principle is that the HAProxy process is totally stateless, and no cleanup is needed after it's killed, so any killing method that works will do the right thing.

HAProxy doesn't write log files, but it relies on the standard syslog protocol to send logs to a remote server (which is often located on the same system).

HAProxy uses its internal clock to enforce timeouts, that is derived from the system's time but where unexpected drift is corrected. This is done by limiting the time spent waiting in poll() for an event, and measuring the time it really took. In practice it never waits more than one second. This explains why, when running strace over a completely idle process, periodic calls to poll() (or any of its variants) surrounded by two gettimeofday() calls are noticed. They are normal, completely harmless and so cheap that the load they imply is totally undetectable at the system scale, so there's nothing abnormal there. Example :

```
16:35:40.002320 gettimeofday({1442759740, 2605}, NULL) = 0
16:35:40.002942 epoll_wait(0, {}, 200, 1000) = 0
16:35:41.007542 gettimeofday({1442759741, 7641}, NULL) = 0
16:35:41.007998 gettimeofday({1442759741, 8114}, NULL) = 0
16:35:41.008391 epoll_wait(0, {}, 200, 1000) = 0
16:35:42.011313 gettimeofday({1442759742, 11411}, NULL) = 0
```

HAProxy is a TCP proxy, not a router. It deals with established connections that have been validated by the kernel, and not with packets of any form nor with sockets in other states (eg: no SYN_RECV nor TIME_WAIT), though their existence may prevent it from binding a port. It relies on the system to accept incoming connections and to initiate outgoing connections. An immediate effect of this is that there is no relation between packets observed on the two sides of a forwarded connection, which can be of different size, numbers and even family. Since a connection may only be accepted from a socket in LISTEN state, all the sockets it is listening to are necessarily visible using the "netstat" utility to show listening sockets. Example :

```
# netstat -ltnp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address   Foreign Address   State    PID/Program nam
tcp        0      0 0.0.0.0:22      0.0.0.0:*         LISTEN   1629/sshd
tcp        0      0 0.0.0.0:80      0.0.0.0:*         LISTEN   2847/haproxy
tcp        0      0 0.0.0.0:443     0.0.0.0:*         LISTEN   2847/haproxy
```

3. Starting HAProxy

Summary Keywords

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

HAProxy is started by invoking the "haproxy" program with a number of arguments passed on the command line. The actual syntax is :

```
  $ haproxy [<options>]*
```

where [<options>]* is any number of options. An option always starts

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
followed by one of more letters, and possibly followed by one or multiple extra
arguments. Without any option, HAProxy displays the help page with a reminder
about supported options. Available options may vary slightly based on the
operating system. A fair number of these options overlap with an equivalent one
if the "global" section. In this case, the command line always has precedence
over the configuration file, so that the command line can be used to quickly
enforce some settings without touching the configuration files. The current
list of options is :

  -- <cfgfile>* : all the arguments following "--" are paths to configuration
    file/directory to be loaded and processed in the declaration order. It is
    mostly useful when relying on the shell to load many files that are
    numerically ordered. See also "-f". The difference between "--" and "-f" is
    that one "-f" must be placed before each file name, while a single "--" is
    needed before all file names. Both options can be used together, the
    command line ordering still applies. When more than one file is specified,
    each file must start on a section boundary, so the first keyword of each
    file must be one of "global", "defaults", "peers", "listen", "frontend",
    "backend", and so on. A file cannot contain just a server list for example.

  -f <cfgfile|cfgdir> : adds <cfgfile> to the list of configuration files to be
    loaded. If <cfgdir> is a directory, all the files (and only files) it
    contains are added in lexical order (using LC_COLLATE=C) to the list of
    configuration files to be loaded ; only files with ".cfg" extension are
    added, only non hidden files (not prefixed with ".") are added.
    Configuration files are loaded and processed in their declaration order.
    This option may be specified multiple times to load multiple files. See
    also "--". The difference between "--" and "-f" is that one "-f" must be
    placed before each file name, while a single "--" is needed before all file
    names. Both options can be used together, the command line ordering still
    applies. When more than one file is specified, each file must start on a
    section boundary, so the first keyword of each file must be one of
    "global", "defaults", "peers", "listen", "frontend", "backend", and so on.
    A file cannot contain just a server list for example.

  -C <dir> : changes to directory <dir> before loading configuration
    files. This is useful when using relative paths. Warning when using
    wildcards after "--" which are in fact replaced by the shell before
    starting haproxy.

  -D : start as a daemon. The process detaches from the current terminal after
    forking, and errors are not reported anymore in the terminal. It is
    equivalent to the "daemon" keyword in the "global" section of the
    configuration. It is recommended to always force it in any init script so
    that a faulty configuration doesn't prevent the system from booting.

  -L <name> : change the local peer name to <name>, which defaults to the local
    hostname. This is used only with peers replication. You can use the
    variable $HAPROXY_LOCALPEER in the configuration file to reference the
    peer name.

  -N <limit> : sets the default per-proxy maxconn to <limit> instead of the
    builtin default value (usually 2000). Only useful for debugging.

  -V : enable verbose mode (disables quiet mode). Reverts the effect of "-q" or
    "quiet".

  -W : master-worker mode. It is equivalent to the "master-worker" keyword in
    the "global" section of the configuration. This mode will launch a "master"
    which will monitor the "workers". Using this mode, you can reload HAProxy
    directly by sending a SIGUSR2 signal to the master.  The master-worker mode
    is compatible either with the foreground or daemon mode.  It is
    recommended to use this mode with multiprocess and systemd.

  -Ws : master-worker mode with support of `notify` type of systemd service.
    This option is only available when HAProxy was built with `USE_SYSTEMD`
    build option enabled.

  -c : only performs a check of the configuration files and exits before trying
```

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
  to bind. The exit status is zero if everything is OK, or non-zero if an
  error is encountered.

-d : enable debug mode. This disables daemon mode, forces the process to stay
  in foreground and to show incoming and outgoing events. It is equivalent to
  the "global" section's "debug" keyword. It must never be used in an init
  script.

-dG : disable use of getaddrinfo() to resolve host names into addresses. It
  can be used when suspecting that getaddrinfo() doesn't work as expected.
  This option was made available because many bogus implementations of
  getaddrinfo() exist on various systems and cause anomalies that are
  difficult to troubleshoot.

-dM[<byte>] : forces memory poisoning, which means that each and every
  memory region allocated with malloc() or pool_alloc() will be filled with
  <byte> before being passed to the caller. When <byte> is not specified, it
  defaults to 0x50 ('P'). While this slightly slows down operations, it is
  useful to reliably trigger issues resulting from missing initializations in
  the code that cause random crashes. Note that -dM0 has the effect of
  turning any malloc() into a calloc(). In any case if a bug appears or
  disappears when using this option it means there is a bug in haproxy, so
  please report it.

-dS : disable use of the splice() system call. It is equivalent to the
  "global" section's "nosplice" keyword. This may be used when splice() is
  suspected to behave improperly or to cause performance issues, or when
  using strace to see the forwarded data (which do not appear when using
  splice()).

-dV : disable SSL verify on the server side. It is equivalent to having
  "ssl-server-verify none" in the "global" section. This is useful when
  trying to reproduce production issues out of the production
  environment. Never use this in an init script as it degrades SSL security
  to the servers.

-db : disable background mode and multi-process mode. The process remains in
  foreground. It is mainly used during development or during small tests, as
  Ctrl-C is enough to stop the process. Never use it in an init script.

-de : disable the use of the "epoll" poller. It is equivalent to the "global"
  section's keyword "noepoll". It is mostly useful when suspecting a bug
  related to this poller. On systems supporting epoll, the fallback will
  generally be the "poll" poller.

-dk : disable the use of the "kqueue" poller. It is equivalent to the
  "global" section's keyword "nokqueue". It is mostly useful when suspecting
  a bug related to this poller. On systems supporting kqueue, the fallback
  will generally be the "poll" poller.

-dp : disable the use of the "poll" poller. It is equivalent to the "global"
  section's keyword "nopoll". It is mostly useful when suspecting a bug
  related to this poller. On systems supporting poll, the fallback will
  generally be the "select" poller, which cannot be disabled and is limited
  to 1024 file descriptors.

-dr : ignore server address resolution failures. It is very common when
  validating a configuration out of production not to have access to the same
  resolvers and to fail on server address resolution, making it difficult to
  test a configuration. This option simply appends the "none" method to the
  list of address resolution methods for all servers, ensuring that even if
  the libc fails to resolve an address, the startup sequence is not
  interrupted.

-m <limit> : limit the total allocatable memory to <limit> megabytes across
  all processes. This may cause some connection refusals or some slowdowns
  depending on the amount of memory needed for normal operations. This is
  mostly used to force the processes to work in a constrained resource usage
  scenario. It is important to note that the memory is not shared between
```

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
  processes, so in a multi-process scenario, this value is first divided by
  global.nbproc before forking.

-n <limit> : limits the per-process connection limit to <limit>. This is
  equivalent to the global section's keyword "maxconn". It has precedence
  over this keyword. This may be used to quickly force lower limits to avoid
  a service outage on systems where resource limits are too low.

-p <file> : write all processes' pids into <file> during startup. This is
  equivalent to the "global" section's keyword "pidfile". The file is opened
  before entering the chroot jail, and after doing the chdir() implied by
  "-C". Each pid appears on its own line.

-q : set "quiet" mode. This disables the output messages. It can be used in
  combination with "-c" to just check if a configuration file is valid or not.

-S <bind>[,bind_options...]: in master-worker mode, bind a master CLI, which
  allows the access to every processes, running or leaving ones.
  For security reasons, it is recommended to bind the master CLI to a local
  UNIX socket. The bind options are the same as the keyword "bind" in
  the configuration file with words separated by commas instead of spaces.

  Note that this socket can't be used to retrieve the listening sockets from
  an old process during a seamless reload.

-sf <pid>* : send the "finish" signal (SIGUSR1) to older processes after boot
  completion to ask them to finish what they are doing and to leave. <pid>
  is a list of pids to signal (one per argument). The list ends on any
  option starting with a "-". It is not a problem if the list of pids is
  empty, so that it can be built on the fly based on the result of a command
  like "pidof" or "pgrep".

-st <pid>* : send the "terminate" signal (SIGTERM) to older processes after
  boot completion to terminate them immediately without finishing what they
  were doing. <pid> is a list of pids to signal (one per argument). The list
  is ends on any option starting with a "-". It is not a problem if the list
  of pids is empty, so that it can be built on the fly based on the result of
  a command like "pidof" or "pgrep".

-v : report the version and build date.

-vv : display the version, build options, libraries versions and usable
  pollers. This output is systematically requested when filing a bug report.

-x <unix_socket> : connect to the specified socket and try to retrieve any
  listening sockets from the old process, and use them instead of trying to
  bind new ones. This is useful to avoid missing any new connection when
  reloading the configuration on Linux. The capability must be enable on the
  stats socket using "expose-fd listeners" in your configuration.
```

A safe way to start HAProxy from an init file consists in forcing the daemon mode, storing existing pids to a pid file and using this pid file to notify older processes to finish before leaving :

```
   haproxy -f /etc/haproxy.cfg \
           -D -p /var/run/haproxy.pid -sf $(cat /var/run/haproxy.pid)
```

When the configuration is split into a few specific files (eg: tcp vs http), it is recommended to use the "-f" option :

```
   haproxy -f /etc/haproxy/global.cfg -f /etc/haproxy/stats.cfg \
           -f /etc/haproxy/default-tcp.cfg -f /etc/haproxy/tcp.cfg \
           -f /etc/haproxy/default-http.cfg -f /etc/haproxy/http.cfg \
           -D -p /var/run/haproxy.pid -sf $(cat /var/run/haproxy.pid)
```

When an unknown number of files is expected, such as customer-specific files, it is recommended to assign them a name starting with a fixed-size sequence number and to use "--" to load them, possibly after loading some defaults :

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
 haproxy -f /etc/haproxy/global.cfg -f /etc/haproxy/stats.cfg \
         -f /etc/haproxy/default-tcp.cfg -f /etc/haproxy/tcp.cfg \
         -f /etc/haproxy/default-http.cfg -f /etc/haproxy/http.cfg \
         -D -p /var/run/haproxy.pid -sf $(cat /var/run/haproxy.pid) \
         -f /etc/haproxy/default-customers.cfg -- /etc/haproxy/customers/*
```

Sometimes a failure to start may happen for whatever reason. Then it is important to verify if the version of HAProxy you are invoking is the expected version and if it supports the features you are expecting (eg: SSL, PCRE, compression, Lua, etc). This can be verified using "haproxy -vv". Some important information such as certain build options, the target system and the versions of the libraries being used are reported there. It is also what you will systematically be asked for when posting a bug report :

```
  $ haproxy -vv
  HA-Proxy version 1.6-dev7-a088d3-4 2015/10/08
  Copyright 2000-2015 Willy Tarreau <willy@haproxy.org>

  Build options :
    TARGET  = linux2628
    CPU     = generic
    CC      = gcc
    CFLAGS  = -pg -O0 -g -fno-strict-aliasing -Wdeclaration-after-statement \
              -DBUFSIZE=8030 -DMAXREWRITE=1030 -DSO_MARK=36 -DTCP_REPAIR=19
    OPTIONS = USE_ZLIB=1 USE_DLMALLOC=1 USE_OPENSSL=1 USE_LUA=1 USE_PCRE=1

  Default settings :
    maxconn = 2000, bufsize = 8030, maxrewrite = 1030, maxpollevents = 200

  Encrypted password support via crypt(3): yes
  Built with zlib version : 1.2.6
  Compression algorithms supported : identity("identity"), deflate("deflate"), \
                                     raw-deflate("deflate"), gzip("gzip")
  Built with OpenSSL version : OpenSSL 1.0.1o 12 Jun 2015
  Running on OpenSSL version : OpenSSL 1.0.1o 12 Jun 2015
  OpenSSL library supports TLS extensions : yes
  OpenSSL library supports SNI : yes
  OpenSSL library supports prefer-server-ciphers : yes
  Built with PCRE version : 8.12 2011-01-15
  PCRE library supports JIT : no (USE_PCRE_JIT not set)
  Built with Lua version : Lua 5.3.1
  Built with transparent proxy support using: IP_TRANSPARENT IP_FREEBIND

  Available polling systems :
        epoll : pref=300,  test result OK
         poll : pref=200,  test result OK
       select : pref=150,  test result OK
  Total: 3 (3 usable), will use epoll.
```

The relevant information that many non-developer users can verify here are :

- the version : 1.6-dev7-a088d3-4 above means the code is currently at commit ID "a088d3" which is the 4th one after after official version "1.6-dev7". Version 1.6-dev7 would show as "1.6-dev7-8c1ad7". What matters here is in fact "1.6-dev7". This is the 7th development version of what will become version 1.6 in the future. A development version not suitable for use in production (unless you know exactly what you are doing). A stable version will show as a 3-numbers version, such as "1.5.14-16f863", indicating the 14th level of fix on top of version 1.5. This is a production-ready version.

- the release date : 2015/10/08. It is represented in the universal year/month/day format. Here this means August 8th, 2015. Given that stable releases are issued every few months (1-2 months at the beginning, sometimes 6 months once the product becomes very stable), if you're seeing an old date here, it means you're probably affected by a number of bugs or security issues that have since been fixed and that it might be worth checking on the official site.

- build options : they are relevant to people who build their packages themselves, they can explain why things are not behaving as expected. For

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
example the development version above was built for Linux 2.6.28 or later,
targeting a generic CPU (no CPU-specific optimizations), and lacks any
code optimization (-O0) so it will perform poorly in terms of performance.

- libraries versions : zlib version is reported as found in the library
itself. In general zlib is considered a very stable product and upgrades
are almost never needed. OpenSSL reports two versions, the version used at
build time and the one being used, as found on the system. These ones may
differ by the last letter but never by the numbers. The build date is also
reported because most OpenSSL bugs are security issues and need to be taken
seriously, so this library absolutely needs to be kept up to date. Seeing a
4-months old version here is highly suspicious and indeed an update was
missed. PCRE provides very fast regular expressions and is highly
recommended. Certain of its extensions such as JIT are not present in all
versions and still young so some people prefer not to build with them,
which is why the build status is reported as well. Regarding the Lua
scripting language, HAProxy expects version 5.3 which is very young since
it was released a little time before HAProxy 1.6. It is important to check
on the Lua web site if some fixes are proposed for this branch.

- Available polling systems will affect the process's scalability when
dealing with more than about one thousand of concurrent connections. These
ones are only available when the correct system was indicated in the TARGET
variable during the build. The "epoll" mechanism is highly recommended on
Linux, and the kqueue mechanism is highly recommended on BSD. Lacking them
will result in poll() or even select() being used, causing a high CPU usage
when dealing with a lot of connections.
```

4. Stopping and restarting HAProxy

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

HAProxy supports a graceful and a hard stop. The hard stop is simple, when the SIGTERM signal is sent to the haproxy process, it immediately quits and all established connections are closed. The graceful stop is triggered when the SIGUSR1 signal is sent to the haproxy process. It consists in only unbinding from listening ports, but continue to process existing connections until they close. Once the last connection is closed, the process leaves.

The hard stop method is used for the "stop" or "restart" actions of the service management script. The graceful stop is used for the "reload" action which tries to seamlessly reload a new configuration in a new process.

Both of these signals may be sent by the new haproxy process itself during a reload or restart, so that they are sent at the latest possible moment and only if absolutely required. This is what is performed by the "-st" (hard) and "-sf" (graceful) options respectively.

In master-worker mode, it is not needed to start a new haproxy process in order to reload the configuration. The master process reacts to the SIGUSR2 signal by reexecuting itself with the -sf parameter followed by the PIDs of the workers. The master will then parse the configuration file and fork new workers.

To understand better how these signals are used, it is important to understand the whole restart mechanism.

First, an existing haproxy process is running. The administrator uses a system specific command such as "/etc/init.d/haproxy reload" to indicate he wants to take the new configuration file into effect. What happens then is the following. First, the service script (/etc/init.d/haproxy or equivalent) will verify that the configuration file parses correctly using "haproxy -c". After that it will try to start haproxy with this configuration file, using "-st" or "-sf".

Then HAProxy tries to bind to all listening ports. If some fatal errors happen (eg: address not present on the system, permission denied), the process quits with an error. If a socket binding fails because a port is already in use, then the process will first send a SIGTTOU signal to all the pids specified in the "-st" or "-sf" pid list. This is what is called the "pause" signal. It instructs all existing haproxy processes to temporarily stop listening to their ports so that the new process can try to bind again. During this time, the old process continues to process existing connections. If the binding still fails (because for example a port is shared with another daemon), then the new process sends a SIGTTIN signal to the old processes to instruct them to resume operations just as if nothing happened. The old processes will then restart listening to the ports and continue to accept connections. Note that this mechanism is system dependent and some operating systems may not support it in multi-process mode.

If the new process manages to bind correctly to all ports, then it sends either the SIGTERM (hard stop in case of "-st") or the SIGUSR1 (graceful stop in case of "-sf") to all processes to notify them that it is now in charge of operations and that the old processes will have to leave, either immediately or once they have finished their job.

It is important to note that during this timeframe, there are two small windows of a few milliseconds each where it is possible that a few connection failures will be noticed during high loads. Typically observed failure rates are around 1 failure during a reload operation every 10000 new connections per second, which means that a heavily loaded site running at 30000 new connections per second may see about 3 failed connection upon every reload. The two situations where this happens are :

  - if the new process fails to bind due to the presence of the old process, it will first have to go through the SIGTTOU+SIGTTIN sequence, which typically lasts about one millisecond for a few tens of frontends, and during which some ports will not be bound to the old process and not yet bound to the new one. HAProxy works around this on systems that support the SO_REUSEPORT socket options, as it allows the new process to bind without first asking the old one to unbind. Most BSD systems have been supporting this almost forever. Linux has been supporting this in version 2.0 and dropped it around 2.2, but some patches were floating around by then. It

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
   was reintroduced in kernel 3.9, so if you are observing a connection
   failure rate above the one mentioned above, please ensure that your kernel
   is 3.9 or newer, or that relevant patches were backported to your kernel
   (less likely).

 - when the old processes close the listening ports, the kernel may not always
   redistribute any pending connection that was remaining in the socket's
   backlog. Under high loads, a SYN packet may happen just before the socket
   is closed, and will lead to an RST packet being sent to the client. In some
   critical environments where even one drop is not acceptable, these ones are
   sometimes dealt with using firewall rules to block SYN packets during the
   reload, forcing the client to retransmit. This is totally system-dependent,
   as some systems might be able to visit other listening queues and avoid
   this RST. A second case concerns the ACK from the client on a local socket
   that was in SYN_RECV state just before the close. This ACK will lead to an
   RST packet while the haproxy process is still not aware of it. This one is
   harder to get rid of, though the firewall filtering rules mentioned above
   will work well if applied one second or so before restarting the process.

For the vast majority of users, such drops will never ever happen since they
don't have enough load to trigger the race conditions. And for most high traffic
users, the failure rate is still fairly within the noise margin provided that at
least SO_REUSEPORT is properly supported on their systems.
```

5. File-descriptor limitations

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

In order to ensure that all incoming connections will successfully be served, HAProxy computes at load time the total number of file descriptors that will be needed during the process's life. A regular Unix process is generally granted 1024 file descriptors by default, and a privileged process can raise this limit itself. This is one reason for starting HAProxy as root and letting it adjust the limit. The default limit of 1024 file descriptors roughly allow about 500 concurrent connections to be processed. The computation is based on the global maxconn parameter which limits the total number of connections per process, the number of listeners, the number of servers which have a health check enabled, the agent checks, the peers, the loggers and possibly a few other technical requirements. A simple rough estimate of this number consists in simply doubling the maxconn value and adding a few tens to get the approximate number of file descriptors needed.

Originally HAProxy did not know how to compute this value, and it was necessary to pass the value using the "ulimit-n" setting in the global section. This explains why even today a lot of configurations are seen with this setting present. Unfortunately it was often miscalculated resulting in connection failures when approaching maxconn instead of throttling incoming connection while waiting for the needed resources. For this reason it is important to remove any vestigial "ulimit-n" setting that can remain from very old versions.

Raising the number of file descriptors to accept even moderate loads is mandatory but comes with some OS-specific adjustments. First, the select() polling system is limited to 1024 file descriptors. In fact on Linux it used to be capable of handling more but since certain OS ship with excessively restrictive SELinux policies forbidding the use of select() with more than 1024 file descriptors, HAProxy now refuses to start in this case in order to avoid any issue at run time. On all supported operating systems, poll() is available and will not suffer from this limitation. It is automatically picked so there is nothing to do to get a working configuration. But poll's becomes very slow when the number of file descriptors increases. While HAProxy does its best to limit this performance impact (eg: via the use of the internal file descriptor cache and batched processing), a good rule of thumb is that using poll() with more than a thousand concurrent connections will use a lot of CPU.

