```
                              ------------------------
                              HAProxy Management Guide
                              ------------------------
                                    version 1.6


This document describes how to start, stop, manage, and troubleshoot HAProxy,
as well as some known limitations and traps to avoid. It does not describe how
to configure it (for this please read configuration.txt).

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
2.    Quick reminder about HAProxy's architecture
3.    Starting HAProxy
4.    Stopping and restarting HAProxy
5.    File-descriptor limitations
6.    Memory management
7.    CPU usage
8.    Logging
9.    Statistics and monitoring
9.1.      CSV format
9.2.      Unix Socket commands
10.   Tricks for easier configuration management
11.   Well-known traps to avoid
12.   Debugging and performance issues
13.   Security considerations


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.


2. Quick reminder about HAProxy's architecture
----------------------------------------------

HAProxy is a single-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.

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 name
  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
-------------------

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 with '-'
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 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
```

```
131       ordering still applies. When more than one file is specified, each file
132       must start on a section boundary, so the first keyword of each file must be
133       one of "global", "defaults", "peers", "listen", "frontend", "backend", and
134       so on. A file cannot contain just a server list for example.
135
136     -f <cfgfile> : adds <cfgfile> to the list of configuration files to be
137       loaded. Configuration files are loaded and processed in their declaration
138       order. This option may be specified multiple times to load multiple files.
139       See also "--". The difference between "--" and "-f" is that one "-f" must
140       be placed before each file name, while a single "--" is needed before all
141       file names. Both options can be used together, the command line ordering
142       still applies. When more than one file is specified, each file must start
143       on a section boundary, so the first keyword of each file must be one of
144       "global", "defaults", "peers", "listen", "frontend", "backend", and so
145       on. A file cannot contain just a server list for example.
146
147     -C <dir> : changes to directory <dir> before loading configuration
148       files. This is useful when using relative paths. Warning when using
149       wildcards after "--" which are in fact replaced by the shell before
150       starting haproxy.
151
152     -D : start as a daemon. The process detaches from the current terminal after
153       forking, and errors are not reported anymore in the terminal. It is
154       equivalent to the "daemon" keyword in the "global" section of the
155       configuration. It is recommended to always force it in any init script so
156       that a faulty configuration doesn't prevent the system from booting.
157
158     -Ds : work in systemd mode. Only used by the systemd wrapper.
159
160     -L <name> : change the local peer name to <name>, which defaults to the local
161       hostname. This is used only with peers replication.
162
163     -N <limit> : sets the default per-proxy maxconn to <limit> instead of the
164       builtin default value (usually 2000). Only useful for debugging.
165
166     -V : enable verbose mode (disables quiet mode). Reverts the effect of "-q" or
167       "quiet".
168
169     -c : only performs a check of the configuration files and exits before trying
170       to bind. The exit status is zero if everything is OK, or non-zero if an
171       error is encountered.
172
173     -d : enable debug mode. This disables daemon mode, forces the process to stay
174       in foreground and to show incoming and outgoing events. It is equivalent to
175       the "global" section's "debug" keyword. It must never be used in an init
176       script.
177
178     -dG : disable use of getaddrinfo() to resolve host names into addresses. It
179       can be used when suspecting that getaddrinfo() doesn't work as expected.
180       This option was made available because many bogus implementations of
181       getaddrinfo() exist on various systems and cause anomalies that are
182       difficult to troubleshoot.
183
184     -dM[<byte>] : forces memory poisonning, which means that each and every
185       memory region allocated with malloc() or pool_alloc2() will be filled with
186       <byte> before being passed to the caller. When <byte> is not specified, it
187       defaults to 0x50 ('P'). While this slightly slows down operations, it is
188       useful to reliably trigger issues resulting from missing initializations in
189       the code that cause random crashes. Note that -dM0 has the effect of
190       turning any malloc() into a calloc(). In any case if a bug appears or
191       disappears when using this option it means there is a bug in haproxy, so
192       please report it.
193
194     -dS : disable use of the splice() system call. It is equivalent to the
195       "global" section's "nosplice" keyword. This may be used when splice() is
```



```
196       suspected to behave improperly or to cause performance issues, or when
197       using strace to see the forwarded data (which do not appear when using
198       splice()).
199
200     -dV : disable SSL verify on the server side. It is equivalent to having
201       "ssl-server-verify none" in the "global" section. This is useful when
202       trying to reproduce production issues out of the production
203       environment. Never use this in an init script as it degrades SSL security
204       to the servers.
205
206     -db : disable background mode and multi-process mode. The process remains in
207       foreground. It is mainly used during development or during small tests, as
208       Ctrl-C is enough to stop the process. Never use it in an init script.
209
210     -de : disable the use of the "epoll" poller. It is equivalent to the "global"
211       section's keyword "noepoll". It is mostly useful when suspecting a bug
212       related to this poller. On systems supporting epoll, the fallback will
213       generally be the "poll" poller.
214
215     -dk : disable the use of the "kqueue" poller. It is equivalent to the
