HAProxy
Configuration Manual

version 1.6
willy tarreau
2015/12/21

This document covers the configuration language as implemented in the version specified above. It does not provide any hint, example or advice. For such documentation, please refer to the Reference Manual or the Architecture Manual. The summary below is meant to help you search sections by name and navigate through the document.

Note to documentation contributors :

This document is formatted with 80 columns per line, with even number of spaces for indentation and without tabs. Please follow these rules strictly so that it remains easily printable everywhere. If a line needs to be printed verbatim and does not fit, please end each line with a backslash ('\') and continue on next line, indented by two characters. It is also sometimes useful to prefix all output lines (logs, console outs) with 3 closing angle brackets ('>>>') in order to help get the difference between inputs and outputs when it can become ambiguous. If you add sections, please update the summary below for easier searching.

Summary

1. Quick reminder about HTTP

When haproxy is running in HTTP mode, both the request and the response are fully analyzed and indexed, thus it becomes possible to build matching criteria on almost anything found in the contents.

However, it is important to understand how HTTP requests and responses are formed, and how HAProxy decomposes them. It will then become easier to write correct rules and to debug existing configurations.

1.1. The HTTP transaction model

The HTTP protocol is transaction-driven. This means that each request will lead to one and only one response. Traditionally, a TCP connection is established from the client to the server, a request is sent by the client on the connection, the server responds and the connection is closed. A new request will involve a new connection :

```
  [CON1] [REQ1] ... [RESP1] [CLO1] [CON2] [REQ2] ... [RESP2] [CLO2] ...
```

In this mode, called the "HTTP close" mode, there are as many connection establishments as there are HTTP transactions. Since the connection is closed by the server after the response, the client does not need to know the content length.

Due to the transactional nature of the protocol, it was possible to improve it to avoid closing a connection between two subsequent transactions. In this mode however, it is mandatory that the server indicates the content length for each response so that the client does not wait indefinitely. For this, a special header is used: "Content-length". This mode is called the "keep-alive" mode :

```
  [CON] [REQ1] ... [RESP1] [REQ2] ... [RESP2] [CLO] ...
```

Its advantages are a reduced latency between transactions, and less processing power required on the server side. It is generally better than the close mode, but not always because the clients often limit their concurrent connections to a smaller value.

A last improvement in the communications is the pipelining mode. It still uses keep-alive, but the client does not wait for the first response to send the second request. This is useful for fetching large number of images composing a page :

```
  [CON] [REQ1] [REQ2] ... [RESP1] [RESP2] [CLO] ...
```

This can obviously have a tremendous benefit on performance because the network latency is eliminated between subsequent requests. Many HTTP agents do not correctly support pipelining since there is no way to associate a response with the corresponding request in HTTP. For this reason, it is mandatory for the server to reply in the exact same order as the requests were received.

By default HAProxy operates in keep-alive mode with regards to persistent connections: for each connection it processes each request and response, and leaves the connection idle on both sides between the end of a response and the start of a new request.

HAProxy supports 5 connection modes :

```
  - keep alive    : all requests and responses are processed (default)
  - tunnel        : only the first request and response are processed,
                    everything else is forwarded with no analysis.
  - passive close : tunnel with "Connection: close" added in both directions.
  - server close  : the server-facing connection is closed after the response.
  - forced close  : the connection is actively closed after end of response.
```

1.2. HTTP request

First, let's consider this HTTP request :

```
  Line     Contents
  number
     1     GET /serv/login.php?lang=en&profile=2 HTTP/1.1
     2     Host: www.mydomain.com
     3     User-agent: my small browser
     4     Accept: image/jpeg, image/gif
     5     Accept: image/png
```

1.2.1. The Request line

Line 1 is the "request line". It is always composed of 3 fields :

```
  - a METHOD      : GET
  - a URI         : /serv/login.php?lang=en&profile=2
```

```
  - a version tag : HTTP/1.1
```

All of them are delimited by what the standard calls LWS (linear white spaces), which are commonly spaces, but can also be tabs or line feeds/carriage returns followed by spaces/tabs. The method itself cannot contain any colon (':') and is limited to alphabetic letters. All those various combinations make it desirable that HAProxy performs the splitting itself rather than leaving it to the user to write a complex or inaccurate regular expression.

The URI itself can have several forms :

- A "relative URI" :

  ```
  /serv/login.php?lang=en&profile=2
  ```

  It is a complete URL without the host part. This is generally what is received by servers, reverse proxies and transparent proxies.

- An "absolute URI", also called a "URL" :

  ```
  http://192.168.0.12:8080/serv/login.php?lang=en&profile=2
  ```

  It is composed of a "scheme" (the protocol name followed by '://'), a host name or address, optionally a colon (':') followed by a port number, then a relative URI beginning at the first slash ('/') after the address part. This is generally what proxies receive, but a server supporting HTTP/1.1 must accept this form too.

- a star ('*') : this form is only accepted in association with the OPTIONS method and is not relayable. It is used to inquiry a next hop's capabilities.

- an address:port combination : 192.168.0.12:80
  This is used with the CONNECT method, which is used to establish TCP tunnels through HTTP proxies, generally for HTTPS, but sometimes for other protocols too.

In a relative URI, two sub-parts are identified. The part before the question mark is called the "path". It is typically the relative path to static objects on the server. The part after the question mark is called the "query string". It is mostly used with GET requests sent to dynamic scripts and is very specific to the language, framework or application in use.

1.2.2. The request headers

The headers start at the second line. They are composed of a name at the beginning of the line, immediately followed by a colon (':'). Traditionally, an LWS is added after the colon but that's not required. Then come the values. Multiple identical headers may be folded into one single line, delimiting the values with commas, provided that their order is respected. This is commonly encountered in the "Cookie:" field. A header may span over multiple lines if the subsequent lines begin with an LWS. In the example in 1.2, lines 4 and 5 define a total of 3 values for the "Accept:" header.

Contrary to a common mis-conception, header names are not case-sensitive, and their values are not either if they refer to other header names (such as the "Connection:" header).

The end of the headers is indicated by the first empty line. People often say that it's a double line feed, which is not exact, even if a double line feed is one valid form of empty line.

Fortunately, HAProxy takes care of all these complex combinations when indexing headers, checking values and counting them, so there is no reason to worry about the way they could be written, but it is important not to accuse an application of being buggy if it does unusual, valid things.

Important note:
   As suggested by RFC2616, HAProxy normalizes headers by replacing line breaks in the middle of headers by LWS in order to join multi-line headers. This is necessary for proper analysis and helps less capable HTTP parsers to work correctly and not to be fooled by such complex constructs.

1.3. HTTP response

An HTTP response looks very much like an HTTP request. Both are called HTTP messages. Let's consider this HTTP response :

```
  Line     Contents
  number
     1     HTTP/1.1 200 OK
     2     Content-length: 350
     3     Content-Type: text/html
```

As a special case, HTTP supports so called "Informational responses" as status codes 1xx. These messages are special in that they don't convey any part of the response, they're just used as sort of a signaling message to ask a client to continue to post its request for instance. In the case of a status 100 response the requested information will be carried by the next non-100 response message following the informational one. This implies that multiple responses may be sent to a single request, and that this only works when keep-alive is enabled (1xx messages are HTTP/1.1 only). HAProxy handles these messages and is able to correctly forward and skip them, and only process the next non-100 response. As such, these messages are neither logged nor transformed, unless explicitly state otherwise. Status 101 messages indicate that the protocol is changing over the same connection and that haproxy must switch to tunnel mode, just as if a CONNECT had occurred. Then the Upgrade header would contain additional information about the type of protocol the connection is switching to.

1.3.1. The Response line

Line 1 is the "response line". It is always composed of 3 fields :

```
  - a version tag : HTTP/1.1
  - a status code : 200
  - a reason      : OK
```

The status code is always 3-digit. The first digit indicates a general status :

```
 - 1xx = informational message to be skipped (eg: 100, 101)
 - 2xx = OK, content is following   (eg: 200, 206)
 - 3xx = OK, no content following   (eg: 302, 304)
 - 4xx = error caused by the client (eg: 401, 403, 404)
 - 5xx = error caused by the server (eg: 500, 502, 503)
```

Please refer to RFC2616 for the detailed meaning of all such codes. The "reason" field is just a hint, but is not parsed by clients. Anything can be found there, but it's a common practice to respect the well-established messages. It can be composed of one or multiple words, such as "OK", "Found", or "Authentication Required".

Haproxy may emit the following status codes by itself :

```
  Code  When / reason
   200  access to stats page, and when replying to monitoring requests
   301  when performing a redirection, depending on the configured code
   302  when performing a redirection, depending on the configured code
   303  when performing a redirection, depending on the configured code
   307  when performing a redirection, depending on the configured code
   308  when performing a redirection, depending on the configured code
   400  for an invalid or too large request
   401  when an authentication is required to perform the action (when
        accessing the stats page)
   403  when a request is forbidden by a "block" ACL or "reqdeny" filter
   408  when the request timeout strikes before the request is complete
   500  when haproxy encounters an unrecoverable internal error, such as a
        memory allocation failure, which should never happen
   502  when the server returns an empty, invalid or incomplete response, or
        when an "rspdeny" filter blocks the response.
   503  when no server was available to handle the request, or in response to
        monitoring requests which match the "monitor fail" condition
   504  when the response timeout strikes before the server responds
```

The error 4xx and 5xx codes above may be customized (see "errorloc" in section 4.2).

1.3.2. The response headers

Response headers work exactly like request headers, and as such, HAProxy uses the same parsing function for both. Please refer to paragraph 1.2.2 for more details.

2. Configuring HAProxy

2.1. Configuration file format

HAProxy's configuration process involves 3 major sources of parameters :

- the arguments from the command-line, which always take precedence
- the "global" section, which sets process-wide parameters
- the proxies sections which can take form of "defaults", "listen", "frontend" and "backend".

The configuration file syntax consists in lines beginning with a keyword referenced in this manual, optionally followed by one or several parameters delimited by spaces.

2.2. Quoting and escaping

HAProxy's configuration introduces a quoting and escaping system similar to many programming languages. The configuration file supports 3 types: escaping with a backslash, weak quoting with double quotes, and strong quoting with single quotes.

If spaces have to be entered in strings, then they must be escaped by preceding them by a backslash ('\') or by quoting them. Backslashes also have to be escaped by doubling or strong quoting them.

Escaping is achieved by preceding a special character by a backslash ('\'):

```
  \    to mark a space and differentiate it from a delimiter
  \#   to mark a hash and differentiate it from a comment
  \\   to use a backslash
  \'   to use a single quote and differentiate it from strong quoting
  \"   to use a double quote and differentiate it from weak quoting
```

Weak quoting is achieved by using double quotes (""). Weak quoting prevents the interpretation of:

```
       space as a parameter separator
  '    single quote as a strong quoting delimiter
  #    hash as a comment start
```

Weak quoting permits the interpretation of variables, if you want to use a non -interpreted dollar within a double quoted string, you should escape it with a backslash ("\$"), it does not work outside weak quoting.

Interpretation of escaping and special characters are not prevented by weak quoting.

Strong quoting is achieved by using single quotes (''). Inside single quotes, nothing is interpreted, it's the efficient way to quote regexes.

Quoted and escaped strings are replaced in memory by their interpreted equivalent, it allows you to perform concatenation.

```
  Example:
      # those are equivalents:
      log-format %{+Q}o\ %t\ %s\ %{-Q}r
      log-format "%{+Q}o %t %s %{-Q}r"
      log-format '%{+Q}o %t %s %{-Q}r'
      log-format "%{+Q}o %t"' %s %{-Q}r'
      log-format "%{+Q}o %t"' %s'\ %{-Q}r

      # those are equivalents:
      reqrep "^([^\ :]*)\ /static/(.*)"     \1\ /\2
      reqrep "^([^ :]*)\ /static/(.*)"     '\1 /\2'
      reqrep "^([^ :]*)\ /static/(.*)"     "\1 /\2"
      reqrep "^([^ :]*)\ /static/(.*)"     "\1\ /\2"
```

2.3. Environment variables

HAProxy's configuration supports environment variables. Those variables are interpreted only within double quotes. Variables are expanded during the configuration parsing. Variable names must be preceded by a dollar ("$") and optionally enclosed with braces ("{}") similarly to what is done in Bourne shell. Variable names can contain alphanumerical characters or the character underscore ("_") but should not start with a digit.

```
  Example:

      bind "fd@${FD_APP1}"

      log "${LOCAL_SYSLOG}:514" local0 notice   # send to local server

      user "$HAPROXY_USER"
```

2.4. Time format

Some parameters involve values representing time, such as timeouts. These values are generally expressed in milliseconds (unless explicitly stated otherwise) but may be expressed in any other unit by suffixing the unit to the numeric value. It is important to consider this because it will not be repeated for every keyword. Supported units are :

```
  - us : microseconds. 1 microsecond = 1/1000000 second
  - ms : milliseconds. 1 millisecond = 1/1000 second. This is the default.
  - s  : seconds. 1s = 1000ms
  - m  : minutes. 1m = 60s = 60000ms
  - h  : hours.   1h = 60m = 3600s = 3600000ms
  - d  : days.    1d = 24h = 1440m = 86400s = 86400000ms
```

2.4. Examples

```
    # Simple configuration for an HTTP proxy listening on port 80 on all
    # interfaces and forwarding requests to a single backend "servers" with a
    # single server "server1" listening on 127.0.0.1:8000
    global
        daemon
        maxconn 256

    defaults
        mode http
        timeout connect 5000ms
        timeout client 50000ms
        timeout server 50000ms

    frontend http-in
        bind *:80
        default_backend servers

    backend servers
        server server1 127.0.0.1:8000 maxconn 32


    # The same configuration defined with a single listen block. Shorter but
    # less expressive, especially in HTTP mode.
    global
        daemon
        maxconn 256

    defaults
        mode http
        timeout connect 5000ms
        timeout client 50000ms
        timeout server 50000ms

    listen http-in
        bind *:80
        server server1 127.0.0.1:8000 maxconn 32
```

Assuming haproxy is in $PATH, test these configurations in a shell with:

```
    $ sudo haproxy -f configuration.conf -c
```

3. Global parameters

Parameters in the "global" section are process-wide and often OS-specific. They are generally set once for all and do not need being changed once correct. Some of them have command-line equivalents.

The following keywords are supported in the "global" section :

 * Process management and security

  - ca-base
  - chroot
  - crt-base
  - cpu-map
  - daemon
  - description
  - deviceatlas-json-file
  - deviceatlas-log-level
  - deviceatlas-separator
  - deviceatlas-properties-cookie
  - external-check
  - gid
  - group
  - log
  - log-tag
  - log-send-hostname
  - lua-load
  - nbproc
  - node
  - pidfile
  - uid
  - ulimit-n
  - user
  - stats
  - ssl-default-bind-ciphers
  - ssl-default-bind-options
  - ssl-default-server-ciphers
  - ssl-default-server-options
  - ssl-dh-param-file
  - ssl-server-verify
  - unix-bind
  - 51degrees-data-file
  - 51degrees-property-name-list
  - 51degrees-property-separator
  - 51degrees-cache-size

* Performance tuning
  - max-spread-checks
  - maxconn
  - maxconnrate
  - maxcomprate
  - maxcompcpuusage
  - maxpipes
  - maxsessrate
  - maxsslconn
  - maxsslrate
  - maxzlibmem
  - noepoll
  - nokqueue
  - nopoll
  - nosplice
  - nogetaddrinfo
  - spread-checks
  - server-state-base
  - server-state-file
  - tune.buffers.limit
  - tune.buffers.reserve
  - tune.bufsize
  - tune.chksize
  - tune.comp.maxlevel
  - tune.http.cookielen
  - tune.http.maxhdr
  - tune.idletimer
  - tune.lua.forced-yield
  - tune.lua.maxmem

  - tune.lua.session-timeout
  - tune.lua.task-timeout
  - tune.lua.service-timeout
  - tune.maxaccept
  - tune.maxpollevents
  - tune.maxrewrite
  - tune.pattern.cache-size
  - tune.pipesize
  - tune.rcvbuf.client
  - tune.rcvbuf.server
  - tune.sndbuf.client
  - tune.sndbuf.server
  - tune.ssl.cachesize
  - tune.ssl.lifetime
  - tune.ssl.force-private-cache
  - tune.ssl.maxrecord
  - tune.ssl.default-dh-param
  - tune.ssl.ssl-ctx-cache-size
  - tune.vars.global-max-size
  - tune.vars.reqres-max-size
  - tune.vars.sess-max-size
  - tune.vars.txn-max-size
  - tune.zlib.memlevel
  - tune.zlib.windowsize

* Debugging
  - debug
  - quiet

3.1. Process management and security

ca-base <dir>
Assigns a default directory to fetch SSL CA certificates and CRLs from when a relative path is used with "ca-file" or "crl-file" directives. Absolute locations specified in "ca-file" and "crl-file" prevail and ignore "ca-base".

chroot <jail dir>
Changes current directory to <jail dir> and performs a chroot() there before dropping privileges. This increases the security level in case an unknown vulnerability would be exploited, since it would make it very hard for the attacker to exploit the system. This only works when the process is started with superuser privileges. It is important to ensure that <jail_dir> is both empty and unwritable to anyone.

cpu-map <"all"|"odd"|"even"|process_num> <cpu-set>...
On Linux 2.6 and above, it is possible to bind a process to a specific CPU set. This means that the process will never run on other CPUs. The "cpu-map" directive specifies CPU sets for process sets. The first argument is the process number to bind. This process must have a number between 1 and 32 or 64, depending on the machine's word size, and any process IDs above nbproc are ignored. It is possible to specify all processes at once using "all", only odd numbers using "odd" or even numbers using "even", just like with the "bind-process" directive. The second and forthcoming arguments are CPU sets. Each CPU set is either a unique number between 0 and 31 or 63 or a range with two such numbers delimited by a dash ('-'). Multiple CPU numbers or ranges may be specified, and the processes will be allowed to bind to all of them. Obviously, multiple "cpu-map" directives may be specified. Each "cpu-map" directive will replace the previous ones when they overlap.

crt-base <dir>
Assigns a default directory to fetch SSL certificates from when a relative path is used with "crtfile" directives. Absolute locations specified after "crtfile" prevail and ignore "crt-base".

```

daemon
  Makes the process fork into background. This is the recommended mode of
  operation. It is equivalent to the command line "-D" argument. It can be
  disabled by the command line "-db" argument.

deviceatlas-json-file <path>
  Sets the path of the DeviceAtlas JSON data file to be loaded by the API.
  The path must be a valid JSON data file and accessible by Haproxy process.

deviceatlas-log-level <value>
  Sets the level of informations returned by the API. This directive is
  optional and set to 0 by default if not set.

deviceatlas-separator <char>
  Sets the character separator for the API properties results. This directive
  is optional and set to | by default if not set.

deviceatlas-properties-cookie <name>
  Sets the client cookie's name used for the detection if the DeviceAtlas
  Client-side component was used during the request. This directive is optional
  and set to DAPROPS by default if not set.

external-check
  Allows the use of an external agent to perform health checks.
  This is disabled by default as a security precaution.
  See "option external-check".

gid <number>
  Changes the process' group ID to <number>. It is recommended that the group
  ID is dedicated to HAProxy or to a small set of similar daemons. HAProxy must
  be started with a user belonging to this group, or with superuser privileges.
  Note that if haproxy is started from a user having supplementary groups, it
  will only be able to drop these groups if started with superuser privileges.
  See also "group" and "uid".

group <group name>
  Similar to "gid" but uses the GID of group name <group name> from /etc/group.
  See also "gid" and "user".

log <address> [len <length>] [format <format>] <facility> [max level [min level]]
  Adds a global syslog server. Up to two global servers can be defined. They
  will receive logs for startups and exits, as well as all logs from proxies
  configured with "log global".

  <address> can be one of:

        - An IPv4 address optionally followed by a colon and a UDP port. If
          no port is specified, 514 is used by default (the standard syslog
          port).

        - An IPv6 address followed by a colon and optionally a UDP port. If
          no port is specified, 514 is used by default (the standard syslog
          port).

        - A filesystem path to a UNIX domain socket, keeping in mind
          considerations for chroot (be sure the path is accessible inside
          the chroot) and uid/gid (be sure the path is appropriately
          writeable).

        You may want to reference some environment variables in the address
        parameter, see section 2.3 about environment variables.

  <length> is an optional maximum line length. Log lines larger than this value
           will be truncated before being sent. The reason is that syslog
```

```
           servers act differently on log line length. All servers support the
           default value of 1024, but some servers simply drop larger lines
           while others do log them. If a server supports long lines, it may
           make sense to set this value here in order to avoid truncating long
           lines. Similarly, if a server drops long lines, it is preferable to
           truncate them before sending them. Accepted values are 80 to 65535
           inclusive. The default value of 1024 is generally fine for all
           standard usages. Some specific cases of long captures or
           JSON-formated logs may require larger values.

  <format> is the log format used when generating syslog messages. It may be
           one of the following :

    rfc3164   The RFC3164 syslog message format. This is the default.
              (https://tools.ietf.org/html/rfc3164)

    rfc5424   The RFC5424 syslog message format.
              (https://tools.ietf.org/html/rfc5424)

  <facility> must be one of the 24 standard syslog facilities :

          kern   user   mail   daemon auth   syslog lpr    news
          uucp   cron   auth2  ftp    ntp    audit  alert  cron2
          local0 local1 local2 local3 local4 local5 local6 local7

  An optional level can be specified to filter outgoing messages. By default,
  all messages are sent. If a maximum level is specified, only messages with a
  severity at least as important as this level will be sent. An optional minimum
  level can be specified. If it is set, logs emitted with a more severe level
  than this one will be capped to this level. This is used to avoid sending
  "emerg" messages on all terminals on some default syslog configurations.
  Eight levels are known :

          emerg  alert  crit   err    warning notice info  debug

log-send-hostname [<string>]
  Sets the hostname field in the syslog header. If optional "string" parameter
  is set the header is set to the string contents, otherwise uses the hostname
  of the system. Generally used if one is not relaying logs through an
  intermediate syslog server or for simply customizing the hostname printed in
  the logs.

log-tag <string>
  Sets the tag field in the syslog header to this string. It defaults to the
  program name as launched from the command line, which usually is "haproxy".
  Sometimes it can be useful to differentiate between multiple processes
  running on the same host. See also the per-proxy "log-tag" directive.

lua-load <file>
  This global directive loads and executes a Lua file. This directive can be
  used multiple times.

nbproc <number>
  Creates <number> processes when going daemon. This requires the "daemon"
  mode. By default, only one process is created, which is the recommended mode
  of operation. For systems limited to small sets of file descriptors per
  process, it may be needed to fork multiple daemons. USING MULTIPLE PROCESSES
  IS HARDER TO DEBUG AND IS REALLY DISCOURAGED. See also "daemon".

pidfile <pidfile>
  Writes pids of all daemons into file <pidfile>. This option is equivalent to
  the "-p" command line argument. The file must be accessible to the user
  starting the process. See also "daemon".

stats bind-process [ all | odd | even | <number 1-64>[-<number 1-64>] ] ...
```

Limits the stats socket to a certain set of processes numbers. By default the stats socket is bound to all processes, causing a warning to be emitted when nbproc is greater than 1 because there is no way to select the target process when connecting. However, by using this setting, it becomes possible to pin the stats socket to a specific set of processes, typically the first one. The warning will automatically be disabled when this setting is used, whatever the number of processes used. The maximum process ID depends on the machine's word size (32 or 64). A better option consists in using the "process" setting of the "stats socket" line to force the process on each line.

server-state-base <directory>

Specifies the directory prefix to be prepended in front of all servers state file names which do not start with a '/'. See also "server-state-file", "load-server-state-from-file" and "server-state-file-name".

server-state-file <file>

Specifies the path to the file containing state of servers. If the path starts with a slash ('/'), it is considered absolute, otherwise it is considered relative to the directory specified using "server-state-base" (if set) or to the current directory. Before reloading HAProxy, it is possible to save the servers' current state using the stats command "show servers state". The output of this command must be written in the file pointed by <file>. When starting up, before handling traffic, HAProxy will read, load and apply state for each server found in the file and available in its current running configuration. See also "server-state-base" and "show servers state", "load-server-state-from-file" and "server-state-file-name"

ssl-default-bind-ciphers <ciphers>

This setting is only available when support for OpenSSL was built in. It sets the default string describing the list of cipher algorithms ("cipher suite") that are negotiated during the SSL/TLS handshake for all "bind" lines which do not explicitly define theirs. The format of the string is defined in "man 1 ciphers" from OpenSSL man pages, and can be for instance a string such as "AES:ALL:!aNULL:!eNULL:+RC4:@STRENGTH" (without quotes). Please check the "bind" keyword for more information.

ssl-default-bind-options [<option>]...

This setting is only available when support for OpenSSL was built in. It sets default ssl-options to force on all "bind" lines. Please check the "bind" keyword to see available options.

```
  Example:
        global
           ssl-default-bind-options no-sslv3 no-tls-tickets
```

ssl-default-server-ciphers <ciphers>

This setting is only available when support for OpenSSL was built in. It sets the default string describing the list of cipher algorithms that are negotiated during the SSL/TLS handshake with the server, for all "server" lines which do not explicitly define theirs. The format of the string is defined in "man 1 ciphers". Please check the "server" keyword for more information.

ssl-default-server-options [<option>]...

This setting is only available when support for OpenSSL was built in. It sets default ssl-options to force on all "server" lines. Please check the "server" keyword to see available options.

ssl-dh-param-file <file>

This setting is only available when support for OpenSSL was built in. It sets the default DH parameters that are used during the SSL/TLS handshake when ephemeral Diffie-Hellman (DHE) key exchange is used, for all "bind" lines which do not explicitely define theirs. It will be overridden by custom DH parameters found in a bind certificate file if any. If custom DH parameters are not specified either by using ssl-dh-param-file or by setting them

directly in the certificate file, pre-generated DH parameters of the size specified by tune.ssl.default-dh-param will be used. Custom parameters are known to be more secure and therefore their use is recommended. Custom DH parameters may be generated by using the OpenSSL command "openssl dhparam <size>", where size should be at least 2048, as 1024-bit DH parameters should not be considered secure anymore.

ssl-server-verify [none|required]

The default behavior for SSL verify on servers side. If specified to 'none', servers certificates are not verified. The default is 'required' except if forced using cmdline option '-dV'.

stats socket [<address:port>|<path>] [param*]

Binds a UNIX socket to <path> or a TCPv4/v6 address to <address:port>. Connections to this socket will return various statistics outputs and even allow some commands to be issued to change some runtime settings. Please consult section 9.2 "Unix Socket commands" of Management Guide for more details.

All parameters supported by "bind" lines are supported, for instance to restrict access to some users or their access rights. Please consult section 5.1 for more information.

stats timeout <timeout, in milliseconds>

The default timeout on the stats socket is set to 10 seconds. It is possible to change this value with "stats timeout". The value must be passed in milliseconds, or be suffixed by a time unit among { us, ms, s, m, h, d }.

stats maxconn <connections>

By default, the stats socket is limited to 10 concurrent connections. It is possible to change this value with "stats maxconn".

uid <number>

Changes the process' user ID to <number>. It is recommended that the user ID is dedicated to HAProxy or to a small set of similar daemons. HAProxy must be started with superuser privileges in order to be able to switch to another one. See also "gid" and "user".

ulimit-n <number>

Sets the maximum number of per-process file-descriptors to <number>. By default, it is automatically computed, so it is recommended not to use this option.

```
unix-bind [ prefix <prefix> ] [ mode <mode> ] [ user <user> ] [ uid <uid> ]
          [ group <group> ] [ gid <gid> ]
```

Fixes common settings to UNIX listening sockets declared in "bind" statements. This is mainly used to simplify declaration of those UNIX sockets and reduce the risk of errors, since those settings are most commonly required but are also process-specific. The <prefix> setting can be used to force all socket path to be relative to that directory. This might be needed to access another component's chroot. Note that those paths are resolved before haproxy chroots itself, so they are absolute. The <mode>, <user>, <uid>, <group> and <gid> all have the same meaning as their homonyms used by the "bind" statement. If both are specified, the "bind" statement has priority, meaning that the "unix-bind" settings may be seen as process-wide default settings.

user <user name>

Similar to "uid" but uses the UID of user name <user name> from /etc/passwd. See also "uid" and "group".

node <name>

Only letters, digits, hyphen and underscore are allowed, like in DNS names.

This statement is useful in HA configurations where two or more processes or servers share the same IP address. By setting a different node-name on all nodes, it becomes easy to immediately spot what server is handling the traffic.

description <text>

Add a text that describes the instance.

Please note that it is required to escape certain characters (# for example) and this text is inserted into a html page so you should avoid using "<" and ">" characters.

51degrees-data-file <file path>

The path of the 51Degrees data file to provide device detection services. The file should be unzipped and accessible by HAProxy with relevavnt permissions.

Please note that this option is only available when haproxy has been compiled with USE_51DEGREES.

51degrees-property-name-list [<string>]

A list of 51Degrees property names to be load from the dataset. A full list of names is available on the 51Degrees website: https://51degrees.com/resources/property-dictionary

Please note that this option is only available when haproxy has been compiled with USE_51DEGREES.

51degrees-property-separator <char>

A char that will be appended to every property value in a response header containing 51Degrees results. If not set that will be set as ','.

Please note that this option is only available when haproxy has been compiled with USE_51DEGREES.

51degrees-cache-size <number>

Sets the size of the 51Degrees converter cache to <number> entries. This is an LRU cache which reminds previous device detections and their results. By default, this cache is disabled.

Please note that this option is only available when haproxy has been compiled with USE_51DEGREES.

3.2. Performance tuning
----

max-spread-checks <delay in milliseconds>

By default, haproxy tries to spread the start of health checks across the smallest health check interval of all the servers in a farm. The principle is to avoid hammering services running on the same server. But when using large check intervals (10 seconds or more), the last servers in the farm take some time before starting to be tested, which can be a problem. This parameter is used to enforce an upper bound on delay between the first and the last check, even if the servers' check intervals are larger. When servers run with shorter intervals, their intervals will be respected though.

maxconn <number>

Sets the maximum per-process number of concurrent connections to <number>. It is equivalent to the command-line argument "-n". Proxies will stop accepting connections when this limit is reached. The "ulimit-n" parameter is automatically adjusted according to this value. See also "ulimit-n". Note: the "select" poller cannot reliably use more than 1024 file descriptors on some platforms. If your platform only supports select and reports "select FAILED" on startup, you need to reduce maxconn until it works (slightly below 500 in general). If this value is not set, it will default to the value set in DEFAULT_MAXCONN at build time (reported in haproxy -vv) if no memory

limit is enforced, or will be computed based on the memory limit, the buffer size, memory allocated to compression, SSL cache size, and use or not of SSL and the associated maxsslconn (which can also be automatic).

maxconnrate <number>

Sets the maximum per-process number of connections per second to <number>. Proxies will stop accepting connections when this limit is reached. It can be used to limit the global capacity regardless of each frontend capacity. It is important to note that this can only be used as a service protection measure, as there will not necessarily be a fair share between frontends when the limit is reached, so it's a good idea to also limit each frontend to some value close to its expected share. Also, lowering tune.maxaccept can improve fairness.

maxcomprate <number>

Sets the maximum per-process input compression rate to <number> kilobytes per second. For each session, if the maximum is reached, the compression level will be decreased during the session. If the maximum is reached at the beginning of a session, the session will not compress at all. If the maximum is not reached, the compression level will be increased up to tune.comp.maxlevel. A value of zero means there is no limit, this is the default value.

maxcompcpuusage <number>

Sets the maximum CPU usage HAProxy can reach before stopping the compression for new requests or decreasing the compression level of current requests. It works like 'maxcomprate' but measures CPU usage instead of incoming data bandwidth. The value is expressed in percent of the CPU used by haproxy. In case of multiple processes (nbproc > 1), each process manages its individual usage. A value of 100 disable the limit. The default value is 100. Setting a lower value will prevent the compression work from slowing the whole process down and from introducing high latencies.

maxpipes <number>

Sets the maximum per-process number of pipes to <number>. Currently, pipes are only used by kernel-based tcp splicing. Since a pipe contains two file descriptors, the "ulimit-n" value will be increased accordingly. The default value is maxconn/4, which seems to be more than enough for most heavy usages. The splice code dynamically allocates and releases pipes, and can fall back to standard copy, so setting this value too low may only impact performance.

maxsessrate <number>

Sets the maximum per-process number of sessions per second to <number>. Proxies will stop accepting connections when this limit is reached. It can be used to limit the global capacity regardless of each frontend capacity. It is important to note that this can only be used as a service protection measure, as there will not necessarily be a fair share between frontends when the limit is reached, so it's a good idea to also limit each frontend to some value close to its expected share. Also, lowering tune.maxaccept can improve fairness.

maxsslconn <number>

Sets the maximum per-process number of concurrent SSL connections to <number>. By default there is no SSL-specific limit, which means that the global maxconn setting will apply to all connections. Setting this limit avoids having openssl use too much memory and crash when malloc returns NULL (since it unfortunately does not reliably check for such conditions). Note that the limit applies both to incoming and outgoing connections, so one connection which is deciphered then ciphered accounts for 2 SSL connections. If this value is not set, but a memory limit is enforced, this value will be automatically computed based on the memory limit, maxconn, the buffer size, memory allocated to compression, SSL cache size, and use of SSL in either frontends, backends or both. If neither maxconn nor maxsslconn are specified when there is a memory limit, haproxy will automatically adjust these values so that 100% of the connections can be made over SSL with no risk, and will consider the sides where it is enabled (frontend, backend, both).

maxsslrate <number>

Sets the maximum per-process number of SSL sessions per second to <number>. SSL listeners will stop accepting connections when this limit is reached. It can be used to limit the global SSL CPU usage regardless of each frontend capacity. It is important to note that this can only be used as a service protection measure, as there will not necessarily be a fair share between frontends when the limit is reached, so it's a good idea to also limit each frontend to some value close to its expected share. It is also important to note that the sessions are accounted before they enter the SSL stack and not after, which also protects the stack against bad handshakes. Also, lowering tune.maxaccept can improve fairness.

maxzlibmem <number>

Sets the maximum amount of RAM in megabytes per process usable by the zlib. When the maximum amount is reached, future sessions will not compress as long as RAM is unavailable. When sets to 0, there is no limit. The default value is 0. The value is available in bytes on the UNIX socket with "show info" on the line "MaxZlibMemUsage", the memory used by zlib is "ZlibMemUsage" in bytes.

noepoll

Disables the use of the "epoll" event polling system on Linux. It is equivalent to the command-line argument "-de". The next polling system used will generally be "poll". See also "nopoll".

nokqueue

Disables the use of the "kqueue" event polling system on BSD. It is equivalent to the command-line argument "-dk". The next polling system used will generally be "poll". See also "nopoll".

nopoll

Disables the use of the "poll" event polling system. It is equivalent to the command-line argument "-dp". The next polling system used will be "select". It should never be needed to disable "poll" since it's available on all platforms supported by HAProxy. See also "nokqueue" and "noepoll".

nosplice

Disables the use of kernel tcp splicing between sockets on Linux. It is equivalent to the command line argument "-dS". Data will then be copied using conventional and more portable recv/send calls. Kernel tcp splicing is limited to some very recent instances of kernel 2.6. Most versions between 2.6.25 and 2.6.28 are buggy and will forward corrupted data, so they must not be used. This option makes it easier to globally disable kernel splicing in case of doubt. See also "option splice-auto", "option splice-request" and "option splice-response".

nogetaddrinfo

Disables the use of getaddrinfo(3) for name resolving. It is equivalent to the command line argument "-dG". Deprecated gethostbyname(3) will be used.

spread-checks <0..50, in percent>

Sometimes it is desirable to avoid sending agent and health checks to servers at exact intervals, for instance when many logical servers are located on the same physical server. With the help of this parameter, it becomes possible to add some randomness in the check interval between 0 and +/- 50%. A value between 2 and 5 seems to show good results. The default value remains at 0.

tune.buffers.limit <number>

Sets a hard limit on the number of buffers which may be allocated per process. The default value is zero which means unlimited. The minimum non-zero value will always be greater than "tune.buffers.reserve" and should ideally always be about twice as large. Forcing this value can be particularly useful to limit the amount of memory a process may take, while retaining a sane behaviour. When this limit is reached, sessions which need a buffer wait for another one to be released by another session. Since buffers are dynamically allocated and released, the waiting time is very short and not perceptible provided that limits remain reasonable. In fact sometimes reducing the limit may even increase performance by increasing the CPU cache's efficiency. Tests have shown good results on average HTTP traffic with a limit to 1/10 of the expected global maxconn setting, which also significantly reduces memory usage. The memory savings come from the fact that a number of connections will not allocate 2*tune.bufsize. It is best not to touch this value unless advised to do so by an haproxy core developer.

tune.buffers.reserve <number>

Sets the number of buffers which are pre-allocated and reserved for use only during memory shortage conditions resulting in failed memory allocations. The minimum value is 2 and is also the default. There is no reason a user would want to change this value, it's mostly aimed at haproxy core developers.

tune.bufsize <number>

Sets the buffer size to this size (in bytes). Lower values allow more sessions to coexist in the same amount of RAM, and higher values allow some applications with very large cookies to work. The default value is 16384 and can be changed at build time. It is strongly recommended not to change this from the default value, as very low values will break some services such as statistics, and values larger than default size will increase memory usage, possibly causing the system to run out of memory. At least the global maxconn parameter should be decreased by the same factor as this one is increased. If HTTP request is larger than (tune.bufsize - tune.maxrewrite), haproxy will return HTTP 400 (Bad Request) error. Similarly if an HTTP response is larger than this size, haproxy will return HTTP 502 (Bad Gateway).

tune.chksize <number>

Sets the check buffer size to this size (in bytes). Higher values may help find string or regex patterns in very large pages, though doing so may imply more memory and CPU usage. The default value is 16384 and can be changed at build time. It is not recommended to change this value, but to use better checks whenever possible.

tune.comp.maxlevel <number>

Sets the maximum compression level. The compression level affects CPU usage during compression. This value affects CPU usage during compression. Each session using compression initializes the compression algorithm with this value. The default value is 1.

tune.http.cookielen <number>

Sets the maximum length of captured cookies. This is the maximum value that the "capture cookie xxx len yyy" will be allowed to take, and any upper value will automatically be truncated to this one. It is important not to set too high a value because all cookie captures still allocate this size whatever their configured value (they share a same pool). This value is per request per response, so the memory allocated is twice this value per connection. When not specified, the limit is set to 63 characters. It is recommended not to change this value.

tune.http.maxhdr <number>

Sets the maximum number of headers in a request. When a request comes with a number of headers greater than this value (including the first line), it is rejected with a "400 Bad Request" status code. Similarly, too large responses are blocked with "502 Bad Gateway". The default value is 101, which is enough for all usages, considering that the widely deployed Apache server uses the same limit. It can be useful to push this limit further to temporarily allow a buggy application to work by the time it gets fixed. Keep in mind that each new header consumes 32bits of memory for each session, so don't push this limit too high.

tune.idletimer <timeout>

Sets the duration after which haproxy will consider that an empty buffer is probably associated with an idle stream. This is used to optimally adjust some packet sizes while forwarding large and small data alternatively. The decision to use splice() or to send large buffers in SSL is modulated by this parameter. The value is in milliseconds between 0 and 65535. A value of zero means that haproxy will not try to detect idle streams. The default is 1000, which seems to correctly detect end user pauses (eg: read a page before clicking). There should be not reason for changing this value. Please check tune.ssl.maxrecord below.

tune.lua.forced-yield <number>

This directive forces the Lua engine to execute a yield each <number> of instructions executed. This permits interruptng a long script and allows the HAProxy scheduler to process other tasks like accepting connections or forwarding traffic. The default value is 10000 instructions. If HAProxy often executes some Lua code but more reactivity is required, this value can be lowered. If the Lua code is quite long and its result is absolutely required to process the data, the <number> can be increased.

tune.lua.maxmem

Sets the maximum amount of RAM in megabytes per process usable by Lua. By default it is zero which means unlimited. It is important to set a limit to ensure that a bug in a script will not result in the system running out of memory.

tune.lua.session-timeout <timeout>

This is the execution timeout for the Lua sessions. This is useful for preventing infinite loops or spending too much time in Lua. This timeout counts only the pure Lua runtime. If the Lua does a sleep, the sleep is not taked in account. The default timeout is 4s.

tune.lua.task-timeout <timeout>

Purpose is the same as "tune.lua.session-timeout", but this timeout is dedicated to the tasks. By default, this timeout isn't set because a task may remain alive during of the lifetime of HAProxy. For example, a task used to check servers.

tune.lua.service-timeout <timeout>

This is the execution timeout for the Lua services. This is useful for preventing infinite loops or spending too much time in Lua. This timeout counts only the pure Lua runtime. If the Lua does a sleep, the sleep is not taked in account. The default timeout is 4s.

tune.maxaccept <number>

Sets the maximum number of consecutive connections a process may accept in a row before switching to other work. In single process mode, higher numbers give better performance at high connection rates. However in multi-process modes, keeping a bit of fairness between processes generally is better to increase performance. This value applies individually to each listener, so that the number of processes a listener is bound to is taken into account. This value defaults to 64. In multi-process mode, it is divided by twice the number of processes the listener is bound to. Setting this value to -1 completely disables the limitation. It should normally not be needed to tweak this value.

tune.maxpollevents <number>

Sets the maximum amount of events that can be processed at once in a call to the polling system. The default value is adapted to the operating system. It has been noticed that reducing it below 200 tends to slightly decrease latency at the expense of network bandwidth, and increasing it above 200 tends to trade latency for slightly increased bandwidth.

tune.maxrewrite <number>

Sets the reserved buffer space to this size in bytes. The reserved space is

used for header rewriting or appending. The first reads on sockets will never fill more than bufsize-maxrewrite. Historically it has defaulted to half of bufsize, though that does not make much sense since there are rarely large numbers of headers to add. Setting it too high prevents processing of large requests or responses. Setting it too low prevents addition of new headers to already large requests or to POST requests. It is generally wise to set it to about 1024. It is automatically readjusted to half of bufsize if it is larger than that. This means you don't have to worry about it when changing bufsize.

tune.pattern.cache-size <number>

Sets the size of the pattern lookup cache to <number> entries. This is an LRU cache which reminds previous lookups and their results. It is used by ACLs and maps on slow pattern lookups, namely the ones using the "sub", "reg", "dir", "dom", "end", "bin" match methods as well as the case-insensitive strings. It applies to pattern expressions which means that it will be able to memorize the result of a lookup among all the patterns specified on a configuration line (including all those loaded from files). It automatically invalidates entries which are updated using HTTP actions or on the CLI. The default cache size is set to 10000 entries, which limits its footprint to about 5 MB on 32-bit systems and 8 MB on 64-bit systems. There is a very low risk of collision in this cache, which is in the order of the size of the cache divided by 2^64. Typically, at 10000 requests per second with the default cache size of 10000 entries, there's 1% chance that a brute force attack could cause a single collision after 60 years, or 0.1% after 6 years. This is considered much lower than the risk of a memory corruption caused by aging components. If this is not acceptable, the cache can be disabled by setting this parameter to 0.

tune.pipesize <number>

Sets the kernel pipe buffer size to this size (in bytes). By default, pipes are the default size for the system. But sometimes when using TCP splicing, it can improve performance to increase pipe sizes, especially if it is suspected that pipes are not filled and that many calls to splice() are performed. This has an impact on the kernel's memory footprint, so this must not be changed if impacts are not understood.

tune.rcvbuf.client <number>
tune.rcvbuf.server <number>

Forces the kernel socket receive buffer size on the client or the server side to the specified value in bytes. This value applies to all TCP/HTTP frontends and backends. It should normally never be set, and the default size (0) lets the kernel autotune this value depending on the amount of available memory. However it can sometimes help to set it to very low values (eg: 4096) in order to save kernel memory by preventing it from buffering too large amounts of received data. Lower values will significantly increase CPU usage though.

tune.sndbuf.client <number>
tune.sndbuf.server <number>

Forces the kernel socket send buffer size on the client or the server side to the specified value in bytes. This value applies to all TCP/HTTP frontends and backends. It should normally never be set, and the default size (0) lets the kernel autotune this value depending on the amount of available memory. However it can sometimes help to set it to very low values (eg: 4096) in order to save kernel memory by preventing it from buffering too large amounts of received data. Lower values will significantly increase CPU usage though. Another use case is to prevent write timeouts with extremely slow clients due to the kernel waiting for a large part of the buffer to be read before notifying haproxy again.

tune.ssl.cachesize <number>

Sets the size of the global SSL session cache, in a number of blocks. A block is large enough to contain an encoded session without peer certificate. An encoded session with peer certificate is stored in multiple blocks depending on the size of the peer certificate. A block uses approximately

200 bytes of memory. The default value may be forced at build time, otherwise defaults to 20000. When the cache is full, the most idle entries are purged and reassigned. Higher values reduce the occurrence of such a purge, hence the number of CPU-intensive SSL handshakes by ensuring that all users keep their session as long as possible. All entries are pre-allocated upon startup and are shared between all processes if "nbproc" is greater than 1. Setting this value to 0 disables the SSL session cache.

tune.ssl.force-private-cache

This boolean disables SSL session cache sharing between all processes. It should normally not be used since it will force many renegotiations due to clients hitting a random process. But it may be required on some operating systems where none of the SSL cache synchronization method may be used. In this case, adding a first layer of hash-based load balancing before the SSL layer might limit the impact of the lack of session sharing.

tune.ssl.lifetime <timeout>

Sets how long a cached SSL session may remain valid. This time is expressed in seconds and defaults to 300 (5 min). It is important to understand that it does not guarantee that sessions will last that long, because if the cache is full, the longest idle sessions will be purged despite their configured lifetime. The real usefulness of this setting is to prevent sessions from being used for too long.

tune.ssl.maxrecord <number>

Sets the maximum amount of bytes passed to SSL_write() at a time. Default value 0 means there is no limit. Over SSL/TLS, the client can decipher the data only once it has received a full record. With large records, it means that clients might have to download up to 16kB of data before starting to process them. Limiting the value can improve page load times on browsers located over high latency or low bandwidth networks. It is suggested to find optimal values which fit into 1 or 2 TCP segments (generally 1448 bytes over Ethernet with TCP timestamps enabled, or 1460 when timestamps are disabled), keeping in mind that SSL/TLS add some overhead. Typical values of 1419 and 2859 gave good results during tests. Use "strace -e trace=write" to find the best value. Haproxy will automatically switch to this setting after an idle stream has been detected (see tune.idletimer above).

tune.ssl.default-dh-param <number>

Sets the maximum size of the Diffie-Hellman parameters used for generating the ephemeral/temporary Diffie-Hellman key in case of DHE key exchange. The final size will try to match the size of the server's RSA (or DSA) key (e.g, a 2048 bits temporary DH key for a 2048 bits RSA key), but will not exceed this maximum value. Default value if 1024. Only 1024 or higher values are allowed. Higher values will increase the CPU load, and values greater than 1024 bits are not supported by Java 7 and earlier clients. This value is not used if static Diffie-Hellman parameters are supplied either directly in the certificate file or by using the ssl-dh-param-file parameter.

tune.ssl.ssl-ctx-cache-size <number>

Sets the size of the cache used to store generated certificates to <number> entries. This is a LRU cache. Because generating a SSL certificate dynamically is expensive, they are cached. The default cache size is set to 1000 entries.

tune.vars.global-max-size <size>
tune.vars.reqres-max-size <size>
tune.vars.sess-max-size <size>
tune.vars.txn-max-size <size>

These four tunes helps to manage the allowed amount of memory used by the variables system. "global" limits the memory for all the systems. "sess" limit the memory by session, "txn" limits the memory by transaction and "reqres" limits the memory for each request or response processing. during the accounting, "sess" embbed "txn" and "txn" embed "reqres".

By example, we considers that "tune.vars.sess-max-size" is fixed to 100, "tune.vars.txn-max-size" is fixed to 100, "tune.vars.reqres-max-size" is also fixed to 100. If we create a variable "txn.var" that contains 100 bytes, we cannot create any more variable in the other contexts.

tune.zlib.memlevel <number>

Sets the memLevel parameter in zlib initialization for each session. It defines how much memory should be allocated for the internal compression state. A value of 1 uses minimum memory but is slow and reduces compression ratio, a value of 9 uses maximum memory for optimal speed. Can be a value between 1 and 9. The default value is 8.

tune.zlib.windowsize <number>

Sets the window size (the size of the history buffer) as a parameter of the zlib initialization for each session. Larger values of this parameter result in better compression at the expense of memory usage. Can be a value between 8 and 15. The default value is 15.

3.3. Debugging

debug

Enables debug mode which dumps to stdout all exchanges, and disables forking into background. It is the equivalent of the command-line argument "-d". It should never be used in a production configuration since it may prevent full system startup.

quiet

Do not display any message during startup. It is equivalent to the command-line argument "-q".

3.4. Userlists

It is possible to control access to frontend/backend/listen sections or to http stats by allowing only authenticated and authorized users. To do this, it is required to create at least one userlist and to define users.

userlist <listname>

Creates new userlist with name <listname>. Many independent userlists can be used to store authentication & authorization data for independent customers.

group <groupname> [users <user>,<user>,(...)]

Adds group <groupname> to the current userlist. It is also possible to attach users to this group by using a comma separated list of names proceeded by "users" keyword.

```
user <username> [password|insecure-password <password>]
                [groups <group>,<group>,(...)]
```

Adds user <username> to the current userlist. Both secure (encrypted) and insecure (unencrypted) passwords can be used. Encrypted passwords are evaluated using the crypt(3) function so depending of the system's capabilities, different algorithms are supported. For example modern Glibc based Linux system supports MD5, SHA-256, SHA-512 and of course classic, DES-based method of encrypting passwords.

Example:

```
      userlist L1
        group G1 users tiger,scott
        group G2 users xdb,scott

        user tiger password $6$k6y3o.eP$JlKBx9za9667qe4(...)xHSwRv6J.C0/D7cV91
        user scott insecure-password elgato
        user xdb insecure-password hello
        userlist L2
          group G1
          group G2

          user tiger password $6$k6y3o.eP$JlKBx(...)xHSwRv6J.C0/D7cV91 groups G1
          user scott insecure-password elgato groups G1,G2
          user xdb insecure-password hello groups G2
```

Please note that both lists are functionally identical.

3.5. Peers

It is possible to propagate entries of any data-types in stick-tables between several haproxy instances over TCP connections in a multi-master fashion. Each instance pushes its local updates and insertions to remote peers. The pushed values overwrite remote ones without aggregation. Interrupted exchanges are automatically detected and recovered from the last known point.
In addition, during a soft restart, the old process connects to the new one using such a TCP connection to push all its entries before the new process tries to connect to other peers. That ensures very fast replication during a reload, it typically takes a fraction of a second even for large tables.
Note that Server IDs are used to identify servers remotely, so it is important that configurations look similar or at least that the same IDs are forced on each server on all participants.

peers <peersect>
  Creates a new peer list with name <peersect>. It is an independent section, which is referenced by one or more stick-tables.

disabled
  Disables a peers section. It disables both listening and any synchronization related to this section. This is provided to disable synchronization of stick tables without having to comment out all "peers" references.

enable
  This re-enables a disabled peers section which was previously disabled.

peer <peername> <ip>:<port>
  Defines a peer inside a peers section.
  If <peername> is set to the local peer name (by default hostname, or forced using "-L" command line option), haproxy will listen for incoming remote peer connection on <ip>:<port>. Otherwise, <ip>:<port> defines where to connect to to join the remote peer, and <peername> is used at the protocol level to identify and validate the remote peer on the server side.

  During a soft restart, local peer <ip>:<port> is used by the old instance to connect the new one and initiate a complete replication (teaching process).

  It is strongly recommended to have the exact same peers declaration on all peers and to only rely on the "-L" command line argument to change the local peer name. This makes it easier to maintain coherent configuration files across all peers.

  You may want to reference some environment variables in the address parameter, see section 2.3 about environment variables.

  Example:

```
    peers mypeers
        peer haproxy1 192.168.0.1:1024
        peer haproxy2 192.168.0.2:1024
        peer haproxy3 10.2.0.1:1024

    backend mybackend
```

```
        mode tcp
        balance roundrobin
        stick-table type ip size 20k peers mypeers
        stick on src

        server srv1 192.168.0.30:80
        server srv2 192.168.0.31:80
```

3.6. Mailers

It is possible to send email alerts when the state of servers changes. If configured email alerts are sent to each mailer that is configured in a mailers section. Email is sent to mailers using SMTP.

mailers <mailersect>
  Creates a new mailer list with the name <mailersect>. It is an independent section which is referenced by one or more proxies.

mailer <mailername> <ip>:<port>
  Defines a mailer inside a mailers section.

  Example:

```
    mailers mymailers
        mailer smtp1 192.168.0.1:587
        mailer smtp2 192.168.0.2:587

    backend mybackend
        mode tcp
        balance roundrobin

        email-alert mailers mymailers
        email-alert from test1@horms.org
        email-alert to test2@horms.org

        server srv1 192.168.0.30:80
        server srv2 192.168.0.31:80
```

4. Proxies

Proxy configuration can be located in a set of sections :
 - defaults [<name>]
 - frontend <name>
 - backend <name>
 - listen   <name>

A "defaults" section sets default parameters for all other sections following its declaration. Those default parameters are reset by the next "defaults" section. See below for the list of parameters which can be set in a "defaults" section. The name is optional but its use is encouraged for better readability.

A "frontend" section describes a set of listening sockets accepting client connections.

A "backend" section describes a set of servers to which the proxy will connect to forward incoming connections.

A "listen" section defines a complete proxy with its frontend and backend parts combined in one section. It is generally useful for TCP-only traffic.

All proxy names must be formed from upper and lower case letters, digits, '-' (dash), '_' (underscore) , '.' (dot) and ':' (colon). ACL names are case-sensitive, which means that "www" and "WWW" are two different proxies.

Historically, all proxy names could overlap, it just caused troubles in the logs. Since the introduction of content switching, it is mandatory that two proxies with overlapping capabilities (frontend/backend) have different names. However, it is still permitted that a frontend and a backend share the same name, as this configuration seems to be commonly encountered.

Right now, two major proxy modes are supported : "tcp", also known as layer 4, and "http", also known as layer 7. In layer 4 mode, HAProxy simply forwards bidirectional traffic between two sides. In layer 7 mode, HAProxy analyzes the protocol, and can interact with it by allowing, blocking, switching, adding, modifying, or removing arbitrary contents in requests or responses, based on arbitrary criteria.

In HTTP mode, the processing applied to requests and responses flowing over a connection depends in the combination of the frontend's HTTP options and the backend's. HAProxy supports 5 connection modes :

- KAL : keep alive ("option http-keep-alive") which is the default mode : all requests and responses are processed, and connections remain open but idle between responses and new requests.

- TUN: tunnel ("option http-tunnel") : this was the default mode for versions 1.0 to 1.5-dev21 : only the first request and response are processed, and everything else is forwarded with no analysis at all. This mode should not be used as it creates lots of trouble with logging and HTTP processing.

- PCL: passive close ("option httpclose") : exactly the same as tunnel mode, but with "Connection: close" appended in both directions to try to make both ends close after the first request/response exchange.

- SCL: server close ("option http-server-close") : the server-facing connection is closed after the end of the response is received, but the client-facing connection remains open.

- FCL: forced close ("option forceclose") : the connection is actively closed after the end of the response.

The effective mode that will be applied to a connection passing through a frontend and a backend can be determined by both proxy modes according to the following matrix, but in short, the modes are symmetric, keep-alive is the weakest option and force close is the strongest.

```
                          Backend mode

                    | KAL | TUN | PCL | SCL | FCL
                ----+-----+-----+-----+-----+----
                KAL | KAL | TUN | PCL | SCL | FCL
                ----+-----+-----+-----+-----+----
                TUN | TUN | TUN | PCL | SCL | FCL
 Frontend       ----+-----+-----+-----+-----+----
   mode         PCL | PCL | PCL | PCL | FCL | FCL
                ----+-----+-----+-----+-----+----
                SCL | SCL | SCL | FCL | SCL | FCL
                ----+-----+-----+-----+-----+----
                FCL | FCL | FCL | FCL | FCL | FCL
```

4.1. Proxy keywords matrix

The following list of keywords is supported. Most of them may only be used in a limited set of section types. Some of them are marked as "deprecated" because they are inherited from an old syntax which may be confusing or functionally

limited, and there are new recommended keywords to replace them. Keywords marked with "(*)" can be optionally inverted using the "no" prefix, eg. "no option contstats". This makes sense when the option has been enabled by default and must be disabled for a specific instance. Such options may also be prefixed with "default" in order to restore default settings regardless of what has been specified in a previous "defaults" section.