For Linux systems base on kernels 2.6 and above, the epoll() system call will be used. It's a much more scalable mechanism relying on callbacks in the kernel that guarantee a constant wake up time regardless of the number of registered monitored file descriptors. It is automatically used where detected, provided that HAProxy had been built for one of the Linux flavors. Its presence and support can be verified using "haproxy -vv".

For BSD systems which support it, kqueue() is available as an alternative. It is much faster than poll() and even slightly faster than epoll() thanks to its batched handling of changes. At least FreeBSD and OpenBSD support it. Just like with Linux's epoll(), its support and availability are reported in the output of "haproxy -vv".

Having a good poller is one thing, but it is mandatory that the process can reach the limits. When HAProxy starts, it immediately sets the new process's file descriptor limits and verifies if it succeeds. In case of failure, it reports it before forking so that the administrator can see the problem. As long as the process is started by as root, there should be no reason for this setting to fail. However, it can fail if the process is started by an unprivileged user. If there is a compelling reason for *not* starting haproxy as root (eg: started by end users, or by a per-application account), then the file descriptor limit can be raised by the system administrator for this specific user. The effectiveness of the setting can be verified by issuing "ulimit -n" from the user's command line. It should reflect the new limit.

Warning: when an unprivileged user's limits are changed in this user's account, it is fairly common that these values are only considered when the user logs in and not at all in some scripts run at system boot time nor in crontabs. This is totally dependent on the operating system, keep in mind to check "ulimit -n" before starting haproxy when running this way. The general advice is never to start haproxy as an unprivileged user for production purposes. Another good reason is that it prevents haproxy from enabling some security protections.

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Once it is certain that the system will allow the haproxy process to use the requested number of file descriptors, two new system-specific limits may be encountered. The first one is the system-wide file descriptor limit, which is the total number of file descriptors opened on the system, covering all processes. When this limit is reached, accept() or socket() will typically return ENFILE. The second one is the per-process hard limit on the number of file descriptors, it prevents setrlimit() from being set higher. Both are very dependent on the operating system. On Linux, the system limit is set at boot based on the amount of memory. It can be changed with the "fs.file-max" sysctl. And the per-process hard limit is set to 1048576 by default, but it can be changed using the "fs.nr_open" sysctl.

File descriptor limitations may be observed on a running process when they are set too low. The strace utility will report that accept() and socket() return "-1 EMFILE" when the process's limits have been reached. In this case, simply raising the "ulimit-n" value (or removing it) will solve the problem. If these system calls return "-1 ENFILE" then it means that the kernel's limits have been reached and that something must be done on a system-wide parameter. These trouble must absolutely be addressed, as they result in high CPU usage (when accept() fails) and failed connections that are generally visible to the user. One solution also consists in lowering the global maxconn value to enforce serialization, and possibly to disable HTTP keep-alive to force connections to be released and reused faster.

6. Memory management

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

HAProxy uses a simple and fast pool-based memory management. Since it relies on a small number of different object types, it's much more efficient to pick new objects from a pool which already contains objects of the appropriate size than to call malloc() for each different size. The pools are organized as a stack or LIFO, so that newly allocated objects are taken from recently released objects still hot in the CPU caches. Pools of similar sizes are merged together, in order to limit memory fragmentation.

By default, since the focus is set on performance, each released object is put back into the pool it came from, and allocated objects are never freed since they are expected to be reused very soon.

On the CLI, it is possible to check how memory is being used in pools thanks to the "show pools" command :

```
> show pools
Dumping pools usage. Use SIGQUIT to flush them.
  - Pool cache_st (16 bytes) : 0 allocated (0 bytes), 0 used, 0 failures, 1 us
  - Pool pipe (32 bytes) : 5 allocated (160 bytes), 5 used, 0 failures, 2 user
  - Pool comp_state (48 bytes) : 3 allocated (144 bytes), 3 used, 0 failures,
  - Pool filter (64 bytes) : 0 allocated (0 bytes), 0 used, 0 failures, 3 user
  - Pool vars (80 bytes) : 0 allocated (0 bytes), 0 used, 0 failures, 2 users,
  - Pool uniqueid (128 bytes) : 0 allocated (0 bytes), 0 used, 0 failures, 2 u
  - Pool task (144 bytes) : 55 allocated (7920 bytes), 55 used, 0 failures, 1
  - Pool session (160 bytes) : 1 allocated (160 bytes), 1 used, 0 failures, 1
  - Pool h2s (208 bytes) : 0 allocated (0 bytes), 0 used, 0 failures, 2 users,
  - Pool h2c (288 bytes) : 0 allocated (0 bytes), 0 used, 0 failures, 1 users,
  - Pool spoe_ctx (304 bytes) : 0 allocated (0 bytes), 0 used, 0 failures, 2 u
  - Pool connection (400 bytes) : 2 allocated (800 bytes), 2 used, 0 failures,
  - Pool hdr_idx (416 bytes) : 0 allocated (0 bytes), 0 used, 0 failures, 1 us
  - Pool dns_resolut (480 bytes) : 0 allocated (0 bytes), 0 used, 0 failures,
  - Pool dns_answer_ (576 bytes) : 0 allocated (0 bytes), 0 used, 0 failures,
  - Pool stream (960 bytes) : 1 allocated (960 bytes), 1 used, 0 failures, 1 u
  - Pool requri (1024 bytes) : 0 allocated (0 bytes), 0 used, 0 failures, 1 us
  - Pool buffer (8030 bytes) : 3 allocated (24090 bytes), 2 used, 0 failures,
  - Pool trash (8062 bytes) : 1 allocated (8062 bytes), 1 used, 0 failures, 1
Total: 19 pools, 42296 bytes allocated, 34266 used.
```

The pool name is only indicative, it's the name of the first object type using this pool. The size in parenthesis is the object size for objects in this pool. Object sizes are always rounded up to the closest multiple of 16 bytes. The number of objects currently allocated and the equivalent number of bytes is reported so that it is easy to know which pool is responsible for the highest memory usage. The number of objects currently in use is reported as well in the "used" field. The difference between "allocated" and "used" corresponds to the objects that have been freed and are available for immediate use. The address at the end of the line is the pool's address, and the following number is the pool index when it exists, or is reported as -1 if no index was assigned.

It is possible to limit the amount of memory allocated per process using the "-m" command line option, followed by a number of megabytes. It covers all of the process's addressable space, so that includes memory used by some libraries as well as the stack, but it is a reliable limit when building a resource constrained system. It works the same way as "ulimit -v" on systems which have it, or "ulimit -d" for the other ones.

If a memory allocation fails due to the memory limit being reached or because the system doesn't have any enough memory, then haproxy will first start to free all available objects from all pools before attempting to allocate memory again. This mechanism of releasing unused memory can be triggered by sending the signal SIGQUIT to the haproxy process. When doing so, the pools state prior to the flush will also be reported to stderr when the process runs in foreground.

During a reload operation, the process switched to the graceful stop state also automatically performs some flushes after releasing any connection so that all possible memory is released to save it for the new process.

7. CPU usage

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

HAProxy normally spends most of its time in the system and a smaller part in userland. A finely tuned 3.5 GHz CPU can sustain a rate about 80000 end-to-end connection setups and closes per second at 100% CPU on a single core. When one core is saturated, typical figures are :
  - 95% system, 5% user for long TCP connections or large HTTP objects
  - 85% system and 15% user for short TCP connections or small HTTP objects in close mode
  - 70% system and 30% user for small HTTP objects in keep-alive mode

The amount of rules processing and regular expressions will increase the user land part. The presence of firewall rules, connection tracking, complex routing tables in the system will instead increase the system part.

On most systems, the CPU time observed during network transfers can be cut in 4 parts :
  - the interrupt part, which concerns all the processing performed upon I/O receipt, before the target process is even known. Typically Rx packets are accounted for in interrupt. On some systems such as Linux where interrupt processing may be deferred to a dedicated thread, it can appear as softirq, and the thread is called ksoftirqd/0 (for CPU 0). The CPU taking care of this load is generally defined by the hardware settings, though in the case of softirq it is often possible to remap the processing to another CPU. This interrupt part will often be perceived as parasitic since it's not associated with any process, but it actually is some processing being done to prepare the work for the process.

  - the system part, which concerns all the processing done using kernel code called from userland. System calls are accounted as system for example. All synchronously delivered Tx packets will be accounted for as system time. If some packets have to be deferred due to queues filling up, they may then be processed in interrupt context later (eg: upon receipt of an ACK opening a TCP window).

  - the user part, which exclusively runs application code in userland. HAProxy runs exclusively in this part, though it makes heavy use of system calls. Rules processing, regular expressions, compression, encryption all add to the user portion of CPU consumption.

  - the idle part, which is what the CPU does when there is nothing to do. For example HAProxy waits for an incoming connection, or waits for some data to leave, meaning the system is waiting for an ACK from the client to push these data.

In practice regarding HAProxy's activity, it is in general reasonably accurate (but totally inexact) to consider that interrupt/softirq are caused by Rx processing in kernel drivers, that user-land is caused by layer 7 processing in HAProxy, and that system time is caused by network processing on the Tx path.

Since HAProxy runs around an event loop, it waits for new events using poll() (or any alternative) and processes all these events as fast as possible before going back to poll() waiting for new events. It measures the time spent waiting in poll() compared to the time spent doing processing events. The ratio of polling time vs total time is called the "idle" time, it's the amount of time spent waiting for something to happen. This ratio is reported in the stats page on the "idle" line, or "Idle_pct" on the CLI. When it's close to 100%, it means the load is extremely low. When it's close to 0%, it means that there is constantly some activity. While it cannot be very accurate on an overloaded system due to other processes possibly preempting the CPU from the haproxy process, it still provides a good estimate about how HAProxy considers it is working : if the load is low and the idle ratio is low as well, it may indicate that HAProxy has a lot of work to do, possibly due to very expensive rules that have to be processed. Conversely, if HAProxy indicates the idle is close to 100% while things are slow, it means that it cannot do anything to speed things up because it is already waiting for incoming data to process. In the example below, haproxy is completely idle :

```
$ echo "show info" | socat - /var/run/haproxy.sock | grep ^Idle
Idle_pct: 100
```

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

When the idle ratio starts to become very low, it is important to tune the system and place processes and interrupts correctly to save the most possible CPU resources for all tasks. If a firewall is present, it may be worth trying to disable it or to tune it to ensure it is not responsible for a large part of the performance limitation. It's worth noting that unloading a stateful firewall generally reduces both the amount of interrupt/softirq and of system usage since such firewalls act both on the Rx and the Tx paths. On Linux, unloading the nf_conntrack and ip_conntrack modules will show whether there is anything to gain. If so, then the module runs with default settings and you'll have to figure how to tune it for better performance. In general this consists in considerably increasing the hash table size. On FreeBSD, "pfctl -d" will disable the "pf" firewall and its stateful engine at the same time.

If it is observed that a lot of time is spent in interrupt/softirq, it is important to ensure that they don't run on the same CPU. Most systems tend to pin the tasks on the CPU where they receive the network traffic because for certain workloads it improves things. But with heavily network-bound workloads it is the opposite as the haproxy process will have to fight against its kernel counterpart. Pinning haproxy to one CPU core and the interrupts to another one, all sharing the same L3 cache tends to sensibly increase network performance because in practice the amount of work for haproxy and the network stack are quite close, so they can almost fill an entire CPU each. On Linux this is done using taskset (for haproxy) or using cpu-map (from the haproxy config), and the interrupts are assigned under /proc/irq. Many network interfaces support multiple queues and multiple interrupts. In general it helps to spread them across a small number of CPU cores provided they all share the same L3 cache. Please always stop irq_balance which always does the worst possible thing on such workloads.

For CPU-bound workloads consisting in a lot of SSL traffic or a lot of compression, it may be worth using multiple processes dedicated to certain tasks, though there is no universal rule here and experimentation will have to be performed.

In order to increase the CPU capacity, it is possible to make HAProxy run as several processes, using the "nbproc" directive in the global section. There are some limitations though :
  - health checks are run per process, so the target servers will get as many checks as there are running processes ;
  - maxconn values and queues are per-process so the correct value must be set to avoid overloading the servers ;
  - outgoing connections should avoid using port ranges to avoid conflicts
  - stick-tables are per process and are not shared between processes ;
  - each peers section may only run on a single process at a time ;
  - the CLI operations will only act on a single process at a time.

With this in mind, it appears that the easiest setup often consists in having one first layer running on multiple processes and in charge for the heavy processing, passing the traffic to a second layer running in a single process. This mechanism is suited to SSL and compression which are the two CPU-heavy features. Instances can easily be chained over UNIX sockets (which are cheaper than TCP sockets and which do not waste ports), and the proxy protocol which is useful to pass client information to the next stage. When doing so, it is generally a good idea to bind all the single-process tasks to process number 1 and extra tasks to next processes, as this will make it easier to generate similar configurations for different machines.

On Linux versions 3.9 and above, running HAProxy in multi-process mode is much more efficient when each process uses a distinct listening socket on the same IP:port ; this will make the kernel evenly distribute the load across all processes instead of waking them all up. Please check the "process" option of the "bind" keyword lines in the configuration manual for more information.

8. Logging

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

For logging, HAProxy always relies on a syslog server since it does not perform any file-system access. The standard way of using it is to send logs over UDP to the log server (by default on port 514). Very commonly this is configured to 127.0.0.1 where the local syslog daemon is running, but it's also used over the network to log to a central server. The central server provides additional benefits especially in active-active scenarios where it is desirable to keep the logs merged in arrival order. HAProxy may also make use of a UNIX socket to send its logs to the local syslog daemon, but it is not recommended at all, because if the syslog server is restarted while haproxy runs, the socket will be replaced and new logs will be lost. Since HAProxy will be isolated inside a chroot jail, it will not have the ability to reconnect to the new socket. It has also been observed in field that the log buffers in use on UNIX sockets are very small and lead to lost messages even at very light loads. But this can be fine for testing however.

It is recommended to add the following directive to the "global" section to make HAProxy log to the local daemon using facility "local0" :

```
      log 127.0.0.1:514 local0
```

and then to add the following one to each "defaults" section or to each frontend and backend section :

```
      log global
```

This way, all logs will be centralized through the global definition of where the log server is.

Some syslog daemons do not listen to UDP traffic by default, so depending on the daemon being used, the syntax to enable this will vary :

  - on sysklogd, you need to pass argument "-r" on the daemon's command line
    so that it listens to a UDP socket for "remote" logs ; note that there is
    no way to limit it to address 127.0.0.1 so it will also receive logs from
    remote systems ;

  - on rsyslogd, the following lines must be added to the configuration file :

```
      $ModLoad imudp
      $UDPServerAddress *
      $UDPServerRun 514
```

  - on syslog-ng, a new source can be created the following way, it then needs
    to be added as a valid source in one of the "log" directives :

```
      source s_udp {
        udp(ip(127.0.0.1) port(514));
      };
```

Please consult your syslog daemon's manual for more information. If no logs are seen in the system's log files, please consider the following tests :

  - restart haproxy. Each frontend and backend logs one line indicating it's
    starting. If these logs are received, it means logs are working.

  - run "strace -tt -s100 -etrace=sendmsg -p <haproxy's pid>" and perform some
    activity that you expect to be logged. You should see the log messages
    being sent using sendmsg() there. If they don't appear, restart using
    strace on top of haproxy. If you still see no logs, it definitely means
    that something is wrong in your configuration.

  - run tcpdump to watch for port 514, for example on the loopback interface if
    the traffic is being sent locally : "tcpdump -As0 -ni lo port 514". If the
    packets are seen there, it's the proof they're sent then the syslogd daemon
    needs to be troubleshooted.

While traffic logs are sent from the frontends (where the incoming connections are accepted), backends also need to be able to send logs in order to report a server state change consecutive to a health check. Please consult HAProxy's

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

configuration manual for more information regarding all possible log settings.

It is convenient to chose a facility that is not used by other daemons. HAProxy examples often suggest "local0" for traffic logs and "local1" for admin logs because they're never seen in field. A single facility would be enough as well. Having separate logs is convenient for log analysis, but it's also important to remember that logs may sometimes convey confidential information, and as such they must not be mixed with other logs that may accidentally be handed out to unauthorized people.

For in-field troubleshooting without impacting the server's capacity too much, it is recommended to make use of the "halog" utility provided with HAProxy. This is sort of a grep-like utility designed to process HAProxy log files at a very fast data rate. Typical figures range between 1 and 2 GB of logs per second. It is capable of extracting only certain logs (eg: search for some classes of HTTP status codes, connection termination status, search by response time ranges, look for errors only), count lines, limit the output to a number of lines, and perform some more advanced statistics such as sorting servers by response time or error counts, sorting URLs by time or count, sorting client addresses by access count, and so on. It is pretty convenient to quickly spot anomalies such as a bot looping on the site, and block them.

9. Statistics and monitoring

It is possible to query HAProxy about its status. The most commonly used mechanism is the HTTP statistics page. This page also exposes an alternative CSV output format for monitoring tools. The same format is provided on the Unix socket.

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

9.1. CSV format

The statistics may be consulted either from the unix socket or from the HTTP page. Both means provide a CSV format whose fields follow. The first line begins with a sharp ('#') and has one word per comma-delimited field which represents the title of the column. All other lines starting at the second one use a classical CSV format using a comma as the delimiter, and the double quote ('"') as an optional text delimiter, but only if the enclosed text is ambiguous (if it contains a quote or a comma). The double-quote character ('"') in the text is doubled ('""'), which is the format that most tools recognize. Please do not insert any column before these ones in order not to break tools which use hard-coded column positions.

In brackets after each field name are the types which may have a value for that field. The types are L (Listeners), F (Frontends), B (Backends), and S (Servers).

0. pxname [LFBS]: proxy name
1. svname [LFBS]: service name (FRONTEND for frontend, BACKEND for backend, any name for server/listener)
2. qcur [..BS]: current queued requests. For the backend this reports the number queued without a server assigned.
3. qmax [..BS]: max value of qcur
4. scur [LFBS]: current sessions
5. smax [LFBS]: max sessions
6. slim [LFBS]: configured session limit
7. stot [LFBS]: cumulative number of sessions
8. bin [LFBS]: bytes in
9. bout [LFBS]: bytes out
10. dreq [LFB.]: requests denied because of security concerns.
    - For tcp this is because of a matched tcp-request content rule.
    - For http this is because of a matched http-request or tarpit rule.
11. dresp [LFBS]: responses denied because of security concerns.
    - For http this is because of a matched http-request rule, or "option checkcache".
12. ereq [LF..]: request errors. Some of the possible causes are:
    - early termination from the client, before the request has been sent.
    - read error from the client
    - client timeout
    - client closed connection
    - various bad requests from the client.
    - request was tarpitted.
13. econ [..BS]: number of requests that encountered an error trying to connect to a backend server. The backend stat is the sum of the stat for all servers of that backend, plus any connection errors not associated with a particular server (such as the backend having no active servers).
14. eresp [..BS]: response errors. srv_abrt will be counted here also. Some other errors are:
    - write error on the client socket (won't be counted for the server stat)
    - failure applying filters to the response.
15. wretr [..BS]: number of times a connection to a server was retried.
16. wredis [..BS]: number of times a request was redispatched to another server. The server value counts the number of times that server was switched away from.
17. status [LFBS]: status (UP/DOWN/NOLB/MAINT/MAINT(via)/MAINT(resolution)...)
18. weight [..BS]: total weight (backend), server weight (server)
19. act [..BS]: number of active servers (backend), server is active (server)
20. bck [..BS]: number of backup servers (backend), server is backup (server)
21. chkfail [...S]: number of failed checks. (Only counts checks failed when the server is up.)
22. chkdown [..BS]: number of UP->DOWN transitions. The backend counter counts transitions to the whole backend being down, rather than the sum of the counters for each server.
23. lastchg [..BS]: number of seconds since the last UP<->DOWN transition
24. downtime [..BS]: total downtime (in seconds). The value for the backend is the downtime for the whole backend, not the sum of the server downtime.
25. qlimit [...S]: configured maxqueue for the server, or nothing in the value is 0 (default, meaning no limit)

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

26. pid [LFBS]: process id (0 for first instance, 1 for second, ...)
27. iid [LFBS]: unique proxy id
28. sid [L..S]: server id (unique inside a proxy)
29. throttle [...S]: current throttle percentage for the server, when slowstart is active, or no value if not in slowstart.
30. lbtot [..BS]: total number of times a server was selected, either for new sessions, or when re-dispatching. The server counter is the number of times that server was selected.
31. tracked [...S]: id of proxy/server if tracking is enabled.
32. type [LFBS]: (0=frontend, 1=backend, 2=server, 3=socket/listener)
33. rate [.FBS]: number of sessions per second over last elapsed second
34. rate_lim [.F..]: configured limit on new sessions per second
35. rate_max [.FBS]: max number of new sessions per second
36. check_status [...S]: status of last health check, one of:
    ```
    UNK     -> unknown
    INI     -> initializing
    SOCKERR -> socket error
    L4OK    -> check passed on layer 4, no upper layers testing enabled
    L4TOUT  -> layer 1-4 timeout
    L4CON   -> layer 1-4 connection problem, for example
               "Connection refused" (tcp rst) or "No route to host" (icmp)
    L6OK    -> check passed on layer 6
    L6TOUT  -> layer 6 (SSL) timeout
    L6RSP   -> layer 6 invalid response - protocol error
    L7OK    -> check passed on layer 7
    L7OKC   -> check conditionally passed on layer 7, for example 404 with
               disable-on-404
    L7TOUT  -> layer 7 (HTTP/SMTP) timeout
    L7RSP   -> layer 7 invalid response - protocol error
    L7STS   -> layer 7 response error, for example HTTP 5xx
    ```
    Notice: If a check is currently running, the last known status will be reported, prefixed with "* ". e. g. "* L7OK".
37. check_code [...S]: layer5-7 code, if available
38. check_duration [...S]: time in ms took to finish last health check
39. hrsp_1xx [.FBS]: http responses with 1xx code
40. hrsp_2xx [.FBS]: http responses with 2xx code
41. hrsp_3xx [.FBS]: http responses with 3xx code
42. hrsp_4xx [.FBS]: http responses with 4xx code
43. hrsp_5xx [.FBS]: http responses with 5xx code
44. hrsp_other [.FBS]: http responses with other codes (protocol error)
45. hanafail [...S]: failed health checks details
46. req_rate [.F..]: HTTP requests per second over last elapsed second
47. req_rate_max [.F..]: max number of HTTP requests per second observed
48. req_tot [.FB.]: total number of HTTP requests received
49. cli_abrt [..BS]: number of data transfers aborted by the client
50. srv_abrt [..BS]: number of data transfers aborted by the server (inc. in eresp)
51. comp_in [.FB.]: number of HTTP response bytes fed to the compressor
52. comp_out [.FB.]: number of HTTP response bytes emitted by the compressor
53. comp_byp [.FB.]: number of bytes that bypassed the HTTP compressor (CPU/BW limit)
54. comp_rsp [.FB.]: number of HTTP responses that were compressed
55. lastsess [..BS]: number of seconds since last session assigned to server/backend
56. last_chk [...S]: last health check contents or textual error
57. last_agt [...S]: last agent check contents or textual error
58. qtime [..BS]: the average queue time in ms over the 1024 last requests
59. ctime [..BS]: the average connect time in ms over the 1024 last requests
60. rtime [..BS]: the average response time in ms over the 1024 last requests (0 for TCP)
61. ttime [..BS]: the average total session time in ms over the 1024 last requests
62. agent_status [...S]: status of last agent check, one of:
    ```
    UNK     -> unknown
    INI     -> initializing
    SOCKERR -> socket error
    L4OK    -> check passed on layer 4, no upper layers testing enabled
    L4TOUT  -> layer 1-4 timeout
    L4CON   -> layer 1-4 connection problem, for example
    ```