216       "global" section's keyword "nokqueue". It is mostly useful when suspecting
217       a bug related to this poller. On systems supporting kqueue, the fallback
218       will generally be the "poll" poller.
219
220     -dp : disable the use of the "poll" poller. It is equivalent to the "global"
221       section's keyword "nopoll". It is mostly useful when suspecting a bug
222       related to this poller. On systems supporting poll, the fallback will
223       generally be the "select" poller, which cannot be disabled and is limited
224       to 1024 file descriptors.
225
226     -m <limit> : limit the total allocatable memory to <limit> megabytes across
227       all processes. This may cause some connection refusals or some slowdowns
228       depending on the amount of memory needed for normal operations. This is
229       mostly used to force the processes to work in a constrained resource usage
230       scenario. It is important to note that the memory is not shared between
231       processes, so in a multi-process scenario, this value is first divided by
232       global.nbproc before forking.
233
234     -n <limit> : limits the per-process connection limit to <limit>. This is
235       equivalent to the global section's keyword "maxconn". It has precedence
236       over this keyword. This may be used to quickly force lower limits to avoid
237       a service outage on systems where resource limits are too low.
238
239     -p <file> : write all processes' pids into <file> during startup. This is
240       equivalent to the "global" section's keyword "pidfile". The file is opened
241       before entering the chroot jail, and after doing the chdir() implied by
242       "-C". Each pid appears on its own line.
243
244     -q : set "quiet" mode. This disables some messages during the configuration
245       parsing and during startup. It can be used in combination with "-c" to
246       just check if a configuration file is valid or not.
247
248     -sf <pid>* : send the "finish" signal (SIGUSR1) to older processes after boot
249       completion to ask them to finish what they are doing and to leave. <pid>
250       is a list of pids to signal (one per argument). The list ends on any
251       option starting with a "-". It is not a problem if the list of pids is
252       empty, so that it can be built on the fly based on the result of a command
253       like "pidof" or "pgrep".
254
255     -st <pid>* : send the "terminate" signal (SIGTERM) to older processes after
256       boot completion to terminate them immediately without finishing what they
257       were doing. <pid> is a list of pids to signal (one per argument). The list
258       is ends on any option starting with a "-". It is not a problem if the list
259       of pids is empty, so that it can be built on the fly based on the result of
260       a command like "pidof" or "pgrep".
```

```
261
262    -v : report the version and build date.
263
264    -vv : display the version, build options, libraries versions and usable
265      pollers. This output is systematically requested when filing a bug report.
266
267  A safe way to start HAProxy from an init file consists in forcing the deamon
268  mode, storing existing pids to a pid file and using this pid file to notify
269  older processes to finish before leaving :
270
271     haproxy -f /etc/haproxy.cfg \
272             -D -p /var/run/haproxy.pid -sf $(cat /var/run/haproxy.pid)
273
274  When the configuration is split into a few specific files (eg: tcp vs http),
275  it is recommended to use the "-f" option :
276
277     haproxy -f /etc/haproxy/global.cfg -f /etc/haproxy/stats.cfg \
278             -f /etc/haproxy/default-tcp.cfg -f /etc/haproxy/tcp.cfg \
279             -f /etc/haproxy/default-http.cfg -f /etc/haproxy/http.cfg \
280             -D -p /var/run/haproxy.pid -sf $(cat /var/run/haproxy.pid)
281
282  When an unknown number of files is expected, such as customer-specific files,
283  it is recommended to assign them a name starting with a fixed-size sequence
284  number and to use "--" to load them, possibly after loading some defaults :
285
286     haproxy -f /etc/haproxy/global.cfg -f /etc/haproxy/stats.cfg \
287             -f /etc/haproxy/default-tcp.cfg -f /etc/haproxy/tcp.cfg \
288             -f /etc/haproxy/default-http.cfg -f /etc/haproxy/http.cfg \
289             -D -p /var/run/haproxy.pid -sf $(cat /var/run/haproxy.pid) \
290             -f /etc/haproxy/default-customers.cfg -- /etc/haproxy/customers/*
291
292  Sometimes a failure to start may happen for whatever reason. Then it is
293  important to verify if the version of HAProxy you are invoking is the expected
294  version and if it supports the features you are expecting (eg: SSL, PCRE,
295  compression, Lua, etc). This can be verified using "haproxy -vv". Some
296  important information such as certain build options, the target system and
297  the versions of the libraries being used are reported there. It is also what
298  you will systematically be asked for when posting a bug report :
299
300    $ haproxy -vv
301    HA-Proxy version 1.6-dev7-a088d3-4 2015/10/08
302    Copyright 2000-2015 Willy Tarreau <willy@haproxy.org>
303
304    Build options :
305      TARGET  = linux2628
306      CPU     = generic
307      CC      = gcc
308      CFLAGS  = -pg -O0 -g -fno-strict-aliasing -Wdeclaration-after-statement \
309                -DBUFSIZE=8030 -DMAXREWRITE=1030 -DSO_MARK=36 -DTCP_REPAIR=19
310      OPTIONS = USE_ZLIB=1 USE_DLMALLOC=1 USE_OPENSSL=1 USE_LUA=1 USE_PCRE=1
311
312    Default settings :
313      maxconn = 2000, bufsize = 8030, maxrewrite = 1030, maxpollevents = 200
314
315    Encrypted password support via crypt(3): yes
316    Built with zlib version : 1.2.6
317    Compression algorithms supported : identity("identity"), deflate("deflate"), \
318                                       raw-deflate("deflate"), gzip("gzip")
319    Built with OpenSSL version : OpenSSL 1.0.1o 12 Jun 2015
320    Running on OpenSSL version : OpenSSL 1.0.1o 12 Jun 2015
321    OpenSSL library supports TLS extensions : yes
322    OpenSSL library supports SNI : yes
323    OpenSSL library supports prefer-server-ciphers : yes
324    Built with PCRE version : 8.12 2011-01-15
325    PCRE library supports JIT : no (USE_PCRE_JIT not set)
```



```
326    Built with Lua version : Lua 5.3.1
327    Built with transparent proxy support using: IP_TRANSPARENT IP_FREEBIND
328
329    Available polling systems :
330          epoll : pref=300,  test result OK