keyword	defaults	frontend	listen	backend
acl	-	X	X	X
appsession	-	-	-	-
backlog	X	X	X	-
balance	X	-	X	X
bind	-	X	X	-
bind-process	X	X	X	X
block	-	X	X	X
capture cookie	-	X	X	-
capture request header	-	X	X	-
capture response header	-	X	X	-
clitimeout (deprecated)	X	X	X	-
compression	X	X	X	X
contimeout (deprecated)	X	-	X	X
cookie	X	-	X	X
declare capture	-	X	X	-
default-server	X	-	X	X
default_backend	X	X	X	-
description	-	X	X	X
disabled	X	X	X	X
dispatch	-	-	X	X
email-alert from	X	X	X	X
email-alert level	X	X	X	X
email-alert mailers	X	X	X	X
email-alert myhostname	X	X	X	X
email-alert to	X	X	X	X
enabled	X	X	X	X
errorfile	X	X	X	X
errorloc	X	X	X	X
errorloc302	X	X	X	X
-- keyword --------------------------	defaults -	frontend -	listen --	backend -
errorloc303	X	X	X	X
force-persist	-	X	X	X
fullconn	X	-	X	X
grace	X	X	X	X
hash-type	X	-	X	X
http-check disable-on-404	X	-	X	X
http-check expect	-	-	X	X
http-check send-state	X	-	X	X
http-request	-	X	X	X
http-response	-	X	X	X
http-reuse	X	-	X	X
http-send-name-header	-	-	X	X
id	-	X	X	X
ignore-persist	-	X	X	X
load-server-state-from-file	X	-	X	X
log (*)	X	X	X	X
log-format	X	X	X	-
log-format-sd	X	X	X	-
log-tag	X	X	X	X
max-keep-alive-queue	X	-	X	X
maxconn	X	X	X	-
mode	X	X	X	X
monitor fail	-	X	X	-
monitor-net	X	X	X	-
monitor-uri	X	X	X	-

```
option abortonclose                  (*)  X          -         X         X
option accept-invalid-http-request   (*)  X          X         X         -
option accept-invalid-http-response  (*)  X          -         X         X
option allbackups                    (*)  X          -         X         X
option checkcache                    (*)  X          -         X         X
option clitcpka                      (*)  X          X         X         -
option contstats                     (*)  X          X         X         -
option dontlog-normal                (*)  X          X         X         -
option dontlognull                   (*)  X          X         X         -
option forceclose                    (*)  X          X         X         X
-- keyword -------------------------- defaults - frontend - listen -- backend -
option forwardfor                         X          X         X         X
option http-buffer-request           (*)  X          X         X         X
option http-ignore-probes            (*)  X          X         X         -
option http-keep-alive               (*)  X          X         X         X
option http-no-delay                 (*)  X          X         X         X
option http-pretend-keepalive        (*)  X          X         X         X
option http-server-close             (*)  X          X         X         X
option http-tunnel                   (*)  X          X         X         X
option http-use-proxy-header         (*)  X          X         X         -
option httpchk                            X          -         X         X
option httpclose                     (*)  X          X         X         X
option httplog                            X          X         X         X
option http_proxy                    (*)  X          X         X         X
option independent-streams           (*)  X          X         X         X
option ldap-check                         X          -         X         X
option external-check                     X          -         X         X
option log-health-checks             (*)  X          -         X         X
option log-separate-errors           (*)  X          X         X         -
option logasap                       (*)  X          X         X         -
option mysql-check                        X          -         X         X
option nolinger                      (*)  X          X         X         X
option originalto                         X          X         X         X
option persist                       (*)  X          -         X         X
option pgsql-check                        X          -         X         X
option prefer-last-server            (*)  X          -         X         X
option redispatch                    (*)  X          -         X         X
option redis-check                        X          -         X         X
option smtpchk                            X          -         X         X
option socket-stats                  (*)  X          X         X         -
option splice-auto                   (*)  X          X         X         X
option splice-request                (*)  X          X         X         X
option splice-response               (*)  X          X         X         X
option srvtcpka                      (*)  X          -         X         X
option ssl-hello-chk                      X          -         X         X
-- keyword -------------------------- defaults - frontend - listen -- backend -
option tcp-check                          X          -         X         X
option tcp-smart-accept              (*)  X          X         X         -
option tcp-smart-connect             (*)  X          -         X         X
option tcpka                              X          X         X         X
option tcplog                             X          X         X         X
option transparent                   (*)  X          -         X         X
external-check command                    X          -         X         X
external-check path                       X          -         X         X
persist rdp-cookie                        X          -         X         X
rate-limit sessions                       X          X         X         -
redirect                                  -          X         X         X
redisp                      (deprecated)  X          -         X         X
redispatch                  (deprecated)  X          -         X         X
reqadd                                    -          X         X         X
reqallow                                  -          X         X         X
reqdel                                    -          X         X         X
reqdeny                                   -          X         X         X
reqiallow                                 -          X         X         X
reqidel                                   -          X         X         X
```

```
reqideny                                  -          X         X         X
reqipass                                  -          X         X         X
reqirep                                   -          X         X         X
reqitarpit                                -          X         X         X
reqpass                                   -          X         X         X
reqrep                                    -          X         X         X
-- keyword -------------------------- defaults - frontend - listen -- backend -
reqtarpit                                 -          X         X         X
retries                                   X          -         X         X
rspadd                                    -          X         X         X
rspdel                                    -          X         X         X
rspdeny                                   -          X         X         X
rspidel                                   -          X         X         X
rspideny                                  -          X         X         X
rspirep                                   -          X         X         X
rsprep                                    -          X         X         X
server                                    -          -         X         X
server-state-file-name                    X          -         X         X
source                                    X          -         X         X
srvtimeout                  (deprecated)  X          -         X         X
stats admin                               -          X         X         X
stats auth                                X          X         X         X
stats enable                              X          X         X         X
stats hide-version                        X          X         X         X
stats http-request                        -          X         X         X
stats realm                               X          X         X         X
stats refresh                             X          X         X         X
stats scope                               X          X         X         X
stats show-desc                           X          X         X         X
stats show-legends                        X          X         X         X
stats show-node                           X          X         X         X
stats uri                                 X          X         X         X
-- keyword -------------------------- defaults - frontend - listen -- backend -
stick match                               -          -         X         X
stick on                                  -          -         X         X
stick store-request                       -          -         X         X
stick store-response                      -          -         X         X
stick-table                               -          -         X         X
tcp-check connect                         -          -         X         X
tcp-check expect                          -          -         X         X
tcp-check send                            -          -         X         X
tcp-check send-binary                     -          -         X         X
tcp-request connection                    -          X         X         -
tcp-request content                       -          X         X         X
tcp-request inspect-delay                 -          X         X         X
tcp-response content                      -          -         X         X
tcp-response inspect-delay                -          -         X         X
timeout check                             X          -         X         X
timeout client                            X          X         X         -
timeout client-fin                        X          X         X         -
timeout clitimeout          (deprecated)  X          X         X         -
timeout connect                           X          -         X         X
timeout contimeout          (deprecated)  X          -         X         X
timeout http-keep-alive                   X          X         X         X
timeout http-request                      X          X         X         X
timeout queue                             X          -         X         X
timeout server                            X          -         X         X
timeout server-fin                        X          -         X         X
timeout srvtimeout          (deprecated)  X          -         X         X
timeout tarpit                            X          X         X         X
timeout tunnel                            X          -         X         X
transparent                 (deprecated)  X          -         X         X
unique-id-format                          X          X         X         -
unique-id-header                          X          X         X         -
use_backend                               -          X         X         -
```

```
use-server                                -          -         X         X
------------------------------------+----------+----------+---------+---------
 keyword                              defaults   frontend   listen    backend


4.2. Alphabetically sorted keywords reference
---------------------------------------------

This section provides a description of each keyword and its usage.


acl <aclname> <criterion> [flags] [operator] <value> ...
  Declare or complete an access list.
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Example:
        acl invalid_src  src          0.0.0.0/7 224.0.0.0/3
        acl invalid_src  src_port     0:1023
        acl local_dst    hdr(host) -i localhost

  See section 7 about ACL usage.


appsession <cookie> len <length> timeout <holdtime>
           [request-learn] [prefix] [mode <path-parameters|query-string>]
  Define session stickiness on an existing application cookie.
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    no    |   yes  |   yes
  Arguments :
    <cookie>   this is the name of the cookie used by the application and which
               HAProxy will have to learn for each new session.

    <length>   this is the max number of characters that will be memorized and
               checked in each cookie value.

    <holdtime> this is the time after which the cookie will be removed from
               memory if unused. If no unit is specified, this time is in
               milliseconds.

    request-learn
               If this option is specified, then haproxy will be able to learn
               the cookie found in the request in case the server does not
               specify any in response. This is typically what happens with
               PHPSESSID cookies, or when haproxy's session expires before
               the application's session and the correct server is selected.
               It is recommended to specify this option to improve reliability.

    prefix     When this option is specified, haproxy will match on the cookie
               prefix (or URL parameter prefix). The appsession value is the
               data following this prefix.

               Example :
               appsession ASPSESSIONID len 64 timeout 3h prefix

               This will match the cookie ASPSESSIONIDXXXX=XXXXX,
               the appsession value will be XXXX=XXXXX.

    mode       This option allows to change the URL parser mode.
               2 modes are currently supported :
               - path-parameters :
                 The parser looks for the appsession in the path parameters
                 part (each parameter is separated by a semi-colon), which is
                 convenient for JSESSIONID for example.
                 This is the default mode if the option is not set.
               - query-string :
```

```
                 In this mode, the parser will look for the appsession in the
                 query string.

  As of version 1.6, appsessions was removed. It is more flexible and more
  convenient to use stick-tables instead, and stick-tables support multi-master
  replication and data conservation across reloads, which appsessions did not.

  See also : "cookie", "capture cookie", "balance", "stick", "stick-table",
             "ignore-persist", "nbproc" and "bind-process".


backlog <conns>
  Give hints to the system about the approximate listen backlog desired size
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments :
    <conns>   is the number of pending connections. Depending on the operating
              system, it may represent the number of already acknowledged
              connections, of non-acknowledged ones, or both.

  In order to protect against SYN flood attacks, one solution is to increase
  the system's SYN backlog size. Depending on the system, sometimes it is just
  tunable via a system parameter, sometimes it is not adjustable at all, and
  sometimes the system relies on hints given by the application at the time of
  the listen() syscall. By default, HAProxy passes the frontend's maxconn value
  to the listen() syscall. On systems which can make use of this value, it can
  sometimes be useful to be able to specify a different value, hence this
  backlog parameter.

  On Linux 2.4, the parameter is ignored by the system. On Linux 2.6, it is
  used as a hint and the system accepts up to the smallest greater power of
  two, and never more than some limits (usually 32768).

  See also : "maxconn" and the target operating system's tuning guide.


balance <algorithm> [ <arguments> ]
balance url_param <param> [check_post]
  Define the load balancing algorithm to be used in a backend.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <algorithm> is the algorithm used to select a server when doing load
                balancing. This only applies when no persistence information
                is available, or when a connection is redispatched to another
                server. <algorithm> may be one of the following :

      roundrobin  Each server is used in turns, according to their weights.
                  This is the smoothest and fairest algorithm when the server's
                  processing time remains equally distributed. This algorithm
                  is dynamic, which means that server weights may be adjusted
                  on the fly for slow starts for instance. It is limited by
                  design to 4095 active servers per backend. Note that in some
                  large farms, when a server becomes up after having been down
                  for a very short time, it may sometimes take a few hundreds
                  requests for it to be re-integrated into the farm and start
                  receiving traffic. This is normal, though very rare. It is
                  indicated here in case you would have the chance to observe
                  it, so that you don't worry.

      static-rr   Each server is used in turns, according to their weights.
                  This algorithm is as similar to roundrobin except that it is
                  static, which means that changing a server's weight on the
                  fly will have no effect. On the other hand, it has no design
                  limitation on the number of servers, and when a server goes
                  up, it is always immediately reintroduced into the farm, once
                  the full map is recomputed. It also uses slightly less CPU to
                  run (around -1%).

      leastconn   The server with the lowest number of connections receives the
                  connection. Round-robin is performed within groups of servers
                  of the same load to ensure that all servers will be used. Use
                  of this algorithm is recommended where very long sessions are
                  expected, such as LDAP, SQL, TSE, etc... but is not very well
                  suited for protocols using short sessions such as HTTP. This
                  algorithm is dynamic, which means that server weights may be
                  adjusted on the fly for slow starts for instance.

      first       The first server with available connection slots receives the
                  connection. The servers are chosen from the lowest numeric
                  identifier to the highest (see server parameter "id"), which
                  defaults to the server's position in the farm. Once a server
                  reaches its maxconn value, the next server is used. It does
                  not make sense to use this algorithm without setting maxconn.
                  The purpose of this algorithm is to always use the smallest
                  number of servers so that extra servers can be powered off
                  during non-intensive hours. This algorithm ignores the server
                  weight, and brings more benefit to long session such as RDP
                  or IMAP than HTTP, though it can be useful there too. In
                  order to use this algorithm efficiently, it is recommended
                  that a cloud controller regularly checks server usage to turn
                  them off when unused, and regularly checks backend queue to
                  turn new servers on when the queue inflates. Alternatively,
                  using "http-check send-state" may inform servers on the load.

      source      The source IP address is hashed and divided by the total
                  weight of the running servers to designate which server will
                  receive the request. This ensures that the same client IP
                  address will always reach the same server as long as no
                  server goes down or up. If the hash result changes due to the
                  number of running servers changing, many clients will be
                  directed to a different server. This algorithm is generally
                  used in TCP mode where no cookie may be inserted. It may also
                  be used on the Internet to provide a best-effort stickiness
                  to clients which refuse session cookies. This algorithm is
                  static by default, which means that changing a server's
                  weight on the fly will have no effect, but this can be
                  changed using "hash-type".

      uri         This algorithm hashes either the left part of the URI (before
                  the question mark) or the whole URI (if the "whole" parameter
                  is present) and divides the hash value by the total weight of
                  the running servers. The result designates which server will
                  receive the request. This ensures that the same URI will
                  always be directed to the same server as long as no server
                  goes up or down. This is used with proxy caches and
                  anti-virus proxies in order to maximize the cache hit rate.
                  Note that this algorithm may only be used in an HTTP backend.
                  This algorithm is static by default, which means that
                  changing a server's weight on the fly will have no effect,
                  but this can be changed using "hash-type".

                  This algorithm supports two optional parameters "len" and
                  "depth", both followed by a positive integer number. These
                  options may be helpful when it is needed to balance servers
                  based on the beginning of the URI only. The "len" parameter
                  indicates that the algorithm should only consider that many
                  characters at the beginning of the URI to compute the hash.
                  Note that having "len" set to 1 rarely makes sense since most
                  URIs start with a leading "/".
```

```

                  The "depth" parameter indicates the maximum directory depth
                  to be used to compute the hash. One level is counted for each
                  slash in the request. If both parameters are specified, the
                  evaluation stops when either is reached.

      url_param   The URL parameter specified in argument will be looked up in
                  the query string of each HTTP GET request.

                  If the modifier "check_post" is used, then an HTTP POST
                  request entity will be searched for the parameter argument,
                  when it is not found in a query string after a question mark
                  ('?') in the URL. The message body will only start to be
                  analyzed once either the advertised amount of data has been
                  received or the request buffer is full. In the unlikely event
                  that chunked encoding is used, only the first chunk is
                  scanned. Parameter values separated by a chunk boundary, may
                  be randomly balanced if at all. This keyword used to support
                  an optional <max_wait> parameter which is now ignored.

                  If the parameter is found followed by an equal sign ('=') and
                  a value, then the value is hashed and divided by the total
                  weight of the running servers. The result designates which
                  server will receive the request.

                  This is used to track user identifiers in requests and ensure
                  that a same user ID will always be sent to the same server as
                  long as no server goes up or down. If no value is found or if
                  the parameter is not found, then a round robin algorithm is
                  applied. Note that this algorithm may only be used in an HTTP
                  backend. This algorithm is static by default, which means
                  that changing a server's weight on the fly will have no
                  effect, but this can be changed using "hash-type".

      hdr(<name>) The HTTP header <name> will be looked up in each HTTP
                  request. Just as with the equivalent ACL 'hdr()' function,
                  the header name in parenthesis is not case sensitive. If the
                  header is absent or if it does not contain any value, the
                  roundrobin algorithm is applied instead.

                  An optional 'use_domain_only' parameter is available, for
                  reducing the hash algorithm to the main domain part with some
                  specific headers such as 'Host'. For instance, in the Host
                  value "haproxy.1wt.eu", only "1wt" will be considered.

                  This algorithm is static by default, which means that
                  changing a server's weight on the fly will have no effect,
                  but this can be changed using "hash-type".

      rdp-cookie
      rdp-cookie(<name>)
                  The RDP cookie <name> (or "mstshash" if omitted) will be
                  looked up and hashed for each incoming TCP request. Just as
                  with the equivalent ACL 'req_rdp_cookie()' function, the name
                  is not case-sensitive. This mechanism is useful as a degraded
                  persistence mode, as it makes it possible to always send the
                  same user (or the same session ID) to the same server. If the
                  cookie is not found, the normal roundrobin algorithm is
                  used instead.

                  Note that for this to work, the frontend must ensure that an
                  RDP cookie is already present in the request buffer. For this
                  you must use 'tcp-request content accept' rule combined with
                  a 'req_rdp_cookie_cnt' ACL.
                  This algorithm is static by default, which means that
                  changing a server's weight on the fly will have no effect,
                  but this can be changed using "hash-type".

                  See also the rdp_cookie pattern fetch function.

    <arguments> is an optional list of arguments which may be needed by some
                algorithms. Right now, only "url_param" and "uri" support an
                optional argument.

  The load balancing algorithm of a backend is set to roundrobin when no other
  algorithm, mode nor option have been set. The algorithm may only be set once
  for each backend.

  Examples :
        balance roundrobin
        balance url_param userid
        balance url_param session_id check_post 64
        balance hdr(User-Agent)
        balance hdr(host)
        balance hdr(Host) use_domain_only

  Note: the following caveats and limitations on using the "check_post"
  extension with "url_param" must be considered :

    - all POST requests are eligible for consideration, because there is no way
      to determine if the parameters will be found in the body or entity which
      may contain binary data. Therefore another method may be required to
      restrict consideration of POST requests that have no URL parameters in
      the body. (see acl reqideny http_end)

    - using a <max_wait> value larger than the request buffer size does not
      make sense and is useless. The buffer size is set at build time, and
      defaults to 16 kB.

    - Content-Encoding is not supported, the parameter search will probably
      fail; and load balancing will fall back to Round Robin.

    - Expect: 100-continue is not supported, load balancing will fall back to
      Round Robin.

    - Transfer-Encoding (RFC2616 3.6.1) is only supported in the first chunk.
      If the entire parameter value is not present in the first chunk, the
      selection of server is undefined (actually, defined by how little
      actually appeared in the first chunk).

    - This feature does not support generation of a 100, 411 or 501 response.

    - In some cases, requesting "check_post" MAY attempt to scan the entire
      contents of a message body. Scanning normally terminates when linear
      white space or control characters are found, indicating the end of what
      might be a URL parameter list. This is probably not a concern with SGML
      type message bodies.

  See also : "dispatch", "cookie", "transparent", "hash-type" and "http_proxy".


bind [<address>]:<port_range> [, ...] [param*]
bind /<path> [, ...] [param*]
  Define one or several listening addresses and/or ports in a frontend.
  May be used in sections :   defaults | frontend | listen | backend
                                  no   |    yes   |   yes  |   no
  Arguments :
    <address>     is optional and can be a host name, an IPv4 address, an IPv6
                  address, or '*'. It designates the address the frontend will
```

```
                  listen on. If unset, all IPv4 addresses of the system will be
                  listened on. The same will apply for '*' or the system's
                  special address "0.0.0.0". The IPv6 equivalent is '::'.
                  Optionally, an address family prefix may be used before the
                  address to force the family regardless of the address format,
                  which can be useful to specify a path to a unix socket with
                  no slash ('/'). Currently supported prefixes are :
                    - 'ipv4@'  -> address is always IPv4
                    - 'ipv6@'  -> address is always IPv6
                    - 'unix@'  -> address is a path to a local unix socket
                    - 'abns@'  -> address is in abstract namespace (Linux only).
                      Note: since abstract sockets are not "rebindable", they
                            do not cope well with multi-process mode during
                            soft-restart, so it is better to avoid them if
                            nbproc is greater than 1. The effect is that if the
                            new process fails to start, only one of the old ones
                            will be able to rebind to the socket.
                    - 'fd@<n>' -> use file descriptor <n> inherited from the
                      parent. The fd must be bound and may or may not already
                      be listening.
                  You may want to reference some environment variables in the
                  address parameter, see section 2.3 about environment
                  variables.

    <port_range>  is either a unique TCP port, or a port range for which the
                  proxy will accept connections for the IP address specified
                  above. The port is mandatory for TCP listeners. Note that in
                  the case of an IPv6 address, the port is always the number
                  after the last colon (':'). A range can either be :
                   - a numerical port (ex: '80')
                   - a dash-delimited ports range explicitly stating the lower
                     and upper bounds (ex: '2000-2100') which are included in
                     the range.

                  Particular care must be taken against port ranges, because
                  every <address:port> couple consumes one socket (= a file
                  descriptor), so it's easy to consume lots of descriptors
                  with a simple range, and to run out of sockets. Also, each
                  <address:port> couple must be used only once among all
                  instances running on a same system. Please note that binding
                  to ports lower than 1024 generally require particular
                  privileges to start the program, which are independent of
                  the 'uid' parameter.

    <path>        is a UNIX socket path beginning with a slash ('/'). This is
                  alternative to the TCP listening port. Haproxy will then
                  receive UNIX connections on the socket located at this place.
                  The path must begin with a slash and by default is absolute.
                  It can be relative to the prefix defined by "unix-bind" in
                  the global section. Note that the total length of the prefix
                  followed by the socket path cannot exceed some system limits
                  for UNIX sockets, which commonly are set to 107 characters.

    <param*>      is a list of parameters common to all sockets declared on the
                  same line. These numerous parameters depend on OS and build
                  options and have a complete section dedicated to them. Please
                  refer to section 5 to for more details.

  It is possible to specify a list of address:port combinations delimited by
  commas. The frontend will then listen on all of these addresses. There is no
  fixed limit to the number of addresses and ports which can be listened on in
  a frontend, as well as there is no limit to the number of "bind" statements
  in a frontend.

  Example :
        listen http_proxy
            bind :80,:443
            bind 10.0.0.1:10080,10.0.0.1:10443
            bind /var/run/ssl-frontend.sock user root mode 600 accept-proxy

        listen http_https_proxy
            bind :80
            bind :443 ssl crt /etc/haproxy/site.pem

        listen http_https_proxy_explicit
            bind ipv6@:80
            bind ipv4@public_ssl:443 ssl crt /etc/haproxy/site.pem
            bind unix@ssl-frontend.sock user root mode 600 accept-proxy

        listen external_bind_app1
            bind "fd@${FD_APP1}"

  Note: regarding Linux's abstract namespace sockets, HAProxy uses the whole
        sun_path length is used for the address length. Some other programs
        such as socat use the string length only by default. Pass the option
        ",unix-tightsocklen=0" to any abstract socket definition in socat to
        make it compatible with HAProxy's.

  See also : "source", "option forwardfor", "unix-bind" and the PROXY protocol
             documentation, and section 5 about bind options.


bind-process [ all | odd | even | <number 1-64>[-<number 1-64>] ] ...
  Limit visibility of an instance to a certain set of processes numbers.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    all           All process will see this instance. This is the default. It
                  may be used to override a default value.

    odd           This instance will be enabled on processes 1,3,5,...63. This
                  option may be combined with other numbers.

    even          This instance will be enabled on processes 2,4,6,...64. This
                  option may be combined with other numbers. Do not use it
                  with less than 2 processes otherwise some instances might be
                  missing from all processes.

    number        The instance will be enabled on this process number or range,
                  whose values must all be between 1 and 32 or 64 depending on
                  the machine's word size. If a proxy is bound to process
                  numbers greater than the configured global.nbproc, it will
                  either be forced to process #1 if a single process was
                  specified, or to all processes otherwise.

  This keyword limits binding of certain instances to certain processes. This
  is useful in order not to have too many processes listening to the same
  ports. For instance, on a dual-core machine, it might make sense to set
  'nbproc 2' in the global section, then distributes the listeners among 'odd'
  and 'even' instances.

  At the moment, it is not possible to reference more than 32 or 64 processes
  using this keyword, but this should be more than enough for most setups.
  Please note that 'all' really means all processes regardless of the machine's
  word size, and is not limited to the first 32 or 64.

  Each "bind" line may further be limited to a subset of the proxy's processes,
  please consult the "process" bind keyword in section 5.1.

  When a frontend has no explicit "bind-process" line, it tries to bind to all
```

```
  the processes referenced by its "bind" lines. That means that frontends can
  easily adapt to their listeners' processes.

  If some backends are referenced by frontends bound to other processes, the
  backend automatically inherits the frontend's processes.

  Example :
        listen app_ip1
            bind 10.0.0.1:80
            bind-process odd

        listen app_ip2
            bind 10.0.0.2:80
            bind-process even

        listen management
            bind 10.0.0.3:80
            bind-process 1 2 3 4

        listen management
            bind 10.0.0.4:80
            bind-process 1-4

  See also : "nbproc" in global section, and "process" in section 5.1.


block { if | unless } <condition>
  Block a layer 7 request if/unless a condition is matched
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes

  The HTTP request will be blocked very early in the layer 7 processing
  if/unless <condition> is matched. A 403 error will be returned if the request
  is blocked. The condition has to reference ACLs (see section 7). This is
  typically used to deny access to certain sensitive resources if some
  conditions are met or not met. There is no fixed limit to the number of
  "block" statements per instance.

  Example:
        acl invalid_src  src          0.0.0.0/7 224.0.0.0/3
        acl invalid_src  src_port     0:1023
        acl local_dst    hdr(host) -i localhost
        block if invalid_src || local_dst

  See section 7 about ACL usage.


capture cookie <name> len <length>
  Capture and log a cookie in the request and in the response.
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   no
  Arguments :
    <name>    is the beginning of the name of the cookie to capture. In order
              to match the exact name, simply suffix the name with an equal
              sign ('='). The full name will appear in the logs, which is
              useful with application servers which adjust both the cookie name
              and value (eg: ASPSESSIONXXXXX).

    <length>  is the maximum number of characters to report in the logs, which
              include the cookie name, the equal sign and the value, all in the
              standard "name=value" form. The string will be truncated on the
              right if it exceeds <length>.

  Only the first cookie is captured. Both the "cookie" request headers and the
  "set-cookie" response headers are monitored. This is particularly useful to
  check for application bugs causing session crossing or stealing between
  users, because generally the user's cookies can only change on a login page.

  When the cookie was not presented by the client, the associated log column
  will report "-". When a request does not cause a cookie to be assigned by the
  server, a "-" is reported in the response column.

  The capture is performed in the frontend only because it is necessary that
  the log format does not change for a given frontend depending on the
  backends. This may change in the future. Note that there can be only one
  "capture cookie" statement in a frontend. The maximum capture length is set
  by the global "tune.http.cookielen" setting and defaults to 63 characters. It
  is not possible to specify a capture in a "defaults" section.

  Example:
        capture cookie ASPSESSION len 32

  See also : "capture request header", "capture response header" as well as
            section 8 about logging.


capture request header <name> len <length>
  Capture and log the last occurrence of the specified request header.
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   no
  Arguments :
    <name>    is the name of the header to capture. The header names are not
              case-sensitive, but it is a common practice to write them as they
              appear in the requests, with the first letter of each word in
              upper case. The header name will not appear in the logs, only the
              value is reported, but the position in the logs is respected.

    <length>  is the maximum number of characters to extract from the value and
              report in the logs. The string will be truncated on the right if
              it exceeds <length>.

  The complete value of the last occurrence of the header is captured. The
  value will be added to the logs between braces ('{}'). If multiple headers
  are captured, they will be delimited by a vertical bar ('|') and will appear
  in the same order they were declared in the configuration. Non-existent
  headers will be logged just as an empty string. Common uses for request
  header captures include the "Host" field in virtual hosting environments, the
  "Content-length" when uploads are supported, "User-agent" to quickly
  differentiate between real users and robots, and "X-Forwarded-For" in proxied
  environments to find where the request came from.

  Note that when capturing headers such as "User-agent", some spaces may be
  logged, making the log analysis more difficult. Thus be careful about what
  you log if you know your log parser is not smart enough to rely on the
  braces.

  There is no limit to the number of captured request headers nor to their
  length, though it is wise to keep them low to limit memory usage per session.
  In order to keep log format consistent for a same frontend, header captures
  can only be declared in a frontend. It is not possible to specify a capture
  in a "defaults" section.

  Example:
        capture request header Host len 15
        capture request header X-Forwarded-For len 15
        capture request header Referer len 15

  See also : "capture cookie", "capture response header" as well as section 8
             about logging.

```

```

capture response header <name> len <length>
  Capture and log the last occurrence of the specified response header.
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   no
  Arguments :
    <name>    is the name of the header to capture. The header names are not
              case-sensitive, but it is a common practice to write them as they
              appear in the response, with the first letter of each word in
              upper case. The header name will not appear in the logs, only the
              value is reported, but the position in the logs is respected.

    <length>  is the maximum number of characters to extract from the value and
              report in the logs. The string will be truncated on the right if
              it exceeds <length>.

  The complete value of the last occurrence of the header is captured. The
  result will be added to the logs between braces ('{}') after the captured
  request headers. If multiple headers are captured, they will be delimited by
  a vertical bar ('|') and will appear in the same order they were declared in
  the configuration. Non-existent headers will be logged just as an empty
  string. Common uses for response header captures include the "Content-length"
  header which indicates how many bytes are expected to be returned, the
  "Location" header to track redirections.

  There is no limit to the number of captured response headers nor to their
  length, though it is wise to keep them low to limit memory usage per session.
  In order to keep log format consistent for a same frontend, header captures
  can only be declared in a frontend. It is not possible to specify a capture
  in a "defaults" section.

  Example:
        capture response header Content-length len 9
        capture response header Location len 15

  See also : "capture cookie", "capture request header" as well as section 8
             about logging.


clitimeout <timeout> (deprecated)
  Set the maximum inactivity time on the client side.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments :
    <timeout> is the timeout value is specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  The inactivity timeout applies when the client is expected to acknowledge or
  send data. In HTTP mode, this timeout is particularly important to consider
  during the first phase, when the client sends the request, and during the
  response while it is reading data sent by the server. The value is specified
  in milliseconds by default, but can be in any other unit if the number is
  suffixed by the unit, as specified at the top of this document. In TCP mode
  (and to a lesser extent, in HTTP mode), it is highly recommended that the
  client timeout remains equal to the server timeout in order to avoid complex
  situations to debug. It is a good practice to cover one or several TCP packet
  losses by specifying timeouts that are slightly above multiples of 3 seconds
  (eg: 4 or 5 seconds).

  This parameter is specific to frontends, but can be specified once for all in
  "defaults" sections. This is in fact one of the easiest solutions not to
  forget about it. An unspecified timeout results in an infinite timeout, which
  is not recommended. Such a usage is accepted and works but reports a warning
  during startup because it may results in accumulation of expired sessions in
  the system if the system's timeouts are not configured either.

  This parameter is provided for compatibility but is currently deprecated.
  Please use "timeout client" instead.

  See also : "timeout client", "timeout http-request", "timeout server", and
             "srvtimeout".

compression algo <algorithm> ...
compression type <mime type> ...
compression offload
  Enable HTTP compression.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    algo     is followed by the list of supported compression algorithms.
    type     is followed by the list of MIME types that will be compressed.
    offload  makes haproxy work as a compression offloader only (see notes).

  The currently supported algorithms are :
    identity     this is mostly for debugging, and it was useful for developing
                 the compression feature. Identity does not apply any change on
                 data.

    gzip         applies gzip compression. This setting is only available when
                 support for zlib or libslz was built in.

    deflate      same as "gzip", but with deflate algorithm and zlib format.
                 Note that this algorithm has ambiguous support on many
                 browsers and no support at all from recent ones. It is
                 strongly recommended not to use it for anything else than
                 experimentation. This setting is only available when support
                 for zlib or libslz was built in.

    raw-deflate  same as "deflate" without the zlib wrapper, and used as an
                 alternative when the browser wants "deflate". All major
                 browsers understand it and despite violating the standards,
                 it is known to work better than "deflate", at least on MSIE
                 and some versions of Safari. Do not use it in conjunction
                 with "deflate", use either one or the other since both react
                 to the same Accept-Encoding token. This setting is only
                 available when support for zlib or libslz was built in.

  Compression will be activated depending on the Accept-Encoding request
  header. With identity, it does not take care of that header.
  If backend servers support HTTP compression, these directives
  will be no-op: haproxy will see the compressed response and will not
  compress again. If backend servers do not support HTTP compression and
  there is Accept-Encoding header in request, haproxy will compress the
  matching response.

  The "offload" setting makes haproxy remove the Accept-Encoding header to
  prevent backend servers from compressing responses. It is strongly
  recommended not to do this because this means that all the compression work
  will be done on the single point where haproxy is located. However in some
  deployment scenarios, haproxy may be installed in front of a buggy gateway
  with broken HTTP compression implementation which can't be turned off.
  In that case haproxy can be used to prevent that gateway from emitting
  invalid payloads. In this case, simply removing the header in the
  configuration does not work because it applies before the header is parsed,
  so that prevents haproxy from compressing. The "offload" setting should
  then be used for such scenarios. Note: for now, the "offload" setting is
  ignored when set in a defaults section.

  Compression is disabled when:
```

```
    * the request does not advertise a supported compression algorithm in the
      "Accept-Encoding" header
    * the response message is not HTTP/1.1
    * HTTP status code is not 200
    * response header "Transfer-Encoding" contains "chunked" (Temporary
      Workaround)
    * response contain neither a "Content-Length" header nor a
      "Transfer-Encoding" whose last value is "chunked"
    * response contains a "Content-Type" header whose first value starts with
      "multipart"
    * the response contains the "no-transform" value in the "Cache-control"
      header
    * User-Agent matches "Mozilla/4" unless it is MSIE 6 with XP SP2, or MSIE 7
      and later
    * The response contains a "Content-Encoding" header, indicating that the
      response is already compressed (see compression offload)

  Note: The compression does not rewrite Etag headers, and does not emit the
        Warning header.

  Examples :
        compression algo gzip
        compression type text/html text/plain

contimeout <timeout> (deprecated)
  Set the maximum time to wait for a connection attempt to a server to succeed.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <timeout> is the timeout value is specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  If the server is located on the same LAN as haproxy, the connection should be
  immediate (less than a few milliseconds). Anyway, it is a good practice to
  cover one or several TCP packet losses by specifying timeouts that are
  slightly above multiples of 3 seconds (eg: 4 or 5 seconds). By default, the
  connect timeout also presets the queue timeout to the same value if this one
  has not been specified. Historically, the contimeout was also used to set the
  tarpit timeout in a listen section, which is not possible in a pure frontend.

  This parameter is specific to backends, but can be specified once for all in
  "defaults" sections. This is in fact one of the easiest solutions not to
  forget about it. An unspecified timeout results in an infinite timeout, which
  is not recommended. Such a usage is accepted and works but reports a warning
  during startup because it may results in accumulation of failed sessions in
  the system if the system's timeouts are not configured either.

  This parameter is provided for backwards compatibility but is currently
  deprecated. Please use "timeout connect", "timeout queue" or "timeout tarpit"
  instead.

  See also : "timeout connect", "timeout queue", "timeout tarpit",
             "timeout server", "contimeout".


cookie <name> [ rewrite | insert | prefix ] [ indirect ] [ nocache ]
              [ postonly ] [ preserve ] [ httponly ] [ secure ]
              [ domain <domain> ]* [ maxidle <idle> ] [ maxlife <life> ]
  Enable cookie-based persistence in a backend.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <name>    is the name of the cookie which will be monitored, modified or
              inserted in order to bring persistence. This cookie is sent to
              the client via a "Set-Cookie" header in the response, and is
              brought back by the client in a "Cookie" header in all requests.
              Special care should be taken to choose a name which does not
              conflict with any likely application cookie. Also, if the same
              backends are subject to be used by the same clients (eg:
              HTTP/HTTPS), care should be taken to use different cookie names
              between all backends if persistence between them is not desired.

    rewrite   This keyword indicates that the cookie will be provided by the
              server and that haproxy will have to modify its value to set the
              server's identifier in it. This mode is handy when the management
              of complex combinations of "Set-cookie" and "Cache-control"
              headers is left to the application. The application can then
              decide whether or not it is appropriate to emit a persistence
              cookie. Since all responses should be monitored, this mode only
              works in HTTP close mode. Unless the application behaviour is
              very complex and/or broken, it is advised not to start with this
              mode for new deployments. This keyword is incompatible with
              "insert" and "prefix".

    insert    This keyword indicates that the persistence cookie will have to
              be inserted by haproxy in server responses if the client did not

              already have a cookie that would have permitted it to access this
              server. When used without the "preserve" option, if the server
              emits a cookie with the same name, it will be remove before
              processing.  For this reason, this mode can be used to upgrade
              existing configurations running in the "rewrite" mode. The cookie
              will only be a session cookie and will not be stored on the
              client's disk. By default, unless the "indirect" option is added,
              the server will see the cookies emitted by the client. Due to
              caching effects, it is generally wise to add the "nocache" or
              "postonly" keywords (see below). The "insert" keyword is not
              compatible with "rewrite" and "prefix".

    prefix    This keyword indicates that instead of relying on a dedicated
              cookie for the persistence, an existing one will be completed.
              This may be needed in some specific environments where the client
              does not support more than one single cookie and the application
              already needs it. In this case, whenever the server sets a cookie
              named <name>, it will be prefixed with the server's identifier
              and a delimiter. The prefix will be removed from all client
              requests so that the server still finds the cookie it emitted.
              Since all requests and responses are subject to being modified,
              this mode requires the HTTP close mode. The "prefix" keyword is
              not compatible with "rewrite" and "insert". Note: it is highly
              recommended not to use "indirect" with "prefix", otherwise server
              cookie updates would not be sent to clients.

    indirect  When this option is specified, no cookie will be emitted to a
              client which already has a valid one for the server which has
              processed the request. If the server sets such a cookie itself,
              it will be removed, unless the "preserve" option is also set. In
              "insert" mode, this will additionally remove cookies from the
              requests transmitted to the server, making the persistence
              mechanism totally transparent from an application point of view.
              Note: it is highly recommended not to use "indirect" with
              "prefix", otherwise server cookie updates would not be sent to
              clients.

    nocache   This option is recommended in conjunction with the insert mode
              when there is a cache between the client and HAProxy, as it
              ensures that a cacheable response will be tagged non-cacheable if
              a cookie needs to be inserted. This is important because if all
              persistence cookies are added on a cacheable home page for
```

```
              instance, then all customers will then fetch the page from an
              outer cache and will all share the same persistence cookie,
              leading to one server receiving much more traffic than others.
              See also the "insert" and "postonly" options.

    postonly  This option ensures that cookie insertion will only be performed
              on responses to POST requests. It is an alternative to the
              "nocache" option, because POST responses are not cacheable, so
              this ensures that the persistence cookie will never get cached.
              Since most sites do not need any sort of persistence before the
              first POST which generally is a login request, this is a very
              efficient method to optimize caching without risking to find a
              persistence cookie in the cache.
              See also the "insert" and "nocache" options.

    preserve  This option may only be used with "insert" and/or "indirect". It
              allows the server to emit the persistence cookie itself. In this
              case, if a cookie is found in the response, haproxy will leave it
              untouched. This is useful in order to end persistence after a
              logout request for instance. For this, the server just has to
              emit a cookie with an invalid value (eg: empty) or with a date in
              the past. By combining this mechanism with the "disable-on-404"
              check option, it is possible to perform a completely graceful
              shutdown because users will definitely leave the server after
              they logout.

    httponly  This option tells haproxy to add an "HttpOnly" cookie attribute
              when a cookie is inserted. This attribute is used so that a
              user agent doesn't share the cookie with non-HTTP components.
              Please check RFC6265 for more information on this attribute.

    secure    This option tells haproxy to add a "Secure" cookie attribute when
              a cookie is inserted. This attribute is used so that a user agent
              never emits this cookie over non-secure channels, which means
              that a cookie learned with this flag will be presented only over
              SSL/TLS connections. Please check RFC6265 for more information on
              this attribute.

    domain    This option allows to specify the domain at which a cookie is
              inserted. It requires exactly one parameter: a valid domain
              name. If the domain begins with a dot, the browser is allowed to
              use it for any host ending with that name. It is also possible to
              specify several domain names by invoking this option multiple
              times. Some browsers might have small limits on the number of
              domains, so be careful when doing that. For the record, sending
              10 domains to MSIE 6 or Firefox 2 works as expected.

    maxidle   This option allows inserted cookies to be ignored after some idle
              time. It only works with insert-mode cookies. When a cookie is
              sent to the client, the date this cookie was emitted is sent too.
              Upon further presentations of this cookie, if the date is older
              than the delay indicated by the parameter (in seconds), it will
              be ignored. Otherwise, it will be refreshed if needed when the
              response is sent to the client. This is particularly useful to
              prevent users who never close their browsers from remaining for
              too long on the same server (eg: after a farm size change). When
              this option is set and a cookie has no date, it is always
              accepted, but gets refreshed in the response. This maintains the
              ability for admins to access their sites. Cookies that have a
              date in the future further than 24 hours are ignored. Doing so
              lets admins fix timezone issues without risking kicking users off
              the site.

    maxlife   This option allows inserted cookies to be ignored after some life
              time, whether they're in use or not. It only works with insert
              mode cookies. When a cookie is first sent to the client, the date
              this cookie was emitted is sent too. Upon further presentations
              of this cookie, if the date is older than the delay indicated by
              the parameter (in seconds), it will be ignored. If the cookie in
              the request has no date, it is accepted and a date will be set.
              Cookies that have a date in the future further than 24 hours are
              ignored. Doing so lets admins fix timezone issues without risking
              kicking users off the site. Contrary to maxidle, this value is
              not refreshed, only the first visit date counts. Both maxidle and
              maxlife may be used at the time. This is particularly useful to
              prevent users who never close their browsers from remaining for
              too long on the same server (eg: after a farm size change). This
              is stronger than the maxidle method in that it forces a
              redispatch after some absolute delay.

  There can be only one persistence cookie per HTTP backend, and it can be
  declared in a defaults section. The value of the cookie will be the value
  indicated after the "cookie" keyword in a "server" statement. If no cookie
  is declared for a given server, the cookie is not set.

  Examples :
        cookie JSESSIONID prefix
        cookie SRV insert indirect nocache
        cookie SRV insert postonly indirect
        cookie SRV insert indirect nocache maxidle 30m maxlife 8h

  See also : "balance source", "capture cookie", "server" and "ignore-persist".


declare capture [ request | response ] len <length>
  Declares a capture slot.
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   no
  Arguments:
    <length> is the length allowed for the capture.

  This declaration is only available in the frontend or listen section, but the
  reserved slot can be used in the backends. The "request" keyword allocates a
  capture slot for use in the request, and "response" allocates a capture slot
  for use in the response.

  See also: "capture-req", "capture-res" (sample converters),
            "capture.req.hdr", "capture.res.hdr" (sample fetches),
            "http-request capture" and "http-response capture".


default-server [param*]
  Change default options for a server in a backend
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments:
    <param*>  is a list of parameters for this server. The "default-server"
              keyword accepts an important number of options and has a complete
              section dedicated to it. Please refer to section 5 for more
              details.

  Example :
        default-server inter 1000 weight 13

  See also: "server" and section 5 about server options


default_backend <backend>
  Specify the backend to use when no "use_backend" rule has been matched.
  May be used in sections :   defaults | frontend | listen | backend
```

```
                                 yes   |    yes   |   yes  |   no
  Arguments :
    <backend> is the name of the backend to use.

  When doing content-switching between frontend and backends using the
  "use_backend" keyword, it is often useful to indicate which backend will be
  used when no rule has matched. It generally is the dynamic backend which
  will catch all undetermined requests.

  Example :

        use_backend     dynamic  if  url_dyn
        use_backend     static   if  url_css url_img extension_img
        default_backend dynamic

  See also : "use_backend"


description <string>
  Describe a listen, frontend or backend.
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments : string

  Allows to add a sentence to describe the related object in the HAProxy HTML
  stats page. The description will be printed on the right of the object name
  it describes.
  No need to backslash spaces in the <string> arguments.


disabled
  Disable a proxy, frontend or backend.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  The "disabled" keyword is used to disable an instance, mainly in order to
  liberate a listening port or to temporarily disable a service. The instance
  will still be created and its configuration will be checked, but it will be
  created in the "stopped" state and will appear as such in the statistics. It
  will not receive any traffic nor will it send any health-checks or logs. It
  is possible to disable many instances at once by adding the "disabled"
  keyword in a "defaults" section.

  See also : "enabled"


dispatch <address>:<port>
  Set a default server address
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    no    |   yes  |   yes
  Arguments :

    <address> is the IPv4 address of the default server. Alternatively, a
              resolvable hostname is supported, but this name will be resolved
              during start-up.

    <ports>   is a mandatory port specification. All connections will be sent
              to this port, and it is not permitted to use port offsets as is
              possible with normal servers.

  The "dispatch" keyword designates a default server for use when no other
  server can take the connection. In the past it was used to forward non
  persistent connections to an auxiliary load balancer. Due to its simple
  syntax, it has also been used for simple TCP relays. It is recommended not to
  use it for more clarity, and to use the "server" directive instead.

  See also : "server"


enabled
  Enable a proxy, frontend or backend.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  The "enabled" keyword is used to explicitly enable an instance, when the
  defaults has been set to "disabled". This is very rarely used.

  See also : "disabled"


errorfile <code> <file>
  Return a file contents instead of errors generated by HAProxy
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <code>    is the HTTP status code. Currently, HAProxy is capable of
              generating codes 200, 400, 403, 405, 408, 429, 500, 502, 503, and
              504.

    <file>    designates a file containing the full HTTP response. It is
              recommended to follow the common practice of appending ".http" to
              the filename so that people do not confuse the response with HTML
              error pages, and to use absolute paths, since files are read
              before any chroot is performed.

  It is important to understand that this keyword is not meant to rewrite
  errors returned by the server, but errors detected and returned by HAProxy.
  This is why the list of supported errors is limited to a small set.

  Code 200 is emitted in response to requests matching a "monitor-uri" rule.

  The files are returned verbatim on the TCP socket. This allows any trick such
  as redirections to another URL or site, as well as tricks to clean cookies,
  force enable or disable caching, etc... The package provides default error
  files returning the same contents as default errors.

  The files should not exceed the configured buffer size (BUFSIZE), which
  generally is 8 or 16 kB, otherwise they will be truncated. It is also wise
  not to put any reference to local contents (eg: images) in order to avoid
  loops between the client and HAProxy when all servers are down, causing an
  error to be returned instead of an image. For better HTTP compliance, it is
  recommended that all header lines end with CR-LF and not LF alone.

  The files are read at the same time as the configuration and kept in memory.
  For this reason, the errors continue to be returned even when the process is
  chrooted, and no file change is considered while the process is running. A
  simple method for developing those files consists in associating them to the
  403 status code and interrogating a blocked URL.

  See also : "errorloc", "errorloc302", "errorloc303"

  Example :
        errorfile 400 /etc/haproxy/errorfiles/400badreq.http
        errorfile 408 /dev/null  # workaround Chrome pre-connect bug
        errorfile 403 /etc/haproxy/errorfiles/403forbid.http
        errorfile 503 /etc/haproxy/errorfiles/503sorry.http

```

```
errorloc <code> <url>
errorloc302 <code> <url>
  Return an HTTP redirection to a URL instead of errors generated by HAProxy
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <code>    is the HTTP status code. Currently, HAProxy is capable of
              generating codes 200, 400, 403, 408, 500, 502, 503, and 504.

    <url>     it is the exact contents of the "Location" header. It may contain
              either a relative URI to an error page hosted on the same site,
              or an absolute URI designating an error page on another site.
              Special care should be given to relative URIs to avoid redirect
              loops if the URI itself may generate the same error (eg: 500).

  It is important to understand that this keyword is not meant to rewrite
  errors returned by the server, but errors detected and returned by HAProxy.
  This is why the list of supported errors is limited to a small set.

  Code 200 is emitted in response to requests matching a "monitor-uri" rule.

  Note that both keyword return the HTTP 302 status code, which tells the
  client to fetch the designated URL using the same HTTP method. This can be
  quite problematic in case of non-GET methods such as POST, because the URL
  sent to the client might not be allowed for something other than GET. To
  workaround this problem, please use "errorloc303" which send the HTTP 303
  status code, indicating to the client that the URL must be fetched with a GET
  request.

  See also : "errorfile", "errorloc303"


errorloc303 <code> <url>
  Return an HTTP redirection to a URL instead of errors generated by HAProxy
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <code>    is the HTTP status code. Currently, HAProxy is capable of
              generating codes 400, 403, 408, 500, 502, 503, and 504.

    <url>     it is the exact contents of the "Location" header. It may contain
              either a relative URI to an error page hosted on the same site,
              or an absolute URI designating an error page on another site.
              Special care should be given to relative URIs to avoid redirect
              loops if the URI itself may generate the same error (eg: 500).

  It is important to understand that this keyword is not meant to rewrite
  errors returned by the server, but errors detected and returned by HAProxy.
  This is why the list of supported errors is limited to a small set.

  Code 200 is emitted in response to requests matching a "monitor-uri" rule.

  Note that both keyword return the HTTP 303 status code, which tells the
  client to fetch the designated URL using the same HTTP GET method. This
  solves the usual problems associated with "errorloc" and the 302 code. It is
  possible that some very old browsers designed before HTTP/1.1 do not support
  it, but no such problem has been reported till now.

  See also : "errorfile", "errorloc", "errorloc302"


email-alert from <emailaddr>
  Declare the from email address to be used in both the envelope and header
  of email alerts.  This is the address that email alerts are sent from.
  May be used in sections:    defaults | frontend | listen | backend
```

```
                                 yes   |    yes   |   yes  |   yes

  Arguments :

    <emailaddr> is the from email address to use when sending email alerts

  Also requires "email-alert mailers" and "email-alert to" to be set
  and if so sending email alerts is enabled for the proxy.

  See also : "email-alert level", "email-alert mailers",
             "email-alert myhostname", "email-alert to", section 3.6 about
             mailers.


email-alert level <level>
  Declare the maximum log level of messages for which email alerts will be
  sent. This acts as a filter on the sending of email alerts.
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes

  Arguments :

    <level> One of the 8 syslog levels:
              emerg alert crit err warning notice info  debug
            The above syslog levels are ordered from lowest to highest.

  By default level is alert

  Also requires "email-alert from", "email-alert mailers" and
  "email-alert to" to be set and if so sending email alerts is enabled
  for the proxy.

  Alerts are sent when :

  * An un-paused server is marked as down and <level> is alert or lower
  * A paused server is marked as down and <level> is notice or lower
  * A server is marked as up or enters the drain state and <level>
    is notice or lower
  * "option log-health-checks" is enabled, <level> is info or lower,
     and a health check status update occurs

  See also : "email-alert from", "email-alert mailers",
             "email-alert myhostname", "email-alert to",
             section 3.6 about mailers.


email-alert mailers <mailersect>
  Declare the mailers to be used when sending email alerts
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes

  Arguments :

    <mailersect> is the name of the mailers section to send email alerts.

  Also requires "email-alert from" and "email-alert to" to be set
  and if so sending email alerts is enabled for the proxy.

  See also : "email-alert from", "email-alert level", "email-alert myhostname",
             "email-alert to", section 3.6 about mailers.


email-alert myhostname <hostname>
  Declare the to hostname address to be used when communicating with
  mailers.
```

```
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes

  Arguments :

    <hostname> is the hostname to use when communicating with mailers

  By default the systems hostname is used.

  Also requires "email-alert from", "email-alert mailers" and
  "email-alert to" to be set and if so sending email alerts is enabled
  for the proxy.

  See also : "email-alert from", "email-alert level", "email-alert mailers",
             "email-alert to", section 3.6 about mailers.


email-alert to <emailaddr>
  Declare both the recipent address in the envelope and to address in the
  header of email alerts. This is the address that email alerts are sent to.
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes

  Arguments :

    <emailaddr> is the to email address to use when sending email alerts

  Also requires "email-alert mailers" and "email-alert to" to be set
  and if so sending email alerts is enabled for the proxy.

  See also : "email-alert from", "email-alert level", "email-alert mailers",
             "email-alert myhostname", section 3.6 about mailers.


force-persist { if | unless } <condition>
  Declare a condition to force persistence on down servers
  May be used in sections:    defaults | frontend | listen | backend
                                  no   |    yes   |   yes  |   yes

  By default, requests are not dispatched to down servers. It is possible to
  force this using "option persist", but it is unconditional and redispatches
  to a valid server if "option redispatch" is set. That leaves with very little
  possibilities to force some requests to reach a server which is artificially
  marked down for maintenance operations.

  The "force-persist" statement allows one to declare various ACL-based
  conditions which, when met, will cause a request to ignore the down status of
  a server and still try to connect to it. That makes it possible to start a
  server, still replying an error to the health checks, and run a specially
  configured browser to test the service. Among the handy methods, one could
  use a specific source IP address, or a specific cookie. The cookie also has
  the advantage that it can easily be added/removed on the browser from a test
  page. Once the service is validated, it is then possible to open the service
  to the world by returning a valid response to health checks.

  The forced persistence is enabled when an "if" condition is met, or unless an
  "unless" condition is met. The final redispatch is always disabled when this
  is used.

  See also : "option redispatch", "ignore-persist", "persist",
             and section 7 about ACL usage.


fullconn <conns>
  Specify at what backend load the servers will reach their maxconn
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <conns>   is the number of connections on the backend which will make the
              servers use the maximal number of connections.

  When a server has a "maxconn" parameter specified, it means that its number
  of concurrent connections will never go higher. Additionally, if it has a
  "minconn" parameter, it indicates a dynamic limit following the backend's
  load. The server will then always accept at least <minconn> connections,
  never more than <maxconn>, and the limit will be on the ramp between both
  values when the backend has less than <conns> concurrent connections. This
  makes it possible to limit the load on the servers during normal loads, but
  push it further for important loads without overloading the servers during
  exceptional loads.

  Since it's hard to get this value right, haproxy automatically sets it to
  10% of the sum of the maxconns of all frontends that may branch to this
  backend (based on "use_backend" and "default_backend" rules). That way it's
  safe to leave it unset. However, "use_backend" involving dynamic names are
  not counted since there is no way to know if they could match or not.

  Example :
     # The servers will accept between 100 and 1000 concurrent connections each
     # and the maximum of 1000 will be reached when the backend reaches 10000
     # connections.
     backend dynamic
        fullconn   10000
        server     srv1   dyn1:80 minconn 100 maxconn 1000
        server     srv2   dyn2:80 minconn 100 maxconn 1000

  See also : "maxconn", "server"


grace <time>
  Maintain a proxy operational for some time after a soft stop
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <time>    is the time (by default in milliseconds) for which the instance
              will remain operational with the frontend sockets still listening
              when a soft-stop is received via the SIGUSR1 signal.

  This may be used to ensure that the services disappear in a certain order.
  This was designed so that frontends which are dedicated to monitoring by an
  external equipment fail immediately while other ones remain up for the time
  needed by the equipment to detect the failure.

  Note that currently, there is very little benefit in using this parameter,
  and it may in fact complicate the soft-reconfiguration process more than
  simplify it.


hash-type <method> <function> <modifier>
  Specify a method to use for mapping hashes to servers
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <method> is the method used to select a server from the hash computed by
             the <function> :

      map-based   the hash table is a static array containing all alive servers.
                  The hashes will be very smooth, will consider weights, but
                  will be static in that weight changes while a server is up
                  will be ignored. This means that there will be no slow start.
```

```
                  Also, since a server is selected by its position in the array,
                  most mappings are changed when the server count changes. This
                  means that when a server goes up or down, or when a server is
                  added to a farm, most connections will be redistributed to
                  different servers. This can be inconvenient with caches for
                  instance.

      consistent  the hash table is a tree filled with many occurrences of each
                  server. The hash key is looked up in the tree and the closest
                  server is chosen. This hash is dynamic, it supports changing
                  weights while the servers are up, so it is compatible with the
                  slow start feature. It has the advantage that when a server
                  goes up or down, only its associations are moved. When a
                  server is added to the farm, only a few part of the mappings
                  are redistributed, making it an ideal method for caches.
                  However, due to its principle, the distribution will never be
                  very smooth and it may sometimes be necessary to adjust a
                  server's weight or its ID to get a more balanced distribution.
                  In order to get the same distribution on multiple load
                  balancers, it is important that all servers have the exact
                  same IDs. Note: consistent hash uses sdbm and avalanche if no
                  hash function is specified.

    <function> is the hash function to be used :

       sdbm   this function was created initially for sdbm (a public-domain
              reimplementation of ndbm) database library. It was found to do
              well in scrambling bits, causing better distribution of the keys
              and fewer splits. It also happens to be a good general hashing
              function with good distribution, unless the total server weight
              is a multiple of 64, in which case applying the avalanche
              modifier may help.

       djb2   this function was first proposed by Dan Bernstein many years ago
              on comp.lang.c. Studies have shown that for certain workload this
              function provides a better distribution than sdbm. It generally
              works well with text-based inputs though it can perform extremely
              poorly with numeric-only input or when the total server weight is
              a multiple of 33, unless the avalanche modifier is also used.

       wt6    this function was designed for haproxy while testing other
              functions in the past. It is not as smooth as the other ones, but
              is much less sensible to the input data set or to the number of
              servers. It can make sense as an alternative to sdbm+avalanche or
              djb2+avalanche for consistent hashing or when hashing on numeric
              data such as a source IP address or a visitor identifier in a URL
              parameter.

       crc32  this is the most common CRC32 implementation as used in Ethernet,
              gzip, PNG, etc. It is slower than the other ones but may provide
              a better distribution or less predictable results especially when
              used on strings.

    <modifier> indicates an optional method applied after hashing the key :

       avalanche   This directive indicates that the result from the hash
                   function above should not be used in its raw form but that
                   a 4-byte full avalanche hash must be applied first. The
                   purpose of this step is to mix the resulting bits from the
                   previous hash in order to avoid any undesired effect when
                   the input contains some limited values or when the number of
                   servers is a multiple of one of the hash's components (64
                   for SDBM, 33 for DJB2). Enabling avalanche tends to make the
                   result less predictable, but it's also not as smooth as when
                   using the original function. Some testing might be needed
                  with some workloads. This hash is one of the many proposed
                  by Bob Jenkins.

  The default hash type is "map-based" and is recommended for most usages. The
  default function is "sdbm", the selection of a function should be based on
  the range of the values being hashed.

  See also : "balance", "server"


http-check disable-on-404
  Enable a maintenance mode upon HTTP/404 response to health-checks
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  When this option is set, a server which returns an HTTP code 404 will be
  excluded from further load-balancing, but will still receive persistent
  connections. This provides a very convenient method for Web administrators
  to perform a graceful shutdown of their servers. It is also important to note
  that a server which is detected as failed while it was in this mode will not
  generate an alert, just a notice. If the server responds 2xx or 3xx again, it
  will immediately be reinserted into the farm. The status on the stats page
  reports "NOLB" for a server in this mode. It is important to note that this
  option only works in conjunction with the "httpchk" option. If this option
  is used with "http-check expect", then it has precedence over it so that 404
  responses will still be considered as soft-stop.

  See also : "option httpchk", "http-check expect"


http-check expect [!] <match> <pattern>
  Make HTTP health checks consider response contents or specific status codes
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <match>   is a keyword indicating how to look for a specific pattern in the
              response. The keyword may be one of "status", "rstatus",
              "string", or "rstring". The keyword may be preceded by an
              exclamation mark ("!") to negate the match. Spaces are allowed
              between the exclamation mark and the keyword. See below for more
              details on the supported keywords.

    <pattern> is the pattern to look for. It may be a string or a regular
              expression. If the pattern contains spaces, they must be escaped
              with the usual backslash ('\').

  By default, "option httpchk" considers that response statuses 2xx and 3xx
  are valid, and that others are invalid. When "http-check expect" is used,
  it defines what is considered valid or invalid. Only one "http-check"
  statement is supported in a backend. If a server fails to respond or times
  out, the check obviously fails. The available matches are :

    status <string> : test the exact string match for the HTTP status code.
                      A health check response will be considered valid if the
                      response's status code is exactly this string. If the
                      "status" keyword is prefixed with "!", then the response
                      will be considered invalid if the status code matches.

    rstatus <regex> : test a regular expression for the HTTP status code.
                      A health check response will be considered valid if the
                      response's status code matches the expression. If the
                      "rstatus" keyword is prefixed with "!", then the response
                      will be considered invalid if the status code matches.
                      This is mostly used to check for multiple codes.
```

```

    string <string> : test the exact string match in the HTTP response body.
                      A health check response will be considered valid if the
                      response's body contains this exact string. If the
                      "string" keyword is prefixed with "!", then the response
                      will be considered invalid if the body contains this
                      string. This can be used to look for a mandatory word at
                      the end of a dynamic page, or to detect a failure when a
                      specific error appears on the check page (eg: a stack
                      trace).

    rstring <regex> : test a regular expression on the HTTP response body.
                      A health check response will be considered valid if the
                      response's body matches this expression. If the "rstring"
                      keyword is prefixed with "!", then the response will be
                      considered invalid if the body matches the expression.
                      This can be used to look for a mandatory word at the end
                      of a dynamic page, or to detect a failure when a specific
                      error appears on the check page (eg: a stack trace).

  It is important to note that the responses will be limited to a certain size
  defined by the global "tune.chksize" option, which defaults to 16384 bytes.
  Thus, too large responses may not contain the mandatory pattern when using
  "string" or "rstring". If a large response is absolutely required, it is
  possible to change the default max size by setting the global variable.
  However, it is worth keeping in mind that parsing very large responses can
  waste some CPU cycles, especially when regular expressions are used, and that
  it is always better to focus the checks on smaller resources.

  Also "http-check expect" doesn't support HTTP keep-alive. Keep in mind that it
  will automatically append a "Connection: close" header, meaning that this
  header should not be present in the request provided by "option httpchk".

  Last, if "http-check expect" is combined with "http-check disable-on-404",
  then this last one has precedence when the server responds with 404.

  Examples :
         # only accept status 200 as valid
         http-check expect status 200

         # consider SQL errors as errors
         http-check expect ! string SQL\ Error

         # consider status 5xx only as errors
         http-check expect ! rstatus ^5

         # check that we have a correct hexadecimal tag before /html
         http-check expect rstring <!--tag:[0-9a-f]*</html>

  See also : "option httpchk", "http-check disable-on-404"


http-check send-state
  Enable emission of a state header with HTTP health checks
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  When this option is set, haproxy will systematically send a special header
  "X-Haproxy-Server-State" with a list of parameters indicating to each server
  how they are seen by haproxy. This can be used for instance when a server is
  manipulated without access to haproxy and the operator needs to know whether
  haproxy still sees it up or not, or if the server is the last one in a farm.

  The header is composed of fields delimited by semi-colons, the first of which
  is a word ("UP", "DOWN", "NOLB"), possibly followed by a number of valid
  checks on the total number before transition, just as appears in the stats
  interface. Next headers are in the form "<variable>=<value>", indicating in
  no specific order some values available in the stats interface :
    - a variable "address", containing the address of the backend server.
      This corresponds to the <address> field in the server declaration. For
      unix domain sockets, it will read "unix".

    - a variable "port", containing the port of the backend server. This
      corresponds to the <port> field in the server declaration. For unix
      domain sockets, it will read "unix".

    - a variable "name", containing the name of the backend followed by a slash
      ("/") then the name of the server. This can be used when a server is
      checked in multiple backends.

    - a variable "node" containing the name of the haproxy node, as set in the
      global "node" variable, otherwise the system's hostname if unspecified.

    - a variable "weight" indicating the weight of the server, a slash ("/")
      and the total weight of the farm (just counting usable servers). This
      helps to know if other servers are available to handle the load when this
      one fails.

    - a variable "scur" indicating the current number of concurrent connections
      on the server, followed by a slash ("/") then the total number of
      connections on all servers of the same backend.

    - a variable "qcur" indicating the current number of requests in the
      server's queue.

  Example of a header received by the application server :
    >>>  X-Haproxy-Server-State: UP 2/3; name=bck/srv2; node=lb1; weight=1/2; \
         scur=13/22; qcur=0

  See also : "option httpchk", "http-check disable-on-404"

http-request { allow | deny | tarpit | auth [realm <realm>] | redirect <rule> |
              add-header <name> <fmt> | set-header <name> <fmt> |
              capture <sample> [ len <length> | id <id> ] |
              del-header <name> | set-nice <nice> | set-log-level <level> |
              replace-header <name> <match-regex> <replace-fmt> |
              replace-value <name> <match-regex> <replace-fmt> |
              set-method <fmt> | set-path <fmt> | set-query <fmt> |
              set-uri <fmt> | set-tos <tos> | set-mark <mark> |
              add-acl(<file name>) <key fmt> |
              del-acl(<file name>) <key fmt> |
              del-map(<file name>) <key fmt> |
              set-map(<file name>) <key fmt> <value fmt> |
              set-var(<var name>) <expr> |
              { track-sc0 | track-sc1 | track-sc2 } <key> [table <table>] |
              sc-inc-gpc0(<sc-id>) |
              sc-set-gpt0(<sc-id>) <int> |
              silent-drop |
             }
             [ { if | unless } <condition> ]
  Access control for Layer 7 requests

  May be used in sections:   defaults | frontend | listen | backend
                                no    |    yes   |   yes  |   yes

  The http-request statement defines a set of rules which apply to layer 7
  processing. The rules are evaluated in their declaration order when they are
  met in a frontend, listen or backend section. Any rule may optionally be
  followed by an ACL-based condition, in which case it will only be evaluated
```

```
  if the condition is true.

  The first keyword is the rule's action. Currently supported actions include :
    - "allow" : this stops the evaluation of the rules and lets the request
      pass the check. No further "http-request" rules are evaluated.

    - "deny" : this stops the evaluation of the rules and immediately rejects
      the request and emits an HTTP 403 error. No further "http-request" rules
      are evaluated.

    - "tarpit" : this stops the evaluation of the rules and immediately blocks
      the request without responding for a delay specified by "timeout tarpit"
      or "timeout connect" if the former is not set. After that delay, if the
      client is still connected, an HTTP error 500 is returned so that the
      client does not suspect it has been tarpitted. Logs will report the flags
      "PT". The goal of the tarpit rule is to slow down robots during an attack
      when they're limited on the number of concurrent requests. It can be very
      efficient against very dumb robots, and will significantly reduce the
      load on firewalls compared to a "deny" rule. But when facing "correctly"
      developed robots, it can make things worse by forcing haproxy and the
      front firewall to support insane number of concurrent connections. See
      also the "silent-drop" action below.

    - "auth" : this stops the evaluation of the rules and immediately responds
      with an HTTP 401 or 407 error code to invite the user to present a valid
      user name and password. No further "http-request" rules are evaluated. An
      optional "realm" parameter is supported, it sets the authentication realm
      that is returned with the response (typically the application's name).

    - "redirect" : this performs an HTTP redirection based on a redirect rule.
      This is exactly the same as the "redirect" statement except that it
      inserts a redirect rule which can be processed in the middle of other
      "http-request" rules and that these rules use the "log-format" strings.
      See the "redirect" keyword for the rule's syntax.

    - "add-header" appends an HTTP header field whose name is specified in
      <name> and whose value is defined by <fmt> which follows the log-format
      rules (see Custom Log Format in section 8.2.4). This is particularly
      useful to pass connection-specific information to the server (eg: the
      client's SSL certificate), or to combine several headers into one. This
      rule is not final, so it is possible to add other similar rules. Note
      that header addition is performed immediately, so one rule might reuse
      the resulting header from a previous rule.

    - "set-header" does the same as "add-header" except that the header name
      is first removed if it existed. This is useful when passing security
      information to the server, where the header must not be manipulated by
      external users. Note that the new value is computed before the removal so
      it is possible to concatenate a value to an existing header.

    - "del-header" removes all HTTP header fields whose name is specified in
      <name>.

    - "replace-header" matches the regular expression in all occurrences of
      header field <name> according to <match-regex>, and replaces them with
      the <replace-fmt> argument. Format characters are allowed in replace-fmt
      and work like in <fmt> arguments in "add-header". The match is only
      case-sensitive. It is important to understand that this action only
      considers whole header lines, regardless of the number of values they
      may contain. This usage is suited to headers naturally containing commas
      in their value, such as If-Modified-Since and so on.

      Example:

        http-request replace-header Cookie foo=([^;]*);(.*) foo=\1;ip=%bi;\2
```

applied to:

```
  Cookie: foo=foobar; expires=Tue, 14-Jun-2016 01:40:45 GMT;
```

outputs:

```
  Cookie: foo=foobar;ip=192.168.1.20; expires=Tue, 14-Jun-2016 01:40:45 GMT;
```

assuming the backend IP is 192.168.1.20

- "replace-value" works like "replace-header" except that it matches the regex against every comma-delimited value of the header field <name> instead of the entire header. This is suited for all headers which are allowed to carry more than one value. An example could be the Accept header.

  Example:

  ```
    http-request replace-value X-Forwarded-For ^192\.168\.(.*)$ 172.16.\1
  ```

  applied to:

  ```
    X-Forwarded-For: 192.168.10.1, 192.168.13.24, 10.0.0.37
  ```

  outputs:

  ```
    X-Forwarded-For: 172.16.10.1, 172.16.13.24, 10.0.0.37
  ```

- "set-method" rewrites the request method with the result of the evaluation of format string <fmt>. There should be very few valid reasons for having to do so as this is more likely to break something than to fix it.

- "set-path" rewrites the request path with the result of the evaluation of format string <fmt>. The query string, if any, is left intact. If a scheme and authority is found before the path, they are left intact as well. If the request doesn't have a path ("*"), this one is replaced with the format. This can be used to prepend a directory component in front of a path for example. See also "set-query" and "set-uri".

  Example :

  ```
      # prepend the host name before the path
      http-request set-path /%[hdr(host)]%[path]
  ```

- "set-query" rewrites the request's query string which appears after the first question mark ("?") with the result of the evaluation of format string <fmt>. The part prior to the question mark is left intact. If the request doesn't contain a question mark and the new value is not empty, then one is added at the end of the URI, followed by the new value. If a question mark was present, it will never be removed even if the value is empty. This can be used to add or remove parameters from the query string. See also "set-query" and "set-uri".

  Example :

  ```
      # replace "%3D" with "=" in the query string
      http-request set-query %[query,regsub(%3D,=,g)]
  ```

- "set-uri" rewrites the request URI with the result of the evaluation of format string <fmt>. The scheme, authority, path and query string are all replaced at once. This can be used to rewrite hosts in front of proxies, or to perform complex modifications to the URI such as moving parts between the path and the query string. See also "set-path" and "set-query".

- "set-nice" sets the "nice" factor of the current request being processed. It only has effect against the other requests being processed at the same time. The default value is 0, unless altered by the "nice" setting on the "bind" line. The accepted range is -1024..1024. The higher the value, the nicest the request will be. Lower values will make the request more important than other ones. This can be useful to improve the speed of some requests, or lower the priority of non-important requests. Using this setting without prior experimentation can cause some major slowdown.

- "set-log-level" is used to change the log level of the current request when a certain condition is met. Valid levels are the 8 syslog levels (see the "log" keyword) plus the special level "silent" which disables logging for this request. This rule is not final so the last matching rule wins. This rule can be useful to disable health checks coming from another equipment.

- "set-tos" is used to set the TOS or DSCP field value of packets sent to the client to the value passed in <tos> on platforms which support this. This value represents the whole 8 bits of the IP TOS field, and can be expressed both in decimal or hexadecimal format (prefixed by "0x"). Note that only the 6 higher bits are used in DSCP or TOS, and the two lower bits are always 0. This can be used to adjust some routing behaviour on border routers based on some information from the request. See RFC 2474, 2597, 3260 and 4594 for more information.

- "set-mark" is used to set the Netfilter MARK on all packets sent to the client to the value passed in <mark> on platforms which support it. This value is an unsigned 32 bit value which can be matched by netfilter and by the routing table. It can be expressed both in decimal or hexadecimal format (prefixed by "0x"). This can be useful to force certain packets to take a different route (for example a cheaper network path for bulk downloads). This works on Linux kernels 2.6.32 and above and requires admin privileges.

- "add-acl" is used to add a new entry into an ACL. The ACL must be loaded from a file (even a dummy empty file). The file name of the ACL to be updated is passed between parentheses. It takes one argument: <key fmt>, which follows log-format rules, to collect content of the new entry. It performs a lookup in the ACL before insertion, to avoid duplicated (or more) values. This lookup is done by a linear search and can be expensive with large lists! It is the equivalent of the "add acl" command from the stats socket, but can be triggered by an HTTP request.

- "del-acl" is used to delete an entry from an ACL. The ACL must be loaded from a file (even a dummy empty file). The file name of the ACL to be updated is passed between parentheses. It takes one argument: <key fmt>, which follows log-format rules, to collect content of the entry to delete. It is the equivalent of the "del acl" command from the stats socket, but can be triggered by an HTTP request.

- "del-map" is used to delete an entry from a MAP. The MAP must be loaded from a file (even a dummy empty file). The file name of the MAP to be updated is passed between parentheses. It takes one argument: <key fmt>, which follows log-format rules, to collect content of the entry to delete. It takes one argument: "file name" It is the equivalent of the "del map" command from the stats socket, but can be triggered by an HTTP request.

- "set-map" is used to add a new entry into a MAP. The MAP must be loaded from a file (even a dummy empty file). The file name of the MAP to be updated is passed between parentheses. It takes 2 arguments: <key fmt>, which follows log-format rules, used to collect MAP key, and <value fmt>, which follows log-format rules, used to collect content for the new entry. It performs a lookup in the MAP before insertion, to avoid duplicated (or more) values. This lookup is done by a linear search and can be expensive with large lists! It is the equivalent of the "set map" command from the stats socket, but can be triggered by an HTTP request.

- capture <sample> [len <length> | id <id>] :
  captures sample expression <sample> from the request buffer, and converts it to a string of at most <len> characters. The resulting string is stored into the next request "capture" slot, so it will possibly appear next to some captured HTTP headers. It will then automatically appear in the logs, and it will be possible to extract it using sample fetch rules to feed it into headers or anything. The length should be limited given that this size will be allocated for each capture during the whole session life. Please check section 7.3 (Fetching samples) and "capture request header" for more information.

  If the keyword "id" is used instead of "len", the action tries to store the captured string in a previously declared capture slot. This is useful to run captures in backends. The slot id can be declared by a previous directive "http-request capture" or with the "declare capture" keyword. If the slot <id> doesn't exist, then HAProxy fails parsing the configuration to prevent unexpected behavior at run time.

- { track-sc0 | track-sc1 | track-sc2 } <key> [table <table>] :
  enables tracking of sticky counters from current request. These rules do not stop evaluation and do not change default action. Three sets of counters may be simultaneously tracked by the same connection. The first "track-sc0" rule executed enables tracking of the counters of the specified table as the first set. The first "track-sc1" rule executed enables tracking of the counters of the specified table as the second set. The first "track-sc2" rule executed enables tracking of the counters of the specified table as the third set. It is a recommended practice to use the first set of counters for the per-frontend counters and the second set for the per-backend ones. But this is just a guideline, all may be used everywhere.

  These actions take one or two arguments :

```
    <key>   is mandatory, and is a sample expression rule as described
            in section 7.3. It describes what elements of the incoming
            request or connection will be analysed, extracted, combined,
            and used to select which table entry to update the counters.

    <table> is an optional table to be used instead of the default one,
            which is the stick-table declared in the current proxy. All
            the counters for the matches and updates for the key will
            then be performed in that table until the session ends.
```

  Once a "track-sc*" rule is executed, the key is looked up in the table and if it is not found, an entry is allocated for it. Then a pointer to that entry is kept during all the session's life, and this entry's counters are updated as often as possible, every time the session's counters are updated, and also systematically when the session ends. Counters are only updated for events that happen after the tracking has been started. As an exception, connection counters and request counters are systematically updated so that they reflect useful information.

  If the entry tracks concurrent connection counters, one connection is counted for as long as the entry is tracked, and the entry will not expire during that time. Tracking counters also provides a performance advantage over just checking the keys, because only one table lookup is performed for all ACL checks that make use of it.

- sc-set-gpt0(<sc-id>) <int> :
  This action sets the GPT0 tag according to the sticky counter designated by <sc-id> and the value of <int>. The expected result is a boolean. If an error occurs, this action silently fails and the actions evaluation continues.

- sc-inc-gpc0(<sc-id>):
  This action increments the GPC0 counter according with the sticky counter designated by <sc-id>. If an error occurs, this action silently fails and the actions evaluation continues.

- set-var(<var-name>) <expr> :
  Is used to set the contents of a variable. The variable is declared inline.

```
    <var-name> The name of the variable starts by an indication about its
               scope. The allowed scopes are:
                 "sess" : the variable is shared with all the session,
                 "txn"  : the variable is shared with all the transaction
                          (request and response)
                 "req"  : the variable is shared only during the request
                          processing
                 "res"  : the variable is shared only during the response
                          processing.
               This prefix is followed by a name. The separator is a '.'.
               The name may only contain characters 'a-z', 'A-Z', '0-9',
               and '_'.

     <expr>    Is a standard HAProxy expression formed by a sample-fetch
               followed by some converters.
```

  Example:

```
     http-request set-var(req.my_var) req.fhdr(user-agent),lower
```

- set-src <expr> :
  Is used to set the source IP address to the value of specified expression. Useful when a proxy in front of HAProxy rewrites source IP, but provides the correct IP in a HTTP header; or you want to mask source IP for privacy.

```
     <expr>    Is a standard HAProxy expression formed by a sample-fetch
               followed by some converters.
```

  Example:

```
     http-request set-src hdr(x-forwarded-for)
     http-request set-src src,ipmask(24)
```

  When set-src is successful, the source port is set to 0.

- "silent-drop" : this stops the evaluation of the rules and makes the client-facing connection suddenly disappear using a system-dependant way that tries to prevent the client from being notified. The effect it then that the client still sees an established connection while there's none on HAProxy. The purpose is to achieve a comparable effect to "tarpit" except that it doesn't use any local resource at all on the machine running HAProxy. It can resist much higher loads than "tarpit", and slow down stronger attackers. It is important to undestand the impact of using this mechanism. All stateful equipments placed between the client and HAProxy (firewalls, proxies, load balancers) will also keep the established connection for a long time and may suffer from this action. On modern Linux systems running with enough privileges, the TCP_REPAIR socket option is used to block the emission of a TCP reset. On other systems, the socket's TTL is reduced to 1 so that the TCP reset doesn't pass the first router, though it's still delivered to local networks. Do not use it unless you fully understand how it works.

There is no limit to the number of http-request statements per instance.

It is important to know that http-request rules are processed very early in the HTTP processing, just after "block" rules and before "reqdel" or "reqrep" or "reqadd" rules. That way, headers added by "add-header"/"set-header" are visible by almost all further ACL rules.

Using "reqadd"/"reqdel"/"reqrep" to manipulate request headers is discouraged in newer versions (>= 1.5). But if you need to use regular expression to delete headers, you can still use "reqdel". Also please use "http-request deny/allow/tarpit" instead of "reqdeny"/"reqpass"/"reqtarpit".