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
                "Connection refused" (tcp rst) or "No route to host" (icmp)
        L7OK    -> agent reported "up"
        L7STS   -> agent reported "fail", "stop", or "down"
63. agent_code [...S]: numeric code reported by agent if any (unused for now)
64. agent_duration [...S]: time in ms taken to finish last check
65. check_desc [...S]: short human-readable description of check_status
66. agent_desc [...S]: short human-readable description of agent_status
67. check_rise [...S]: server's "rise" parameter used by checks
68. check_fall [...S]: server's "fall" parameter used by checks
69. check_health [...S]: server's health check value between 0 and rise+fall-1
70. agent_rise [...S]: agent's "rise" parameter, normally 1
71. agent_fall [...S]: agent's "fall" parameter, normally 1
72. agent_health [...S]: agent's health parameter, between 0 and rise+fall-1
73. addr [L..S]: address:port or "unix". IPv6 has brackets around the address.
74: cookie [..BS]: server's cookie value or backend's cookie name
75: mode [LFBS]: proxy mode (tcp, http, health, unknown)
76: algo [..B.]: load balancing algorithm
77: conn_rate [.F..]: number of connections over the last elapsed second
78: conn_rate_max [.F..]: highest known conn_rate
79: conn_tot [.F..]: cumulative number of connections
80: intercepted [.FB.]: cum. number of intercepted requests (monitor, stats)
81: dcon [LF..]: requests denied by "tcp-request connection" rules
82: dses [LF..]: requests denied by "tcp-request session" rules
83: wrew [LFBS]: cumulative number of failed header rewriting warnings
84: connect [..BS]: cumulative number of connection establishment attempts
85: reuse [..BS]: cumulative number of connection reuses
86: cache_lookups [.FB.]: cumulative number of cache lookups
87: cache_hits [.FB.]: cumulative number of cache hits
88: srv_icur [...S]: current number of idle connections available for reuse
89: src_ilim [...S]: limit on the number of available idle connections
90. qtime_max [..BS]: the maximum observed queue time in ms
91. ctime_max [..BS]: the maximum observed connect time in ms
92. rtime_max [..BS]: the maximum observed response time in ms (0 for TCP)
93. ttime_max [..BS]: the maximum observed total session time in ms
```

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

9.2. Typed output format

Both "show info" and "show stat" support a mode where each output value comes with its type and sufficient information to know how the value is supposed to be aggregated between processes and how it evolves.

In all cases, the output consists in having a single value per line with all the information split into fields delimited by colons (':').

The first column designates the object or metric being dumped. Its format is specific to the command producing this output and will not be described in this section. Usually it will consist in a series of identifiers and field names.

The second column contains 3 characters respectively indicating the origin, the nature and the scope of the value being reported. The first character (the origin) indicates where the value was extracted from. Possible characters are :

M The value is a metric. It is valid at one instant any may change depending on its nature .

S The value is a status. It represents a discrete value which by definition cannot be aggregated. It may be the status of a server ("UP" or "DOWN"), the PID of the process, etc.

K The value is a sorting key. It represents an identifier which may be used to group some values together because it is unique among its class. All internal identifiers are keys. Some names can be listed as keys if they are unique (eg: a frontend name is unique). In general keys come from the configuration, even though some of them may automatically be assigned. For most purposes keys may be considered as equivalent to configuration.

C The value comes from the configuration. Certain configuration values make sense on the output, for example a concurrent connection limit or a cookie name. By definition these values are the same in all processes started from the same configuration file.

P The value comes from the product itself. There are very few such values, most common use is to report the product name, version and release date. These elements are also the same between all processes.

The second character (the nature) indicates the nature of the information carried by the field in order to let an aggregator decide on what operation to use to aggregate multiple values. Possible characters are :

A The value represents an age since a last event. This is a bit different from the duration in that an age is automatically computed based on the current date. A typical example is how long ago did the last session happen on a server. Ages are generally aggregated by taking the minimum value and do not need to be stored.

a The value represents an already averaged value. The average response times and server weights are of this nature. Averages can typically be averaged between processes.

C The value represents a cumulative counter. Such measures perpetually increase until they wrap around. Some monitoring protocols need to tell the difference between a counter and a gauge to report a different type. In general counters may simply be summed since they represent events or volumes. Examples of metrics of this nature are connection counts or byte counts.

D The value represents a duration for a status. There are a few usages of this, most of them include the time taken by the last health check and the time a server has spent down. Durations are generally not summed, most of the time the maximum will be retained to compute an SLA.

G The value represents a gauge. It's a measure at one instant. The memory usage or the current number of active connections are of this nature. Metrics of this type are typically summed during aggregation.

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
  L   The value represents a limit (generally a configured one). By nature,
      limits are harder to aggregate since they are specific to the point where
      they were retrieved. In certain situations they may be summed or be kept
      separate.

  M   The value represents a maximum. In general it will apply to a gauge and
      keep the highest known value. An example of such a metric could be the
      maximum amount of concurrent connections that was encountered in the
      product's life time. To correctly aggregate maxima, you are supposed to
      output a range going from the maximum of all maxima and the sum of all
      of them. There is indeed no way to know if they were encountered
      simultaneously or not.

  m   The value represents a minimum. In general it will apply to a gauge and
      keep the lowest known value. An example of such a metric could be the
      minimum amount of free memory pools that was encountered in the product's
      life time. To correctly aggregate minima, you are supposed to output a
      range going from the minimum of all minima and the sum of all of them.
      There is indeed no way to know if they were encountered simultaneously
      or not.

  N   The value represents a name, so it is a string. It is used to report
      proxy names, server names and cookie names. Names have configuration or
      keys as their origin and are supposed to be the same among all processes.

  O   The value represents a free text output. Outputs from various commands,
      returns from health checks, node descriptions are of such nature.

  R   The value represents an event rate. It's a measure at one instant. It is
      quite similar to a gauge except that the recipient knows that this measure
      moves slowly and may decide not to keep all values. An example of such a
      metric is the measured amount of connections per second. Metrics of this
      type are typically summed during aggregation.

  T   The value represents a date or time. A field emitting the current date
      would be of this type. The method to aggregate such information is left
      as an implementation choice. For now no field uses this type.

The third character (the scope) indicates what extent the value reflects. Some
elements may be per process while others may be per configuration or per system.
The distinction is important to know whether or not a single value should be
kept during aggregation or if values have to be aggregated. The following
characters are currently supported :

  C   The value is valid for a whole cluster of nodes, which is the set of nodes
      communicating over the peers protocol. An example could be the amount of
      entries present in a stick table that is replicated with other peers. At
      the moment no metric use this scope.

  P   The value is valid only for the process reporting it. Most metrics use
      this scope.

  S   The value is valid for the whole service, which is the set of processes
      started together from the same configuration file. All metrics originating
      from the configuration use this scope. Some other metrics may use it as
      well for some shared resources (eg: shared SSL cache statistics).

  s   The value is valid for the whole system, such as the system's hostname,
      current date or resource usage. At the moment this scope is not used by
      any metric.

Consumers of these information will generally have enough of these 3 characters
to determine how to accurately report aggregated information across multiple
processes.

After this column, the third column indicates the type of the field, among "s32"
(signed 32-bit integer), "s64" (signed 64-bit integer), "u32" (unsigned 32-bit
integer), "u64" (unsigned 64-bit integer), "str" (string). It is important to
```

know the type before parsing the value in order to properly read it. For example
a string containing only digits is still a string an not an integer (eg: an
error code extracted by a check).

Then the fourth column is the value itself, encoded according to its type.
Strings are dumped as-is immediately after the colon without any leading space.
If a string contains a colon, it will appear normally. This means that the
output should not be exclusively split around colons or some check outputs
or server addresses might be truncated.

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

9.3. Unix Socket commands

The stats socket is not enabled by default. In order to enable it, it is necessary to add one line in the global section of the haproxy configuration. A second line is recommended to set a larger timeout, always appreciated when issuing commands by hand :

```
    global
        stats socket /var/run/haproxy.sock mode 600 level admin
        stats timeout 2m
```

It is also possible to add multiple instances of the stats socket by repeating the line, and make them listen to a TCP port instead of a UNIX socket. This is never done by default because this is dangerous, but can be handy in some situations :

```
    global
        stats socket /var/run/haproxy.sock mode 600 level admin
        stats socket ipv4@192.168.0.1:9999 level admin
        stats timeout 2m
```

To access the socket, an external utility such as "socat" is required. Socat is a swiss-army knife to connect anything to anything. We use it to connect terminals to the socket, or a couple of stdin/stdout pipes to it for scripts. The two main syntaxes we'll use are the following :

```
    # socat /var/run/haproxy.sock stdio
    # socat /var/run/haproxy.sock readline
```

The first one is used with scripts. It is possible to send the output of a script to haproxy, and pass haproxy's output to another script. That's useful for retrieving counters or attack traces for example.

The second one is only useful for issuing commands by hand. It has the benefit that the terminal is handled by the readline library which supports line editing and history, which is very convenient when issuing repeated commands (eg: watch a counter).

The socket supports two operation modes :
  - interactive
  - non-interactive

The non-interactive mode is the default when socat connects to the socket. In this mode, a single line may be sent. It is processed as a whole, responses are sent back, and the connection closes after the end of the response. This is the mode that scripts and monitoring tools use. It is possible to send multiple commands in this mode, they need to be delimited by a semi-colon (';'). For example :

```
    # echo "show info;show stat;show table" | socat /var/run/haproxy stdio
```

If a command needs to use a semi-colon or a backslash (eg: in a value), it must be preceded by a backslash ('\').

The interactive mode displays a prompt ('>') and waits for commands to be entered on the line, then processes them, and displays the prompt again to wait for a new command. This mode is entered via the "prompt" command which must be sent on the first line in non-interactive mode. The mode is a flip switch, if "prompt" is sent in interactive mode, it is disabled and the connection closes after processing the last command of the same line.

For this reason, when debugging by hand, it's quite common to start with the "prompt" command :

```
   # socat /var/run/haproxy readline
   prompt
   > show info
   ...
   >
```

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Since multiple commands may be issued at once, haproxy uses the empty line as a delimiter to mark an end of output for each command, and takes care of ensuring that no command can emit an empty line on output. A script can thus easily parse the output even when multiple commands were pipelined on a single line.

Some commands may take an optional payload. To add one to a command, the first line needs to end with the "<<\n" pattern. The next lines will be treated as the payload and can contain as many lines as needed. To validate a command with a payload, it needs to end with an empty line.

Limitations do exist: the length of the whole buffer passed to the CLI must not be greater than tune.bfsize and the pattern "<<" must not be glued to the last word of the line.

When entering a paylod while in interactive mode, the prompt will change from "> " to "+ ".

It is important to understand that when multiple haproxy processes are started on the same sockets, any process may pick up the request and will output its own stats.

The list of commands currently supported on the stats socket is provided below. If an unknown command is sent, haproxy displays the usage message which reminds all supported commands. Some commands support a more complex syntax, generally it will explain what part of the command is invalid when this happens.

Some commands require a higher level of privilege to work. If you do not have enough privilege, you will get an error "Permission denied". Please check the "level" option of the "bind" keyword lines in the configuration manual for more information.

add acl <acl> <pattern>

Add an entry into the acl <acl>. <acl> is the #<id> or the <file> returned by "show acl". This command does not verify if the entry already exists. This command cannot be used if the reference <acl> is a file also used with a map. In this case, you must use the command "add map" in place of "add acl".

add map <map> <key> <value>
add map <map> <payload>

Add an entry into the map <map> to associate the value <value> to the key <key>. This command does not verify if the entry already exists. It is mainly used to fill a map after a clear operation. Note that if the reference <map> is a file and is shared with a map, this map will contain also a new pattern entry. Using the payload syntax it is possible to add multiple key/value pairs by entering them on separate lines. On each new line, the first word is the key and the rest of the line is considered to be the value which can even contains spaces.

Example:

```
# socat /tmp/sock1 -
prompt

> add map #-1 <<
+ key1 value1
+ key2 value2 with spaces
+ key3 value3 also with spaces
+ key4 value4

>
```

clear counters

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Clear the max values of the statistics counters in each proxy (frontend & backend) and in each server. The accumulated counters are not affected. The internal activity counters reported by "show activity" are also reset. This can be used to get clean counters after an incident, without having to restart nor to clear traffic counters. This command is restricted and can only be issued on sockets configured for levels "operator" or "admin".

clear counters all

Clear all statistics counters in each proxy (frontend & backend) and in each server. This has the same effect as restarting. This command is restricted and can only be issued on sockets configured for level "admin".

clear acl <acl>

Remove all entries from the acl <acl>. <acl> is the #<id> or the <file> returned by "show acl". Note that if the reference <acl> is a file and is shared with a map, this map will be also cleared.

clear map <map>

Remove all entries from the map <map>. <map> is the #<id> or the <file> returned by "show map". Note that if the reference <map> is a file and is shared with a acl, this acl will be also cleared.

clear table <table> [data.<type> <operator> <value>] | [key <key>

Remove entries from the stick-table <table>.

This is typically used to unblock some users complaining they have been abusively denied access to a service, but this can also be used to clear some stickiness entries matching a server that is going to be replaced (see "show table" below for details). Note that sometimes, removal of an entry will be refused because it is currently tracked by a session. Retrying a few seconds later after the session ends is usual enough.

In the case where no options arguments are given all entries will be removed.

When the "data." form is used entries matching a filter applied using the stored data (see "stick-table" in section 4.2) are removed. A stored data type must be specified in <type>, and this data type must be stored in the table otherwise an error is reported. The data is compared according to <operator> with the 64-bit integer <value>. Operators are the same as with the ACLs :

- eq : match entries whose data is equal to this value
- ne : match entries whose data is not equal to this value
- le : match entries whose data is less than or equal to this value
- ge : match entries whose data is greater than or equal to this value
- lt : match entries whose data is less than this value
- gt : match entries whose data is greater than this value

When the key form is used the entry <key> is removed. The key must be of the same type as the table, which currently is limited to IPv4, IPv6, integer and string.

Example :

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
    $ echo "show table http_proxy" | socat stdio /tmp/sock1
>>> # table: http_proxy, type: ip, size:204800, used:2
>>> 0x80e6a4c: key=127.0.0.1 use=0 exp=3594729 gpc0=0 conn_rate(30000)=1 \
      bytes_out_rate(60000)=187
>>> 0x80e6a80: key=127.0.0.2 use=0 exp=3594740 gpc0=1 conn_rate(30000)=10 \
      bytes_out_rate(60000)=191

    $ echo "clear table http_proxy key 127.0.0.1" | socat stdio /tmp/sock1

    $ echo "show table http_proxy" | socat stdio /tmp/sock1
>>> # table: http_proxy, type: ip, size:204800, used:1
>>> 0x80e6a80: key=127.0.0.2 use=0 exp=3594740 gpc0=1 conn_rate(30000)=10 \
      bytes_out_rate(60000)=191
    $ echo "clear table http_proxy data.gpc0 eq 1" | socat stdio /tmp/sock1
    $ echo "show table http_proxy" | socat stdio /tmp/sock1
>>> # table: http_proxy, type: ip, size:204800, used:1
```

debug dev <command> [args]*

Call a developer-specific command. Only supported when haproxy is built with DEBUG_DEV defined. Supported commands are then listed in the help message. All of these commands require admin privileges, and must never appear on a production system as most of them are unsafe and dangerous.

del acl <acl> [<key>|#<ref>]

Delete all the acl entries from the acl <acl> corresponding to the key <key>. <acl> is the #<id> or the <file> returned by "show acl". If the <ref> is used, this command delete only the listed reference. The reference can be found with listing the content of the acl. Note that if the reference <acl> is a file and is shared with a map, the entry will be also deleted in the map.

del map <map> [<key>|#<ref>]

Delete all the map entries from the map <map> corresponding to the key <key>. <map> is the #<id> or the <file> returned by "show map". If the <ref> is used, this command delete only the listed reference. The reference can be found with listing the content of the map. Note that if the reference <map> is a file and is shared with a acl, the entry will be also deleted in the map.

disable agent <backend>/<server>

Mark the auxiliary agent check as temporarily stopped.

In the case where an agent check is being run as a auxiliary check, due to the agent-check parameter of a server directive, new checks are only initialized when the agent is in the enabled. Thus, disable agent will prevent any new agent checks from begin initiated until the agent re-enabled using enable agent.

When an agent is disabled the processing of an auxiliary agent check that was initiated while the agent was set as enabled is as follows: All results that would alter the weight, specifically "drain" or a weight returned by the agent, are ignored. The processing of agent check is otherwise unchanged.

The motivation for this feature is to allow the weight changing effects of the agent checks to be paused to allow the weight of a server to be configured using set weight without being overridden by the agent.

This command is restricted and can only be issued on sockets configured for level "admin".

disable dynamic-cookie backend <backend>

Disable the generation of dynamic cookies for the backend <backend>

disable frontend <frontend>

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Mark the frontend as temporarily stopped. This corresponds to the mode which is used during a soft restart : the frontend releases the port but can be enabled again if needed. This should be used with care as some non-Linux OSes are unable to enable it back. This is intended to be used in environments where stopping a proxy is not even imaginable but a misconfigured proxy must be fixed. That way it's possible to release the port and bind it into another process to restore operations. The frontend will appear with status "STOP" on the stats page.

The frontend may be specified either by its name or by its numeric ID, prefixed with a sharp ('#').

This command is restricted and can only be issued on sockets configured for level "admin".

disable health <backend>/<server>

Mark the primary health check as temporarily stopped. This will disable sending of health checks, and the last health check result will be ignored. The server will be in unchecked state and considered UP unless an auxiliary agent check forces it down.

This command is restricted and can only be issued on sockets configured for level "admin".

disable server <backend>/<server>

Mark the server DOWN for maintenance. In this mode, no more checks will be performed on the server until it leaves maintenance.
If the server is tracked by other servers, those servers will be set to DOWN during the maintenance.

In the statistics page, a server DOWN for maintenance will appear with a "MAINT" status, its tracking servers with the "MAINT(via)" one.

Both the backend and the server may be specified either by their name or by their numeric ID, prefixed with a sharp ('#').

This command is restricted and can only be issued on sockets configured for level "admin".

enable agent <backend>/<server>

Resume auxiliary agent check that was temporarily stopped.

See "disable agent" for details of the effect of temporarily starting and stopping an auxiliary agent.

This command is restricted and can only be issued on sockets configured for level "admin".

enable dynamic-cookie backend <backend>

Enable the generation of dynamic cookies for the backend <backend>.
A secret key must also be provided.

enable frontend <frontend>

Resume a frontend which was temporarily stopped. It is possible that some of the listening ports won't be able to bind anymore (eg: if another process took them since the 'disable frontend' operation). If this happens, an error is displayed. Some operating systems might not be able to resume a frontend which was disabled.

The frontend may be specified either by its name or by its numeric ID, prefixed with a sharp ('#').

This command is restricted and can only be issued on sockets configured for level "admin".

enable health <backend>/<server>

Resume a primary health check that was temporarily stopped. This will enable
sending of health checks again. Please see "disable health" for details.

This command is restricted and can only be issued on sockets configured for
level "admin".

enable server <backend>/<server>

If the server was previously marked as DOWN for maintenance, this marks the
server UP and checks are re-enabled.

Both the backend and the server may be specified either by their name or by
their numeric ID, prefixed with a sharp ('#').

This command is restricted and can only be issued on sockets configured for
level "admin".

get map <map> <value>

get acl <acl> <value>

Lookup the value <value> in the map <map> or in the ACL <acl>. <map> or <acl>
are the #<id> or the <file> returned by "show map" or "show acl". This command
returns all the matching patterns associated with this map. This is useful for
debugging maps and ACLs. The output format is composed by one line par
matching type. Each line is composed by space-delimited series of words.

The first two words are:

```
   <match method>:   The match method applied. It can be "found", "bool",
                     "int", "ip", "bin", "len", "str", "beg", "sub", "dir",
                     "dom", "end" or "reg".

   <match result>:   The result. Can be "match" or "no-match".
```

The following words are returned only if the pattern matches an entry.

```
   <index type>:     "tree" or "list". The internal lookup algorithm.

   <case>:           "case-insensitive" or "case-sensitive". The
                     interpretation of the case.

   <entry matched>:  match="<entry>". Return the matched pattern. It is
                     useful with regular expressions.
```

The two last word are used to show the returned value and its type. With the
"acl" case, the pattern doesn't exist.