331           poll : pref=200,  test result OK
332         select : pref=150,  test result OK
333    Total: 3 (3 usable), will use epoll.
334
335  The relevant information that many non-developer users can verify here are :
336    - the version : 1.6-dev7-a088d3-4 above means the code is currently at commit
337      ID "a088d3" which is the 4th one after after official version "1.6-dev7".
338      Version 1.6-dev7 would show as "1.6-dev7-8c1ad7". What matters here is in
339      fact "1.6-dev7". This is the 7th development version of what will become
340      version 1.6 in the future. A development version not suitable for use in
341      production (unless you know exactly what you are doing). A stable version
342      will show as a 3-numbers version, such as "1.5.14-16f863", indicating the
343      14th level of fix on top of version 1.5. This is a production-ready version.
344
345    - the release date : 2015/10/08. It is represented in the universal
346      year/month/day format. Here this means August 8th, 2015. Given that stable
347      releases are issued every few months (1-2 months at the beginning, sometimes
348      6 months once the product becomes very stable), if you're seeing an old date
349      here, it means you're probably affected by a number of bugs or security
350      issues that have since been fixed and that it might be worth checking on the
351      official site.
352
353    - build options : they are relevant to people who build their packages
354      themselves, they can explain why things are not behaving as expected. For
355      example the development version above was built for Linux 2.6.28 or later,
356      targetting a generic CPU (no CPU-specific optimizations), and lacks any
357      code optimization (-O0) so it will perform poorly in terms of performance.
358
359    - libraries versions : zlib version is reported as found in the library
360      itself. In general zlib is considered a very stable product and upgrades
361      are almost never needed. OpenSSL reports two versions, the version used at
362      build time and the one being used, as found on the system. These ones may
363      differ by the last letter but never by the numbers. The build date is also
364      reported because most OpenSSL bugs are security issues and need to be taken
365      seriously, so this library absolutely needs to be kept up to date. Seeing a
366      4-months old version here is highly suspicious and indeed an update was
367      missed. PCRE provides very fast regular expressions and is highly
368      recommended. Certain of its extensions such as JIT are not present in all
369      versions and still young so some people prefer not to build with them,
370      which is why the biuld status is reported as well. Regarding the Lua
371      scripting language, HAProxy expects version 5.3 which is very young since
372      it was released a little time before HAProxy 1.6. It is important to check
373      on the Lua web site if some fixes are proposed for this branch.
374
375    - Available polling systems will affect the process's scalability when
376      dealing with more than about one thousand of concurrent connections. These
377      ones are only available when the correct system was indicated in the TARGET
378      variable during the build. The "epoll" mechanism is highly recommended on
379      Linux, and the kqueue mechanism is highly recommended on BSD. Lacking them
380      will result in poll() or even select() being used, causing a high CPU usage
381      when dealing with a lot of connections.
382
383
384  4. Stopping and restarting HAProxy
385  ----------------------------------
386
387  HAProxy supports a graceful and a hard stop. The hard stop is simple, when the
388  SIGTERM signal is sent to the haproxy process, it immediately quits and all
389  established connections are closed. The graceful stop is triggered when the
390  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.

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. Not that this mechanism is system dependant 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 was reintroduced in kernel 3.9, so if you are observing a connection failure rate above the one mentionned 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 mentionned 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

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 vestigal "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 ot 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.

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

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 pipe (32 bytes) : 5 allocated (160 bytes), 5 used, 3 users [SHARED]
    - Pool hlua_com (48 bytes) : 0 allocated (0 bytes), 0 used, 1 users [SHARED]
    - Pool vars (64 bytes) : 0 allocated (0 bytes), 0 used, 2 users [SHARED]
    - Pool task (112 bytes) : 5 allocated (560 bytes), 5 used, 1 users [SHARED]
    - Pool session (128 bytes) : 1 allocated (128 bytes), 1 used, 2 users [SHARED]
    - Pool http_txn (272 bytes) : 0 allocated (0 bytes), 0 used, 1 users [SHARED]
    - Pool connection (352 bytes) : 2 allocated (704 bytes), 2 used, 1 users [SHARED]
    - Pool hdr_idx (416 bytes) : 0 allocated (0 bytes), 0 used, 1 users [SHARED]
    - Pool stream (864 bytes) : 1 allocated (864 bytes), 1 used, 1 users [SHARED]
    - Pool requri (1024 bytes) : 0 allocated (0 bytes), 0 used, 1 users [SHARED]
    - Pool buffer (8064 bytes) : 3 allocated (24192 bytes), 2 used, 1 users [SHARED]
  Total: 11 pools, 26608 bytes allocated, 18544 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.

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

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

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), adn 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

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
configuration manual for more information regarding all possible log settings.

It is convenient to chose a facility that is not used by other deamons. 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 accidently 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.

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 connections
  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)...)
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)
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
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 [.F..]: 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

9.2. 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
```