Example:
```
      acl nagios src 192.168.129.3
      acl local_net src 192.168.0.0/16
      acl auth_ok http_auth(L1)

      http-request allow if nagios
      http-request allow if local_net auth_ok
      http-request auth realm Gimme if local_net auth_ok
      http-request deny
```

Example:
```
      acl auth_ok http_auth_group(L1) G1
      http-request auth unless auth_ok
```

Example:
```
      http-request set-header X-Haproxy-Current-Date %T
      http-request set-header X-SSL                  %[ssl_fc]
      http-request set-header X-SSL-Session_ID       %[ssl_fc_session_id,hex]
      http-request set-header X-SSL-Client-Verify    %[ssl_c_verify]
      http-request set-header X-SSL-Client-DN        %{+Q}[ssl_c_s_dn]
      http-request set-header X-SSL-Client-CN        %{+Q}[ssl_c_s_dn(cn)]
      http-request set-header X-SSL-Issuer           %{+Q}[ssl_c_i_dn]
      http-request set-header X-SSL-Client-NotBefore %{+Q}[ssl_c_notbefore]
      http-request set-header X-SSL-Client-NotAfter  %{+Q}[ssl_c_notafter]
```

Example:
```
      acl key req.hdr(X-Add-Acl-Key) -m found
      acl add path /addacl
      acl del path /delacl

      acl myhost hdr(Host) -f myhost.lst

      http-request add-acl(myhost.lst) %[req.hdr(X-Add-Acl-Key)] if key add
      http-request del-acl(myhost.lst) %[req.hdr(X-Add-Acl-Key)] if key del
```

Example:
```
      acl value  req.hdr(X-Value) -m found
      acl setmap path /setmap
      acl delmap path /delmap

      use_backend bk_appli if { hdr(Host),map_str(map.lst) -m found }

      http-request set-map(map.lst) %[src] %[req.hdr(X-Value)] if setmap value
      http-request del-map(map.lst) %[src]                    if delmap
```

See also : "stats http-request", section 3.4 about userlists and section 7 about ACL usage.

```
http-response { allow | deny | add-header <name> <fmt> | set-nice <nice> |
                capture <sample> id <id> | redirect <rule> |
                set-header <name> <fmt> | del-header <name> |
                replace-header <name> <regex-match> <replace-fmt> |
                replace-value <name> <regex-match> <replace-fmt> |
                set-status <status> |
                set-log-level <level> | set-mark <mark> | set-tos <tos> |
                add-acl(<file name>) <key fmt> |
```

```
                del-acl(<file name>) <key fmt> |
                del-map(<file name>) <key fmt> |
                set-map(<file name>) <key fmt> <value fmt> |
                set-var(<var-name>) <expr> |
                sc-inc-gpc0(<sc-id>) |
                sc-set-gpt0(<sc-id>) <int> |
                silent-drop |
              }
              [ { if | unless } <condition> ]
```
Access control for Layer 7 responses

```
May be used in sections:   defaults | frontend | listen | backend
                              no    |    yes   |   yes  |   yes
```

The http-response statement defines a set of rules which apply to layer 7 processing. The rules are evaluated in their declaration order when they are met in a frontend, listen or backend section. Any rule may optionally be followed by an ACL-based condition, in which case it will only be evaluated if the condition is true. Since these rules apply on responses, the backend rules are applied first, followed by the frontend's rules.

The first keyword is the rule's action. Currently supported actions include :

- "allow" : this stops the evaluation of the rules and lets the response pass the check. No further "http-response" rules are evaluated for the current section.

- "deny" : this stops the evaluation of the rules and immediately rejects the response and emits an HTTP 502 error. No further "http-response" rules are evaluated.

- "add-header" appends an HTTP header field whose name is specified in <name> and whose value is defined by <fmt> which follows the log-format rules (see Custom Log Format in section 8.2.4). This may be used to send a cookie to a client for example, or to pass some internal information. This rule is not final, so it is possible to add other similar rules. Note that header addition is performed immediately, so one rule might reuse the resulting header from a previous rule.

- "set-header" does the same as "add-header" except that the header name is first removed if it existed. This is useful when passing security information to the server, where the header must not be manipulated by external users.

- "del-header" removes all HTTP header fields whose name is specified in <name>.

- "replace-header" matches the regular expression in all occurrences of header field <name> according to <match-regex>, and replaces them with the <replace-fmt> argument. Format characters are allowed in replace-fmt and work like in <fmt> arguments in "add-header". The match is only case-sensitive. It is important to understand that this action only considers whole header lines, regardless of the number of values they may contain. This usage is suited to headers naturally containing commas in their value, such as Set-Cookie, Expires and so on.

  Example:

  ```
    http-response replace-header Set-Cookie (C=[^;]*);(.*) \1;ip=%bi;\2
  ```

  applied to:

  ```
    Set-Cookie: C=1; expires=Tue, 14-Jun-2016 01:40:45 GMT
  ```

  outputs:

```
    Set-Cookie: C=1;ip=192.168.1.20; expires=Tue, 14-Jun-2016 01:40:45 GMT
```

assuming the backend IP is 192.168.1.20.

- "replace-value" works like "replace-header" except that it matches the regex against every comma-delimited value of the header field <name> instead of the entire header. This is suited for all headers which are allowed to carry more than one value. An example could be the Accept header.

  Example:

  ```
    http-response replace-value Cache-control ^public$ private
  ```

  applied to:

  ```
    Cache-Control: max-age=3600, public
  ```

  outputs:

  ```
    Cache-Control: max-age=3600, private
  ```

- "set-status" replaces the response status code with <status> which must be an integer between 100 and 999. Note that the reason is automatically adapted to the new code.

  Example:

  ```
    # return "431 Request Header Fields Too Large"
    http-response set-status 431
  ```

- "set-nice" sets the "nice" factor of the current request being processed. It only has effect against the other requests being processed at the same time. The default value is 0, unless altered by the "nice" setting on the "bind" line. The accepted range is -1024..1024. The higher the value, the nicest the request will be. Lower values will make the request more important than other ones. This can be useful to improve the speed of some requests, or lower the priority of non-important requests. Using this setting without prior experimentation can cause some major slowdown.

- "set-log-level" is used to change the log level of the current request when a certain condition is met. Valid levels are the 8 syslog levels (see the "log" keyword) plus the special level "silent" which disables logging for this request. This rule is not final so the last matching rule wins. This rule can be useful to disable health checks coming from another equipment.

- "set-tos" is used to set the TOS or DSCP field value of packets sent to the client to the value passed in <tos> on platforms which support this. This value represents the whole 8 bits of the IP TOS field, and can be expressed both in decimal or hexadecimal format (prefixed by "0x"). Note that only the 6 higher bits are used in DSCP or TOS, and the two lower bits are always 0. This can be used to adjust some routing behaviour on border routers based on some information from the request. See RFC 2474, 2597, 3260 and 4594 for more information.

- "set-mark" is used to set the Netfilter MARK on all packets sent to the client to the value passed in <mark> on platforms which support it. This value is an unsigned 32 bit value which can be matched by netfilter and by the routing table. It can be expressed both in decimal or hexadecimal format (prefixed by "0x"). This can be useful to force certain packets to take a different route (for example a cheaper network path for bulk downloads). This works on Linux kernels 2.6.32 and above and requires admin privileges.

- "add-acl" is used to add a new entry into an ACL. The ACL must be loaded from a file (even a dummy empty file). The file name of the ACL to be updated is passed between parentheses. It takes one argument: <key fmt>, which follows log-format rules, to collect content of the new entry. It performs a lookup in the ACL before insertion, to avoid duplicated (or more) values. This lookup is done by a linear search and can be expensive with large lists! It is the equivalent of the "add acl" command from the stats socket, but can be triggered by an HTTP response.

- "del-acl" is used to delete an entry from an ACL. The ACL must be loaded from a file (even a dummy empty file). The file name of the ACL to be updated is passed between parentheses. It takes one argument: <key fmt>, which follows log-format rules, to collect content of the entry to delete. It is the equivalent of the "del acl" command from the stats socket, but can be triggered by an HTTP response.

- "del-map" is used to delete an entry from a MAP. The MAP must be loaded from a file (even a dummy empty file). The file name of the MAP to be updated is passed between parentheses. It takes one argument: <key fmt>, which follows log-format rules, to collect content of the entry to delete. It takes one argument: "file name" It is the equivalent of the "del map" command from the stats socket, but can be triggered by an HTTP response.

- "set-map" is used to add a new entry into a MAP. The MAP must be loaded from a file (even a dummy empty file). The file name of the MAP to be updated is passed between parentheses. It takes 2 arguments: <key fmt>, which follows log-format rules, used to collect MAP key, and <value fmt>, which follows log-format rules, used to collect content for the new entry. It performs a lookup in the MAP before insertion, to avoid duplicated (or more) values. This lookup is done by a linear search and can be expensive with large lists! It is the equivalent of the "set map" command from the stats socket, but can be triggered by an HTTP response.

- capture <sample> id <id> :
  captures sample expression <sample> from the response buffer, and converts it to a string. The resulting string is stored into the next request "capture" slot, so it will possibly appear next to some captured HTTP headers. It will then automatically appear in the logs, and it will be possible to extract it using sample fetch rules to feed it into headers or anything. Please check section 7.3 (Fetching samples) and "capture response header" for more information.

  The keyword "id" is the id of the capture slot which is used for storing the string. The capture slot must be defined in an associated frontend. This is useful to run captures in backends. The slot id can be declared by a previous directive "http-response capture" or with the "declare capture" keyword.
  If the slot <id> doesn't exist, then HAProxy fails parsing the configuration to prevent unexpected behavior at run time.

- "redirect" : this performs an HTTP redirection based on a redirect rule. This supports a format string similarly to "http-request redirect" rules, with the exception that only the "location" type of redirect is possible on the response. See the "redirect" keyword for the rule's syntax. When a redirect rule is applied during a response, connections to the server are closed so that no data can be forwarded from the server to the client.

- set-var(<var-name>) expr:
  Is used to set the contents of a variable. The variable is declared inline.

```
    <var-name> The name of the variable starts by an indication about its
               scope. The allowed scopes are:
                 "sess" : the variable is shared with all the session,
                 "txn"  : the variable is shared with all the transaction
```

```
                              (request and response)
                     "req"  : the variable is shared only during the request
                              processing
                     "res"  : the variable is shared only during the response
                              processing.
                   This prefix is followed by a name. The separator is a '.'.
                   The name may only contain characters 'a-z', 'A-Z', '0-9',
                   and '_'.

         <expr>    Is a standard HAProxy expression formed by a sample-fetch
                   followed by some converters.

      Example:

         http-response set-var(sess.last_redir) res.hdr(location)

    - sc-set-gpt0(<sc-id>) <int> :
      This action sets the GPT0 tag according to the sticky counter designated
      by <sc-id> and the value of <int>. The expected result is a boolean. If
      an error occurs, this action silently fails and the actions evaluation
      continues.

    - sc-inc-gpc0(<sc-id>):
      This action increments the GPC0 counter according with the sticky counter
      designated by <sc-id>. If an error occurs, this action silently fails and
      the actions evaluation continues.

    - "silent-drop" : this stops the evaluation of the rules and makes the
      client-facing connection suddenly disappear using a system-dependant way
      that tries to prevent the client from being notified. The effect it then
      that the client still sees an established connection while there's none
      on HAProxy. The purpose is to achieve a comparable effect to "tarpit"
      except that it doesn't use any local resource at all on the machine
      running HAProxy. It can resist much higher loads than "tarpit", and slow
      down stronger attackers. It is important to undestand the impact of using
      this mechanism. All stateful equipments placed between the client and
      HAProxy (firewalls, proxies, load balancers) will also keep the
      established connection for a long time and may suffer from this action.
      On modern Linux systems running with enough privileges, the TCP_REPAIR
      socket option is used to block the emission of a TCP reset. On other
      systems, the socket's TTL is reduced to 1 so that the TCP reset doesn't
      pass the first router, though it's still delivered to local networks. Do
      not use it unless you fully understand how it works.

  There is no limit to the number of http-response statements per instance.

  It is important to know that http-response rules are processed very early in
  the HTTP processing, before "rspdel" or "rsprep" or "rspadd" rules. That way,
  headers added by "add-header"/"set-header" are visible by almost all further ACL
  rules.

  Using "rspadd"/"rspdel"/"rsprep" to manipulate request headers is discouraged
  in newer versions (>= 1.5). But if you need to use regular expression to
  delete headers, you can still use "rspdel". Also please use
  "http-response deny" instead of "rspdeny".

  Example:
         acl key_acl res.hdr(X-Acl-Key) -m found

         acl myhost hdr(Host) -f myhost.lst

         http-response add-acl(myhost.lst) %[res.hdr(X-Acl-Key)] if key_acl
         http-response del-acl(myhost.lst) %[res.hdr(X-Acl-Key)] if key_acl

  Example:
```

```
         acl value  res.hdr(X-Value) -m found

         use_backend bk_appli if { hdr(Host),map_str(map.lst) -m found }

         http-response set-map(map.lst) %[src] %[res.hdr(X-Value)] if value
         http-response del-map(map.lst) %[src]                     if ! value

  See also : "http-request", section 3.4 about userlists and section 7 about
             ACL usage.


http-reuse { never | safe | aggressive | always }
  Declare how idle HTTP connections may be shared between requests

  May be used in sections:   defaults | frontend | listen | backend
                               yes    |    no    |   yes  |   yes

  By default, a connection established between haproxy and the backend server
  belongs to the session that initiated it. The downside is that between the
  response and the next request, the connection remains idle and is not used.
  In many cases for performance reasons it is desirable to make it possible to
  reuse these idle connections to serve other requests from different sessions.
  This directive allows to tune this behaviour.

  The argument indicates the desired connection reuse strategy :

    - "never"  : idle connections are never shared between sessions. This is
                 the default choice. It may be enforced to cancel a different
                 strategy inherited from a defaults section or for
                 troubleshooting. For example, if an old bogus application
                 considers that multiple requests over the same connection come
                 from the same client and it is not possible to fix the
                 application, it may be desirable to disable connection sharing
                 in a single backend. An example of such an application could
                 be an old haproxy using cookie insertion in tunnel mode and
                 not checking any request past the first one.

    - "safe"   : this is the recommended strategy. The first request of a
                 session is always sent over its own connection, and only
                 subsequent requests may be dispatched over other existing
                 connections. This ensures that in case the server closes the
                 connection when the request is being sent, the browser can
                 decide to silently retry it. Since it is exactly equivalent to
                 regular keep-alive, there should be no side effects.

    - "aggressive" : this mode may be useful in webservices environments where
                 all servers are not necessarily known and where it would be
                 appreciable to deliver most first requests over existing
                 connections. In this case, first requests are only delivered
                 over existing connections that have been reused at least once,
                 proving that the server correctly supports connection reuse.
                 It should only be used when it's sure that the client can
                 retry a failed request once in a while and where the benefit
                 of aggressive connection reuse significantly outweights the
                 downsides of rare connection failures.

    - "always" : this mode is only recommended when the path to the server is
                 known for never breaking existing connections quickly after
                 releasing them. It allows the first request of a session to be
                 sent to an existing connection. This can provide a significant
                 performance increase over the "safe" strategy when the backend
                 is a cache farm, since such components tend to show a
                 consistent behaviour and will benefit from the connection
                 sharing. It is recommended that the "http-keep-alive" timeout
                 remains low in this mode so that no dead connections remain
```

```
                  usable. In most cases, this will lead to the same performance
                  gains as "aggressive" but with more risks. It should only be
                  used when it improves the situation over "aggressive".

  When http connection sharing is enabled, a great care is taken to respect the
  connection properties and compatiblities. Specifically :
    - connections made with "usesrc" followed by a client-dependant value
      ("client", "clientip", "hdr_ip") are marked private and never shared ;

    - connections sent to a server with a TLS SNI extension are marked private
      and are never shared ;

    - connections receiving a status code 401 or 407 expect some authentication
      to be sent in return. Due to certain bogus authentication schemes (such
      as NTLM) relying on the connection, these connections are marked private
      and are never shared ;

  No connection pool is involved, once a session dies, the last idle connection
  it was attached to is deleted at the same time. This ensures that connections
  may not last after all sessions are closed.

  Note: connection reuse improves the accuracy of the "server maxconn" setting,
  because almost no new connection will be established while idle connections
  remain available. This is particularly true with the "always" strategy.

  See also : "option http-keep-alive", "server maxconn"


http-send-name-header [<header>]
  Add the server name to a request. Use the header string given by <header>

  May be used in sections:   defaults | frontend | listen | backend
                               yes    |    no    |   yes  |   yes

  Arguments :

    <header>  The header string to use to send the server name

  The "http-send-name-header" statement causes the name of the target
  server to be added to the headers of an HTTP request.  The name
  is added with the header string proved.

  See also : "server"

id <value>
  Set a persistent ID to a proxy.
  May be used in sections :   defaults | frontend | listen | backend
                                  no   |    yes   |   yes  |   yes
  Arguments : none

  Set a persistent ID for the proxy. This ID must be unique and positive.
  An unused ID will automatically be assigned if unset. The first assigned
  value will be 1. This ID is currently only returned in statistics.


ignore-persist { if | unless } <condition>
  Declare a condition to ignore persistence
  May be used in sections:    defaults | frontend | listen | backend
                                  no   |    yes   |   yes  |   yes

  By default, when cookie persistence is enabled, every requests containing
  the cookie are unconditionally persistent (assuming the target server is up
  and running).

  The "ignore-persist" statement allows one to declare various ACL-based
```

```
  conditions which, when met, will cause a request to ignore persistence.
  This is sometimes useful to load balance requests for static files, which
  often don't require persistence. This can also be used to fully disable
  persistence for a specific User-Agent (for example, some web crawler bots).

  The persistence is ignored when an "if" condition is met, or unless an
  "unless" condition is met.

  See also : "force-persist", "cookie", and section 7 about ACL usage.

load-server-state-from-file { global | local | none }
  Allow seamless reload of HAProxy
  May be used in sections:    defaults | frontend | listen | backend
                                  yes   |    no    |   yes  |   yes

  This directive points HAProxy to a file where server state from previous
  running process has been saved. That way, when starting up, before handling
  traffic, the new process can apply old states to servers exactly has if no
  reload occured. The purpose of the "load-server-state-from-file" directive is
  to tell haproxy which file to use. For now, only 2 arguments to either prevent
  loading state or load states from a file containing all backends and servers.
  The state file can be generated by running the command "show servers state"
  over the stats socket and redirect output.

  The format of the file is versionned and is very specific. To understand it,
  please read the documentation of the "show servers state" command (chapter
  9.2 of Management Guide).

  Arguments:
    global     load the content of the file pointed by the global directive
               named "server-state-file".

    local      load the content of the file pointed by the directive
               "server-state-file-name" if set. If not set, then the backend
               name is used as a file name.

    none       don't load any stat for this backend

  Notes:
    - server's IP address is not updated unless DNS resolution is enabled on
      the server. It means that if a server IP address has been changed using
      the stat socket, this information won't be re-applied after reloading.

    - server's weight is applied from previous running process unless it has
      has changed between previous and new configuration files.

  Example 1:

  Minimal configuration:

    global
     stats socket /tmp/socket
     server-state-file /tmp/server_state

    defaults
     load-server-state-from-file global

    backend bk
     server s1 127.0.0.1:22 check weight 11
     server s2 127.0.0.1:22 check weight 12

  Then one can run :

    socat /tmp/socket - <<< "show servers state" > /tmp/server_state
  Content of the file /tmp/server_state would be like this:

    1
    # <field names skipped for the doc example>
    1 bk 1 s1 127.0.0.1 2 0 11 11 4 6 3 4 6 0 0
    1 bk 2 s2 127.0.0.1 2 0 12 12 4 6 3 4 6 0 0

  Example 2:

  Minimal configuration:

    global
     stats socket /tmp/socket
     server-state-base /etc/haproxy/states

    defaults
     load-server-state-from-file local

    backend bk
     server s1 127.0.0.1:22 check weight 11
     server s2 127.0.0.1:22 check weight 12

  Then one can run :

    socat /tmp/socket - <<< "show servers state bk" > /etc/haproxy/states/bk

  Content of the file /etc/haproxy/states/bk would be like this:

    1
    # <field names skipped for the doc example>
    1 bk 1 s1 127.0.0.1 2 0 11 11 4 6 3 4 6 0 0
    1 bk 2 s2 127.0.0.1 2 0 12 12 4 6 3 4 6 0 0

  See also: "server-state-file", "server-state-file-name", and
  "show servers state"


log global
log <address> [len <length>] <facility> [<level> [<minlevel>]]
no log
  Enable per-instance logging of events and traffic.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes

  Prefix :
    no         should be used when the logger list must be flushed. For example,
               if you don't want to inherit from the default logger list. This
               prefix does not allow arguments.

  Arguments :
    global     should be used when the instance's logging parameters are the
               same as the global ones. This is the most common usage. "global"
               replaces <address>, <facility> and <level> with those of the log
               entries found in the "global" section. Only one "log global"
               statement may be used per instance, and this form takes no other
               parameter.

    <address>  indicates where to send the logs. It takes the same format as
               for the "global" section's logs, and can be one of :

               - An IPv4 address optionally followed by a colon (':') and a UDP
                 port. If no port is specified, 514 is used by default (the
                 standard syslog port).

               - An IPv6 address followed by a colon (':') and optionally a UDP
```

```
                 port. If no port is specified, 514 is used by default (the
                 standard syslog port).

               - A filesystem path to a UNIX domain socket, keeping in mind
                 considerations for chroot (be sure the path is accessible
                 inside the chroot) and uid/gid (be sure the path is
                 appropriately writeable).

              You may want to reference some environment variables in the
              address parameter, see section 2.3 about environment variables.

    <length>   is an optional maximum line length. Log lines larger than this
               value will be truncated before being sent. The reason is that
               syslog servers act differently on log line length. All servers
               support the default value of 1024, but some servers simply drop
               larger lines while others do log them. If a server supports long
               lines, it may make sense to set this value here in order to avoid
               truncating long lines. Similarly, if a server drops long lines,
               it is preferable to truncate them before sending them. Accepted
               values are 80 to 65535 inclusive. The default value of 1024 is
               generally fine for all standard usages. Some specific cases of
               long captures or JSON-formated logs may require larger values.

    <facility> must be one of the 24 standard syslog facilities :

                 kern   user   mail   daemon auth   syslog lpr    news
                 uucp   cron   auth2  ftp    ntp    audit  alert  cron2
                 local0 local1 local2 local3 local4 local5 local6 local7

    <level>    is optional and can be specified to filter outgoing messages. By
               default, all messages are sent. If a level is specified, only
               messages with a severity at least as important as this level
               will be sent. An optional minimum level can be specified. If it
               is set, logs emitted with a more severe level than this one will
               be capped to this level. This is used to avoid sending "emerg"
               messages on all terminals on some default syslog configurations.
               Eight levels are known :

                 emerg  alert  crit   err    warning notice info  debug

  It is important to keep in mind that it is the frontend which decides what to
  log from a connection, and that in case of content switching, the log entries
  from the backend will be ignored. Connections are logged at level "info".

  However, backend log declaration define how and where servers status changes
  will be logged. Level "notice" will be used to indicate a server going up,
  "warning" will be used for termination signals and definitive service
  termination, and "alert" will be used for when a server goes down.

  Note : According to RFC3164, messages are truncated to 1024 bytes before
         being emitted.

  Example :
    log global
    log 127.0.0.1:514 local0 notice         # only send important events
    log 127.0.0.1:514 local0 notice notice  # same but limit output level
    log "${LOCAL_SYSLOG}:514" local0 notice   # send to local server


log-format <string>
  Specifies the log format string to use for traffic logs
  May be used in sections:    defaults | frontend | listen | backend
                                  yes   |    yes   |   yes  |    no

  This directive specifies the log format string that will be used for all logs
  resulting from traffic passing through the frontend using this line. If the
  directive is used in a defaults section, all subsequent frontends will use
  the same log format. Please see section 8.2.4 which covers the log format
  string in depth.

log-format-sd <string>
  Specifies the RFC5424 structured-data log format string
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |    no

  This directive specifies the RFC5424 structured-data log format string that
  will be used for all logs resulting from traffic passing through the frontend
  using this line. If the directive is used in a defaults section, all
  subsequent frontends will use the same log format. Please see section 8.2.4
  which covers the log format string in depth.

  See https://tools.ietf.org/html/rfc5424#section-6.3 for more information
  about the RFC5424 structured-data part.

  Note : This log format string will be used only for loggers that have set
         log format to "rfc5424".

  Example :
    log-format-sd [exampleSDID@1234\ bytes=\"%B\"\ status=\"%ST\"]


log-tag <string>
  Specifies the log tag to use for all outgoing logs
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes

  Sets the tag field in the syslog header to this string. It defaults to the
  log-tag set in the global section, otherwise the program name as launched
  from the command line, which usually is "haproxy". Sometimes it can be useful
  to differentiate between multiple processes running on the same host, or to
  differentiate customer instances running in the same process. In the backend,
  logs about servers up/down will use this tag. As a hint, it can be convenient
  to set a log-tag related to a hosted customer in a defaults section then put
  all the frontends and backends for that customer, then start another customer
  in a new defaults section. See also the global "log-tag" directive.

max-keep-alive-queue <value>
  Set the maximum server queue size for maintaining keep-alive connections
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |     no   |   yes  |   yes

  HTTP keep-alive tries to reuse the same server connection whenever possible,
  but sometimes it can be counter-productive, for example if a server has a lot
  of connections while other ones are idle. This is especially true for static
  servers.

  The purpose of this setting is to set a threshold on the number of queued
  connections at which haproxy stops trying to reuse the same server and prefers
  to find another one. The default value, -1, means there is no limit. A value
  of zero means that keep-alive requests will never be queued. For very close
  servers which can be reached with a low latency and which are not sensible to
  breaking keep-alive, a low value is recommended (eg: local static server can
  use a value of 10 or less). For remote servers suffering from a high latency,
  higher values might be needed to cover for the latency and/or the cost of
  picking a different server.

  Note that this has no impact on responses which are maintained to the same
  server consecutively to a 401 response. They will still go to the same server
  even if they have to be queued.

```

```
  See also : "option http-server-close", "option prefer-last-server", server
             "maxconn" and cookie persistence.


maxconn <conns>
  Fix the maximum number of concurrent connections on a frontend
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments :
    <conns>   is the maximum number of concurrent connections the frontend will
              accept to serve. Excess connections will be queued by the system
              in the socket's listen queue and will be served once a connection
              closes.

  If the system supports it, it can be useful on big sites to raise this limit
  very high so that haproxy manages connection queues, instead of leaving the
  clients with unanswered connection attempts. This value should not exceed the
  global maxconn. Also, keep in mind that a connection contains two buffers
  of 8kB each, as well as some other data resulting in about 17 kB of RAM being
  consumed per established connection. That means that a medium system equipped
  with 1GB of RAM can withstand around 40000-50000 concurrent connections if
  properly tuned.

  Also, when <conns> is set to large values, it is possible that the servers
  are not sized to accept such loads, and for this reason it is generally wise
  to assign them some reasonable connection limits.

  By default, this value is set to 2000.

  See also : "server", global section's "maxconn", "fullconn"


mode { tcp|http|health }
  Set the running mode or protocol of the instance
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    tcp       The instance will work in pure TCP mode. A full-duplex connection
              will be established between clients and servers, and no layer 7
              examination will be performed. This is the default mode. It
              should be used for SSL, SSH, SMTP, ...

    http      The instance will work in HTTP mode. The client request will be
              analyzed in depth before connecting to any server. Any request
              which is not RFC-compliant will be rejected. Layer 7 filtering,
              processing and switching will be possible. This is the mode which
              brings HAProxy most of its value.

    health    The instance will work in "health" mode. It will just reply "OK"
              to incoming connections and close the connection. Alternatively,
              If the "httpchk" option is set, "HTTP/1.0 200 OK" will be sent
              instead. Nothing will be logged in either case. This mode is used
              to reply to external components health checks. This mode is
              deprecated and should not be used anymore as it is possible to do
              the same and even better by combining TCP or HTTP modes with the
              "monitor" keyword.

  When doing content switching, it is mandatory that the frontend and the
  backend are in the same mode (generally HTTP), otherwise the configuration
  will be refused.

  Example :
     defaults http_instances
         mode http

  See also : "monitor", "monitor-net"


monitor fail { if | unless } <condition>
  Add a condition to report a failure to a monitor HTTP request.
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   no
  Arguments :
    if <cond>     the monitor request will fail if the condition is satisfied,
                  and will succeed otherwise. The condition should describe a
                  combined test which must induce a failure if all conditions
                  are met, for instance a low number of servers both in a
                  backend and its backup.

    unless <cond> the monitor request will succeed only if the condition is
                  satisfied, and will fail otherwise. Such a condition may be
                  based on a test on the presence of a minimum number of active
                  servers in a list of backends.

  This statement adds a condition which can force the response to a monitor
  request to report a failure. By default, when an external component queries
  the URI dedicated to monitoring, a 200 response is returned. When one of the
  conditions above is met, haproxy will return 503 instead of 200. This is
  very useful to report a site failure to an external component which may base
  routing advertisements between multiple sites on the availability reported by
  haproxy. In this case, one would rely on an ACL involving the "nbsrv"
  criterion. Note that "monitor fail" only works in HTTP mode. Both status
  messages may be tweaked using "errorfile" or "errorloc" if needed.

  Example:
     frontend www
        mode http
        acl site_dead nbsrv(dynamic) lt 2
        acl site_dead nbsrv(static)  lt 2
        monitor-uri   /site_alive
        monitor fail  if site_dead

  See also : "monitor-net", "monitor-uri", "errorfile", "errorloc"


monitor-net <source>
  Declare a source network which is limited to monitor requests
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments :
    <source>  is the source IPv4 address or network which will only be able to
              get monitor responses to any request. It can be either an IPv4
              address, a host name, or an address followed by a slash ('/')
              followed by a mask.

  In TCP mode, any connection coming from a source matching <source> will cause
  the connection to be immediately closed without any log. This allows another
  equipment to probe the port and verify that it is still listening, without
  forwarding the connection to a remote server.

  In HTTP mode, a connection coming from a source matching <source> will be
  accepted, the following response will be sent without waiting for a request,
  then the connection will be closed : "HTTP/1.0 200 OK". This is normally
  enough for any front-end HTTP probe to detect that the service is UP and
  running without forwarding the request to a backend server. Note that this
  response is sent in raw format, without any transformation. This is important
  as it means that it will not be SSL-encrypted on SSL listeners.

  Monitor requests are processed very early, just after tcp-request connection
  ACLs which are the only ones able to block them. These connections are short
```

```
  lived and never wait for any data from the client. They cannot be logged, and
  it is the intended purpose. They are only used to report HAProxy's health to
  an upper component, nothing more. Please note that "monitor fail" rules do
  not apply to connections intercepted by "monitor-net".

  Last, please note that only one "monitor-net" statement can be specified in
  a frontend. If more than one is found, only the last one will be considered.

  Example :
    # addresses .252 and .253 are just probing us.
    frontend www
        monitor-net 192.168.0.252/31

  See also : "monitor fail", "monitor-uri"


monitor-uri <uri>
  Intercept a URI used by external components' monitor requests
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments :
    <uri>     is the exact URI which we want to intercept to return HAProxy's
              health status instead of forwarding the request.

  When an HTTP request referencing <uri> will be received on a frontend,
  HAProxy will not forward it nor log it, but instead will return either
  "HTTP/1.0 200 OK" or "HTTP/1.0 503 Service unavailable", depending on failure
  conditions defined with "monitor fail". This is normally enough for any
  front-end HTTP probe to detect that the service is UP and running without
  forwarding the request to a backend server. Note that the HTTP method, the
  version and all headers are ignored, but the request must at least be valid
  at the HTTP level. This keyword may only be used with an HTTP-mode frontend.

  Monitor requests are processed very early. It is not possible to block nor
  divert them using ACLs. They cannot be logged either, and it is the intended
  purpose. They are only used to report HAProxy's health to an upper component,
  nothing more. However, it is possible to add any number of conditions using
  "monitor fail" and ACLs so that the result can be adjusted to whatever check
  can be imagined (most often the number of available servers in a backend).

  Example :
    # Use /haproxy_test to report haproxy's status
    frontend www
        mode http
        monitor-uri /haproxy_test

  See also : "monitor fail", "monitor-net"


option abortonclose
no option abortonclose
  Enable or disable early dropping of aborted requests pending in queues.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |     no   |   yes  |   yes
  Arguments : none

  In presence of very high loads, the servers will take some time to respond.
  The per-instance connection queue will inflate, and the response time will
  increase respective to the size of the queue times the average per-session
  response time. When clients will wait for more than a few seconds, they will
  often hit the "STOP" button on their browser, leaving a useless request in
  the queue, and slowing down other users, and the servers as well, because the
  request will eventually be served, then aborted at the first error
  encountered while delivering the response.
  As there is no way to distinguish between a full STOP and a simple output
  close on the client side, HTTP agents should be conservative and consider
  that the client might only have closed its output channel while waiting for
  the response. However, this introduces risks of congestion when lots of users
  do the same, and is completely useless nowadays because probably no client at
  all will close the session while waiting for the response. Some HTTP agents
  support this behaviour (Squid, Apache, HAProxy), and others do not (TUX, most
  hardware-based load balancers). So the probability for a closed input channel
  to represent a user hitting the "STOP" button is close to 100%, and the risk
  of being the single component to break rare but valid traffic is extremely
  low, which adds to the temptation to be able to abort a session early while
  still not served and not pollute the servers.

  In HAProxy, the user can choose the desired behaviour using the option
  "abortonclose". By default (without the option) the behaviour is HTTP
  compliant and aborted requests will be served. But when the option is
  specified, a session with an incoming channel closed will be aborted while
  it is still possible, either pending in the queue for a connection slot, or
  during the connection establishment if the server has not yet acknowledged
  the connection request. This considerably reduces the queue size and the load
  on saturated servers when users are tempted to click on STOP, which in turn
  reduces the response time for other users.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "timeout queue" and server's "maxconn" and "maxqueue" parameters


option accept-invalid-http-request
no option accept-invalid-http-request
  Enable or disable relaxing of HTTP request parsing
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments : none

  By default, HAProxy complies with RFC7230 in terms of message parsing. This
  means that invalid characters in header names are not permitted and cause an
  error to be returned to the client. This is the desired behaviour as such
  forbidden characters are essentially used to build attacks exploiting server
  weaknesses, and bypass security filtering. Sometimes, a buggy browser or
  server will emit invalid header names for whatever reason (configuration,
  implementation) and the issue will not be immediately fixed. In such a case,
  it is possible to relax HAProxy's header name parser to accept any character
  even if that does not make sense, by specifying this option. Similarly, the
  list of characters allowed to appear in a URI is well defined by RFC3986, and
  chars 0-31, 32 (space), 34 ('"'), 60 ('<'), 62 ('>'), 92 ('\'), 94 ('^'), 96
  ('`'), 123 ('{'), 124 ('|'), 125 ('}'), 127 (delete) and anything above are
  not allowed at all. Haproxy always blocks a number of them (0..32, 127). The
  remaining ones are blocked by default unless this option is enabled. This
  option also relaxes the test on the HTTP version, it allows HTTP/0.9 requests
  to pass through (no version specified) and multiple digits for both the major
  and the minor version.

  This option should never be enabled by default as it hides application bugs
  and open security breaches. It should only be deployed after a problem has
  been confirmed.

  When this option is enabled, erroneous header names will still be accepted in
  requests, but the complete request will be captured in order to permit later
  analysis using the "show errors" request on the UNIX stats socket. Similarly,
  requests containing invalid chars in the URI part will be logged. Doing this
  also helps confirming that the issue has been solved.

  If this option has been enabled in a "defaults" section, it can be disabled
```

```
  in a specific instance by prepending the "no" keyword before it.

  See also : "option accept-invalid-http-response" and "show errors" on the
             stats socket.


option accept-invalid-http-response
no option accept-invalid-http-response
  Enable or disable relaxing of HTTP response parsing
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |     no   |   yes  |   yes
  Arguments : none

  By default, HAProxy complies with RFC7230 in terms of message parsing. This
  means that invalid characters in header names are not permitted and cause an
  error to be returned to the client. This is the desired behaviour as such
  forbidden characters are essentially used to build attacks exploiting server
  weaknesses, and bypass security filtering. Sometimes, a buggy browser or
  server will emit invalid header names for whatever reason (configuration,
  implementation) and the issue will not be immediately fixed. In such a case,
  it is possible to relax HAProxy's header name parser to accept any character
  even if that does not make sense, by specifying this option. This option also
  relaxes the test on the HTTP version format, it allows multiple digits for
  both the major and the minor version.

  This option should never be enabled by default as it hides application bugs
  and open security breaches. It should only be deployed after a problem has
  been confirmed.

  When this option is enabled, erroneous header names will still be accepted in
  responses, but the complete response will be captured in order to permit
  later analysis using the "show errors" request on the UNIX stats socket.
  Doing this also helps confirming that the issue has been solved.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option accept-invalid-http-request" and "show errors" on the
             stats socket.


option allbackups
no option allbackups
  Use either all backup servers at a time or only the first one
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |     no   |   yes  |   yes
  Arguments : none

  By default, the first operational backup server gets all traffic when normal
  servers are all down. Sometimes, it may be preferred to use multiple backups
  at once, because one will not be enough. When "option allbackups" is enabled,
  the load balancing will be performed among all backup servers when all normal
  ones are unavailable. The same load balancing algorithm will be used and the
  servers' weights will be respected. Thus, there will not be any priority
  order between the backup servers anymore.

  This option is mostly used with static server farms dedicated to return a
  "sorry" page when an application is completely offline.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.


option checkcache
no option checkcache
```

```
  Analyze all server responses and block responses with cacheable cookies
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  Some high-level frameworks set application cookies everywhere and do not
  always let enough control to the developer to manage how the responses should
  be cached. When a session cookie is returned on a cacheable object, there is a
  high risk of session crossing or stealing between users traversing the same
  caches. In some situations, it is better to block the response than to let
  some sensitive session information go in the wild.

  The option "checkcache" enables deep inspection of all server responses for
  strict compliance with HTTP specification in terms of cacheability. It
  carefully checks "Cache-control", "Pragma" and "Set-cookie" headers in server
  response to check if there's a risk of caching a cookie on a client-side
  proxy. When this option is enabled, the only responses which can be delivered
  to the client are :
    - all those without "Set-Cookie" header ;
    - all those with a return code other than 200, 203, 206, 300, 301, 410,
      provided that the server has not set a "Cache-control: public" header ;
    - all those that come from a POST request, provided that the server has not
      set a 'Cache-Control: public' header ;
    - those with a 'Pragma: no-cache' header
    - those with a 'Cache-control: private' header
    - those with a 'Cache-control: no-store' header
    - those with a 'Cache-control: max-age=0' header
    - those with a 'Cache-control: s-maxage=0' header
    - those with a 'Cache-control: no-cache' header
    - those with a 'Cache-control: no-cache="set-cookie"' header
    - those with a 'Cache-control: no-cache="set-cookie,' header
      (allowing other fields after set-cookie)

  If a response doesn't respect these requirements, then it will be blocked
  just as if it was from an "rspdeny" filter, with an "HTTP 502 bad gateway".
  The session state shows "PH--" meaning that the proxy blocked the response
  during headers processing. Additionally, an alert will be sent in the logs so
  that admins are informed that there's something to be fixed.

  Due to the high impact on the application, the application should be tested
  in depth with the option enabled before going to production. It is also a
  good practice to always activate it during tests, even if it is not used in
  production, as it will report potentially dangerous application behaviours.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.


option clitcpka
no option clitcpka
  Enable or disable the sending of TCP keepalive packets on the client side
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments : none

  When there is a firewall or any session-aware component between a client and
  a server, and when the protocol involves very long sessions with long idle
  periods (eg: remote desktops), there is a risk that one of the intermediate
  components decides to expire a session which has remained idle for too long.

  Enabling socket-level TCP keep-alives makes the system regularly send packets
  to the other end of the connection, leaving it active. The delay between
  keep-alive probes is controlled by the system only and depends both on the
  operating system and its tuning parameters.

```

```
  It is important to understand that keep-alive packets are neither emitted nor
  received at the application level. It is only the network stacks which sees
  them. For this reason, even if one side of the proxy already uses keep-alives
  to maintain its connection alive, those keep-alive packets will not be
  forwarded to the other side of the proxy.

  Please note that this has nothing to do with HTTP keep-alive.

  Using option "clitcpka" enables the emission of TCP keep-alive probes on the
  client side of a connection, which should help when session expirations are
  noticed between HAProxy and a client.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option srvtcpka", "option tcpka"


option contstats
  Enable continuous traffic statistics updates
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments : none

  By default, counters used for statistics calculation are incremented
  only when a session finishes. It works quite well when serving small
  objects, but with big ones (for example large images or archives) or
  with A/V streaming, a graph generated from haproxy counters looks like
  a hedgehog. With this option enabled counters get incremented continuously,
  during a whole session. Recounting touches a hotpath directly so
  it is not enabled by default, as it has small performance impact (~0.5%).


option dontlog-normal
no option dontlog-normal
  Enable or disable logging of normal, successful connections
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments : none

  There are large sites dealing with several thousand connections per second
  and for which logging is a major pain. Some of them are even forced to turn
  logs off and cannot debug production issues. Setting this option ensures that
  normal connections, those which experience no error, no timeout, no retry nor
  redispatch, will not be logged. This leaves disk space for anomalies. In HTTP
  mode, the response status code is checked and return codes 5xx will still be
  logged.

  It is strongly discouraged to use this option as most of the time, the key to
  complex issues is in the normal logs which will not be logged here. If you
  need to separate logs, see the "log-separate-errors" option instead.

  See also : "log", "dontlognull", "log-separate-errors" and section 8 about
             logging.


option dontlognull
no option dontlognull
  Enable or disable logging of null connections
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments : none

  In certain environments, there are components which will regularly connect to
  various systems to ensure that they are still alive. It can be the case from
```

another load balancer as well as from monitoring systems. By default, even a simple port probe or scan will produce a log. If those connections pollute the logs too much, it is possible to enable option "dontlognull" to indicate that a connection on which no data has been transferred will not be logged, which typically corresponds to those probes. Note that errors will still be returned to the client and accounted for in the stats. If this is not what is desired, option http-ignore-probes can be used instead.

It is generally recommended not to use this option in uncontrolled environments (eg: internet), otherwise scans and other malicious activities would not be logged.

If this option has been enabled in a "defaults" section, it can be disabled in a specific instance by prepending the "no" keyword before it.

```
  See also : "log", "http-ignore-probes", "monitor-net", "monitor-uri", and
             section 8 about logging.
```

option forceclose
no option forceclose

Enable or disable active connection closing after response is transferred.

```
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none
```

Some HTTP servers do not necessarily close the connections when they receive the "Connection: close" set by "option httpclose", and if the client does not close either, then the connection remains open till the timeout expires. This causes high number of simultaneous connections on the servers and shows high global session times in the logs.

When this happens, it is possible to use "option forceclose". It will actively close the outgoing server channel as soon as the server has finished to respond and release some resources earlier than with "option httpclose".

This option may also be combined with "option http-pretend-keepalive", which will disable sending of the "Connection: close" header, but will still cause the connection to be closed once the whole response is received.

This option disables and replaces any previous "option httpclose", "option http-server-close", "option http-keep-alive", or "option http-tunnel".

If this option has been enabled in a "defaults" section, it can be disabled in a specific instance by prepending the "no" keyword before it.

See also : "option httpclose" and "option http-pretend-keepalive"

option forwardfor [except <network>] [header <name>] [if-none]

Enable insertion of the X-Forwarded-For header to requests sent to servers

```
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <network> is an optional argument used to disable this option for sources
              matching <network>
    <name>    an optional argument to specify a different "X-Forwarded-For"
              header name.
```

Since HAProxy works in reverse-proxy mode, the servers see its IP address as their client address. This is sometimes annoying when the client's IP address is expected in server logs. To solve this problem, the well-known HTTP header "X-Forwarded-For" may be added by HAProxy to all requests sent to the server. This header contains a value representing the client's IP address. Since this header is always appended at the end of the existing header list, the server

must be configured to always use the last occurrence of this header only. See the server's manual to find how to enable use of this standard header. Note that only the last occurrence of the header must be used, since it is really possible that the client has already brought one.

The keyword "header" may be used to supply a different header name to replace the default "X-Forwarded-For". This can be useful where you might already have a "X-Forwarded-For" header from a different application (eg: stunnel), and you need preserve it. Also if your backend server doesn't use the "X-Forwarded-For" header and requires different one (eg: Zeus Web Servers require "X-Cluster-Client-IP").

Sometimes, a same HAProxy instance may be shared between a direct client access and a reverse-proxy access (for instance when an SSL reverse-proxy is used to decrypt HTTPS traffic). It is possible to disable the addition of the header for a known source address or network by adding the "except" keyword followed by the network address. In this case, any source IP matching the network will not cause an addition of this header. Most common uses are with private networks or 127.0.0.1.

Alternatively, the keyword "if-none" states that the header will only be added if it is not present. This should only be used in perfectly trusted environment, as this might cause a security issue if headers reaching haproxy are under the control of the end-user.

This option may be specified either in the frontend or in the backend. If at least one of them uses it, the header will be added. Note that the backend's setting of the header subargument takes precedence over the frontend's if both are defined. In the case of the "if-none" argument, if at least one of the frontend or the backend does not specify it, it wants the addition to be mandatory, so it wins.

```
  Examples :
    # Public HTTP address also used by stunnel on the same machine
    frontend www
        mode http
        option forwardfor except 127.0.0.1  # stunnel already adds the header

    # Those servers want the IP Address in X-Client
    backend www
        mode http
        option forwardfor header X-Client

  See also : "option httpclose", "option http-server-close",
             "option forceclose", "option http-keep-alive"
```

option http-buffer-request
no option http-buffer-request

Enable or disable waiting for whole HTTP request body before proceeding

```
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none
```

It is sometimes desirable to wait for the body of an HTTP request before taking a decision. This is what is being done by "balance url_param" for example. The first use case is to buffer requests from slow clients before connecting to the server. Another use case consists in taking the routing decision based on the request body's contents. This option placed in a frontend or backend forces the HTTP processing to wait until either the whole body is received, or the request buffer is full, or the first chunk is complete in case of chunked encoding. It can have undesired side effects with some applications abusing HTTP by expecting unbufferred transmissions between the frontend and the backend, so this should definitely not be used by default.

See also : "option http-no-delay", "timeout http-request"

```
option http-ignore-probes
no option http-ignore-probes
  Enable or disable logging of null connections and request timeouts
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments : none
```

Recently some browsers started to implement a "pre-connect" feature consisting in speculatively connecting to some recently visited web sites just in case the user would like to visit them. This results in many connections being established to web sites, which end up in 408 Request Timeout if the timeout strikes first, or 400 Bad Request when the browser decides to close them first. These ones pollute the log and feed the error counters. There was already "option dontlognull" but it's insufficient in this case. Instead, this option does the following things :

- prevent any 400/408 message from being sent to the client if nothing was received over a connection before it was closed ;
- prevent any log from being emitted in this situation ;
- prevent any error counter from being incremented

That way the empty connection is silently ignored. Note that it is better not to use this unless it is clear that it is needed, because it will hide real problems. The most common reason for not receiving a request and seeing a 408 is due to an MTU inconsistency between the client and an intermediary element such as a VPN, which blocks too large packets. These issues are generally seen with POST requests as well as GET with large cookies. The logs are often the only way to detect them.

If this option has been enabled in a "defaults" section, it can be disabled in a specific instance by prepending the "no" keyword before it.

See also : "log", "dontlognull", "errorfile", and section 8 about logging.

```
option http-keep-alive
no option http-keep-alive
  Enable or disable HTTP keep-alive from client to server
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none
```

By default HAProxy operates in keep-alive mode with regards to persistent connections: for each connection it processes each request and response, and leaves the connection idle on both sides between the end of a response and the start of a new request. This mode may be changed by several options such as "option http-server-close", "option forceclose", "option httpclose" or "option http-tunnel". This option allows to set back the keep-alive mode, which can be useful when another mode was used in a defaults section.

Setting "option http-keep-alive" enables HTTP keep-alive mode on the client- and server- sides. This provides the lowest latency on the client side (slow network) and the fastest session reuse on the server side at the expense of maintaining idle connections to the servers. In general, it is possible with this option to achieve approximately twice the request rate that the "http-server-close" option achieves on small objects. There are mainly two situations where this option may be useful :

- when the server is non-HTTP compliant and authenticates the connection instead of requests (eg: NTLM authentication)

- when the cost of establishing the connection to the server is significant compared to the cost of retrieving the associated object from the server.

This last case can happen when the server is a fast static server of cache. In this case, the server will need to be properly tuned to support high enough connection counts because connections will last until the client sends another request.

If the client request has to go to another backend or another server due to content switching or the load balancing algorithm, the idle connection will immediately be closed and a new one re-opened. Option "prefer-last-server" is available to try optimize server selection so that if the server currently attached to an idle connection is usable, it will be used.

In general it is preferred to use "option http-server-close" with application servers, and some static servers might benefit from "option http-keep-alive".

At the moment, logs will not indicate whether requests came from the same session or not. The accept date reported in the logs corresponds to the end of the previous request, and the request time corresponds to the time spent waiting for a new request. The keep-alive request time is still bound to the timeout defined by "timeout http-keep-alive" or "timeout http-request" if not set.

This option disables and replaces any previous "option httpclose", "option http-server-close", "option forceclose" or "option http-tunnel". When backend and frontend options differ, all of these 4 options have precedence over "option http-keep-alive".

```
See also : "option forceclose", "option http-server-close",
           "option prefer-last-server", "option http-pretend-keepalive",
           "option httpclose", and "1.1. The HTTP transaction model".
```

```
option http-no-delay
no option http-no-delay
  Instruct the system to favor low interactive delays over performance in HTTP
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none
```

In HTTP, each payload is unidirectional and has no notion of interactivity. Any agent is expected to queue data somewhat for a reasonably low delay. There are some very rare server-to-server applications that abuse the HTTP protocol and expect the payload phase to be highly interactive, with many interleaved data chunks in both directions within a single request. This is absolutely not supported by the HTTP specification and will not work across most proxies or servers. When such applications attempt to do this through haproxy, it works but they will experience high delays due to the network optimizations which favor performance by instructing the system to wait for enough data to be available in order to only send full packets. Typical delays are around 200 ms per round trip. Note that this only happens with abnormal uses. Normal uses such as CONNECT requests nor WebSockets are not affected.

When "option http-no-delay" is present in either the frontend or the backend used by a connection, all such optimizations will be disabled in order to make the exchanges as fast as possible. Of course this offers no guarantee on the functionality, as it may break at any other place. But if it works via HAProxy, it will work as fast as possible. This option should never be used by default, and should never be used at all unless such a buggy application is discovered. The impact of using this option is an increase of bandwidth usage and CPU usage, which may significantly lower performance in high latency environments.

See also : "option http-buffer-request"

```
option http-pretend-keepalive
no option http-pretend-keepalive
  Define whether haproxy will announce keepalive to the server or not
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  When running with "option http-server-close" or "option forceclose", haproxy
  adds a "Connection: close" header to the request forwarded to the server.
  Unfortunately, when some servers see this header, they automatically refrain
  from using the chunked encoding for responses of unknown length, while this
  is totally unrelated. The immediate effect is that this prevents haproxy from
  maintaining the client connection alive. A second effect is that a client or
  a cache could receive an incomplete response without being aware of it, and
  consider the response complete.

  By setting "option http-pretend-keepalive", haproxy will make the server
  believe it will keep the connection alive. The server will then not fall back
  to the abnormal undesired above. When haproxy gets the whole response, it
  will close the connection with the server just as it would do with the
  "forceclose" option. That way the client gets a normal response and the
  connection is correctly closed on the server side.

  It is recommended not to enable this option by default, because most servers
  will more efficiently close the connection themselves after the last packet,
  and release its buffers slightly earlier. Also, the added packet on the
  network could slightly reduce the overall peak performance. However it is
  worth noting that when this option is enabled, haproxy will have slightly
  less work to do. So if haproxy is the bottleneck on the whole architecture,
  enabling this option might save a few CPU cycles.

  This option may be set both in a frontend and in a backend. It is enabled if
  at least one of the frontend or backend holding a connection has it enabled.
  This option may be combined with "option httpclose", which will cause
  keepalive to be announced to the server and close to be announced to the
  client. This practice is discouraged though.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option forceclose", "option http-server-close", and
             "option http-keep-alive"


option http-server-close
no option http-server-close
  Enable or disable HTTP connection closing on the server side
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  By default HAProxy operates in keep-alive mode with regards to persistent
  connections: for each connection it processes each request and response, and
  leaves the connection idle on both sides between the end of a response and
  the start of a new request. This mode may be changed by several options such
  as "option http-server-close", "option forceclose", "option httpclose" or
  "option http-tunnel". Setting "option http-server-close" enables HTTP
  connection-close mode on the server side while keeping the ability to support
  HTTP keep-alive and pipelining on the client side.  This provides the lowest
  latency on the client side (slow network) and the fastest session reuse on
  the server side to save server resources, similarly to "option forceclose".
  It also permits non-keepalive capable servers to be served in keep-alive mode
  to the clients if they conform to the requirements of RFC2616. Please note
```

```
  that some servers do not always conform to those requirements when they see
  "Connection: close" in the request. The effect will be that keep-alive will
  never be used. A workaround consists in enabling "option
  http-pretend-keepalive".

  At the moment, logs will not indicate whether requests came from the same
  session or not. The accept date reported in the logs corresponds to the end
  of the previous request, and the request time corresponds to the time spent
  waiting for a new request. The keep-alive request time is still bound to the
  timeout defined by "timeout http-keep-alive" or "timeout http-request" if
  not set.

  This option may be set both in a frontend and in a backend. It is enabled if
  at least one of the frontend or backend holding a connection has it enabled.
  It disables and replaces any previous "option httpclose", "option forceclose",
  "option http-tunnel" or "option http-keep-alive". Please check section 4
  ("Proxies") to see how this option combines with others when frontend and
  backend options differ.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option forceclose", "option http-pretend-keepalive",
             "option httpclose", "option http-keep-alive", and
             "1.1. The HTTP transaction model".


option http-tunnel
no option http-tunnel
  Disable or enable HTTP connection processing after first transaction
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  By default HAProxy operates in keep-alive mode with regards to persistent
  connections: for each connection it processes each request and response, and
  leaves the connection idle on both sides between the end of a response and
  the start of a new request. This mode may be changed by several options such
  as "option http-server-close", "option forceclose", "option httpclose" or
  "option http-tunnel".

  Option "http-tunnel" disables any HTTP processing past the first request and
  the first response. This is the mode which was used by default in versions
  1.0 to 1.5-dev21. It is the mode with the lowest processing overhead, which
  is normally not needed anymore unless in very specific cases such as when
  using an in-house protocol that looks like HTTP but is not compatible, or
  just to log one request per client in order to reduce log size. Note that
  everything which works at the HTTP level, including header parsing/addition,
  cookie processing or content switching will only work for the first request
  and will be ignored after the first response.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option forceclose", "option http-server-close",
             "option httpclose", "option http-keep-alive", and
             "1.1. The HTTP transaction model".


option http-use-proxy-header
no option http-use-proxy-header
  Make use of non-standard Proxy-Connection header instead of Connection
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments : none

  While RFC2616 explicitly states that HTTP/1.1 agents must use the
  Connection header to indicate their wish of persistent or non-persistent
  connections, both browsers and proxies ignore this header for proxied
  connections and make use of the undocumented, non-standard Proxy-Connection
  header instead. The issue begins when trying to put a load balancer between
  browsers and such proxies, because there will be a difference between what
  haproxy understands and what the client and the proxy agree on.

  By setting this option in a frontend, haproxy can automatically switch to use
  that non-standard header if it sees proxied requests. A proxied request is
  defined here as one where the URI begins with neither a '/' nor a '*'. The
  choice of header only affects requests passing through proxies making use of
  one of the "httpclose", "forceclose" and "http-server-close" options. Note
  that this option can only be specified in a frontend and will affect the
  request along its whole life.

  Also, when this option is set, a request which requires authentication will
  automatically switch to use proxy authentication headers if it is itself a
  proxied request. That makes it possible to check or enforce authentication in
  front of an existing proxy.

  This option should normally never be used, except in front of a proxy.

  See also : "option httpclose", "option forceclose" and "option
             http-server-close".


option httpchk
option httpchk <uri>
option httpchk <method> <uri>
option httpchk <method> <uri> <version>
  Enable HTTP protocol to check on the servers health
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <method>  is the optional HTTP method used with the requests. When not set,
              the "OPTIONS" method is used, as it generally requires low server
              processing and is easy to filter out from the logs. Any method
              may be used, though it is not recommended to invent non-standard
              ones.

    <uri>     is the URI referenced in the HTTP requests. It defaults to " / "
              which is accessible by default on almost any server, but may be
              changed to any other URI. Query strings are permitted.

    <version> is the optional HTTP version string. It defaults to "HTTP/1.0"
              but some servers might behave incorrectly in HTTP 1.0, so turning
              it to HTTP/1.1 may sometimes help. Note that the Host field is
              mandatory in HTTP/1.1, and as a trick, it is possible to pass it
              after "\r\n" following the version string.

  By default, server health checks only consist in trying to establish a TCP
  connection. When "option httpchk" is specified, a complete HTTP request is
  sent once the TCP connection is established, and responses 2xx and 3xx are
  considered valid, while all other ones indicate a server failure, including
  the lack of any response.

  The port and interval are specified in the server configuration.

  This option does not necessarily require an HTTP backend, it also works with
  plain TCP backends. This is particularly useful to check simple scripts bound
  to some dedicated ports using the inetd daemon.

  Examples :
```

```
        # Relay HTTPS traffic to Apache instance and check service availability
        # using HTTP request "OPTIONS * HTTP/1.1" on port 80.
        backend https_relay
            mode tcp
            option httpchk OPTIONS * HTTP/1.1\r\nHost:\ www
            server apache1 192.168.1.1:443 check port 80

  See also : "option ssl-hello-chk", "option smtpchk", "option mysql-check",
             "option pgsql-check", "http-check" and the "check", "port" and
             "inter" server options.


option httpclose
no option httpclose
  Enable or disable passive HTTP connection closing
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  By default HAProxy operates in keep-alive mode with regards to persistent
  connections: for each connection it processes each request and response, and
  leaves the connection idle on both sides between the end of a response and
  the start of a new request. This mode may be changed by several options such
  as "option http-server-close", "option forceclose", "option httpclose" or
  "option http-tunnel".

  If "option httpclose" is set, HAProxy will work in HTTP tunnel mode and check
  if a "Connection: close" header is already set in each direction, and will
  add one if missing. Each end should react to this by actively closing the TCP
  connection after each transfer, thus resulting in a switch to the HTTP close
  mode. Any "Connection" header different from "close" will also be removed.
  Note that this option is deprecated since what it does is very cheap but not
  reliable. Using "option http-server-close" or "option forceclose" is strongly
  recommended instead.

  It seldom happens that some servers incorrectly ignore this header and do not
  close the connection even though they reply "Connection: close". For this
  reason, they are not compatible with older HTTP 1.0 browsers. If this happens
  it is possible to use the "option forceclose" which actively closes the
  request connection once the server responds. Option "forceclose" also
  releases the server connection earlier because it does not have to wait for
  the client to acknowledge it.

  This option may be set both in a frontend and in a backend. It is enabled if
  at least one of the frontend or backend holding a connection has it enabled.
  It disables and replaces any previous "option http-server-close",
  "option forceclose", "option http-keep-alive" or "option http-tunnel". Please
  check section 4 ("Proxies") to see how this option combines with others when
  frontend and backend options differ.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option forceclose", "option http-server-close" and
             "1.1. The HTTP transaction model".


option httplog [ clf ]
  Enable logging of HTTP request, session state and timers
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    clf       if the "clf" argument is added, then the output format will be
              the CLF format instead of HAProxy's default HTTP format. You can
              use this when you need to feed HAProxy's logs through a specific
              log analyser which only support the CLF format and which is not
              extensible.

  By default, the log output format is very poor, as it only contains the
  source and destination addresses, and the instance name. By specifying
  "option httplog", each log line turns into a much richer format including,
  but not limited to, the HTTP request, the connection timers, the session
  status, the connections numbers, the captured headers and cookies, the
  frontend, backend and server name, and of course the source address and
  ports.

  This option may be set either in the frontend or the backend.

  Specifying only "option httplog" will automatically clear the 'clf' mode
  if it was set by default.

  See also :  section 8 about logging.


option http_proxy
no option http_proxy
  Enable or disable plain HTTP proxy mode
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  It sometimes happens that people need a pure HTTP proxy which understands
  basic proxy requests without caching nor any fancy feature. In this case,
  it may be worth setting up an HAProxy instance with the "option http_proxy"
  set. In this mode, no server is declared, and the connection is forwarded to
  the IP address and port found in the URL after the "http://" scheme.

  No host address resolution is performed, so this only works when pure IP
  addresses are passed. Since this option's usage perimeter is rather limited,
  it will probably be used only by experts who know they need exactly it. Last,
  if the clients are susceptible of sending keep-alive requests, it will be
  needed to add "option httpclose" to ensure that all requests will correctly
  be analyzed.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  Example :
    # this backend understands HTTP proxy requests and forwards them directly.
    backend direct_forward
        option httpclose
        option http_proxy

  See also : "option httpclose"


option independent-streams
no option independent-streams
  Enable or disable independent timeout processing for both directions
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |  yes
  Arguments : none

  By default, when data is sent over a socket, both the write timeout and the
  read timeout for that socket are refreshed, because we consider that there is
  activity on that socket, and we have no other means of guessing if we should
  receive data or not.

  While this default behaviour is desirable for almost all applications, there
  exists a situation where it is desirable to disable it, and only refresh the
```

```
  read timeout if there are incoming data. This happens on sessions with large
  timeouts and low amounts of exchanged data such as telnet session. If the
  server suddenly disappears, the output data accumulates in the system's
  socket buffers, both timeouts are correctly refreshed, and there is no way
  to know the server does not receive them, so we don't timeout. However, when
  the underlying protocol always echoes sent data, it would be enough by itself
  to detect the issue using the read timeout. Note that this problem does not
  happen with more verbose protocols because data won't accumulate long in the
  socket buffers.

  When this option is set on the frontend, it will disable read timeout updates
  on data sent to the client. There probably is little use of this case. When
  the option is set on the backend, it will disable read timeout updates on
  data sent to the server. Doing so will typically break large HTTP posts from
  slow lines, so use it with caution.

  Note: older versions used to call this setting "option independant-streams"
        with a spelling mistake. This spelling is still supported but
        deprecated.

  See also : "timeout client", "timeout server" and "timeout tunnel"


option ldap-check
  Use LDAPv3 health checks for server testing
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  It is possible to test that the server correctly talks LDAPv3 instead of just
  testing that it accepts the TCP connection. When this option is set, an
  LDAPv3 anonymous simple bind message is sent to the server, and the response
  is analyzed to find an LDAPv3 bind response message.

  The server is considered valid only when the LDAP response contains success
  resultCode (http://tools.ietf.org/html/rfc4511#section-4.1.9).

  Logging of bind requests is server dependent see your documentation how to
  configure it.

  Example :
        option ldap-check

  See also : "option httpchk"


option external-check
  Use external processes for server health checks
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes

  It is possible to test the health of a server using an external command.
  This is achieved by running the executable set using "external-check
  command".

  Requires the "external-check" global to be set.

  See also : "external-check", "external-check command", "external-check path"


option log-health-checks
no option log-health-checks
  Enable or disable logging of health checks status updates
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |  yes
  Arguments : none

  By default, failed health check are logged if server is UP and successful
  health checks are logged if server is DOWN, so the amount of additional
  information is limited.

  When this option is enabled, any change of the health check status or to
  the server's health will be logged, so that it becomes possible to know
  that a server was failing occasional checks before crashing, or exactly when
  it failed to respond a valid HTTP status, then when the port started to
  reject connections, then when the server stopped responding at all.

  Note that status changes not caused by health checks (eg: enable/disable on
  the CLI) are intentionally not logged by this option.

  See also: "option httpchk", "option ldap-check", "option mysql-check",
            "option pgsql-check", "option redis-check", "option smtpchk",
            "option tcp-check", "log" and section 8 about logging.


option log-separate-errors
no option log-separate-errors
  Change log level for non-completely successful connections
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments : none

  Sometimes looking for errors in logs is not easy. This option makes haproxy
  raise the level of logs containing potentially interesting information such
  as errors, timeouts, retries, redispatches, or HTTP status codes 5xx. The
  level changes from "info" to "err". This makes it possible to log them
  separately to a different file with most syslog daemons. Be careful not to
  remove them from the original file, otherwise you would lose ordering which
  provides very important information.

  Using this option, large sites dealing with several thousand connections per
  second may log normal traffic to a rotating buffer and only archive smaller
  error logs.

  See also : "log", "dontlognull", "dontlog-normal" and section 8 about
             logging.


option logasap
no option logasap
  Enable or disable early logging of HTTP requests
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments : none

  By default, HTTP requests are logged upon termination so that the total
  transfer time and the number of bytes appear in the logs. When large objects
  are being transferred, it may take a while before the request appears in the
  logs. Using "option logasap", the request gets logged as soon as the server
  sends the complete headers. The only missing information in the logs will be
  the total number of bytes which will indicate everything except the amount
  of data transferred, and the total time which will not take the transfer
  time into account. In such a situation, it's a good practice to capture the
  "Content-Length" response header so that the logs at least indicate how many
  bytes are expected to be transferred.

  Examples :
      listen http_proxy 0.0.0.0:80
          mode http
          option httplog
```

```
          option logasap
          log 192.168.2.200 local3

    >>> Feb  6 12:14:14 localhost \
          haproxy[14389]: 10.0.1.2:33317 [06/Feb/2009:12:14:14.655] http-in \
          static/srv1 9/10/7/14/+30 200 +243 - - ---- 3/1/1/1/0 1/0 \
          "GET /image.iso HTTP/1.0"

  See also : "option httplog", "capture response header", and section 8 about
             logging.


option mysql-check [ user <username> [ post-41 ] ]
  Use MySQL health checks for server testing
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <username> This is the username which will be used when connecting to MySQL
               server.
    post-41    Send post v4.1 client compatible checks

  If you specify a username, the check consists of sending two MySQL packet,
  one Client Authentication packet, and one QUIT packet, to correctly close
  MySQL session. We then parse the MySQL Handshake Initialisation packet and/or
  Error packet. It is a basic but useful test which does not produce error nor
  aborted connect on the server. However, it requires adding an authorization
  in the MySQL table, like this :

      USE mysql;
      INSERT INTO user (Host,User) values ('<ip_of_haproxy>','<username>');
      FLUSH PRIVILEGES;

  If you don't specify a username (it is deprecated and not recommended), the
  check only consists in parsing the Mysql Handshake Initialisation packet or
  Error packet, we don't send anything in this mode. It was reported that it
  can generate lockout if check is too frequent and/or if there is not enough
  traffic. In fact, you need in this case to check MySQL "max_connect_errors"
  value as if a connection is established successfully within fewer than MySQL
  "max_connect_errors" attempts after a previous connection was interrupted,
  the error count for the host is cleared to zero. If HAProxy's server get
  blocked, the "FLUSH HOSTS" statement is the only way to unblock it.

  Remember that this does not check database presence nor database consistency.
  To do this, you can use an external check with xinetd for example.

  The check requires MySQL >=3.22, for older version, please use TCP check.

  Most often, an incoming MySQL server needs to see the client's IP address for
  various purposes, including IP privilege matching and connection logging.
  When possible, it is often wise to masquerade the client's IP address when
  connecting to the server using the "usesrc" argument of the "source" keyword,
  which requires the transparent proxy feature to be compiled in, and the MySQL
  server to route the client via the machine hosting haproxy.

  See also: "option httpchk"


option nolinger
no option nolinger
  Enable or disable immediate session resource cleaning after close
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  When clients or servers abort connections in a dirty way (eg: they are
```

physically disconnected), the session timeouts triggers and the session is closed. But it will remain in FIN_WAIT1 state for some time in the system, using some resources and possibly limiting the ability to establish newer connections.

When this happens, it is possible to activate "option nolinger" which forces the system to immediately remove any socket's pending data on close. Thus, the session is instantly purged from the system's tables. This usually has side effects such as increased number of TCP resets due to old retransmits getting immediately rejected. Some firewalls may sometimes complain about this too.

For this reason, it is not recommended to use this option when not absolutely needed. You know that you need it when you have thousands of FIN_WAIT1 sessions on your system (TIME_WAIT ones do not count).

This option may be used both on frontends and backends, depending on the side where it is required. Use it on the frontend for clients, and on the backend for servers.

If this option has been enabled in a "defaults" section, it can be disabled in a specific instance by prepending the "no" keyword before it.

option originalto [except <network>] [header <name>]

Enable insertion of the X-Original-To header to requests sent to servers

```
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <network> is an optional argument used to disable this option for sources
              matching <network>
    <name>    an optional argument to specify a different "X-Original-To"
              header name.
```

Since HAProxy can work in transparent mode, every request from a client can be redirected to the proxy and HAProxy itself can proxy every request to a complex SQUID environment and the destination host from SO_ORIGINAL_DST will be lost. This is annoying when you want access rules based on destination ip addresses. To solve this problem, a new HTTP header "X-Original-To" may be added by HAProxy to all requests sent to the server. This header contains a value representing the original destination IP address. Since this must be configured to always use the last occurrence of this header only. Note that only the last occurrence of the header must be used, since it is really possible that the client has already brought one.

The keyword "header" may be used to supply a different header name to replace the default "X-Original-To". This can be useful where you might already have a "X-Original-To" header from a different application, and you need preserve it. Also if your backend server doesn't use the "X-Original-To" header and requires different one.

Sometimes, a same HAProxy instance may be shared between a direct client access and a reverse-proxy access (for instance when an SSL reverse-proxy is used to decrypt HTTPS traffic). It is possible to disable the addition of the header for a known source address or network by adding the "except" keyword followed by the network address. In this case, any source IP matching the network will not cause an addition of this header. Most common uses are with private networks or 127.0.0.1.