```
   return=nothing:        No return because there are no "map".
   return="<value>":      The value returned in the string format.
   return=cannot-display: The value cannot be converted as string.

   type="<type>":         The type of the returned sample.
```

get weight <backend>/<server>

Report the current weight and the initial weight of server <server> in
backend <backend> or an error if either doesn't exist. The initial weight is
the one that appears in the configuration file. Both are normally equal
unless the current weight has been changed. Both the backend and the server
may be specified either by their name or by their numeric ID, prefixed with a
sharp ('#').

help

Print the list of known keywords and their basic usage. The same help screen
is also displayed for unknown commands.

prompt

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Toggle the prompt at the beginning of the line and enter or leave interactive mode. In interactive mode, the connection is not closed after a command completes. Instead, the prompt will appear again, indicating the user that the interpreter is waiting for a new command. The prompt consists in a right angle bracket followed by a space "> ". This mode is particularly convenient when one wants to periodically check information such as stats or errors. It is also a good idea to enter interactive mode before issuing a "help" command.

quit

Close the connection when in interactive mode.

set dynamic-cookie-key backend <backend> <value>

Modify the secret key used to generate the dynamic persistent cookies. This will break the existing sessions.

set map <map> [<key>|#<ref>] <value>

Modify the value corresponding to each key <key> in a map <map>. <map> is the #<id> or <file> returned by "show map". If the <ref> is used in place of <key>, only the entry pointed by <ref> is changed. The new value is <value>.

set maxconn frontend <frontend> <value>

Dynamically change the specified frontend's maxconn setting. Any positive value is allowed including zero, but setting values larger than the global maxconn does not make much sense. If the limit is increased and connections were pending, they will immediately be accepted. If it is lowered to a value below the current number of connections, new connections acceptation will be delayed until the threshold is reached. The frontend might be specified by either its name or its numeric ID prefixed with a sharp ('#').

set maxconn server <backend/server> <value>

Dynamically change the specified server's maxconn setting. Any positive value is allowed including zero, but setting values larger than the global maxconn does not make much sense.

set maxconn global <maxconn>

Dynamically change the global maxconn setting within the range defined by the initial global maxconn setting. If it is increased and connections were pending, they will immediately be accepted. If it is lowered to a value below the current number of connections, new connections acceptation will be delayed until the threshold is reached. A value of zero restores the initial setting.

set profiling { tasks } { auto | on | off }

Enables or disables CPU profiling for the indicated subsystem. This is equivalent to setting or clearing the "profiling" settings in the "global" section of the configuration file. Please also see "show profiling".

set rate-limit connections global <value>

Change the process-wide connection rate limit, which is set by the global 'maxconnrate' setting. A value of zero disables the limitation. This limit applies to all frontends and the change has an immediate effect. The value is passed in number of connections per second.

set rate-limit http-compression global <value>

Change the maximum input compression rate, which is set by the global 'maxcomprate' setting. A value of zero disables the limitation. The value is passed in number of kilobytes per second. The value is available in the "show info" on the line "CompressBpsRateLim" in bytes.

set rate-limit sessions global <value>

Change the process-wide session rate limit, which is set by the global 'maxsessrate' setting. A value of zero disables the limitation. This limit applies to all frontends and the change has an immediate effect. The value is passed in number of sessions per second.

set rate-limit ssl-sessions global <value>

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Change the process-wide SSL session rate limit, which is set by the global 'maxsslrate' setting. A value of zero disables the limitation. This limit applies to all frontends and the change has an immediate effect. The value is passed in number of sessions per second sent to the SSL stack. It applies before the handshake in order to protect the stack against handshake abuses.

set server <backend>/<server> addr <ip4 or ip6 address> [port <port>

Replace the current IP address of a server by the one provided.
Optionally, the port can be changed using the 'port' parameter.
Note that changing the port also support switching from/to port mapping (notation with +X or -Y), only if a port is configured for the health check.

set server <backend>/<server> agent [up | down]

Force a server's agent to a new state. This can be useful to immediately switch a server's state regardless of some slow agent checks for example. Note that the change is propagated to tracking servers if any.

set server <backend>/<server> agent-addr <addr>

Change addr for servers agent checks. Allows to migrate agent-checks to another address at runtime. You can specify both IP and hostname, it will be resolved.

set server <backend>/<server> agent-send <value>

Change agent string sent to agent check target. Allows to update string while changing server address to keep those two matching.

set server <backend>/<server> health [up | stopping | down]

Force a server's health to a new state. This can be useful to immediately switch a server's state regardless of some slow health checks for example. Note that the change is propagated to tracking servers if any.

set server <backend>/<server> check-port <port>

Change the port used for health checking to <port>

set server <backend>/<server> state [ready | drain | maint]

Force a server's administrative state to a new state. This can be useful to disable load balancing and/or any traffic to a server. Setting the state to "ready" puts the server in normal mode, and the command is the equivalent of the "enable server" command. Setting the state to "maint" disables any traffic to the server as well as any health checks. This is the equivalent of the "disable server" command. Setting the mode to "drain" only removes the server from load balancing but still allows it to be checked and to accept new persistent connections. Changes are propagated to tracking servers if any.

set server <backend>/<server> weight <weight>[%]

Change a server's weight to the value passed in argument. This is the exact equivalent of the "set weight" command below.

set server <backend>/<server> fqdn <FQDN>

Change a server's FQDN to the value passed in argument. This requires the internal run-time DNS resolver to be configured and enabled for this server.

set severity-output [none | number | string]

Change the severity output format of the stats socket connected to for the duration of the current session.

set ssl ocsp-response <response | payload>

This command is used to update an OCSP Response for a certificate (see "crt" on "bind" lines). Same controls are performed as during the initial loading of the response. The <response> must be passed as a base64 encoded string of the DER encoded response from the OCSP server. This command is not supported with BoringSSL.

Example:

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
openssl ocsp -issuer issuer.pem -cert server.pem \
             -host ocsp.issuer.com:80 -respout resp.der
echo "set ssl ocsp-response $(base64 -w 10000 resp.der)" | \
             socat stdio /var/run/haproxy.stat

using the payload syntax:
echo -e "set ssl ocsp-response <<\n$(base64 resp.der)\n" | \
             socat stdio /var/run/haproxy.stat
```

set ssl tls-key <id> <tlskey>

Set the next TLS key for the <id> listener to <tlskey>. This key becomes the ultimate key, while the penultimate one is used for encryption (others just decrypt). The oldest TLS key present is overwritten. <id> is either a numeric #<id> or <file> returned by "show tls-keys". <tlskey> is a base64 encoded 48 or 80 bits TLS ticket key (ex. openssl rand 80 | openssl base64 -A).

set table <table> key <key> [data.<data_type> <value>]*

Create or update a stick-table entry in the table. If the key is not present, an entry is inserted. See stick-table in section 4.2 to find all possible values for <data_type>. The most likely use consists in dynamically entering entries for source IP addresses, with a flag in gpc0 to dynamically block an IP address or affect its quality of service. It is possible to pass multiple data_types in a single call.

set timeout cli <delay>

Change the CLI interface timeout for current connection. This can be useful during long debugging sessions where the user needs to constantly inspect some indicators without being disconnected. The delay is passed in seconds.

set weight <backend>/<server> <weight>[%]

Change a server's weight to the value passed in argument. If the value ends with the '%' sign, then the new weight will be relative to the initially configured weight. Absolute weights are permitted between 0 and 256. Relative weights must be positive with the resulting absolute weight is capped at 256. Servers which are part of a farm running a static load-balancing algorithm have stricter limitations because the weight cannot change once set. Thus for these servers, the only accepted values are 0 and 100% (or 0 and the initial weight). Changes take effect immediately, though certain LB algorithms require a certain amount of requests to consider changes. A typical usage of this command is to disable a server during an update by setting its weight to zero, then to enable it again after the update by setting it back to 100%. This command is restricted and can only be issued on sockets configured for level "admin". Both the backend and the server may be specified either by their name or by their numeric ID, prefixed with a sharp ('#').

show acl [<acl>]

Dump info about acl converters. Without argument, the list of all available acls is returned. If a <acl> is specified, its contents are dumped. <acl> if the #<id> or <file>. The dump format is the same than the map even for the sample value. The data returned are not a list of available ACL, but are the list of all patterns composing any ACL. Many of these patterns can be shared with maps.

show backend

Dump the list of backends available in the running process

show cli level

Display the CLI level of the current CLI session. The result could be 'admin', 'operator' or 'user'. See also the 'operator' and 'user' commands.

Example :

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
$ socat /tmp/sock1 readline
prompt
> operator
> show cli level
operator
> user
> show cli level
user
> operator
Permission denied
```

operator

Decrease the CLI level of the current CLI session to operator. It can't be increase. See also "show cli level"

user

Decrease the CLI level of the current CLI session to user. It can't be increase. See also "show cli level"

show activity

Reports some counters about internal events that will help developers and more generally people who know haproxy well enough to narrow down the causes of reports of abnormal behaviours. A typical example would be a properly running process never sleeping and eating 100% of the CPU. The output fields will be made of one line per metric, and per-thread counters on the same line. These counters are 32-bit and will wrap during the process's life, which is not a problem since calls to this command will typically be performed twice. The fields are purposely not documented so that their exact meaning is verified in the code where the counters are fed. These values are also reset by the "clear counters" command.

show cli sockets

List CLI sockets. The output format is composed of 3 fields separated by spaces. The first field is the socket address, it can be a unix socket, a ipv4 address:port couple or a ipv6 one. Socket of other types won't be dump. The second field describe the level of the socket: 'admin', 'user' or 'operator'. The last field list the processes on which the socket is bound, separated by commas, it can be numbers or 'all'.

Example :

```
$ echo 'show cli sockets' | socat stdio /tmp/sock1
# socket lvl processes
/tmp/sock1 admin all
127.0.0.1:9999 user 2,3,4
127.0.0.2:9969 user 2
[::1]:9999 operator 2
```

show cache

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
List the configured caches and the objects stored in each cache tree.

$ echo 'show cache' | socat stdio /tmp/sock1
0x7f6ac6c5b03a: foobar (shctx:0x7f6ac6c5b000, available blocks:3918)
       1          2             3                              4

1. pointer to the cache structure
2. cache name
3. pointer to the mmap area (shctx)
4. number of blocks available for reuse in the shctx

0x7f6ac6c5b4cc hash:286881868 size:39114 (39 blocks), refcount:9, expire:237
         1               2            3        4            5           6

1. pointer to the cache entry
2. first 32 bits of the hash
3. size of the object in bytes
4. number of blocks used for the object
5. number of transactions using the entry
6. expiration time, can be negative if already expired
```

show env [<name>]

```
Dump one or all environment variables known by the process. Without any
argument, all variables are dumped. With an argument, only the specified
variable is dumped if it exists. Otherwise "Variable not found" is emitted.
Variables are dumped in the same format as they are stored or returned by the
"env" utility, that is, "<name>=<value>". This can be handy when debugging
certain configuration files making heavy use of environment variables to
ensure that they contain the expected values. This command is restricted and
can only be issued on sockets configured for levels "operator" or "admin".
```

show errors [<iid>|<proxy>] [request|response]

```
Dump last known request and response errors collected by frontends and
backends. If <iid> is specified, the limit the dump to errors concerning
either frontend or backend whose ID is <iid>. Proxy ID "-1" will cause
all instances to be dumped. If a proxy name is specified instead, its ID
will be used as the filter. If "request" or "response" is added after the
proxy name or ID, only request or response errors will be dumped. This
command is restricted and can only be issued on sockets configured for
levels "operator" or "admin".

The errors which may be collected are the last request and response errors
caused by protocol violations, often due to invalid characters in header
names. The report precisely indicates what exact character violated the
protocol. Other important information such as the exact date the error was
detected, frontend and backend names, the server name (when known), the
internal session ID and the source address which has initiated the session
are reported too.

All characters are returned, and non-printable characters are encoded. The
most common ones (\t = 9, \n = 10, \r = 13 and \e = 27) are encoded as one
letter following a backslash. The backslash itself is encoded as '\\' to
avoid confusion. Other non-printable characters are encoded '\xNN' where
NN is the two-digits hexadecimal representation of the character's ASCII
code.

Lines are prefixed with the position of their first character, starting at 0
for the beginning of the buffer. At most one input line is printed per line,
and large lines will be broken into multiple consecutive output lines so that
the output never goes beyond 79 characters wide. It is easy to detect if a
line was broken, because it will not end with '\n' and the next line's offset
will be followed by a '+' sign, indicating it is a continuation of previous
line.
```

Example :

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
    $ echo "show errors -1 response" | socat stdio /tmp/sock1
>>> [04/Mar/2009:15:46:56.081] backend http-in (#2) : invalid response
      src 127.0.0.1, session #54, frontend fe-eth0 (#1), server s2 (#1)
      response length 213 bytes, error at position 23:

      00000  HTTP/1.0 200 OK\r\n
      00017  header/bizarre:blah\r\n
      00038  Location: blah\r\n
      00054  Long-line: this is a very long line which should b
      00104+ e broken into multiple lines on the output buffer,
      00154+  otherwise it would be too large to print in a ter
      00204+ minal\r\n
      00211  \r\n

In the example above, we see that the backend "http-in" which has internal
ID 2 has blocked an invalid response from its server s2 which has internal
ID 1. The request was on session 54 initiated by source 127.0.0.1 and
received by frontend fe-eth0 whose ID is 1. The total response length was
213 bytes when the error was detected, and the error was at byte 23. This
is the slash ('/') in header name "header/bizarre", which is not a valid
HTTP character for a header name.
```

show fd [<fd>]

Dump the list of either all open file descriptors or just the one number <fd> if specified. This is only aimed at developers who need to observe internal states in order to debug complex issues such as abnormal CPU usages. One fd is reported per lines, and for each of them, its state in the poller using upper case letters for enabled flags and lower case for disabled flags, using "P" for "polled", "R" for "ready", "A" for "active", the events status using "H" for "hangup", "E" for "error", "O" for "output", "P" for "priority" and "I" for "input", a few other flags like "N" for "new" (just added into the fd cache), "U" for "updated" (received an update in the fd cache), "L" for "linger_risk", "C" for "cloned", then the cached entry position, the pointer to the internal owner, the pointer to the I/O callback and its name when known. When the owner is a connection, the connection flags, and the target are reported (frontend, proxy or server). When the owner is a listener, the listener's state and its frontend are reported. There is no point in using this command without a good knowledge of the internals. It's worth noting that the output format may evolve over time so this output must not be parsed by tools designed to be durable.

show info [typed|json]

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Dump info about haproxy status on current process. If "typed" is passed as an optional argument, field numbers, names and types are emitted as well so that external monitoring products can easily retrieve, possibly aggregate, then report information found in fields they don't know. Each field is dumped on its own line. If "json" is passed as an optional argument then information provided by "typed" output is provided in JSON format as a list of JSON objects. By default, the format contains only two columns delimited by a colon (':'). The left one is the field name and the right one is the value. It is very important to note that in typed output format, the dump for a single object is contiguous so that there is no need for a consumer to store everything at once.

When using the typed output format, each line is made of 4 columns delimited by colons (':'). The first column is a dot-delimited series of 3 elements. The first element is the numeric position of the field in the list (starting at zero). This position shall not change over time, but holes are to be expected, depending on build options or if some fields are deleted in the future. The second element is the field name as it appears in the default "show info" output. The third element is the relative process number starting at 1.

The rest of the line starting after the first colon follows the "typed output format" described in the section above. In short, the second column (after the first ':') indicates the origin, nature and scope of the variable. The third column indicates the type of the field, among "s32", "s64", "u32", "u64" and "str". Then the fourth column is the value itself, which the consumer knows how to parse thanks to column 3 and how to process thanks to column 2.

Thus the overall line format in typed mode is :

```
    <field_pos>.<field_name>.<process_num>:<tags>:<type>:<value>
```

Example :

```
> show info
Name: HAProxy
Version: 1.7-dev1-de52ea-146
Release_date: 2016/03/11
Nbproc: 1
Process_num: 1
Pid: 28105
Uptime: 0d 0h00m04s
Uptime_sec: 4
Memmax_MB: 0
PoolAlloc_MB: 0
PoolUsed_MB: 0
PoolFailed: 0
(...)

> show info typed
0.Name.1:POS:str:HAProxy
1.Version.1:POS:str:1.7-dev1-de52ea-146
2.Release_date.1:POS:str:2016/03/11
3.Nbproc.1:CGS:u32:1
4.Process_num.1:KGP:u32:1
5.Pid.1:SGP:u32:28105
6.Uptime.1:MDP:str:0d 0h00m08s
7.Uptime_sec.1:MDP:u32:8
8.Memmax_MB.1:CLP:u32:0
9.PoolAlloc_MB.1:MGP:u32:0
10.PoolUsed_MB.1:MGP:u32:0
11.PoolFailed.1:MCP:u32:0
(...)
```

In the typed format, the presence of the process ID at the end of the first column makes it very easy to visually aggregate outputs from multiple processes.

Example :

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
$ ( echo show info typed | socat /var/run/haproxy.sock1 ;    \
    echo show info typed | socat /var/run/haproxy.sock2 ) |  \
  sort -t . -k 1,1n -k 2,2 -k 3,3n
0.Name.1:POS:str:HAProxy
0.Name.2:POS:str:HAProxy
1.Version.1:POS:str:1.7-dev1-868ab3-148
1.Version.2:POS:str:1.7-dev1-868ab3-148
2.Release_date.1:POS:str:2016/03/11
2.Release_date.2:POS:str:2016/03/11
3.Nbproc.1:CGS:u32:2
3.Nbproc.2:CGS:u32:2
4.Process_num.1:KGP:u32:1
4.Process_num.2:KGP:u32:2
5.Pid.1:SGP:u32:30120
5.Pid.2:SGP:u32:30121
6.Uptime.1:MDP:str:0d 0h01m28s
6.Uptime.2:MDP:str:0d 0h01m28s
(...)
```

The format of JSON output is described in a schema which may be output using "show schema json".

The JSON output contains no extra whitespace in order to reduce the volume of output. For human consumption passing the output through a pretty printer may be helpful. Example :

```
$ echo "show info json" | socat /var/run/haproxy.sock stdio | \
  python -m json.tool
```

The JSON output contains no extra whitespace in order to reduce the volume of output. For human consumption passing the output through a pretty printer may be helpful. Example :

```
$ echo "show info json" | socat /var/run/haproxy.sock stdio | \
  python -m json.tool
```

show map [<map>]

Dump info about map converters. Without argument, the list of all available maps is returned. If a <map> is specified, its contents are dumped. <map> is the #<id> or <file>. The first column is a unique identifier. It can be used as reference for the operation "del map" and "set map". The second column is the pattern and the third column is the sample if available. The data returned are not directly a list of available maps, but are the list of all patterns composing any map. Many of these patterns can be shared with ACL.

show peers [<peers section>]

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
Dump info about the peers configured in "peers" sections. Without argument,
the list of the peers belonging to all the "peers" sections are listed. If
<peers section> is specified, only the information about the peers belonging
to this "peers" section are dumped.

Here are two examples of outputs where hostA, hostB and hostC peers belong to
"sharedlb" peers sections. Only hostA and hostB are connected. Only hostA has
sent data to hostB.

$ echo "show peers" | socat - /tmp/hostA
0x55deb0224320: [15/Apr/2019:11:28:01] id=sharedlb state=0 flags=0x3 \
  resync_timeout=<PAST> task_calls=45122
    0x55deb022b540: id=hostC(remote) addr=127.0.0.12:10002 status=CONN \
      reconnect=4s confirm=0
      flags=0x0
    0x55deb022a440: id=hostA(local) addr=127.0.0.10:10000 status=NONE \
      reconnect=<NEVER> confirm=0
      flags=0x0
    0x55deb0227d70: id=hostB(remote) addr=127.0.0.11:10001 status=ESTA
      reconnect=2s confirm=0
      flags=0x20000200 appctx:0x55deb028fba0 st0=7 st1=0 task_calls=14456 \
        state=EST
      xprt=RAW src=127.0.0.1:37257 addr=127.0.0.10:10000
      remote_table:0x55deb0224a10 id=stkt local_id=1 remote_id=1
      last_local_table:0x55deb0224a10 id=stkt local_id=1 remote_id=1
      shared tables:
        0x55deb0224a10 local_id=1 remote_id=1 flags=0x0 remote_data=0x65
          last_acked=0 last_pushed=3 last_get=0 teaching_origin=0 update=3
          table:0x55deb022d6a0 id=stkt update=3 localupdate=3 \
            commitupdate=3 syncing=0

$ echo "show peers" | socat - /tmp/hostB
0x55871b5ab320: [15/Apr/2019:11:28:03] id=sharedlb state=0 flags=0x3 \
  resync_timeout=<PAST> task_calls=3
    0x55871b5b2540: id=hostC(remote) addr=127.0.0.12:10002 status=CONN \
      reconnect=3s confirm=0
      flags=0x0
    0x55871b5b1440: id=hostB(local) addr=127.0.0.11:10001 status=NONE \
      reconnect=<NEVER> confirm=0
      flags=0x0
    0x55871b5aed70: id=hostA(remote) addr=127.0.0.10:10000 status=ESTA \
      reconnect=2s confirm=0
      flags=0x20000200 appctx:0x7fa46800ee00 st0=7 st1=0 task_calls=62356 \
        state=EST
      remote_table:0x55871b5ab960 id=stkt local_id=1 remote_id=1
      last_local_table:0x55871b5ab960 id=stkt local_id=1 remote_id=1
      shared tables:
        0x55871b5ab960 local_id=1 remote_id=1 flags=0x0 remote_data=0x65
          last_acked=3 last_pushed=0 last_get=3 teaching_origin=0 update=0
          table:0x55871b5b46a0 id=stkt update=1 localupdate=0 \
            commitupdate=0 syncing=0
```

show pools

```
Dump the status of internal memory pools. This is useful to track memory
usage when suspecting a memory leak for example. It does exactly the same
as the SIGQUIT when running in foreground except that it does not flush
the pools.
```

show profiling

```
Dumps the current profiling settings, one per line, as well as the command
needed to change them.
```

show resolvers [<resolvers section id>]

```
Dump statistics for the given resolvers section, or all resolvers sections
if no section is supplied.