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
   ...
   >
```

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.

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.

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 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.

clear counters
  Clear the max values of the statistics counters in each proxy (frontend & backend) and in each server. The cumulated counters are not affected. 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 :
```
        $ 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
```

```
1171
1172  del acl <acl> [<key>|#<ref>]
1173    Delete all the acl entries from the acl <acl> corresponding to the key <key>.
1174    <acl> is the #<id> or the <file> returned by "show acl". If the <ref> is used,
1175    this command delete only the listed reference. The reference can be found with
1176    listing the content of the acl. Note that if the reference <acl> is a file and
1177    is shared with a map, the entry will be also deleted in the map.
1178
1179  del map <map> [<key>|#<ref>]
1180    Delete all the map entries from the map <map> corresponding to the key <key>.
1181    <map> is the #<id> or the <file> returned by "show map". If the <ref> is used,
1182    this command delete only the listed reference. The reference can be found with
1183    listing the content of the map. Note that if the reference <map> is a file and
1184    is shared with a acl, the entry will be also deleted in the map.
1185
1186  disable agent <backend>/<server>
1187    Mark the auxiliary agent check as temporarily stopped.
1188
1189    In the case where an agent check is being run as a auxiliary check, due
1190    to the agent-check parameter of a server directive, new checks are only
1191    initialised when the agent is in the enabled. Thus, disable agent will
1192    prevent any new agent checks from begin initiated until the agent
1193    re-enabled using enable agent.
1194
1195    When an agent is disabled the processing of an auxiliary agent check that
1196    was initiated while the agent was set as enabled is as follows: All
1197    results that would alter the weight, specifically "drain" or a weight
1198    returned by the agent, are ignored. The processing of agent check is
1199    otherwise unchanged.
1200
1201    The motivation for this feature is to allow the weight changing effects
1202    of the agent checks to be paused to allow the weight of a server to be
1203    configured using set weight without being overridden by the agent.
1204
1205    This command is restricted and can only be issued on sockets configured for
1206    level "admin".
1207
1208  disable frontend <frontend>
1209    Mark the frontend as temporarily stopped. This corresponds to the mode which
1210    is used during a soft restart : the frontend releases the port but can be
1211    enabled again if needed. This should be used with care as some non-Linux OSes
1212    are unable to enable it back. This is intended to be used in environments
1213    where stopping a proxy is not even imaginable but a misconfigured proxy must
1214    be fixed. That way it's possible to release the port and bind it into another
1215    process to restore operations. The frontend will appear with status "STOP"
1216    on the stats page.
1217
1218    The frontend may be specified either by its name or by its numeric ID,
1219    prefixed with a sharp ('#').
1220
1221    This command is restricted and can only be issued on sockets configured for
1222    level "admin".
1223
1224  disable health <backend>/<server>
1225    Mark the primary health check as temporarily stopped. This will disable
1226    sending of health checks, and the last health check result will be ignored.
1227    The server will be in unchecked state and considered UP unless an auxiliary
1228    agent check forces it down.
1229
1230    This command is restricted and can only be issued on sockets configured for
1231    level "admin".
1232
1233  disable server <backend>/<server>
1234    Mark the server DOWN for maintenance. In this mode, no more checks will be
1235    performed on the server until it leaves maintenance.
```