This option may be specified either in the frontend or in the backend. If at least one of them uses it, the header will be added. Note that the backend's setting of the header subargument takes precedence over the frontend's if both are defined.

Examples :

```
    # Original Destination address
    frontend www
        mode http
        option originalto except 127.0.0.1

    # Those servers want the IP Address in X-Client-Dst
    backend www
        mode http
        option originalto header X-Client-Dst
```

See also : "option httpclose", "option http-server-close", "option forceclose"

option persist
no option persist

Enable or disable forced persistence on down servers

```
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none
```

When an HTTP request reaches a backend with a cookie which references a dead server, by default it is redispatched to another server. It is possible to force the request to be sent to the dead server first using "option persist" if absolutely needed. A common use case is when servers are under extreme load and spend their time flapping. In this case, the users would still be directed to the server they opened the session on, in the hope they would be correctly served. It is recommended to use "option redispatch" in conjunction with this option so that in the event it would not be possible to connect to the server at all (server definitely dead), the client would finally be redirected to another valid server.

If this option has been enabled in a "defaults" section, it can be disabled in a specific instance by prepending the "no" keyword before it.

See also : "option redispatch", "retries", "force-persist"

option pgsql-check [user <username>]

Use PostgreSQL health checks for server testing

```
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <username> This is the username which will be used when connecting to
               PostgreSQL server.
```

The check sends a PostgreSQL StartupMessage and waits for either Authentication request or ErrorResponse message. It is a basic but useful test which does not produce error nor aborted connect on the server. This check is identical with the "mysql-check".

See also: "option httpchk"

option prefer-last-server
no option prefer-last-server

Allow multiple load balanced requests to remain on the same server

```
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none
```

When the load balancing algorithm in use is not deterministic, and a previous request was sent to a server to which haproxy still holds a connection, it is sometimes desirable that subsequent requests on a same session go to the same server as much as possible. Note that this is different from persistence, as

```
  we only indicate a preference which haproxy tries to apply without any form
  of warranty. The real use is for keep-alive connections sent to servers. When
  this option is used, haproxy will try to reuse the same connection that is
  attached to the server instead of rebalancing to another server, causing a
  close of the connection. This can make sense for static file servers. It does
  not make much sense to use this in combination with hashing algorithms. Note,
  haproxy already automatically tries to stick to a server which sends a 401 or
  to a proxy which sends a 407 (authentication required). This is mandatory for
  use with the broken NTLM authentication challenge, and significantly helps in
  troubleshooting some faulty applications. Option prefer-last-server might be
  desirable in these environments as well, to avoid redistributing the traffic
  after every other response.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also: "option http-keep-alive"


option redispatch
option redispatch <interval>
no option redispatch
  Enable or disable session redistribution in case of connection failure
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <interval> The optional integer value that controls how often redispatches
               occur when retrying connections. Positive value P indicates a
               redispatch is desired on every Pth retry, and negative value
               N indicate a redispath is desired on the Nth retry prior to the
               last retry. For example, the default of -1 preserves the
               historical behaviour of redispatching on the last retry, a
               positive value of 1 would indicate a redispatch on every retry,
               and a positive value of 3 would indicate a redispatch on every
               third retry. You can disable redispatches with a value of 0.


  In HTTP mode, if a server designated by a cookie is down, clients may
  definitely stick to it because they cannot flush the cookie, so they will not
  be able to access the service anymore.

  Specifying "option redispatch" will allow the proxy to break their
  persistence and redistribute them to a working server.

  It also allows to retry connections to another server in case of multiple
  connection failures. Of course, it requires having "retries" set to a nonzero
  value.

  This form is the preferred form, which replaces both the "redispatch" and
  "redisp" keywords.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "redispatch", "retries", "force-persist"


option redis-check
  Use redis health checks for server testing
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  It is possible to test that the server correctly talks REDIS protocol instead
  of just testing that it accepts the TCP connection. When this option is set,
```

```
  a PING redis command is sent to the server, and the response is analyzed to
  find the "+PONG" response message.

  Example :
        option redis-check

  See also : "option httpchk"


option smtpchk
option smtpchk <hello> <domain>
  Use SMTP health checks for server testing
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <hello>   is an optional argument. It is the "hello" command to use. It can
              be either "HELO" (for SMTP) or "EHLO" (for ESTMP). All other
              values will be turned into the default command ("HELO").

    <domain>  is the domain name to present to the server. It may only be
              specified (and is mandatory) if the hello command has been
              specified. By default, "localhost" is used.

  When "option smtpchk" is set, the health checks will consist in TCP
  connections followed by an SMTP command. By default, this command is
  "HELO localhost". The server's return code is analyzed and only return codes
  starting with a "2" will be considered as valid. All other responses,
  including a lack of response will constitute an error and will indicate a
  dead server.

  This test is meant to be used with SMTP servers or relays. Depending on the
  request, it is possible that some servers do not log each connection attempt,
  so you may want to experiment to improve the behaviour. Using telnet on port
  25 is often easier than adjusting the configuration.

  Most often, an incoming SMTP server needs to see the client's IP address for
  various purposes, including spam filtering, anti-spoofing and logging. When
  possible, it is often wise to masquerade the client's IP address when
  connecting to the server using the "usesrc" argument of the "source" keyword,
  which requires the transparent proxy feature to be compiled in.

  Example :
        option smtpchk HELO mydomain.org

  See also : "option httpchk", "source"


option socket-stats
no option socket-stats

  Enable or disable collecting & providing separate statistics for each socket.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no

  Arguments : none


option splice-auto
no option splice-auto
  Enable or disable automatic kernel acceleration on sockets in both directions
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  When this option is enabled either on a frontend or on a backend, haproxy
  will automatically evaluate the opportunity to use kernel tcp splicing to
  forward data between the client and the server, in either direction. Haproxy
  uses heuristics to estimate if kernel splicing might improve performance or
  not. Both directions are handled independently. Note that the heuristics used
  are not much aggressive in order to limit excessive use of splicing. This
  option requires splicing to be enabled at compile time, and may be globally
  disabled with the global option "nosplice". Since splice uses pipes, using it
  requires that there are enough spare pipes.

  Important note: kernel-based TCP splicing is a Linux-specific feature which
  first appeared in kernel 2.6.25. It offers kernel-based acceleration to
  transfer data between sockets without copying these data to user-space, thus
  providing noticeable performance gains and CPU cycles savings. Since many
  early implementations are buggy, corrupt data and/or are inefficient, this
  feature is not enabled by default, and it should be used with extreme care.
  While it is not possible to detect the correctness of an implementation,
  2.6.29 is the first version offering a properly working implementation. In
  case of doubt, splicing may be globally disabled using the global "nosplice"
  keyword.

  Example :
        option splice-auto

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option splice-request", "option splice-response", and global
             options "nosplice" and "maxpipes"


option splice-request
no option splice-request
  Enable or disable automatic kernel acceleration on sockets for requests
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  When this option is enabled either on a frontend or on a backend, haproxy
  will use kernel tcp splicing whenever possible to forward data going from
  the client to the server. It might still use the recv/send scheme if there
  are no spare pipes left. This option requires splicing to be enabled at
  compile time, and may be globally disabled with the global option "nosplice".
  Since splice uses pipes, using it requires that there are enough spare pipes.

  Important note: see "option splice-auto" for usage limitations.

  Example :
        option splice-request

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option splice-auto", "option splice-response", and global options
             "nosplice" and "maxpipes"


option splice-response
no option splice-response
  Enable or disable automatic kernel acceleration on sockets for responses
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  When this option is enabled either on a frontend or on a backend, haproxy
  will use kernel tcp splicing whenever possible to forward data going from
```

```
  the server to the client. It might still use the recv/send scheme if there
  are no spare pipes left. This option requires splicing to be enabled at
  compile time, and may be globally disabled with the global option "nosplice".
  Since splice uses pipes, using it requires that there are enough spare pipes.

  Important note: see "option splice-auto" for usage limitations.

  Example :
        option splice-response

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option splice-auto", "option splice-request", and global options
             "nosplice" and "maxpipes"


option srvtcpka
no option srvtcpka
  Enable or disable the sending of TCP keepalive packets on the server side
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  When there is a firewall or any session-aware component between a client and
  a server, and when the protocol involves very long sessions with long idle
  periods (eg: remote desktops), there is a risk that one of the intermediate
  components decides to expire a session which has remained idle for too long.

  Enabling socket-level TCP keep-alives makes the system regularly send packets
  to the other end of the connection, leaving it active. The delay between
  keep-alive probes is controlled by the system only and depends both on the
  operating system and its tuning parameters.

  It is important to understand that keep-alive packets are neither emitted nor
  received at the application level. It is only the network stacks which sees
  them. For this reason, even if one side of the proxy already uses keep-alives
  to maintain its connection alive, those keep-alive packets will not be
  forwarded to the other side of the proxy.

  Please note that this has nothing to do with HTTP keep-alive.

  Using option "srvtcpka" enables the emission of TCP keep-alive probes on the
  server side of a connection, which should help when session expirations are
  noticed between HAProxy and a server.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option clitcpka", "option tcpka"


option ssl-hello-chk
  Use SSLv3 client hello health checks for server testing
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  When some SSL-based protocols are relayed in TCP mode through HAProxy, it is
  possible to test that the server correctly talks SSL instead of just testing
  that it accepts the TCP connection. When "option ssl-hello-chk" is set, pure
  SSLv3 client hello messages are sent once the connection is established to
  the server, and the response is analyzed to find an SSL server hello message.
  The server is considered valid only when the response contains this server
  hello message.
```

All servers tested till there correctly reply to SSLv3 client hello messages, and most servers tested do not even log the requests containing only hello messages, which is appreciable.

Note that this check works even when SSL support was not built into haproxy because it forges the SSL message. When SSL support is available, it is best to use native SSL health checks instead of this one.

See also: "option httpchk", "check-ssl"

option tcp-check

Perform health checks using tcp-check send/expect sequences

```
May be used in sections:   defaults | frontend | listen | backend
                             yes    |    no    |   yes  |   yes
```

This health check method is intended to be combined with "tcp-check" command lists in order to support send/expect types of health check sequences.

TCP checks currently support 4 modes of operations :

- no "tcp-check" directive : the health check only consists in a connection attempt, which remains the default mode.

- "tcp-check send" or "tcp-check send-binary" only is mentioned : this is used to send a string along with a connection opening. With some protocols, it helps sending a "QUIT" message for example that prevents the server from logging a connection error for each health check. The check result will still be based on the ability to open the connection only.

- "tcp-check expect" only is mentioned : this is used to test a banner. The connection is opened and haproxy waits for the server to present some contents which must validate some rules. The check result will be based on the matching between the contents and the rules. This is suited for POP, IMAP, SMTP, FTP, SSH, TELNET.

- both "tcp-check send" and "tcp-check expect" are mentioned : this is used to test a hello-type protocol. Haproxy sends a message, the server responds and its response is analysed. the check result will be based on the matching between the response contents and the rules. This is often suited for protocols which require a binding or a request/response model. LDAP, MySQL, Redis and SSL are example of such protocols, though they already all have their dedicated checks with a deeper understanding of the respective protocols.
  In this mode, many questions may be sent and many answers may be analysed.

A fifth mode can be used to insert comments in different steps of the script.

For each tcp-check rule you create, you can add a "comment" directive, followed by a string. This string will be reported in the log and stderr in debug mode. It is useful to make user-friendly error reporting. The "comment" is of course optional.

Examples :

```
        # perform a POP check (analyse only server's banner)
        option tcp-check
        tcp-check expect string +OK\ POP3\ ready comment POP\ protocol

        # perform an IMAP check (analyse only server's banner)
        option tcp-check
        tcp-check expect string *\ OK\ IMAP4\ ready comment IMAP\ protocol
```

```
        # look for the redis master server after ensuring it speaks well
        # redis protocol, then it exits properly.
        # (send a command then analyse the response 3 times)
        option tcp-check
        tcp-check comment PING\ phase
        tcp-check send PING\r\n
        tcp-check expect string +PONG
        tcp-check comment role\ check
        tcp-check send info\ replication\r\n
        tcp-check expect string role:master
        tcp-check comment QUIT\ phase
        tcp-check send QUIT\r\n
        tcp-check expect string +OK

        forge a HTTP request, then analyse the response
        (send many headers before analyzing)
        option tcp-check
        tcp-check comment forge\ and\ send\ HTTP\ request
        tcp-check send HEAD\ /\ HTTP/1.1\r\n
        tcp-check send Host:\ www.mydomain.com\r\n
        tcp-check send User-Agent:\ HAProxy\ tcpcheck\r\n
        tcp-check send \r\n
        tcp-check expect rstring HTTP/1\..\ (2..|3..) comment check\ HTTP\ response
```

See also : "tcp-check expect", "tcp-check send"

option tcp-smart-accept
no option tcp-smart-accept

Enable or disable the saving of one ACK packet during the accept sequence

```
May be used in sections :   defaults | frontend | listen | backend
                              yes    |    yes   |   yes  |   no
```

Arguments : none

When an HTTP connection request comes in, the system acknowledges it on behalf of HAProxy, then the client immediately sends its request, and the system acknowledges it too while it is notifying HAProxy about the new connection. HAProxy then reads the request and responds. This means that we have one TCP ACK sent by the system for nothing, because the request could very well be acknowledged by HAProxy when it sends its response.

For this reason, in HTTP mode, HAProxy automatically asks the system to avoid sending this useless ACK on platforms which support it (currently at least Linux). It must not cause any problem, because the system will send it anyway after 40 ms if the response takes more time than expected to come.

During complex network debugging sessions, it may be desirable to disable this optimization because delayed ACKs can make troubleshooting more complex when trying to identify where packets are delayed. It is then possible to fall back to normal behaviour by specifying "no option tcp-smart-accept".

It is also possible to force it for non-HTTP proxies by simply specifying "option tcp-smart-accept". For instance, it can make sense with some services such as SMTP where the server speaks first.

It is recommended to avoid forcing this option in a defaults section. In case of doubt, consider setting it back to automatic values by prepending the "default" keyword before it, or disabling it using the "no" keyword.

See also : "option tcp-smart-connect"

option tcp-smart-connect

```
no option tcp-smart-connect
  Enable or disable the saving of one ACK packet during the connect sequence
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  On certain systems (at least Linux), HAProxy can ask the kernel not to
  immediately send an empty ACK upon a connection request, but to directly
  send the buffer request instead. This saves one packet on the network and
  thus boosts performance. It can also be useful for some servers, because they
  immediately get the request along with the incoming connection.

  This feature is enabled when "option tcp-smart-connect" is set in a backend.
  It is not enabled by default because it makes network troubleshooting more
  complex.

  It only makes sense to enable it with protocols where the client speaks first
  such as HTTP. In other situations, if there is no data to send in place of
  the ACK, a normal ACK is sent.

  If this option has been enabled in a "defaults" section, it can be disabled
  in a specific instance by prepending the "no" keyword before it.

  See also : "option tcp-smart-accept"


option tcpka
  Enable or disable the sending of TCP keepalive packets on both sides
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  When there is a firewall or any session-aware component between a client and
  a server, and when the protocol involves very long sessions with long idle
  periods (eg: remote desktops), there is a risk that one of the intermediate
  components decides to expire a session which has remained idle for too long.

  Enabling socket-level TCP keep-alives makes the system regularly send packets
  to the other end of the connection, leaving it active. The delay between
  keep-alive probes is controlled by the system only and depends both on the
  operating system and its tuning parameters.

  It is important to understand that keep-alive packets are neither emitted nor
  received at the application level. It is only the network stacks which sees
  them. For this reason, even if one side of the proxy already uses keep-alives
  to maintain its connection alive, those keep-alive packets will not be
  forwarded to the other side of the proxy.

  Please note that this has nothing to do with HTTP keep-alive.

  Using option "tcpka" enables the emission of TCP keep-alive probes on both
  the client and server sides of a connection. Note that this is meaningful
  only in "defaults" or "listen" sections. If this option is used in a
  frontend, only the client side will get keep-alives, and if this option is
  used in a backend, only the server side will get keep-alives. For this
  reason, it is strongly recommended to explicitly use "option clitcpka" and
  "option srvtcpka" when the configuration is split between frontends and
  backends.

  See also : "option clitcpka", "option srvtcpka"


option tcplog
  Enable advanced logging of TCP connections with session state and timers
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  By default, the log output format is very poor, as it only contains the
  source and destination addresses, and the instance name. By specifying
  "option tcplog", each log line turns into a much richer format including, but
  not limited to, the connection timers, the session status, the connections
  numbers, the frontend, backend and server name, and of course the source
  address and ports. This option is useful for pure TCP proxies in order to
  find which of the client or server disconnects or times out. For normal HTTP
  proxies, it's better to use "option httplog" which is even more complete.

  This option may be set either in the frontend or the backend.

  See also :  "option httplog", and section 8 about logging.


option transparent
no option transparent
  Enable client-side transparent proxying
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  This option was introduced in order to provide layer 7 persistence to layer 3
  load balancers. The idea is to use the OS's ability to redirect an incoming
  connection for a remote address to a local process (here HAProxy), and let
  this process know what address was initially requested. When this option is
  used, sessions without cookies will be forwarded to the original destination
  IP address of the incoming request (which should match that of another
  equipment), while requests with cookies will still be forwarded to the
  appropriate server.

  Note that contrary to a common belief, this option does NOT make HAProxy
  present the client's IP to the server when establishing the connection.

  See also: the "usesrc" argument of the "source" keyword, and the
            "transparent" option of the "bind" keyword.


external-check command <command>
  Executable to run when performing an external-check
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes

  Arguments :
    <command> is the external command to run

  The arguments passed to the to the command are:

  <proxy_address> <proxy_port> <server_address> <server_port>

  The <proxy_address> and <proxy_port> are derived from the first listener
  that is either IPv4, IPv6 or a UNIX socket. In the case of a UNIX socket
  listener the proxy_address will be the path of the socket and the
  <proxy_port> will be the string "NOT_USED". In a backend section, it's not
  possible to determine a listener, and both <proxy_address> and <proxy_port>
  will have the string value "NOT_USED".

  Some values are also provided through environment variables.

  Environment variables :
    HAPROXY_PROXY_ADDR      The first bind address if available (or empty if not
                            applicable, for example in a "backend" section).
    HAPROXY_PROXY_ID        The backend id.

    HAPROXY_PROXY_NAME      The backend name.

    HAPROXY_PROXY_PORT      The first bind port if available (or empty if not
                            applicable, for example in a "backend" section or
                            for a UNIX socket).

    HAPROXY_SERVER_ADDR     The server address.

    HAPROXY_SERVER_CURCONN  The current number of connections on the server.

    HAPROXY_SERVER_ID       The server id.

    HAPROXY_SERVER_MAXCONN  The server max connections.

    HAPROXY_SERVER_NAME     The server name.

    HAPROXY_SERVER_PORT     The server port if available (or empty for a UNIX
                            socket).

    PATH                    The PATH environment variable used when executing
                            the command may be set using "external-check path".

  If the command executed and exits with a zero status then the check is
  considered to have passed, otherwise the check is considered to have
  failed.

  Example :
        external-check command /bin/true

  See also : "external-check", "option external-check", "external-check path"


external-check path <path>
  The value of the PATH environment variable used when running an external-check
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes

  Arguments :
    <path> is the path used when executing external command to run

  The default path is "".

  Example :
        external-check path "/usr/bin:/bin"

  See also : "external-check", "option external-check",
             "external-check command"


persist rdp-cookie
persist rdp-cookie(<name>)
  Enable RDP cookie-based persistence
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <name>    is the optional name of the RDP cookie to check. If omitted, the
              default cookie name "msts" will be used. There currently is no
              valid reason to change this name.

  This statement enables persistence based on an RDP cookie. The RDP cookie
  contains all information required to find the server in the list of known
  servers. So when this option is set in the backend, the request is analysed
  and if an RDP cookie is found, it is decoded. If it matches a known server
```

```
  which is still UP (or if "option persist" is set), then the connection is
  forwarded to this server.

  Note that this only makes sense in a TCP backend, but for this to work, the
  frontend must have waited long enough to ensure that an RDP cookie is present
  in the request buffer. This is the same requirement as with the "rdp-cookie"
  load-balancing method. Thus it is highly recommended to put all statements in
  a single "listen" section.

  Also, it is important to understand that the terminal server will emit this
  RDP cookie only if it is configured for "token redirection mode", which means
  that the "IP address redirection" option is disabled.

  Example :
        listen tse-farm
            bind :3389
            # wait up to 5s for an RDP cookie in the request
            tcp-request inspect-delay 5s
            tcp-request content accept if RDP_COOKIE
            # apply RDP cookie persistence
            persist rdp-cookie
            # if server is unknown, let's balance on the same cookie.
            # alternatively, "balance leastconn" may be useful too.
            balance rdp-cookie
            server srv1 1.1.1.1:3389
            server srv2 1.1.1.2:3389

  See also : "balance rdp-cookie", "tcp-request", the "req_rdp_cookie" ACL and
  the rdp_cookie pattern fetch function.


rate-limit sessions <rate>
  Set a limit on the number of new sessions accepted per second on a frontend
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments :
    <rate>    The <rate> parameter is an integer designating the maximum number
              of new sessions per second to accept on the frontend.

  When the frontend reaches the specified number of new sessions per second, it
  stops accepting new connections until the rate drops below the limit again.
  During this time, the pending sessions will be kept in the socket's backlog
  (in system buffers) and haproxy will not even be aware that sessions are
  pending. When applying very low limit on a highly loaded service, it may make
  sense to increase the socket's backlog using the "backlog" keyword.

  This feature is particularly efficient at blocking connection-based attacks
  or service abuse on fragile servers. Since the session rate is measured every
  millisecond, it is extremely accurate. Also, the limit applies immediately,
  no delay is needed at all to detect the threshold.

  Example : limit the connection rate on SMTP to 10 per second max
        listen smtp
            mode tcp
            bind :25
            rate-limit sessions 10
            server 127.0.0.1:1025

  Note : when the maximum rate is reached, the frontend's status is not changed
         but its sockets appear as "WAITING" in the statistics if the
         "socket-stats" option is enabled.

  See also : the "backlog" keyword and the "fe_sess_rate" ACL criterion.
```

```
redirect location <loc> [code <code>] <option> [{if | unless} <condition>]
redirect prefix   <pfx> [code <code>] <option> [{if | unless} <condition>]
redirect scheme   <sch> [code <code>] <option> [{if | unless} <condition>]
  Return an HTTP redirection if/unless a condition is matched
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes

  If/unless the condition is matched, the HTTP request will lead to a redirect
  response. If no condition is specified, the redirect applies unconditionally.

  Arguments :
    <loc>     With "redirect location", the exact value in <loc> is placed into
              the HTTP "Location" header. When used in an "http-request" rule,
              <loc> value follows the log-format rules and can include some
              dynamic values (see Custom Log Format in section 8.2.4).

    <pfx>     With "redirect prefix", the "Location" header is built from the
              concatenation of <pfx> and the complete URI path, including the
              query string, unless the "drop-query" option is specified (see
              below). As a special case, if <pfx> equals exactly "/", then
              nothing is inserted before the original URI. It allows one to
              redirect to the same URL (for instance, to insert a cookie). When
              used in an "http-request" rule, <pfx> value follows the log-format
              rules and can include some dynamic values (see Custom Log Format
              in section 8.2.4).

    <sch>     With "redirect scheme", then the "Location" header is built by
              concatenating <sch> with "://" then the first occurrence of the
              "Host" header, and then the URI path, including the query string
              unless the "drop-query" option is specified (see below). If no
              path is found or if the path is "*", then "/" is used instead. If
              no "Host" header is found, then an empty host component will be
              returned, which most recent browsers interpret as redirecting to
              the same host. This directive is mostly used to redirect HTTP to
              HTTPS. When used in an "http-request" rule, <sch> value follows
              the log-format rules and can include some dynamic values (see
              Custom Log Format in section 8.2.4).

    <code>    The code is optional. It indicates which type of HTTP redirection
              is desired. Only codes 301, 302, 303, 307 and 308 are supported,
              with 302 used by default if no code is specified. 301 means
              "Moved permanently", and a browser may cache the Location. 302
              means "Moved temporarily" and means that the browser should not
              cache the redirection. 303 is equivalent to 302 except that the
              browser will fetch the location with a GET method. 307 is just
              like 302 but makes it clear that the same method must be reused.
              Likewise, 308 replaces 301 if the same method must be used.

    <option>  There are several options which can be specified to adjust the
              expected behaviour of a redirection :

      - "drop-query"
        When this keyword is used in a prefix-based redirection, then the
        location will be set without any possible query-string, which is useful
        for directing users to a non-secure page for instance. It has no effect
        with a location-type redirect.

      - "append-slash"
        This keyword may be used in conjunction with "drop-query" to redirect
        users who use a URL not ending with a '/' to the same one with the '/'.
        It can be useful to ensure that search engines will only see one URL.
        For this, a return code 301 is preferred.

      - "set-cookie NAME[=value]"
        A "Set-Cookie" header will be added with NAME (and optionally "=value")
```

```
        to the response. This is sometimes used to indicate that a user has
        been seen, for instance to protect against some types of DoS. No other
        cookie option is added, so the cookie will be a session cookie. Note
        that for a browser, a sole cookie name without an equal sign is
        different from a cookie with an equal sign.

      - "clear-cookie NAME[=]"
        A "Set-Cookie" header will be added with NAME (and optionally "="), but
        with the "Max-Age" attribute set to zero. This will tell the browser to
        delete this cookie. It is useful for instance on logout pages. It is
        important to note that clearing the cookie "NAME" will not remove a
        cookie set with "NAME=value". You have to clear the cookie "NAME=" for
        that, because the browser makes the difference.

  Example: move the login URL only to HTTPS.
        acl clear      dst_port  80
        acl secure     dst_port  8080
        acl login_page url_beg   /login
        acl logout     url_beg   /logout
        acl uid_given  url_reg   /login?userid=[^&]+
        acl cookie_set hdr_sub(cookie) SEEN=1

        redirect prefix   https://mysite.com set-cookie SEEN=1 if !cookie_set
        redirect prefix   https://mysite.com           if login_page !secure
        redirect prefix   http://mysite.com drop-query if login_page !uid_given
        redirect location http://mysite.com/           if !login_page secure
        redirect location / clear-cookie USERID=       if logout

  Example: send redirects for request for articles without a '/'.
        acl missing_slash path_reg ^/article/[^/]*$
        redirect code 301 prefix / drop-query append-slash if missing_slash

  Example: redirect all HTTP traffic to HTTPS when SSL is handled by haproxy.
        redirect scheme https if !{ ssl_fc }

  Example: append 'www.' prefix in front of all hosts not having it
        http-request redirect code 301 location www.%[hdr(host)]%[req.uri] \
          unless { hdr_beg(host) -i www }

  See section 7 about ACL usage.


redisp (deprecated)
redispatch (deprecated)
  Enable or disable session redistribution in case of connection failure
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  In HTTP mode, if a server designated by a cookie is down, clients may
  definitely stick to it because they cannot flush the cookie, so they will not
  be able to access the service anymore.

  Specifying "redispatch" will allow the proxy to break their persistence and
  redistribute them to a working server.

  It also allows to retry last connection to another server in case of multiple
  connection failures. Of course, it requires having "retries" set to a nonzero
  value.

  This form is deprecated, do not use it in any new configuration, use the new
  "option redispatch" instead.

  See also : "option redispatch"
```

```

reqadd  <string> [{if | unless} <cond>]
  Add a header at the end of the HTTP request
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <string>  is the complete line to be added. Any space or known delimiter
              must be escaped using a backslash ('\'). Please refer to section
              6 about HTTP header manipulation for more information.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  A new line consisting in <string> followed by a line feed will be added after
  the last header of an HTTP request.

  Header transformations only apply to traffic which passes through HAProxy,
  and not to traffic generated by HAProxy, such as health-checks or error
  responses.

  Example : add "X-Proto: SSL" to requests coming via port 81
     acl is-ssl  dst_port       81
     reqadd      X-Proto:\ SSL  if is-ssl

  See also: "rspadd", "http-request", section 6 about HTTP header manipulation,
            and section 7 about ACLs.


reqallow  <search> [{if | unless} <cond>]
reqiallow <search> [{if | unless} <cond>] (ignore case)
  Definitely allow an HTTP request if a line matches a regular expression
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <search>  is the regular expression applied to HTTP headers and to the
              request line. This is an extended regular expression. Parenthesis
              grouping is supported and no preliminary backslash is required.
              Any space or known delimiter must be escaped using a backslash
              ('\'). The pattern applies to a full line at a time. The
              "reqallow" keyword strictly matches case while "reqiallow"
              ignores case.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  A request containing any line which matches extended regular expression
  <search> will mark the request as allowed, even if any later test would
  result in a deny. The test applies both to the request line and to request
  headers. Keep in mind that URLs in request line are case-sensitive while
  header names are not.

  It is easier, faster and more powerful to use ACLs to write access policies.
  Reqdeny, reqallow and reqpass should be avoided in new designs.

  Example :
     # allow www.* but refuse *.local
     reqiallow ^Host:\ www\.
     reqideny  ^Host:\ .*\.local

  See also: "reqdeny", "block", "http-request",  section 6 about HTTP header
            manipulation, and section 7 about ACLs.


reqdel  <search> [{if | unless} <cond>]
reqidel <search> [{if | unless} <cond>]  (ignore case)
```

```
  Delete all headers matching a regular expression in an HTTP request
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <search>  is the regular expression applied to HTTP headers and to the
              request line. This is an extended regular expression. Parenthesis
              grouping is supported and no preliminary backslash is required.
              Any space or known delimiter must be escaped using a backslash
              ('\'). The pattern applies to a full line at a time. The "reqdel"
              keyword strictly matches case while "reqidel" ignores case.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  Any header line matching extended regular expression <search> in the request
  will be completely deleted. Most common use of this is to remove unwanted
  and/or dangerous headers or cookies from a request before passing it to the
  next servers.

  Header transformations only apply to traffic which passes through HAProxy,
  and not to traffic generated by HAProxy, such as health-checks or error
  responses. Keep in mind that header names are not case-sensitive.

  Example :
     # remove X-Forwarded-For header and SERVER cookie
     reqidel ^X-Forwarded-For:.*
     reqidel ^Cookie:.*SERVER=

  See also: "reqadd", "reqrep", "rspdel", "http-request", section 6 about
            HTTP header manipulation, and section 7 about ACLs.


reqdeny  <search> [{if | unless} <cond>]
reqideny <search> [{if | unless} <cond>]  (ignore case)
  Deny an HTTP request if a line matches a regular expression
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <search>  is the regular expression applied to HTTP headers and to the
              request line. This is an extended regular expression. Parenthesis
              grouping is supported and no preliminary backslash is required.
              Any space or known delimiter must be escaped using a backslash
              ('\'). The pattern applies to a full line at a time. The
              "reqdeny" keyword strictly matches case while "reqideny" ignores
              case.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  A request containing any line which matches extended regular expression
  <search> will mark the request as denied, even if any later test would
  result in an allow. The test applies both to the request line and to request
  headers. Keep in mind that URLs in request line are case-sensitive while
  header names are not.

  A denied request will generate an "HTTP 403 forbidden" response once the
  complete request has been parsed. This is consistent with what is practiced
  using ACLs.

  It is easier, faster and more powerful to use ACLs to write access policies.
  Reqdeny, reqallow and reqpass should be avoided in new designs.

  Example :
     # refuse *.local, then allow www.*
     reqideny  ^Host:\ .*\.local
    reqiallow ^Host:\ www\.

  See also: "reqallow", "rspdeny", "block", "http-request", section 6 about
            HTTP header manipulation, and section 7 about ACLs.


reqpass  <search> [{if | unless} <cond>]
reqipass <search> [{if | unless} <cond>]  (ignore case)
  Ignore any HTTP request line matching a regular expression in next rules
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <search>  is the regular expression applied to HTTP headers and to the
              request line. This is an extended regular expression. Parenthesis
              grouping is supported and no preliminary backslash is required.
              Any space or known delimiter must be escaped using a backslash
              ('\'). The pattern applies to a full line at a time. The
              "reqpass" keyword strictly matches case while "reqipass" ignores
              case.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  A request containing any line which matches extended regular expression
  <search> will skip next rules, without assigning any deny or allow verdict.
  The test applies both to the request line and to request headers. Keep in
  mind that URLs in request line are case-sensitive while header names are not.

  It is easier, faster and more powerful to use ACLs to write access policies.
  Reqdeny, reqallow and reqpass should be avoided in new designs.

  Example :
     # refuse *.local, then allow www.*, but ignore "www.private.local"
     reqipass  ^Host:\ www.private\.local
     reqideny  ^Host:\ .*\.local
     reqiallow ^Host:\ www\.

  See also: "reqallow", "reqdeny", "block", "http-request", section 6 about
            HTTP header manipulation, and section 7 about ACLs.


reqrep  <search> <string> [{if | unless} <cond>]
reqirep <search> <string> [{if | unless} <cond>]   (ignore case)
  Replace a regular expression with a string in an HTTP request line
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <search>  is the regular expression applied to HTTP headers and to the
              request line. This is an extended regular expression. Parenthesis
              grouping is supported and no preliminary backslash is required.
              Any space or known delimiter must be escaped using a backslash
              ('\'). The pattern applies to a full line at a time. The "reqrep"
              keyword strictly matches case while "reqirep" ignores case.

    <string>  is the complete line to be added. Any space or known delimiter
              must be escaped using a backslash ('\'). References to matched
              pattern groups are possible using the common \N form, with N
              being a single digit between 0 and 9. Please refer to section
              6 about HTTP header manipulation for more information.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  Any line matching extended regular expression <search> in the request (both
  the request line and header lines) will be completely replaced with <string>.
```

```
  Most common use of this is to rewrite URLs or domain names in "Host" headers.

  Header transformations only apply to traffic which passes through HAProxy,
  and not to traffic generated by HAProxy, such as health-checks or error
  responses. Note that for increased readability, it is suggested to add enough
  spaces between the request and the response. Keep in mind that URLs in
  request line are case-sensitive while header names are not.

  Example :
     # replace "/static/" with "/" at the beginning of any request path.
     reqrep ^([^\ :]*)\ /static/(.*)     \1\ /\2
     # replace "www.mydomain.com" with "www" in the host name.
     reqirep ^Host:\ www.mydomain.com   Host:\ www

  See also: "reqadd", "reqdel", "rsprep", "tune.bufsize", "http-request",
            section 6 about HTTP header manipulation, and section 7 about ACLs.


reqtarpit  <search> [{if | unless} <cond>]
reqitarpit <search> [{if | unless} <cond>]  (ignore case)
  Tarpit an HTTP request containing a line matching a regular expression
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <search>  is the regular expression applied to HTTP headers and to the
              request line. This is an extended regular expression. Parenthesis
              grouping is supported and no preliminary backslash is required.
              Any space or known delimiter must be escaped using a backslash
              ('\'). The pattern applies to a full line at a time. The
              "reqtarpit" keyword strictly matches case while "reqitarpit"
              ignores case.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  A request containing any line which matches extended regular expression
  <search> will be tarpitted, which means that it will connect to nowhere, will
  be kept open for a pre-defined time, then will return an HTTP error 500 so
  that the attacker does not suspect it has been tarpitted. The status 500 will
  be reported in the logs, but the completion flags will indicate "PT". The
  delay is defined by "timeout tarpit", or "timeout connect" if the former is
  not set.

  The goal of the tarpit is to slow down robots attacking servers with
  identifiable requests. Many robots limit their outgoing number of connections
  and stay connected waiting for a reply which can take several minutes to
  come. Depending on the environment and attack, it may be particularly
  efficient at reducing the load on the network and firewalls.

  Examples :
     # ignore user-agents reporting any flavour of "Mozilla" or "MSIE", but
     # block all others.
     reqipass   ^User-Agent:\.*(Mozilla|MSIE)
     reqitarpit ^User-Agent:

     # block bad guys
     acl badguys src 10.1.0.3 172.16.13.20/28
     reqitarpit . if badguys

  See also: "reqallow", "reqdeny", "reqpass", "http-request",  section 6
            about HTTP header manipulation, and section 7 about ACLs.


retries <value>
  Set the number of retries to perform on a server after a connection failure
  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <value>   is the number of times a connection attempt should be retried on
              a server when a connection either is refused or times out. The
              default value is 3.

  It is important to understand that this value applies to the number of
  connection attempts, not full requests. When a connection has effectively
  been established to a server, there will be no more retry.

  In order to avoid immediate reconnections to a server which is restarting,
  a turn-around timer of min("timeout connect", one second) is applied before
  a retry occurs.

  When "option redispatch" is set, the last retry may be performed on another
  server even if a cookie references a different server.

  See also : "option redispatch"


rspadd <string> [{if | unless} <cond>]
  Add a header at the end of the HTTP response
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <string>  is the complete line to be added. Any space or known delimiter
              must be escaped using a backslash ('\'). Please refer to section
              6 about HTTP header manipulation for more information.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  A new line consisting in <string> followed by a line feed will be added after
  the last header of an HTTP response.

  Header transformations only apply to traffic which passes through HAProxy,
  and not to traffic generated by HAProxy, such as health-checks or error
  responses.

  See also: "rspdel" "reqadd", "http-response", section 6 about HTTP header
            manipulation, and section 7 about ACLs.


rspdel  <search> [{if | unless} <cond>]
rspidel <search> [{if | unless} <cond>]  (ignore case)
  Delete all headers matching a regular expression in an HTTP response
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <search>  is the regular expression applied to HTTP headers and to the
              response line. This is an extended regular expression, so
              parenthesis grouping is supported and no preliminary backslash
              is required. Any space or known delimiter must be escaped using
              a backslash ('\'). The pattern applies to a full line at a time.
              The "rspdel" keyword strictly matches case while "rspidel"
              ignores case.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  Any header line matching extended regular expression <search> in the response
  will be completely deleted. Most common use of this is to remove unwanted
  and/or sensitive headers or cookies from a response before passing it to the
  client.
```

```

  Header transformations only apply to traffic which passes through HAProxy,
  and not to traffic generated by HAProxy, such as health-checks or error
  responses. Keep in mind that header names are not case-sensitive.

  Example :
     # remove the Server header from responses
     rspidel ^Server:.*

  See also: "rspadd", "rsprep", "reqdel", "http-response",  section 6 about
            HTTP header manipulation, and section 7 about ACLs.


rspdeny  <search> [{if | unless} <cond>]
rspideny <search> [{if | unless} <cond>]  (ignore case)
  Block an HTTP response if a line matches a regular expression
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <search>  is the regular expression applied to HTTP headers and to the
              response line. This is an extended regular expression, so
              parenthesis grouping is supported and no preliminary backslash
              is required. Any space or known delimiter must be escaped using
              a backslash ('\'). The pattern applies to a full line at a time.
              The "rspdeny" keyword strictly matches case while "rspideny"
              ignores case.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  A response containing any line which matches extended regular expression
  <search> will mark the request as denied. The test applies both to the
  response line and to response headers. Keep in mind that header names are not
  case-sensitive.

  Main use of this keyword is to prevent sensitive information leak and to
  block the response before it reaches the client. If a response is denied, it
  will be replaced with an HTTP 502 error so that the client never retrieves
  any sensitive data.

  It is easier, faster and more powerful to use ACLs to write access policies.
  Rspdeny should be avoided in new designs.

  Example :
     # Ensure that no content type matching ms-word will leak
     rspideny  ^Content-type:\.*/ms-word

  See also: "reqdeny", "acl", "block", "http-response",  section 6 about
            HTTP header manipulation and section 7 about ACLs.


rsprep  <search> <string> [{if | unless} <cond>]
rspirep <search> <string> [{if | unless} <cond>]  (ignore case)
  Replace a regular expression with a string in an HTTP response line
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <search>  is the regular expression applied to HTTP headers and to the
              response line. This is an extended regular expression, so
              parenthesis grouping is supported and no preliminary backslash
              is required. Any space or known delimiter must be escaped using
              a backslash ('\'). The pattern applies to a full line at a time.
              The "rsprep" keyword strictly matches case while "rspirep"
              ignores case.

    <string>  is the complete line to be added. Any space or known delimiter
              must be escaped using a backslash ('\'). References to matched
              pattern groups are possible using the common \N form, with N
              being a single digit between 0 and 9. Please refer to section
              6 about HTTP header manipulation for more information.

    <cond>    is an optional matching condition built from ACLs. It makes it
              possible to ignore this rule when other conditions are not met.

  Any line matching extended regular expression <search> in the response (both
  the response line and header lines) will be completely replaced with
  <string>. Most common use of this is to rewrite Location headers.

  Header transformations only apply to traffic which passes through HAProxy,
  and not to traffic generated by HAProxy, such as health-checks or error
  responses. Note that for increased readability, it is suggested to add enough
  spaces between the request and the response. Keep in mind that header names
  are not case-sensitive.

  Example :
     # replace "Location: 127.0.0.1:8080" with "Location: www.mydomain.com"
     rspirep ^Location:\ 127.0.0.1:8080    Location:\ www.mydomain.com

  See also: "rspadd", "rspdel", "reqrep", "http-response", section 6 about
            HTTP header manipulation, and section 7 about ACLs.


server <name> <address>[:[port]] [param*]
  Declare a server in a backend
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    no    |   yes  |   yes
  Arguments :
    <name>    is the internal name assigned to this server. This name will
              appear in logs and alerts.  If "http-send-name-header" is
              set, it will be added to the request header sent to the server.

    <address> is the IPv4 or IPv6 address of the server. Alternatively, a
              resolvable hostname is supported, but this name will be resolved
              during start-up. Address "0.0.0.0" or "*" has a special meaning.
              It indicates that the connection will be forwarded to the same IP
              address as the one from the client connection. This is useful in
              transparent proxy architectures where the client's connection is
              intercepted and haproxy must forward to the original destination
              address. This is more or less what the "transparent" keyword does
              except that with a server it's possible to limit concurrency and
              to report statistics. Optionally, an address family prefix may be
              used before the address to force the family regardless of the
              address format, which can be useful to specify a path to a unix
              socket with no slash ('/'). Currently supported prefixes are :
                    - 'ipv4@'  -> address is always IPv4
                    - 'ipv6@'  -> address is always IPv6
                    - 'unix@'  -> address is a path to a local unix socket
                    - 'abns@'  -> address is in abstract namespace (Linux only)
              You may want to reference some environment variables in the
              address parameter, see section 2.3 about environment
              variables.

    <port>    is an optional port specification. If set, all connections will
              be sent to this port. If unset, the same port the client
              connected to will be used. The port may also be prefixed by a "+"
              or a "-". In this case, the server's port will be determined by
              adding this value to the client's port.

    <param*>  is a list of parameters for this server. The "server" keywords
              accepts an important number of options and has a complete section
```

```
              dedicated to it. Please refer to section 5 for more details.

  Examples :
        server first  10.1.1.1:1080 cookie first  check inter 1000
        server second 10.1.1.2:1080 cookie second check inter 1000
        server transp ipv4@
        server backup "${SRV_BACKUP}:1080" backup
        server www1_dc1 "${LAN_DC1}.101:80"
        server www1_dc2 "${LAN_DC2}.101:80"

  Note: regarding Linux's abstract namespace sockets, HAProxy uses the whole
        sun_path length is used for the address length. Some other programs
        such as socat use the string length only by default. Pass the option
        ",unix-tightsocklen=0" to any abstract socket definition in socat to
        make it compatible with HAProxy's.

  See also: "default-server", "http-send-name-header" and section 5 about
             server options

server-state-file-name [<file>]
  Set the server state file to read, load and apply to servers available in
  this backend. It only applies when the directive "load-server-state-from-file"
  is set to "local". When <file> is not provided or if this directive is not
  set, then backend name is used. If <file> starts with a slash '/', then it is
  considered as an absolute path. Otherwise, <file> is concatenated to the
  global directive "server-state-file-base".

  Example: the minimal configuration below would make HAProxy look for the
           state server file '/etc/haproxy/states/bk':

    global
      server-state-file-base /etc/haproxy/states

   backend bk
      load-server-state-from-file

  See also: "server-state-file-base", "load-server-state-from-file", and
  "show servers state"

source <addr>[:<port>] [usesrc { <addr2>[:<port2>] | client | clientip } ]
source <addr>[:<port>] [usesrc { <addr2>[:<port2>] | hdr_ip(<hdr>[,<occ>]) } ]
source <addr>[:<port>] [interface <name>]
  Set the source address for outgoing connections
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <addr>    is the IPv4 address HAProxy will bind to before connecting to a
              server. This address is also used as a source for health checks.

              The default value of 0.0.0.0 means that the system will select
              the most appropriate address to reach its destination. Optionally
              an address family prefix may be used before the address to force
              the family regardless of the address format, which can be useful
              to specify a path to a unix socket with no slash ('/'). Currently
              supported prefixes are :
                - 'ipv4@' -> address is always IPv4
                - 'ipv6@' -> address is always IPv6
                - 'unix@' -> address is a path to a local unix socket
                - 'abns@' -> address is in abstract namespace (Linux only)
              You may want to reference some environment variables in the
              address parameter, see section 2.3 about environment variables.

    <port>    is an optional port. It is normally not needed but may be useful
              in some very specific contexts. The default value of zero means
              the system will select a free port. Note that port ranges are not
```

supported in the backend. If you want to force port ranges, you have to specify them on each "server" line.

<addr2> is the IP address to present to the server when connections are forwarded in full transparent proxy mode. This is currently only supported on some patched Linux kernels. When this address is specified, clients connecting to the server will be presented with this address, while health checks will still use the address <addr>.

<port2> is the optional port to present to the server when connections are forwarded in full transparent proxy mode (see <addr2> above). The default value of zero means the system will select a free port.

<hdr> is the name of a HTTP header in which to fetch the IP to bind to. This is the name of a comma-separated header list which can contain multiple IP addresses. By default, the last occurrence is used. This is designed to work with the X-Forwarded-For header and to automatically bind to the client's IP address as seen by previous proxy, typically Stunnel. In order to use another occurrence from the last one, please see the <occ> parameter below. When the header (or occurrence) is not found, no binding is performed so that the proxy's default IP address is used. Also keep in mind that the header name is case insensitive, as for any HTTP header.

<occ> is the occurrence number of a value to be used in a multi-value header. This is to be used in conjunction with "hdr_ip(<hdr>)", in order to specify which occurrence to use for the source IP address. Positive values indicate a position from the first occurrence, 1 being the first one. Negative values indicate positions relative to the last one, -1 being the last one. This is helpful for situations where an X-Forwarded-For header is set at the entry point of an infrastructure and must be used several proxy layers away. When this value is not specified, -1 is assumed. Passing a zero here disables the feature.

<name> is an optional interface name to which to bind to for outgoing traffic. On systems supporting this features (currently, only Linux), this allows one to bind all traffic to the server to this interface even if it is not the one the system would select based on routing tables. This should be used with extreme care. Note that using this option requires root privileges.

The "source" keyword is useful in complex environments where a specific address only is allowed to connect to the servers. It may be needed when a private address must be used through a public gateway for instance, and it is known that the system cannot determine the adequate source address by itself.

An extension which is available on certain patched Linux kernels may be used through the "usesrc" optional keyword. It makes it possible to connect to the servers with an IP address which does not belong to the system itself. This is called "full transparent proxy mode". For this to work, the destination servers have to route their traffic back to this address through the machine running HAProxy, and IP forwarding must generally be enabled on this machine.

In this "full transparent proxy" mode, it is possible to force a specific IP address to be presented to the servers. This is not much used in fact. A more common use is to tell HAProxy to present the client's IP address. For this, there are two methods :

- present the client's IP and port addresses. This is the most transparent mode, but it can cause problems when IP connection tracking is enabled on the machine, because a same connection may be seen twice with different states. However, this solution presents the huge advantage of not limiting the system to the 64k outgoing address+port couples, because all of the client ranges may be used.

- present only the client's IP address and select a spare port. This solution is still quite elegant but slightly less transparent (downstream firewalls logs will not match upstream's). It also presents the downside of limiting the number of concurrent connections to the usual 64k ports. However, since the upstream and downstream ports are different, local IP connection tracking on the machine will not be upset by the reuse of the same session.

This option sets the default source for all servers in the backend. It may also be specified in a "defaults" section. Finer source address specification is possible at the server level using the "source" server option. Refer to section 5 for more information.

In order to work, "usesrc" requires root privileges.

Examples :

```
      backend private
          # Connect to the servers using our 192.168.1.200 source address
          source 192.168.1.200

      backend transparent_ssl1
          # Connect to the SSL farm from the client's source address
          source 192.168.1.200 usesrc clientip

      backend transparent_ssl2
          # Connect to the SSL farm from the client's source address and port
          # not recommended if IP conntrack is present on the local machine.
          source 192.168.1.200 usesrc client

      backend transparent_ssl3
          # Connect to the SSL farm from the client's source address. It
          # is more conntrack-friendly.
          source 192.168.1.200 usesrc clientip

      backend transparent_smtp
          # Connect to the SMTP farm from the client's source address/port
          # with Tproxy version 4.
          source 0.0.0.0 usesrc clientip

      backend transparent_http
          # Connect to the servers using the client's IP as seen by previous
          # proxy.
          source 0.0.0.0 usesrc hdr_ip(x-forwarded-for,-1)
```

See also : the "source" server option in section 5, the Tproxy patches for the Linux kernel on www.balabit.com, the "bind" keyword.

srvtimeout <timeout> (deprecated)

Set the maximum inactivity time on the server side.

```
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
```

Arguments :

<timeout> is the timeout value specified in milliseconds by default, but can be in any other unit if the number is suffixed by the unit, as explained at the top of this document.

The inactivity timeout applies when the server is expected to acknowledge or send data. In HTTP mode, this timeout is particularly important to consider during the first phase of the server's response, when it has to send the headers, as it directly represents the server's processing time for the request. To find out what value to put there, it's often good to start with what would be considered as unacceptable response times, then check the logs to observe the response time distribution, and adjust the value accordingly.

The value is specified in milliseconds by default, but can be in any other unit if the number is suffixed by the unit, as specified at the top of this document. In TCP mode (and to a lesser extent, in HTTP mode), it is highly recommended that the client timeout remains equal to the server timeout in order to avoid complex situations to debug. Whatever the expected server response times, it is a good practice to cover at least one or several TCP packet losses by specifying timeouts that are slightly above multiples of 3 seconds (eg: 4 or 5 seconds minimum).

This parameter is specific to backends, but can be specified once for all in "defaults" sections. This is in fact one of the easiest solutions not to forget about it. An unspecified timeout results in an infinite timeout, which is not recommended. Such a usage is accepted and works but reports a warning during startup because it may results in accumulation of expired sessions in the system if the system's timeouts are not configured either.

This parameter is provided for compatibility but is currently deprecated. Please use "timeout server" instead.

```
  See also : "timeout server", "timeout tunnel", "timeout client" and
             "clitimeout".
```

stats admin { if | unless } <cond>

```
  Enable statistics admin level if/unless a condition is matched
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
```

This statement enables the statistics admin level if/unless a condition is matched.

The admin level allows to enable/disable servers from the web interface. By default, statistics page is read-only for security reasons.

```
  Note : Consider not using this feature in multi-process mode (nbproc > 1)
         unless you know what you do : memory is not shared between the
         processes, which can result in random behaviours.
```

Currently, the POST request is limited to the buffer size minus the reserved buffer space, which means that if the list of servers is too long, the request won't be processed. It is recommended to alter few servers at a time.

```
  Example :
    # statistics admin level only for localhost
    backend stats_localhost
        stats enable
        stats admin if LOCALHOST

  Example :
    # statistics admin level always enabled because of the authentication
    backend stats_auth
        stats enable
        stats auth  admin:AdMiN123
        stats admin if TRUE

  Example :
    # statistics admin level depends on the authenticated user
    userlist stats-auth
        group admin    users admin
        user  admin    insecure-password AdMiN123
```

```
        group readonly users haproxy
        user  haproxy  insecure-password haproxy

    backend stats_auth
        stats enable
        acl AUTH       http_auth(stats-auth)
        acl AUTH_ADMIN http_auth_group(stats-auth) admin
        stats http-request auth unless AUTH
        stats admin if AUTH_ADMIN

  See also : "stats enable", "stats auth", "stats http-request", "nbproc",
             "bind-process", section 3.4 about userlists and section 7 about
             ACL usage.
```

stats auth <user>:<passwd>

```
  Enable statistics with authentication and grant access to an account
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <user>    is a user name to grant access to

    <passwd>  is the cleartext password associated to this user
```

This statement enables statistics with default settings, and restricts access to declared users only. It may be repeated as many times as necessary to allow as many users as desired. When a user tries to access the statistics without a valid account, a "401 Forbidden" response will be returned so that the browser asks the user to provide a valid user and password. The real which will be returned to the browser is configurable using "stats realm".

Since the authentication method is HTTP Basic Authentication, the passwords circulate in cleartext on the network. Thus, it was decided that the configuration file would also use cleartext passwords to remind the users that those ones should not be sensitive and not shared with any other account.

It is also possible to reduce the scope of the proxies which appear in the report using "stats scope".

Though this statement alone is enough to enable statistics reporting, it is recommended to set all other settings in order to avoid relying on default unobvious parameters.

```
  Example :
    # public access (limited to this backend only)
    backend public_www
        server srv1 192.168.0.1:80
        stats enable
        stats hide-version
        stats scope   .
        stats uri     /admin?stats
        stats realm   Haproxy\ Statistics
        stats auth    admin1:AdMiN123
        stats auth    admin2:AdMiN321

    # internal monitoring access (unlimited)
    backend private_monitoring
        stats enable
        stats uri     /admin?stats
        stats refresh 5s

  See also : "stats enable", "stats realm", "stats scope", "stats uri"
```

stats enable

```
  Enable statistics reporting with default settings
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  This statement enables statistics reporting with default settings defined
  at build time. Unless stated otherwise, these settings are used :
    - stats uri   : /haproxy?stats
    - stats realm : "HAProxy Statistics"
    - stats auth  : no authentication
    - stats scope : no restriction

  Though this statement alone is enough to enable statistics reporting, it is
  recommended to set all other settings in order to avoid relying on default
  unobvious parameters.

  Example :
    # public access (limited to this backend only)
    backend public_www
        server srv1 192.168.0.1:80
        stats enable
        stats hide-version
        stats scope   .
        stats uri     /admin?stats
        stats realm   Haproxy\ Statistics
        stats auth    admin1:AdMiN123
        stats auth    admin2:AdMiN321

    # internal monitoring access (unlimited)
    backend private_monitoring
        stats enable
        stats uri     /admin?stats
        stats refresh 5s

  See also : "stats auth", "stats realm", "stats uri"


stats hide-version
  Enable statistics and hide HAProxy version reporting
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  By default, the stats page reports some useful status information along with
  the statistics. Among them is HAProxy's version. However, it is generally
  considered dangerous to report precise version to anyone, as it can help them
  target known weaknesses with specific attacks. The "stats hide-version"
  statement removes the version from the statistics report. This is recommended
  for public sites or any site with a weak login/password.

  Though this statement alone is enough to enable statistics reporting, it is
  recommended to set all other settings in order to avoid relying on default
  unobvious parameters.

  Example :
    # public access (limited to this backend only)
    backend public_www
        server srv1 192.168.0.1:80
        stats enable
        stats hide-version
        stats scope   .
        stats uri     /admin?stats
        stats realm   Haproxy\ Statistics
        stats auth    admin1:AdMiN123
        stats auth    admin2:AdMiN321
```

```

    # internal monitoring access (unlimited)
    backend private_monitoring
        stats enable
        stats uri     /admin?stats
        stats refresh 5s

  See also : "stats auth", "stats enable", "stats realm", "stats uri"


stats http-request { allow | deny | auth [realm <realm>] }
             [ { if | unless } <condition> ]
  Access control for statistics

  May be used in sections:   defaults | frontend | listen | backend
                                no    |    no    |   yes  |   yes

  As "http-request", these set of options allow to fine control access to
  statistics. Each option may be followed by if/unless and acl.
  First option with matched condition (or option without condition) is final.
  For "deny" a 403 error will be returned, for "allow" normal processing is
  performed, for "auth" a 401/407 error code is returned so the client
  should be asked to enter a username and password.

  There is no fixed limit to the number of http-request statements per
  instance.

  See also : "http-request", section 3.4 about userlists and section 7
             about ACL usage.


stats realm <realm>
  Enable statistics and set authentication realm
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <realm>   is the name of the HTTP Basic Authentication realm reported to
              the browser. The browser uses it to display it in the pop-up
              inviting the user to enter a valid username and password.

  The realm is read as a single word, so any spaces in it should be escaped
  using a backslash ('\').

  This statement is useful only in conjunction with "stats auth" since it is
  only related to authentication.

  Though this statement alone is enough to enable statistics reporting, it is
  recommended to set all other settings in order to avoid relying on default
  unobvious parameters.

  Example :
    # public access (limited to this backend only)
    backend public_www
        server srv1 192.168.0.1:80
        stats enable
        stats hide-version
        stats scope   .
        stats uri     /admin?stats
        stats realm   Haproxy\ Statistics
        stats auth    admin1:AdMiN123
        stats auth    admin2:AdMiN321

    # internal monitoring access (unlimited)
    backend private_monitoring
        stats enable
        stats uri     /admin?stats
        stats refresh 5s

  See also : "stats auth", "stats enable", "stats uri"


stats refresh <delay>
  Enable statistics with automatic refresh
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <delay>   is the suggested refresh delay, specified in seconds, which will
              be returned to the browser consulting the report page. While the
              browser is free to apply any delay, it will generally respect it
              and refresh the page this every seconds. The refresh interval may
              be specified in any other non-default time unit, by suffixing the
              unit after the value, as explained at the top of this document.

  This statement is useful on monitoring displays with a permanent page
  reporting the load balancer's activity. When set, the HTML report page will
  include a link "refresh"/"stop refresh" so that the user can select whether
  he wants automatic refresh of the page or not.

  Though this statement alone is enough to enable statistics reporting, it is
  recommended to set all other settings in order to avoid relying on default
  unobvious parameters.

  Example :
    # public access (limited to this backend only)
    backend public_www
        server srv1 192.168.0.1:80
        stats enable
        stats hide-version
        stats scope   .
        stats uri     /admin?stats
        stats realm   Haproxy\ Statistics
        stats auth    admin1:AdMiN123
        stats auth    admin2:AdMiN321

    # internal monitoring access (unlimited)
    backend private_monitoring
        stats enable
        stats uri     /admin?stats
        stats refresh 5s

  See also : "stats auth", "stats enable", "stats realm", "stats uri"


stats scope { <name> | "." }
  Enable statistics and limit access scope
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <name>    is the name of a listen, frontend or backend section to be
              reported. The special name "." (a single dot) designates the
              section in which the statement appears.

  When this statement is specified, only the sections enumerated with this
  statement will appear in the report. All other ones will be hidden. This
  statement may appear as many times as needed if multiple sections need to be
  reported. Please note that the name checking is performed as simple string
  comparisons, and that it is never checked that a give section name really
  exists.

  Though this statement alone is enough to enable statistics reporting, it is
```

```
  recommended to set all other settings in order to avoid relying on default
  unobvious parameters.

  Example :
    # public access (limited to this backend only)
    backend public_www
        server srv1 192.168.0.1:80
        stats enable
        stats hide-version
        stats scope   .
        stats uri     /admin?stats
        stats realm   Haproxy\ Statistics
        stats auth    admin1:AdMiN123
        stats auth    admin2:AdMiN321

    # internal monitoring access (unlimited)
    backend private_monitoring
        stats enable
        stats uri     /admin?stats
        stats refresh 5s

  See also : "stats auth", "stats enable", "stats realm", "stats uri"


stats show-desc [ <desc> ]
  Enable reporting of a description on the statistics page.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes

    <desc>    is an optional description to be reported. If unspecified, the
              description from global section is automatically used instead.

  This statement is useful for users that offer shared services to their
  customers, where node or description should be different for each customer.

  Though this statement alone is enough to enable statistics reporting, it is
  recommended to set all other settings in order to avoid relying on default
  unobvious parameters.  By default description is not shown.

  Example :
    # internal monitoring access (unlimited)
    backend private_monitoring
        stats enable
        stats show-desc Master node for Europe, Asia, Africa
        stats uri       /admin?stats
        stats refresh   5s

  See also: "show-node", "stats enable", "stats uri" and "description" in
            global section.


stats show-legends
  Enable reporting additional information on the statistics page
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments : none

  Enable reporting additional information on the statistics page :
    - cap: capabilities (proxy)
    - mode: one of tcp, http or health (proxy)
    - id: SNMP ID (proxy, socket, server)
    - IP (socket, server)
    - cookie (backend, server)

  Though this statement alone is enough to enable statistics reporting, it is
  recommended to set all other settings in order to avoid relying on default
  unobvious parameters.  Default behaviour is not to show this information.

  See also: "stats enable", "stats uri".


stats show-node [ <name> ]
  Enable reporting of a host name on the statistics page.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments:
    <name>    is an optional name to be reported. If unspecified, the
              node name from global section is automatically used instead.

  This statement is useful for users that offer shared services to their
  customers, where node or description might be different on a stats page
  provided for each customer.  Default behaviour is not to show host name.

  Though this statement alone is enough to enable statistics reporting, it is
  recommended to set all other settings in order to avoid relying on default
  unobvious parameters.

  Example:
    # internal monitoring access (unlimited)
    backend private_monitoring
        stats enable
        stats show-node Europe-1
        stats uri       /admin?stats
        stats refresh   5s

  See also: "show-desc", "stats enable", "stats uri", and "node" in global
            section.


stats uri <prefix>
  Enable statistics and define the URI prefix to access them
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <prefix>  is the prefix of any URI which will be redirected to stats. This
              prefix may contain a question mark ('?') to indicate part of a
              query string.

  The statistics URI is intercepted on the relayed traffic, so it appears as a
  page within the normal application. It is strongly advised to ensure that the
  selected URI will never appear in the application, otherwise it will never be
  possible to reach it in the application.

  The default URI compiled in haproxy is "/haproxy?stats", but this may be
  changed at build time, so it's better to always explicitly specify it here.
  It is generally a good idea to include a question mark in the URI so that
  intermediate proxies refrain from caching the results. Also, since any string
  beginning with the prefix will be accepted as a stats request, the question
  mark helps ensuring that no valid URI will begin with the same words.

  It is sometimes very convenient to use "/" as the URI prefix, and put that
  statement in a "listen" instance of its own. That makes it easy to dedicate
  an address or a port to statistics only.

  Though this statement alone is enough to enable statistics reporting, it is
  recommended to set all other settings in order to avoid relying on default
  unobvious parameters.

  Example :
    # public access (limited to this backend only)
```

```
    backend public_www
        server srv1 192.168.0.1:80
        stats enable
        stats hide-version
        stats scope   .
        stats uri     /admin?stats
        stats realm   Haproxy\ Statistics
        stats auth    admin1:AdMiN123
        stats auth    admin2:AdMiN321

    # internal monitoring access (unlimited)
    backend private_monitoring
        stats enable
        stats uri     /admin?stats
        stats refresh 5s

  See also : "stats auth", "stats enable", "stats realm"


stick match <pattern> [table <table>] [{if | unless} <cond>]
  Define a request pattern matching condition to stick a user to a server
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    no    |   yes  |   yes

  Arguments :
    <pattern>  is a sample expression rule as described in section 7.3. It
               describes what elements of the incoming request or connection
               will be analysed in the hope to find a matching entry in a
               stickiness table. This rule is mandatory.

    <table>    is an optional stickiness table name. If unspecified, the same
               backend's table is used. A stickiness table is declared using
               the "stick-table" statement.

    <cond>     is an optional matching condition. It makes it possible to match
               on a certain criterion only when other conditions are met (or
               not met). For instance, it could be used to match on a source IP
               address except when a request passes through a known proxy, in
               which case we'd match on a header containing that IP address.

  Some protocols or applications require complex stickiness rules and cannot
  always simply rely on cookies nor hashing. The "stick match" statement
  describes a rule to extract the stickiness criterion from an incoming request
  or connection. See section 7 for a complete list of possible patterns and
  transformation rules.

  The table has to be declared using the "stick-table" statement. It must be of
  a type compatible with the pattern. By default it is the one which is present
  in the same backend. It is possible to share a table with other backends by
  referencing it using the "table" keyword. If another table is referenced,
  the server's ID inside the backends are used. By default, all server IDs
  start at 1 in each backend, so the server ordering is enough. But in case of
  doubt, it is highly recommended to force server IDs using their "id" setting.

  It is possible to restrict the conditions where a "stick match" statement
  will apply, using "if" or "unless" followed by a condition. See section 7 for
  ACL based conditions.

  There is no limit on the number of "stick match" statements. The first that
  applies and matches will cause the request to be directed to the same server
  as was used for the request which created the entry. That way, multiple
  matches can be used as fallbacks.

  The stick rules are checked after the persistence cookies, so they will not
  affect stickiness if a cookie has already been used to select a server. That
  way, it becomes very easy to insert cookies and match on IP addresses in
  order to maintain stickiness between HTTP and HTTPS.

  Note : Consider not using this feature in multi-process mode (nbproc > 1)
         unless you know what you do : memory is not shared between the
         processes, which can result in random behaviours.

  Example :
    # forward SMTP users to the same server they just used for POP in the
    # last 30 minutes
    backend pop
        mode tcp
        balance roundrobin
        stick store-request src
        stick-table type ip size 200k expire 30m
        server s1 192.168.1.1:110
        server s2 192.168.1.1:110

    backend smtp
        mode tcp
        balance roundrobin
        stick match src table pop
        server s1 192.168.1.1:25
        server s2 192.168.1.1:25

  See also : "stick-table", "stick on", "nbproc", "bind-process" and section 7
             about ACLs and samples fetching.


stick on <pattern> [table <table>] [{if | unless} <condition>]
  Define a request pattern to associate a user to a server
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    no    |   yes  |   yes

  Note : This form is exactly equivalent to "stick match" followed by
         "stick store-request", all with the same arguments. Please refer
         to both keywords for details. It is only provided as a convenience
         for writing more maintainable configurations.

  Note : Consider not using this feature in multi-process mode (nbproc > 1)
         unless you know what you do : memory is not shared between the
         processes, which can result in random behaviours.

  Examples :
    # The following form ...
    stick on src table pop if !localhost

    # ...is strictly equivalent to this one :
    stick match src table pop if !localhost
    stick store-request src table pop if !localhost


    # Use cookie persistence for HTTP, and stick on source address for HTTPS as
    # well as HTTP without cookie. Share the same table between both accesses.
    backend http
        mode http
        balance roundrobin
        stick on src table https
        cookie SRV insert indirect nocache
        server s1 192.168.1.1:80 cookie s1
        server s2 192.168.1.1:80 cookie s2

    backend https
        mode tcp
        balance roundrobin
```

```
        stick-table type ip size 200k expire 30m
        stick on src
        server s1 192.168.1.1:443
        server s2 192.168.1.1:443

  See also : "stick match", "stick store-request", "nbproc" and "bind-process".


stick store-request <pattern> [table <table>] [{if | unless} <condition>]
  Define a request pattern used to create an entry in a stickiness table
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    no    |   yes  |   yes

  Arguments :
    <pattern>  is a sample expression rule as described in section 7.3. It
               describes what elements of the incoming request or connection
               will be analysed, extracted and stored in the table once a
               server is selected.

    <table>    is an optional stickiness table name. If unspecified, the same
               backend's table is used. A stickiness table is declared using
               the "stick-table" statement.

    <cond>     is an optional storage condition. It makes it possible to store
               certain criteria only when some conditions are met (or not met).
               For instance, it could be used to store the source IP address
               except when the request passes through a known proxy, in which
               case we'd store a converted form of a header containing that IP
               address.

  Some protocols or applications require complex stickiness rules and cannot
  always simply rely on cookies nor hashing. The "stick store-request" statement
  describes a rule to decide what to extract from the request and when to do
  it, in order to store it into a stickiness table for further requests to
  match it using the "stick match" statement. Obviously the extracted part must
  make sense and have a chance to be matched in a further request. Storing a
  client's IP address for instance often makes sense. Storing an ID found in a
  URL parameter also makes sense. Storing a source port will almost never make
  any sense because it will be randomly matched. See section 7 for a complete
  list of possible patterns and transformation rules.

  The table has to be declared using the "stick-table" statement. It must be of
  a type compatible with the pattern. By default it is the one which is present
  in the same backend. It is possible to share a table with other backends by
  referencing it using the "table" keyword. If another table is referenced,
  the server's ID inside the backends are used. By default, all server IDs
  start at 1 in each backend, so the server ordering is enough. But in case of
  doubt, it is highly recommended to force server IDs using their "id" setting.

  It is possible to restrict the conditions where a "stick store-request"
  statement will apply, using "if" or "unless" followed by a condition. This
  condition will be evaluated while parsing the request, so any criteria can be
  used. See section 7 for ACL based conditions.

  There is no limit on the number of "stick store-request" statements, but
  there is a limit of 8 simultaneous stores per request or response. This
  makes it possible to store up to 8 criteria, all extracted from either the
  request or the response, regardless of the number of rules. Only the 8 first
  ones which match will be kept. Using this, it is possible to feed multiple
  tables at once in the hope to increase the chance to recognize a user on
  another protocol or access method. Using multiple store-request rules with
  the same table is possible and may be used to find the best criterion to rely
  on, by arranging the rules by decreasing preference order. Only the first
  extracted criterion for a given table will be stored. All subsequent store-
  request rules referencing the same table will be skipped and their ACLs will
  not be evaluated.

  The "store-request" rules are evaluated once the server connection has been
  established, so that the table will contain the real server that processed
  the request.

  Note : Consider not using this feature in multi-process mode (nbproc > 1)
         unless you know what you do : memory is not shared between the
         processes, which can result in random behaviours.

  Example :
    # forward SMTP users to the same server they just used for POP in the
    # last 30 minutes
    backend pop
        mode tcp
        balance roundrobin
        stick store-request src
        stick-table type ip size 200k expire 30m
        server s1 192.168.1.1:110
        server s2 192.168.1.1:110

    backend smtp
        mode tcp
        balance roundrobin
        stick match src table pop
        server s1 192.168.1.1:25
        server s2 192.168.1.1:25

  See also : "stick-table", "stick on", "nbproc", "bind-process" and section 7
             about ACLs and sample fetching.


stick-table type {ip | integer | string [len <length>] | binary [len <length>]}
            size <size> [expire <expire>] [nopurge] [peers <peersect>]
            [store <data_type>]*
  Configure the stickiness table for the current section
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes

  Arguments :
    ip         a table declared with "type ip" will only store IPv4 addresses.
               This form is very compact (about 50 bytes per entry) and allows
               very fast entry lookup and stores with almost no overhead. This
               is mainly used to store client source IP addresses.

    ipv6       a table declared with "type ipv6" will only store IPv6 addresses.
               This form is very compact (about 60 bytes per entry) and allows
               very fast entry lookup and stores with almost no overhead. This
               is mainly used to store client source IP addresses.

    integer    a table declared with "type integer" will store 32bit integers
               which can represent a client identifier found in a request for
               instance.

    string     a table declared with "type string" will store substrings of up
               to <len> characters. If the string provided by the pattern
               extractor is larger than <len>, it will be truncated before
               being stored. During matching, at most <len> characters will be
               compared between the string in the table and the extracted
               pattern. When not specified, the string is automatically limited
               to 32 characters.

    binary     a table declared with "type binary" will store binary blocks
               of <len> bytes. If the block provided by the pattern
               extractor is larger than <len>, it will be truncated before
```

```
               being stored. If the block provided by the sample expression
               is shorter than <len>, it will be padded by 0. When not
               specified, the block is automatically limited to 32 bytes.

    <length>   is the maximum number of characters that will be stored in a
               "string" type table (See type "string" above). Or the number
               of bytes of the block in "binary" type table. Be careful when
               changing this parameter as memory usage will proportionally
               increase.

    <size>     is the maximum number of entries that can fit in the table. This
               value directly impacts memory usage. Count approximately
               50 bytes per entry, plus the size of a string if any. The size
               supports suffixes "k", "m", "g" for 2^10, 2^20 and 2^30 factors.

    [nopurge]  indicates that we refuse to purge older entries when the table
               is full. When not specified and the table is full when haproxy
               wants to store an entry in it, it will flush a few of the oldest
               entries in order to release some space for the new ones. This is
               most often the desired behaviour. In some specific cases, it
               be desirable to refuse new entries instead of purging the older
               ones. That may be the case when the amount of data to store is
               far above the hardware limits and we prefer not to offer access
               to new clients than to reject the ones already connected. When
               using this parameter, be sure to properly set the "expire"
               parameter (see below).

    <peersect> is the name of the peers section to use for replication. Entries
               which associate keys to server IDs are kept synchronized with
               the remote peers declared in this section. All entries are also
               automatically learned from the local peer (old process) during a
               soft restart.

               NOTE : each peers section may be referenced only by tables
                      belonging to the same unique process.

    <expire>   defines the maximum duration of an entry in the table since it
               was last created, refreshed or matched. The expiration delay is
               defined using the standard time format, similarly as the various
               timeouts. The maximum duration is slightly above 24 days. See
               section 2.2 for more information. If this delay is not specified,
               the session won't automatically expire, but older entries will
               be removed once full. Be sure not to use the "nopurge" parameter
               if not expiration delay is specified.

   <data_type> is used to store additional information in the stick-table. This
               may be used by ACLs in order to control various criteria related
               to the activity of the client matching the stick-table. For each
               item specified here, the size of each entry will be inflated so
               that the additional data can fit. Several data types may be
               stored with an entry. Multiple data types may be specified after
               the "store" keyword, as a comma-separated list. Alternatively,
               it is possible to repeat the "store" keyword followed by one or
               several data types. Except for the "server_id" type which is
               automatically detected and enabled, all data types must be
               explicitly declared to be stored. If an ACL references a data
               type which is not stored, the ACL will simply not match. Some
               data types require an argument which must be passed just after
               the type between parenthesis. See below for the supported data
               types and their arguments.

  The data types that can be stored with an entry are the following :
    - server_id : this is an integer which holds the numeric ID of the server a
      request was assigned to. It is used by the "stick match", "stick store",
      and "stick on" rules. It is automatically enabled when referenced.

    - gpc0 : first General Purpose Counter. It is a positive 32-bit integer
      integer which may be used for anything. Most of the time it will be used
      to put a special tag on some entries, for instance to note that a
      specific behaviour was detected and must be known for future matches.

    - gpc0_rate(<period>) : increment rate of the first General Purpose Counter
      over a period. It is a positive 32-bit integer integer which may be used
      for anything. Just like <gpc0>, it counts events, but instead of keeping
      a cumulative count, it maintains the rate at which the counter is
      incremented. Most of the time it will be used to measure the frequency of
      occurrence of certain events (eg: requests to a specific URL).

    - conn_cnt : Connection Count. It is a positive 32-bit integer which counts
      the absolute number of connections received from clients which matched
      this entry. It does not mean the connections were accepted, just that
      they were received.

    - conn_cur : Current Connections. It is a positive 32-bit integer which
      stores the concurrent connection counts for the entry. It is incremented
      once an incoming connection matches the entry, and decremented once the
      connection leaves. That way it is possible to know at any time the exact
      number of concurrent connections for an entry.

    - conn_rate(<period>) : frequency counter (takes 12 bytes). It takes an
      integer parameter <period> which indicates in milliseconds the length
      of the period over which the average is measured. It reports the average
      incoming connection rate over that period, in connections per period. The
      result is an integer which can be matched using ACLs.

    - sess_cnt : Session Count. It is a positive 32-bit integer which counts
      the absolute number of sessions received from clients which matched this
      entry. A session is a connection that was accepted by the layer 4 rules.

    - sess_rate(<period>) : frequency counter (takes 12 bytes). It takes an
      integer parameter <period> which indicates in milliseconds the length
      of the period over which the average is measured. It reports the average
      incoming session rate over that period, in sessions per period. The
      result is an integer which can be matched using ACLs.

    - http_req_cnt : HTTP request Count. It is a positive 32-bit integer which
      counts the absolute number of HTTP requests received from clients which
      matched this entry. It does not matter whether they are valid requests or
      not. Note that this is different from sessions when keep-alive is used on
      the client side.

    - http_req_rate(<period>) : frequency counter (takes 12 bytes). It takes an
      integer parameter <period> which indicates in milliseconds the length
      of the period over which the average is measured. It reports the average
      HTTP request rate over that period, in requests per period. The result is
      an integer which can be matched using ACLs. It does not matter whether
      they are valid requests or not. Note that this is different from sessions
      when keep-alive is used on the client side.

    - http_err_cnt : HTTP Error Count. It is a positive 32-bit integer which
      counts the absolute number of HTTP requests errors induced by clients
      which matched this entry. Errors are counted on invalid and truncated
      requests, as well as on denied or tarpitted requests, and on failed
      authentications. If the server responds with 4xx, then the request is
      also counted as an error since it's an error triggered by the client
      (eg: vulnerability scan).

    - http_err_rate(<period>) : frequency counter (takes 12 bytes). It takes an
      integer parameter <period> which indicates in milliseconds the length
      of the period over which the average is measured. It reports the average
```

```
      HTTP request error rate over that period, in requests per period (see
      http_err_cnt above for what is accounted as an error). The result is an
      integer which can be matched using ACLs.

    - bytes_in_cnt : client to server byte count. It is a positive 64-bit
      integer which counts the cumulated amount of bytes received from clients
      which matched this entry. Headers are included in the count. This may be
      used to limit abuse of upload features on photo or video servers.

    - bytes_in_rate(<period>) : frequency counter (takes 12 bytes). It takes an
      integer parameter <period> which indicates in milliseconds the length
      of the period over which the average is measured. It reports the average
      incoming bytes rate over that period, in bytes per period. It may be used
      to detect users which upload too much and too fast. Warning: with large
      uploads, it is possible that the amount of uploaded data will be counted
      once upon termination, thus causing spikes in the average transfer speed
      instead of having a smooth one. This may partially be smoothed with
      "option contstats" though this is not perfect yet. Use of byte_in_cnt is
      recommended for better fairness.

    - bytes_out_cnt : server to client byte count. It is a positive 64-bit
      integer which counts the cumulated amount of bytes sent to clients which
      matched this entry. Headers are included in the count. This may be used
      to limit abuse of bots sucking the whole site.

    - bytes_out_rate(<period>) : frequency counter (takes 12 bytes). It takes
      an integer parameter <period> which indicates in milliseconds the length
      of the period over which the average is measured. It reports the average
      outgoing bytes rate over that period, in bytes per period. It may be used
      to detect users which download too much and too fast. Warning: with large
      transfers, it is possible that the amount of transferred data will be
      counted once upon termination, thus causing spikes in the average
      transfer speed instead of having a smooth one. This may partially be
      smoothed with "option contstats" though this is not perfect yet. Use of
      byte_out_cnt is recommended for better fairness.

  There is only one stick-table per proxy. At the moment of writing this doc,
  it does not seem useful to have multiple tables per proxy. If this happens
  to be required, simply create a dummy backend with a stick-table in it and
  reference it.

  It is important to understand that stickiness based on learning information
  has some limitations, including the fact that all learned associations are
  lost upon restart. In general it can be good as a complement but not always
  as an exclusive stickiness.

  Last, memory requirements may be important when storing many data types.
  Indeed, storing all indicators above at once in each entry requires 116 bytes
  per entry, or 116 MB for a 1-million entries table. This is definitely not
  something that can be ignored.

  Example:
        # Keep track of counters of up to 1 million IP addresses over 5 minutes
        # and store a general purpose counter and the average connection rate
        # computed over a sliding window of 30 seconds.
        stick-table type ip size 1m expire 5m store gpc0,conn_rate(30s)

  See also : "stick match", "stick on", "stick store-request", section 2.2
             about time format and section 7 about ACLs.


stick store-response <pattern> [table <table>] [{if | unless} <condition>]
  Define a request pattern used to create an entry in a stickiness table
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    no    |   yes  |   yes

  Arguments :
    <pattern>  is a sample expression rule as described in section 7.3. It
               describes what elements of the response or connection will
               be analysed, extracted and stored in the table once a
               server is selected.

    <table>    is an optional stickiness table name. If unspecified, the same
               backend's table is used. A stickiness table is declared using
               the "stick-table" statement.

    <cond>     is an optional storage condition. It makes it possible to store
               certain criteria only when some conditions are met (or not met).
               For instance, it could be used to store the SSL session ID only
               when the response is a SSL server hello.

  Some protocols or applications require complex stickiness rules and cannot
  always simply rely on cookies nor hashing. The "stick store-response"
  statement  describes a rule to decide what to extract from the response and
  when to do it, in order to store it into a stickiness table for further
  requests to match it using the "stick match" statement. Obviously the
  extracted part must make sense and have a chance to be matched in a further
  request. Storing an ID found in a header of a response makes sense.
  See section 7 for a complete list of possible patterns and transformation
  rules.

  The table has to be declared using the "stick-table" statement. It must be of
  a type compatible with the pattern. By default it is the one which is present
  in the same backend. It is possible to share a table with other backends by
  referencing it using the "table" keyword. If another table is referenced,
  the server's ID inside the backends are used. By default, all server IDs
  start at 1 in each backend, so the server ordering is enough. But in case of
  doubt, it is highly recommended to force server IDs using their "id" setting.

  It is possible to restrict the conditions where a "stick store-response"
  statement will apply, using "if" or "unless" followed by a condition. This
  condition will be evaluated while parsing the response, so any criteria can
  be used. See section 7 for ACL based conditions.

  There is no limit on the number of "stick store-response" statements, but
  there is a limit of 8 simultaneous stores per request or response. This
  makes it possible to store up to 8 criteria, all extracted from either the
  request or the response, regardless of the number of rules. Only the 8 first
  ones which match will be kept. Using this, it is possible to feed multiple
  tables at once in the hope to increase the chance to recognize a user on
  another protocol or access method. Using multiple store-response rules with
  the same table is possible and may be used to find the best criterion to rely
  on, by arranging the rules by decreasing preference order. Only the first
  extracted criterion for a given table will be stored. All subsequent store-
  response rules referencing the same table will be skipped and their ACLs will
  not be evaluated. However, even if a store-request rule references a table, a
  store-response rule may also use the same table. This means that each table
  may learn exactly one element from the request and one element from the
  response at once.

  The table will contain the real server that processed the request.

  Example :
    # Learn SSL session ID from both request and response and create affinity.
    backend https
        mode tcp
        balance roundrobin
        # maximum SSL session ID length is 32 bytes.
        stick-table type binary len 32 size 30k expire 30m