For each name server, the following counters are reported:
  sent: number of DNS requests sent to this server
  valid: number of DNS valid responses received from this server
  update: number of DNS responses used to update the server's IP address
  cname: number of CNAME responses
  cname_error: CNAME errors encountered with this server
  any_err: number of empty response (IE: server does not support ANY type)
  nx: non existent domain response received from this server
  timeout: how many time this server did not answer in time
  refused: number of requests refused by this server
  other: any other DNS errors
  invalid: invalid DNS response (from a protocol point of view)
  too_big: too big response
  outdated: number of response arrived too late (after an other name server)
```

show servers state [<backend>]

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
Dump the state of the servers found in the running configuration. A backend
name or identifier may be provided to limit the output to this backend only.

The dump has the following format:
 - first line contains the format version (1 in this specification);
 - second line contains the column headers, prefixed by a sharp ('#');
 - third line and next ones contain data;
 - each line starting by a sharp ('#') is considered as a comment.

Since multiple versions of the output may co-exist, below is the list of
fields and their order per file format version :
 1:
   be_id:                       Backend unique id.
   be_name:                     Backend label.
   srv_id:                      Server unique id (in the backend).
   srv_name:                    Server label.
   srv_addr:                    Server IP address.
   srv_op_state:                Server operational state (UP/DOWN/...).
                                  0 = SRV_ST_STOPPED
                                    The server is down.
                                  1 = SRV_ST_STARTING
                                    The server is warming up (up but
                                    throttled).
                                  2 = SRV_ST_RUNNING
                                    The server is fully up.
                                  3 = SRV_ST_STOPPING
                                    The server is up but soft-stopping
                                    (eg: 404).
   srv_admin_state:             Server administrative state (MAINT/DRAIN/...).
                                The state is actually a mask of values :
                                  0x01 = SRV_ADMF_FMAINT
                                    The server was explicitly forced into
                                    maintenance.
                                  0x02 = SRV_ADMF_IMAINT
                                    The server has inherited the maintenance
                                    status from a tracked server.
                                  0x04 = SRV_ADMF_CMAINT
                                    The server is in maintenance because of
                                    the configuration.
                                  0x08 = SRV_ADMF_FDRAIN
                                    The server was explicitly forced into
                                    drain state.
                                  0x10 = SRV_ADMF_IDRAIN
                                    The server has inherited the drain status
                                    from a tracked server.
                                  0x20 = SRV_ADMF_RMAINT
                                    The server is in maintenance because of an
                                    IP address resolution failure.
                                  0x40 = SRV_ADMF_HMAINT
                                    The server FQDN was set from stats socket.

   srv_uweight:                 User visible server's weight.
   srv_iweight:                 Server's initial weight.
   srv_time_since_last_change:  Time since last operational change.
   srv_check_status:            Last health check status.
   srv_check_result:            Last check result (FAILED/PASSED/...).
                                  0 = CHK_RES_UNKNOWN
                                    Initialized to this by default.
                                  1 = CHK_RES_NEUTRAL
                                    Valid check but no status information.
                                  2 = CHK_RES_FAILED
                                    Check failed.
                                  3 = CHK_RES_PASSED
                                    Check succeeded and server is fully up
                                    again.
                                  4 = CHK_RES_CONDPASS
                                    Check reports the server doesn't want new
                                    sessions.
   srv_check_health:            Checks rise / fall current counter.
```

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.


```
srv_check_state:          State of the check (ENABLED/PAUSED/...).
                          The state is actually a mask of values :
                            0x01 = CHK_ST_INPROGRESS
                              A check is currently running.
                            0x02 = CHK_ST_CONFIGURED
                              This check is configured and may be
                              enabled.
                            0x04 = CHK_ST_ENABLED
                              This check is currently administratively
                              enabled.
                            0x08 = CHK_ST_PAUSED
                              Checks are paused because of maintenance
                              (health only).
srv_agent_state:          State of the agent check (ENABLED/PAUSED/...).
                          This state uses the same mask values as
                          "srv_check_state", adding this specific one :
                            0x10 = CHK_ST_AGENT
                              Check is an agent check (otherwise it's a
                              health check).
bk_f_forced_id:           Flag to know if the backend ID is forced by
                          configuration.
srv_f_forced_id:          Flag to know if the server's ID is forced by
                          configuration.
srv_fqdn:                 Server FQDN.
srv_port:                 Server port.
srvrecord:                DNS SRV record associated to this SRV.
```

show sess

```
Dump all known sessions. Avoid doing this on slow connections as this can
be huge. This command is restricted and can only be issued on sockets
configured for levels "operator" or "admin". Note that on machines with
quickly recycled connections, it is possible that this output reports less
entries than really exist because it will dump all existing sessions up to
the last one that was created before the command was entered; those which
die in the mean time will not appear.
```

show sess <id>

```
Display a lot of internal information about the specified session identifier.
This identifier is the first field at the beginning of the lines in the dumps
of "show sess" (it corresponds to the session pointer). Those information are
useless to most users but may be used by haproxy developers to troubleshoot a
complex bug. The output format is intentionally not documented so that it can
freely evolve depending on demands. You may find a description of all fields
returned in src/dumpstats.c

The special id "all" dumps the states of all sessions, which must be avoided
as much as possible as it is highly CPU intensive and can take a lot of time.
```

show stat [{<iid>|<proxy>} <type> <sid>] [typed|json]

```
Dump statistics using the CSV format; using the extended typed output
format described in the section above if "typed" is passed after the
other arguments; or in JSON if "json" is passed after the other arguments
. By passing <id>, <type> and <sid>, it is possible to dump only selected
items :
  - <iid> is a proxy ID, -1 to dump everything. Alternatively, a proxy name
    <proxy> may be specified. In this case, this proxy's ID will be used as
    the ID selector.
  - <type> selects the type of dumpable objects : 1 for frontends, 2 for
     backends, 4 for servers, -1 for everything. These values can be ORed,
     for example:
        1 + 2     = 3   -> frontend + backend.
        1 + 2 + 4 = 7   -> frontend + backend + server.
  - <sid> is a server ID, -1 to dump everything from the selected proxy.
```

Example :

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
    $ echo "show info;show stat" | socat stdio unix-connect:/tmp/sock1
>>> Name: HAProxy
    Version: 1.4-dev2-49
    Release_date: 2009/09/23
    Nbproc: 1
    Process_num: 1
    (...)

    # pxname,svname,qcur,qmax,scur,smax,slim,stot,bin,bout,dreq,  (...)
    stats,FRONTEND,,,0,0,1000,0,0,0,0,0,,,,,OPEN,,,,,,,,,1,1,0, (...)
    stats,BACKEND,0,0,0,0,1000,0,0,0,0,0,,0,0,0,0,UP,0,0,0,,0,250,(...)
    (...)
    www1,BACKEND,0,0,0,0,1000,0,0,0,0,0,,0,0,0,0,UP,1,1,0,,0,250, (...)

    $
```

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

In this example, two commands have been issued at once. That way it's easy to find which process the stats apply to in multi-process mode. This is not needed in the typed output format as the process number is reported on each line. Notice the empty line after the information output which marks the end of the first block. A similar empty line appears at the end of the second block (stats) so that the reader knows the output has not been truncated.

When "typed" is specified, the output format is more suitable to monitoring tools because it provides numeric positions and indicates the type of each output field. Each value stands on its own line with process number, element number, nature, origin and scope. This same format is available via the HTTP stats by passing ";typed" after the URI. It is very important to note that in typed output format, the dump for a single object is contiguous so that there is no need for a consumer to store everything at once.

When using the typed output format, each line is made of 4 columns delimited by colons (':'). The first column is a dot-delimited series of 5 elements. The first element is a letter indicating the type of the object being described. At the moment the following object types are known : 'F' for a frontend, 'B' for a backend, 'L' for a listener, and 'S' for a server. The second element The second element is a positive integer representing the unique identifier of the proxy the object belongs to. It is equivalent to the "iid" column of the CSV output and matches the value in front of the optional "id" directive found in the frontend or backend section. The third element is a positive integer containing the unique object identifier inside the proxy, and corresponds to the "sid" column of the CSV output. ID 0 is reported when dumping a frontend or a backend. For a listener or a server, this corresponds to their respective ID inside the proxy. The fourth element is the numeric position of the field in the list (starting at zero). This position shall not change over time, but holes are to be expected, depending on build options or if some fields are deleted in the future. The fifth element is the field name as it appears in the CSV output. The sixth element is a positive integer and is the relative process number starting at 1.

The rest of the line starting after the first colon follows the "typed output format" described in the section above. In short, the second column (after the first ':') indicates the origin, nature and scope of the variable. The third column indicates the type of the field, among "s32", "s64", "u32", "u64" and "str". Then the fourth column is the value itself, which the consumer knows how to parse thanks to column 3 and how to process thanks to column 2.

Thus the overall line format in typed mode is :

```
<obj>.<px_id>.<id>.<fpos>.<fname>.<process_num>:<tags>:<type>:<value>
```

Here's an example of typed output format :

```
$ echo "show stat typed" | socat stdio unix-connect:/tmp/sock1
F.2.0.0.pxname.1:MGP:str:private-frontend
F.2.0.1.svname.1:MGP:str:FRONTEND
F.2.0.8.bin.1:MGP:u64:0
F.2.0.9.bout.1:MGP:u64:0
F.2.0.40.hrsp_2xx.1:MGP:u64:0
L.2.1.0.pxname.1:MGP:str:private-frontend
L.2.1.1.svname.1:MGP:str:sock-1
L.2.1.17.status.1:MGP:str:OPEN
L.2.1.73.addr.1:MGP:str:0.0.0.0:8001
S.3.13.60.rtime.1:MCP:u32:0
S.3.13.61.ttime.1:MCP:u32:0
S.3.13.62.agent_status.1:MGP:str:L4TOUT
S.3.13.64.agent_duration.1:MGP:u64:2001
S.3.13.65.check_desc.1:MCP:str:Layer4 timeout
S.3.13.66.agent_desc.1:MCP:str:Layer4 timeout
S.3.13.67.check_rise.1:MCP:u32:2
S.3.13.68.check_fall.1:MCP:u32:3
S.3.13.69.check_health.1:SGP:u32:0
S.3.13.70.agent_rise.1:MaP:u32:1
S.3.13.71.agent_fall.1:SGP:u32:1
S.3.13.72.agent_health.1:SGP:u32:1
```

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
S.3.13.73.addr.1:MCP:str:1.255.255.255:8888
S.3.13.75.mode.1:MAP:str:http
B.3.0.0.pxname.1:MGP:str:private-backend
B.3.0.1.svname.1:MGP:str:BACKEND
B.3.0.2.qcur.1:MGP:u32:0
B.3.0.3.qmax.1:MGP:u32:0
B.3.0.4.scur.1:MGP:u32:0
B.3.0.5.smax.1:MGP:u32:0
B.3.0.6.slim.1:MGP:u32:1000
B.3.0.55.lastsess.1:MMP:s32:-1
(...)
```

In the typed format, the presence of the process ID at the end of the first column makes it very easy to visually aggregate outputs from multiple processes, as show in the example below where each line appears for each process :

```
$ ( echo show stat typed | socat /var/run/haproxy.sock1 - ; \
    echo show stat typed | socat /var/run/haproxy.sock2 - ) | \
  sort -t . -k 1,1 -k 2,2n -k 3,3n -k 4,4n -k 5,5 -k 6,6n
B.3.0.0.pxname.1:MGP:str:private-backend
B.3.0.0.pxname.2:MGP:str:private-backend
B.3.0.1.svname.1:MGP:str:BACKEND
B.3.0.1.svname.2:MGP:str:BACKEND
B.3.0.2.qcur.1:MGP:u32:0
B.3.0.2.qcur.2:MGP:u32:0
B.3.0.3.qmax.1:MGP:u32:0
B.3.0.3.qmax.2:MGP:u32:0
B.3.0.4.scur.1:MGP:u32:0
B.3.0.4.scur.2:MGP:u32:0
B.3.0.5.smax.1:MGP:u32:0
B.3.0.5.smax.2:MGP:u32:0
B.3.0.6.slim.1:MGP:u32:1000
B.3.0.6.slim.2:MGP:u32:1000
(...)
```

The format of JSON output is described in a schema which may be output using "show schema json".

The JSON output contains no extra whitespace in order to reduce the volume of output. For human consumption passing the output through a pretty printer may be helpful. Example :

```
$ echo "show stat json" | socat /var/run/haproxy.sock stdio | \
  python -m json.tool
```

The JSON output contains no extra whitespace in order to reduce the volume of output. For human consumption passing the output through a pretty printer may be helpful. Example :

```
$ echo "show stat json" | socat /var/run/haproxy.sock stdio | \
  python -m json.tool
```

show startup-logs

Dump all messages emitted during the startup of the current haproxy process, each startup-logs buffer is unique to its haproxy worker.

show table

Dump general information on all known stick-tables. Their name is returned (the name of the proxy which holds them), their type (currently zero, always IP), their size in maximum possible number of entries, and the number of entries currently in use.

Example :

```
    $ echo "show table" | socat stdio /tmp/sock1
>>> # table: front_pub, type: ip, size:204800, used:171454
>>> # table: back_rdp, type: ip, size:204800, used:0
```

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

show table <name> [data.<type> <operator> <value>] | [key <key>]

Dump contents of stick-table <name>. In this mode, a first line of generic information about the table is reported as with "show table", then all entries are dumped. Since this can be quite heavy, it is possible to specify a filter in order to specify what entries to display.

When the "data." form is used the filter applies to the stored data (see "stick-table" in section 4.2). A stored data type must be specified in <type>, and this data type must be stored in the table otherwise an error is reported. The data is compared according to <operator> with the 64-bit integer <value>. Operators are the same as with the ACLs :

```
  - eq : match entries whose data is equal to this value
  - ne : match entries whose data is not equal to this value
  - le : match entries whose data is less than or equal to this value
  - ge : match entries whose data is greater than or equal to this value
  - lt : match entries whose data is less than this value
  - gt : match entries whose data is greater than this value
```

When the key form is used the entry <key> is shown. The key must be of the same type as the table, which currently is limited to IPv4, IPv6, integer, and string.

Example :

```
    $ echo "show table http_proxy" | socat stdio /tmp/sock1
>>> # table: http_proxy, type: ip, size:204800, used:2
>>> 0x80e6a4c: key=127.0.0.1 use=0 exp=3594729 gpc0=0 conn_rate(30000)=1  \
      bytes_out_rate(60000)=187
>>> 0x80e6a80: key=127.0.0.2 use=0 exp=3594740 gpc0=1 conn_rate(30000)=10 \
      bytes_out_rate(60000)=191

    $ echo "show table http_proxy data.gpc0 gt 0" | socat stdio /tmp/sock1
>>> # table: http_proxy, type: ip, size:204800, used:2
>>> 0x80e6a80: key=127.0.0.2 use=0 exp=3594740 gpc0=1 conn_rate(30000)=10 \
      bytes_out_rate(60000)=191

    $ echo "show table http_proxy data.conn_rate gt 5" | \
        socat stdio /tmp/sock1
>>> # table: http_proxy, type: ip, size:204800, used:2
>>> 0x80e6a80: key=127.0.0.2 use=0 exp=3594740 gpc0=1 conn_rate(30000)=10 \
      bytes_out_rate(60000)=191

    $ echo "show table http_proxy key 127.0.0.2" | \
        socat stdio /tmp/sock1
>>> # table: http_proxy, type: ip, size:204800, used:2
>>> 0x80e6a80: key=127.0.0.2 use=0 exp=3594740 gpc0=1 conn_rate(30000)=10 \
      bytes_out_rate(60000)=191
```

When the data criterion applies to a dynamic value dependent on time such as a bytes rate, the value is dynamically computed during the evaluation of the entry in order to decide whether it has to be dumped or not. This means that such a filter could match for some time then not match anymore because as time goes, the average event rate drops.

It is possible to use this to extract lists of IP addresses abusing the service, in order to monitor them or even blacklist them in a firewall.

Example :

```
$ echo "show table http_proxy data.gpc0 gt 0" \
  | socat stdio /tmp/sock1 \
  | fgrep 'key=' | cut -d' ' -f2 | cut -d= -f2 > abusers-ip.txt
  ( or | awk '/key/{ print a[split($2,a,"=")]; }' )
```

show threads

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Dumps some internal states and structures for each thread, that may be useful to help developers understand a problem. The output tries to be readable by showing one block per thread. When haproxy is built with USE_THREAD_DUMP=1, an advanced dump mechanism involving thread signals is used so that each thread can dump its own state in turn. Without this option, the thread processing the command shows all its details but the other ones are less detailed. A star ('*') is displayed in front of the thread handling the command. A right angle bracket ('>') may also be displayed in front of threads which didn't make any progress since last invocation of this command, indicating a bug in the code which must absolutely be reported. When this happens between two threads it usually indicates a deadlock. If a thread is alone, it's a different bug like a corrupted list. In all cases the process needs is not fully functional anymore and needs to be restarted.

The output format is purposely not documented so that it can easily evolve as new needs are identified, without having to maintain any form of backwards compatibility, and just like with "show activity", the values are meaningless without the code at hand.

show tls-keys [id|*]

Dump all loaded TLS ticket keys references. The TLS ticket key reference ID and the file from which the keys have been loaded is shown. Both of those can be used to update the TLS keys using "set ssl tls-key". If an ID is specified as parameter, it will dump the tickets, using * it will dump every keys from every references.

show schema json

Dump the schema used for the output of "show info json" and "show stat json".

The contains no extra whitespace in order to reduce the volume of output. For human consumption passing the output through a pretty printer may be helpful. Example :

```
$ echo "show schema json" | socat /var/run/haproxy.sock stdio | \
  python -m json.tool
```

The schema follows "JSON Schema" (json-schema.org) and accordingly verifiers may be used to verify the output of "show info json" and "show stat json" against the schema.

show version

Show the version of the current HAProxy process. This is available from master and workers CLI.

Example:

```
$ echo "show version" | socat /var/run/haproxy.sock stdio
2.4.9

$ echo "show version" | socat /var/run/haproxy-master.sock stdio
2.5.0
```

shutdown frontend <frontend>

Completely delete the specified frontend. All the ports it was bound to will be released. It will not be possible to enable the frontend anymore after this operation. This is intended to be used in environments where stopping a proxy is not even imaginable but a misconfigured proxy must be fixed. That way it's possible to release the port and bind it into another process to restore operations. The frontend will not appear at all on the stats page once it is terminated.

The frontend may be specified either by its name or by its numeric ID, prefixed with a sharp ('#').

This command is restricted and can only be issued on sockets configured for level "admin".

shutdown session <id>

Immediately terminate the session matching the specified session identifier. This identifier is the first field at the beginning of the lines in the dumps of "show sess" (it corresponds to the session pointer). This can be used to terminate a long-running session without waiting for a timeout or when an endless transfer is ongoing. Such terminated sessions are reported with a 'K' flag in the logs.

shutdown sessions server <backend>/<server>

Immediately terminate all the sessions attached to the specified server. This can be used to terminate long-running sessions after a server is put into maintenance mode, for instance. Such terminated sessions are reported with a 'K' flag in the logs.

9.4. Master CLI

The master CLI is a socket bound to the master process in master-worker mode. This CLI gives access to the unix socket commands in every running or leaving processes and allows a basic supervision of those processes.

The master CLI is configurable only from the haproxy program arguments with the -S option. This option also takes bind options separated by commas.

Example:

```
# haproxy -W -S 127.0.0.1:1234 -f test1.cfg
# haproxy -Ws -S /tmp/master-socket,uid,1000,gid,1000,mode,600 -f test1.cfg
# haproxy -W -S /tmp/master-socket,level,user -f test1.cfg
```

The master CLI introduces a new 'show proc' command to surpervise the processes:

Example:

```
$ echo 'show proc' | socat /var/run/haproxy-master.sock -
#<PID>          <type>          <relative PID>  <reloads>       <uptime>
1162            master          0               5               0d00h02m07s
c9-7
# workers
1271            worker          1               0               0d00h00m00s
c9-7
1272            worker          2               0               0d00h00m00s
c9-7
# old workers
1233            worker          [was: 1]        3               0d00h00m43s
f6-289
```

In this example, the master has been reloaded 5 times but one of the old worker is still running and survived 3 reloads. You could access the CLI of this worker to understand what's going on.

When the prompt is enabled (via the "prompt" command), the context the CLI is working on is displayed in the prompt. The master is identified by the "master" string, and other processes are identified with their PID. In case the last reload failed, the master prompt will be changed to "master[ReloadFailed]>" so that it becomes visible that the process is still running on the previous configuration and that the new configuration is not operational.

The master CLI uses a special prefix notation to access the multiple processes. This notation is easily identifiable as it begins by a @.

A @ prefix can be followed by a relative process number or by an exclamation point and a PID. (e.g. @1 or @!1271). A @ alone could be use to specify the master. Leaving processes are only accessible with the PID as relative process number are only usable with the current processes.

Examples:

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
$ socat /var/run/haproxy-master.sock readline
prompt
master> @1 show info; @2 show info
[...]
Process_num: 1
Pid: 1271
[...]
Process_num: 2
Pid: 1272
[...]
master>

$ echo '@!1271 show info; @!1272 show info' | socat /var/run/haproxy-master.soc
[...]
```

A prefix could be use as a command, which will send every next commands to the specified process.

Examples:

```
$ socat /var/run/haproxy-master.sock readline
prompt
master> @1
1271> show info
[...]
1271> show stat
[...]
1271> @
master>

$ echo '@1; show info; show stat; @2; show info; show stat' | socat /var/run/ha
ock -
[...]
```

You can also reload the HAProxy master process with the "reload" command which does the same as a `kill -USR2` on the master process, provided that the user has at least "operator" or "admin" privileges.

Example:

```
$ echo "reload" | socat /var/run/haproxy-master.sock
```

Note that a reload will close the connection to the master CLI.

10. Tricks for easier configuration management

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

It is very common that two HAProxy nodes constituting a cluster share exactly the same configuration modulo a few addresses. Instead of having to maintain a duplicate configuration for each node, which will inevitably diverge, it is possible to include environment variables in the configuration. Thus multiple configuration may share the exact same file with only a few different system wide environment variables. This started in version 1.5 where only addresses were allowed to include environment variables, and 1.6 goes further by supporting environment variables everywhere. The syntax is the same as in the UNIX shell, a variable starts with a dollar sign ('$'), followed by an opening curly brace ('{'), then the variable name followed by the closing brace ('}'). Except for addresses, environment variables are only interpreted in arguments surrounded with double quotes (this was necessary not to break existing setups using regular expressions involving the dollar symbol).

Environment variables also make it convenient to write configurations which are expected to work on various sites where only the address changes. It can also permit to remove passwords from some configs. Example below where the the file "site1.env" file is sourced by the init script upon startup :

```
$ cat site1.env
LISTEN=192.168.1.1
CACHE_PFX=192.168.11
SERVER_PFX=192.168.22
LOGGER=192.168.33.1
STATSLP=admin:pa$$w0rd
ABUSERS=/etc/haproxy/abuse.lst
TIMEOUT=10s

$ cat haproxy.cfg
global
    log "${LOGGER}:514" local0

defaults
    mode http
    timeout client "${TIMEOUT}"
    timeout server "${TIMEOUT}"
    timeout connect 5s

frontend public
    bind "${LISTEN}:80"
    http-request reject if { src -f "${ABUSERS}" }
    stats uri /stats
    stats auth "${STATSLP}"
    use_backend cache if { path_end .jpg .css .ico }
    default_backend server

backend cache
    server cache1 "${CACHE_PFX}.1:18080" check
    server cache2 "${CACHE_PFX}.2:18080" check

backend server
    server cache1 "${SERVER_PFX}.1:8080" check
    server cache2 "${SERVER_PFX}.2:8080" check
```

11. Well-known traps to avoid

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Once in a while, someone reports that after a system reboot, the haproxy service wasn't started, and that once they start it by hand it works. Most often, these people are running a clustered IP address mechanism such as keepalived, to assign the service IP address to the master node only, and while it used to work when they used to bind haproxy to address 0.0.0.0, it stopped working after they bound it to the virtual IP address. What happens here is that when the service starts, the virtual IP address is not yet owned by the local node, so when HAProxy wants to bind to it, the system rejects this because it is not a local IP address. The fix doesn't consist in delaying the haproxy service startup (since it wouldn't stand a restart), but instead to properly configure the system to allow binding to non-local addresses. This is easily done on Linux by setting the net.ipv4.ip_nonlocal_bind sysctl to 1. This is also needed in order to transparently intercept the IP traffic that passes through HAProxy for a specific target address.

Multi-process configurations involving source port ranges may apparently seem to work but they will cause some random failures under high loads because more than one process may try to use the same source port to connect to the same server, which is not possible. The system will report an error and a retry will happen, picking another port. A high value in the "retries" parameter may hide the effect to a certain extent but this also comes with increased CPU usage and processing time. Logs will also report a certain number of retries. For this reason, port ranges should be avoided in multi-process configurations.

Since HAProxy uses SO_REUSEPORT and supports having multiple independent processes bound to the same IP:port, during troubleshooting it can happen that an old process was not stopped before a new one was started. This provides absurd test results which tend to indicate that any change to the configuration is ignored. The reason is that in fact even the new process is restarted with a new configuration, the old one also gets some incoming connections and processes them, returning unexpected results. When in doubt, just stop the new process and try again. If it still works, it very likely means that an old process remains alive and has to be stopped. Linux's "netstat -lntp" is of good help here.

When adding entries to an ACL from the command line (eg: when blacklisting a source address), it is important to keep in mind that these entries are not synchronized to the file and that if someone reloads the configuration, these updates will be lost. While this is often the desired effect (for blacklisting) it may not necessarily match expectations when the change was made as a fix for a problem. See the "add acl" action of the CLI interface.

12. Debugging and performance issue

When HAProxy is started with the "-d" option, it will stay in the foreground and will print one line per event, such as an incoming connection, the end of a connection, and for each request or response header line seen. This debug output is emitted before the contents are processed, so they don't consider the local modifications. The main use is to show the request and response without having to run a network sniffer. The output is less readable when multiple connections are handled in parallel, though the "debug2ansi" and "debug2html" scripts found in the examples/ directory definitely help here by coloring the output.

If a request or response is rejected because HAProxy finds it is malformed, the best thing to do is to connect to the CLI and issue "show errors", which will report the last captured faulty request and response for each frontend and backend, with all the necessary information to indicate precisely the first character of the input stream that was rejected. This is sometimes needed to prove to customers or to developers that a bug is present in their code. In this case it is often possible to relax the checks (but still keep the captures) using "option accept-invalid-http-request" or its equivalent for responses coming from the server "option accept-invalid-http-response". Please see the configuration manual for more details.

Example :