```
1236    If the server is tracked by other servers, those servers will be set to DOWN
1237    during the maintenance.
1238
1239    In the statistics page, a server DOWN for maintenance will appear with a
1240    "MAINT" status, its tracking servers with the "MAINT(via)" one.
1241
1242    Both the backend and the server may be specified either by their name or by
1243    their numeric ID, prefixed with a sharp ('#').
1244
1245    This command is restricted and can only be issued on sockets configured for
1246    level "admin".
1247
1248  enable agent <backend>/<server>
1249    Resume auxiliary agent check that was temporarily stopped.
1250
1251    See "disable agent" for details of the effect of temporarily starting
1252    and stopping an auxiliary agent.
1253
1254    This command is restricted and can only be issued on sockets configured for
1255    level "admin".
1256
1257  enable frontend <frontend>
1258    Resume a frontend which was temporarily stopped. It is possible that some of
1259    the listening ports won't be able to bind anymore (eg: if another process
1260    took them since the 'disable frontend' operation). If this happens, an error
1261    is displayed. Some operating systems might not be able to resume a frontend
1262    which was disabled.
1263
1264    The frontend may be specified either by its name or by its numeric ID,
1265    prefixed with a sharp ('#').
1266
1267    This command is restricted and can only be issued on sockets configured for
1268    level "admin".
1269
1270  enable health <backend>/<server>
1271    Resume a primary health check that was temporarily stopped. This will enable
1272    sending of health checks again. Please see "disable health" for details.
1273
1274    This command is restricted and can only be issued on sockets configured for
1275    level "admin".
1276
1277  enable server <backend>/<server>
1278    If the server was previously marked as DOWN for maintenance, this marks the
1279    server UP and checks are re-enabled.
1280
1281    Both the backend and the server may be specified either by their name or by
1282    their numeric ID, prefixed with a sharp ('#').
1283
1284    This command is restricted and can only be issued on sockets configured for
1285    level "admin".
1286
1287  get map <map> <value>
1288  get acl <acl> <value>
1289    Lookup the value <value> in the map <map> or in the ACL <acl>. <map> or <acl>
1290    are the #<id> or the <file> returned by "show map" or "show acl". This command
1291    returns all the matching patterns associated with this map. This is useful for
1292    debugging maps and ACLs. The output format is composed by one line par
1293    matching type. Each line is composed by space-delimited series of words.
1294
1295    The first two words are:
1296
1297       <match method>:   The match method applied. It can be "found", "bool",
1298                         "int", "ip", "bin", "len", "str", "beg", "sub", "dir",
1299                         "dom", "end" or "reg".
1300
```