```

```
        acl clienthello req_ssl_hello_type 1
        acl serverhello rep_ssl_hello_type 2

        # use tcp content accepts to detects ssl client and server hello.
        tcp-request inspect-delay 5s
        tcp-request content accept if clienthello

        # no timeout on response inspect delay by default.
        tcp-response content accept if serverhello

        # SSL session ID (SSLID) may be present on a client or server hello.
        # Its length is coded on 1 byte at offset 43 and its value starts
        # at offset 44.

        # Match and learn on request if client hello.
        stick on payload_lv(43,1) if clienthello

        # Learn on response if server hello.
        stick store-response payload_lv(43,1) if serverhello

        server s1 192.168.1.1:443
        server s2 192.168.1.1:443

  See also : "stick-table", "stick on", and section 7 about ACLs and pattern
             extraction.


tcp-check connect [params*]
  Opens a new connection
  May be used in sections:   defaults | frontend | listen | backend
                               no     |    no    |   yes  |   yes

  When an application lies on more than a single TCP port or when HAProxy
  load-balance many services in a single backend, it makes sense to probe all
  the services individually before considering a server as operational.

  When there are no TCP port configured on the server line neither server port
  directive, then the 'tcp-check connect port <port>' must be the first step
  of the sequence.

  In a tcp-check ruleset a 'connect' is required, it is also mandatory to start
  the ruleset with a 'connect' rule. Purpose is to ensure admin know what they
  do.

  Parameters :
    They are optional and can be used to describe how HAProxy should open and
    use the TCP connection.

    port      if not set, check port or server port is used.
              It tells HAProxy where to open the connection to.
              <port> must be a valid TCP port source integer, from 1 to 65535.

    send-proxy   send a PROXY protocol string

    ssl          opens a ciphered connection

    Examples:
         # check HTTP and HTTPs services on a server.
         # first open port 80 thanks to server line port directive, then
         # tcp-check opens port 443, ciphered and run a request on it:
         option tcp-check
         tcp-check connect
         tcp-check send GET\ /\ HTTP/1.0\r\n
         tcp-check send Host:\ haproxy.1wt.eu\r\n
         tcp-check send \r\n
         tcp-check expect rstring (2..|3..)
         tcp-check connect port 443 ssl
         tcp-check send GET\ /\ HTTP/1.0\r\n
         tcp-check send Host:\ haproxy.1wt.eu\r\n
         tcp-check send \r\n
         tcp-check expect rstring (2..|3..)
         server www 10.0.0.1 check port 80

         # check both POP and IMAP from a single server:
         option tcp-check
         tcp-check connect port 110
         tcp-check expect string +OK\ POP3\ ready
         tcp-check connect port 143
         tcp-check expect string *\ OK\ IMAP4\ ready
         server mail 10.0.0.1 check

  See also : "option tcp-check", "tcp-check send", "tcp-check expect"


tcp-check expect [!] <match> <pattern>
  Specify data to be collected and analysed during a generic health check
  May be used in sections:   defaults | frontend | listen | backend
                               no     |    no    |   yes  |   yes

  Arguments :
    <match>   is a keyword indicating how to look for a specific pattern in the
              response. The keyword may be one of "string", "rstring" or
              binary.
              The keyword may be preceded by an exclamation mark ("!") to negate
              the match. Spaces are allowed between the exclamation mark and the
              keyword. See below for more details on the supported keywords.

    <pattern> is the pattern to look for. It may be a string or a regular
              expression. If the pattern contains spaces, they must be escaped
              with the usual backslash ('\').
              If the match is set to binary, then the pattern must be passed as
              a serie of hexadecimal digits in an even number. Each sequence of
              two digits will represent a byte. The hexadecimal digits may be
              used upper or lower case.


  The available matches are intentionally similar to their http-check cousins :

    string <string> : test the exact string matches in the response buffer.
                      A health check response will be considered valid if the
                      response's buffer contains this exact string. If the
                      "string" keyword is prefixed with "!", then the response
                      will be considered invalid if the body contains this
                      string. This can be used to look for a mandatory pattern
                      in a protocol response, or to detect a failure when a
                      specific error appears in a protocol banner.

    rstring <regex> : test a regular expression on the response buffer.
                      A health check response will be considered valid if the
                      response's buffer matches this expression. If the
                      "rstring" keyword is prefixed with "!", then the response
                      will be considered invalid if the body matches the
                      expression.

    binary <hexstring> : test the exact string in its hexadecimal form matches
                         in the response buffer. A health check response will
                         be considered valid if the response's buffer contains
                         this exact hexadecimal string.
                         Purpose is to match data on binary protocols.

```

```
  It is important to note that the responses will be limited to a certain size
  defined by the global "tune.chksize" option, which defaults to 16384 bytes.
  Thus, too large responses may not contain the mandatory pattern when using
  "string", "rstring" or binary. If a large response is absolutely required, it
  is possible to change the default max size by setting the global variable.
  However, it is worth keeping in mind that parsing very large responses can
  waste some CPU cycles, especially when regular expressions are used, and that
  it is always better to focus the checks on smaller resources. Also, in its
  current state, the check will not find any string nor regex past a null
  character in the response. Similarly it is not possible to request matching
  the null character.

  Examples :
         # perform a POP check
         option tcp-check
         tcp-check expect string +OK\ POP3\ ready

         # perform an IMAP check
         option tcp-check
         tcp-check expect string *\ OK\ IMAP4\ ready

         # look for the redis master server
         option tcp-check
         tcp-check send PING\r\n
         tcp-check expect string +PONG
         tcp-check send info\ replication\r\n
         tcp-check expect string role:master
         tcp-check send QUIT\r\n
         tcp-check expect string +OK


  See also : "option tcp-check", "tcp-check connect", "tcp-check send",
             "tcp-check send-binary", "http-check expect", tune.chksize


tcp-check send <data>
  Specify a string to be sent as a question during a generic health check
  May be used in sections:   defaults | frontend | listen | backend
                               no     |    no    |   yes  |   yes

    <data> : the data to be sent as a question during a generic health check
             session. For now, <data> must be a string.

  Examples :
         # look for the redis master server
         option tcp-check
         tcp-check send info\ replication\r\n
         tcp-check expect string role:master

  See also : "option tcp-check", "tcp-check connect", "tcp-check expect",
             "tcp-check send-binary", tune.chksize


tcp-check send-binary <hexstring>
  Specify an hexa digits string to be sent as a binary question during a raw
  tcp health check
  May be used in sections:   defaults | frontend | listen | backend
                               no     |    no    |   yes  |   yes

    <data> : the data to be sent as a question during a generic health check
             session. For now, <data> must be a string.
    <hexstring> : test the exact string in its hexadecimal form matches in the
                  response buffer. A health check response will be considered
                  valid if the response's buffer contains this exact
                  hexadecimal string.
                  Purpose is to send binary data to ask on binary protocols.

  Examples :
         # redis check in binary
         option tcp-check
         tcp-check send-binary 50494e470d0a # PING\r\n
         tcp-check expect binary 2b504F4e47 # +PONG


  See also : "option tcp-check", "tcp-check connect", "tcp-check expect",
             "tcp-check send", tune.chksize


tcp-request connection <action> [{if | unless} <condition>]
  Perform an action on an incoming connection depending on a layer 4 condition
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   no
  Arguments :
    <action>    defines the action to perform if the condition applies. See
                below.

    <condition> is a standard layer4-only ACL-based condition (see section 7).

  Immediately after acceptance of a new incoming connection, it is possible to
  evaluate some conditions to decide whether this connection must be accepted
  or dropped or have its counters tracked. Those conditions cannot make use of
  any data contents because the connection has not been read from yet, and the
  buffers are not yet allocated. This is used to selectively and very quickly
  accept or drop connections from various sources with a very low overhead. If
  some contents need to be inspected in order to take the decision, the
  "tcp-request content" statements must be used instead.

  The "tcp-request connection" rules are evaluated in their exact declaration
  order. If no rule matches or if there is no rule, the default action is to
  accept the incoming connection. There is no specific limit to the number of
  rules which may be inserted.

  Four types of actions are supported :
    - accept :
        accepts the connection if the condition is true (when used with "if")
        or false (when used with "unless"). The first such rule executed ends
        the rules evaluation.

    - reject :
        rejects the connection if the condition is true (when used with "if")
        or false (when used with "unless"). The first such rule executed ends
        the rules evaluation. Rejected connections do not even become a
        session, which is why they are accounted separately for in the stats,
        as "denied connections". They are not considered for the session
        rate-limit and are not logged either. The reason is that these rules
        should only be used to filter extremely high connection rates such as
        the ones encountered during a massive DDoS attack. Under these extreme
        conditions, the simple action of logging each event would make the
        system collapse and would considerably lower the filtering capacity. If
        logging is absolutely desired, then "tcp-request content" rules should
        be used instead.

    - expect-proxy layer4 :
        configures the client-facing connection to receive a PROXY protocol
        header before any byte is read from the socket. This is equivalent to
        having the "accept-proxy" keyword on the "bind" line, except that using
        the TCP rule allows the PROXY protocol to be accepted only for certain
        IP address ranges using an ACL. This is convenient when multiple layers
        of load balancers are passed through by traffic coming from public
        hosts.
```

```

    - capture <sample> len <length> :
        This only applies to "tcp-request content" rules. It captures sample
        expression <sample> from the request buffer, and converts it to a
        string of at most <len> characters. The resulting string is stored into
        the next request "capture" slot, so it will possibly appear next to
        some captured HTTP headers. It will then automatically appear in the
        logs, and it will be possible to extract it using sample fetch rules to
        feed it into headers or anything. The length should be limited given
        that this size will be allocated for each capture during the whole
        session life. Please check section 7.3 (Fetching samples) and "capture
        request header" for more information.

    - { track-sc0 | track-sc1 | track-sc2 } <key> [table <table>] :
        enables tracking of sticky counters from current connection. These
        rules do not stop evaluation and do not change default action. 3 sets
        of counters may be simultaneously tracked by the same connection. The
        first "track-sc0" rule executed enables tracking of the counters of the
        specified table as the first set. The first "track-sc1" rule executed
        enables tracking of the counters of the specified table as the second
        set. The first "track-sc2" rule executed enables tracking of the
        counters of the specified table as the third set. It is a recommended
        practice to use the first set of counters for the per-frontend counters
        and the second set for the per-backend ones. But this is just a
        guideline, all may be used everywhere.

        These actions take one or two arguments :
          <key>   is mandatory, and is a sample expression rule as described
                  in section 7.3. It describes what elements of the incoming
                  request or connection will be analysed, extracted, combined,
                  and used to select which table entry to update the counters.
                  Note that "tcp-request connection" cannot use content-based
                  fetches.

         <table>  is an optional table to be used instead of the default one,
                  which is the stick-table declared in the current proxy. All
                  the counters for the matches and updates for the key will
                  then be performed in that table until the session ends.

        Once a "track-sc*" rule is executed, the key is looked up in the table
        and if it is not found, an entry is allocated for it. Then a pointer to
        that entry is kept during all the session's life, and this entry's
        counters are updated as often as possible, every time the session's
        counters are updated, and also systematically when the session ends.
        Counters are only updated for events that happen after the tracking has
        been started. For example, connection counters will not be updated when
        tracking layer 7 information, since the connection event happens before
        layer7 information is extracted.

        If the entry tracks concurrent connection counters, one connection is
        counted for as long as the entry is tracked, and the entry will not
        expire during that time. Tracking counters also provides a performance
        advantage over just checking the keys, because only one table lookup is
        performed for all ACL checks that make use of it.

    - sc-inc-gpc0(<sc-id>):
        The "sc-inc-gpc0" increments the GPC0 counter according to the sticky
        counter designated by <sc-id>. If an error occurs, this action silently
        fails and the actions evaluation continues.

    - sc-set-gpt0(<sc-id>) <int>:
        This action sets the GPT0 tag according to the sticky counter designated
        by <sc-id> and the value of <int>. The expected result is a boolean. If
        an error occurs, this action silently fails and the actions evaluation
        continues.
```

- "silent-drop" :
  This stops the evaluation of the rules and makes the client-facing connection suddenly disappear using a system-dependant way that tries to prevent the client from being notified. The effect it then that the client still sees an established connection while there's none on HAProxy. The purpose is to achieve a comparable effect to "tarpit" except that it doesn't use any local resource at all on the machine running HAProxy. It can resist much higher loads than "tarpit", and slow down stronger attackers. It is important to undestand the impact of using this mechanism. All stateful equipments placed between the client and HAProxy (firewalls, proxies, load balancers) will also keep the established connection for a long time and may suffer from this action. On modern Linux systems running with enough privileges, the TCP_REPAIR socket option is used to block the emission of a TCP reset. On other systems, the socket's TTL is reduced to 1 so that the TCP reset doesn't pass the first router, though it's still delivered to local networks. Do not use it unless you fully understand how it works.

Note that the "if/unless" condition is optional. If no condition is set on the action, it is simply performed unconditionally. That can be useful for "track-sc*" actions as well as for changing the default action to a reject.

Example: accept all connections from white-listed hosts, reject too fast connection without counting them, and track accepted connections. This results in connection rate being capped from abusive sources.

```
tcp-request connection accept if { src -f /etc/haproxy/whitelist.lst }
tcp-request connection reject if { src_conn_rate gt 10 }
tcp-request connection track-sc0 src
```

Example: accept all connections from white-listed hosts, count all other connections and reject too fast ones. This results in abusive ones being blocked as long as they don't slow down.

```
tcp-request connection accept if { src -f /etc/haproxy/whitelist.lst }
tcp-request connection track-sc0 src
tcp-request connection reject if { sc0_conn_rate gt 10 }
```

Example: enable the PROXY protocol for traffic coming from all known proxies.

```
tcp-request connection expect-proxy layer4 if { src -f proxies.lst }
```

See section 7 about ACL usage.

See also : "tcp-request content", "stick-table"

```
tcp-request content <action> [{if | unless} <condition>]
  Perform an action on a new session depending on a layer 4-7 condition
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <action>    defines the action to perform if the condition applies. See
                below.

    <condition> is a standard layer 4-7 ACL-based condition (see section 7).
```

A request's contents can be analysed at an early stage of request processing called "TCP content inspection". During this stage, ACL-based rules are evaluated every time the request contents are updated, until either an "accept" or a "reject" rule matches, or the TCP request inspection delay expires with no matching rule.

The first difference between these rules and "tcp-request connection" rules

is that "tcp-request content" rules can make use of contents to take a decision. Most often, these decisions will consider a protocol recognition or validity. The second difference is that content-based rules can be used in both frontends and backends. In case of HTTP keep-alive with the client, all tcp-request content rules are evaluated again, so haproxy keeps a record of what sticky counters were assigned by a "tcp-request connection" versus a "tcp-request content" rule, and flushes all the content-related ones after processing an HTTP request, so that they may be evaluated again by the rules being evaluated again for the next request. This is of particular importance when the rule tracks some L7 information or when it is conditioned by an L7-based ACL, since tracking may change between requests.

Content-based rules are evaluated in their exact declaration order. If no rule matches or if there is no rule, the default action is to accept the contents. There is no specific limit to the number of rules which may be inserted.

Several types of actions are supported :

- accept : the request is accepted
- reject : the request is rejected and the connection is closed
- capture : the specified sample expression is captured
- { track-sc0 | track-sc1 | track-sc2 } <key> [table <table>]
- sc-inc-gpc0(<sc-id>)
- set-gpt0(<sc-id>) <int>
- set-var(<var-name>) <expr>
- silent-drop

They have the same meaning as their counter-parts in "tcp-request connection" so please refer to that section for a complete description.

While there is nothing mandatory about it, it is recommended to use the track-sc0 in "tcp-request connection" rules, track-sc1 for "tcp-request content" rules in the frontend, and track-sc2 for "tcp-request content" rules in the backend, because that makes the configuration more readable and easier to troubleshoot, but this is just a guideline and all counters may be used everywhere.

Note that the "if/unless" condition is optional. If no condition is set on the action, it is simply performed unconditionally. That can be useful for "track-sc*" actions as well as for changing the default action to a reject.

It is perfectly possible to match layer 7 contents with "tcp-request content" rules, since HTTP-specific ACL matches are able to preliminarily parse the contents of a buffer before extracting the required data. If the buffered contents do not parse as a valid HTTP message, then the ACL does not match. The parser which is involved there is exactly the same as for all other HTTP processing, so there is no risk of parsing something differently. In an HTTP backend connected to from an HTTP frontend, it is guaranteed that HTTP contents will always be immediately present when the rule is evaluated first.

Tracking layer7 information is also possible provided that the information are present when the rule is processed. The rule processing engine is able to wait until the inspect delay expires when the data to be tracked is not yet available.

The "set-var" is used to set the content of a variable. The variable is declared inline.

```
  <var-name> The name of the variable starts by an indication about its scope.
             The allowed scopes are:
               "sess" : the variable is shared with all the session,
               "txn"  : the variable is shared with all the transaction
                        (request and response)
               "req"  : the variable is shared only during the request
                        processing
```

```
                 "res"  : the variable is shared only during the response
                          processing.
               This prefix is followed by a name. The separator is a '.'.
               The name may only contain characters 'a-z', 'A-Z', '0-9' and '_'.

    <expr>     Is a standard HAProxy expression formed by a sample-fetch
               followed by some converters.

  Example:

        tcp-request content set-var(sess.my_var) src

  Example:
        # Accept HTTP requests containing a Host header saying "example.com"
        # and reject everything else.
        acl is_host_com hdr(Host) -i example.com
        tcp-request inspect-delay 30s
        tcp-request content accept if is_host_com
        tcp-request content reject

  Example:
        # reject SMTP connection if client speaks first
        tcp-request inspect-delay 30s
        acl content_present req_len gt 0
        tcp-request content reject if content_present

        # Forward HTTPS connection only if client speaks
        tcp-request inspect-delay 30s
        acl content_present req_len gt 0
        tcp-request content accept if content_present
        tcp-request content reject

  Example:
        # Track the last IP from X-Forwarded-For
        tcp-request inspect-delay 10s
        tcp-request content track-sc0 hdr(x-forwarded-for,-1)

  Example:
        # track request counts per "base" (concatenation of Host+URL)
        tcp-request inspect-delay 10s
        tcp-request content track-sc0 base table req-rate

  Example: track per-frontend and per-backend counters, block abusers at the
           frontend when the backend detects abuse.

        frontend http
            # Use General Purpose Couter 0 in SC0 as a global abuse counter
            # protecting all our sites
            stick-table type ip size 1m expire 5m store gpc0
            tcp-request connection track-sc0 src
            tcp-request connection reject if { sc0_get_gpc0 gt 0 }
            ...
            use_backend http_dynamic if { path_end .php }

        backend http_dynamic
            # if a source makes too fast requests to this dynamic site (tracked
            # by SC1), block it globally in the frontend.
            stick-table type ip size 1m expire 5m store http_req_rate(10s)
            acl click_too_fast sc1_http_req_rate gt 10
            acl mark_as_abuser sc0_inc_gpc0 gt 0
            tcp-request content track-sc1 src
            tcp-request content reject if click_too_fast mark_as_abuser

  See section 7 about ACL usage.
```

```
  See also : "tcp-request connection", "tcp-request inspect-delay"


tcp-request inspect-delay <timeout>
  Set the maximum allowed time to wait for data during content inspection
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    yes   |   yes  |   yes
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  People using haproxy primarily as a TCP relay are often worried about the
  risk of passing any type of protocol to a server without any analysis. In
  order to be able to analyze the request contents, we must first withhold
  the data then analyze them. This statement simply enables withholding of
  data for at most the specified amount of time.

  TCP content inspection applies very early when a connection reaches a
  frontend, then very early when the connection is forwarded to a backend. This
  means that a connection may experience a first delay in the frontend and a
  second delay in the backend if both have tcp-request rules.

  Note that when performing content inspection, haproxy will evaluate the whole
  rules for every new chunk which gets in, taking into account the fact that
  those data are partial. If no rule matches before the aforementioned delay,
  a last check is performed upon expiration, this time considering that the
  contents are definitive. If no delay is set, haproxy will not wait at all
  and will immediately apply a verdict based on the available information.
  Obviously this is unlikely to be very useful and might even be racy, so such
  setups are not recommended.

  As soon as a rule matches, the request is released and continues as usual. If
  the timeout is reached and no rule matches, the default policy will be to let
  it pass through unaffected.

  For most protocols, it is enough to set it to a few seconds, as most clients
  send the full request immediately upon connection. Add 3 or more seconds to
  cover TCP retransmits but that's all. For some protocols, it may make sense
  to use large values, for instance to ensure that the client never talks
  before the server (eg: SMTP), or to wait for a client to talk before passing
  data to the server (eg: SSL). Note that the client timeout must cover at
  least the inspection delay, otherwise it will expire first. If the client
  closes the connection or if the buffer is full, the delay immediately expires
  since the contents will not be able to change anymore.

  See also : "tcp-request content accept", "tcp-request content reject",
             "timeout client".


tcp-response content <action> [{if | unless} <condition>]
  Perform an action on a session response depending on a layer 4-7 condition
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    no    |   yes  |   yes
  Arguments :
    <action>    defines the action to perform if the condition applies. See
                below.

    <condition> is a standard layer 4-7 ACL-based condition (see section 7).

  Response contents can be analysed at an early stage of response processing
  called "TCP content inspection". During this stage, ACL-based rules are
  evaluated every time the response contents are updated, until either an
  "accept", "close" or a "reject" rule matches, or a TCP response inspection
  delay is set and expires with no matching rule.

  Most often, these decisions will consider a protocol recognition or validity.

  Content-based rules are evaluated in their exact declaration order. If no
  rule matches or if there is no rule, the default action is to accept the
  contents. There is no specific limit to the number of rules which may be
  inserted.

  Several types of actions are supported :
    - accept :
        accepts the response if the condition is true (when used with "if")
        or false (when used with "unless"). The first such rule executed ends
        the rules evaluation.

    - close :
        immediately closes the connection with the server if the condition is
        true (when used with "if"), or false (when used with "unless"). The
        first such rule executed ends the rules evaluation. The main purpose of
        this action is to force a connection to be finished between a client
        and a server after an exchange when the application protocol expects
        some long time outs to elapse first. The goal is to eliminate idle
        connections which take significant resources on servers with certain
        protocols.

    - reject :
        rejects the response if the condition is true (when used with "if")
        or false (when used with "unless"). The first such rule executed ends
        the rules evaluation. Rejected session are immediately closed.

    - set-var(<var-name>) <expr>
        Sets a variable.

    - sc-inc-gpc0(<sc-id>):
        This action increments the GPC0 counter according to the sticky
        counter designated by <sc-id>. If an error occurs, this action fails
        silently and the actions evaluation continues.

    - sc-set-gpt0(<sc-id>) <int> :
        This action sets the GPT0 tag according to the sticky counter designated
        by <sc-id> and the value of <int>. The expected result is a boolean. If
        an error occurs, this action silently fails and the actions evaluation
        continues.

    - "silent-drop" :
        This stops the evaluation of the rules and makes the client-facing
        connection suddenly disappear using a system-dependant way that tries
        to prevent the client from being notified. The effect it then that the
        client still sees an established connection while there's none on
        HAProxy. The purpose is to achieve a comparable effect to "tarpit"
        except that it doesn't use any local resource at all on the machine
        running HAProxy. It can resist much higher loads than "tarpit", and
        slow down stronger attackers. It is important to undestand the impact
        of using this mechanism. All stateful equipments placed between the
        client and HAProxy (firewalls, proxies, load balancers) will also keep
        the established connection for a long time and may suffer from this
        action.  On modern Linux systems running with enough privileges, the
        TCP_REPAIR socket option is used to block the emission of a TCP
        reset. On other systems, the socket's TTL is reduced to 1 so that the
        TCP reset doesn't pass the first router, though it's still delivered to
        local networks. Do not use it unless you fully understand how it works.

  Note that the "if/unless" condition is optional. If no condition is set on
  the action, it is simply performed unconditionally. That can be useful for
  for changing the default action to a reject.

```

```
  It is perfectly possible to match layer 7 contents with "tcp-response
  content" rules, but then it is important to ensure that a full response has
  been buffered, otherwise no contents will match. In order to achieve this,
  the best solution involves detecting the HTTP protocol during the inspection
  period.

  The "set-var" is used to set the content of a variable. The variable is
  declared inline.

    <var-name> The name of the variable starts by an indication about its scope.
               The allowed scopes are:
                 "sess" : the variable is shared with all the session,
                 "txn"  : the variable is shared with all the transaction
                          (request and response)
                 "req"  : the variable is shared only during the request
                          processing
                 "res"  : the variable is shared only during the response
                          processing.
               This prefix is followed by a name. The separator is a '.'.
               The name may only contain characters 'a-z', 'A-Z', '0-9' and '_'.

    <expr>     Is a standard HAProxy expression formed by a sample-fetch
               followed by some converters.

  Example:

        tcp-request content set-var(sess.my_var) src

  See section 7 about ACL usage.

  See also : "tcp-request content", "tcp-response inspect-delay"


tcp-response inspect-delay <timeout>
  Set the maximum allowed time to wait for a response during content inspection
  May be used in sections :   defaults | frontend | listen | backend
                                 no    |    no    |   yes  |   yes
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  See also : "tcp-response content", "tcp-request inspect-delay".


timeout check <timeout>
  Set additional check timeout, but only after a connection has been already
  established.

  May be used in sections:    defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments:
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  If set, haproxy uses min("timeout connect", "inter") as a connect timeout
  for check and "timeout check" as an additional read timeout. The "min" is
  used so that people running with *very* long "timeout connect" (eg. those
  who needed this due to the queue or tarpit) do not slow down their checks.
  (Please also note that there is no valid reason to have such long connect
  timeouts, because "timeout queue" and "timeout tarpit" can always be used to
  avoid that).

  If "timeout check" is not set haproxy uses "inter" for complete check
  timeout (connect + read) exactly like all <1.3.15 version.

  In most cases check request is much simpler and faster to handle than normal
  requests and people may want to kick out laggy servers so this timeout should
  be smaller than "timeout server".

  This parameter is specific to backends, but can be specified once for all in
  "defaults" sections. This is in fact one of the easiest solutions not to
  forget about it.

  See also: "timeout connect", "timeout queue", "timeout server",
            "timeout tarpit".


timeout client <timeout>
timeout clitimeout <timeout> (deprecated)
  Set the maximum inactivity time on the client side.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  The inactivity timeout applies when the client is expected to acknowledge or
  send data. In HTTP mode, this timeout is particularly important to consider
  during the first phase, when the client sends the request, and during the
  response while it is reading data sent by the server. The value is specified
  in milliseconds by default, but can be in any other unit if the number is
  suffixed by the unit, as specified at the top of this document. In TCP mode
  (and to a lesser extent, in HTTP mode), it is highly recommended that the
  client timeout remains equal to the server timeout in order to avoid complex
  situations to debug. It is a good practice to cover one or several TCP packet
  losses by specifying timeouts that are slightly above multiples of 3 seconds
  (eg: 4 or 5 seconds). If some long-lived sessions are mixed with short-lived
  sessions (eg: WebSocket and HTTP), it's worth considering "timeout tunnel",
  which overrides "timeout client" and "timeout server" for tunnels, as well as
  "timeout client-fin" for half-closed connections.

  This parameter is specific to frontends, but can be specified once for all in
  "defaults" sections. This is in fact one of the easiest solutions not to
  forget about it. An unspecified timeout results in an infinite timeout, which
  is not recommended. Such a usage is accepted and works but reports a warning
  during startup because it may results in accumulation of expired sessions in
  the system if the system's timeouts are not configured either.

  This parameter replaces the old, deprecated "clitimeout". It is recommended
  to use it to write new configurations. The form "timeout clitimeout" is
  provided only by backwards compatibility but its use is strongly discouraged.

  See also : "clitimeout", "timeout server", "timeout tunnel".


timeout client-fin <timeout>
  Set the inactivity timeout on the client side for half-closed connections.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  The inactivity timeout applies when the client is expected to acknowledge or
  send data while one direction is already shut down. This timeout is different
  from "timeout client" in that it only applies to connections which are closed
```

```
  in one direction. This is particularly useful to avoid keeping connections in
  FIN_WAIT state for too long when clients do not disconnect cleanly. This
  problem is particularly common long connections such as RDP or WebSocket.
  Note that this timeout can override "timeout tunnel" when a connection shuts
  down in one direction.

  This parameter is specific to frontends, but can be specified once for all in
  "defaults" sections. By default it is not set, so half-closed connections
  will use the other timeouts (timeout.client or timeout.tunnel).

  See also : "timeout client", "timeout server-fin", and "timeout tunnel".


timeout connect <timeout>
timeout contimeout <timeout> (deprecated)
  Set the maximum time to wait for a connection attempt to a server to succeed.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  If the server is located on the same LAN as haproxy, the connection should be
  immediate (less than a few milliseconds). Anyway, it is a good practice to
  cover one or several TCP packet losses by specifying timeouts that are
  slightly above multiples of 3 seconds (eg: 4 or 5 seconds). By default, the
  connect timeout also presets both queue and tarpit timeouts to the same value
  if these have not been specified.

  This parameter is specific to backends, but can be specified once for all in
  "defaults" sections. This is in fact one of the easiest solutions not to
  forget about it. An unspecified timeout results in an infinite timeout, which
  is not recommended. Such a usage is accepted and works but reports a warning
  during startup because it may results in accumulation of failed sessions in
  the system if the system's timeouts are not configured either.

  This parameter replaces the old, deprecated "contimeout". It is recommended
  to use it to write new configurations. The form "timeout contimeout" is
  provided only by backwards compatibility but its use is strongly discouraged.

  See also: "timeout check", "timeout queue", "timeout server", "contimeout",
            "timeout tarpit".


timeout http-keep-alive <timeout>
  Set the maximum allowed time to wait for a new HTTP request to appear
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  By default, the time to wait for a new request in case of keep-alive is set
  by "timeout http-request". However this is not always convenient because some
  people want very short keep-alive timeouts in order to release connections
  faster, and others prefer to have larger ones but still have short timeouts
  once the request has started to present itself.

  The "http-keep-alive" timeout covers these needs. It will define how long to
  wait for a new HTTP request to start coming after a response was sent. Once
  the first byte of request has been seen, the "http-request" timeout is used
  to wait for the complete request to come. Note that empty lines prior to a
  new request do not refresh the timeout and are not counted as a new request.

  There is also another difference between the two timeouts : when a connection
  expires during timeout http-keep-alive, no error is returned, the connection
  just closes. If the connection expires in "http-request" while waiting for a
  connection to complete, a HTTP 408 error is returned.

  In general it is optimal to set this value to a few tens to hundreds of
  milliseconds, to allow users to fetch all objects of a page at once but
  without waiting for further clicks. Also, if set to a very small value (eg:
  1 millisecond) it will probably only accept pipelined requests but not the
  non-pipelined ones. It may be a nice trade-off for very large sites running
  with tens to hundreds of thousands of clients.

  If this parameter is not set, the "http-request" timeout applies, and if both
  are not set, "timeout client" still applies at the lower level. It should be
  set in the frontend to take effect, unless the frontend is in TCP mode, in
  which case the HTTP backend's timeout will be used.

  See also : "timeout http-request", "timeout client".


timeout http-request <timeout>
  Set the maximum allowed time to wait for a complete HTTP request
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  In order to offer DoS protection, it may be required to lower the maximum
  accepted time to receive a complete HTTP request without affecting the client
  timeout. This helps protecting against established connections on which
  nothing is sent. The client timeout cannot offer a good protection against
  this abuse because it is an inactivity timeout, which means that if the
  attacker sends one character every now and then, the timeout will not
  trigger. With the HTTP request timeout, no matter what speed the client
  types, the request will be aborted if it does not complete in time. When the
  timeout expires, an HTTP 408 response is sent to the client to inform it
  about the problem, and the connection is closed. The logs will report
  termination codes "cR". Some recent browsers are having problems with this
  standard, well-documented behaviour, so it might be needed to hide the 408
  code using "option http-ignore-probes" or "errorfile 408 /dev/null". See
  more details in the explanations of the "cR" termination code in section 8.5.

  By default, this timeout only applies to the header part of the request,
  and not to any data. As soon as the empty line is received, this timeout is
  not used anymore. When combined with "option http-buffer-request", this
  timeout also applies to the body of the request..
  It is used again on keep-alive connections to wait for a second
  request if "timeout http-keep-alive" is not set.

  Generally it is enough to set it to a few seconds, as most clients send the
  full request immediately upon connection. Add 3 or more seconds to cover TCP
  retransmits but that's all. Setting it to very low values (eg: 50 ms) will
  generally work on local networks as long as there are no packet losses. This
  will prevent people from sending bare HTTP requests using telnet.

  If this parameter is not set, the client timeout still applies between each
  chunk of the incoming request. It should be set in the frontend to take
  effect, unless the frontend is in TCP mode, in which case the HTTP backend's
  timeout will be used.

  See also : "errorfile", "http-ignore-probes", "timeout http-keep-alive", and
             "timeout client", "option http-buffer-request".
```

```


timeout queue <timeout>
  Set the maximum time to wait in the queue for a connection slot to be free
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  When a server's maxconn is reached, connections are left pending in a queue
  which may be server-specific or global to the backend. In order not to wait
  indefinitely, a timeout is applied to requests pending in the queue. If the
  timeout is reached, it is considered that the request will almost never be
  served, so it is dropped and a 503 error is returned to the client.

  The "timeout queue" statement allows to fix the maximum time for a request to
  be left pending in a queue. If unspecified, the same value as the backend's
  connection timeout ("timeout connect") is used, for backwards compatibility
  with older versions with no "timeout queue" parameter.

  See also : "timeout connect", "contimeout".


timeout server <timeout>
timeout srvtimeout <timeout> (deprecated)
  Set the maximum inactivity time on the server side.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  The inactivity timeout applies when the server is expected to acknowledge or
  send data. In HTTP mode, this timeout is particularly important to consider
  during the first phase of the server's response, when it has to send the
  headers, as it directly represents the server's processing time for the
  request. To find out what value to put there, it's often good to start with
  what would be considered as unacceptable response times, then check the logs
  to observe the response time distribution, and adjust the value accordingly.

  The value is specified in milliseconds by default, but can be in any other
  unit if the number is suffixed by the unit, as specified at the top of this
  document. In TCP mode (and to a lesser extent, in HTTP mode), it is highly
  recommended that the client timeout remains equal to the server timeout in
  order to avoid complex situations to debug. Whatever the expected server
  response times, it is a good practice to cover at least one or several TCP
  packet losses by specifying timeouts that are slightly above multiples of 3
  seconds (eg: 4 or 5 seconds minimum). If some long-lived sessions are mixed
  with short-lived sessions (eg: WebSocket and HTTP), it's worth considering
  "timeout tunnel", which overrides "timeout client" and "timeout server" for
  tunnels.

  This parameter is specific to backends, but can be specified once for all in
  "defaults" sections. This is in fact one of the easiest solutions not to
  forget about it. An unspecified timeout results in an infinite timeout, which
  is not recommended. Such a usage is accepted and works but reports a warning
  during startup because it may results in accumulation of expired sessions in
  the system if the system's timeouts are not configured either.

  This parameter replaces the old, deprecated "srvtimeout". It is recommended
  to use it to write new configurations. The form "timeout srvtimeout" is
  provided only by backwards compatibility but its use is strongly discouraged.

  See also : "srvtimeout", "timeout client" and "timeout tunnel".


timeout server-fin <timeout>
  Set the inactivity timeout on the server side for half-closed connections.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  The inactivity timeout applies when the server is expected to acknowledge or
  send data while one direction is already shut down. This timeout is different
  from "timeout server" in that it only applies to connections which are closed
  in one direction. This is particularly useful to avoid keeping connections in
  FIN_WAIT state for too long when a remote server does not disconnect cleanly.
  This problem is particularly common long connections such as RDP or WebSocket.
  Note that this timeout can override "timeout tunnel" when a connection shuts
  down in one direction. This setting was provided for completeness, but in most
  situations, it should not be needed.

  This parameter is specific to backends, but can be specified once for all in
  "defaults" sections. By default it is not set, so half-closed connections
  will use the other timeouts (timeout.server or timeout.tunnel).

  See also : "timeout client-fin", "timeout server", and "timeout tunnel".


timeout tarpit <timeout>
  Set the duration for which tarpitted connections will be maintained
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   yes
  Arguments :
    <timeout> is the tarpit duration specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  When a connection is tarpitted using "reqtarpit", it is maintained open with
  no activity for a certain amount of time, then closed. "timeout tarpit"
  defines how long it will be maintained open.

  The value is specified in milliseconds by default, but can be in any other
  unit if the number is suffixed by the unit, as specified at the top of this
  document. If unspecified, the same value as the backend's connection timeout
  ("timeout connect") is used, for backwards compatibility with older versions
  with no "timeout tarpit" parameter.

  See also : "timeout connect", "contimeout".


timeout tunnel <timeout>
  Set the maximum inactivity time on the client and server side for tunnels.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments :
    <timeout> is the timeout value specified in milliseconds by default, but
              can be in any other unit if the number is suffixed by the unit,
              as explained at the top of this document.

  The tunnel timeout applies when a bidirectional connection is established
  between a client and a server, and the connection remains inactive in both
  directions. This timeout supersedes both the client and server timeouts once
  the connection becomes a tunnel. In TCP, this timeout is used as soon as no
```

```
  analyser remains attached to either connection (eg: tcp content rules are
  accepted). In HTTP, this timeout is used when a connection is upgraded (eg:
  when switching to the WebSocket protocol, or forwarding a CONNECT request
  to a proxy), or after the first response when no keepalive/close option is
  specified.

  Since this timeout is usually used in conjunction with long-lived connections,
  it usually is a good idea to also set "timeout client-fin" to handle the
  situation where a client suddenly disappears from the net and does not
  acknowledge a close, or sends a shutdown and does not acknowledge pending
  data anymore. This can happen in lossy networks where firewalls are present,
  and is detected by the presence of large amounts of sessions in a FIN_WAIT
  state.

  The value is specified in milliseconds by default, but can be in any other
  unit if the number is suffixed by the unit, as specified at the top of this
  document. Whatever the expected normal idle time, it is a good practice to
  cover at least one or several TCP packet losses by specifying timeouts that
  are slightly above multiples of 3 seconds (eg: 4 or 5 seconds minimum).

  This parameter is specific to backends, but can be specified once for all in
  "defaults" sections. This is in fact one of the easiest solutions not to
  forget about it.

  Example :
        defaults http
            option http-server-close
            timeout connect 5s
            timeout client 30s
            timeout client-fin 30s
            timeout server 30s
            timeout tunnel  1h    # timeout to use with WebSocket and CONNECT

  See also : "timeout client", "timeout client-fin", "timeout server".


transparent (deprecated)
  Enable client-side transparent proxying
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    no    |   yes  |   yes
  Arguments : none

  This keyword was introduced in order to provide layer 7 persistence to layer
  3 load balancers. The idea is to use the OS's ability to redirect an incoming
  connection for a remote address to a local process (here HAProxy), and let
  this process know what address was initially requested. When this option is
  used, sessions without cookies will be forwarded to the original destination
  IP address of the incoming request (which should match that of another
  equipment), while requests with cookies will still be forwarded to the
  appropriate server.

  The "transparent" keyword is deprecated, use "option transparent" instead.

  Note that contrary to a common belief, this option does NOT make HAProxy
  present the client's IP to the server when establishing the connection.

  See also: "option transparent"

unique-id-format <string>
  Generate a unique ID for each request.
  May be used in sections :   defaults | frontend | listen | backend
                                 yes   |    yes   |   yes  |   no
  Arguments :
    <string>   is a log-format string.
  This keyword creates a ID for each request using the custom log format. A
  unique ID is useful to trace a request passing through many components of
  a complex infrastructure. The newly created ID may also be logged using the
  %ID tag the log-format string.

  The format should be composed from elements that are guaranteed to be
  unique when combined together. For instance, if multiple haproxy instances
  are involved, it might be important to include the node name. It is often
  needed to log the incoming connection's source and destination addresses
  and ports. Note that since multiple requests may be performed over the same
  connection, including a request counter may help differentiate them.
  Similarly, a timestamp may protect against a rollover of the counter.
  Logging the process ID will avoid collisions after a service restart.

  It is recommended to use hexadecimal notation for many fields since it
  makes them more compact and saves space in logs.

  Example:

        unique-id-format %{+X}o\ %ci:%cp_%fi:%fp_%Ts_%rt:%pid

        will generate:

               7F000001:8296_7F00001E:1F90_4F7B0A69_0003:790A

  See also: "unique-id-header"

unique-id-header <name>
  Add a unique ID header in the HTTP request.
  May be used in sections :   defaults | frontend | listen | backend
                                  yes   |    yes   |   yes  |   no
  Arguments :
    <name>   is the name of the header.

  Add a unique-id header in the HTTP request sent to the server, using the
  unique-id-format. It can't work if the unique-id-format doesn't exist.

  Example:

        unique-id-format %{+X}o\ %ci:%cp_%fi:%fp_%Ts_%rt:%pid
        unique-id-header X-Unique-ID

        will generate:

           X-Unique-ID: 7F000001:8296_7F00001E:1F90_4F7B0A69_0003:790A

    See also: "unique-id-format"

use_backend <backend> [{if | unless} <condition>]
  Switch to a specific backend if/unless an ACL-based condition is matched.
  May be used in sections :   defaults | frontend | listen | backend
                                  no    |    yes   |   yes  |   no
  Arguments :
    <backend>   is the name of a valid backend or "listen" section, or a
                "log-format" string resolving to a backend name.

    <condition> is a condition composed of ACLs, as described in section 7. If
                it is omitted, the rule is unconditionally applied.

  When doing content-switching, connections arrive on a frontend and are then
  dispatched to various backends depending on a number of conditions. The
  relation between the conditions and the backends is described with the
  "use_backend" keyword. While it is normally used with HTTP processing, it can
  also be used in pure TCP, either without content using stateless ACLs (eg:
  source address validation) or combined with a "tcp-request" rule to wait for
```

```
  some payload.

  There may be as many "use_backend" rules as desired. All of these rules are
  evaluated in their declaration order, and the first one which matches will
  assign the backend.

  In the first form, the backend will be used if the condition is met. In the
  second form, the backend will be used if the condition is not met. If no
  condition is valid, the backend defined with "default_backend" will be used.
  If no default backend is defined, either the servers in the same section are
  used (in case of a "listen" section) or, in case of a frontend, no server is
  used and a 503 service unavailable response is returned.

  Note that it is possible to switch from a TCP frontend to an HTTP backend. In
  this case, either the frontend has already checked that the protocol is HTTP,
  and backend processing will immediately follow, or the backend will wait for
  a complete HTTP request to get in. This feature is useful when a frontend
  must decode several protocols on a unique port, one of them being HTTP.

  When <backend> is a simple name, it is resolved at configuration time, and an
  error is reported if the specified backend does not exist. If <backend> is
  a log-format string instead, no check may be done at configuration time, so
  the backend name is resolved dynamically at run time. If the resulting
  backend name does not correspond to any valid backend, no other rule is
  evaluated, and the default_backend directive is applied instead. Note that
  when using dynamic backend names, it is highly recommended to use a prefix
  that no other backend uses in order to ensure that an unauthorized backend
  cannot be forced from the request.

  It is worth mentioning that "use_backend" rules with an explicit name are
  used to detect the association between frontends and backends to compute the
  backend's "fullconn" setting. This cannot be done for dynamic names.

  See also: "default_backend", "tcp-request", "fullconn", "log-format", and
            section 7 about ACLs.


use-server <server> if <condition>
use-server <server> unless <condition>
  Only use a specific server if/unless an ACL-based condition is matched.
  May be used in sections :   defaults | frontend | listen | backend
                                  no    |    no    |   yes  |   yes
  Arguments :
    <server>    is the name of a valid server in the same backend section.

    <condition> is a condition composed of ACLs, as described in section 7.

  By default, connections which arrive to a backend are load-balanced across
  the available servers according to the configured algorithm, unless a
  persistence mechanism such as a cookie is used and found in the request.

  Sometimes it is desirable to forward a particular request to a specific
  server without having to declare a dedicated backend for this server. This
  can be achieved using the "use-server" rules. These rules are evaluated after
  the "redirect" rules and before evaluating cookies, and they have precedence
  on them. There may be as many "use-server" rules as desired. All of these
  rules are evaluated in their declaration order, and the first one which
  matches will assign the server.

  If a rule designates a server which is down, and "option persist" is not used
  and no force-persist rule was validated, it is ignored and evaluation goes on
  with the next rules until one matches.

  In the first form, the server will be used if the condition is met. In the
  second form, the server will be used if the condition is not met. If no
```

condition is valid, the processing continues and the server will be assigned according to other persistence mechanisms.

Note that even if a rule is matched, cookie processing is still performed but does not assign the server. This allows prefixed cookies to have their prefix stripped.

The "use-server" statement works both in HTTP and TCP mode. This makes it suitable for use with content-based inspection. For instance, a server could be selected in a farm according to the TLS SNI field. And if these servers have their weight set to zero, they will not be used for other traffic.

Example :

```
   # intercept incoming TLS requests based on the SNI field
   use-server www if { req_ssl_sni -i www.example.com }
   server     www 192.168.0.1:443 weight 0
   use-server mail if { req_ssl_sni -i mail.example.com }
   server     mail 192.168.0.1:587 weight 0
   use-server imap if { req_ssl_sni -i imap.example.com }
   server     mail 192.168.0.1:993 weight 0
   # all the rest is forwarded to this server
   server  default 192.168.0.2:443 check
```

See also: "use_backend", section 5 about server and section 7 about ACLs.

5. Bind and Server options

The "bind", "server" and "default-server" keywords support a number of settings depending on some build options and on the system HAProxy was built on. These settings generally each consist in one word sometimes followed by a value, written on the same line as the "bind" or "server" line. All these options are described in this section.

5.1. Bind options

The "bind" keyword supports a certain number of settings which are all passed as arguments on the same line. The order in which those arguments appear makes no importance, provided that they appear after the bind address. All of these parameters are optional. Some of them consist in a single words (booleans), while other ones expect a value after them. In this case, the value must be provided immediately after the setting name.

The currently supported settings are the following ones.

accept-proxy
  Enforces the use of the PROXY protocol over any connection accepted by any of the sockets declared on the same line. Versions 1 and 2 of the PROXY protocol are supported and correctly detected. The PROXY protocol dictates the layer 3/4 addresses of the incoming connection to be used everywhere an address is used, with the only exception of "tcp-request connection" rules which will only see the real connection address. Logs will reflect the addresses indicated in the protocol, unless it is violated, in which case the real address will still be used. This keyword combined with support from external components can be used as an efficient and reliable alternative to the X-Forwarded-For mechanism which is not always reliable and not even always usable. See also "tcp-request connection expect-proxy" for a finer-grained setting of which client is allowed to use the protocol.

alpn <protocols>
  This enables the TLS ALPN extension and advertises the specified protocol list as supported on top of ALPN. The protocol list consists in a comma-delimited list of protocol names, for instance: "http/1.1,http/1.0" (without quotes). This requires that the SSL library is build with support for TLS extensions enabled (check with haproxy -vv). The ALPN extension replaces the initial NPN extension.

backlog <backlog>
  Sets the socket's backlog to this value. If unspecified, the frontend's backlog is used instead, which generally defaults to the maxconn value.

ecdhe <named curve>
  This setting is only available when support for OpenSSL was built in. It sets the named curve (RFC 4492) used to generate ECDH ephemeral keys. By default, used named curve is prime256v1.

ca-file <cafile>
  This setting is only available when support for OpenSSL was built in. It designates a PEM file from which to load CA certificates used to verify client's certificate.

ca-ignore-err [all|<errorID>,...]
  This setting is only available when support for OpenSSL was built in. Sets a comma separated list of errorIDs to ignore during verify at depth > 0. If set to 'all', all errors are ignored. SSL handshake is not aborted if an error is ignored.

ca-sign-file <cafile>
  This setting is only available when support for OpenSSL was built in. It designates a PEM file containing both the CA certificate and the CA private key used to create and sign server's certificates. This is a mandatory setting when the dynamic generation of certificates is enabled. See 'generate-certificates' for details.

ca-sign-passphrase <passphrase>
  This setting is only available when support for OpenSSL was built in. It is the CA private key passphrase. This setting is optional and used only when the dynamic generation of certificates is enabled. See 'generate-certificates' for details.

ciphers <ciphers>
  This setting is only available when support for OpenSSL was built in. It sets the string describing the list of cipher algorithms ("cipher suite") that are negotiated during the SSL/TLS handshake. The format of the string is defined in "man 1 ciphers" from OpenSSL man pages, and can be for instance a string such as "AES:ALL:!aNULL:!eNULL:+RC4:@STRENGTH" (without quotes).

crl-file <crlfile>
  This setting is only available when support for OpenSSL was built in. It designates a PEM file from which to load certificate revocation list used to verify client's certificate.

crt <cert>
  This setting is only available when support for OpenSSL was built in. It designates a PEM file containing both the required certificates and any associated private keys. This file can be built by concatenating multiple PEM files into one (e.g. cat cert.pem key.pem > combined.pem). If your CA requires an intermediate certificate, this can also be concatenated into this file.

  If the OpenSSL used supports Diffie-Hellman, parameters present in this file are loaded.

  If a directory name is used instead of a PEM file, then all files found in that directory will be loaded in alphabetic order unless their name ends with '.issuer', '.ocsp' or '.sctl' (reserved extensions). This directive may be specified multiple times in order to load certificates from multiple files or directories. The certificates will be presented to clients who provide a valid TLS Server Name Indication field matching one of their CN or alt subjects. Wildcards are supported, where a wildcard character '*' is used instead of the first hostname component (eg: *.example.org matches www.example.org but not www.sub.example.org).

If no SNI is provided by the client or if the SSL library does not support TLS extensions, or if the client provides an SNI hostname which does not match any certificate, then the first loaded certificate will be presented. This means that when loading certificates from a directory, it is highly recommended to load the default one first as a file or to ensure that it will always be the first one in the directory.

Note that the same cert may be loaded multiple times without side effects.

Some CAs (such as Godaddy) offer a drop down list of server types that do not include HAProxy when obtaining a certificate. If this happens be sure to choose a webserver that the CA believes requires an intermediate CA (for Godaddy, selection Apache Tomcat will get the correct bundle, but many others, e.g. nginx, result in a wrong bundle that will not work for some clients).

For each PEM file, haproxy checks for the presence of file at the same path suffixed by ".ocsp". If such file is found, support for the TLS Certificate Status Request extension (also known as "OCSP stapling") is automatically enabled. The content of this file is optional. If not empty, it must contain a valid OCSP Response in DER format. In order to be valid an OCSP Response must comply with the following rules: it has to indicate a good status, it has to be a single response for the certificate of the PEM file, and it has to be valid at the moment of addition. If these rules are not respected the OCSP Response is ignored and a warning is emitted. In order to identify which certificate an OCSP Response applies to, the issuer's certificate is necessary. If the issuer's certificate is not found in the PEM file, it will be loaded from a file at the same path as the PEM file suffixed by ".issuer" if it exists otherwise it will fail with an error.

For each PEM file, haproxy also checks for the presence of file at the same path suffixed by ".sctl". If such file is found, support for Certificate Transparency (RFC6962) TLS extension is enabled. The file must contain a valid Signed Certificate Timestamp List, as described in RFC. File is parsed to check basic syntax, but no signatures are verified.

crt-ignore-err <errors>

This setting is only available when support for OpenSSL was built in. Sets a comma separated list of errorIDs to ignore during verify at depth == 0. If set to 'all', all errors are ignored. SSL handshake is not aborted if an error is ignored.

crt-list <file>

This setting is only available when support for OpenSSL was built in. It designates a list of PEM file with an optional list of SNI filter per certificate, with the following format for each line :

```
      <crtfile> [[!]<snifilter> ...]
```

Wildcards are supported in the SNI filter. Negative filter are also supported, only useful in combination with a wildcard filter to exclude a particular SNI. The certificates will be presented to clients who provide a valid TLS Server Name Indication field matching one of the SNI filters. If no SNI filter is specified, the CN and alt subjects are used. This directive may be specified multiple times. See the "crt" option for more information. The default certificate is still needed to meet OpenSSL expectations. If it is not used, the 'strict-sni' option may be used.

defer-accept

Is an optional keyword which is supported only on certain Linux kernels. It states that a connection will only be accepted once some data arrive on it, or at worst after the first retransmit. This should be used only on protocols for which the client talks first (eg: HTTP). It can slightly improve performance by ensuring that most of the request is already available when the connection is accepted. On the other hand, it will not be able to detect connections which don't talk. It is important to note that this option is broken in all kernels up to 2.6.31, as the connection is never accepted until the client talks. This can cause issues with front firewalls which would see an established connection while the proxy will only see it in SYN_RECV. This option is only supported on TCPv4/TCPv6 sockets and ignored by other ones.

force-sslv3

This option enforces use of SSLv3 only on SSL connections instantiated from this listener. SSLv3 is generally less expensive than the TLS counterparts for high connection rates. This option is also available on global statement "ssl-default-bind-options". See also "no-tlsv*" and "no-sslv3".

force-tlsv10

This option enforces use of TLSv1.0 only on SSL connections instantiated from this listener. This option is also available on global statement "ssl-default-bind-options". See also "no-tlsv*" and "no-sslv3".

force-tlsv11

This option enforces use of TLSv1.1 only on SSL connections instantiated from this listener. This option is also available on global statement "ssl-default-bind-options". See also "no-tlsv*", and "no-sslv3".

force-tlsv12

This option enforces use of TLSv1.2 only on SSL connections instantiated from this listener. This option is also available on global statement "ssl-default-bind-options". See also "no-tlsv*", and "no-sslv3".

generate-certificates

This setting is only available when support for OpenSSL was built in. It enables the dynamic SSL certificates generation. A CA certificate and its private key are necessary (see 'ca-sign-file'). When HAProxy is configured as a transparent forward proxy, SSL requests generate errors because of a common name mismatch on the certificate presented to the client. With this option enabled, HAProxy will try to forge a certificate using the SNI hostname indicated by the client. This is done only if no certificate matches the SNI hostname (see 'crt-list'). If an error occurs, the default certificate is used, else the 'strict-sni' option is set.
It can also be used when HAProxy is configured as a reverse proxy to ease the deployment of an architecture with many backends.