```
> show errors
Total events captured on [13/Oct/2015:13:43:47.169] : 1

[13/Oct/2015:13:43:40.918] frontend HAProxyLocalStats (#2): invalid request
  backend <NONE> (#-1), server <NONE> (#-1), event #0
  src 127.0.0.1:51981, session #0, session flags 0x00000080
  HTTP msg state 26, msg flags 0x00000000, tx flags 0x00000000
  HTTP chunk len 0 bytes, HTTP body len 0 bytes
  buffer flags 0x00808002, out 0 bytes, total 31 bytes
  pending 31 bytes, wrapping at 8040, error at position 13:

  00000  GET /invalid request HTTP/1.1\r\n
```


Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

The output of "show info" on the CLI provides a number of useful information regarding the maximum connection rate ever reached, maximum SSL key rate ever reached, and in general all information which can help to explain temporary issues regarding CPU or memory usage. Example :

```
> show info
Name: HAProxy
Version: 1.6-dev7-e32d18-17
Release_date: 2015/10/12
Nbproc: 1
Process_num: 1
Pid: 7949
Uptime: 0d 0h02m39s
Uptime_sec: 159
Memmax_MB: 0
Ulimit-n: 120032
Maxsock: 120032
Maxconn: 60000
Hard_maxconn: 60000
CurrConns: 0
CumConns: 3
CumReq: 3
MaxSslConns: 0
CurrSslConns: 0
CumSslConns: 0
Maxpipes: 0
PipesUsed: 0
PipesFree: 0
ConnRate: 0
ConnRateLimit: 0
MaxConnRate: 1
SessRate: 0
SessRateLimit: 0
MaxSessRate: 1
SslRate: 0
SslRateLimit: 0
MaxSslRate: 0
SslFrontendKeyRate: 0
SslFrontendMaxKeyRate: 0
SslFrontendSessionReuse_pct: 0
SslBackendKeyRate: 0
SslBackendMaxKeyRate: 0
SslCacheLookups: 0
SslCacheMisses: 0
CompressBpsIn: 0
CompressBpsOut: 0
CompressBpsRateLim: 0
ZlibMemUsage: 0
MaxZlibMemUsage: 0
Tasks: 5
Run_queue: 1
Idle_pct: 100
node: wtap
description:
```

When an issue seems to randomly appear on a new version of HAProxy (eg: every second request is aborted, occasional crash, etc), it is worth trying to enable memory poisoning so that each call to malloc() is immediately followed by the filling of the memory area with a configurable byte. By default this byte is 0x50 (ASCII for 'P'), but any other byte can be used, including zero (which will have the same effect as a calloc() and which may make issues disappear). Memory poisoning is enabled on the command line using the "-dM" option. It slightly hurts performance and is not recommended for use in production. If an issue happens all the time with it or never happens when poisoning uses byte zero, it clearly means you've found a bug and you definitely need to report it. Otherwise if there's no clear change, the problem it is not related.

When debugging some latency issues, it is important to use both strace and tcpdump on the local machine, and another tcpdump on the remote system. The

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

reason for this is that there are delays everywhere in the processing chain and it is important to know which one is causing latency to know where to act. In practice, the local tcpdump will indicate when the input data come in. Strace will indicate when haproxy receives these data (using recv/recvfrom). Warning, openssl uses read()/write() syscalls instead of recv()/send(). Strace will also show when haproxy sends the data, and tcpdump will show when the system sends these data to the interface. Then the external tcpdump will show when the data sent are really received (since the local one only shows when the packets are queued). The benefit of sniffing on the local system is that strace and tcpdump will use the same reference clock. Strace should be used with "-tts200" to get complete timestamps and report large enough chunks of data to read them. Tcpdump should be used with "-nvvttSs0" to report full packets, real sequence numbers and complete timestamps.

In practice, received data are almost always immediately received by haproxy (unless the machine has a saturated CPU or these data are invalid and not delivered). If these data are received but not sent, it generally is because the output buffer is saturated (ie: recipient doesn't consume the data fast enough). This can be confirmed by seeing that the polling doesn't notify of the ability to write on the output file descriptor for some time (it's often easier to spot in the strace output when the data finally leave and then roll back to see when the write event was notified). It generally matches an ACK received from the recipient, and detected by tcpdump. Once the data are sent, they may spend some time in the system doing nothing. Here again, the TCP congestion window may be limited and not allow these data to leave, waiting for an ACK to open the window. If the traffic is idle and the data take 40 ms or 200 ms to leave, it's a different issue (which is not an issue), it's the fact that the Nagle algorithm prevents empty packets from leaving immediately, in hope that they will be merged with subsequent data. HAProxy automatically disables Nagle in pure TCP mode and in tunnels. However it definitely remains enabled when forwarding an HTTP body (and this contributes to the performance improvement there by reducing the number of packets). Some HTTP non-compliant applications may be sensitive to the latency when delivering incomplete HTTP response messages. In this case you will have to enable "option http-no-delay" to disable Nagle in order to work around their design, keeping in mind that any other proxy in the chain may similarly be impacted. If tcpdump reports that data leave immediately but the other end doesn't see them quickly, it can mean there is a congested WAN link, a congested LAN with flow control enabled and preventing the data from leaving, or more commonly that HAProxy is in fact running in a virtual machine and that for whatever reason the hypervisor has decided that the data didn't need to be sent immediately. In virtualized environments, latency issues are almost always caused by the virtualization layer, so in order to save time, it's worth first comparing tcpdump in the VM and on the external components. Any difference has to be credited to the hypervisor and its accompanying drivers.

When some TCP SACK segments are seen in tcpdump traces (using -vv), it always means that the side sending them has got the proof of a lost packet. While not seeing them doesn't mean there are no losses, seeing them definitely means the network is lossy. Losses are normal on a network, but at a rate where SACKs are not noticeable at the naked eye. If they appear a lot in the traces, it is worth investigating exactly what happens and where the packets are lost. HTTP doesn't cope well with TCP losses, which introduce huge latencies.

The "netstat -i" command will report statistics per interface. An interface where the Rx-Ovr counter grows indicates that the system doesn't have enough resources to receive all incoming packets and that they're lost before being processed by the network driver. Rx-Drp indicates that some received packets were lost in the network stack because the application doesn't process them fast enough. This can happen during some attacks as well. Tx-Drp means that the output queues were full and packets had to be dropped. When using TCP it should be very rare, but will possibly indicate a saturated outgoing link.

13. Security considerations

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
HAProxy is designed to run with very limited privileges. The standard way to
use it is to isolate it into a chroot jail and to drop its privileges to a
non-root user without any permissions inside this jail so that if any future
vulnerability were to be discovered, its compromise would not affect the rest
of the system.

In order to perform a chroot, it first needs to be started as a root user. It is
pointless to build hand-made chroots to start the process there, these ones are
painful to build, are never properly maintained and always contain way more
bugs than the main file-system. And in case of compromise, the intruder can use
the purposely built file-system. Unfortunately many administrators confuse
"start as root" and "run as root", resulting in the uid change to be done prior
to starting haproxy, and reducing the effective security restrictions.

HAProxy will need to be started as root in order to :
  - adjust the file descriptor limits
  - bind to privileged port numbers
  - bind to a specific network interface
  - transparently listen to a foreign address
  - isolate itself inside the chroot jail
  - drop to another non-privileged UID

HAProxy may require to be run as root in order to :
  - bind to an interface for outgoing connections
  - bind to privileged source ports for outgoing connections
  - transparently bind to a foreign address for outgoing connections

Most users will never need the "run as root" case. But the "start as root"
covers most usages.

A safe configuration will have :

  - a chroot statement pointing to an empty location without any access
    permissions. This can be prepared this way on the UNIX command line :

      # mkdir /var/empty && chmod 0 /var/empty || echo "Failed"

    and referenced like this in the HAProxy configuration's global section :

      chroot /var/empty

  - both a uid/user and gid/group statements in the global section :

      user haproxy
      group haproxy

  - a stats socket whose mode, uid and gid are set to match the user and/or
    group allowed to access the CLI so that nobody may access it :

      stats socket /var/run/haproxy.stat uid hatop gid hatop mode 600
```

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

HAPROXY COMMUNITY EDITION (http://www.haprox

Starter Guide

version 2.0.35

```
This document doesn't provide any configuration help or hints, but it explains
where to find the relevant documents. The summary below is meant to help you
search sections by name and navigate through the document.

Note to documentation contributors :
    This document is formatted with 80 columns per line, with even number of
    spaces for indentation and without tabs. Please follow these rules strictly
    so that it remains easily printable everywhere. If you add sections, please
    update the summary below for easier searching.
```

Summary

1. **Available documentation**

2. **Quick introduction to load balancing and load balancers**

3. **Introduction to HAProxy**
3.1. What HAProxy is and is not
3.2. How HAProxy works
3.3. Basic features
3.3.1. Proxying
3.3.2. SSL
3.3.3. Monitoring
3.3.4. High availability
3.3.5. Load balancing
3.3.6. Stickiness
3.3.7. Sampling and converting information
3.3.8. Maps
3.3.9. ACLs and conditions
3.3.10. Content switching
3.3.11. Stick-tables
3.3.12. Formatted strings
3.3.13. HTTP rewriting and redirection
3.3.14. Server protection
3.3.15. Logging
3.3.16. Statistics
3.4. Advanced features
3.4.1. Management
3.4.2. System-specific capabilities
3.4.3. Scripting
3.5. Sizing
3.6. How to get HAProxy

4. **Companion products and alternatives**
4.1. Apache HTTP server
4.2. NGINX
4.3. Varnish
4.4. Alternatives

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

1. Available documentation

```
The complete HAProxy documentation is contained in the following documents.
Please ensure to consult the relevant documentation to save time and to get the
most accurate response to your needs. Also please refrain from sending questions
to the mailing list whose responses are present in these documents.

  - intro.txt (this document) : it presents the basics of load balancing,
    HAProxy as a product, what it does, what it doesn't do, some known traps to
    avoid, some OS-specific limitations, how to get it, how it evolves, how to
    ensure you're running with all known fixes, how to update it, complements
    and alternatives.

  - management.txt : it explains how to start haproxy, how to manage it at
    runtime, how to manage it on multiple nodes, and how to proceed with
    seamless upgrades.

  - configuration.txt : the reference manual details all configuration keywords
    and their options. It is used when a configuration change is needed.

  - coding-style.txt : this is for developers who want to propose some code to
    the project. It explains the style to adopt for the code. It is not very
    strict and not all the code base completely respects it, but contributions
    which diverge too much from it will be rejected.

  - proxy-protocol.txt : this is the de-facto specification of the PROXY
    protocol which is implemented by HAProxy and a number of third party
    products.

  - README : how to build HAProxy from sources
```

2. Quick introduction to load balancin load balancers

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

Load balancing consists in aggregating multiple components in order to achieve a total processing capacity above each component's individual capacity, without any intervention from the end user and in a scalable way. This results in more operations being performed simultaneously by the time it takes a component to perform only one. A single operation however will still be performed on a single component at a time and will not get faster than without load balancing. It always requires at least as many operations as available components and an efficient load balancing mechanism to make use of all components and to fully benefit from the load balancing. A good example of this is the number of lanes on a highway which allows as many cars to pass during the same time frame without increasing their individual speed.

Examples of load balancing :

  - Process scheduling in multi-processor systems
  - Link load balancing (e.g. EtherChannel, Bonding)
  - IP address load balancing (e.g. ECMP, DNS round-robin)
  - Server load balancing (via load balancers)

The mechanism or component which performs the load balancing operation is called a load balancer. In web environments these components are called a "network load balancer", and more commonly a "load balancer" given that this activity is by far the best known case of load balancing.

A load balancer may act :

  - at the link level : this is called link load balancing, and it consists in choosing what network link to send a packet to;

  - at the network level : this is called network load balancing, and it consists in choosing what route a series of packets will follow;

  - at the server level : this is called server load balancing and it consists in deciding what server will process a connection or request.

Two distinct technologies exist and address different needs, though with some overlapping. In each case it is important to keep in mind that load balancing consists in diverting the traffic from its natural flow and that doing so always requires a minimum of care to maintain the required level of consistency between all routing decisions.

The first one acts at the packet level and processes packets more or less individually. There is a 1-to-1 relation between input and output packets, so it is possible to follow the traffic on both sides of the load balancer using a regular network sniffer. This technology can be very cheap and extremely fast. It is usually implemented in hardware (ASICs) allowing to reach line rate, such as switches doing ECMP. Usually stateless, it can also be stateful (consider the session a packet belongs to and called layer4-LB or L4), may support DSR (direct server return, without passing through the LB again) if the packets were not modified, but provides almost no content awareness. This technology is very well suited to network-level load balancing, though it is sometimes used for very basic server load balancing at high speed.

The second one acts on session contents. It requires that the input streams is reassembled and processed as a whole. The contents may be modified, and the output stream is segmented into new packets. For this reason it is generally performed by proxies and they're often called layer 7 load balancers or L7. This implies that there are two distinct connections on each side, and that there is no relation between input and output packets sizes nor counts. Clients and servers are not required to use the same protocol (for example IPv4 vs IPv6, clear vs SSL). The operations are always stateful, and the return traffic must pass through the load balancer. The extra processing comes with a cost so it's not always possible to achieve line rate, especially with small packets. On the other hand, it offers wide possibilities and is generally achieved by pure software, even if embedded into hardware appliances. This technology is very well suited for server load balancing.

Packet-based load balancers are generally deployed in cut-through mode, so they are installed on the normal path of the traffic and divert it according to the

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

configuration. The return traffic doesn't necessarily pass through the load balancer. Some modifications may be applied to the network destination address in order to direct the traffic to the proper destination. In this case, it is mandatory that the return traffic passes through the load balancer. If the routes doesn't make this possible, the load balancer may also replace the packets' source address with its own in order to force the return traffic to pass through it.

Proxy-based load balancers are deployed as a server with their own IP addresses and ports, without architecture changes. Sometimes this requires to perform some adaptations to the applications so that clients are properly directed to the load balancer's IP address and not directly to the server's. Some load balancers may have to adjust some servers' responses to make this possible (e.g. the HTTP Location header field used in HTTP redirects). Some proxy-based load balancers may intercept traffic for an address they don't own, and spoof the client's address when connecting to the server. This allows them to be deployed as if they were a regular router or firewall, in a cut-through mode very similar to the packet based load balancers. This is particularly appreciated for products which combine both packet mode and proxy mode. In this case DSR is obviously still not possible and the return traffic still has to be routed back to the load balancer.

A very scalable layered approach would consist in having a front router which receives traffic from multiple load balanced links, and uses ECMP to distribute this traffic to a first layer of multiple stateful packet-based load balancers (L4). These L4 load balancers in turn pass the traffic to an even larger number of proxy-based load balancers (L7), which have to parse the contents to decide what server will ultimately receive the traffic.

The number of components and possible paths for the traffic increases the risk of failure; in very large environments, it is even normal to permanently have a few faulty components being fixed or replaced. Load balancing done without awareness of the whole stack's health significantly degrades availability. For this reason, any sane load balancer will verify that the components it intends to deliver the traffic to are still alive and reachable, and it will stop delivering traffic to faulty ones. This can be achieved using various methods.

The most common one consists in periodically sending probes to ensure the component is still operational. These probes are called "health checks". They must be representative of the type of failure to address. For example a ping-based check will not detect that a web server has crashed and doesn't listen to a port anymore, while a connection to the port will verify this, and a more advanced request may even validate that the server still works and that the database it relies on is still accessible. Health checks often involve a few retries to cover for occasional measuring errors. The period between checks must be small enough to ensure the faulty component is not used for too long after an error occurs.

Other methods consist in sampling the production traffic sent to a destination to observe if it is processed correctly or not, and to evict the components which return inappropriate responses. However this requires to sacrifice a part of the production traffic and this is not always acceptable. A combination of these two mechanisms provides the best of both worlds, with both of them being used to detect a fault, and only health checks to detect the end of the fault. A last method involves centralized reporting : a central monitoring agent periodically updates all load balancers about all components' state. This gives a global view of the infrastructure to all components, though sometimes with less accuracy or responsiveness. It's best suited for environments with many load balancers and many servers.

Layer 7 load balancers also face another challenge known as stickiness or persistence. The principle is that they generally have to direct multiple subsequent requests or connections from a same origin (such as an end user) to the same target. The best known example is the shopping cart on an online store. If each click leads to a new connection, the user must always be sent to the server which holds his shopping cart. Content-awareness makes it easier to spot some elements in the request to identify the server to deliver it to, but that's not always enough. For example if the source address is used as a key to pick a server, it can be decided that a hash-based algorithm will be

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

used and that a given IP address will always be sent to the same server based on a divide of the address by the number of available servers. But if one server fails, the result changes and all users are suddenly sent to a different server and lose their shopping cart. The solution against this issue consists in memorizing the chosen target so that each time the same visitor is seen, he's directed to the same server regardless of the number of available servers. The information may be stored in the load balancer's memory, in which case it may have to be replicated to other load balancers if it's not alone, or it may be stored in the client's memory using various methods provided that the client is able to present this information back with every request (cookie insertion, redirection to a sub-domain, etc). This mechanism provides the extra benefit of not having to rely on unstable or unevenly distributed information (such as the source IP address). This is in fact the strongest reason to adopt a layer 7 load balancer instead of a layer 4 one.

In order to extract information such as a cookie, a host header field, a URL or whatever, a load balancer may need to decrypt SSL/TLS traffic and even possibly to re-encrypt it when passing it to the server. This expensive task explains why in some high-traffic infrastructures, sometimes there may be a lot of load balancers.

Since a layer 7 load balancer may perform a number of complex operations on the traffic (decrypt, parse, modify, match cookies, decide what server to send to, etc), it can definitely cause some trouble and will very commonly be accused of being responsible for a lot of trouble that it only revealed. Often it will be discovered that servers are unstable and periodically go up and down, or for web servers, that they deliver pages with some hard-coded links forcing the clients to connect directly to one specific server without passing via the load balancer, or that they take ages to respond under high load causing timeouts. That's why logging is an extremely important aspect of layer 7 load balancing. Once a trouble is reported, it is important to figure if the load balancer took a wrong decision and if so why so that it doesn't happen anymore.

3. Introduction to HAProxy

HAProxy is written as "HAProxy" to designate the product, and as "haproxy" to designate the executable program, software package or a process. However, both are commonly used for both purposes, and are pronounced H-A-Proxy. Very early, "haproxy" used to stand for "high availability proxy" and the name was written in two separate words, though by now it means nothing else than "HAProxy".

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.1. What HAProxy is and isn't

```
HAProxy is :

  - a TCP proxy : it can accept a TCP connection from a listening socket,
    connect to a server and attach these sockets together allowing traffic to
    flow in both directions;

  - an HTTP reverse-proxy (called a "gateway" in HTTP terminology) : it presents
    itself as a server, receives HTTP requests over connections accepted on a
    listening TCP socket, and passes the requests from these connections to
    servers using different connections.

  - an SSL terminator / initiator / offloader : SSL/TLS may be used on the
    connection coming from the client, on the connection going to the server,
    or even on both connections.

  - a TCP normalizer : since connections are locally terminated by the operating
    system, there is no relation between both sides, so abnormal traffic such as
    invalid packets, flag combinations, window advertisements, sequence numbers,
    incomplete connections (SYN floods), or so will not be passed to the other
    side. This protects fragile TCP stacks from protocol attacks, and also
    allows to optimize the connection parameters with the client without having
    to modify the servers' TCP stack settings.

  - an HTTP normalizer : when configured to process HTTP traffic, only valid
    complete requests are passed. This protects against a lot of protocol-based
    attacks. Additionally, protocol deviations for which there is a tolerance
    in the specification are fixed so that they don't cause problem on the
    servers (e.g. multiple-line headers).

  - an HTTP fixing tool : it can modify / fix / add / remove / rewrite the URL
    or any request or response header. This helps fixing interoperability issues
    in complex environments.

  - a content-based switch : it can consider any element from the request to
    decide what server to pass the request or connection to. Thus it is possible
    to handle multiple protocols over a same port (e.g. HTTP, HTTPS, SSH).

  - a server load balancer : it can load balance TCP connections and HTTP
    requests. In TCP mode, load balancing decisions are taken for the whole
    connection. In HTTP mode, decisions are taken per request.

  - a traffic regulator : it can apply some rate limiting at various points,
    protect the servers against overloading, adjust traffic priorities based on
    the contents, and even pass such information to lower layers and outer
    network components by marking packets.

  - a protection against DDoS and service abuse : it can maintain a wide number
    of statistics per IP address, URL, cookie, etc and detect when an abuse is
    happening, then take action (slow down the offenders, block them, send them
    to outdated contents, etc).

  - an observation point for network troubleshooting : due to the precision of
    the information reported in logs, it is often used to narrow down some
    network-related issues.

  - an HTTP compression offloader : it can compress responses which were not
    compressed by the server, thus reducing the page load time for clients with
    poor connectivity or using high-latency, mobile networks.

HAProxy is not :

  - an explicit HTTP proxy, i.e. the proxy that browsers use to reach the
    internet. There are excellent open-source software dedicated for this task,
    such as Squid. However HAProxy can be installed in front of such a proxy to
    provide load balancing and high availability.

  - a caching proxy : it will return the contents received from the server as-is
```

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

and will not interfere with any caching policy. There are excellent open-source software for this task such as Varnish. HAProxy can be installed in front of such a cache to provide SSL offloading, and scalability through smart load balancing.