```
      <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
  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 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 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>
  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>
  Replace the current IP address of a server by the one provided.

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> 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> 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 ssl ocsp-response <response>
  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.
```

```
1431
1432      Example:
1433        openssl ocsp -issuer issuer.pem -cert server.pem \
1434                     -host ocsp.issuer.com:80 -respout resp.der
1435        echo "set ssl ocsp-response $(base64 -w 10000 resp.der)" | \
1436                     socat stdio /var/run/haproxy.stat
1437
1438    set ssl tls-key <id> <tlskey>
1439      Set the next TLS key for the <id> listener to <tlskey>. This key becomes the
1440      ultimate key, while the penultimate one is used for encryption (others just
1441      decrypt). The oldest TLS key present is overwritten. <id> is either a numeric
1442      #<id> or <file> returned by "show tls-keys". <tlskey> is a base64 encoded 48
1443      bit TLS ticket key (ex. openssl rand -base64 48).
1444
1445    set table <table> key <key> [data.<data_type> <value>]*
1446      Create or update a stick-table entry in the table. If the key is not present,
1447      an entry is inserted. See stick-table in section 4.2 to find all possible
1448      values for <data_type>. The most likely use consists in dynamically entering
1449      entries for source IP addresses, with a flag in gpc0 to dynamically block an
1450      IP address or affect its quality of service. It is possible to pass multiple
1451      data_types in a single call.
1452
1453    set timeout cli <delay>
1454      Change the CLI interface timeout for current connection. This can be useful
1455      during long debugging sessions where the user needs to constantly inspect
1456      some indicators without being disconnected. The delay is passed in seconds.
1457
1458    set weight <backend>/<server> <weight>[%]
1459      Change a server's weight to the value passed in argument. If the value ends
1460      with the '%' sign, then the new weight will be relative to the initially
1461      configured weight.  Absolute weights are permitted between 0 and 256.
1462      Relative weights must be positive with the resulting absolute weight is
1463      capped at 256.  Servers which are part of a farm running a static
1464      load-balancing algorithm have stricter limitations because the weight
1465      cannot change once set. Thus for these servers, the only accepted values
1466      are 0 and 100% (or 0 and the initial weight). Changes take effect
1467      immediately, though certain LB algorithms require a certain amount of
1468      requests to consider changes. A typical usage of this command is to
1469      disable a server during an update by setting its weight to zero, then to
1470      enable it again after the update by setting it back to 100%. This command
1471      is restricted and can only be issued on sockets configured for level
1472      "admin". Both the backend and the server may be specified either by their
1473      name or by their numeric ID, prefixed with a sharp ('#').
1474
1475    show errors [<iid>]
1476      Dump last known request and response errors collected by frontends and
1477      backends. If <iid> is specified, the limit the dump to errors concerning
1478      either frontend or backend whose ID is <iid>. This command is restricted
1479      and can only be issued on sockets configured for levels "operator" or
1480      "admin".
1481
1482      The errors which may be collected are the last request and response errors
1483      caused by protocol violations, often due to invalid characters in header
1484      names. The report precisely indicates what exact character violated the
1485      protocol. Other important information such as the exact date the error was
1486      detected, frontend and backend names, the server name (when known), the
1487      internal session ID and the source address which has initiated the session
1488      are reported too.
1489
1490      All characters are returned, and non-printable characters are encoded. The
1491      most common ones (\t = 9, \n = 10, \r = 13 and \e = 27) are encoded as one
1492      letter following a backslash. The backslash itself is encoded as '\\' to
1493      avoid confusion. Other non-printable characters are encoded '\xNN' where
1494      NN is the two-digits hexadecimal representation of the character's ASCII
1495      code.
```