Creating a SSL certificate is an expensive operation, so a LRU cache is used to store forged certificates (see 'tune.ssl.ssl-ctx-cache-size'). It increases the HAProxy's memroy footprint to reduce latency when the same certificate is used many times.

gid <gid>

Sets the group of the UNIX sockets to the designated system gid. It can also be set by default in the global section's "unix-bind" statement. Note that some platforms simply ignore this. This setting is equivalent to the "group" setting except that the group ID is used instead of its name. This setting is ignored by non UNIX sockets.

group <group>

Sets the group of the UNIX sockets to the designated system group. It can also be set by default in the global section's "unix-bind" statement. Note that some platforms simply ignore this. This setting is equivalent to the "gid" setting except that the group name is used instead of its gid. This setting is ignored by non UNIX sockets.

id <id>

Fixes the socket ID. By default, socket IDs are automatically assigned, but sometimes it is more convenient to fix them to ease monitoring. This value must be strictly positive and unique within the listener/frontend. This option can only be used when defining only a single socket.

interface <interface>

Restricts the socket to a specific interface. When specified, only packets received from that particular interface are processed by the socket. This is currently only supported on Linux. The interface must be a primary system interface, not an aliased interface. It is also possible to bind multiple frontends to the same address if they are bound to different interfaces. Note that binding to a network interface requires root privileges. This parameter is only compatible with TCPv4/TCPv6 sockets.

level <level>

This setting is used with the stats sockets only to restrict the nature of the commands that can be issued on the socket. It is ignored by other sockets. <level> can be one of :

- "user" is the least privileged level ; only non-sensitive stats can be read, and no change is allowed. It would make sense on systems where it is not easy to restrict access to the socket.
- "operator" is the default level and fits most common uses. All data can be read, and only non-sensitive changes are permitted (eg: clear max counters).
- "admin" should be used with care, as everything is permitted (eg: clear all counters).

maxconn <maxconn>

Limits the sockets to this number of concurrent connections. Extraneous connections will remain in the system's backlog until a connection is released. If unspecified, the limit will be the same as the frontend's maxconn. Note that in case of port ranges or multiple addresses, the same value will be applied to each socket. This setting enables different limitations on expensive sockets, for instance SSL entries which may easily eat all memory.

mode <mode>

Sets the octal mode used to define access permissions on the UNIX socket. It can also be set by default in the global section's "unix-bind" statement. Note that some platforms simply ignore this. This setting is ignored by non UNIX sockets.

mss <maxseg>

Sets the TCP Maximum Segment Size (MSS) value to be advertised on incoming connections. This can be used to force a lower MSS for certain specific ports, for instance for connections passing through a VPN. Note that this relies on a kernel feature which is theoretically supported under Linux but was buggy in all versions prior to 2.6.28. It may or may not work on other operating systems. It may also not change the advertised value but change the effective size of outgoing segments. The commonly advertised value for TCPv4 over Ethernet networks is 1460 = 1500(MTU) - 40(IP+TCP). If this value is positive, it will be used as the advertised MSS. If it is negative, it will indicate by how much to reduce the incoming connection's advertised MSS for outgoing segments. This parameter is only compatible with TCP v4/v6 sockets.

name <name>

Sets an optional name for these sockets, which will be reported on the stats page.

namespace <name>

On Linux, it is possible to specify which network namespace a socket will belong to. This directive makes it possible to explicitly bind a listener to a namespace different from the default one. Please refer to your operating system's documentation to find more details about network namespaces.

nice <nice>

Sets the 'niceness' of connections initiated from the socket. Value must be in the range -1024..1024 inclusive, and defaults to zero. Positive values means that such connections are more friendly to others and easily offer their place in the scheduler. On the opposite, negative values mean that connections want to run with a higher priority than others. The difference only happens under high loads when the system is close to saturation. Negative values are appropriate for low-latency or administration services, and high values are generally recommended for CPU intensive tasks such as SSL processing or bulk transfers which are less sensible to latency. For example, it may make sense to use a positive value for an SMTP socket and a negative one for an RDP socket.

no-sslv3

This setting is only available when support for OpenSSL was built in. It disables support for SSLv3 on any sockets instantiated from the listener when SSL is supported. Note that SSLv2 is forced disabled in the code and cannot be enabled using any configuration option. This option is also available on global statement "ssl-default-bind-options". See also "force-tls*", and "force-sslv3".

no-tls-tickets

This setting is only available when support for OpenSSL was built in. It disables the stateless session resumption (RFC 5077 TLS Ticket extension) and force to use stateful session resumption. Stateless session resumption is more expensive in CPU usage. This option is also available on global statement "ssl-default-bind-options".

no-tlsv10

This setting is only available when support for OpenSSL was built in. It disables support for TLSv1.0 on any sockets instantiated from the listener when SSL is supported. Note that SSLv2 is forced disabled in the code and cannot be enabled using any configuration option. This option is also available on global statement "ssl-default-bind-options". See also "force-tlsv*", and "force-sslv3".

no-tlsv11

This setting is only available when support for OpenSSL was built in. It disables support for TLSv1.1 on any sockets instantiated from the listener when SSL is supported. Note that SSLv2 is forced disabled in the code and cannot be enabled using any configuration option. This option is also available on global statement "ssl-default-bind-options". See also "force-tlsv*", and "force-sslv3".

no-tlsv12

This setting is only available when support for OpenSSL was built in. It disables support for TLSv1.2 on any sockets instantiated from the listener when SSL is supported. Note that SSLv2 is forced disabled in the code and cannot be enabled using any configuration option. This option is also available on global statement "ssl-default-bind-options". See also "force-tlsv*", and "force-sslv3".

npn <protocols>

This enables the NPN TLS extension and advertises the specified protocol list as supported on top of NPN. The protocol list consists in a comma-delimited list of protocol names, for instance: "http/1.1,http/1.0" (without quotes). This requires that the SSL library is build with support for TLS extensions enabled (check with haproxy -vv). Note that the NPN extension has been replaced with the ALPN extension (see the "alpn" keyword).

process [all | odd | even | <number 1-64>[-<number 1-64>]]

This restricts the list of processes on which this listener is allowed to run. It does not enforce any process but eliminates those which do not match. If the frontend uses a "bind-process" setting, the intersection between the two is applied. If in the end the listener is not allowed to run on any remaining process, a warning is emitted, and the listener will either run on the first process of the listener if a single process was specified, or on all of its processes if multiple processes were specified. For the unlikely case where several ranges are needed, this directive may be repeated. The main purpose of this directive is to be used with the stats sockets and have one different socket per process. The second purpose is to have multiple bind lines sharing the same IP:port but not the same process in a listener, so that the system can distribute the incoming connections into multiple queues and allow a smoother inter-process load balancing. Currently Linux 3.9 and above is known for supporting this. See also "bind-process" and "nbproc".

ssl

This setting is only available when support for OpenSSL was built in. It enables SSL deciphering on connections instantiated from this listener. A certificate is necessary (see "crt" above). All contents in the buffers will appear in clear text, so that ACLs and HTTP processing will only have access to deciphered contents.

strict-sni

This setting is only available when support for OpenSSL was built in. The SSL/TLS negotiation is allow only if the client provided an SNI which match a certificate. The default certificate is not used. See the "crt" option for more information.

tcp-ut <delay>

Sets the TCP User Timeout for all incoming connections instanciated from this listening socket. This option is available on Linux since version 2.6.37. It allows haproxy to configure a timeout for sockets which contain data not receiving an acknoledgement for the configured delay. This is especially useful on long-lived connections experiencing long idle periods such as remote terminals or database connection pools, where the client and server timeouts must remain high to allow a long period of idle, but where it is important to detect that the client has disappeared in order to release all resources associated with its connection (and the server's session). The argument is a delay expressed in milliseconds by default. This only works for regular TCP connections, and is ignored for other protocols.

tfo

Is an optional keyword which is supported only on Linux kernels >= 3.7. It enables TCP Fast Open on the listening socket, which means that clients which support this feature will be able to send a request and receive a response during the 3-way handshake starting from second connection, thus saving one round-trip after the first connection. This only makes sense with protocols that use high connection rates and where each round trip matters. This can possibly cause issues with many firewalls which do not accept data on SYN packets, so this option should only be enabled once well tested. This option is only supported on TCPv4/TCPv6 sockets and ignored by other ones. You may need to build HAProxy with USE_TFO=1 if your libc doesn't define TCP_FASTOPEN.

tls-ticket-keys <keyfile>

Sets the TLS ticket keys file to load the keys from. The keys need to be 48 bytes long, encoded with base64 (ex. openssl rand -base64 48). Number of keys is specified by the TLS_TICKETS_NO build option (default 3) and at least as many keys need to be present in the file. Last TLS_TICKETS_NO keys will be used for decryption and the penultimate one for encryption. This enables easy key rotation by just appending new key to the file and reloading the process. Keys must be periodically rotated (ex. every 12h) or Perfect Forward Secrecy is compromised. It is also a good idea to keep the keys off any permanent storage such as hard drives (hint: use tmpfs and don't swap those files). Lifetime hint can be changed using tune.ssl.timeout.

transparent

Is an optional keyword which is supported only on certain Linux kernels. It

indicates that the addresses will be bound even if they do not belong to the local machine, and that packets targeting any of these addresses will be intercepted just as if the addresses were locally configured. This normally requires that IP forwarding is enabled. Caution! do not use this with the default address '*', as it would redirect any traffic for the specified port. This keyword is available only when HAProxy is built with USE_LINUX_TPROXY=1. This parameter is only compatible with TCPv4 and TCPv6 sockets, depending on kernel version. Some distribution kernels include backports of the feature, so check for support with your vendor.

v4v6

Is an optional keyword which is supported only on most recent systems including Linux kernels >= 2.4.21. It is used to bind a socket to both IPv4 and IPv6 when it uses the default address. Doing so is sometimes necessary on systems which bind to IPv6 only by default. It has no effect on non-IPv6 sockets, and is overridden by the "v6only" option.

v6only

Is an optional keyword which is supported only on most recent systems including Linux kernels >= 2.4.21. It is used to bind a socket to IPv6 only when it uses the default address. Doing so is sometimes preferred to doing it system-wide as it is per-listener. It has no effect on non-IPv6 sockets and has precedence over the "v4v6" option.

uid <uid>

Sets the owner of the UNIX sockets to the designated system uid. It can also be set by default in the global section's "unix-bind" statement. Note that some platforms simply ignore this. This setting is equivalent to the "user" setting except that the user numeric ID is used instead of its name. This setting is ignored by non UNIX sockets.

user <user>

Sets the owner of the UNIX sockets to the designated system user. It can also be set by default in the global section's "unix-bind" statement. Note that some platforms simply ignore this. This setting is equivalent to the "uid" setting except that the user name is used instead of its uid. This setting is ignored by non UNIX sockets.

verify [none|optional|required]

This setting is only available when support for OpenSSL was built in. If set to 'none', client certificate is not requested. This is the default. In other cases, a client certificate is requested. If the client does not provide a certificate after the request and if 'verify' is set to 'required', then the handshake is aborted, while it would have succeeded if set to 'optional'. The certificate provided by the client is always verified using CAs from 'ca-file' and optional CRLs from 'crl-file'. On verify failure the handshake is aborted, regardless of the 'verify' option, unless the error code exactly matches one of those listed with 'ca-ignore-err' or 'crt-ignore-err'.

5.2. Server and default-server options

The "server" and "default-server" keywords support a certain number of settings which are all passed as arguments on the server line. The order in which those arguments appear does not count, and they are all optional. Some of those settings are single words (booleans) while others expect one or several values after them. In this case, the values must immediately follow the setting name. Except default-server, all those settings must be specified after the server's address if they are used:

```
  server <name> <address>[:port] [settings ...]
  default-server [settings ...]
```

The currently supported settings are the following ones.

addr <ipv4|ipv6>

Using the "addr" parameter, it becomes possible to use a different IP address to send health-checks. On some servers, it may be desirable to dedicate an IP address to specific component able to perform complex tests which are more suitable to health-checks than the application. This parameter is ignored if the "check" parameter is not set. See also the "port" parameter.

Supported in default-server: No

agent-check

Enable an auxiliary agent check which is run independently of a regular health check. An agent health check is performed by making a TCP connection to the port set by the "agent-port" parameter and reading an ASCII string. The string is made of a series of words delimited by spaces, tabs or commas in any order, optionally terminated by '\r' and/or '\n', each consisting of :

- An ASCII representation of a positive integer percentage, e.g. "75%". Values in this format will set the weight proportional to the initial weight of a server as configured when haproxy starts. Note that a zero weight is reported on the stats page as "DRAIN" since it has the same effect on the server (it's removed from the LB farm).

- The word "ready". This will turn the server's administrative state to the READY mode, thus cancelling any DRAIN or MAINT state

- The word "drain". This will turn the server's administrative state to the DRAIN mode, thus it will not accept any new connections other than those that are accepted via persistence.

- The word "maint". This will turn the server's administrative state to the MAINT mode, thus it will not accept any new connections at all, and health checks will be stopped.

- The words "down", "failed", or "stopped", optionally followed by a description string after a sharp ('#'). All of these mark the server's operating state as DOWN, but since the word itself is reported on the stats page, the difference allows an administrator to know if the situation was expected or not : the service may intentionally be stopped, may appear up but fail some validity tests, or may be seen as down (eg: missing process, or port not responding).

- The word "up" sets back the server's operating state as UP if health checks also report that the service is accessible.

Parameters which are not advertised by the agent are not changed. For example, an agent might be designed to monitor CPU usage and only report a relative weight and never interact with the operating status. Similarly, an agent could be designed as an end-user interface with 3 radio buttons allowing an administrator to change only the administrative state. However, it is important to consider that only the agent may revert its own actions, so if a server is set to DRAIN mode or to DOWN state using the agent, the agent must implement the other equivalent actions to bring the service into operations again.

Failure to connect to the agent is not considered an error as connectivity is tested by the regular health check which is enabled by the "check" parameter. Warning though, it is not a good idea to stop an agent after it reports "down", since only an agent reporting "up" will be able to turn the server up again. Note that the CLI on the Unix stats socket is also able to force an agent's result in order to workaround a bogus agent if needed.

Requires the "agent-port" parameter to be set. See also the "agent-inter" parameter.

Supported in default-server: No

agent-inter <delay>

The "agent-inter" parameter sets the interval between two agent checks to <delay> milliseconds. If left unspecified, the delay defaults to 2000 ms.

Just as with every other time-based parameter, it may be entered in any other explicit unit among { us, ms, s, m, h, d }. The "agent-inter" parameter also serves as a timeout for agent checks "timeout check" is not set. In order to reduce "resonance" effects when multiple servers are hosted on the same hardware, the agent and health checks of all servers are started with a small time offset between them. It is also possible to add some random noise in the agent and health checks interval using the global "spread-checks" keyword. This makes sense for instance when a lot of backends use the same servers.

See also the "agent-check" and "agent-port" parameters.

Supported in default-server: Yes

agent-port <port>

The "agent-port" parameter sets the TCP port used for agent checks.

See also the "agent-check" and "agent-inter" parameters.

Supported in default-server: Yes

backup

When "backup" is present on a server line, the server is only used in load balancing when all other non-backup servers are unavailable. Requests coming with a persistence cookie referencing the server will always be served though. By default, only the first operational backup server is used, unless the "allbackups" option is set in the backend. See also the "allbackups" option.

Supported in default-server: No

ca-file <cafile>

This setting is only available when support for OpenSSL was built in. It designates a PEM file from which to load CA certificates used to verify server's certificate.

Supported in default-server: No

check

This option enables health checks on the server. By default, a server is always considered available. If "check" is set, the server is available when accepting periodic TCP connections, to ensure that it is really able to serve requests. The default address and port to send the tests to are those of the server, and the default source is the same as the one defined in the backend. It is possible to change the address using the "addr" parameter, the port using the "port" parameter, the source address using the "source" address, and the interval and timers using the "inter", "rise" and "fall" parameters. The request method is define in the backend using the "httpchk", "smtpchk", "mysql-check", "pgsql-check" and "ssl-hello-chk" options. Please refer to those options and parameters for more information.

Supported in default-server: No

check-send-proxy

This option forces emission of a PROXY protocol line with outgoing health checks, regardless of whether the server uses send-proxy or not for the normal traffic. By default, the PROXY protocol is enabled for health checks if it is already enabled for normal traffic and if no "port" nor "addr" directive is present. However, if such a directive is present, the "check-send-proxy" option needs to be used to force the use of the

```
  protocol. See also the "send-proxy" option for more information.

  Supported in default-server: No

check-ssl
  This option forces encryption of all health checks over SSL, regardless of
  whether the server uses SSL or not for the normal traffic. This is generally
  used when an explicit "port" or "addr" directive is specified and SSL health
  checks are not inherited. It is important to understand that this option
  inserts an SSL transport layer below the checks, so that a simple TCP connect
  check becomes an SSL connect, which replaces the old ssl-hello-chk. The most
  common use is to send HTTPS checks by combining "httpchk" with SSL checks.
  All SSL settings are common to health checks and traffic (eg: ciphers).
  See the "ssl" option for more information.

  Supported in default-server: No

ciphers <ciphers>
  This option sets the string describing the list of cipher algorithms that is
  is negotiated during the SSL/TLS handshake with the server. The format of the
  string is defined in "man 1 ciphers". When SSL is used to communicate with
  servers on the local network, it is common to see a weaker set of algorithms
  than what is used over the internet. Doing so reduces CPU usage on both the
  server and haproxy while still keeping it compatible with deployed software.
  Some algorithms such as RC4-SHA1 are reasonably cheap. If no security at all
  is needed and just connectivity, using DES can be appropriate.

  Supported in default-server: No

cookie <value>
  The "cookie" parameter sets the cookie value assigned to the server to
  <value>. This value will be checked in incoming requests, and the first
  operational server possessing the same value will be selected. In return, in
  cookie insertion or rewrite modes, this value will be assigned to the cookie
  sent to the client. There is nothing wrong in having several servers sharing
  the same cookie value, and it is in fact somewhat common between normal and
  backup servers. See also the "cookie" keyword in backend section.

  Supported in default-server: No

crl-file <crlfile>
  This setting is only available when support for OpenSSL was built in. It
  designates a PEM file from which to load certificate revocation list used
  to verify server's certificate.

  Supported in default-server: No

crt <cert>
  This setting is only available when support for OpenSSL was built in.
  It designates a PEM file from which to load both a certificate and the
  associated private key. This file can be built by concatenating both PEM
  files into one. This certificate will be sent if the server send a client
  certificate request.

  Supported in default-server: No

disabled
  The "disabled" keyword starts the server in the "disabled" state. That means
  that it is marked down in maintenance mode, and no connection other than the
  ones allowed by persist mode will reach it. It is very well suited to setup
  new servers, because normal traffic will never reach them, while it is still
  possible to test the service by making use of the force-persist mechanism.

  Supported in default-server: No

```

```
error-limit <count>
  If health observing is enabled, the "error-limit" parameter specifies the
  number of consecutive errors that triggers event selected by the "on-error"
  option. By default it is set to 10 consecutive errors.

  Supported in default-server: Yes

  See also the "check", "error-limit" and "on-error".

fall <count>
  The "fall" parameter states that a server will be considered as dead after
  <count> consecutive unsuccessful health checks. This value defaults to 3 if
  unspecified. See also the "check", "inter" and "rise" parameters.

  Supported in default-server: Yes

force-sslv3
  This option enforces use of SSLv3 only when SSL is used to communicate with
  the server. SSLv3 is generally less expensive than the TLS counterparts for
  high connection rates. This option is also available on global statement
  "ssl-default-server-options". See also "no-tlsv*", "no-sslv3".

  Supported in default-server: No

force-tlsv10
  This option enforces use of TLSv1.0 only when SSL is used to communicate with
  the server. This option is also available on global statement
  "ssl-default-server-options". See also "no-tlsv*", "no-sslv3".

  Supported in default-server: No

force-tlsv11
  This option enforces use of TLSv1.1 only when SSL is used to communicate with
  the server. This option is also available on global statement
  "ssl-default-server-options". See also "no-tlsv*", "no-sslv3".

  Supported in default-server: No

force-tlsv12
  This option enforces use of TLSv1.2 only when SSL is used to communicate with
  the server. This option is also available on global statement
  "ssl-default-server-options". See also "no-tlsv*", "no-sslv3".

  Supported in default-server: No

id <value>
  Set a persistent ID for the server. This ID must be positive and unique for
  the proxy. An unused ID will automatically be assigned if unset. The first
  assigned value will be 1. This ID is currently only returned in statistics.

  Supported in default-server: No

inter <delay>
fastinter <delay>
downinter <delay>
  The "inter" parameter sets the interval between two consecutive health checks
  to <delay> milliseconds. If left unspecified, the delay defaults to 2000 ms.
  It is also possible to use "fastinter" and "downinter" to optimize delays
  between checks depending on the server state :

             Server state                 |         Interval used
    --------------------------------------+----------------------------------
     UP 100% (non-transitional)           | "inter"
    --------------------------------------+----------------------------------
     Transitionally UP (going down "fall"), | "fastinter" if set,
Transitionally DOWN (going up "rise"), | "inter" otherwise.
  or yet unchecked.                      |
 ----------------------------------------+---------------------------------
  DOWN 100% (non-transitional)           | "downinter" if set,
                                         | "inter" otherwise.
 ----------------------------------------+---------------------------------
```

Just as with every other time-based parameter, they can be entered in any other explicit unit among { us, ms, s, m, h, d }. The "inter" parameter also serves as a timeout for health checks sent to servers if "timeout check" is not set. In order to reduce "resonance" effects when multiple servers are hosted on the same hardware, the agent and health checks of all servers are started with a small time offset between them. It is also possible to add some random noise in the agent and health checks interval using the global "spread-checks" keyword. This makes sense for instance when a lot of backends use the same servers.

Supported in default-server: Yes

maxconn <maxconn>

The "maxconn" parameter specifies the maximal number of concurrent connections that will be sent to this server. If the number of incoming concurrent requests goes higher than this value, they will be queued, waiting for a connection to be released. This parameter is very important as it can save fragile servers from going down under extreme loads. If a "minconn" parameter is specified, the limit becomes dynamic. The default value is "0" which means unlimited. See also the "minconn" and "maxqueue" parameters, and the backend's "fullconn" keyword.

Supported in default-server: Yes

maxqueue <maxqueue>

The "maxqueue" parameter specifies the maximal number of connections which will wait in the queue for this server. If this limit is reached, next requests will be redispatched to other servers instead of indefinitely waiting to be served. This will break persistence but may allow people to quickly re-log in when the server they try to connect to is dying. The default value is "0" which means the queue is unlimited. See also the "maxconn" and "minconn" parameters.

Supported in default-server: Yes

minconn <minconn>

When the "minconn" parameter is set, the maxconn limit becomes a dynamic limit following the backend's load. The server will always accept at least <minconn> connections, never more than <maxconn>, and the limit will be on the ramp between both values when the backend has less than <fullconn> concurrent connections. This makes it possible to limit the load on the server during normal loads, but push it further for important loads without overloading the server during exceptional loads. See also the "maxconn" and "maxqueue" parameters, as well as the "fullconn" backend keyword.

Supported in default-server: Yes

namespace <name>

On Linux, it is possible to specify which network namespace a socket will belong to. This directive makes it possible to explicitly bind a server to a namespace different from the default one. Please refer to your operating system's documentation to find more details about network namespaces.

no-ssl-reuse

This option disables SSL session reuse when SSL is used to communicate with the server. It will force the server to perform a full handshake for every new connection. It's probably only useful for benchmarking, troubleshooting, and for paranoid users.

Supported in default-server: No

no-sslv3

This option disables support for SSLv3 when SSL is used to communicate with the server. Note that SSLv2 is disabled in the code and cannot be enabled using any configuration option. See also "force-sslv3", "force-tlsv*".

Supported in default-server: No

no-tls-tickets

This setting is only available when support for OpenSSL was built in. It disables the stateless session resumption (RFC 5077 TLS Ticket extension) and force to use stateful session resumption. Stateless session resumption is more expensive in CPU usage for servers. This option is also available on global statement "ssl-default-server-options".

Supported in default-server: No

no-tlsv10

This option disables support for TLSv1.0 when SSL is used to communicate with the server. Note that SSLv2 is disabled in the code and cannot be enabled using any configuration option. TLSv1 is more expensive than SSLv3 so it often makes sense to disable it when communicating with local servers. This option is also available on global statement "ssl-default-server-options". See also "force-sslv3", "force-tlsv*".

Supported in default-server: No

no-tlsv11

This option disables support for TLSv1.1 when SSL is used to communicate with the server. Note that SSLv2 is disabled in the code and cannot be enabled using any configuration option. TLSv1 is more expensive than SSLv3 so it often makes sense to disable it when communicating with local servers. This option is also available on global statement "ssl-default-server-options". See also "force-sslv3", "force-tlsv*".

Supported in default-server: No

no-tlsv12

This option disables support for TLSv1.2 when SSL is used to communicate with the server. Note that SSLv2 is disabled in the code and cannot be enabled using any configuration option. TLSv1 is more expensive than SSLv3 so it often makes sense to disable it when communicating with local servers. This option is also available on global statement "ssl-default-server-options". See also "force-sslv3", "force-tlsv*".

Supported in default-server: No

non-stick

Never add connections allocated to this sever to a stick-table. This may be used in conjunction with backup to ensure that stick-table persistence is disabled for backup servers.

Supported in default-server: No

observe <mode>

This option enables health adjusting based on observing communication with the server. By default this functionality is disabled and enabling it also requires to enable health checks. There are two supported modes: "layer4" and "layer7". In layer4 mode, only successful/unsuccessful tcp connections are significant. In layer7, which is only allowed for http proxies, responses received from server are verified, like valid/wrong http code, unparsable headers, a timeout, etc. Valid status codes include 100 to 499, 501 and 505.

```
  Supported in default-server: No

  See also the "check", "on-error" and "error-limit".

on-error <mode>
  Select what should happen when enough consecutive errors are detected.
  Currently, four modes are available:
  - fastinter: force fastinter
  - fail-check: simulate a failed check, also forces fastinter (default)
  - sudden-death: simulate a pre-fatal failed health check, one more failed
    check will mark a server down, forces fastinter
  - mark-down: mark the server immediately down and force fastinter

  Supported in default-server: Yes

  See also the "check", "observe" and "error-limit".

on-marked-down <action>
  Modify what occurs when a server is marked down.
  Currently one action is available:
  - shutdown-sessions: Shutdown peer sessions. When this setting is enabled,
    all connections to the server are immediately terminated when the server
    goes down. It might be used if the health check detects more complex cases
    than a simple connection status, and long timeouts would cause the service
    to remain unresponsive for too long a time. For instance, a health check
    might detect that a database is stuck and that there's no chance to reuse
    existing connections anymore. Connections killed this way are logged with
    a 'D' termination code (for "Down").

  Actions are disabled by default

  Supported in default-server: Yes

on-marked-up <action>
  Modify what occurs when a server is marked up.
  Currently one action is available:
  - shutdown-backup-sessions: Shutdown sessions on all backup servers. This is
    done only if the server is not in backup state and if it is not disabled
    (it must have an effective weight > 0). This can be used sometimes to force
    an active server to take all the traffic back after recovery when dealing
    with long sessions (eg: LDAP, SQL, ...). Doing this can cause more trouble
    than it tries to solve (eg: incomplete transactions), so use this feature
    with extreme care. Sessions killed because a server comes up are logged
    with an 'U' termination code (for "Up").

  Actions are disabled by default

  Supported in default-server: Yes

port <port>
  Using the "port" parameter, it becomes possible to use a different port to
  send health-checks. On some servers, it may be desirable to dedicate a port
  to a specific component able to perform complex tests which are more suitable
  to health-checks than the application. It is common to run a simple script in
  inetd for instance. This parameter is ignored if the "check" parameter is not
  set. See also the "addr" parameter.

  Supported in default-server: Yes

redir <prefix>
  The "redir" parameter enables the redirection mode for all GET and HEAD
  requests addressing this server. This means that instead of having HAProxy
  forward the request to the server, it will send an "HTTP 302" response with
  the "Location" header composed of this prefix immediately followed by the
  requested URI beginning at the leading '/' of the path component. That means
```

```
  that no trailing slash should be used after <prefix>. All invalid requests
  will be rejected, and all non-GET or HEAD requests will be normally served by
  the server. Note that since the response is completely forged, no header
  mangling nor cookie insertion is possible in the response. However, cookies in
  requests are still analysed, making this solution completely usable to direct
  users to a remote location in case of local disaster. Main use consists in
  increasing bandwidth for static servers by having the clients directly
  connect to them. Note: never use a relative location here, it would cause a
  loop between the client and HAProxy!

  Example :  server srv1 192.168.1.1:80 redir http://image1.mydomain.com check

  Supported in default-server: No

rise <count>
  The "rise" parameter states that a server will be considered as operational
  after <count> consecutive successful health checks. This value defaults to 2
  if unspecified. See also the "check", "inter" and "fall" parameters.

  Supported in default-server: Yes

resolve-prefer <family>
  When DNS resolution is enabled for a server and multiple IP addresses from
  different families are returned, HAProxy will prefer using an IP address
  from the family mentioned in the "resolve-prefer" parameter.
  Available families: "ipv4" and "ipv6"

  Default value: ipv6

  Supported in default-server: Yes

  Example: server s1 app1.domain.com:80 resolvers mydns resolve-prefer ipv6

resolvers <id>
  Points to an existing "resolvers" section to resolve current server's
  hostname.
  In order to be operational, DNS resolution requires that health check is
  enabled on the server. Actually, health checks triggers the DNS resolution.
  You must precise one 'resolvers' parameter on each server line where DNS
  resolution is required.

  Supported in default-server: No

  Example: server s1 app1.domain.com:80 check resolvers mydns

  See also chapter 5.3

send-proxy
  The "send-proxy" parameter enforces use of the PROXY protocol over any
  connection established to this server. The PROXY protocol informs the other
  end about the layer 3/4 addresses of the incoming connection, so that it can
  know the client's address or the public address it accessed to, whatever the
  upper layer protocol. For connections accepted by an "accept-proxy" listener,
  the advertised address will be used. Only TCPv4 and TCPv6 address families
  are supported. Other families such as Unix sockets, will report an UNKNOWN
  family. Servers using this option can fully be chained to another instance of
  haproxy listening with an "accept-proxy" setting. This setting must not be
  used if the server isn't aware of the protocol. When health checks are sent
  to the server, the PROXY protocol is automatically used when this option is
  set, unless there is an explicit "port" or "addr" directive, in which case an
  explicit "check-send-proxy" directive would also be needed to use the PROXY
  protocol. See also the "accept-proxy" option of the "bind" keyword.

  Supported in default-server: No
```

send-proxy-v2

The "send-proxy-v2" parameter enforces use of the PROXY protocol version 2 over any connection established to this server. The PROXY protocol informs the other end about the layer 3/4 addresses of the incoming connection, so that it can know the client's address or the public address it accessed to, whatever the upper layer protocol. This setting must not be used if the server isn't aware of this version of the protocol. See also the "send-proxy" option of the "bind" keyword.

Supported in default-server: No

send-proxy-v2-ssl

The "send-proxy-v2-ssl" parameter enforces use of the PROXY protocol version 2 over any connection established to this server. The PROXY protocol informs the other end about the layer 3/4 addresses of the incoming connection, so that it can know the client's address or the public address it accessed to, whatever the upper layer protocol. In addition, the SSL information extension of the PROXY protocol is added to the PROXY protocol header. This setting must not be used if the server isn't aware of this version of the protocol. See also the "send-proxy-v2" option of the "bind" keyword.

Supported in default-server: No

send-proxy-v2-ssl-cn

The "send-proxy-v2-ssl" parameter enforces use of the PROXY protocol version 2 over any connection established to this server. The PROXY protocol informs the other end about the layer 3/4 addresses of the incoming connection, so that it can know the client's address or the public address it accessed to, whatever the upper layer protocol. In addition, the SSL information extension of the PROXY protocol, along along with the Common Name from the subject of the client certificate (if any), is added to the PROXY protocol header. This setting must not be used if the server isn't aware of this version of the protocol. See also the "send-proxy-v2" option of the "bind" keyword.

Supported in default-server: No

slowstart <start_time_in_ms>

The "slowstart" parameter for a server accepts a value in milliseconds which indicates after how long a server which has just come back up will run at full speed. Just as with every other time-based parameter, it can be entered in any other explicit unit among { us, ms, s, m, h, d }. The speed grows linearly from 0 to 100% during this time. The limitation applies to two parameters :

- maxconn: the number of connections accepted by the server will grow from 1 to 100% of the usual dynamic limit defined by (minconn,maxconn,fullconn).

- weight: when the backend uses a dynamic weighted algorithm, the weight grows linearly from 1 to 100%. In this case, the weight is updated at every health-check. For this reason, it is important that the "inter" parameter is smaller than the "slowstart", in order to maximize the number of steps.

The slowstart never applies when haproxy starts, otherwise it would cause trouble to running servers. It only applies when a server has been previously seen as failed.

Supported in default-server: Yes

sni <expression>

The "sni" parameter evaluates the sample fetch expression, converts it to a string and uses the result as the host name sent in the SNI TLS extension to the server. A typical use case is to send the SNI received from the client in a bridged HTTPS scenario, using the "ssl_fc_sni" sample fetch for the expression, though alternatives such as req.hdr(host) can also make sense.

Supported in default-server: no

source <addr>[:<pl>[-<ph>]] [usesrc { <addr2>[:<port2>] | client | clientip }]
source <addr>[:<port>] [usesrc { <addr2>[:<port2>] | hdr_ip(<hdr>[,<occ>]) }]
source <addr>[:<pl>[-<ph>]] [interface <name>] ...

The "source" parameter sets the source address which will be used when connecting to the server. It follows the exact same parameters and principle as the backend "source" keyword, except that it only applies to the server referencing it. Please consult the "source" keyword for details.

Additionally, the "source" statement on a server line allows one to specify a source port range by indicating the lower and higher bounds delimited by a dash ('-'). Some operating systems might require a valid IP address when a source port range is specified. It is permitted to have the same IP/range for several servers. Doing so makes it possible to bypass the maximum of 64k total concurrent connections. The limit will then reach 64k connections per server.

Supported in default-server: No

ssl

This option enables SSL ciphering on outgoing connections to the server. It is critical to verify server certificates using "verify" when using SSL to connect to servers, otherwise the communication is prone to trivial man in the-middle attacks rendering SSL useless. When this option is used, health checks are automatically sent in SSL too unless there is a "port" or an "addr" directive indicating the check should be sent to a different location. See the "check-ssl" option to force SSL health checks.

Supported in default-server: No

tcp-ut <delay>

Sets the TCP User Timeout for all outgoing connections to this server. This option is available on Linux since version 2.6.37. It allows haproxy to configure a timeout for sockets which contain data not receiving an acknoledgement for the configured delay. This is especially useful on long-lived connections experiencing long idle periods such as remote terminals or database connection pools, where the client and server timeouts must remain high to allow a long period of idle, but where it is important to detect that the server has disappeared in order to release all resources associated with its connection (and the client's session). One typical use case is also to force dead server connections to die when health checks are too slow or during a soft reload since health checks are then disabled. The argument is a delay expressed in milliseconds by default. This only works for regular TCP connections, and is ignored for other protocols.

track [<proxy>/]<server>

This option enables ability to set the current state of the server by tracking another one. It is possible to track a server which itself tracks another server, provided that at the end of the chain, a server has health checks enabled. If <proxy> is omitted the current one is used. If disable-on-404 is used, it has to be enabled on both proxies.

Supported in default-server: No

verify [none|required]

This setting is only available when support for OpenSSL was built in. If set to 'none', server certificate is not verified. In the other case, The certificate provided by the server is verified using CAs from 'ca-file' and optional CRLs from 'crl-file'. If 'ssl_server_verify' is not specified in global section, this is the default. On verify failure the handshake is aborted. It is critically important to verify server certificates when using SSL to connect to servers, otherwise the communication is prone to trivial man-in-the-middle attacks rendering SSL totally useless.

Supported in default-server: No

verifyhost <hostname>

This setting is only available when support for OpenSSL was built in, and only takes effect if 'verify required' is also specified. When set, the hostnames in the subject and subjectAlternateNames of the certificate provided by the server are checked. If none of the hostnames in the certificate match the specified hostname, the handshake is aborted. The hostnames in the server-provided certificate may include wildcards.

Supported in default-server: No

weight <weight>

The "weight" parameter is used to adjust the server's weight relative to other servers. All servers will receive a load proportional to their weight relative to the sum of all weights, so the higher the weight, the higher the load. The default weight is 1, and the maximal value is 256. A value of 0 means the server will not participate in load-balancing but will still accept persistent connections. If this parameter is used to distribute the load according to server's capacity, it is recommended to start with values which can both grow and shrink, for instance between 10 and 100 to leave enough room above and below for later adjustments.

Supported in default-server: Yes

5.3. Server IP address resolution using DNS

HAProxy allows using a host name on the server line to retrieve its IP address using name servers. By default, HAProxy resolves the name when parsing the configuration file, at startup and cache the result for the process' life. This is not sufficient in some cases, such as in Amazon where a server's IP can change after a reboot or an ELB Virtual IP can change based on current workload.
This chapter describes how HAProxy can be configured to process server's name resolution at run time.
Whether run time server name resolution has been enable or not, HAProxy will carry on doing the first resolution when parsing the configuration.

Bear in mind that DNS resolution is triggered by health checks. This makes health checks mandatory to allow DNS resolution.

5.3.1. Global overview

As we've seen in introduction, name resolution in HAProxy occurs at two different steps of the process life:

1. when starting up, HAProxy parses the server line definition and matches a host name. It uses libc functions to get the host name resolved. This resolution relies on /etc/resolv.conf file.

2. at run time, when HAProxy gets prepared to run a health check on a server, it verifies if the current name resolution is still considered as valid. If not, it processes a new resolution, in parallel of the health check.

A few other events can trigger a name resolution at run time:
- when a server's health check ends up in a connection timeout: this may be because the server has a new IP address. So we need to trigger a name resolution to know this new IP.

A few things important to notice:
- all the name servers are queried in the mean time. HAProxy will process the

first valid response.

- a resolution is considered as invalid (NX, timeout, refused), when all the servers return an error.

5.3.2. The resolvers section

This section is dedicated to host information related to name resolution in HAProxy.
There can be as many as resolvers section as needed. Each section can contain many name servers.

When multiple name servers are configured in a resolvers section, then HAProxy uses the first valid response. In case of invalid responses, only the last one is treated. Purpose is to give the chance to a slow server to deliver a valid answer after a fast faulty or outdated server.

When each server returns a different error type, then only the last error is used by HAProxy to decide what type of behavior to apply.

Two types of behavior can be applied:
1. stop DNS resolution
2. replay the DNS query with a new query type
   In such case, the following types are applied in this exact order:
   1. ANY query type
   2. query type corresponding to family pointed by resolve-prefer server's parameter
   3. remaining family type

HAProxy stops DNS resolution when the following errors occur:
- invalid DNS response packet
- wrong name in the query section of the response
- NX domain
- Query refused by server
- CNAME not pointing to an IP address

HAProxy tries a new query type when the following errors occur:
- no Answer records in the response
- DNS response truncated
- Error in DNS response
- No expected DNS records found in the response
- name server timeout

For example, with 2 name servers configured in a resolvers section:
- first response is valid and is applied directly, second response is ignored
- first response is invalid and second one is valid, then second response is applied;
- first response is a NX domain and second one a truncated response, then HAProxy replays the query with a new type;
- first response is truncated and second one is a NX Domain, then HAProxy stops resolution.

resolvers <resolvers id>

Creates a new name server list labelled <resolvers id>

A resolvers section accept the following parameters:

nameserver <id> <ip>:<port>

DNS server description:
```
    <id>   : label of the server, should be unique
    <ip>   : IP address of the server
    <port> : port where the DNS service actually runs
```

```

hold <status> <period>
  Defines <period> during which the last name resolution should be kept based
  on last resolution <status>
    <status> : last name resolution status. Only "valid" is accepted for now.
    <period> : interval between two successive name resolution when the last
               answer was in <status>. It follows the HAProxy time format.
               <period> is in milliseconds by default.

  Default value is 10s for "valid".

  Note: since the name resolution is triggered by the health checks, a new
        resolution is triggered after <period> modulo the <inter> parameter of
        the healch check.

resolve_retries <nb>
  Defines the number <nb> of queries to send to resolve a server name before
  giving up.
  Default value: 3

  A retry occurs on name server timeout or when the full sequence of DNS query
  type failover is over and we need to start up from the default ANY query
  type.

timeout <event> <time>
  Defines timeouts related to name resolution
     <event> : the event on which the <time> timeout period applies to.
               events available are:
               - retry: time between two DNS queries, when no response have
                        been received.
                        Default value: 1s
     <time>  : time related to the event. It follows the HAProxy time format.
               <time> is expressed in milliseconds.

Example of a resolvers section (with default values):

   resolvers mydns
     nameserver dns1 10.0.0.1:53
     nameserver dns2 10.0.0.2:53
     resolve_retries       3
     timeout retry         1s
     hold valid           10s


6. HTTP header manipulation
---------------------------

In HTTP mode, it is possible to rewrite, add or delete some of the request and
response headers based on regular expressions. It is also possible to block a
request or a response if a particular header matches a regular expression,
which is enough to stop most elementary protocol attacks, and to protect
against information leak from the internal network.

If HAProxy encounters an "Informational Response" (status code 1xx), it is able
to process all rsp* rules which can allow, deny, rewrite or delete a header,
but it will refuse to add a header to any such messages as this is not
HTTP-compliant. The reason for still processing headers in such responses is to
stop and/or fix any possible information leak which may happen, for instance
because another downstream equipment would unconditionally add a header, or if
a server name appears there. When such messages are seen, normal processing
still occurs on the next non-informational messages.

This section covers common usage of the following keywords, described in detail
in section 4.2 :

```

```
  - reqadd     <string>
  - reqallow   <search>
  - reqiallow  <search>
  - reqdel     <search>
  - reqidel    <search>
  - reqdeny    <search>
  - reqideny   <search>
  - reqpass    <search>
  - reqipass   <search>
  - reqrep     <search> <replace>
  - reqirep    <search> <replace>
  - reqtarpit  <search>
  - reqitarpit <search>
  - rspadd     <string>
  - rspdel     <search>
  - rspidel    <search>
  - rspdeny    <search>
  - rspideny   <search>
  - rsprep     <search> <replace>
  - rspirep    <search> <replace>

With all these keywords, the same conventions are used. The <search> parameter
is a POSIX extended regular expression (regex) which supports grouping through
parenthesis (without the backslash). Spaces and other delimiters must be
prefixed with a backslash ('\') to avoid confusion with a field delimiter.
Other characters may be prefixed with a backslash to change their meaning :

  \t   for a tab
  \r   for a carriage return (CR)
  \n   for a new line (LF)
  \    to mark a space and differentiate it from a delimiter
  \#   to mark a sharp and differentiate it from a comment
  \\   to use a backslash in a regex
  \\\\ to use a backslash in the text (*2 for regex, *2 for haproxy)
  \xXX to write the ASCII hex code XX as in the C language

The <replace> parameter contains the string to be used to replace the largest
portion of text matching the regex. It can make use of the special characters
above, and can reference a substring which is delimited by parenthesis in the
regex, by writing a backslash ('\') immediately followed by one digit from 0 to
9 indicating the group position (0 designating the entire line). This practice
is very common to users of the "sed" program.

The <string> parameter represents the string which will systematically be added
after the last header line. It can also use special character sequences above.

Notes related to these keywords :
---------------------------------
  - these keywords are not always convenient to allow/deny based on header
    contents. It is strongly recommended to use ACLs with the "block" keyword
    instead, resulting in far more flexible and manageable rules.

  - lines are always considered as a whole. It is not possible to reference
    a header name only or a value only. This is important because of the way
    headers are written (notably the number of spaces after the colon).

  - the first line is always considered as a header, which makes it possible to
    rewrite or filter HTTP requests URIs or response codes, but in turn makes
    it harder to distinguish between headers and request line. The regex prefix
    ^[^\ \t]*[\ \t] matches any HTTP method followed by a space, and the prefix
    ^[^ \t:]*: matches any header name followed by a colon.

  - for performances reasons, the number of characters added to a request or to
    a response is limited at build time to values between 1 and 4 kB. This
    should normally be far more than enough for most usages. If it is too short
```

on occasional usages, it is possible to gain some space by removing some useless headers before adding new ones.

- keywords beginning with "reqi" and "rspi" are the same as their counterpart without the 'i' letter except that they ignore case when matching patterns.

- when a request passes through a frontend then a backend, all req* rules from the frontend will be evaluated, then all req* rules from the backend will be evaluated. The reverse path is applied to responses.

- req* statements are applied after "block" statements, so that "block" is always the first one, but before "use_backend" in order to permit rewriting before switching.

7. Using ACLs and fetching samples

Haproxy is capable of extracting data from request or response streams, from client or server information, from tables, environmental information etc... The action of extracting such data is called fetching a sample. Once retrieved, these samples may be used for various purposes such as a key to a stick-table, but most common usages consist in matching them against predefined constant data called patterns.

7.1. ACL basics

The use of Access Control Lists (ACL) provides a flexible solution to perform content switching and generally to take decisions based on content extracted from the request, the response or any environmental status. The principle is simple :

- extract a data sample from a stream, table or the environment
- optionally apply some format conversion to the extracted sample
- apply one or multiple pattern matching methods on this sample
- perform actions only when a pattern matches the sample

The actions generally consist in blocking a request, selecting a backend, or adding a header.

In order to define a test, the "acl" keyword is used. The syntax is :

```
   acl <aclname> <criterion> [flags] [operator] [<value>] ...
```

This creates a new ACL <aclname> or completes an existing one with new tests. Those tests apply to the portion of request/response specified in <criterion> and may be adjusted with optional flags [flags]. Some criteria also support an operator which may be specified before the set of values. Optionally some conversion operators may be applied to the sample, and they will be specified as a comma-delimited list of keywords just after the first keyword. The values are of the type supported by the criterion, and are separated by spaces.

ACL names must be formed from upper and lower case letters, digits, '-' (dash), '_' (underscore) , '.' (dot) and ':' (colon). ACL names are case-sensitive, which means that "my_acl" and "My_Acl" are two different ACLs.

There is no enforced limit to the number of ACLs. The unused ones do not affect performance, they just consume a small amount of memory.

The criterion generally is the name of a sample fetch method, or one of its ACL specific declinations. The default test method is implied by the output type of this sample fetch method. The ACL declinations can describe alternate matching methods of a same sample fetch method. The sample fetch methods are the only

ones supporting a conversion.

Sample fetch methods return data which can be of the following types :
- boolean
- integer (signed or unsigned)
- IPv4 or IPv6 address
- string
- data block

Converters transform any of these data into any of these. For example, some converters might convert a string to a lower-case string while other ones would turn a string to an IPv4 address, or apply a netmask to an IP address. The resulting sample is of the type of the last converter applied to the list, which defaults to the type of the sample fetch method.

Each sample or converter returns data of a specific type, specified with its keyword in this documentation. When an ACL is declared using a standard sample fetch method, certain types automatically involved a default matching method which are summarized in the table below :

Sample or converter output type	Default matching method
boolean	bool
integer	int
ip	ip
string	str
binary	none, use "-m"

Note that in order to match a binary samples, it is mandatory to specify a matching method, see below.

The ACL engine can match these types against patterns of the following types :
- boolean
- integer or integer range
- IP address / network
- string (exact, substring, suffix, prefix, subdir, domain)
- regular expression
- hex block

The following ACL flags are currently supported :

```
   -i : ignore case during matching of all subsequent patterns.
   -f : load patterns from a file.
   -m : use a specific pattern matching method
   -n : forbid the DNS resolutions
   -M : load the file pointed by -f like a map file.
   -u : force the unique id of the ACL
   -- : force end of flags. Useful when a string looks like one of the flags.
```

The "-f" flag is followed by the name of a file from which all lines will be read as individual values. It is even possible to pass multiple "-f" arguments if the patterns are to be loaded from multiple files. Empty lines as well as lines beginning with a sharp ('#') will be ignored. All leading spaces and tabs will be stripped. If it is absolutely necessary to insert a valid pattern beginning with a sharp, just prefix it with a space so that it is not taken for a comment. Depending on the data type and match method, haproxy may load the lines into a binary tree, allowing very fast lookups. This is true for IPv4 and exact string matching. In this case, duplicates will automatically be removed.

The "-M" flag allows an ACL to use a map file. If this flag is set, the file is parsed as two column file. The first column contains the patterns used by the ACL, and the second column contain the samples. The sample can be used later by a map. This can be useful in some rare cases where an ACL would just be used to check for the existence of a pattern in a map before a mapping is applied.

The "-u" flag forces the unique id of the ACL. This unique id is used with the socket interface to identify ACL and dynamically change its values. Note that a file is always identified by its name even if an id is set.

Also, note that the "-i" flag applies to subsequent entries and not to entries loaded from files preceding it. For instance :

```
    acl valid-ua hdr(user-agent) -f exact-ua.lst -i -f generic-ua.lst test
```

In this example, each line of "exact-ua.lst" will be exactly matched against the "user-agent" header of the request. Then each line of "generic-ua" will be case-insensitively matched. Then the word "test" will be insensitively matched as well.

The "-m" flag is used to select a specific pattern matching method on the input sample. All ACL-specific criteria imply a pattern matching method and generally do not need this flag. However, this flag is useful with generic sample fetch methods to describe how they're going to be matched against the patterns. This is required for sample fetches which return data type for which there is no obvious matching method (eg: string or binary). When "-m" is specified and followed by a pattern matching method name, this method is used instead of the default one for the criterion. This makes it possible to match contents in ways that were not initially planned, or with sample fetch methods which return a string. The matching method also affects the way the patterns are parsed.

The "-n" flag forbids the dns resolutions. It is used with the load of ip files. By default, if the parser cannot parse ip address it considers that the parsed string is maybe a domain name and try dns resolution. The flag "-n" disable this resolution. It is useful for detecting malformed ip lists. Note that if the DNS server is not reachable, the haproxy configuration parsing may last many minutes waiting fir the timeout. During this time no error messages are displayed. The flag "-n" disable this behavior. Note also that during the runtime, this function is disabled for the dynamic acl modifications.

There are some restrictions however. Not all methods can be used with all sample fetch methods. Also, if "-m" is used in conjunction with "-f", it must be placed first. The pattern matching method must be one of the following :

- "found" : only check if the requested sample could be found in the stream, but do not compare it against any pattern. It is recommended not to pass any pattern to avoid confusion. This matching method is particularly useful to detect presence of certain contents such as headers, cookies, etc... even if they are empty and without comparing them to anything nor counting them.

- "bool" : check the value as a boolean. It can only be applied to fetches which return a boolean or integer value, and takes no pattern. Value zero or false does not match, all other values do match.

- "int" : match the value as an integer. It can be used with integer and boolean samples. Boolean false is integer 0, true is integer 1.

- "ip" : match the value as an IPv4 or IPv6 address. It is compatible with IP address samples only, so it is implied and never needed.

- "bin" : match the contents against an hexadecimal string representing a binary sequence. This may be used with binary or string samples.

- "len" : match the sample's length as an integer. This may be used with binary or string samples.

- "str" : exact match : match the contents against a string. This may be used with binary or string samples.

- "sub" : substring match : check that the contents contain at least one of the provided string patterns. This may be used with binary or string samples.

- "reg" : regex match : match the contents against a list of regular expressions. This may be used with binary or string samples.

- "beg" : prefix match : check that the contents begin like the provided string patterns. This may be used with binary or string samples.

- "end" : suffix match : check that the contents end like the provided string patterns. This may be used with binary or string samples.

- "dir" : subdir match : check that a slash-delimited portion of the contents exactly matches one of the provided string patterns. This may be used with binary or string samples.

- "dom" : domain match : check that a dot-delimited portion of the contents exactly match one of the provided string patterns. This may be used with binary or string samples.

For example, to quickly detect the presence of cookie "JSESSIONID" in an HTTP request, it is possible to do :

```
    acl jsess_present cook(JSESSIONID) -m found
```

In order to apply a regular expression on the 500 first bytes of data in the buffer, one would use the following acl :

```
    acl script_tag payload(0,500) -m reg -i <script>
```

On systems where the regex library is much slower when using "-i", it is possible to convert the sample to lowercase before matching, like this :

```
    acl script_tag payload(0,500),lower -m reg <script>
```

All ACL-specific criteria imply a default matching method. Most often, these criteria are composed by concatenating the name of the original sample fetch method and the matching method. For example, "hdr_beg" applies the "beg" match to samples retrieved using the "hdr" fetch method. Since all ACL-specific criteria rely on a sample fetch method, it is always possible instead to use the original sample fetch method and the explicit matching method using "-m".

If an alternate match is specified using "-m" on an ACL-specific criterion, the matching method is simply applied to the underlying sample fetch method. For example, all ACLs below are exact equivalent :

```
    acl short_form  hdr_beg(host)        www.
    acl alternate1  hdr_beg(host) -m beg www.
    acl alternate2  hdr_dom(host) -m beg www.
    acl alternate3  hdr(host)     -m beg www.
```

The table below summarizes the compatibility matrix between sample or converter types and the pattern types to fetch against. It indicates for each compatible combination the name of the matching method to be used, surrounded with angle brackets ">" and "<" when the method is the default one and will work by default without "-m".

	Input sample type				
pattern type	boolean	integer	ip	string	binary
none (presence only)	found	found	found	found	found
none (boolean value)	> bool <	bool		bool	
integer (value)	int	> int <	int	int	
integer (length)	len	len	len	len	len
IP address			> ip <	ip	ip
exact string	str	str	str	> str <	str
prefix	beg	beg	beg	beg	beg
suffix	end	end	end	end	end
substring	sub	sub	sub	sub	sub
subdir	dir	dir	dir	dir	dir
domain	dom	dom	dom	dom	dom
regex	reg	reg	reg	reg	reg
hex block				bin	bin

7.1.1. Matching booleans

In order to match a boolean, no value is needed and all values are ignored. Boolean matching is used by default for all fetch methods of type "boolean". When boolean matching is used, the fetched value is returned as-is, which means that a boolean "true" will always match and a boolean "false" will never match.

Boolean matching may also be enforced using "-m bool" on fetch methods which return an integer value. Then, integer value 0 is converted to the boolean "false" and all other values are converted to "true".

7.1.2. Matching integers

Integer matching applies by default to integer fetch methods. It can also be enforced on boolean fetches using "-m int". In this case, "false" is converted to the integer 0, and "true" is converted to the integer 1.

Integer matching also supports integer ranges and operators. Note that integer matching only applies to positive values. A range is a value expressed with a lower and an upper bound separated with a colon, both of which may be omitted.

For instance, "1024:65535" is a valid range to represent a range of unprivileged ports, and "1024:" would also work. "0:1023" is a valid representation of privileged ports, and ":1023" would also work.

As a special case, some ACL functions support decimal numbers which are in fact two integers separated by a dot. This is used with some version checks for instance. All integer properties apply to those decimal numbers, including ranges and operators.

For an easier usage, comparison operators are also supported. Note that using operators with ranges does not make much sense and is strongly discouraged. Similarly, it does not make much sense to perform order comparisons with a set of values.

Available operators for integer matching are :

```
  eq : true if the tested value equals at least one value
  ge : true if the tested value is greater than or equal to at least one value
  gt : true if the tested value is greater than at least one value
  le : true if the tested value is less than or equal to at least one value
  lt : true if the tested value is less than at least one value
```

For instance, the following ACL matches any negative Content-Length header :

```
  acl negative-length hdr_val(content-length) lt 0
```

This one matches SSL versions between 3.0 and 3.1 (inclusive) :

```
  acl sslv3 req_ssl_ver 3:3.1
```

7.1.3. Matching strings

String matching applies to string or binary fetch methods, and exists in 6 different forms :

- exact match   (-m str) : the extracted string must exactly match the patterns ;

- substring match (-m sub) : the patterns are looked up inside the extracted string, and the ACL matches if any of them is found inside ;

- prefix match   (-m beg) : the patterns are compared with the beginning of the extracted string, and the ACL matches if any of them matches.

- suffix match   (-m end) : the patterns are compared with the end of the extracted string, and the ACL matches if any of them matches.

- subdir match   (-m sub) : the patterns are looked up inside the extracted string, delimited with slashes ("/"), and the ACL matches if any of them matches.

- domain match   (-m dom) : the patterns are looked up inside the extracted string, delimited with dots ("."), and the ACL matches if any of them matches.

String matching applies to verbatim strings as they are passed, with the exception of the backslash ("\") which makes it possible to escape some characters such as the space. If the "-i" flag is passed before the first string, then the matching will be performed ignoring the case. In order to match the string "-i", either set it second, or pass the "--" flag before the first string. Same applies of course to match the string "--".

7.1.4. Matching regular expressions (regexes)

Just like with string matching, regex matching applies to verbatim strings as they are passed, with the exception of the backslash ("\") which makes it possible to escape some characters such as the space. If the "-i" flag is passed before the first regex, then the matching will be performed ignoring the case. In order to match the string "-i", either set it second, or pass the "--" flag before the first string. Same principle applies of course to match the string "--".

7.1.5. Matching arbitrary data blocks

It is possible to match some extracted samples against a binary block which may not safely be represented as a string. For this, the patterns must be passed as a series of hexadecimal digits in an even number, when the match method is set to binary. Each sequence of two digits will represent a byte. The hexadecimal digits may be used upper or lower case.

Example :

```
    # match "Hello\n" in the input stream (\x48 \x65 \x6c \x6c \x6f \x0a)
    acl hello payload(0,6) -m bin 48656c6c6f0a
```

7.1.6. Matching IPv4 and IPv6 addresses

IPv4 addresses values can be specified either as plain addresses or with a netmask appended, in which case the IPv4 address matches whenever it is within the network. Plain addresses may also be replaced with a resolvable host name, but this practice is generally discouraged as it makes it more difficult to read and debug configurations. If hostnames are used, you should at least ensure that they are present in /etc/hosts so that the configuration does not depend on any random DNS match at the moment the configuration is parsed.

IPv6 may be entered in their usual form, with or without a netmask appended. Only bit counts are accepted for IPv6 netmasks. In order to avoid any risk of trouble with randomly resolved IP addresses, host names are never allowed in IPv6 patterns.

HAProxy is also able to match IPv4 addresses with IPv6 addresses in the following situations :

- tested address is IPv4, pattern address is IPv4, the match applies in IPv4 using the supplied mask if any.
- tested address is IPv6, pattern address is IPv6, the match applies in IPv6 using the supplied mask if any.
- tested address is IPv6, pattern address is IPv4, the match applies in IPv4 using the pattern's mask if the IPv6 address matches with 2002:IPV4::, ::IPV4 or ::ffff:IPV4, otherwise it fails.
- tested address is IPv4, pattern address is IPv6, the IPv4 address is first converted to IPv6 by prefixing ::ffff: in front of it, then the match is applied in IPv6 using the supplied IPv6 mask.

7.2. Using ACLs to form conditions

Some actions are only performed upon a valid condition. A condition is a combination of ACLs with operators. 3 operators are supported :

- AND (implicit)
- OR (explicit with the "or" keyword or the "||" operator)
- Negation with the exclamation mark ("!")

A condition is formed as a disjunctive form:

```
   [!]acl1 [!]acl2 ... [!]acln  { or [!]acl1 [!]acl2 ... [!]acln } ...
```

Such conditions are generally used after an "if" or "unless" statement, indicating when the condition will trigger the action.

For instance, to block HTTP requests to the "*" URL with methods other than "OPTIONS", as well as POST requests without content-length, and GET or HEAD requests with a content-length greater than 0, and finally every request which is not either GET/HEAD/POST/OPTIONS !

```
   acl missing_cl hdr_cnt(Content-length) eq 0
   block if HTTP_URL_STAR !METH_OPTIONS || METH_POST missing_cl
   block if METH_GET HTTP_CONTENT
   block unless METH_GET or METH_POST or METH_OPTIONS
```

To select a different backend for requests to static contents on the "www" site and to every request on the "img", "video", "download" and "ftp" hosts :

```
   acl url_static  path_beg         /static /images /img /css
   acl url_static  path_end         .gif .png .jpg .css .js
   acl host_www    hdr_beg(host) -i www
   acl host_static hdr_beg(host) -i img. video. download. ftp.

   # now use backend "static" for all static-only hosts, and for static urls
   # of host "www". Use backend "www" for the rest.
   use_backend static if host_static or host_www url_static
   use_backend www    if host_www
```

It is also possible to form rules using "anonymous ACLs". Those are unnamed ACL expressions that are built on the fly without needing to be declared. They must be enclosed between braces, with a space before and after each brace (because the braces must be seen as independent words). Example :

The following rule :

```
    acl missing_cl hdr_cnt(Content-length) eq 0
    block if METH_POST missing_cl
```

Can also be written that way :

```
    block if METH_POST { hdr_cnt(Content-length) eq 0 }
```

It is generally not recommended to use this construct because it's a lot easier to leave errors in the configuration when written that way. However, for very simple rules matching only one source IP address for instance, it can make more sense to use them than to declare ACLs with random names. Another example of good use is the following :

With named ACLs :

```
     acl site_dead nbsrv(dynamic) lt 2
     acl site_dead nbsrv(static)  lt 2
     monitor fail  if site_dead
```

With anonymous ACLs :

```
     monitor fail if { nbsrv(dynamic) lt 2 } || { nbsrv(static) lt 2 }
```

See section 4.2 for detailed help on the "block" and "use_backend" keywords.

7.3. Fetching samples

Historically, sample fetch methods were only used to retrieve data to match against patterns using ACLs. With the arrival of stick-tables, a new class of sample fetch methods was created, most often sharing the same syntax as their ACL counterpart. These sample fetch methods are also known as "fetches". As of now, ACLs and fetches have converged. All ACL fetch methods have been made available as fetch methods, and ACLs may use any sample fetch method as well.

This section details all available sample fetch methods and their output type. Some sample fetch methods have deprecated aliases that are used to maintain compatibility with existing configurations. They are then explicitly marked as deprecated and should not be used in new setups.

The ACL derivatives are also indicated when available, with their respective matching methods. These ones all have a well defined default pattern matching method, so it is never necessary (though allowed) to pass the "-m" option to indicate how the sample will be matched using ACLs.

As indicated in the sample type versus matching compatibility matrix above, when using a generic sample fetch method in an ACL, the "-m" option is mandatory unless the sample type is one of boolean, integer, IPv4 or IPv6. When the same keyword exists as an ACL keyword and as a standard fetch method, the ACL engine will automatically pick the ACL-only one by default.

Some of these keywords support one or multiple mandatory arguments, and one or multiple optional arguments. These arguments are strongly typed and are checked when the configuration is parsed so that there is no risk of running with an incorrect argument (eg: an unresolved backend name). Fetch function arguments are passed between parenthesis and are delimited by commas. When an argument is optional, it will be indicated below between square brackets ('[ ]'). When all arguments are optional, the parenthesis may be omitted.

Thus, the syntax of a standard sample fetch method is one of the following :

```
   - name
   - name(arg1)
   - name(arg1,arg2)
```

7.3.1. Converters

Sample fetch methods may be combined with transformations to be applied on top of the fetched sample (also called "converters"). These combinations form what is called "sample expressions" and the result is a "sample". Initially this was only supported by "stick on" and "stick store-request" directives but this has now be extended to all places where samples may be used (acls, log-format, unique-id-format, add-header, ...).

These transformations are enumerated as a series of specific keywords after the sample fetch method. These keywords may equally be appended immediately after the fetch keyword's argument, delimited by a comma. These keywords can also support some arguments (eg: a netmask) which must be passed in parenthesis.

A certain category of converters are bitwise and arithmetic operators which support performing basic operations on integers. Some bitwise operations are supported (and, or, xor, cpl) and some arithmetic operations are supported (add, sub, mul, div, mod, neg). Some comparators are provided (odd, even, not, bool) which make it possible to report a match without having to write an ACL.

The currently available list of transformation keywords include :

add(<value>)

  Adds <value> to the input value of type signed integer, and returns the result as a signed integer. <value> can be a numeric value or a variable name. The name of the variable starts by an indication about its scope. The allowed scopes are:

```
    "sess" : the variable is shared with all the session,
    "txn"  : the variable is shared with all the transaction (request and
             response),
    "req"  : the variable is shared only during the request processing,
    "res"  : the variable is shared only during the response processing.
```

  This prefix is followed by a name. The separator is a '.'. The name may only contain characters 'a-z', 'A-Z', '0-9' and '_'.

and(<value>)

  Performs a bitwise "AND" between <value> and the input value of type signed integer, and returns the result as an signed integer. <value> can be a numeric value or a variable name. The name of the variable starts by an indication about its scope. The allowed scopes are:

```
    "sess" : the variable is shared with all the session,
    "txn"  : the variable is shared with all the transaction (request and
             response),
    "req"  : the variable is shared only during the request processing,
    "res"  : the variable is shared only during the response processing.
```

  This prefix is followed by a name. The separator is a '.'. The name may only contain characters 'a-z', 'A-Z', '0-9' and '_'.

base64

  Converts a binary input sample to a base64 string. It is used to log or transfer binary content in a way that can be reliably transferred (eg: an SSL ID can be copied in a header).

bool

  Returns a boolean TRUE if the input value of type signed integer is non-null, otherwise returns FALSE. Used in conjunction with and(), it can be used to report true/false for bit testing on input values (eg: verify the presence of a flag).

bytes(<offset>[,<length>])

  Extracts some bytes from an input binary sample. The result is a binary sample starting at an offset (in bytes) of the original sample and optionnaly truncated at the given length.

cpl

  Takes the input value of type signed integer, applies a ones-complement (flips all bits) and returns the result as an signed integer.

crc32([<avalanche>])

  Hashes a binary input sample into an unsigned 32-bit quantity using the CRC32 hash function. Optionally, it is possible to apply a full avalanche hash function to the output if the optional <avalanche> argument equals 1. This converter uses the same functions as used by the various hash-based load balancing algorithms, so it will provide exactly the same results. It is provided for compatibility with other software which want a CRC32 to be computed on some input keys, so it follows the most common implementation as found in Ethernet, Gzip, PNG, etc... It is slower than the other algorithms but may provide a better or at least less predictable distribution. It must not be used for security purposes as a 32-bit hash is trivial to break. See also "djb2", "sdbm", "wt6" and the "hash-type" directive.

da-csv-conv(<prop>[,<prop>*])

  Asks the DeviceAtlas converter to identify the User Agent string passed on input, and to emit a string made of the concatenation of the properties enumerated in argument, delimited by the separator defined by the global keyword "deviceatlas-property-separator", or by default the pipe character ('|'). There's a limit of 5 different properties imposed by the haproxy configuration language.

```
  Example:
    frontend www
      bind *:8881
      default_backend servers
      http-request set-header X-DeviceAtlas-Data %[req.fhdr(User-Agent),da-csv(prima…
```

debug

  This converter is used as debug tool. It dumps on screen the content and the

```
  type of the input sample. The sample is returned as is on its output. This
  converter only exists when haproxy was built with debugging enabled.

div(<value>)
  Divides the input value of type signed integer by <value>, and returns the
  result as an signed integer. If <value> is null, the largest unsigned
  integer is returned (typically 2^63-1). <value> can be a numeric value or a
  variable name. The name of the variable starts by an indication about it
  scope. The scope allowed are:
    "sess" : the variable is shared with all the session,
    "txn"  : the variable is shared with all the transaction (request and
             response),
    "req"  : the variable is shared only during the request processing,
    "res"  : the variable is shared only during the response processing.
  This prefix is followed by a name. The separator is a '.'. The name may only
  contain characters 'a-z', 'A-Z', '0-9' and '_'.

djb2([<avalanche>])
  Hashes a binary input sample into an unsigned 32-bit quantity using the DJB2
  hash function. Optionally, it is possible to apply a full avalanche hash
  function to the output if the optional <avalanche> argument equals 1. This
  converter uses the same functions as used by the various hash-based load
  balancing algorithms, so it will provide exactly the same results. It is
  mostly intended for debugging, but can be used as a stick-table entry to
  collect rough statistics. It must not be used for security purposes as a
  32-bit hash is trivial to break. See also "crc32", "sdbm", "wt6" and the
  "hash-type" directive.

even
  Returns a boolean TRUE if the input value of type signed integer is even
  otherwise returns FALSE. It is functionally equivalent to "not,and(1),bool".

field(<index>,<delimiters>)
  Extracts the substring at the given index considering given delimiters from
  an input string. Indexes start at 1 and delimiters are a string formatted
  list of chars.

hex
  Converts a binary input sample to an hex string containing two hex digits per
  input byte. It is used to log or transfer hex dumps of some binary input data
  in a way that can be reliably transferred (eg: an SSL ID can be copied in a
  header).

http_date([<offset>])
  Converts an integer supposed to contain a date since epoch to a string
  representing this date in a format suitable for use in HTTP header fields. If
  an offset value is specified, then it is a number of seconds that is added to
  the date before the conversion is operated. This is particularly useful to
  emit Date header fields, Expires values in responses when combined with a
  positive offset, or Last-Modified values when the offset is negative.

in_table(<table>)
  Uses the string representation of the input sample to perform a look up in
  the specified table. If the key is not found in the table, a boolean false
  is returned. Otherwise a boolean true is returned. This can be used to verify
  the presence of a certain key in a table tracking some elements (eg: whether
  or not a source IP address or an Authorization header was already seen).

ipmask(<mask>)
  Apply a mask to an IPv4 address, and use the result for lookups and storage.
  This can be used to make all hosts within a certain mask to share the same
  table entries and as such use the same server. The mask can be passed in
  dotted form (eg: 255.255.255.0) or in CIDR form (eg: 24).

json([<input-code>])
```

```
  Escapes the input string and produces an ASCII ouput string ready to use as a
  JSON string. The converter tries to decode the input string according to the
  <input-code> parameter. It can be "ascii", "utf8", "utf8s", "utf8"" or
  "utf8ps". The "ascii" decoder never fails. The "utf8" decoder detects 3 types
  of errors:
   - bad UTF-8 sequence (lone continuation byte, bad number of continuation
     bytes, ...)
   - invalid range (the decoded value is within a UTF-8 prohibited range),
   - code overlong (the value is encoded with more bytes than necessary).

  The UTF-8 JSON encoding can produce a "too long value" error when the UTF-8
  character is greater than 0xffff because the JSON string escape specification
  only authorizes 4 hex digits for the value encoding. The UTF-8 decoder exists
  in 4 variants designated by a combination of two suffix letters : "p" for
  "permissive" and "s" for "silently ignore". The behaviors of the decoders
  are :
   - "ascii"  : never fails ;
   - "utf8"   : fails on any detected errors ;
   - "utf8s"  : never fails, but removes characters corresponding to errors ;
   - "utf8p"  : accepts and fixes the overlong errors, but fails on any other
                error ;
   - "utf8ps" : never fails, accepts and fixes the overlong errors, but removes
                characters corresponding to the other errors.

  This converter is particularly useful for building properly escaped JSON for
  logging to servers which consume JSON-formated traffic logs.

  Example:
     capture request header user-agent len 150
     capture request header Host len 15
     log-format {"ip":"%[src]","user-agent":"%[capture.req.hdr(1),json]"}

  Input request from client 127.0.0.1:
     GET / HTTP/1.0
     User-Agent: Very "Ugly" UA 1/2

  Output log:
     {"ip":"127.0.0.1","user-agent":"Very \"Ugly\" UA 1\/2"}

language(<value>[,<default>])
  Returns the value with the highest q-factor from a list as extracted from the
  "accept-language" header using "req.fhdr". Values with no q-factor have a
  q-factor of 1. Values with a q-factor of 0 are dropped. Only values which
  belong to the list of semi-colon delimited <values> will be considered. The
  argument <value> syntax is "lang[;lang[;lang[;...]]]". If no value matches the
  given list and a default value is provided, it is returned. Note that language
  names may have a variant after a dash ('-'). If this variant is present in the
  list, it will be matched, but if it is not, only the base language is checked.
  The match is case-sensitive, and the output string is always one of those
  provided in arguments.  The ordering of arguments is meaningless, only the
  ordering of the values in the request counts, as the first value among
  multiple sharing the same q-factor is used.

  Example :

    # this configuration switches to the backend matching a
    # given language based on the request :

    acl es req.fhdr(accept-language),language(es;fr;en) -m str es
    acl fr req.fhdr(accept-language),language(es;fr;en) -m str fr
    acl en req.fhdr(accept-language),language(es;fr;en) -m str en
    use_backend spanish if es
    use_backend french  if fr
    use_backend english if en
    default_backend choose_your_language
```

lower
  Convert a string sample to lower case. This can only be placed after a string sample fetch function or after a transformation keyword returning a string type. The result is of type string.

ltime(<format>[,<offset>])
  Converts an integer supposed to contain a date since epoch to a string representing this date in local time using a format defined by the <format> string using strftime(3). The purpose is to allow any date format to be used in logs. An optional <offset> in seconds may be applied to the input date (positive or negative). See the strftime() man page for the format supported by your operating system. See also the utime converter.