- a data scrubber : it will not modify the body of requests nor responses.

- a web server : during startup, it isolates itself inside a chroot jail and drops its privileges, so that it will not perform any single file-system access once started. As such it cannot be turned into a web server. There are excellent open-source software for this such as Apache or Nginx, and HAProxy can be installed in front of them to provide load balancing and high availability.

- a packet-based load balancer : it will not see IP packets nor UDP datagrams, will not perform NAT or even less DSR. These are tasks for lower layers. Some kernel-based components such as IPVS (Linux Virtual Server) already do this pretty well and complement perfectly with HAProxy.

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.2. How HAProxy works

HAProxy is a single-threaded, event-driven, non-blocking engine combining a very fast I/O layer with a priority-based scheduler. As it is designed with a data forwarding goal in mind, its architecture is optimized to move data as fast as possible with the least possible operations. As such it implements a layered model offering bypass mechanisms at each level ensuring data doesn't reach higher levels unless needed. Most of the processing is performed in the kernel, and HAProxy does its best to help the kernel do the work as fast as possible by giving some hints or by avoiding certain operation when it guesses they could be grouped later. As a result, typical figures show 15% of the processing time spent in HAProxy versus 85% in the kernel in TCP or HTTP close mode, and about 30% for HAProxy versus 70% for the kernel in HTTP keep-alive mode.

A single process can run many proxy instances; configurations as large as 300000 distinct proxies in a single process were reported to run fine. Thus there is usually no need to start more than one process for all instances.

It is possible to make HAProxy run over multiple processes, but it comes with a few limitations. In general it doesn't make sense in HTTP close or TCP modes because the kernel-side doesn't scale very well with some operations such as connect(). It scales pretty well for HTTP keep-alive mode but the performance that can be achieved out of a single process generally outperforms common needs by an order of magnitude. It does however make sense when used as an SSL offloader, and this feature is well supported in multi-process mode.

HAProxy only requires the haproxy executable and a configuration file to run. For logging it is highly recommended to have a properly configured syslog daemon and log rotations in place. The configuration files are parsed before starting, then HAProxy tries to bind all listening sockets, and refuses to start if anything fails. Past this point it cannot fail anymore. This means that there are no runtime failures and that if it accepts to start, it will work until it is stopped.

Once HAProxy is started, it does exactly 3 things :

  - process incoming connections;

  - periodically check the servers' status (known as health checks);

  - exchange information with other haproxy nodes.

Processing incoming connections is by far the most complex task as it depends on a lot of configuration possibilities, but it can be summarized as the 9 steps below :

  - accept incoming connections from listening sockets that belong to a
    configuration entity known as a "frontend", which references one or multiple
    listening addresses;

  - apply the frontend-specific processing rules to these connections that may
    result in blocking them, modifying some headers, or intercepting them to
    execute some internal applets such as the statistics page or the CLI;

  - pass these incoming connections to another configuration entity representing
    a server farm known as a "backend", which contains the list of servers and
    the load balancing strategy for this server farm;

  - apply the backend-specific processing rules to these connections;

  - decide which server to forward the connection to according to the load
    balancing strategy;

  - apply the backend-specific processing rules to the response data;

  - apply the frontend-specific processing rules to the response data;

  - emit a log to report what happened in fine details;

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

- in HTTP, loop back to the second step to wait for a new request, otherwise close the connection.

Frontends and backends are sometimes considered as half-proxies, since they only look at one side of an end-to-end connection; the frontend only cares about the clients while the backend only cares about the servers. HAProxy also supports full proxies which are exactly the union of a frontend and a backend. When HTTP processing is desired, the configuration will generally be split into frontends and backends as they open a lot of possibilities since any frontend may pass a connection to any backend. With TCP-only proxies, using frontends and backends rarely provides a benefit and the configuration can be more readable with full proxies.

3.3. Basic features

This section will enumerate a number of features that HAProxy implements, some of which are generally expected from any modern load balancer, and some of which are a direct benefit of HAProxy's architecture. More advanced features will be detailed in the next section.

3.3.1. Basic features : Proxying

Proxying is the action of transferring data between a client and a server over two independent connections. The following basic features are supported by HAProxy regarding proxying and connection management :

- Provide the server with a clean connection to protect them against any client-side defect or attack;
- Listen to multiple IP addresses and/or ports, even port ranges;
- Transparent accept : intercept traffic targeting any arbitrary IP address that doesn't even belong to the local system;
- Server port doesn't need to be related to listening port, and may even be translated by a fixed offset (useful with ranges);
- Transparent connect : spoof the client's (or any) IP address if needed when connecting to the server;
- Provide a reliable return IP address to the servers in multi-site LBs;
- Offload the server thanks to buffers and possibly short-lived connections to reduce their concurrent connection count and their memory footprint;
- Optimize TCP stacks (e.g. SACK), congestion control, and reduce RTT impacts;
- Support different protocol families on both sides (e.g. IPv4/IPv6/Unix);
- Timeout enforcement : HAProxy supports multiple levels of timeouts depending on the stage the connection is, so that a dead client or server, or an attacker cannot be granted resources for too long;
- Protocol validation: HTTP, SSL, or payload are inspected and invalid protocol elements are rejected, unless instructed to accept them anyway;
- Policy enforcement : ensure that only what is allowed may be forwarded;
- Both incoming and outgoing connections may be limited to certain network namespaces (Linux only), making it easy to build a cross-container, multi-tenant load balancer;
- PROXY protocol presents the client's IP address to the server even for non-HTTP traffic. This is an HAProxy extension that was adopted by a number of third-party products by now, at least these ones at the time of writing :
  - client : haproxy, stud, stunnel, exaproxy, ELB, squid
  - server : haproxy, stud, postfix, exim, nginx, squid, node.js, varnish

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.2. Basic features : SSL

HAProxy's SSL stack is recognized as one of the most featureful according to Google's engineers (http://istlsfastyet.com/). The most commonly used features making it quite complete are :

- SNI-based multi-hosting with no limit on sites count and focus on performance. At least one deployment is known for running 50000 domains with their respective certificates;

- support for wildcard certificates reduces the need for many certificates ;

- certificate-based client authentication with configurable policies on failure to present a valid certificate. This allows to present a different server farm to regenerate the client certificate for example;

- authentication of the backend server ensures the backend server is the real one and not a man in the middle;

- authentication with the backend server lets the backend server know it's really the expected haproxy node that is connecting to it;

- TLS NPN and ALPN extensions make it possible to reliably offload SPDY/HTTP2 connections and pass them in clear text to backend servers;

- OCSP stapling further reduces first page load time by delivering inline an OCSP response when the client requests a Certificate Status Request;

- Dynamic record sizing provides both high performance and low latency, and significantly reduces page load time by letting the browser start to fetch new objects while packets are still in flight;

- permanent access to all relevant SSL/TLS layer information for logging, access control, reporting etc. These elements can be embedded into HTTP header or even as a PROXY protocol extension so that the offloaded server gets all the information it would have had if it performed the SSL termination itself.

- Detect, log and block certain known attacks even on vulnerable SSL libs, such as the Heartbleed attack affecting certain versions of OpenSSL.

- support for stateless session resumption (RFC 5077 TLS Ticket extension). TLS tickets can be updated from CLI which provides them means to implement Perfect Forward Secrecy by frequently rotating the tickets.

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.3. Basic features : Monitoring

HAProxy focuses a lot on availability. As such it cares about servers state, and about reporting its own state to other network components :

- Servers' state is continuously monitored using per-server parameters. This ensures the path to the server is operational for regular traffic;

- Health checks support two hysteresis for up and down transitions in order to protect against state flapping;

- Checks can be sent to a different address/port/protocol : this makes it easy to check a single service that is considered representative of multiple ones, for example the HTTPS port for an HTTP+HTTPS server.

- Servers can track other servers and go down simultaneously : this ensures that servers hosting multiple services can fail atomically and that no one will be sent to a partially failed server;

- Agents may be deployed on the server to monitor load and health : a server may be interested in reporting its load, operational status, administrative status independently from what health checks can see. By running a simple agent on the server, it's possible to consider the server's view of its own health in addition to the health checks validating the whole path;

- Various check methods are available : TCP connect, HTTP request, SMTP hello, SSL hello, LDAP, SQL, Redis, send/expect scripts, all with/without SSL;

- State change is notified in the logs and stats page with the failure reason (e.g. the HTTP response received at the moment the failure was detected). An e-mail can also be sent to a configurable address upon such a change ;

- Server state is also reported on the stats interface and can be used to take routing decisions so that traffic may be sent to different farms depending on their sizes and/or health (e.g. loss of an inter-DC link);

- HAProxy can use health check requests to pass information to the servers, such as their names, weight, the number of other servers in the farm etc. so that servers can adjust their response and decisions based on this knowledge (e.g. postpone backups to keep more CPU available);

- Servers can use health checks to report more detailed state than just on/off (e.g. I would like to stop, please stop sending new visitors);

- HAProxy itself can report its state to external components such as routers or other load balancers, allowing to build very complete multi-path and multi-layer infrastructures.

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.4. Basic features : High availability

Just like any serious load balancer, HAProxy cares a lot about availability to ensure the best global service continuity :

- Only valid servers are used ; the other ones are automatically evicted from load balancing farms ; under certain conditions it is still possible to force to use them though;

- Support for a graceful shutdown so that it is possible to take servers out of a farm without affecting any connection;

- Backup servers are automatically used when active servers are down and replace them so that sessions are not lost when possible. This also allows to build multiple paths to reach the same server (e.g. multiple interfaces);

- Ability to return a global failed status for a farm when too many servers are down. This, combined with the monitoring capabilities makes it possible for an upstream component to choose a different LB node for a given service;

- Stateless design makes it easy to build clusters : by design, HAProxy does its best to ensure the highest service continuity without having to store information that could be lost in the event of a failure. This ensures that a takeover is the most seamless possible;

- Integrates well with standard VRRP daemon keepalived : HAProxy easily tells keepalived about its state and copes very well with floating virtual IP addresses. Note: only use IP redundancy protocols (VRRP/CARP) over cluster-based solutions (Heartbeat, ...) as they're the ones offering the fastest, most seamless, and most reliable switchover.

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.5. Basic features : Load balancing

HAProxy offers a fairly complete set of load balancing features, most of which are unfortunately not available in a number of other load balancing products :

- no less than 9 load balancing algorithms are supported, some of which apply to input data to offer an infinite list of possibilities. The most common ones are round-robin (for short connections, pick each server in turn), leastconn (for long connections, pick the least recently used of the servers with the lowest connection count), source (for SSL farms or terminal server farms, the server directly depends on the client's source address), URI (for HTTP caches, the server directly depends on the HTTP URI), hdr (the server directly depends on the contents of a specific HTTP header field), first (for short-lived virtual machines, all connections are packed on the smallest possible subset of servers so that unused ones can be powered down);

- all algorithms above support per-server weights so that it is possible to accommodate from different server generations in a farm, or direct a small fraction of the traffic to specific servers (debug mode, running the next version of the software, etc);

- dynamic weights are supported for round-robin, leastconn and consistent hashing ; this allows server weights to be modified on the fly from the CLI or even by an agent running on the server;

- slow-start is supported whenever a dynamic weight is supported; this allows a server to progressively take the traffic. This is an important feature for fragile application servers which require to compile classes at runtime as well as cold caches which need to fill up before being run at full throttle;

- hashing can apply to various elements such as client's source address, URL components, query string element, header field values, POST parameter, RDP cookie;

- consistent hashing protects server farms against massive redistribution when adding or removing servers in a farm. That's very important in large cache farms and it allows slow-start to be used to refill cold caches;

- a number of internal metrics such as the number of connections per server, per backend, the amount of available connection slots in a backend etc makes it possible to build very advanced load balancing strategies.

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.6. Basic features : Stickiness

Application load balancing would be useless without stickiness. HAProxy provides a fairly comprehensive set of possibilities to maintain a visitor on the same server even across various events such as server addition/removal, down/up cycles, and some methods are designed to be resistant to the distance between multiple load balancing nodes in that they don't require any replication :

- stickiness information can be individually matched and learned from different places if desired. For example a JSESSIONID cookie may be matched both in a cookie and in the URL. Up to 8 parallel sources can be learned at the same time and each of them may point to a different stick-table;

- stickiness information can come from anything that can be seen within a request or response, including source address, TCP payload offset and length, HTTP query string elements, header field values, cookies, and so on.

- stick-tables are replicated between all nodes in a multi-master fashion;

- commonly used elements such as SSL-ID or RDP cookies (for TSE farms) are directly accessible to ease manipulation;

- all sticking rules may be dynamically conditioned by ACLs;

- it is possible to decide not to stick to certain servers, such as backup servers, so that when the nominal server comes back, it automatically takes the load back. This is often used in multi-path environments;

- in HTTP it is often preferred not to learn anything and instead manipulate a cookie dedicated to stickiness. For this, it's possible to detect, rewrite, insert or prefix such a cookie to let the client remember what server was assigned;

- the server may decide to change or clean the stickiness cookie on logout, so that leaving visitors are automatically unbound from the server;

- using ACL-based rules it is also possible to selectively ignore or enforce stickiness regardless of the server's state; combined with advanced health checks, that helps admins verify that the server they're installing is up and running before presenting it to the whole world;

- an innovative mechanism to set a maximum idle time and duration on cookies ensures that stickiness can be smoothly stopped on devices which are never closed (smartphones, TVs, home appliances) without having to store them on persistent storage;

- multiple server entries may share the same stickiness keys so that stickiness is not lost in multi-path environments when one path goes down;

- soft-stop ensures that only users with stickiness information will continue to reach the server they've been assigned to but no new users will go there.

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.7. Basic features : Sampling and converting informati

HAProxy supports information sampling using a wide set of "sample fetch functions". The principle is to extract pieces of information known as samples, for immediate use. This is used for stickiness, to build conditions, to produce information in logs or to enrich HTTP headers.

Samples can be fetched from various sources :

- constants : integers, strings, IP addresses, binary blocks;
- the process : date, environment variables, server/frontend/backend/process state, byte/connection counts/rates, queue length, random generator, ...
- variables : per-session, per-request, per-response variables;
- the client connection : source and destination addresses and ports, and all related statistics counters;
- the SSL client session : protocol, version, algorithm, cipher, key size, session ID, all client and server certificate fields, certificate serial, SNI, ALPN, NPN, client support for certain extensions;
- request and response buffers contents : arbitrary payload at offset/length, data length, RDP cookie, decoding of SSL hello type, decoding of TLS SNI;
- HTTP (request and response) : method, URI, path, query string arguments, status code, headers values, positional header value, cookies, captures, authentication, body elements;

A sample may then pass through a number of operators known as "converters" to experience some transformation. A converter consumes a sample and produces a new one, possibly of a completely different type. For example, a converter may be used to return only the integer length of the input string, or could turn a string to upper case. Any arbitrary number of converters may be applied in series to a sample before final use. Among all available sample converters, the following ones are the most commonly used :

- arithmetic and logic operators : they make it possible to perform advanced computation on input data, such as computing ratios, percentages or simply converting from one unit to another one;
- IP address masks are useful when some addresses need to be grouped by larger networks;
- data representation : URL-decode, base64, hex, JSON strings, hashing;
- string conversion : extract substrings at fixed positions, fixed length, extract specific fields around certain delimiters, extract certain words, change case, apply regex-based substitution;
- date conversion : convert to HTTP date format, convert local to UTC and conversely, add or remove offset;
- lookup an entry in a stick table to find statistics or assigned server;
- map-based key-to-value conversion from a file (mostly used for geolocation).

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.8. Basic features : Maps

Maps are a powerful type of converter consisting in loading a two-columns file into memory at boot time, then looking up each input sample from the first column and either returning the corresponding pattern on the second column if the entry was found, or returning a default value. The output information also being a sample, it can in turn experience other transformations including other map lookups. Maps are most commonly used to translate the client's IP address to an AS number or country code since they support a longest match for network addresses but they can be used for various other purposes.

Part of their strength comes from being updatable on the fly either from the CLI or from certain actions using other samples, making them capable of storing and retrieving information between subsequent accesses. Another strength comes from the binary tree based indexation which makes them extremely fast even when they contain hundreds of thousands of entries, making geolocation very cheap and easy to set up.

3.3.9. Basic features : ACLs and conditions

Most operations in HAProxy can be made conditional. Conditions are built by combining multiple ACLs using logic operators (AND, OR, NOT). Each ACL is a series of tests based on the following elements :

  - a sample fetch method to retrieve the element to test ;

  - an optional series of converters to transform the element ;

  - a list of patterns to match against ;

  - a matching method to indicate how to compare the patterns with the sample

For example, the sample may be taken from the HTTP "Host" header, it could then be converted to lower case, then matched against a number of regex patterns using the regex matching method.

Technically, ACLs are built on the same core as the maps, they share the exact same internal structure, pattern matching methods and performance. The only real difference is that instead of returning a sample, they only return "found" or or "not found". In terms of usage, ACL patterns may be declared inline in the configuration file and do not require their own file. ACLs may be named for ease of use or to make configurations understandable. A named ACL may be declared multiple times and it will evaluate all definitions in turn until one matches.

About 13 different pattern matching methods are provided, among which IP address mask, integer ranges, substrings, regex. They work like functions, and just like with any programming language, only what is needed is evaluated, so when a condition involving an OR is already true, next ones are not evaluated, and similarly when a condition involving an AND is already false, the rest of the condition is not evaluated.

There is no practical limit to the number of declared ACLs, and a handful of commonly used ones are provided. However experience has shown that setups using a lot of named ACLs are quite hard to troubleshoot and that sometimes using anonymous ACLs inline is easier as it requires less references out of the scope being analyzed.

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.10. Basic features : Content switching

HAProxy implements a mechanism known as content-based switching. The principle is that a connection or request arrives on a frontend, then the information carried with this request or connection are processed, and at this point it is possible to write ACLs-based conditions making use of these information to decide what backend will process the request. Thus the traffic is directed to one backend or another based on the request's contents. The most common example consists in using the Host header and/or elements from the path (sub-directories or file-name extensions) to decide whether an HTTP request targets a static object or the application, and to route static objects traffic to a backend made of fast and light servers, and all the remaining traffic to a more complex application server, thus constituting a fine-grained virtual hosting solution. This is quite convenient to make multiple technologies coexist as a more global solution.

Another use case of content-switching consists in using different load balancing algorithms depending on various criteria. A cache may use a URI hash while an application would use round-robin.

Last but not least, it allows multiple customers to use a small share of a common resource by enforcing per-backend (thus per-customer connection limits).

Content switching rules scale very well, though their performance may depend on the number and complexity of the ACLs in use. But it is also possible to write dynamic content switching rules where a sample value directly turns into a backend name and without making use of ACLs at all. Such configurations have been reported to work fine at least with 300000 backends in production.

3.3.11. Basic features : Stick-tables

Stick-tables are commonly used to store stickiness information, that is, to keep a reference to the server a certain visitor was directed to. The key is then the identifier associated with the visitor (its source address, the SSL ID of the connection, an HTTP or RDP cookie, the customer number extracted from the URL or from the payload, ...) and the stored value is then the server's identifier.

Stick tables may use 3 different types of samples for their keys : integers, strings and addresses. Only one stick-table may be referenced in a proxy, and it is designated everywhere with the proxy name. Up to 8 keys may be tracked in parallel. The server identifier is committed during request or response processing once both the key and the server are known.

Stick-table contents may be replicated in active-active mode with other HAProxy nodes known as "peers" as well as with the new process during a reload operation so that all load balancing nodes share the same information and take the same routing decision if client's requests are spread over multiple nodes.

Since stick-tables are indexed on what allows to recognize a client, they are often also used to store extra information such as per-client statistics. The extra statistics take some extra space and need to be explicitly declared. The type of statistics that may be stored includes the input and output bandwidth, the number of concurrent connections, the connection rate and count over a period, the amount and frequency of errors, some specific tags and counters, etc. In order to support keeping such information without being forced to stick to a given server, a special "tracking" feature is implemented and allows to track up to 3 simultaneous keys from different tables at the same time regardless of stickiness rules. Each stored statistics may be searched, dumped and cleared from the CLI and adds to the live troubleshooting capabilities.

While this mechanism can be used to surclass a returning visitor or to adjust the delivered quality of service depending on good or bad behavior, it is mostly used to fight against service abuse and more generally DDoS as it allows to build complex models to detect certain bad behaviors at a high processing speed.

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.12. Basic features : Formatted strings

There are many places where HAProxy needs to manipulate character strings, such as logs, redirects, header additions, and so on. In order to provide the greatest flexibility, the notion of Formatted strings was introduced, initially for logging purposes, which explains why it's still called "log-format". These strings contain escape characters allowing to introduce various dynamic data including variables and sample fetch expressions into strings, and even to adjust the encoding while the result is being turned into a string (for example, adding quotes). This provides a powerful way to build header contents or to customize log lines. Additionally, in order to remain simple to build most common strings, about 50 special tags are provided as shortcuts for information commonly used in logs.

3.3.13. Basic features : HTTP rewriting and redirection

Installing a load balancer in front of an application that was never designed for this can be a challenging task without the proper tools. One of the most commonly requested operation in this case is to adjust requests and response headers to make the load balancer appear as the origin server and to fix hard coded information. This comes with changing the path in requests (which is strongly advised against), modifying Host header field, modifying the Location response header field for redirects, modifying the path and domain attribute for cookies, and so on. It also happens that a number of servers are somewhat verbose and tend to leak too much information in the response, making them more vulnerable to targeted attacks. While it's theoretically not the role of a load balancer to clean this up, in practice it's located at the best place in the infrastructure to guarantee that everything is cleaned up.

Similarly, sometimes the load balancer will have to intercept some requests and respond with a redirect to a new target URL. While some people tend to confuse redirects and rewriting, these are two completely different concepts, since the rewriting makes the client and the server see different things (and disagree on the location of the page being visited) while redirects ask the client to visit the new URL so that it sees the same location as the server.

In order to do this, HAProxy supports various possibilities for rewriting and redirects, among which :

- regex-based URL and header rewriting in requests and responses. Regex are the most commonly used tool to modify header values since they're easy to manipulate and well understood;

- headers may also be appended, deleted or replaced based on formatted strings so that it is possible to pass information there (e.g. client side TLS algorithm and cipher);

- HTTP redirects can use any 3xx code to a relative, absolute, or completely dynamic (formatted string) URI;

- HTTP redirects also support some extra options such as setting or clearing a specific cookie, dropping the query string, appending a slash if missing, and so on;

- all operations support ACL-based conditions;

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.14. Basic features : Server protection

HAProxy does a lot to maximize service availability, and for this it takes large efforts to protect servers against overloading and attacks. The first and most important point is that only complete and valid requests are forwarded to the servers. The initial reason is that HAProxy needs to find the protocol elements it needs to stay synchronized with the byte stream, and the second reason is that until the request is complete, there is no way to know if some elements will change its semantics. The direct benefit from this is that servers are not exposed to invalid or incomplete requests. This is a very effective protection against slowloris attacks, which have almost no impact on HAProxy.