```
1496
1497      Lines are prefixed with the position of their first character, starting at 0
1498      for the beginning of the buffer. At most one input line is printed per line,
1499      and large lines will be broken into multiple consecutive output lines so that
1500      the output never goes beyond 79 characters wide. It is easy to detect if a
1501      line was broken, because it will not end with '\n' and the next line's offset
1502      will be followed by a '+' sign, indicating it is a continuation of previous
1503      line.
1504
1505      Example :
1506            $ echo "show errors" | socat stdio /tmp/sock1
1507        >>> [04/Mar/2009:15:46:56.081] backend http-in (#2) : invalid response
1508              src 127.0.0.1, session #54, frontend fe-eth0 (#1), server s2 (#1)
1509              response length 213 bytes, error at position 23:
1510
1511              00000  HTTP/1.0 200 OK\r\n
1512              00017  header/bizarre:blah\r\n
1513              00038  Location: blah\r\n
1514              00054  Long-line: this is a very long line which should b
1515              00104+ e broken into multiple lines on the output buffer,
1516              00154+  otherwise it would be too large to print in a ter
1517              00204+ minal\r\n
1518              00211  \r\n
1519
1520        In the example above, we see that the backend "http-in" which has internal
1521        ID 2 has blocked an invalid response from its server s2 which has internal
1522        ID 1. The request was on session 54 initiated by source 127.0.0.1 and
1523        received by frontend fe-eth0 whose ID is 1. The total response length was
1524        213 bytes when the error was detected, and the error was at byte 23. This
1525        is the slash ('/') in header name "header/bizarre", which is not a valid
1526        HTTP character for a header name.
1527
1528    show backend
1529      Dump the list of backends available in the running process
1530
1531    show info
1532      Dump info about haproxy status on current process.
1533
1534    show map [<map>]
1535      Dump info about map converters. Without argument, the list of all available
1536      maps is returned. If a <map> is specified, its contents are dumped. <map> is
1537      the #<id> or <file>. The first column is a unique identifier. It can be used
1538      as reference for the operation "del map" and "set map". The second column is
1539      the pattern and the third column is the sample if available. The data returned
1540      are not directly a list of available maps, but are the list of all patterns
1541      composing any map. Many of these patterns can be shared with ACL.
1542
1543    show acl [<acl>]
1544      Dump info about acl converters. Without argument, the list of all available
1545      acls is returned. If a <acl> is specified, its contents are dumped. <acl> if
1546      the #<id> or <file>. The dump format is the same than the map even for the
1547      sample value. The data returned are not a list of available ACL, but are the
1548      list of all patterns composing any ACL. Many of these patterns can be shared
1549      with maps.
1550
1551    show pools
1552      Dump the status of internal memory pools. This is useful to track memory
1553      usage when suspecting a memory leak for example. It does exactly the same
1554      as the SIGQUIT when running in foreground except that it does not flush
1555      the pools.
1556
1557    show servers state [<backend>]
1558      Dump the state of the servers found in the running configuration. A backend
1559      name or identifier may be provided to limit the output to this backend only.
1560
```

```
   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 ouptput 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/...).
                                    In source code: SRV_ST_*.
      srv_admin_state:              Server administrative state (MAINT/DRAIN/...).
                                    In source code: SRV_ADMF_*.
      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/...).
                                    In source code: CHK_RES_*.
      srv_check_health:             Checks rise / fall current counter.
      srv_check_state:              State of the check (ENABLED/PAUSED/...).
                                    In source code: CHK_ST_*.
      srv_agent_state:              State of the agent check (ENABLED/PAUSED/...).
                                    In source code: CHK_ST_*.
      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.

  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".

  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> <type> <sid>]
    Dump statistics in the CSV format. By passing <id>, <type> and <sid>, it is
    possible to dump only selected items :
      - <iid> is a proxy ID, -1 to dump everything
      - <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 :
          $ 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,,,0,0,1000,0,0,0,0,0,0,,,,,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, (...)

          $

      Here, two commands have been issued at once. That way it's easy to find
      which process the stats apply to in multi-process mode. 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.

  show stat 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 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

  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 tls-keys
   Dump all loaded TLS ticket keys. 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".

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.


10. Tricks for easier configuration management
----------------------------------------------

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

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 independant 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 issues

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
```

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 poisonning 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 poisonning 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 poisoonning 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 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 congestionned WAN link, a congestionned 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 indicte a saturated outgoing link.

13. Security considerations

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 perfom 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 foreing 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

```