  Example :

```
      # Emit two colons, one with the local time and another with ip:port
      # Eg:  20140710162350 127.0.0.1:57325
      log-format %[date,ltime(%Y%m%d%H%M%S)]\ %ci:%cp
```

map(<map_file>[,<default_value>])
map_<match_type>(<map_file>[,<default_value>])
map_<match_type>_<output_type>(<map_file>[,<default_value>])
  Search the input value from <map_file> using the <match_type> matching method, and return the associated value converted to the type <output_type>. If the input value cannot be found in the <map_file>, the converter returns the <default_value>. If the <default_value> is not set, the converter fails and acts as if no input value could be fetched. If the <match_type> is not set, it defaults to "str". Likewise, if the <output_type> is not set, it defaults to "str". For convenience, the "map" keyword is an alias for "map_str" and maps a string to another string.

  It is important to avoid overlapping between the keys : IP addresses and strings are stored in trees, so the first of the finest match will be used. Other keys are stored in lists, so the first matching occurrence will be used.

  The following array contains the list of all map functions avalaible sorted by input type, match type and output type.

input type	match method	output type str	output type int	output type ip
str	str	map_str	map_str_int	map_str_ip
str	beg	map_beg	map_beg_int	map_end_ip
str	sub	map_sub	map_sub_int	map_sub_ip
str	dir	map_dir	map_dir_int	map_dir_ip
str	dom	map_dom	map_dom_int	map_dom_ip
str	end	map_end	map_end_int	map_end_ip
str	reg	map_reg	map_reg_int	map_reg_ip
int	int	map_int	map_int_int	map_int_ip
ip	ip	map_ip	map_ip_int	map_ip_ip

  The file contains one key + value per line. Lines which start with '#' are ignored, just like empty lines. Leading tabs and spaces are stripped. The key is then the first "word" (series of non-space/tabs characters), and the value is what follows this series of space/tab till the end of the line excluding trailing spaces/tabs.

  Example :

```
     # this is a comment and is ignored
        2.22.246.0/23    United Kingdom      \n
     <-><-----------><--><------------><---->
      |       |       |         |        `- trailing spaces ignored
      |       |       |         `---------- value
      |       |       `-------------------- middle spaces ignored
      |       `---------------------------- key
      `------------------------------------ leading spaces ignored
```

mod(<value>)
  Divides the input value of type signed integer by <value>, and returns the remainder as an signed integer. If <value> is null, then zero is returned. <value> can be a numeric value or a variable name. The name of the variable starts by an indication about its scope. The allowed scopes are:
    "sess" : the variable is shared with all the session,
    "txn" : the variable is shared with all the transaction (request and response),
    "req" : the variable is shared only during the request processing,
    "res" : the variable is shared only during the response processing.
  This prefix is followed by a name. The separator is a '.'. The name may only contain characters 'a-z', 'A-Z', '0-9' and '_'.

mul(<value>)
  Multiplies the input value of type signed integer by <value>, and returns the product as an signed integer. In case of overflow, the largest possible value for the sign is returned so that the operation doesn't wrap around. <value> can be a numeric value or a variable name. The name of the variable starts by an indication about its scope. The allowed scopes are:
    "sess" : the variable is shared with all the session,
    "txn" : the variable is shared with all the transaction (request and response),
    "req" : the variable is shared only during the request processing,
    "res" : the variable is shared only during the response processing.
  This prefix is followed by a name. The separator is a '.'. The name may only contain characters 'a-z', 'A-Z', '0-9' and '_'.

neg
  Takes the input value of type signed integer, computes the opposite value, and returns the remainder as an signed integer. 0 is identity. This operator is provided for reversed subtracts : in order to subtract the input from a constant, simply perform a "neg,add(value)".

not
  Returns a boolean FALSE if the input value of type signed integer is non-null, otherwise returns TRUE. Used in conjunction with and(), it can be used to report true/false for bit testing on input values (eg: verify the absence of a flag).

odd
  Returns a boolean TRUE if the input value of type signed integer is odd otherwise returns FALSE. It is functionally equivalent to "and(1),bool".

or(<value>)
  Performs a bitwise "OR" between <value> and the input value of type signed integer, and returns the result as an signed integer. <value> can be a numeric value or a variable name. The name of the variable starts by an indication about its scope. The allowed scopes are:
    "sess" : the variable is shared with all the session,
    "txn" : the variable is shared with all the transaction (request and response),
    "req" : the variable is shared only during the request processing,
    "res" : the variable is shared only during the response processing.

```
  This prefix is followed by a name. The separator is a '.'. The name may only
  contain characters 'a-z', 'A-Z', '0-9' and '_'.

regsub(<regex>,<subst>[,<flags>])
  Applies a regex-based substitution to the input string. It does the same
  operation as the well-known "sed" utility with "s/<regex>/<subst>/". By
  default it will replace in the input string the first occurrence of the
  largest part matching the regular expression <regex> with the substitution
  string <subst>. It is possible to replace all occurrences instead by adding
  the flag "g" in the third argument <flags>. It is also possible to make the
  regex case insensitive by adding the flag "i" in <flags>. Since <flags> is a
  string, it is made up from the concatenation of all desired flags. Thus if
  both "i" and "g" are desired, using "gi" or "ig" will have the same effect.
  It is important to note that due to the current limitations of the
  configuration parser, some characters such as closing parenthesis or comma
  are not possible to use in the arguments. The first use of this converter is
  to replace certain characters or sequence of characters with other ones.

  Example :

     # de-duplicate "/" in header "x-path".
     # input:  x-path: /////a///b/c/xzxyz/
     # output: x-path: /a/b/c/xzxyz/
     http-request set-header x-path %[hdr(x-path),regsub(/+,/,g)]

capture-req(<id>)
  Capture the string entry in the request slot <id> and returns the entry as
  is. If the slot doesn't exist, the capture fails silently.

  See also: "declare capture", "http-request capture",
            "http-response capture", "capture.req.hdr" and
            "capture.res.hdr" (sample fetches).

capture-res(<id>)
  Capture the string entry in the response slot <id> and returns the entry as
  is. If the slot doesn't exist, the capture fails silently.

  See also: "declare capture", "http-request capture",
            "http-response capture", "capture.req.hdr" and
            "capture.res.hdr" (sample fetches).

sdbm([<avalanche>])
  Hashes a binary input sample into an unsigned 32-bit quantity using the SDBM
  hash function. Optionally, it is possible to apply a full avalanche hash
  function to the output if the optional <avalanche> argument equals 1. This
  converter uses the same functions as used by the various hash-based load
  balancing algorithms, so it will provide exactly the same results. It is
  mostly intended for debugging, but can be used as a stick-table entry to
  collect rough statistics. It must not be used for security purposes as a
  32-bit hash is trivial to break. See also "crc32", "djb2", "wt6" and the
  "hash-type" directive.

set-var(<var name>)
  Sets a variable with the input content and return the content on the output as
  is. The variable keep the value and the associated input type. The name of the
  variable starts by an indication about it scope. The scope allowed are:
    "sess" : the variable is shared with all the session,
    "txn"  : the variable is shared with all the transaction (request and
             response),
    "req"  : the variable is shared only during the request processing,
    "res"  : the variable is shared only during the response processing.
  This prefix is followed by a name. The separator is a '.'. The name may only
  contain characters 'a-z', 'A-Z', '0-9' and '_'.

sub(<value>)
```

```
  Subtracts <value> from the input value of type signed integer, and returns
  the result as an signed integer. Note: in order to subtract the input from
  a constant, simply perform a "neg,add(value)". <value> can be a numeric value
  or a variable name. The name of the variable starts by an indication about its
  scope. The allowed scopes are:
    "sess" : the variable is shared with all the session,
    "txn"  : the variable is shared with all the transaction (request and
             response),
    "req"  : the variable is shared only during the request processing,
    "res"  : the variable is shared only during the response processing.
  This prefix is followed by a name. The separator is a '.'. The name may only
  contain characters 'a-z', 'A-Z', '0-9' and '_'.

table_bytes_in_rate(<table>)
  Uses the string representation of the input sample to perform a look up in
  the specified table. If the key is not found in the table, integer value zero
  is returned. Otherwise the converter returns the average client-to-server
  bytes rate associated with the input sample in the designated table, measured
  in amount of bytes over the period configured in the table. See also the
  sc_bytes_in_rate sample fetch keyword.


table_bytes_out_rate(<table>)
  Uses the string representation of the input sample to perform a look up in
  the specified table. If the key is not found in the table, integer value zero
  is returned. Otherwise the converter returns the average server-to-client
  bytes rate associated with the input sample in the designated table, measured
  in amount of bytes over the period configured in the table. See also the
  sc_bytes_out_rate sample fetch keyword.

table_conn_cnt(<table>)
  Uses the string representation of the input sample to perform a look up in
  the specified table. If the key is not found in the table, integer value zero
  is returned. Otherwise the converter returns the cumulated amount of incoming
  connections associated with the input sample in the designated table. See
  also the sc_conn_cnt sample fetch keyword.

table_conn_cur(<table>)
  Uses the string representation of the input sample to perform a look up in
  the specified table. If the key is not found in the table, integer value zero
  is returned. Otherwise the converter returns the current amount of concurrent
  tracked connections associated with the input sample in the designated table.
  See also the sc_conn_cur sample fetch keyword.

table_conn_rate(<table>)
  Uses the string representation of the input sample to perform a look up in
  the specified table. If the key is not found in the table, integer value zero
  is returned. Otherwise the converter returns the average incoming connection
  rate associated with the input sample in the designated table. See also the
  sc_conn_rate sample fetch keyword.

table_gpt0(<table>)
  Uses the string representation of the input sample to perform a look up in
  the specified table. If the key is not found in the table, boolean value zero
  is returned. Otherwise the converter returns the current value of the first
  general purpose tag associated with the input sample in the designated table.
  See also the sc_get_gpt0 sample fetch keyword.

table_gpc0(<table>)
  Uses the string representation of the input sample to perform a look up in
  the specified table. If the key is not found in the table, integer value zero
  is returned. Otherwise the converter returns the current value of the first
  general purpose counter associated with the input sample in the designated
  table. See also the sc_get_gpc0 sample fetch keyword.
```

table_gpc0_rate(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the converter returns the frequency which the gpc0 counter was incremented over the configured period in the table, associated with the input sample in the designated table. See also the sc_get_gpc0_rate sample fetch keyword.

table_http_err_cnt(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the converter returns the cumulated amount of HTTP errors associated with the input sample in the designated table. See also the sc_http_err_cnt sample fetch keyword.

table_http_err_rate(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the average rate of HTTP errors associated with the input sample in the designated table, measured in amount of errors over the period configured in the table. See also the sc_http_err_rate sample fetch keyword.

table_http_req_cnt(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the converter returns the cumulated amount of HTTP requests associated with the input sample in the designated table. See also the sc_http_req_cnt sample fetch keyword.

table_http_req_rate(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the average rate of HTTP requests associated with the input sample in the designated table, measured in amount of requests over the period configured in the table. See also the sc_http_req_rate sample fetch keyword.

table_kbytes_in(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the converter returns the cumulated amount of client-to-server data associated with the input sample in the designated table, measured in kilobytes. The test is currently performed on 32-bit integers, which limits values to 4 terabytes. See also the sc_kbytes_in sample fetch keyword.

table_kbytes_out(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the converter returns the cumulated amount of server-to-client data associated with the input sample in the designated table, measured in kilobytes. The test is currently performed on 32-bit integers, which limits values to 4 terabytes. See also the sc_kbytes_out sample fetch keyword.

table_server_id(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the converter returns the server ID associated with the input sample in the designated table. A server ID is associated to a sample by a "stick" rule when a connection to a server succeeds. A server ID zero means that no server is associated with this key.

table_sess_cnt(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the converter returns the cumulated amount of incoming sessions associated with the input sample in the designated table. Note that a session here refers to an incoming connection being accepted by the "tcp-request connection" rulesets. See also the sc_sess_cnt sample fetch keyword.

table_sess_rate(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the converter returns the average incoming session rate associated with the input sample in the designated table. Note that a session here refers to an incoming connection being accepted by the "tcp-request connection" rulesets. See also the sc_sess_rate sample fetch keyword.

table_trackers(<table>)

Uses the string representation of the input sample to perform a look up in the specified table. If the key is not found in the table, integer value zero is returned. Otherwise the converter returns the current amount of concurrent connections tracking the same key as the input sample in the designated table. It differs from table_conn_cur in that it does not rely on any stored information but on the table's reference count (the "use" value which is returned by "show table" on the CLI). This may sometimes be more suited for layer7 tracking. It can be used to tell a server how many concurrent connections there are from a given address for example. See also the sc_trackers sample fetch keyword.

upper

Convert a string sample to upper case. This can only be placed after a string sample fetch function or after a transformation keyword returning a string type. The result is of type string.

url_dec

Takes an url-encoded string provided as input and returns the decoded version as output. The input and the output are of type string.

utime(<format>[,<offset>])

Converts an integer supposed to contain a date since epoch to a string representing this date in UTC time using a format defined by the <format> string using strftime(3). The purpose is to allow any date format to be used in logs. An optional <offset> in seconds may be applied to the input date (positive or negative). See the strftime() man page for the format supported by your operating system. See also the ltime converter.

Example :

```
    # Emit two colons, one with the UTC time and another with ip:port
    # Eg:  20140710162350 127.0.0.1:57325
    log-format %[date,utime(%Y%m%d%H%M%S)]\ %ci:%cp
```

word(<index>,<delimiters>)

Extracts the nth word considering given delimiters from an input string. Indexes start at 1 and delimiters are a string formatted list of chars.

wt6([<avalanche>])

Hashes a binary input sample into an unsigned 32-bit quantity using the WT6 hash function. Optionally, it is possible to apply a full avalanche hash function to the output if the optional <avalanche> argument equals 1. This converter uses the same functions as used by the various hash-based load balancing algorithms, so it will provide exactly the same results. It is mostly intended for debugging, but can be used as a stick-table entry to collect rough statistics. It must not be used for security purposes as a 32-bit hash is trivial to break. See also "crc32", "djb2", "sdbm", and the "hash-type" directive.

xor(<value>)
  Performs a bitwise "XOR" (exclusive OR) between <value> and the input value of type signed integer, and returns the result as an signed integer. <value> can be a numeric value or a variable name. The name of the variable starts by an indication about its scope. The allowed scopes are:

```
    "sess" : the variable is shared with all the session,
    "txn"  : the variable is shared with all the transaction (request and
             response),
    "req"  : the variable is shared only during the request processing,
    "res"  : the variable is shared only during the response processing.
```

  This prefix is followed by a name. The separator is a '.'. The name may only contain characters 'a-z', 'A-Z', '0-9' and '_'.

7.3.2. Fetching samples from internal states

A first set of sample fetch methods applies to internal information which does not even relate to any client information. These ones are sometimes used with "monitor-fail" directives to report an internal status to external watchers. The sample fetch methods described in this section are usable anywhere.

always_false : boolean
  Always returns the boolean "false" value. It may be used with ACLs as a temporary replacement for another one when adjusting configurations.

always_true : boolean
  Always returns the boolean "true" value. It may be used with ACLs as a temporary replacement for another one when adjusting configurations.

avg_queue([<backend>]) : integer
  Returns the total number of queued connections of the designated backend divided by the number of active servers. The current backend is used if no backend is specified. This is very similar to "queue" except that the size of the farm is considered, in order to give a more accurate measurement of the time it may take for a new connection to be processed. The main usage is with ACL to return a sorry page to new users when it becomes certain they will get a degraded service, or to pass to the backend servers in a header so that they decide to work in degraded mode or to disable some functions to speed up the processing a bit. Note that in the event there would not be any active server anymore, twice the number of queued connections would be considered as the measured value. This is a fair estimate, as we expect one server to get back soon anyway, but we still prefer to send new traffic to another backend if in better shape. See also the "queue", "be_conn", and "be_sess_rate" sample fetches.

be_conn([<backend>]) : integer
  Applies to the number of currently established connections on the backend, possibly including the connection being evaluated. If no backend name is specified, the current one is used. But it is also possible to check another backend. It can be used to use a specific farm when the nominal one is full. See also the "fe_conn", "queue" and "be_sess_rate" criteria.

be_sess_rate([<backend>]) : integer
  Returns an integer value corresponding to the sessions creation rate on the backend, in number of new sessions per second. This is used with ACLs to switch to an alternate backend when an expensive or fragile one reaches too high a session rate, or to limit abuse of service (eg. prevent sucking of an online dictionary). It can also be useful to add this element to logs using a log-format directive.

  Example :

```
        # Redirect to an error page if the dictionary is requested too often
        backend dynamic
            mode http
            acl being_scanned be_sess_rate gt 100
            redirect location /denied.html if being_scanned
```

bin(<hexa>) : bin
  Returns a binary chain. The input is the hexadecimal representation of the string.

bool(<bool>) : bool
  Returns a boolean value. <bool> can be 'true', 'false', '1' or '0'. 'false' and '0' are the same. 'true' and '1' are the same.

connslots([<backend>]) : integer
  Returns an integer value corresponding to the number of connection slots still available in the backend, by totaling the maximum amount of connections on all servers and the maximum queue size. This is probably only used with ACLs.

  The basic idea here is to be able to measure the number of connection "slots" still available (connection + queue), so that anything beyond that (intended usage; see "use_backend" keyword) can be redirected to a different backend.

  'connslots' = number of available server connection slots, + number of available server queue slots.

  Note that while "fe_conn" may be used, "connslots" comes in especially useful when you have a case of traffic going to one single ip, splitting into multiple backends (perhaps using ACLs to do name-based load balancing) and you want to be able to differentiate between different backends, and their available "connslots". Also, whereas "nbsrv" only measures servers that are actually *down*, this fetch is more fine-grained and looks into the number of available connection slots as well. See also "queue" and "avg_queue".

  OTHER CAVEATS AND NOTES: at this point in time, the code does not take care of dynamic connections. Also, if any of the server maxconn, or maxqueue is 0, then this fetch clearly does not make sense, in which case the value returned will be -1.

date([<offset>]) : integer
  Returns the current date as the epoch (number of seconds since 01/01/1970). If an offset value is specified, then it is a number of seconds that is added to the current date before returning the value. This is particularly useful to compute relative dates, as both positive and negative offsets are allowed. It is useful combined with the http_date converter.

  Example :

```
     # set an expires header to now+1 hour in every response
     http-response set-header Expires %[date(3600),http_date]
```

env(<name>) : string
  Returns a string containing the value of environment variable <name>. As a reminder, environment variables are per-process and are sampled when the process starts. This can be useful to pass some information to a next hop server, or with ACLs to take specific action when the process is started a certain way.

  Examples :

```
      # Pass the Via header to next hop with the local hostname in it
      http-request add-header Via 1.1\ %[env(HOSTNAME)]

      # reject cookie-less requests when the STOP environment variable is set
      http-request deny if !{ cook(SESSIONID) -m found } { env(STOP) -m found }
```

```
fe_conn([<frontend>]) : integer
  Returns the number of currently established connections on the frontend,
  possibly including the connection being evaluated. If no frontend name is
  specified, the current one is used. But it is also possible to check another
  frontend. It can be used to return a sorry page before hard-blocking, or to
  use a specific backend to drain new requests when the farm is considered
  full.  This is mostly used with ACLs but can also be used to pass some
  statistics to servers in HTTP headers. See also the "dst_conn", "be_conn",
  "fe_sess_rate" fetches.

fe_sess_rate([<frontend>]) : integer
  Returns an integer value corresponding to the sessions creation rate on the
  frontend, in number of new sessions per second. This is used with ACLs to
  limit the incoming session rate to an acceptable range in order to prevent
  abuse of service at the earliest moment, for example when combined with other
  layer 4 ACLs in order to force the clients to wait a bit for the rate to go
  down below the limit. It can also be useful to add this element to logs using
  a log-format directive. See also the "rate-limit sessions" directive for use
  in frontends.

  Example :
        # This frontend limits incoming mails to 10/s with a max of 100
        # concurrent connections. We accept any connection below 10/s, and
        # force excess clients to wait for 100 ms. Since clients are limited to
        # 100 max, there cannot be more than 10 incoming mails per second.
        frontend mail
            bind :25
            mode tcp
            maxconn 100
            acl too_fast fe_sess_rate ge 10
            tcp-request inspect-delay 100ms
            tcp-request content accept if ! too_fast
            tcp-request content accept if WAIT_END

int(<integer>) : signed integer
  Returns a signed integer.

ipv4(<ipv4>) : ipv4
  Returns an ipv4.

ipv6(<ipv6>) : ipv6
  Returns an ipv6.

meth(<method>) : method
  Returns a method.

nbproc : integer
  Returns an integer value corresponding to the number of processes that were
  started (it equals the global "nbproc" setting). This is useful for logging
  and debugging purposes.

nbsrv([<backend>]) : integer
  Returns an integer value corresponding to the number of usable servers of
  either the current backend or the named backend. This is mostly used with
  ACLs but can also be useful when added to logs. This is normally used to
  switch to an alternate backend when the number of servers is too low to
  to handle some load. It is useful to report a failure when combined with
  "monitor fail".

proc : integer
  Returns an integer value corresponding to the position of the process calling
  the function, between 1 and global.nbproc. This is useful for logging and
  debugging purposes.

queue([<backend>]) : integer
```

```
  Returns the total number of queued connections of the designated backend,
  including all the connections in server queues. If no backend name is
  specified, the current one is used, but it is also possible to check another
  one. This is useful with ACLs or to pass statistics to backend servers. This
  can be used to take actions when queuing goes above a known level, generally
  indicating a surge of traffic or a massive slowdown on the servers. One
  possible action could be to reject new users but still accept old ones. See
  also the "avg_queue", "be_conn", and "be_sess_rate" fetches.

rand([<range>]) : integer
  Returns a random integer value within a range of <range> possible values,
  starting at zero. If the range is not specified, it defaults to 2^32, which
  gives numbers between 0 and 4294967295. It can be useful to pass some values
  needed to take some routing decisions for example, or just for debugging
  purposes. This random must not be used for security purposes.

srv_conn([<backend>/]<server>) : integer
  Returns an integer value corresponding to the number of currently established
  connections on the designated server, possibly including the connection being
  evaluated. If <backend> is omitted, then the server is looked up in the
  current backend. It can be used to use a specific farm when one server is
  full, or to inform the server about our view of the number of active
  connections with it. See also the "fe_conn", "be_conn" and "queue" fetch
  methods.

srv_is_up([<backend>/]<server>) : boolean
  Returns true when the designated server is UP, and false when it is either
  DOWN or in maintenance mode. If <backend> is omitted, then the server is
  looked up in the current backend. It is mainly used to take action based on
  an external status reported via a health check (eg: a geographical site's
  availability). Another possible use which is more of a hack consists in
  using dummy servers as boolean variables that can be enabled or disabled from
  the CLI, so that rules depending on those ACLs can be tweaked in realtime.

srv_sess_rate([<backend>/]<server>) : integer
  Returns an integer corresponding to the sessions creation rate on the
  designated server, in number of new sessions per second. If <backend> is
  omitted, then the server is looked up in the current backend. This is mostly
  used with ACLs but can make sense with logs too. This is used to switch to an
  alternate backend when an expensive or fragile one reaches too high a session
  rate, or to limit abuse of service (eg. prevent latent requests from
  overloading servers).

  Example :
        # Redirect to a separate back
        acl srv1_full srv_sess_rate(be1/srv1) gt 50
        acl srv2_full srv_sess_rate(be1/srv2) gt 50
        use_backend be2 if srv1_full or srv2_full

stopping : boolean
  Returns TRUE if the process calling the function is currently stopping. This
  can be useful for logging, or for relaxing certain checks or helping close
  certain connections upon graceful shutdown.

str(<string>) : string
  Returns a string.

table_avl([<table>]) : integer
  Returns the total number of available entries in the current proxy's
  stick-table or in the designated stick-table. See also table_cnt.

table_cnt([<table>]) : integer
  Returns the total number of entries currently in use in the current proxy's
  stick-table or in the designated stick-table. See also src_conn_cnt and
  table_avl for other entry counting methods.
```

var(<var-name>) : undefined
  Returns a variable with the stored type. If the variable is not set, the sample fetch fails. The name of the variable starts by an indication about its scope. The scope allowed are:

```
    "sess" : the variable is shared with all the session,
    "txn"  : the variable is shared with all the transaction (request and
             response),
    "req"  : the variable is shared only during the request processing,
    "res"  : the variable is shared only during the response processing.
```

  This prefix is followed by a name. The separator is a '.'. The name may only contain characters 'a-z', 'A-Z', '0-9' and '_'.

7.3.3. Fetching samples at Layer 4

The layer 4 usually describes just the transport layer which in haproxy is closest to the connection, where no content is yet made available. The fetch methods described here are usable as low as the "tcp-request connection" rule sets unless they require some future information. Those generally include TCP/IP addresses and ports, as well as elements from stick-tables related to the incoming connection. For retrieving a value from a sticky counters, the counter number can be explicitly set as 0, 1, or 2 using the pre-defined "sc0_", "sc1_", or "sc2_" prefix, or it can be specified as the first integer argument when using the "sc_" prefix. An optional table may be specified with the "sc*" form, in which case the currently tracked key will be looked up into this alternate table instead of the table currently being tracked.

be_id : integer
  Returns an integer containing the current backend's id. It can be used in frontends with responses to check which backend processed the request.

dst : ip
  This is the destination IPv4 address of the connection on the client side, which is the address the client connected to. It can be useful when running in transparent mode. It is of type IP and works on both IPv4 and IPv6 tables. On IPv6 tables, IPv4 address is mapped to its IPv6 equivalent, according to RFC 4291.

dst_conn : integer
  Returns an integer value corresponding to the number of currently established connections on the same socket including the one being evaluated. It is normally used with ACLs but can as well be used to pass the information to servers in an HTTP header or in logs. It can be used to either return a sorry page before hard-blocking, or to use a specific backend to drain new requests when the socket is considered saturated. This offers the ability to assign different limits to different listening ports or addresses. See also the "fe_conn" and "be_conn" fetches.

dst_port : integer
  Returns an integer value corresponding to the destination TCP port of the connection on the client side, which is the port the client connected to. This might be used when running in transparent mode, when assigning dynamic ports to some clients for a whole application session, to stick all users to a same server, or to pass the destination port information to a server using an HTTP header.

fe_id : integer
  Returns an integer containing the current frontend's id. It can be used in backends to check from which backend it was called, or to stick all users coming via a same frontend to the same server.

sc_bytes_in_rate(<ctr>[,<table>]) : integer
sc0_bytes_in_rate([<table>]) : integer

sc1_bytes_in_rate([<table>]) : integer
sc2_bytes_in_rate([<table>]) : integer
  Returns the average client-to-server bytes rate from the currently tracked counters, measured in amount of bytes over the period configured in the table. See also src_bytes_in_rate.

sc_bytes_out_rate(<ctr>[,<table>]) : integer
sc0_bytes_out_rate([<table>]) : integer
sc1_bytes_out_rate([<table>]) : integer
sc2_bytes_out_rate([<table>]) : integer
  Returns the average server-to-client bytes rate from the currently tracked counters, measured in amount of bytes over the period configured in the table. See also src_bytes_out_rate.

sc_clr_gpc0(<ctr>[,<table>]) : integer
sc0_clr_gpc0([<table>]) : integer
sc1_clr_gpc0([<table>]) : integer
sc2_clr_gpc0([<table>]) : integer
  Clears the first General Purpose Counter associated to the currently tracked counters, and returns its previous value. Before the first invocation, the stored value is zero, so first invocation will always return zero. This is typically used as a second ACL in an expression in order to mark a connection when a first ACL was verified :

```
        # block if 5 consecutive requests continue to come faster than 10 sess
        # per second, and reset the counter as soon as the traffic slows down.
        acl abuse sc0_http_req_rate gt 10
        acl kill  sc0_inc_gpc0 gt 5
        acl save  sc0_clr_gpc0 ge 0
        tcp-request connection accept if !abuse save
        tcp-request connection reject if abuse kill
```

sc_conn_cnt(<ctr>[,<table>]) : integer
sc0_conn_cnt([<table>]) : integer
sc1_conn_cnt([<table>]) : integer
sc2_conn_cnt([<table>]) : integer
  Returns the cumulated number of incoming connections from currently tracked counters. See also src_conn_cnt.

sc_conn_cur(<ctr>[,<table>]) : integer
sc0_conn_cur([<table>]) : integer
sc1_conn_cur([<table>]) : integer
sc2_conn_cur([<table>]) : integer
  Returns the current amount of concurrent connections tracking the same tracked counters. This number is automatically incremented when tracking begins and decremented when tracking stops. See also src_conn_cur.

sc_conn_rate(<ctr>[,<table>]) : integer
sc0_conn_rate([<table>]) : integer
sc1_conn_rate([<table>]) : integer
sc2_conn_rate([<table>]) : integer
  Returns the average connection rate from the currently tracked counters, measured in amount of connections over the period configured in the table. See also src_conn_rate.

sc_get_gpc0(<ctr>[,<table>]) : integer
sc0_get_gpc0([<table>]) : integer
sc1_get_gpc0([<table>]) : integer
sc2_get_gpc0([<table>]) : integer
  Returns the value of the first General Purpose Counter associated to the currently tracked counters. See also src_get_gpc0 and sc/sc0/sc1/sc2_inc_gpc0.

sc_get_gpt0(<ctr>[,<table>]) : integer
sc0_get_gpt0([<table>]) : integer
sc1_get_gpt0([<table>]) : integer

```
sc2_get_gpt0([<table>]) : integer
  Returns the value of the first General Purpose Tag associated to the
  currently tracked counters. See also src_get_gpt0.

sc_gpc0_rate(<ctr>[,<table>]) : integer
sc0_gpc0_rate([<table>]) : integer
sc1_gpc0_rate([<table>]) : integer
sc2_gpc0_rate([<table>]) : integer
  Returns the average increment rate of the first General Purpose Counter
  associated to the currently tracked counters. It reports the frequency
  which the gpc0 counter was incremented over the configured period. See also
  src_gpc0_rate, sc/sc0/sc1/sc2_get_gpc0, and sc/sc0/sc1/sc2_inc_gpc0. Note
  that the "gpc0_rate" counter must be stored in the stick-table for a value to
  be returned, as "gpc0" only holds the event count.

sc_http_err_cnt(<ctr>[,<table>]) : integer
sc0_http_err_cnt([<table>]) : integer
sc1_http_err_cnt([<table>]) : integer
sc2_http_err_cnt([<table>]) : integer
  Returns the cumulated number of HTTP errors from the currently tracked
  counters. This includes the both request errors and 4xx error responses.
  See also src_http_err_cnt.

sc_http_err_rate(<ctr>[,<table>]) : integer
sc0_http_err_rate([<table>]) : integer
sc1_http_err_rate([<table>]) : integer
sc2_http_err_rate([<table>]) : integer
  Returns the average rate of HTTP errors from the currently tracked counters,
  measured in amount of errors over the period configured in the table. This
  includes the both request errors and 4xx error responses. See also
  src_http_err_rate.

sc_http_req_cnt(<ctr>[,<table>]) : integer
sc0_http_req_cnt([<table>]) : integer
sc1_http_req_cnt([<table>]) : integer
sc2_http_req_cnt([<table>]) : integer
  Returns the cumulated number of HTTP requests from the currently tracked
  counters. This includes every started request, valid or not. See also
  src_http_req_cnt.

sc_http_req_rate(<ctr>[,<table>]) : integer
sc0_http_req_rate([<table>]) : integer
sc1_http_req_rate([<table>]) : integer
sc2_http_req_rate([<table>]) : integer
  Returns the average rate of HTTP requests from the currently tracked
  counters, measured in amount of requests over the period configured in
  the table. This includes every started request, valid or not. See also
  src_http_req_rate.

sc_inc_gpc0(<ctr>[,<table>]) : integer
sc0_inc_gpc0([<table>]) : integer
sc1_inc_gpc0([<table>]) : integer
sc2_inc_gpc0([<table>]) : integer
  Increments the first General Purpose Counter associated to the currently
  tracked counters, and returns its new value. Before the first invocation,
  the stored value is zero, so first invocation will increase it to 1 and will
  return 1. This is typically used as a second ACL in an expression in order
  to mark a connection when a first ACL was verified :

        acl abuse sc0_http_req_rate gt 10
        acl kill  sc0_inc_gpc0 gt 0
        tcp-request connection reject if abuse kill

sc_kbytes_in(<ctr>[,<table>]) : integer
sc0_kbytes_in([<table>]) : integer
```

```
sc1_kbytes_in([<table>]) : integer
sc2_kbytes_in([<table>]) : integer
  Returns the total amount of client-to-server data from the currently tracked
  counters, measured in kilobytes. The test is currently performed on 32-bit
  integers, which limits values to 4 terabytes. See also src_kbytes_in.

sc_kbytes_out(<ctr>[,<table>]) : integer
sc0_kbytes_out([<table>]) : integer
sc1_kbytes_out([<table>]) : integer
sc2_kbytes_out([<table>]) : integer
  Returns the total amount of server-to-client data from the currently tracked
  counters, measured in kilobytes. The test is currently performed on 32-bit
  integers, which limits values to 4 terabytes. See also src_kbytes_out.

sc_sess_cnt(<ctr>[,<table>]) : integer
sc0_sess_cnt([<table>]) : integer
sc1_sess_cnt([<table>]) : integer
sc2_sess_cnt([<table>]) : integer
  Returns the cumulated number of incoming connections that were transformed
  into sessions, which means that they were accepted by a "tcp-request
  connection" rule, from the currently tracked counters. A backend may count
  more sessions than connections because each connection could result in many
  backend sessions if some HTTP keep-alive is performed over the connection
  with the client. See also src_sess_cnt.

sc_sess_rate(<ctr>[,<table>]) : integer
sc0_sess_rate([<table>]) : integer
sc1_sess_rate([<table>]) : integer
sc2_sess_rate([<table>]) : integer
  Returns the average session rate from the currently tracked counters,
  measured in amount of sessions over the period configured in the table. A
  session is a connection that got past the early "tcp-request connection"
  rules. A backend may count more sessions than connections because each
  connection could result in many backend sessions if some HTTP keep-alive is
  performed over the connection with the client. See also src_sess_rate.

sc_tracked(<ctr>[,<table>]) : boolean
sc0_tracked([<table>]) : boolean
sc1_tracked([<table>]) : boolean
sc2_tracked([<table>]) : boolean
  Returns true if the designated session counter is currently being tracked by
  the current session. This can be useful when deciding whether or not we want
  to set some values in a header passed to the server.

sc_trackers(<ctr>[,<table>]) : integer
sc0_trackers([<table>]) : integer
sc1_trackers([<table>]) : integer
sc2_trackers([<table>]) : integer
  Returns the current amount of concurrent connections tracking the same
  tracked counters. This number is automatically incremented when tracking
  begins and decremented when tracking stops. It differs from sc0_conn_cur in
  that it does not rely on any stored information but on the table's reference
  count (the "use" value which is returned by "show table" on the CLI). This
  may sometimes be more suited for layer7 tracking. It can be used to tell a
  server how many concurrent connections there are from a given address for
  example.

so_id : integer
  Returns an integer containing the current listening socket's id. It is useful
  in frontends involving many "bind" lines, or to stick all users coming via a
  same socket to the same server.

src : ip
  This is the source IPv4 address of the client of the session.  It is of type
  IP and works on both IPv4 and IPv6 tables. On IPv6 tables, IPv4 addresses are
  mapped to their IPv6 equivalent, according to RFC 4291. Note that it is the
  TCP-level source address which is used, and not the address of a client
  behind a proxy. However if the "accept-proxy" bind directive is used, it can
  be the address of a client behind another PROXY-protocol compatible component
  for all rule sets except "tcp-request connection" which sees the real address.

  Example:
       # add an HTTP header in requests with the originating address' country
       http-request set-header X-Country %[src,map_ip(geoip.lst)]

src_bytes_in_rate([<table>]) : integer
  Returns the average bytes rate from the incoming connection's source address
  in the current proxy's stick-table or in the designated stick-table, measured
  in amount of bytes over the period configured in the table. If the address is
  not found, zero is returned. See also sc/sc0/sc1/sc2_bytes_in_rate.

src_bytes_out_rate([<table>]) : integer
  Returns the average bytes rate to the incoming connection's source address in
  the current proxy's stick-table or in the designated stick-table, measured in
  amount of bytes over the period configured in the table. If the address is
  not found, zero is returned. See also sc/sc0/sc1/sc2_bytes_out_rate.

src_clr_gpc0([<table>]) : integer
  Clears the first General Purpose Counter associated to the incoming
  connection's source address in the current proxy's stick-table or in the
  designated stick-table, and returns its previous value. If the address is not
  found, an entry is created and 0 is returned. This is typically used as a
  second ACL in an expression in order to mark a connection when a first ACL
  was verified :

        # block if 5 consecutive requests continue to come faster than 10 sess
        # per second, and reset the counter as soon as the traffic slows down.
        acl abuse src_http_req_rate gt 10
        acl kill  src_inc_gpc0 gt 5
        acl save  src_clr_gpc0 ge 0
        tcp-request connection accept if !abuse save
        tcp-request connection reject if abuse kill

src_conn_cnt([<table>]) : integer
  Returns the cumulated number of connections initiated from the current
  incoming connection's source address in the current proxy's stick-table or in
  the designated stick-table. If the address is not found, zero is returned.
  See also sc/sc0/sc1/sc2_conn_cnt.

src_conn_cur([<table>]) : integer
  Returns the current amount of concurrent connections initiated from the
  current incoming connection's source address in the current proxy's
  stick-table or in the designated stick-table. If the address is not found,
  zero is returned. See also sc/sc0/sc1/sc2_conn_cur.

src_conn_rate([<table>]) : integer
  Returns the average connection rate from the incoming connection's source
  address in the current proxy's stick-table or in the designated stick-table,
  measured in amount of connections over the period configured in the table. If
  the address is not found, zero is returned. See also sc/sc0/sc1/sc2_conn_rate.

src_get_gpc0([<table>]) : integer
  Returns the value of the first General Purpose Counter associated to the
  incoming connection's source address in the current proxy's stick-table or in
  the designated stick-table. If the address is not found, zero is returned.
  See also sc/sc0/sc1/sc2_get_gpc0 and src_inc_gpc0.

src_get_gpt0([<table>]) : integer
  Returns the value of the first General Purpose Tag associated to the
  incoming connection's source address in the current proxy's stick-table or in
```

```
  the designated stick-table. If the address is not found, zero is returned.
  See also sc/sc0/sc1/sc2_get_gpt0.

src_gpc0_rate([<table>]) : integer
  Returns the average increment rate of the first General Purpose Counter
  associated to the incoming connection's source address in the current proxy's
  stick-table or in the designated stick-table. It reports the frequency
  which the gpc0 counter was incremented over the configured period. See also
  sc/sc0/sc1/sc2_gpc0_rate, src_get_gpc0, and sc/sc0/sc1/sc2_inc_gpc0. Note
  that the "gpc0_rate" counter must be stored in the stick-table for a value to
  be returned, as "gpc0" only holds the event count.

src_http_err_cnt([<table>]) : integer
  Returns the cumulated number of HTTP errors from the incoming connection's
  source address in the current proxy's stick-table or in the designated
  stick-table. This includes the both request errors and 4xx error responses.
  See also sc/sc0/sc1/sc2_http_err_cnt. If the address is not found, zero is
  returned.

src_http_err_rate([<table>]) : integer
  Returns the average rate of HTTP errors from the incoming connection's source
  address in the current proxy's stick-table or in the designated stick-table,
  measured in amount of errors over the period configured in the table. This
  includes the both request errors and 4xx error responses. If the address is
  not found, zero is returned. See also sc/sc0/sc1/sc2_http_err_rate.

src_http_req_cnt([<table>]) : integer
  Returns the cumulated number of HTTP requests from the incoming connection's
  source address in the current proxy's stick-table or in the designated stick-
  table. This includes every started request, valid or not. If the address is
  not found, zero is returned. See also sc/sc0/sc1/sc2_http_req_cnt.

src_http_req_rate([<table>]) : integer
  Returns the average rate of HTTP requests from the incoming connection's
  source address in the current proxy's stick-table or in the designated stick-
  table, measured in amount of requests over the period configured in the
  table. This includes every started request, valid or not. If the address is
  not found, zero is returned. See also sc/sc0/sc1/sc2_http_req_rate.

src_inc_gpc0([<table>]) : integer
  Increments the first General Purpose Counter associated to the incoming
  connection's source address in the current proxy's stick-table or in the
  designated stick-table, and returns its new value. If the address is not
  found, an entry is created and 1 is returned. See also sc0/sc2/sc2_inc_gpc0.
  This is typically used as a second ACL in an expression in order to mark a
  connection when a first ACL was verified :

        acl abuse src_http_req_rate gt 10
        acl kill  src_inc_gpc0 gt 0
        tcp-request connection reject if abuse kill

src_kbytes_in([<table>]) : integer
  Returns the total amount of data received from the incoming connection's
  source address in the current proxy's stick-table or in the designated
  stick-table, measured in kilobytes. If the address is not found, zero is
  returned. The test is currently performed on 32-bit integers, which limits
  values to 4 terabytes. See also sc/sc0/sc1/sc2_kbytes_in.

src_kbytes_out([<table>]) : integer
  Returns the total amount of data sent to the incoming connection's source
  address in the current proxy's stick-table or in the designated stick-table,
  measured in kilobytes. If the address is not found, zero is returned. The
  test is currently performed on 32-bit integers, which limits values to 4
  terabytes. See also sc/sc0/sc1/sc2_kbytes_out.
```

src_port : integer
  Returns an integer value corresponding to the TCP source port of the connection on the client side, which is the port the client connected from. Usage of this function is very limited as modern protocols do not care much about source ports nowadays.

src_sess_cnt([<table>]) : integer
  Returns the cumulated number of connections initiated from the incoming connection's source IPv4 address in the current proxy's stick-table or in the designated stick-table, that were transformed into sessions, which means that they were accepted by "tcp-request" rules. If the address is not found, zero is returned. See also sc/sc0/sc1/sc2_sess_cnt.

src_sess_rate([<table>]) : integer
  Returns the average session rate from the incoming connection's source address in the current proxy's stick-table or in the designated stick-table, measured in amount of sessions over the period configured in the table. A session is a connection that went past the early "tcp-request" rules. If the address is not found, zero is returned. See also sc/sc0/sc1/sc2_sess_rate.

src_updt_conn_cnt([<table>]) : integer
  Creates or updates the entry associated to the incoming connection's source address in the current proxy's stick-table or in the designated stick-table. This table must be configured to store the "conn_cnt" data type, otherwise the match will be ignored. The current count is incremented by one, and the expiration timer refreshed. The updated count is returned, so this match can't return zero. This was used to reject service abusers based on their source address. Note: it is recommended to use the more complete "track-sc*" actions in "tcp-request" rules instead.

  Example :

```
        # This frontend limits incoming SSH connections to 3 per 10 second for
        # each source address, and rejects excess connections until a 10 second
        # silence is observed. At most 20 addresses are tracked.
        listen ssh
            bind :22
            mode tcp
            maxconn 100
            stick-table type ip size 20 expire 10s store conn_cnt
            tcp-request content reject if { src_updt_conn_cnt gt 3 }
            server local 127.0.0.1:22
```

srv_id : integer
  Returns an integer containing the server's id when processing the response. While it's almost only used with ACLs, it may be used for logging or debugging.

7.3.4. Fetching samples at Layer 5

The layer 5 usually describes just the session layer which in haproxy is closest to the session once all the connection handshakes are finished, but when no content is yet made available. The fetch methods described here are usable as low as the "tcp-request content" rule sets unless they require some future information. Those generally include the results of SSL negotiations.

ssl_bc : boolean
  Returns true when the back connection was made via an SSL/TLS transport layer and is locally deciphered. This means the outgoing connection was made other a server with the "ssl" option.

ssl_bc_alg_keysize : integer
  Returns the symmetric cipher key size supported in bits when the outgoing connection was made over an SSL/TLS transport layer.

ssl_bc_cipher : string
  Returns the name of the used cipher when the outgoing connection was made over an SSL/TLS transport layer.

ssl_bc_protocol : string
  Returns the name of the used protocol when the outgoing connection was made over an SSL/TLS transport layer.

ssl_bc_unique_id : binary
  When the outgoing connection was made over an SSL/TLS transport layer, returns the TLS unique ID as defined in RFC5929 section 3. The unique id can be encoded to base64 using the converter: "ssl_bc_unique_id,base64".

ssl_bc_session_id : binary
  Returns the SSL ID of the back connection when the outgoing connection was made over an SSL/TLS transport layer. It is useful to log if we want to know if session was reused or not.

ssl_bc_use_keysize : integer
  Returns the symmetric cipher key size used in bits when the outgoing connection was made over an SSL/TLS transport layer.

ssl_c_ca_err : integer
  When the incoming connection was made over an SSL/TLS transport layer, returns the ID of the first error detected during verification of the client certificate at depth > 0, or 0 if no error was encountered during this verification process. Please refer to your SSL library's documentation to find the exhaustive list of error codes.

ssl_c_ca_err_depth : integer
  When the incoming connection was made over an SSL/TLS transport layer, returns the depth in the CA chain of the first error detected during the verification of the client certificate. If no error is encountered, 0 is returned.

ssl_c_der : binary
  Returns the DER formatted certificate presented by the client when the incoming connection was made over an SSL/TLS transport layer. When used for an ACL, the value(s) to match against can be passed in hexadecimal form.

ssl_c_err : integer
  When the incoming connection was made over an SSL/TLS transport layer, returns the ID of the first error detected during verification at depth 0, or 0 if no error was encountered during this verification process. Please refer to your SSL library's documentation to find the exhaustive list of error codes.

ssl_c_i_dn([<entry>[,<occ>]]) : string
  When the incoming connection was made over an SSL/TLS transport layer, returns the full distinguished name of the issuer of the certificate presented by the client when no <entry> is specified, or the value of the first given entry found from the beginning of the DN. If a positive/negative occurrence number is specified as the optional second argument, it returns the value of the nth given entry value from the beginning/end of the DN. For instance, "ssl_c_i_dn(OU,2)" the second organization unit, and "ssl_c_i_dn(CN)" retrieves the common name.

ssl_c_key_alg : string
  Returns the name of the algorithm used to generate the key of the certificate presented by the client when the incoming connection was made over an SSL/TLS transport layer.

ssl_c_notafter : string
  Returns the end date presented by the client as a formatted string

YYMMDDhhmmss[Z] when the incoming connection was made over an SSL/TLS transport layer.

ssl_c_notbefore : string
Returns the start date presented by the client as a formatted string YYMMDDhhmmss[Z] when the incoming connection was made over an SSL/TLS transport layer.

ssl_c_s_dn([<entry>[,<occ>]]) : string
When the incoming connection was made over an SSL/TLS transport layer, returns the full distinguished name of the subject of the certificate presented by the client when no <entry> is specified, or the value of the first given entry found from the beginning of the DN. If a positive/negative occurrence number is specified as the optional second argument, it returns the value of the nth given entry value from the beginning/end of the DN. For instance, "ssl_c_s_dn(OU,2)" the second organization unit, and "ssl_c_s_dn(CN)" retrieves the common name.

ssl_c_serial : binary
Returns the serial of the certificate presented by the client when the incoming connection was made over an SSL/TLS transport layer. When used for an ACL, the value(s) to match against can be passed in hexadecimal form.

ssl_c_sha1 : binary
Returns the SHA-1 fingerprint of the certificate presented by the client when the incoming connection was made over an SSL/TLS transport layer. This can be used to stick a client to a server, or to pass this information to a server. Note that the output is binary, so if you want to pass that signature to the server, you need to encode it in hex or base64, such as in the example below:

```
    http-request set-header X-SSL-Client-SHA1 %[ssl_c_sha1,hex]
```

ssl_c_sig_alg : string
Returns the name of the algorithm used to sign the certificate presented by the client when the incoming connection was made over an SSL/TLS transport layer.

ssl_c_used : boolean
Returns true if current SSL session uses a client certificate even if current connection uses SSL session resumption. See also "ssl_fc_has_crt".

ssl_c_verify : integer
Returns the verify result error ID when the incoming connection was made over an SSL/TLS transport layer, otherwise zero if no error is encountered. Please refer to your SSL library's documentation for an exhaustive list of error codes.

ssl_c_version : integer
Returns the version of the certificate presented by the client when the incoming connection was made over an SSL/TLS transport layer.

ssl_f_der : binary
Returns the DER formatted certificate presented by the frontend when the incoming connection was made over an SSL/TLS transport layer. When used for an ACL, the value(s) to match against can be passed in hexadecimal form.

ssl_f_i_dn([<entry>[,<occ>]]) : string
When the incoming connection was made over an SSL/TLS transport layer, returns the full distinguished name of the issuer of the certificate presented by the frontend when no <entry> is specified, or the value of the first given entry found from the beginning of the DN. If a positive/negative occurrence number is specified as the optional second argument, it returns the value of the nth given entry value from the beginning/end of the DN. For instance, "ssl_f_i_dn(OU,2)" the second organization unit, and "ssl_f_i_dn(CN)" retrieves the common name.

ssl_f_key_alg : string
Returns the name of the algorithm used to generate the key of the certificate presented by the frontend when the incoming connection was made over an SSL/TLS transport layer.

ssl_f_notafter : string
Returns the end date presented by the frontend as a formatted string YYMMDDhhmmss[Z] when the incoming connection was made over an SSL/TLS transport layer.

ssl_f_notbefore : string
Returns the start date presented by the frontend as a formatted string YYMMDDhhmmss[Z] when the incoming connection was made over an SSL/TLS transport layer.

ssl_f_s_dn([<entry>[,<occ>]]) : string
When the incoming connection was made over an SSL/TLS transport layer, returns the full distinguished name of the subject of the certificate presented by the frontend when no <entry> is specified, or the value of the first given entry found from the beginning of the DN. If a positive/negative occurrence number is specified as the optional second argument, it returns the value of the nth given entry value from the beginning/end of the DN. For instance, "ssl_f_s_dn(OU,2)" the second organization unit, and "ssl_f_s_dn(CN)" retrieves the common name.

ssl_f_serial : binary
Returns the serial of the certificate presented by the frontend when the incoming connection was made over an SSL/TLS transport layer. When used for an ACL, the value(s) to match against can be passed in hexadecimal form.

ssl_f_sha1 : binary
Returns the SHA-1 fingerprint of the certificate presented by the frontend when the incoming connection was made over an SSL/TLS transport layer. This can be used to know which certificate was chosen using SNI.

ssl_f_sig_alg : string
Returns the name of the algorithm used to sign the certificate presented by the frontend when the incoming connection was made over an SSL/TLS transport layer.

ssl_f_version : integer
Returns the version of the certificate presented by the frontend when the incoming connection was made over an SSL/TLS transport layer.

ssl_fc : boolean
Returns true when the front connection was made via an SSL/TLS transport layer and is locally deciphered. This means it has matched a socket declared with a "bind" line having the "ssl" option.

Example :

```
      # This passes "X-Proto: https" to servers when client connects over SSL
      listen http-https
          bind :80
          bind :443 ssl crt /etc/haproxy.pem
          http-request add-header X-Proto https if { ssl_fc }
```

ssl_fc_alg_keysize : integer
Returns the symmetric cipher key size supported in bits when the incoming connection was made over an SSL/TLS transport layer.

ssl_fc_alpn : string
This extracts the Application Layer Protocol Negotiation field from an incoming connection made via a TLS transport layer and locally deciphered by haproxy. The result is a string containing the protocol name advertised by

```
  the client. The SSL library must have been built with support for TLS
  extensions enabled (check haproxy -vv). Note that the TLS ALPN extension is
  not advertised unless the "alpn" keyword on the "bind" line specifies a
  protocol list. Also, nothing forces the client to pick a protocol from this
  list, any other one may be requested. The TLS ALPN extension is meant to
  replace the TLS NPN extension. See also "ssl_fc_npn".

ssl_fc_cipher : string
  Returns the name of the used cipher when the incoming connection was made
  over an SSL/TLS transport layer.

ssl_fc_has_crt : boolean
  Returns true if a client certificate is present in an incoming connection over
  SSL/TLS transport layer. Useful if 'verify' statement is set to 'optional'.
  Note: on SSL session resumption with Session ID or TLS ticket, client
  certificate is not present in the current connection but may be retrieved
  from the cache or the ticket. So prefer "ssl_c_used" if you want to check if
  current SSL session uses a client certificate.

ssl_fc_has_sni : boolean
  This checks for the presence of a Server Name Indication TLS extension (SNI)
  in an incoming connection was made over an SSL/TLS transport layer. Returns
  true when the incoming connection presents a TLS SNI field. This requires
  that the SSL library is build with support for TLS extensions enabled (check
  haproxy -vv).

ssl_fc_is_resumed: boolean
  Returns true if the SSL/TLS session has been resumed through the use of
  SSL session cache or TLS tickets.

ssl_fc_npn : string
  This extracts the Next Protocol Negotiation field from an incoming connection
  made via a TLS transport layer and locally deciphered by haproxy. The result
  is a string containing the protocol name advertised by the client. The SSL
  library must have been built with support for TLS extensions enabled (check
  haproxy -vv). Note that the TLS NPN extension is not advertised unless the
  "npn" keyword on the "bind" line specifies a protocol list. Also, nothing
  forces the client to pick a protocol from this list, any other one may be
  requested. Please note that the TLS NPN extension was replaced with ALPN.

ssl_fc_protocol : string
  Returns the name of the used protocol when the incoming connection was made
  over an SSL/TLS transport layer.

ssl_fc_unique_id : binary
  When the incoming connection was made over an SSL/TLS transport layer,
  returns the TLS unique ID as defined in RFC5929 section 3. The unique id
  can be encoded to base64 using the converter: "ssl_bc_unique_id,base64".

ssl_fc_session_id : binary
  Returns the SSL ID of the front connection when the incoming connection was
  made over an SSL/TLS transport layer. It is useful to stick a given client to
  a server. It is important to note that some browsers refresh their session ID
  every few minutes.

ssl_fc_sni : string
  This extracts the Server Name Indication TLS extension (SNI) field from an
  incoming connection made via an SSL/TLS transport layer and locally
  deciphered by haproxy. The result (when present) typically is a string
  matching the HTTPS host name (253 chars or less). The SSL library must have
  been built with support for TLS extensions enabled (check haproxy -vv).

  This fetch is different from "req_ssl_sni" above in that it applies to the
  connection being deciphered by haproxy and not to SSL contents being blindly
  forwarded. See also "ssl_fc_sni_end" and "ssl_fc_sni_reg" below. This
```

```
  requires that the SSL library is build with support for TLS extensions
  enabled (check haproxy -vv).

  ACL derivatives :
    ssl_fc_sni_end : suffix match
    ssl_fc_sni_reg : regex match

ssl_fc_use_keysize : integer
  Returns the symmetric cipher key size used in bits when the incoming
  connection was made over an SSL/TLS transport layer.


7.3.5. Fetching samples from buffer contents (Layer 6)
------------------------------------------------------

Fetching samples from buffer contents is a bit different from the previous
sample fetches above because the sampled data are ephemeral. These data can
only be used when they're available and will be lost when they're forwarded.
For this reason, samples fetched from buffer contents during a request cannot
be used in a response for example. Even while the data are being fetched, they
can change. Sometimes it is necessary to set some delays or combine multiple
sample fetch methods to ensure that the expected data are complete and usable,
for example through TCP request content inspection. Please see the "tcp-request
content" keyword for more detailed information on the subject.

payload(<offset>,<length>) : binary (deprecated)
  This is an alias for "req.payload" when used in the context of a request (eg:
  "stick on", "stick match"), and for "res.payload" when used in the context of
  a response such as in "stick store response".

payload_lv(<offset1>,<length>[,<offset2>]) : binary (deprecated)
  This is an alias for "req.payload_lv" when used in the context of a request
  (eg: "stick on", "stick match"), and for "res.payload_lv" when used in the
  context of a response such as in "stick store response".

req.len : integer
req_len : integer (deprecated)
  Returns an integer value corresponding to the number of bytes present in the
  request buffer. This is mostly used in ACL. It is important to understand
  that this test does not return false as long as the buffer is changing. This
  means that a check with equality to zero will almost always immediately match
  at the beginning of the session, while a test for more data will wait for
  that data to come in and return false only when haproxy is certain that no
  more data will come in. This test was designed to be used with TCP request
  content inspection.

req.payload(<offset>,<length>) : binary
  This extracts a binary block of <length> bytes and starting at byte <offset>
  in the request buffer. As a special case, if the <length> argument is zero,
  the the whole buffer from <offset> to the end is extracted. This can be used
  with ACLs in order to check for the presence of some content in a buffer at
  any location.

  ACL alternatives :
    payload(<offset>,<length>) : hex binary match

req.payload_lv(<offset1>,<length>[,<offset2>]) : binary
  This extracts a binary block whose size is specified at <offset1> for <length>
  bytes, and which starts at <offset2> if specified or just after the length in
  the request buffer. The <offset2> parameter also supports relative offsets if
  prepended with a '+' or '-' sign.

  ACL alternatives :
    payload_lv(<offset1>,<length>[,<offset2>]) : hex binary match

```

Example : please consult the example from the "stick store-response" keyword.

req.proto_http : boolean
req_proto_http : boolean (deprecated)

Returns true when data in the request buffer look like HTTP and correctly parses as such. It is the same parser as the common HTTP request parser which is used so there should be no surprises. The test does not match until the request is complete, failed or timed out. This test may be used to report the protocol in TCP logs, but the biggest use is to block TCP request analysis until a complete HTTP request is present in the buffer, for example to track a header.

Example:

```
      # track request counts per "base" (concatenation of Host+URL)
      tcp-request inspect-delay 10s
      tcp-request content reject if !HTTP
      tcp-request content track-sc0 base table req-rate
```

req.rdp_cookie([<name>]) : string
rdp_cookie([<name>]) : string (deprecated)

When the request buffer looks like the RDP protocol, extracts the RDP cookie <name>, or any cookie if unspecified. The parser only checks for the first cookie, as illustrated in the RDP protocol specification. The cookie name is case insensitive. Generally the "MSTS" cookie name will be used, as it can contain the user name of the client connecting to the server if properly configured on the client. The "MSTSHASH" cookie is often used as well for session stickiness to servers.

This differs from "balance rdp-cookie" in that any balancing algorithm may be used and thus the distribution of clients to backend servers is not linked to a hash of the RDP cookie. It is envisaged that using a balancing algorithm such as "balance roundrobin" or "balance leastconn" will lead to a more even distribution of clients to backend servers than the hash used by "balance rdp-cookie".

ACL derivatives :
  req_rdp_cookie([<name>]) : exact string match

Example :

```
 listen tse-farm
     bind 0.0.0.0:3389
     # wait up to 5s for an RDP cookie in the request
     tcp-request inspect-delay 5s
     tcp-request content accept if RDP_COOKIE
     # apply RDP cookie persistence
     persist rdp-cookie
     # Persist based on the mstshash cookie
     # This is only useful makes sense if
     # balance rdp-cookie is not used
     stick-table type string size 204800
     stick on req.rdp_cookie(mstshash)
     server srv1 1.1.1.1:3389
     server srv1 1.1.1.2:3389
```

See also : "balance rdp-cookie", "persist rdp-cookie", "tcp-request" and the "req_rdp_cookie" ACL.

req.rdp_cookie_cnt([name]) : integer
rdp_cookie_cnt([name]) : integer (deprecated)

Tries to parse the request buffer as RDP protocol, then returns an integer corresponding to the number of RDP cookies found. If an optional cookie name is passed, only cookies matching this name are considered. This is mostly used in ACL.

ACL derivatives :

  req_rdp_cookie_cnt([<name>]) : integer match

req.ssl_ec_ext : boolean

Returns a boolean identifying if client sent the Supported Elliptic Curves Extension as defined in RFC4492, section 5.1. within the SSL ClientHello message. This can be used to present ECC compatible clients with EC certificate and to use RSA for all others, on the same IP address. Note that this only applies to raw contents found in the request buffer and not to contents deciphered via an SSL data layer, so this will not work with "bind" lines having the "ssl" option.

req.ssl_hello_type : integer
req_ssl_hello_type : integer (deprecated)

Returns an integer value containing the type of the SSL hello message found in the request buffer if the buffer contains data that parse as a complete SSL (v3 or superior) client hello message. Note that this only applies to raw contents found in the request buffer and not to contents deciphered via an SSL data layer, so this will not work with "bind" lines having the "ssl" option. This is mostly used in ACL to detect presence of an SSL hello message that is supposed to contain an SSL session ID usable for stickiness.

req.ssl_sni : string
req_ssl_sni : string (deprecated)

Returns a string containing the value of the Server Name TLS extension sent by a client in a TLS stream passing through the request buffer if the buffer contains data that parse as a complete SSL (v3 or superior) client hello message. Note that this only applies to raw contents found in the request buffer and not to contents deciphered via an SSL data layer, so this will not work with "bind" lines having the "ssl" option. SNI normally contains the name of the host the client tries to connect to (for recent browsers). SNI is useful for allowing or denying access to certain hosts when SSL/TLS is used by the client. This test was designed to be used with TCP request content inspection. If content switching is needed, it is recommended to first wait for a complete client hello (type 1), like in the example below. See also "ssl_fc_sni".