Another important point is that HAProxy contains buffers to store requests and responses, and that by only sending a request to a server when it's complete and by reading the whole response very quickly from the local network, the server side connection is used for a very short time and this preserves server resources as much as possible.

A direct extension to this is that HAProxy can artificially limit the number of concurrent connections or outstanding requests to a server, which guarantees that the server will never be overloaded even if it continuously runs at 100% of its capacity during traffic spikes. All excess requests will simply be queued to be processed when one slot is released. In the end, this huge resource savings most often ensures so much better server response times that it ends up actually being faster than by overloading the server. Queued requests may be redispatched to other servers, or even aborted in queue when the client aborts, which also protects the servers against the "reload effect", where each click on "reload" by a visitor on a slow-loading page usually induces a new request and maintains the server in an overloaded state.

The slow-start mechanism also protects restarting servers against high traffic levels while they're still finalizing their startup or compiling some classes.

Regarding the protocol-level protection, it is possible to relax the HTTP parser to accept non standard-compliant but harmless requests or responses and even to fix them. This allows bogus applications to be accessible while a fix is being developed. In parallel, offending messages are completely captured with a detailed report that help developers spot the issue in the application. The most dangerous protocol violations are properly detected and dealt with and fixed. For example malformed requests or responses with two Content-length headers are either fixed if the values are exactly the same, or rejected if they differ, since it becomes a security problem. Protocol inspection is not limited to HTTP, it is also available for other protocols like TLS or RDP.

When a protocol violation or attack is detected, there are various options to respond to the user, such as returning the common "HTTP 400 bad request", closing the connection with a TCP reset, or faking an error after a long delay ("tarpit") to confuse the attacker. All of these contribute to protecting the servers by discouraging the offending client from pursuing an attack that becomes very expensive to maintain.

HAProxy also proposes some more advanced options to protect against accidental data leaks and session crossing. Not only it can log suspicious server responses but it will also log and optionally block a response which might affect a given visitors' confidentiality. One such example is a cacheable cookie appearing in a cacheable response and which may result in an intermediary cache to deliver it to another visitor, causing an accidental session sharing.

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.3.15. Basic features : Logging

Logging is an extremely important feature for a load balancer, first because a load balancer is often wrongly accused of causing the problems it reveals, and second because it is placed at a critical point in an infrastructure where all normal and abnormal activity needs to be analyzed and correlated with other components.

HAProxy provides very detailed logs, with millisecond accuracy and the exact connection accept time that can be searched in firewalls logs (e.g. for NAT correlation). By default, TCP and HTTP logs are quite detailed an contain everything needed for troubleshooting, such as source IP address and port, frontend, backend, server, timers (request receipt duration, queue duration, connection setup time, response headers time, data transfer time), global process state, connection counts, queue status, retries count, detailed stickiness actions and disconnect reasons, header captures with a safe output encoding. It is then possible to extend or replace this format to include any sampled data, variables, captures, resulting in very detailed information. For example it is possible to log the number of cumulative requests or number of different URLs visited by a client.

The log level may be adjusted per request using standard ACLs, so it is possible to automatically silent some logs considered as pollution and instead raise warnings when some abnormal behavior happen for a small part of the traffic (e.g. too many URLs or HTTP errors for a source address). Administrative logs are also emitted with their own levels to inform about the loss or recovery of a server for example.

Each frontend and backend may use multiple independent log outputs, which eases multi-tenancy. Logs are preferably sent over UDP, maybe JSON-encoded, and are truncated after a configurable line length in order to guarantee delivery.

3.3.16. Basic features : Statistics

HAProxy provides a web-based statistics reporting interface with authentication, security levels and scopes. It is thus possible to provide each hosted customer with his own page showing only his own instances. This page can be located in a hidden URL part of the regular web site so that no new port needs to be opened. This page may also report the availability of other HAProxy nodes so that it is easy to spot if everything works as expected at a glance. The view is synthetic with a lot of details accessible (such as error causes, last access and last change duration, etc), which are also accessible as a CSV table that other tools may import to draw graphs. The page may self-refresh to be used as a monitoring page on a large display. In administration mode, the page also allows to change server state to ease maintenance operations.

3.4. Advanced features

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.4.1. Advanced features : Management

HAProxy is designed to remain extremely stable and safe to manage in a regular production environment. It is provided as a single executable file which doesn't require any installation process. Multiple versions can easily coexist, meaning that it's possible (and recommended) to upgrade instances progressively by order of importance instead of migrating all of them at once. Configuration files are easily versioned. Configuration checking is done off-line so it doesn't require to restart a service that will possibly fail. During configuration checks, a number of advanced mistakes may be detected (e.g. a rule hiding another one, or stickiness that will not work) and detailed warnings and configuration hints are proposed to fix them. Backwards configuration file compatibility goes very far away in time, with version 1.5 still fully supporting configurations for versions 1.1 written 13 years before, and 1.6 only dropping support for almost unused, obsolete keywords that can be done differently. The configuration and software upgrade mechanism is smooth and non disruptive in that it allows old and new processes to coexist on the system, each handling its own connections. System status, build options, and library compatibility are reported on startup.

Some advanced features allow an application administrator to smoothly stop a server, detect when there's no activity on it anymore, then take it off-line, stop it, upgrade it and ensure it doesn't take any traffic while being upgraded, then test it again through the normal path without opening it to the public, and all of this without touching HAProxy at all. This ensures that even complicated production operations may be done during opening hours with all technical resources available.

The process tries to save resources as much as possible, uses memory pools to save on allocation time and limit memory fragmentation, releases payload buffers as soon as their contents are sent, and supports enforcing strong memory limits above which connections have to wait for a buffer to become available instead of allocating more memory. This system helps guarantee memory usage in certain strict environments.

A command line interface (CLI) is available as a UNIX or TCP socket, to perform a number of operations and to retrieve troubleshooting information. Everything done on this socket doesn't require a configuration change, so it is mostly used for temporary changes. Using this interface it is possible to change a server's address, weight and status, to consult statistics and clear counters, dump and clear stickiness tables, possibly selectively by key criteria, dump and kill client-side and server-side connections, dump captured errors with a detailed analysis of the exact cause and location of the error, dump, add and remove entries from ACLs and maps, update TLS shared secrets, apply connection limits and rate limits on the fly to arbitrary frontends (useful in shared hosting environments), and disable a specific frontend to release a listening port (useful when daytime operations are forbidden and a fix is needed nonetheless).

For environments where SNMP is mandatory, at least two agents exist, one is provided with the HAProxy sources and relies on the Net-SNMP Perl module. Another one is provided with the commercial packages and doesn't require Perl. Both are roughly equivalent in terms of coverage.

It is often recommended to install 4 utilities on the machine where HAProxy is deployed :

- socat (in order to connect to the CLI, though certain forks of netcat can also do it to some extents);

- halog from the latest HAProxy version : this is the log analysis tool, it parses native TCP and HTTP logs extremely fast (1 to 2 GB per second) and extracts useful information and statistics such as requests per URL, per source address, URLs sorted by response time or error rate, termination codes etc. It was designed to be deployed on the production servers to help troubleshoot live issues so it has to be there ready to be used;

- tcpdump : this is highly recommended to take the network traces needed to troubleshoot an issue that was made visible in the logs. There is a moment where application and haproxy's analysis will diverge and the network traces

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

are the only way to say who's right and who's wrong. It's also fairly common to detect bugs in network stacks and hypervisors thanks to tcpdump;

- strace : it is tcpdump's companion. It will report what HAProxy really sees and will help sort out the issues the operating system is responsible for from the ones HAProxy is responsible for. Strace is often requested when a bug in HAProxy is suspected;

3.4.2. Advanced features : System-specific capabilities

Depending on the operating system HAProxy is deployed on, certain extra features may be available or needed. While it is supported on a number of platforms, HAProxy is primarily developed on Linux, which explains why some features are only available on this platform.

The transparent bind and connect features, the support for binding connections to a specific network interface, as well as the ability to bind multiple processes to the same IP address and ports are only available on Linux and BSD systems, though only Linux performs a kernel-side load balancing of the incoming requests between the available processes.

On Linux, there are also a number of extra features and optimizations including support for network namespaces (also known as "containers") allowing HAProxy to be a gateway between all containers, the ability to set the MSS, Netfilter marks and IP TOS field on the client side connection, support for TCP FastOpen on the listening side, TCP user timeouts to let the kernel quickly kill connections when it detects the client has disappeared before the configured timeouts, TCP splicing to let the kernel forward data between the two sides of a connections thus avoiding multiple memory copies, the ability to enable the "defer-accept" bind option to only get notified of an incoming connection once data become available in the kernel buffers, and the ability to send the request with the ACK confirming a connect (sometimes called "piggy-back") which is enabled with the "tcp-smart-connect" option. On Linux, HAProxy also takes great care of manipulating the TCP delayed ACKs to save as many packets as possible on the network.

Some systems have an unreliable clock which jumps back and forth in the past and in the future. This used to happen with some NUMA systems where multiple processors didn't see the exact same time of day, and recently it became more common in virtualized environments where the virtual clock has no relation with the real clock, resulting in huge time jumps (sometimes up to 30 seconds have been observed). This causes a lot of trouble with respect to timeout enforcement in general. Due to this flaw of these systems, HAProxy maintains its own monotonic clock which is based on the system's clock but where drift is measured and compensated for. This ensures that even with a very bad system clock, timers remain reasonably accurate and timeouts continue to work. Note that this problem affects all the software running on such systems and is not specific to HAProxy. The common effects are spurious timeouts or application freezes. Thus if this behavior is detected on a system, it must be fixed, regardless of the fact that HAProxy protects itself against it.

3.4.3. Advanced features : Scripting

HAProxy can be built with support for the Lua embedded language, which opens a wide area of new possibilities related to complex manipulation of requests or responses, routing decisions, statistics processing and so on. Using Lua it is even possible to establish parallel connections to other servers to exchange information. This way it becomes possible (though complex) to develop an authentication system for example. Please refer to the documentation in the file "doc/lua-api/index.rst" for more information on how to use Lua.

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.5. Sizing

Typical CPU usage figures show 15% of the processing time spent in HAProxy versus 85% in the kernel in TCP or HTTP close mode, and about 30% for HAProxy versus 70% for the kernel in HTTP keep-alive mode. This means that the operating system and its tuning have a strong impact on the global performance.

Usages vary a lot between users, some focus on bandwidth, other ones on request rate, others on connection concurrency, others on SSL performance. This section aims at providing a few elements to help with this task.

It is important to keep in mind that every operation comes with a cost, so each individual operation adds its overhead on top of the other ones, which may be negligible in certain circumstances, and which may dominate in other cases.

When processing the requests from a connection, we can say that :

- forwarding data costs less than parsing request or response headers;
- parsing request or response headers cost less than establishing then closing a connection to a server;
- establishing an closing a connection costs less than a TLS resume operation;
- a TLS resume operation costs less than a full TLS handshake with a key computation;
- an idle connection costs less CPU than a connection whose buffers hold data;
- a TLS context costs even more memory than a connection with data;

So in practice, it is cheaper to process payload bytes than header bytes, thus it is easier to achieve high network bandwidth with large objects (few requests per volume unit) than with small objects (many requests per volume unit). This explains why maximum bandwidth is always measured with large objects, while request rate or connection rates are measured with small objects.

Some operations scale well on multiple processes spread over multiple CPUs, and others don't scale as well. Network bandwidth doesn't scale very far because the CPU is rarely the bottleneck for large objects, it's mostly the network bandwidth and data buses to reach the network interfaces. The connection rate doesn't scale well over multiple processors due to a few locks in the system when dealing with the local ports table. The request rate over persistent connections scales very well as it doesn't involve much memory nor network bandwidth and doesn't require to access locked structures. TLS key computation scales very well as it's totally CPU-bound. TLS resume scales moderately well, but reaches its limits around 4 processes where the overhead of accessing the shared table offsets the small gains expected from more power.

The performance numbers one can expect from a very well tuned system are in the following range. It is important to take them as orders of magnitude and to expect significant variations in any direction based on the processor, IRQ setting, memory type, network interface type, operating system tuning and so on.

The following numbers were found on a Core i7 running at 3.7 GHz equipped with a dual-port 10 Gbps NICs running Linux kernel 3.10, HAProxy 1.6 and OpenSSL 1.0.2. HAProxy was running as a single process on a single dedicated CPU core, and two extra cores were dedicated to network interrupts :

- 20 Gbps of maximum network bandwidth in clear text for objects 256 kB or higher, 10 Gbps for 41kB or higher;
- 4.6 Gbps of TLS traffic using AES256-GCM cipher with large objects;
- 83000 TCP connections per second from client to server;
- 82000 HTTP connections per second from client to server;
- 97000 HTTP requests per second in server-close mode (keep-alive with the

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
  client, close with the server);

- 243000 HTTP requests per second in end-to-end keep-alive mode;

- 300000 filtered TCP connections per second (anti-DDoS)

- 160000 HTTPS requests per second in keep-alive mode over persistent TLS
  connections;

- 13100 HTTPS requests per second using TLS resumed connections;

- 1300 HTTPS connections per second using TLS connections renegotiated with
  RSA2048;

- 20000 concurrent saturated connections per GB of RAM, including the memory
  required for system buffers; it is possible to do better with careful tuning
  but this result it easy to achieve.

- about 8000 concurrent TLS connections (client-side only) per GB of RAM,
  including the memory required for system buffers;

- about 5000 concurrent end-to-end TLS connections (both sides) per GB of
  RAM including the memory required for system buffers;

Thus a good rule of thumb to keep in mind is that the request rate is divided
by 10 between TLS keep-alive and TLS resume, and between TLS resume and TLS
renegotiation, while it's only divided by 3 between HTTP keep-alive and HTTP
close. Another good rule of thumb is to remember that a high frequency core
with AES instructions can do around 5 Gbps of AES-GCM per core.

Having more cores rarely helps (except for TLS) and is even counter-productive
due to the lower frequency. In general a small number of high frequency cores
is better.

Another good rule of thumb is to consider that on the same server, HAProxy will
be able to saturate :

  - about 5-10 static file servers or caching proxies;

  - about 100 anti-virus proxies;

  - and about 100-1000 application servers depending on the technology in use.
```

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

3.6. How to get HAProxy

HAProxy is an open source project covered by the GPLv2 license, meaning that everyone is allowed to redistribute it provided that access to the sources is also provided upon request, especially if any modifications were made.

HAProxy evolves as a main development branch called "master" or "mainline", from which new branches are derived once the code is considered stable. A lot of web sites run some development branches in production on a voluntarily basis, either to participate to the project or because they need a bleeding edge feature, and their feedback is highly valuable to fix bugs and judge the overall quality and stability of the version being developed.

The new branches that are created when the code is stable enough constitute a stable version and are generally maintained for several years, so that there is no emergency to migrate to a newer branch even when you're not on the latest. Once a stable branch is issued, it may only receive bug fixes, and very rarely minor feature updates when that makes users' life easier. All fixes that go into a stable branch necessarily come from the master branch. This guarantees that no fix will be lost after an upgrade. For this reason, if you fix a bug, please make the patch against the master branch, not the stable branch. You may even discover it was already fixed. This process also ensures that regressions in a stable branch are extremely rare, so there is never any excuse for not upgrading to the latest version in your current branch.

Branches are numbered with two digits delimited with a dot, such as "1.6". A complete version includes one or two sub-version numbers indicating the level of fix. For example, version 1.5.14 is the 14th fix release in branch 1.5 after version 1.5.0 was issued. It contains 126 fixes for individual bugs, 24 updates on the documentation, and 75 other backported patches, most of which were needed to fix the aforementioned 126 bugs. An existing feature may never be modified nor removed in a stable branch, in order to guarantee that upgrades within the same branch will always be harmless.

HAProxy is available from multiple sources, at different release rhythms :

  - The official community web site : http://www.haproxy.org/ : this site provides the sources of the latest development release, all stable releases, as well as nightly snapshots for each branch. The release cycle is not fast, several months between stable releases, or between development snapshots. Very old versions are still supported there. Everything is provided as sources only, so whatever comes from there needs to be rebuilt and/or repackaged;

  - A number of operating systems such as Linux distributions and BSD ports. These systems generally provide long-term maintained versions which do not always contain all the fixes from the official ones, but which at least contain the critical fixes. It often is a good option for most users who do not seek advanced configurations and just want to keep updates easy;

  - Commercial versions from http://www.haproxy.com/ : these are supported professional packages built for various operating systems or provided as appliances, based on the latest stable versions and including a number of features backported from the next release for which there is a strong demand. It is the best option for users seeking the latest features with the reliability of a stable branch, the fastest response time to fix bugs, or simply support contracts on top of an open source product;

In order to ensure that the version you're using is the latest one in your branch, you need to proceed this way :

  - verify which HAProxy executable you're running : some systems ship it by default and administrators install their versions somewhere else on the system, so it is important to verify in the startup scripts which one is used;

  - determine which source your HAProxy version comes from. For this, it's generally sufficient to type "haproxy -v". A development version will

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

```
   appear like this, with the "dev" word after the branch number :

     HA-Proxy version 1.6-dev3-385ecc-68 2015/08/18

   A stable version will appear like this, as well as unmodified stable
   versions provided by operating system vendors :

     HA-Proxy version 1.5.14 2015/07/02

   And a nightly snapshot of a stable version will appear like this with an
   hexadecimal sequence after the version, and with the date of the snapshot
   instead of the date of the release :

     HA-Proxy version 1.5.14-e4766ba 2015/07/29

   Any other format may indicate a system-specific package with its own
   patch set. For example HAProxy Enterprise versions will appear with the
   following format (<branch>-<latest commit>-<revision>) :

     HA-Proxy version 1.5.0-994126-357 2015/07/02

 - for system-specific packages, you have to check with your vendor's package
   repository or update system to ensure that your system is still supported,
   and that fixes are still provided for your branch. For community versions
   coming from haproxy.org, just visit the site, verify the status of your
   branch and compare the latest version with yours to see if you're on the
   latest one. If not you can upgrade. If your branch is not maintained
   anymore, you're definitely very late and will have to consider an upgrade
   to a more recent branch (carefully read the README when doing so).

HAProxy will have to be updated according to the source it came from. Usually it
follows the system vendor's way of upgrading a package. If it was taken from
sources, please read the README file in the sources directory after extracting
the sources and follow the instructions for your operating system.
```

4. Companion products and alternati

HAProxy integrates fairly well with certain products listed below, which is why they are mentioned here even if not directly related to HAProxy.

4.1. Apache HTTP server

Apache is the de-facto standard HTTP server. It's a very complete and modular project supporting both file serving and dynamic contents. It can serve as a frontend for some application servers. It can even proxy requests and cache responses. In all of these use cases, a front load balancer is commonly needed. Apache can work in various modes, some being heavier than others. Certain modules still require the heavier pre-forked model and will prevent Apache from scaling well with a high number of connections. In this case HAProxy can provide a tremendous help by enforcing the per-server connection limits to a safe value and will significantly speed up the server and preserve its resources that will be better used by the application.

Apache can extract the client's address from the X-Forwarded-For header by using the "mod_rpaf" extension. HAProxy will automatically feed this header when "option forwardfor" is specified in its configuration. HAProxy may also offer a nice protection to Apache when exposed to the internet, where it will better resist a wide number of types of DoS attacks.

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**

4.2. NGINX

NGINX is the second de-facto standard HTTP server. Just like Apache, it covers a wide range of features. NGINX is built on a similar model as HAProxy so it has no problem dealing with tens of thousands of concurrent connections. When used as a gateway to some applications (e.g. using the included PHP FPM) it can often be beneficial to set up some frontend connection limiting to reduce the load on the PHP application. HAProxy will clearly be useful there both as a regular load balancer and as the traffic regulator to speed up PHP by decongesting it. Also since both products use very little CPU thanks to their event-driven architecture, it's often easy to install both of them on the same system. NGINX implements HAProxy's PROXY protocol, thus it is easy for HAProxy to pass the client's connection information to NGINX so that the application gets all the relevant information. Some benchmarks have also shown that for large static file serving, implementing consistent hash on HAProxy in front of NGINX can be beneficial by optimizing the OS' cache hit ratio, which is basically multiplied by the number of server nodes.

4.3. Varnish

Varnish is a smart caching reverse-proxy, probably best described as a web application accelerator. Varnish doesn't implement SSL/TLS and wants to dedicate all of its CPU cycles to what it does best. Varnish also implements HAProxy's PROXY protocol so that HAProxy can very easily be deployed in front of Varnish as an SSL offloader as well as a load balancer and pass it all relevant client information. Also, Varnish naturally supports decompression from the cache when a server has provided a compressed object, but doesn't compress however. HAProxy can then be used to compress outgoing data when backend servers do not implement compression, though it's rarely a good idea to compress on the load balancer unless the traffic is low.

When building large caching farms across multiple nodes, HAProxy can make use of consistent URL hashing to intelligently distribute the load to the caching nodes and avoid cache duplication, resulting in a total cache size which is the sum of all caching nodes.

4.4. Alternatives

Linux Virtual Server (LVS or IPVS) is the layer 4 load balancer included within the Linux kernel. It works at the packet level and handles TCP and UDP. In most cases it's more a complement than an alternative since it doesn't have layer 7 knowledge at all.

Pound is another well-known load balancer. It's much simpler and has much less features than HAProxy but for many very basic setups both can be used. Its author has always focused on code auditability first and wants to maintain the set of features low. Its thread-based architecture scales less well with high connection counts, but it's a good product.

Pen is a quite light load balancer. It supports SSL, maintains persistence using a fixed-size table of its clients' IP addresses. It supports a packet-oriented mode allowing it to support direct server return and UDP to some extents. It is meant for small loads (the persistence table only has 2048 entries).

NGINX can do some load balancing to some extents, though it's clearly not its primary function. Production traffic is used to detect server failures, the load balancing algorithms are more limited, and the stickiness is very limited. But it can make sense in some simple deployment scenarios where it is already present. The good thing is that since it integrates very well with HAProxy, there's nothing wrong with adding HAProxy later when its limits have been reached.

Varnish also does some load balancing of its backend servers and does support real health checks. It doesn't implement stickiness however, so just like with NGINX, as long as stickiness is not needed that can be enough to start with. And similarly, since HAProxy and Varnish integrate so well together, it's easy to add it later into the mix to complement the feature set.

Summary

Keyboard navigation :

You can use **left** and **right** arrow keys to navigate between chapters.

Converted with haproxy-dconv (https://github.com/cbonte/haproxy-dconv) v**0.4.2-15** on **2024/04/05**