ACL derivatives :
  req_ssl_sni : exact string match

Examples :

```
   # Wait for a client hello for at most 5 seconds
   tcp-request inspect-delay 5s
   tcp-request content accept if { req_ssl_hello_type 1 }
   use_backend bk_allow if { req_ssl_sni -f allowed_sites }
   default_backend bk_sorry_page
```

req.ssl_st_ext : integer

Returns 0 if the client didn't send a SessionTicket TLS Extension (RFC5077)
Returns 1 if the client sent SessionTicket TLS Extension
Returns 2 if the client also sent non-zero length TLS SessionTicket
Note that this only applies to raw contents found in the request buffer and not to contents deciphered via an SSL data layer, so this will not work with "bind" lines having the "ssl" option. This can for example be used to detect whether the client sent a SessionTicket or not and stick it accordingly, if no SessionTicket then stick on SessionID or don't stick as there's no server side state is there when SessionTickets are in use.

req.ssl_ver : integer
req_ssl_ver : integer (deprecated)

Returns an integer value containing the version of the SSL/TLS protocol of a stream present in the request buffer. Both SSLv2 hello messages and SSLv3 messages are supported. TLSv1 is announced as SSL version 3.1. The value is composed of the major version multiplied by 65536, added to the minor version. Note that this only applies to raw contents found in the request buffer and not to contents deciphered via an SSL data layer, so this will not

```
  work with "bind" lines having the "ssl" option. The ACL version of the test
  matches against a decimal notation in the form MAJOR.MINOR (eg: 3.1). This
  fetch is mostly used in ACL.

  ACL derivatives :
    req_ssl_ver : decimal match

res.len : integer
  Returns an integer value corresponding to the number of bytes present in the
  response buffer. This is mostly used in ACL. It is important to understand
  that this test does not return false as long as the buffer is changing. This
  means that a check with equality to zero will almost always immediately match
  at the beginning of the session, while a test for more data will wait for
  that data to come in and return false only when haproxy is certain that no
  more data will come in. This test was designed to be used with TCP response
  content inspection.

res.payload(<offset>,<length>) : binary
  This extracts a binary block of <length> bytes and starting at byte <offset>
  in the response buffer. As a special case, if the <length> argument is zero,
  the the whole buffer from <offset> to the end is extracted. This can be used
  with ACLs in order to check for the presence of some content in a buffer at
  any location.

res.payload_lv(<offset1>,<length>[,<offset2>]) : binary
  This extracts a binary block whose size is specified at <offset1> for <length>
  bytes, and which starts at <offset2> if specified or just after the length in
  the response buffer. The <offset2> parameter also supports relative offsets
  if prepended with a '+' or '-' sign.

  Example : please consult the example from the "stick store-response" keyword.

res.ssl_hello_type : integer
rep_ssl_hello_type : integer (deprecated)
  Returns an integer value containing the type of the SSL hello message found
  in the response buffer if the buffer contains data that parses as a complete
  SSL (v3 or superior) hello message. Note that this only applies to raw
  contents found in the response buffer and not to contents deciphered via an
  SSL data layer, so this will not work with "server" lines having the "ssl"
  option. This is mostly used in ACL to detect presence of an SSL hello message
  that is supposed to contain an SSL session ID usable for stickiness.

wait_end : boolean
  This fetch either returns true when the inspection period is over, or does
  not fetch. It is only used in ACLs, in conjunction with content analysis to
  avoid returning a wrong verdict early.  It may also be used to delay some
  actions, such as a delayed reject for some special addresses. Since it either
  stops the rules evaluation or immediately returns true, it is recommended to
  use this acl as the last one in a rule.  Please note that the default ACL
  "WAIT_END" is always usable without prior declaration. This test was designed
  to be used with TCP request content inspection.

  Examples :
     # delay every incoming request by 2 seconds
     tcp-request inspect-delay 2s
     tcp-request content accept if WAIT_END

     # don't immediately tell bad guys they are rejected
     tcp-request inspect-delay 10s
     acl goodguys src 10.0.0.0/24
     acl badguys  src 10.0.1.0/24
     tcp-request content accept if goodguys
     tcp-request content reject if badguys WAIT_END
     tcp-request content reject

```

7.3.6. Fetching HTTP samples (Layer 7)

```
-------------------------------------

It is possible to fetch samples from HTTP contents, requests and responses.
This application layer is also called layer 7. It is only possible to fetch the
data in this section when a full HTTP request or response has been parsed from
its respective request or response buffer. This is always the case with all
HTTP specific rules and for sections running with "mode http". When using TCP
content inspection, it may be necessary to support an inspection delay in order
to let the request or response come in first. These fetches may require a bit
more CPU resources than the layer 4 ones, but not much since the request and
response are indexed.

base : string
  This returns the concatenation of the first Host header and the path part of
  the request, which starts at the first slash and ends before the question
  mark. It can be useful in virtual hosted environments to detect URL abuses as
  well as to improve shared caches efficiency. Using this with a limited size
  stick table also allows one to collect statistics about most commonly
  requested objects by host/path. With ACLs it can allow simple content
  switching rules involving the host and the path at the same time, such as
  "www.example.com/favicon.ico". See also "path" and "uri".

  ACL derivatives :
    base     : exact string match
    base_beg : prefix match
    base_dir : subdir match
    base_dom : domain match
    base_end : suffix match
    base_len : length match
    base_reg : regex match
    base_sub : substring match

base32 : integer
  This returns a 32-bit hash of the value returned by the "base" fetch method
  above. This is useful to track per-URL activity on high traffic sites without
  having to store all URLs. Instead a shorter hash is stored, saving a lot of
  memory. The output type is an unsigned integer. The hash function used is
  SDBM with full avalanche on the output. Technically, base32 is exactly equal
  to "base,sdbm(1)".

base32+src : binary
  This returns the concatenation of the base32 fetch above and the src fetch
  below. The resulting type is of type binary, with a size of 8 or 20 bytes
  depending on the source address family. This can be used to track per-IP,
  per-URL counters.

capture.req.hdr(<idx>) : string
  This extracts the content of the header captured by the "capture request
  header", idx is the position of the capture keyword in the configuration.
  The first entry is an index of 0. See also: "capture request header".

capture.req.method : string
  This extracts the METHOD of an HTTP request. It can be used in both request
  and response. Unlike "method", it can be used in both request and response
  because it's allocated.

capture.req.uri : string
  This extracts the request's URI, which starts at the first slash and ends
  before the first space in the request (without the host part). Unlike "path"
  and "url", it can be used in both request and response because it's
  allocated.

capture.req.ver : string
```

```
  This extracts the request's HTTP version and returns either "HTTP/1.0" or
  "HTTP/1.1". Unlike "req.ver", it can be used in both request, response, and
  logs because it relies on a persistent flag.

capture.res.hdr(<idx>) : string
  This extracts the content of the header captured by the "capture response
  header", idx is the position of the capture keyword in the configuration.
  The first entry is an index of 0.
  See also: "capture response header"

capture.res.ver : string
  This extracts the response's HTTP version and returns either "HTTP/1.0" or
  "HTTP/1.1". Unlike "res.ver", it can be used in logs because it relies on a
  persistent flag.

req.body : binary
  This returns the HTTP request's available body as a block of data. It
  requires that the request body has been buffered made available using
  "option http-buffer-request". In case of chunked-encoded body, currently only
  the first chunk is analyzed.

req.body_param([<name>) : string
  This fetch assumes that the body of the POST request is url-encoded. The user
  can check if the "content-type" contains the value
  "application/x-www-form-urlencoded". This extracts the first occurrence of the
  parameter <name> in the body, which ends before '&'. The parameter name is
  case-sensitive. If no name is given, any parameter will match, and the first
  one will be returned. The result is a string corresponding to the value of the
  parameter <name> as presented in the request body (no URL decoding is
  performed). Note that the ACL version of this fetch iterates over multiple
  parameters and will iteratively report all parameters values if no name is
  given.

req.body_len : integer
  This returns the length of the HTTP request's available body in bytes. It may
  be lower than the advertised length if the body is larger than the buffer. It
  requires that the request body has been buffered made available using
  "option http-buffer-request".

req.body_size : integer
  This returns the advertised length of the HTTP request's body in bytes. It
  will represent the advertised Content-Length header, or the size of the first
  chunk in case of chunked encoding. In order to parse the chunks, it requires
  that the request body has been buffered made available using
  "option http-buffer-request".

req.cook([<name>]) : string
cook([<name>]) : string (deprecated)
  This extracts the last occurrence of the cookie name <name> on a "Cookie"
  header line from the request, and returns its value as string. If no name is
  specified, the first cookie value is returned. When used with ACLs, all
  matching cookies are evaluated. Spaces around the name and the value are
  ignored as requested by the Cookie header specification (RFC6265). The cookie
  name is case-sensitive. Empty cookies are valid, so an empty cookie may very
  well return an empty value if it is present. Use the "found" match to detect
  presence. Use the res.cook() variant for response cookies sent by the server.

  ACL derivatives :
    cook([<name>])     : exact string match
    cook_beg([<name>]) : prefix match
    cook_dir([<name>]) : subdir match
    cook_dom([<name>]) : domain match
    cook_end([<name>]) : suffix match
    cook_len([<name>]) : length match
    cook_reg([<name>]) : regex match
```

```
    cook_sub([<name>]) : substring match

req.cook_cnt([<name>]) : integer
cook_cnt([<name>]) : integer (deprecated)
  Returns an integer value representing the number of occurrences of the cookie
  <name> in the request, or all cookies if <name> is not specified.

req.cook_val([<name>]) : integer
cook_val([<name>]) : integer (deprecated)
  This extracts the last occurrence of the cookie name <name> on a "Cookie"
  header line from the request, and converts its value to an integer which is
  returned. If no name is specified, the first cookie value is returned. When
  used in ACLs, all matching names are iterated over until a value matches.

cookie([<name>]) : string (deprecated)
  This extracts the last occurrence of the cookie name <name> on a "Cookie"
  header line from the request, or a "Set-Cookie" header from the response, and
  returns its value as a string. A typical use is to get multiple clients
  sharing a same profile use the same server. This can be similar to what
  "appsession" did with the "request-learn" statement, but with support for
  multi-peer synchronization and state keeping across restarts. If no name is
  specified, the first cookie value is returned. This fetch should not be used
  anymore and should be replaced by req.cook() or res.cook() instead as it
  ambiguously uses the direction based on the context where it is used.

hdr([<name>[,<occ>]]) : string
  This is equivalent to req.hdr() when used on requests, and to res.hdr() when
  used on responses. Please refer to these respective fetches for more details.
  In case of doubt about the fetch direction, please use the explicit ones.
  Note that contrary to the hdr() sample fetch method, the hdr_* ACL keywords
  unambiguously apply to the request headers.

req.fhdr(<name>[,<occ>]) : string
  This extracts the last occurrence of header <name> in an HTTP request. When
  used from an ACL, all occurrences are iterated over until a match is found.
  Optionally, a specific occurrence might be specified as a position number.
  Positive values indicate a position from the first occurrence, with 1 being
  the first one. Negative values indicate positions relative to the last one,
  with -1 being the last one. It differs from req.hdr() in that any commas
  present in the value are returned and are not used as delimiters. This is
  sometimes useful with headers such as User-Agent.

req.fhdr_cnt([<name>]) : integer
  Returns an integer value representing the number of occurrences of request
  header field name <name>, or the total number of header fields if <name> is
  not specified. Contrary to its req.hdr_cnt() cousin, this function returns
  the number of full line headers and does not stop on commas.

req.hdr([<name>[,<occ>]]) : string
  This extracts the last occurrence of header <name> in an HTTP request. When
  used from an ACL, all occurrences are iterated over until a match is found.
  Optionally, a specific occurrence might be specified as a position number.
  Positive values indicate a position from the first occurrence, with 1 being
  the first one. Negative values indicate positions relative to the last one,
  with -1 being the last one. A typical use is with the X-Forwarded-For header
  once converted to IP, associated with an IP stick-table. The function
  considers any comma as a delimiter for distinct values. If full-line headers
  are desired instead, use req.fhdr(). Please carefully check RFC2616 to know
  how certain headers are supposed to be parsed. Also, some of them are case
  insensitive (eg: Connection).

  ACL derivatives :
    hdr([<name>[,<occ>]])     : exact string match
    hdr_beg([<name>[,<occ>]]) : prefix match
    hdr_dir([<name>[,<occ>]]) : subdir match
    hdr_dom([<name>[,<occ>]]) : domain match
    hdr_end([<name>[,<occ>]]) : suffix match
    hdr_len([<name>[,<occ>]]) : length match
    hdr_reg([<name>[,<occ>]]) : regex match
    hdr_sub([<name>[,<occ>]]) : substring match

req.hdr_cnt([<name>]) : integer
hdr_cnt([<header>]) : integer (deprecated)
  Returns an integer value representing the number of occurrences of request
  header field name <name>, or the total number of header field values if
  <name> is not specified. It is important to remember that one header line may
  count as several headers if it has several values. The function considers any
  comma as a delimiter for distinct values. If full-line headers are desired
  instead, req.fhdr_cnt() should be used instead. With ACLs, it can be used to
  detect presence, absence or abuse of a specific header, as well as to block
  request smuggling attacks by rejecting requests which contain more than one
  of certain headers. See "req.hdr" for more information on header matching.

req.hdr_ip([<name>[,<occ>]]) : ip
hdr_ip([<name>[,<occ>]]) : ip (deprecated)
  This extracts the last occurrence of header <name> in an HTTP request,
  converts it to an IPv4 or IPv6 address and returns this address. When used
  with ACLs, all occurrences are checked, and if <name> is omitted, every value
  of every header is checked. Optionally, a specific occurrence might be
  specified as a position number. Positive values indicate a position from the
  first occurrence, with 1 being the first one.  Negative values indicate
  positions relative to the last one, with -1 being the last one. A typical use
  is with the X-Forwarded-For and X-Client-IP headers.

req.hdr_val([<name>[,<occ>]]) : integer
hdr_val([<name>[,<occ>]]) : integer (deprecated)
  This extracts the last occurrence of header <name> in an HTTP request, and
  converts it to an integer value. When used with ACLs, all occurrences are
  checked, and if <name> is omitted, every value of every header is checked.
  Optionally, a specific occurrence might be specified as a position number.
  Positive values indicate a position from the first occurrence, with 1 being
  the first one. Negative values indicate positions relative to the last one,
  with -1 being the last one. A typical use is with the X-Forwarded-For header.

http_auth(<userlist>) : boolean
  Returns a boolean indicating whether the authentication data received from
  the client match a username & password stored in the specified userlist. This
  fetch function is not really useful outside of ACLs. Currently only http
  basic auth is supported.

http_auth_group(<userlist>) : string
  Returns a string corresponding to the user name found in the authentication
  data received from the client if both the user name and password are valid
  according to the specified userlist. The main purpose is to use it in ACLs
  where it is then checked whether the user belongs to any group within a list.
  This fetch function is not really useful outside of ACLs. Currently only http
  basic auth is supported.

  ACL derivatives :
    http_auth_group(<userlist>) : group ...
    Returns true when the user extracted from the request and whose password is
    valid according to the specified userlist belongs to at least one of the
    groups.

http_first_req : boolean
  Returns true when the request being processed is the first one of the
  connection. This can be used to add or remove headers that may be missing
  from some requests when a request is not the first one, or to help grouping
  requests in the logs.

```

```
method : integer + string
  Returns an integer value corresponding to the method in the HTTP request. For
  example, "GET" equals 1 (check sources to establish the matching). Value 9
  means "other method" and may be converted to a string extracted from the
  stream. This should not be used directly as a sample, this is only meant to
  be used from ACLs, which transparently convert methods from patterns to these
  integer + string values. Some predefined ACL already check for most common
  methods.

  ACL derivatives :
    method : case insensitive method match

  Example :
      # only accept GET and HEAD requests
      acl valid_method method GET HEAD
      http-request deny if ! valid_method

path : string
  This extracts the request's URL path, which starts at the first slash and
  ends before the question mark (without the host part). A typical use is with
  prefetch-capable caches, and with portals which need to aggregate multiple
  information from databases and keep them in caches. Note that with outgoing
  caches, it would be wiser to use "url" instead. With ACLs, it's typically
  used to match exact file names (eg: "/login.php"), or directory parts using
  the derivative forms. See also the "url" and "base" fetch methods.

  ACL derivatives :
    path     : exact string match
    path_beg : prefix match
    path_dir : subdir match
    path_dom : domain match
    path_end : suffix match
    path_len : length match
    path_reg : regex match
    path_sub : substring match

query : string
  This extracts the request's query string, which starts after the first
  question mark. If no question mark is present, this fetch returns nothing. If
  a question mark is present but nothing follows, it returns an empty string.
  This means it's possible to easily know whether a query string is present
  using the "found" matching method. This fetch is the completemnt of "path"
  which stops before the question mark.

req.hdr_names([<delim>]) : string
  This builds a string made from the concatenation of all header names as they
  appear in the request when the rule is evaluated. The default delimiter is
  the comma (',') but it may be overridden as an optional argument <delim>. In
  this case, only the first character of <delim> is considered.

req.ver : string
req_ver : string (deprecated)
  Returns the version string from the HTTP request, for example "1.1". This can
  be useful for logs, but is mostly there for ACL. Some predefined ACL already
  check for versions 1.0 and 1.1.

  ACL derivatives :
    req_ver : exact string match

res.comp : boolean
  Returns the boolean "true" value if the response has been compressed by
  HAProxy, otherwise returns boolean "false". This may be used to add
  information in the logs.

res.comp_algo : string
```

```
  Returns a string containing the name of the algorithm used if the response
  was compressed by HAProxy, for example : "deflate". This may be used to add
  some information in the logs.

res.cook([<name>]) : string
scook([<name>]) : string (deprecated)
  This extracts the last occurrence of the cookie name <name> on a "Set-Cookie"
  header line from the response, and returns its value as string. If no name is
  specified, the first cookie value is returned.

  ACL derivatives :
    scook([<name>] : exact string match

res.cook_cnt([<name>]) : integer
scook_cnt([<name>]) : integer (deprecated)
  Returns an integer value representing the number of occurrences of the cookie
  <name> in the response, or all cookies if <name> is not specified. This is
  mostly useful when combined with ACLs to detect suspicious responses.

res.cook_val([<name>]) : integer
scook_val([<name>]) : integer (deprecated)
  This extracts the last occurrence of the cookie name <name> on a "Set-Cookie"
  header line from the response, and converts its value to an integer which is
  returned. If no name is specified, the first cookie value is returned.

res.fhdr([<name>[,<occ>]]) : string
  This extracts the last occurrence of header <name> in an HTTP response, or of
  the last header if no <name> is specified. Optionally, a specific occurrence
  might be specified as a position number. Positive values indicate a position
  from the first occurrence, with 1 being the first one. Negative values
  indicate positions relative to the last one, with -1 being the last one. It
  differs from res.hdr() in that any commas present in the value are returned
  and are not used as delimiters. If this is not desired, the res.hdr() fetch
  should be used instead. This is sometimes useful with headers such as Date or
  Expires.

res.fhdr_cnt([<name>]) : integer
  Returns an integer value representing the number of occurrences of response
  header field name <name>, or the total number of header fields if <name> is
  not specified. Contrary to its res.hdr_cnt() cousin, this function returns
  the number of full line headers and does not stop on commas. If this is not
  desired, the res.hdr_cnt() fetch should be used instead.

res.hdr([<name>[,<occ>]]) : string
shdr([<name>[,<occ>]]) : string (deprecated)
  This extracts the last occurrence of header <name> in an HTTP response, or of
  the last header if no <name> is specified. Optionally, a specific occurrence
  might be specified as a position number. Positive values indicate a position
  from the first occurrence, with 1 being the first one. Negative values
  indicate positions relative to the last one, with -1 being the last one. This
  can be useful to learn some data into a stick-table. The function considers
  any comma as a delimiter for distinct values. If this is not desired, the
  res.fhdr() fetch should be used instead.

  ACL derivatives :
    shdr([<name>[,<occ>]])     : exact string match
    shdr_beg([<name>[,<occ>]]) : prefix match
    shdr_dir([<name>[,<occ>]]) : subdir match
    shdr_dom([<name>[,<occ>]]) : domain match
    shdr_end([<name>[,<occ>]]) : suffix match
    shdr_len([<name>[,<occ>]]) : length match
    shdr_reg([<name>[,<occ>]]) : regex match
    shdr_sub([<name>[,<occ>]]) : substring match

res.hdr_cnt([<name>]) : integer
```

```
shdr_cnt([<name>]) : integer (deprecated)
  Returns an integer value representing the number of occurrences of response
  header field name <name>, or the total number of header fields if <name> is
  not specified. The function considers any comma as a delimiter for distinct
  values. If this is not desired, the res.fhdr_cnt() fetch should be used
  instead.

res.hdr_ip([<name>[,<occ>]]) : ip
shdr_ip([<name>[,<occ>]]) : ip (deprecated)
  This extracts the last occurrence of header <name> in an HTTP response,
  convert it to an IPv4 or IPv6 address and returns this address. Optionally, a
  specific occurrence might be specified as a position number. Positive values
  indicate a position from the first occurrence, with 1 being the first one.
  Negative values indicate positions relative to the last one, with -1 being
  the last one. This can be useful to learn some data into a stick table.

res.hdr_names([<delim>]) : string
  This builds a string made from the concatenation of all header names as they
  appear in the response when the rule is evaluated. The default delimiter is
  the comma (',') but it may be overridden as an optional argument <delim>. In
  this case, only the first character of <delim> is considered.

res.hdr_val([<name>[,<occ>]]) : integer
shdr_val([<name>[,<occ>]]) : integer (deprecated)
  This extracts the last occurrence of header <name> in an HTTP response, and
  converts it to an integer value. Optionally, a specific occurrence might be
  specified as a position number. Positive values indicate a position from the
  first occurrence, with 1 being the first one. Negative values indicate
  positions relative to the last one, with -1 being the last one. This can be
  useful to learn some data into a stick table.

res.ver : string
resp_ver : string (deprecated)
  Returns the version string from the HTTP response, for example "1.1". This
  can be useful for logs, but is mostly there for ACL.

  ACL derivatives :
    resp_ver : exact string match

set-cookie([<name>]) : string (deprecated)
  This extracts the last occurrence of the cookie name <name> on a "Set-Cookie"
  header line from the response and uses the corresponding value to match. This
  can be comparable to what "appsession" did with default options, but with
  support for multi-peer synchronization and state keeping across restarts.

  This fetch function is deprecated and has been superseded by the "res.cook"
  fetch. This keyword will disappear soon.

status : integer
  Returns an integer containing the HTTP status code in the HTTP response, for
  example, 302. It is mostly used within ACLs and integer ranges, for example,
  to remove any Location header if the response is not a 3xx.

url : string
  This extracts the request's URL as presented in the request. A typical use is
  with prefetch-capable caches, and with portals which need to aggregate
  multiple information from databases and keep them in caches. With ACLs, using
  "path" is preferred over using "url", because clients may send a full URL as
  is normally done with proxies. The only real use is to match "*" which does
  not match in "path", and for which there is already a predefined ACL. See
  also "path" and "base".

  ACL derivatives :
    url     : exact string match
    url_beg : prefix match
    url_dir : subdir match
    url_dom : domain match
    url_end : suffix match
    url_len : length match
    url_reg : regex match
    url_sub : substring match
```

url_ip : ip

This extracts the IP address from the request's URL when the host part is presented as an IP address. Its use is very limited. For instance, a monitoring system might use this field as an alternative for the source IP in order to test what path a given source address would follow, or to force an entry in a table for a given source address. With ACLs it can be used to restrict access to certain systems through a proxy, for example when combined with option "http_proxy".

url_port : integer

This extracts the port part from the request's URL. Note that if the port is not specified in the request, port 80 is assumed. With ACLs it can be used to restrict access to certain systems through a proxy, for example when combined with option "http_proxy".

urlp([<name>[,<delim>]]) : string
url_param([<name>[,<delim>]]) : string

This extracts the first occurrence of the parameter <name> in the query string, which begins after either '?' or <delim>, and which ends before '&', ';' or <delim>. The parameter name is case-sensitive. If no name is given, any parameter will match, and the first one will be returned. The result is a string corresponding to the value of the parameter <name> as presented in the request (no URL decoding is performed). This can be used for session stickiness based on a client ID, to extract an application cookie passed as a URL parameter, or in ACLs to apply some checks. Note that the ACL version of this fetch iterates over multiple parameters and will iteratively report all parameters values if no name is given

```
  ACL derivatives :
    urlp(<name>[,<delim>])     : exact string match
    urlp_beg(<name>[,<delim>]) : prefix match
    urlp_dir(<name>[,<delim>]) : subdir match
    urlp_dom(<name>[,<delim>]) : domain match
    urlp_end(<name>[,<delim>]) : suffix match
    urlp_len(<name>[,<delim>]) : length match
    urlp_reg(<name>[,<delim>]) : regex match
    urlp_sub(<name>[,<delim>]) : substring match


  Example :
      # match http://example.com/foo?PHPSESSIONID=some_id
      stick on urlp(PHPSESSIONID)
      # match http://example.com/foo;JSESSIONID=some_id
      stick on urlp(JSESSIONID,;)
```

urlp_val([<name>[,<delim>])] : integer

See "urlp" above. This one extracts the URL parameter <name> in the request and converts it to an integer value. This can be used for session stickiness based on a user ID for example, or with ACLs to match a page number or price.

7.4. Pre-defined ACLs

Some predefined ACLs are hard-coded so that they do not have to be declared in every frontend which needs them. They all have their names in upper case in order to avoid confusion. Their equivalence is provided below.

ACL name	Equivalent to	Usage
FALSE	always_false	never match
HTTP	req_proto_http	match if protocol is valid HTTP
HTTP_1.0	req_ver 1.0	match HTTP version 1.0
HTTP_1.1	req_ver 1.1	match HTTP version 1.1
HTTP_CONTENT	hdr_val(content-length) gt 0	match an existing content-length
HTTP_URL_ABS	url_reg ^[^/:]*://	match absolute URL with scheme
HTTP_URL_SLASH	url_beg /	match URL beginning with "/"
HTTP_URL_STAR	url *	match URL equal to "*"
LOCALHOST	src 127.0.0.1/8	match connection from local host
METH_CONNECT	method CONNECT	match HTTP CONNECT method
METH_GET	method GET HEAD	match HTTP GET or HEAD method
METH_HEAD	method HEAD	match HTTP HEAD method
METH_OPTIONS	method OPTIONS	match HTTP OPTIONS method
METH_POST	method POST	match HTTP POST method
METH_TRACE	method TRACE	match HTTP TRACE method
RDP_COOKIE	req_rdp_cookie_cnt gt 0	match presence of an RDP cookie
REQ_CONTENT	req_len gt 0	match data in the request buffer
TRUE	always_true	always match
WAIT_END	wait_end	wait for end of content analysis

8. Logging

One of HAProxy's strong points certainly lies is its precise logs. It probably provides the finest level of information available for such a product, which is very important for troubleshooting complex environments. Standard information provided in logs include client ports, TCP/HTTP state timers, precise session state at termination and precise termination cause, information about decisions to direct traffic to a server, and of course the ability to capture arbitrary headers.

In order to improve administrators reactivity, it offers a great transparency about encountered problems, both internal and external, and it is possible to send logs to different sources at the same time with different level filters :

- global process-level logs (system errors, start/stop, etc..)
- per-instance system and internal errors (lack of resource, bugs, ...)
- per-instance external troubles (servers up/down, max connections)
- per-instance activity (client connections), either at the establishment or at the termination.
- per-request control of log-level, eg:

```
      http-request set-log-level silent if sensitive_request
```

The ability to distribute different levels of logs to different log servers allow several production teams to interact and to fix their problems as soon as possible. For example, the system team might monitor system-wide errors, while the application team might be monitoring the up/down for their servers in real time, and the security team might analyze the activity logs with one hour delay.

8.1. Log levels

TCP and HTTP connections can be logged with information such as the date, time, source IP address, destination address, connection duration, response times, HTTP request, HTTP return code, number of bytes transmitted, conditions in which the session ended, and even exchanged cookies values. For example track a particular user's problems. All messages may be sent to up to two syslog servers. Check the "log" keyword in section 4.2 for more information about log facilities.

8.2. Log formats

HAProxy supports 5 log formats. Several fields are common between these formats and will be detailed in the following sections. A few of them may vary slightly with the configuration, due to indicators specific to certain options. The supported formats are as follows :

- the default format, which is very basic and very rarely used. It only provides very basic information about the incoming connection at the moment it is accepted : source IP:port, destination IP:port, and frontend-name. This mode will eventually disappear so it will not be described to great extents.
- the TCP format, which is more advanced. This format is enabled when "option tcplog" is set on the frontend. HAProxy will then usually wait for the connection to terminate before logging. This format provides much richer information, such as timers, connection counts, queue size, etc... This format is recommended for pure TCP proxies.
- the HTTP format, which is the most advanced for HTTP proxying. This format is enabled when "option httplog" is set on the frontend. It provides the same information as the TCP format with some HTTP-specific fields such as the request, the status code, and captures of headers and cookies. This format is recommended for HTTP proxies.
- the CLF HTTP format, which is equivalent to the HTTP format, but with the fields arranged in the same order as the CLF format. In this mode, all timers, captures, flags, etc... appear one per field after the end of the common fields, in the same order they appear in the standard HTTP format.
- the custom log format, allows you to make your own log line.

Next sections will go deeper into details for each of these formats. Format specification will be performed on a "field" basis. Unless stated otherwise, a field is a portion of text delimited by any number of spaces. Since syslog servers are susceptible of inserting fields at the beginning of a line, it is always assumed that the first field is the one containing the process name and identifier.

Note : Since log lines may be quite long, the log examples in sections below might be broken into multiple lines. The example log lines will be prefixed with 3 closing angle brackets ('>>>') and each time a log is broken into multiple lines, each non-final line will end with a backslash ('\') and the next line will start indented by two characters.

8.2.1. Default log format

This format is used when no specific option is set. The log is emitted as soon as the connection is accepted. One should note that this currently is the only format which logs the request's destination IP and ports.

```
  Example :
        listen www
            mode http
            log global
            server srv1 127.0.0.1:8000

    >>> Feb  6 12:12:09 localhost \
          haproxy[14385]: Connect from 10.0.1.2:33312 to 10.0.3.31:8012 \
          (www/HTTP)
```

Field	Format	Extract from the example above
1	process_name '[' pid ']:'	haproxy[14385]:
2	'Connect from'	Connect from
3	source_ip ':' source_port	10.0.1.2:33312
4	'to'	to
5	destination_ip ':' destination_port	10.0.3.31:8012
6	'(' frontend_name '/' mode ')'	(www/HTTP)

Detailed fields description :

- "source_ip" is the IP address of the client which initiated the connection.
- "source_port" is the TCP port of the client which initiated the connection.
- "destination_ip" is the IP address the client connected to.
- "destination_port" is the TCP port the client connected to.
- "frontend_name" is the name of the frontend (or listener) which received and processed the connection.
- "mode is the mode the frontend is operating (TCP or HTTP).

In case of a UNIX socket, the source and destination addresses are marked as "unix:" and the ports reflect the internal ID of the socket which accepted the connection (the same ID as reported in the stats).

It is advised not to use this deprecated format for newer installations as it will eventually disappear.

8.2.2. TCP log format

The TCP format is used when "option tcplog" is specified in the frontend, and is the recommended format for pure TCP proxies. It provides a lot of precious information for troubleshooting. Since this format includes timers and byte counts, the log is normally emitted at the end of the session. It can be emitted earlier if "option logasap" is specified, which makes sense in most environments with long sessions such as remote terminals. Sessions which match the "monitor" rules are never logged. It is also possible not to emit logs for sessions for which no data were exchanged between the client and the server, by specifying "option dontlognull" in the frontend. Successful connections will not be logged if "option dontlog-normal" is specified in the frontend. A few fields may slightly vary depending on some configuration options, those are marked with a star ('*') after the field name below.

```
  Example :
        frontend fnt
            mode tcp
            option tcplog
            log global
            default_backend bck

        backend bck
            server srv1 127.0.0.1:8000

    >>> Feb  6 12:12:56 localhost \
          haproxy[14387]: 10.0.1.2:33313 [06/Feb/2009:12:12:51.443] fnt \
          bck/srv1 0/0/5007 212 -- 0/0/0/0/3 0/0
```

Field	Format	Extract from the example above
1	process_name '[' pid ']:'	haproxy[14387]:
2	client_ip ':' client_port	10.0.1.2:33313
3	'[' accept_date ']'	[06/Feb/2009:12:12:51.443]
4	frontend_name	fnt
5	backend_name '/' server_name	bck/srv1
6	Tw '/' Tc '/' Tt*	0/0/5007
7	bytes_read*	212
8	termination_state	--

```
      9   actconn '/' feconn '/' beconn '/' srv_conn '/' retries*    0/0/0/0/3
     10   srv_queue '/' backend_queue                                      0/0
```

Detailed fields description :

- "client_ip" is the IP address of the client which initiated the TCP connection to haproxy. If the connection was accepted on a UNIX socket instead, the IP address would be replaced with the word "unix". Note that when the connection is accepted on a socket configured with "accept-proxy" and the PROXY protocol is correctly used, then the logs will reflect the forwarded connection's information.

- "client_port" is the TCP port of the client which initiated the connection. If the connection was accepted on a UNIX socket instead, the port would be replaced with the ID of the accepting socket, which is also reported in the stats interface.

- "accept_date" is the exact date when the connection was received by haproxy (which might be very slightly different from the date observed on the network if there was some queuing in the system's backlog). This is usually the same date which may appear in any upstream firewall's log.

- "frontend_name" is the name of the frontend (or listener) which received and processed the connection.

- "backend_name" is the name of the backend (or listener) which was selected to manage the connection to the server. This will be the same as the frontend if no switching rule has been applied, which is common for TCP applications.

- "server_name" is the name of the last server to which the connection was sent, which might differ from the first one if there were connection errors and a redispatch occurred. Note that this server belongs to the backend which processed the request. If the connection was aborted before reaching a server, "<NOSRV>" is indicated instead of a server name.

- "Tw" is the total time in milliseconds spent waiting in the various queues. It can be "-1" if the connection was aborted before reaching the queue. See "Timers" below for more details.

- "Tc" is the total time in milliseconds spent waiting for the connection to establish to the final server, including retries. It can be "-1" if the connection was aborted before a connection could be established. See "Timers" below for more details.

- "Tt" is the total time in milliseconds elapsed between the accept and the last close. It covers all possible processing. There is one exception, if "option logasap" was specified, then the time counting stops at the moment the log is emitted. In this case, a '+' sign is prepended before the value, indicating that the final one will be larger. See "Timers" below for more details.

- "bytes_read" is the total number of bytes transmitted from the server to the client when the log is emitted. If "option logasap" is specified, the this value will be prefixed with a '+' sign indicating that the final one may be larger. Please note that this value is a 64-bit counter, so log analysis tools must be able to handle it without overflowing.

- "termination_state" is the condition the session was in when the session ended. This indicates the session state, which side caused the end of session to happen, and for what reason (timeout, error, ...). The normal flags should be "--", indicating the session was closed by either end with no data remaining in buffers. See below "Session state at disconnection" for more details.

- "actconn" is the total number of concurrent connections on the process when the session was logged. It is useful to detect when some per-process system limits have been reached. For instance, if actconn is close to 512 when multiple connection errors occur, chances are high that the system limits the process to use a maximum of 1024 file descriptors and that all of them are used. See section 3 "Global parameters" to find how to tune the system.

- "feconn" is the total number of concurrent connections on the frontend when the session was logged. It is useful to estimate the amount of resource required to sustain high loads, and to detect when the frontend's "maxconn" has been reached. Most often when this value increases by huge jumps, it is because there is congestion on the backend servers, but sometimes it can be caused by a denial of service attack.

- "beconn" is the total number of concurrent connections handled by the backend when the session was logged. It includes the total number of concurrent connections active on servers as well as the number of connections pending in queues. It is useful to estimate the amount of additional servers needed to support high loads for a given application. Most often when this value increases by huge jumps, it is because there is congestion on the backend servers, but sometimes it can be caused by a denial of service attack.

- "srv_conn" is the total number of concurrent connections still active on the server when the session was logged. It can never exceed the server's configured "maxconn" parameter. If this value is very often close or equal to the server's "maxconn", it means that traffic regulation is involved a lot, meaning that either the server's maxconn value is too low, or that there aren't enough servers to process the load with an optimal response time. When only one of the server's "srv_conn" is high, it usually means that this server has some trouble causing the connections to take longer to be processed than on other servers.

- "retries" is the number of connection retries experienced by this session when trying to connect to the server. It must normally be zero, unless a server is being stopped at the same moment the connection was attempted. Frequent retries generally indicate either a network problem between haproxy and the server, or a misconfigured system backlog on the server preventing new connections from being queued. This field may optionally be prefixed with a '+' sign, indicating that the session has experienced a redispatch after the maximal retry count has been reached on the initial server. In this case, the server name appearing in the log is the one the connection was redispatched to, and not the first one, though both may sometimes be the same in case of hashing for instance. So as a general rule of thumb, when a '+' is present in front of the retry count, this count should not be attributed to the logged server.

- "srv_queue" is the total number of requests which were processed before this one in the server queue. It is zero when the request has not gone through the server queue. It makes it possible to estimate the approximate server's response time by dividing the time spent in queue by the number of requests in the queue. It is worth noting that if a session experiences a redispatch and passes through two server queues, their positions will be cumulated. A request should not pass through both the server queue and the backend queue unless a redispatch occurs.

- "backend_queue" is the total number of requests which were processed before this one in the backend's global queue. It is zero when the request has not gone through the global queue. It makes it possible to estimate the average queue length, which easily translates into a number of missing servers when divided by a server's "maxconn" parameter. It is worth noting that if a session experiences a redispatch, it may pass twice in the backend's queue, and then both positions will be cumulated. A request should not pass through both the server queue and the backend queue unless a redispatch occurs.

8.2.3. HTTP log format

The HTTP format is the most complete and the best suited for HTTP proxies. It is enabled by when "option httplog" is specified in the frontend. It provides the same level of information as the TCP format with additional features which are specific to the HTTP protocol. Just like the TCP format, the log is usually emitted at the end of the session, unless "option logasap" is specified, which generally only makes sense for download sites. A session which matches the "monitor" rules will never logged. It is also possible not to log sessions for which no data were sent by the client by specifying "option dontlognull" in the frontend. Successful connections will not be logged if "option dontlog-normal" is specified in the frontend.

Most fields are shared with the TCP log, some being different. A few fields may slightly vary depending on some configuration options. Those ones are marked with a star ('*') after the field name below.

```
  Example :
        frontend http-in
            mode http
            option httplog
            log global
            default_backend bck

        backend static
            server srv1 127.0.0.1:8000

    >>> Feb  6 12:14:14 localhost \
          haproxy[14389]: 10.0.1.2:33317 [06/Feb/2009:12:14:14.655] http-in \
          static/srv1 10/0/30/69/109 200 2750 - - ---- 1/1/1/1/0 0/0 {1wt.eu} \
          {} "GET /index.html HTTP/1.1"

  Field   Format                                Extract from the example above
      1   process_name '[' pid ']:'                            haproxy[14389]:
      2   client_ip ':' client_port                             10.0.1.2:33317
      3   '[' accept_date ']'                       [06/Feb/2009:12:14:14.655]
      4   frontend_name                                                http-in
      5   backend_name '/' server_name                             static/srv1
      6   Tq '/' Tw '/' Tc '/' Tr '/' Tt*                       10/0/30/69/109
      7   status_code                                                      200
      8   bytes_read*                                                     2750
      9   captured_request_cookie                                            -
     10   captured_response_cookie                                           -
     11   termination_state                                               ----
     12   actconn '/' feconn '/' beconn '/' srv_conn '/' retries*    1/1/1/1/0
     13   srv_queue '/' backend_queue                                      0/0
     14   '{' captured_request_headers* '}'                   {haproxy.1wt.eu}
     15   '{' captured_response_headers* '}'                                {}
     16   '"' http_request '"'                      "GET /index.html HTTP/1.1"
```

Detailed fields description :

- "client_ip" is the IP address of the client which initiated the TCP connection to haproxy. If the connection was accepted on a UNIX socket instead, the IP address would be replaced with the word "unix". Note that when the connection is accepted on a socket configured with "accept-proxy" and the PROXY protocol is correctly used, then the logs will reflect the forwarded connection's information.

- "client_port" is the TCP port of the client which initiated the connection. If the connection was accepted on a UNIX socket instead, the port would be replaced with the ID of the accepting socket, which is also reported in the stats interface.

- "accept_date" is the exact date when the TCP connection was received by haproxy (which might be very slightly different from the date observed on the network if there was some queuing in the system's backlog). This is usually the same date which may appear in any upstream firewall's log. This does not depend on the fact that the client has sent the request or not.

- "frontend_name" is the name of the frontend (or listener) which received and processed the connection.

- "backend_name" is the name of the backend (or listener) which was selected to manage the connection to the server. This will be the same as the frontend if no switching rule has been applied.

- "server_name" is the name of the last server to which the connection was sent, which might differ from the first one if there were connection errors and a redispatch occurred. Note that this server belongs to the backend which processed the request. If the request was aborted before reaching a server, "<NOSRV>" is indicated instead of a server name. If the request was intercepted by the stats subsystem, "<STATS>" is indicated instead.

- "Tq" is the total time in milliseconds spent waiting for the client to send a full HTTP request, not counting data. It can be "-1" if the connection was aborted before a complete request could be received. It should always be very small because a request generally fits in one single packet. Large times here generally indicate network trouble between the client and haproxy. See "Timers" below for more details.

- "Tw" is the total time in milliseconds spent waiting in the various queues. It can be "-1" if the connection was aborted before reaching the queue. See "Timers" below for more details.

- "Tc" is the total time in milliseconds spent waiting for the connection to establish to the final server, including retries. It can be "-1" if the request was aborted before a connection could be established. See "Timers" below for more details.

- "Tr" is the total time in milliseconds spent waiting for the server to send a full HTTP response, not counting data. It can be "-1" if the request was aborted before a complete response could be received. It generally matches the server's processing time for the request, though it may be altered by the amount of data sent by the client to the server. Large times here on "GET" requests generally indicate an overloaded server. See "Timers" below for more details.

- "Tt" is the total time in milliseconds elapsed between the accept and the last close. It covers all possible processing. There is one exception, if "option logasap" was specified, then the time counting stops at the moment the log is emitted. In this case, a '+' sign is prepended before the value, indicating that the final one will be larger. See "Timers" below for more details.

- "status_code" is the HTTP status code returned to the client. This status is generally set by the server, but it might also be set by haproxy when the server cannot be reached or when its response is blocked by haproxy.

- "bytes_read" is the total number of bytes transmitted to the client when the log is emitted. This does include HTTP headers. If "option logasap" is specified, the this value will be prefixed with a '+' sign indicating that the final one may be larger. Please note that this value is a 64-bit counter, so log analysis tools must be able to handle it without overflowing.

- "captured_request_cookie" is an optional "name=value" entry indicating that the client had this cookie in the request. The cookie name and its maximum length are defined by the "capture cookie" statement in the frontend configuration. The field is a single dash ('-') when the option is not set. Only one cookie may be captured, it is generally used to track session ID exchanges between a client and a server to detect session crossing between clients due to application bugs. For more details, please consult the section "Capturing HTTP headers and cookies" below.

- "captured_response_cookie" is an optional "name=value" entry indicating that the server has returned a cookie with its response. The cookie name and its maximum length are defined by the "capture cookie" statement in the frontend configuration. The field is a single dash ('-') when the option is not set. Only one cookie may be captured, it is generally used to track session ID exchanges between a client and a server to detect session crossing between clients due to application bugs. For more details, please consult the section "Capturing HTTP headers and cookies" below.

- "termination_state" is the condition the session was in when the session ended. This indicates the session state, which side caused the end of session to happen, for what reason (timeout, error, ...), just like in TCP logs, and information about persistence operations on cookies in the last two characters. The normal flags should begin with "--", indicating the session was closed by either end with no data remaining in buffers. See below "Session state at disconnection" for more details.

- "actconn" is the total number of concurrent connections on the process when the session was logged. It is useful to detect when some per-process system limits have been reached. For instance, if actconn is close to 512 or 1024 when multiple connection errors occur, chances are high that the system limits the process to use a maximum of 1024 file descriptors and that all of them are used. See section 3 "Global parameters" to find how to tune the system.

- "feconn" is the total number of concurrent connections on the frontend when the session was logged. It is useful to estimate the amount of resource required to sustain high loads, and to detect when the frontend's "maxconn" has been reached. Most often when this value increases by huge jumps, it is because there is congestion on the backend servers, but sometimes it can be caused by a denial of service attack.

- "beconn" is the total number of concurrent connections handled by the backend when the session was logged. It includes the total number of concurrent connections active on servers as well as the number of connections pending in queues. It is useful to estimate the amount of additional servers needed to support high loads for a given application. Most often when this value increases by huge jumps, it is because there is congestion on the backend servers, but sometimes it can be caused by a denial of service attack.

- "srv_conn" is the total number of concurrent connections still active on the server when the session was logged. It can never exceed the server's configured "maxconn" parameter. If this value is very often close or equal to the server's "maxconn", it means that traffic regulation is involved a lot, meaning that either the server's maxconn value is too low, or that there aren't enough servers to process the load with an optimal response time. When only one of the server's "srv_conn" is high, it usually means that this server has some trouble causing the requests to take longer to be processed than on other servers.

- "retries" is the number of connection retries experienced by this session when trying to connect to the server. It must normally be zero, unless a server is being stopped at the same moment the connection was attempted. Frequent retries generally indicate either a network problem between haproxy and the server, or a misconfigured system backlog on the server preventing new connections from being queued. This field may optionally be prefixed with a '+' sign, indicating that the session has experienced a

redispatch after the maximal retry count has been reached on the initial server. In this case, the server name appearing in the log is the one the connection was redispatched to, and not the first one, though both may sometimes be the same in case of hashing for instance. So as a general rule of thumb, when a '+' is present in front of the retry count, this count should not be attributed to the logged server.

- "srv_queue" is the total number of requests which were processed before this one in the server queue. It is zero when the request has not gone through the server queue. It makes it possible to estimate the approximate server's response time by dividing the time spent in queue by the number of requests in the queue. It is worth noting that if a session experiences a redispatch and passes through two server queues, their positions will be cumulated. A request should not pass through both the server queue and the backend queue unless a redispatch occurs.

- "backend_queue" is the total number of requests which were processed before this one in the backend's global queue. It is zero when the request has not gone through the global queue. It makes it possible to estimate the average queue length, which easily translates into a number of missing servers when divided by a server's "maxconn" parameter. It is worth noting that if a session experiences a redispatch, it may pass twice in the backend's queue, and then both positions will be cumulated. A request should not pass through both the server queue and the backend queue unless a redispatch occurs.

- "captured_request_headers" is a list of headers captured in the request due to the presence of the "capture request header" statement in the frontend. Multiple headers can be captured, they will be delimited by a vertical bar ('|'). When no capture is enabled, the braces do not appear, causing a shift of remaining fields. It is important to note that this field may contain spaces, and that using it requires a smarter log parser than when it's not used. Please consult the section "Capturing HTTP headers and cookies" below for more details.

- "captured_response_headers" is a list of headers captured in the response due to the presence of the "capture response header" statement in the frontend. Multiple headers can be captured, they will be delimited by a vertical bar ('|'). When no capture is enabled, the braces do not appear, causing a shift of remaining fields. It is important to note that this field may contain spaces, and that using it requires a smarter log parser than when it's not used. Please consult the section "Capturing HTTP headers and cookies" below for more details.

- "http_request" is the complete HTTP request line, including the method, request and HTTP version string. Non-printable characters are encoded (see below the section "Non-printable characters"). This is always the last field, and it is always delimited by quotes and is the only one which can contain quotes. If new fields are added to the log format, they will be added before this field. This field might be truncated if the request is huge and does not fit in the standard syslog buffer (1024 characters). This is the reason why this field must always remain the last one.

8.2.4. Custom log format

The directive log-format allows you to customize the logs in http mode and tcp mode. It takes a string as argument.

HAproxy understands some log format variables. % precedes log format variables. Variables can take arguments using braces ('{}'), and multiple arguments are separated by commas within the braces. Flags may be added or removed by prefixing them with a '+' or '-' sign.

Special variable "%o" may be used to propagate its flags to all other variables on the same format string. This is particularly handy with quoted string formats ("Q").

If a variable is named between square brackets ('[' .. ']') then it is used as a sample expression rule (see section 7.3). This it useful to add some less common information such as the client's SSL certificate's DN, or to log the key that would be used to store an entry into a stick table.

Note: spaces must be escaped. A space character is considered as a separator. In order to emit a verbatim '%', it must be preceded by another '%' resulting in '%%'. HAProxy will automatically merge consecutive separators.

Flags are :
  * Q: quote a string
  * X: hexadecimal representation (IPs, Ports, %Ts, %rt, %pid)

  Example:

```
    log-format %T\ %t\ Some\ Text
    log-format %{+Q}o\ %t\ %s\ %{-Q}r
```

At the moment, the default HTTP format is defined this way :

```
    log-format %ci:%cp\ [%t]\ %ft\ %b/%s\ %Tq/%Tw/%Tc/%Tr/%Tt\ %ST\ %B\ %CC\ \
               %CS\ %tsc\ %ac/%fc/%bc/%sc/%rc\ %sq/%bq\ %hr\ %hs\ %{+Q}r
```

the default CLF format is defined this way :

```
    log-format %{+Q}o\ %{-Q}ci\ -\ -\ [%T]\ %r\ %ST\ %B\ \"\"\ \"\"\ %cp\ \
               %ms\ %ft\ %b\ %s\ \%Tq\ %Tw\ %Tc\ %Tr\ %Tt\ %tsc\ %ac\ %fc\ \
               %bc\ %sc\ %rc\ %sq\ %bq\ %CC\ %CS\ \%hrl\ %hsl
```

and the default TCP format is defined this way :

```
    log-format %ci:%cp\ [%t]\ %ft\ %b/%s\ %Tw/%Tc/%Tt\ %B\ %ts\ \
               %ac/%fc/%bc/%sc/%rc\ %sq/%bq
```

Please refer to the table below for currently defined variables :

R	var	field name (8.2.2 and 8.2.3 for description)	type
	%o	special variable, apply flags on all next var	
	%B	bytes_read (from server to client)	numeric
H	%CC	captured_request_cookie	string
H	%CS	captured_response_cookie	string
	%H	hostname	string
H	%HM	HTTP method (ex: POST)	string
H	%HP	HTTP request URI without query string (path)	string
H	%HQ	HTTP request URI query string (ex: ?bar=baz)	string
H	%HU	HTTP request URI (ex: /foo?bar=baz)	string
H	%HV	HTTP version (ex: HTTP/1.0)	string
	%ID	unique-id	string
	%ST	status_code	numeric
	%T	gmt_date_time	date
	%Tc	Tc	numeric
	%Tl	local_date_time	date
H	%Tq	Tq	numeric
H	%Tr	Tr	numeric
	%Ts	timestamp	numeric
	%Tt	Tt	numeric
	%Tw	Tw	numeric
	%U	bytes_uploaded (from client to server)	numeric
	%ac	actconn	numeric
	%b	backend_name	string
	%bc	beconn (backend concurrent connections)	numeric
	%bi	backend_source_ip (connecting address)	IP
	%bp	backend_source_port (connecting address)	numeric
	%bq	backend_queue	numeric
	%ci	client_ip (accepted address)	IP
	%cp	client_port (accepted address)	numeric
	%f	frontend_name	string
	%fc	feconn (frontend concurrent connections)	numeric
	%fi	frontend_ip (accepting address)	IP
	%fp	frontend_port (accepting address)	numeric
	%ft	frontend_name_transport ('~' suffix for SSL)	string
	%lc	frontend_log_counter	numeric
	%hr	captured_request_headers default style	string
	%hrl	captured_request_headers CLF style	string list
	%hs	captured_response_headers default style	string
	%hsl	captured_response_headers CLF style	string list
	%ms	accept date milliseconds (left-padded with 0)	numeric
	%pid	PID	numeric
H	%r	http_request	string
	%rc	retries	numeric
	%rt	request_counter (HTTP req or TCP session)	numeric
	%s	server_name	string
	%sc	srv_conn (server concurrent connections)	numeric
	%si	server_IP (target address)	IP
	%sp	server_port (target address)	numeric
	%sq	srv_queue	numeric
S	%sslc	ssl_ciphers (ex: AES-SHA)	string
S	%sslv	ssl_version (ex: TLSv1)	string
	%t	date_time (with millisecond resolution)	date
	%ts	termination_state	string
H	%tsc	termination_state with cookie status	string

  R = Restrictions : H = mode http only ; S = SSL only

8.2.5. Error log format

When an incoming connection fails due to an SSL handshake or an invalid PROXY protocol header, haproxy will log the event using a shorter, fixed line format. By default, logs are emitted at the LOG_INFO level, unless the option "log-separate-errors" is set in the backend, in which case the LOG_ERR level will be used. Connections on which no data are exchanged (eg: probes) are not logged if the "dontlognull" option is set.

The format looks like this :

```
  >>> Dec  3 18:27:14 localhost \
        haproxy[6103]: 127.0.0.1:56059 [03/Dec/2012:17:35:10.380] frt/f1: \
        Connection error during SSL handshake
```

Field	Format	Extract from the example above
1	process_name '[' pid ']:'	haproxy[6103]:
2	client_ip ':' client_port	127.0.0.1:56059
3	'[' accept_date ']'	[03/Dec/2012:17:35:10.380]
4	frontend_name "/" bind_name ":"	frt/f1:
5	message	Connection error during SSL handshake

These fields just provide minimal information to help debugging connection failures.

8.3. Advanced logging options

Some advanced logging options are often looked for but are not easy to find out just by looking at the various options. Here is an entry point for the few options which can enable better logging. Please refer to the keywords reference for more information about their usage.

8.3.1. Disabling logging of external tests

It is quite common to have some monitoring tools perform health checks on haproxy. Sometimes it will be a layer 3 load-balancer such as LVS or any commercial load-balancer, and sometimes it will simply be a more complete monitoring system such as Nagios. When the tests are very frequent, users often ask how to disable logging for those checks. There are three possibilities :

- if connections come from everywhere and are just TCP probes, it is often desired to simply disable logging of connections without data exchange, by setting "option dontlognull" in the frontend. It also disables logging of port scans, which may or may not be desired.

- if the connection come from a known source network, use "monitor-net" to declare this network as monitoring only. Any host in this network will then only be able to perform health checks, and their requests will not be logged. This is generally appropriate to designate a list of equipment such as other load-balancers.

- if the tests are performed on a known URI, use "monitor-uri" to declare this URI as dedicated to monitoring. Any host sending this request will only get the result of a health-check, and the request will not be logged.

8.3.2. Logging before waiting for the session to terminate

The problem with logging at end of connection is that you have no clue about what is happening during very long sessions, such as remote terminal sessions or large file downloads. This problem can be worked around by specifying "option logasap" in the frontend. Haproxy will then log as soon as possible, just before data transfer begins. This means that in case of TCP, it will still log the connection status to the server, and in case of HTTP, it will log just after processing the server headers. In this case, the number of bytes reported is the number of header bytes sent to the client. In order to avoid confusion with normal logs, the total time field and the number of bytes are prefixed with a '+' sign which means that real numbers are certainly larger.

8.3.3. Raising log level upon errors

Sometimes it is more convenient to separate normal traffic from errors logs, for instance in order to ease error monitoring from log files. When the option "log-separate-errors" is used, connections which experience errors, timeouts, retries, redispatches or HTTP status codes 5xx will see their syslog level raised from "info" to "err". This will help a syslog daemon store the log in a separate file. It is very important to keep the errors in the normal traffic file too, so that log ordering is not altered. You should also be careful if you already have configured your syslog daemon to store all logs higher than "notice" in an "admin" file, because the "err" level is higher than "notice".

8.3.4. Disabling logging of successful connections

Although this may sound strange at first, some large sites have to deal with multiple thousands of logs per second and are experiencing difficulties keeping them intact for a long time or detecting errors within them. If the option "dontlog-normal" is set on the frontend, all normal connections will not be logged. In this regard, a normal connection is defined as one without any error, timeout, retry nor redispatch. In HTTP, the status code is checked too, and a response with a status 5xx is not considered normal and will be logged too. Of course, doing is is really discouraged as it will remove most of the useful information from the logs. Do this only if you have no other alternative.

8.4. Timing events

Timers provide a great help in troubleshooting network problems. All values are reported in milliseconds (ms). These timers should be used in conjunction with the session termination flags. In TCP mode with "option tcplog" set on the frontend, 3 control points are reported under the form "Tw/Tc/Tt", and in HTTP mode, 5 control points are reported under the form "Tq/Tw/Tc/Tr/Tt" :

- Tq: total time to get the client request (HTTP mode only). It's the time elapsed between the moment the client connection was accepted and the moment the proxy received the last HTTP header. The value "-1" indicates that the end of headers (empty line) has never been seen. This happens when the client closes prematurely or times out.

- Tw: total time spent in the queues waiting for a connection slot. It accounts for backend queue as well as the server queues, and depends on the queue size, and the time needed for the server to complete previous requests. The value "-1" means that the request was killed before reaching the queue, which is generally what happens with invalid or denied requests.

- Tc: total time to establish the TCP connection to the server. It's the time elapsed between the moment the proxy sent the connection request, and the moment it was acknowledged by the server, or between the TCP SYN packet and the matching SYN/ACK packet in return. The value "-1" means that the connection never established.

- Tr: server response time (HTTP mode only). It's the time elapsed between the moment the TCP connection was established to the server and the moment the server sent its complete response headers. It purely shows its request processing time, without the network overhead due to the data transmission. It is worth noting that when the client has data to send to the server, for instance during a POST request, the time already runs, and this can distort apparent response time. For this reason, it's generally wise not to trust too much this field for POST requests initiated from clients behind an untrusted network. A value of "-1" here means that the last the response header (empty line) was never seen, most likely because the server timeout stroke before the server managed to process the request.

- Tt: total session duration time, between the moment the proxy accepted it and the moment both ends were closed. The exception is when the "logasap" option is specified. In this case, it only equals (Tq+Tw+Tc+Tr), and is prefixed with a '+' sign. From this field, we can deduce "Td", the data transmission time, by subtracting other timers when valid :

  ```
  Td = Tt - (Tq + Tw + Tc + Tr)
  ```

  Timers with "-1" values have to be excluded from this equation. In TCP mode, "Tq" and "Tr" have to be excluded too. Note that "Tt" can never be negative.

These timers provide precious indications on trouble causes. Since the TCP

```
protocol defines retransmit delays of 3, 6, 12... seconds, we know for sure
that timers close to multiples of 3s are nearly always related to lost packets
due to network problems (wires, negotiation, congestion). Moreover, if "Tt" is
close to a timeout value specified in the configuration, it often means that a
session has been aborted on timeout.

Most common cases :

  - If "Tq" is close to 3000, a packet has probably been lost between the
    client and the proxy. This is very rare on local networks but might happen
    when clients are on far remote networks and send large requests. It may
    happen that values larger than usual appear here without any network cause.
    Sometimes, during an attack or just after a resource starvation has ended,
    haproxy may accept thousands of connections in a few milliseconds. The time
    spent accepting these connections will inevitably slightly delay processing
    of other connections, and it can happen that request times in the order of
    a few tens of milliseconds are measured after a few thousands of new
    connections have been accepted at once. Setting "option http-server-close"
    may display larger request times since "Tq" also measures the time spent
    waiting for additional requests.

  - If "Tc" is close to 3000, a packet has probably been lost between the
    server and the proxy during the server connection phase. This value should
    always be very low, such as 1 ms on local networks and less than a few tens
    of ms on remote networks.

  - If "Tr" is nearly always lower than 3000 except some rare values which seem
    to be the average majored by 3000, there are probably some packets lost
    between the proxy and the server.

  - If "Tt" is large even for small byte counts, it generally is because
    neither the client nor the server decides to close the connection, for
    instance because both have agreed on a keep-alive connection mode. In order
    to solve this issue, it will be needed to specify "option httpclose" on
    either the frontend or the backend. If the problem persists, it means that
    the server ignores the "close" connection mode and expects the client to
    close. Then it will be required to use "option forceclose". Having the
    smallest possible 'Tt' is important when connection regulation is used with
    the "maxconn" option on the servers, since no new connection will be sent
    to the server until another one is released.

Other noticeable HTTP log cases ('xx' means any value to be ignored) :

  Tq/Tw/Tc/Tr/+Tt  The "option logasap" is present on the frontend and the log
                   was emitted before the data phase. All the timers are valid
                   except "Tt" which is shorter than reality.

  -1/xx/xx/xx/Tt   The client was not able to send a complete request in time
                   or it aborted too early. Check the session termination flags
                   then "timeout http-request" and "timeout client" settings.

  Tq/-1/xx/xx/Tt   It was not possible to process the request, maybe because
                   servers were out of order, because the request was invalid
                   or forbidden by ACL rules. Check the session termination
                   flags.

  Tq/Tw/-1/xx/Tt   The connection could not establish on the server. Either it
                   actively refused it or it timed out after Tt-(Tq+Tw) ms.
                   Check the session termination flags, then check the
                   "timeout connect" setting. Note that the tarpit action might
                   return similar-looking patterns, with "Tw" equal to the time
                   the client connection was maintained open.

  Tq/Tw/Tc/-1/Tt   The server has accepted the connection but did not return
                   a complete response in time, or it closed its connection
```

```
                   unexpectedly after Tt-(Tq+Tw+Tc) ms. Check the session
                   termination flags, then check the "timeout server" setting.


8.5. Session state at disconnection
-----------------------------------

TCP and HTTP logs provide a session termination indicator in the
"termination_state" field, just before the number of active connections. It is
2-characters long in TCP mode, and is extended to 4 characters in HTTP mode,
each of which has a special meaning :

  - On the first character, a code reporting the first event which caused the
    session to terminate :

        C : the TCP session was unexpectedly aborted by the client.

        S : the TCP session was unexpectedly aborted by the server, or the
            server explicitly refused it.

        P : the session was prematurely aborted by the proxy, because of a
            connection limit enforcement, because a DENY filter was matched,
            because of a security check which detected and blocked a dangerous
            error in server response which might have caused information leak
            (eg: cacheable cookie).

        L : the session was locally processed by haproxy and was not passed to
            a server. This is what happens for stats and redirects.

        R : a resource on the proxy has been exhausted (memory, sockets, source
            ports, ...). Usually, this appears during the connection phase, and
            system logs should contain a copy of the precise error. If this
            happens, it must be considered as a very serious anomaly which
            should be fixed as soon as possible by any means.

        I : an internal error was identified by the proxy during a self-check.
            This should NEVER happen, and you are encouraged to report any log
            containing this, because this would almost certainly be a bug. It
            would be wise to preventively restart the process after such an
            event too, in case it would be caused by memory corruption.

        D : the session was killed by haproxy because the server was detected
            as down and was configured to kill all connections when going down.

        U : the session was killed by haproxy on this backup server because an
            active server was detected as up and was configured to kill all
            backup connections when going up.

        K : the session was actively killed by an admin operating on haproxy.

        c : the client-side timeout expired while waiting for the client to
            send or receive data.

        s : the server-side timeout expired while waiting for the server to
            send or receive data.

        - : normal session completion, both the client and the server closed
            with nothing left in the buffers.

  - on the second character, the TCP or HTTP session state when it was closed :

        R : the proxy was waiting for a complete, valid REQUEST from the client
            (HTTP mode only). Nothing was sent to any server.

        Q : the proxy was waiting in the QUEUE for a connection slot. This can
```

```
              only happen when servers have a 'maxconn' parameter set. It can
              also happen in the global queue after a redispatch consecutive to
              a failed attempt to connect to a dying server. If no redispatch is
              reported, then no connection attempt was made to any server.

          C : the proxy was waiting for the CONNECTION to establish on the
              server. The server might at most have noticed a connection attempt.

          H : the proxy was waiting for complete, valid response HEADERS from the
              server (HTTP only).

          D : the session was in the DATA phase.

          L : the proxy was still transmitting LAST data to the client while the
              server had already finished. This one is very rare as it can only
              happen when the client dies while receiving the last packets.

          T : the request was tarpitted. It has been held open with the client
              during the whole "timeout tarpit" duration or until the client
              closed, both of which will be reported in the "Tw" timer.

          - : normal session completion after end of data transfer.

  - the third character tells whether the persistence cookie was provided by
    the client (only in HTTP mode) :

          N : the client provided NO cookie. This is usually the case for new
              visitors, so counting the number of occurrences of this flag in the
              logs generally indicate a valid trend for the site frequentation.

          I : the client provided an INVALID cookie matching no known server.
              This might be caused by a recent configuration change, mixed
              cookies between HTTP/HTTPS sites, persistence conditionally
              ignored, or an attack.

          D : the client provided a cookie designating a server which was DOWN,
              so either "option persist" was used and the client was sent to
              this server, or it was not set and the client was redispatched to
              another server.

          V : the client provided a VALID cookie, and was sent to the associated
              server.

          E : the client provided a valid cookie, but with a last date which was
              older than what is allowed by the "maxidle" cookie parameter, so
              the cookie is consider EXPIRED and is ignored. The request will be
              redispatched just as if there was no cookie.

          O : the client provided a valid cookie, but with a first date which was
              older than what is allowed by the "maxlife" cookie parameter, so
              the cookie is consider too OLD and is ignored. The request will be
              redispatched just as if there was no cookie.

          U : a cookie was present but was not used to select the server because
              some other server selection mechanism was used instead (typically a
              "use-server" rule).

          - : does not apply (no cookie set in configuration).

  - the last character reports what operations were performed on the persistence
    cookie returned by the server (only in HTTP mode) :

          N : NO cookie was provided by the server, and none was inserted either.

          I : no cookie was provided by the server, and the proxy INSERTED one.
```

```
              Note that in "cookie insert" mode, if the server provides a cookie,
              it will still be overwritten and reported as "I" here.

          U : the proxy UPDATED the last date in the cookie that was presented by
              the client. This can only happen in insert mode with "maxidle". It
              happens every time there is activity at a different date than the
              date indicated in the cookie. If any other change happens, such as
              a redispatch, then the cookie will be marked as inserted instead.

          P : a cookie was PROVIDED by the server and transmitted as-is.

          R : the cookie provided by the server was REWRITTEN by the proxy, which
              happens in "cookie rewrite" or "cookie prefix" modes.

          D : the cookie provided by the server was DELETED by the proxy.

          - : does not apply (no cookie set in configuration).

The combination of the two first flags gives a lot of information about what
was happening when the session terminated, and why it did terminate. It can be
helpful to detect server saturation, network troubles, local system resource
starvation, attacks, etc...

The most common termination flags combinations are indicated below. They are
alphabetically sorted, with the lowercase set just after the upper case for
easier finding and understanding.

  Flags   Reason

     --   Normal termination.

     CC   The client aborted before the connection could be established to the
          server. This can happen when haproxy tries to connect to a recently
          dead (or unchecked) server, and the client aborts while haproxy is
          waiting for the server to respond or for "timeout connect" to expire.

     CD   The client unexpectedly aborted during data transfer. This can be
          caused by a browser crash, by an intermediate equipment between the
          client and haproxy which decided to actively break the connection,
          by network routing issues between the client and haproxy, or by a
          keep-alive session between the server and the client terminated first
          by the client.

     cD   The client did not send nor acknowledge any data for as long as the
          "timeout client" delay. This is often caused by network failures on
          the client side, or the client simply leaving the net uncleanly.

     CH   The client aborted while waiting for the server to start responding.
          It might be the server taking too long to respond or the client
          clicking the 'Stop' button too fast.

     cH   The "timeout client" stroke while waiting for client data during a
          POST request. This is sometimes caused by too large TCP MSS values
          for PPPoE networks which cannot transport full-sized packets. It can
          also happen when client timeout is smaller than server timeout and
          the server takes too long to respond.

     CQ   The client aborted while its session was queued, waiting for a server
          with enough empty slots to accept it. It might be that either all the
          servers were saturated or that the assigned server was taking too
          long a time to respond.

     CR   The client aborted before sending a full HTTP request. Most likely
          the request was typed by hand using a telnet client, and aborted
          too early. The HTTP status code is likely a 400 here. Sometimes this
```

might also be caused by an IDS killing the connection between haproxy and the client. "option http-ignore-probes" can be used to ignore connections without any data transfer.

cR The "timeout http-request" stroke before the client sent a full HTTP request. This is sometimes caused by too large TCP MSS values on the client side for PPPoE networks which cannot transport full-sized packets, or by clients sending requests by hand and not typing fast enough, or forgetting to enter the empty line at the end of the request. The HTTP status code is likely a 408 here. Note: recently, some browsers started to implement a "pre-connect" feature consisting in speculatively connecting to some recently visited web sites just in case the user would like to visit them. This results in many connections being established to web sites, which end up in 408 Request Timeout if the timeout strikes first, or 400 Bad Request when the browser decides to close them first. These ones pollute the log and feed the error counters. Some versions of some browsers have even been reported to display the error code. It is possible to work around the undesirable effects of this behaviour by adding "option http-ignore-probes" in the frontend, resulting in connections with zero data transfer to be totally ignored. This will definitely hide the errors of people experiencing connectivity issues though.

CT The client aborted while its session was tarpitted. It is important to check if this happens on valid requests, in order to be sure that no wrong tarpit rules have been written. If a lot of them happen, it might make sense to lower the "timeout tarpit" value to something closer to the average reported "Tw" timer, in order not to consume resources for just a few attackers.

LR The request was intercepted and locally handled by haproxy. Generally it means that this was a redirect or a stats request.

SC The server or an equipment between it and haproxy explicitly refused the TCP connection (the proxy received a TCP RST or an ICMP message in return). Under some circumstances, it can also be the network stack telling the proxy that the server is unreachable (eg: no route, or no ARP response on local network). When this happens in HTTP mode, the status code is likely a 502 or 503 here.

sC The "timeout connect" stroke before a connection to the server could complete. When this happens in HTTP mode, the status code is likely a 503 or 504 here.

SD The connection to the server died with an error during the data transfer. This usually means that haproxy has received an RST from the server or an ICMP message from an intermediate equipment while exchanging data with the server. This can be caused by a server crash or by a network issue on an intermediate equipment.

sD The server did not send nor acknowledge any data for as long as the "timeout server" setting during the data phase. This is often caused by too short timeouts on L4 equipments before the server (firewalls, load-balancers, ...), as well as keep-alive sessions maintained between the client and the server expiring first on haproxy.

SH The server aborted before sending its full HTTP response headers, or it crashed while processing the request. Since a server aborting at this moment is very rare, it would be wise to inspect its logs to control whether it crashed and why. The logged request may indicate a small set of faulty requests, demonstrating bugs in the application. Sometimes this might also be caused by an IDS killing the connection between haproxy and the server.

sH The "timeout server" stroke before the server could return its

response headers. This is the most common anomaly, indicating too long transactions, probably caused by server or database saturation. The immediate workaround consists in increasing the "timeout server" setting, but it is important to keep in mind that the user experience will suffer from these long response times. The only long term solution is to fix the application.

sQ The session spent too much time in queue and has been expired. See the "timeout queue" and "timeout connect" settings to find out how to fix this if it happens too often. If it often happens massively in short periods, it may indicate general problems on the affected servers due to I/O or database congestion, or saturation caused by external attacks.

PC The proxy refused to establish a connection to the server because the process' socket limit has been reached while attempting to connect. The global "maxconn" parameter may be increased in the configuration so that it does not happen anymore. This status is very rare and might happen when the global "ulimit-n" parameter is forced by hand.

PD The proxy blocked an incorrectly formatted chunked encoded message in a request or a response, after the server has emitted its headers. In most cases, this will indicate an invalid message from the server to the client. Haproxy supports chunk sizes of up to 2GB - 1 (2147483647 bytes). Any larger size will be considered as an error.

PH The proxy blocked the server's response, because it was invalid, incomplete, dangerous (cache control), or matched a security filter. In any case, an HTTP 502 error is sent to the client. One possible cause for this error is an invalid syntax in an HTTP header name containing unauthorized characters. It is also possible but quite rare, that the proxy blocked a chunked-encoding request from the client due to an invalid syntax, before the server responded. In this case, an HTTP 400 error is sent to the client and reported in the logs.

PR The proxy blocked the client's HTTP request, either because of an invalid HTTP syntax, in which case it returned an HTTP 400 error to the client, or because a deny filter matched, in which case it returned an HTTP 403 error.

PT The proxy blocked the client's request and has tarpitted its connection before returning it a 500 server error. Nothing was sent to the server. The connection was maintained open for as long as reported by the "Tw" timer field.

RC A local resource has been exhausted (memory, sockets, source ports) preventing the connection to the server from establishing. The error logs will tell precisely what was missing. This is very rare and can only be solved by proper system tuning.

The combination of the two last flags gives a lot of information about how persistence was handled by the client, the server and by haproxy. This is very important to troubleshoot disconnections, when users complain they have to re-authenticate. The commonly encountered flags are :

-- Persistence cookie is not enabled.

NN No cookie was provided by the client, none was inserted in the response. For instance, this can be in insert mode with "postonly" set on a GET request.

II A cookie designating an invalid server was provided by the client, a valid one was inserted in the response. This typically happens when a "server" entry is removed from the configuration, since its cookie value can be presented by a client when no other server knows it.

NI   No cookie was provided by the client, one was inserted in the response. This typically happens for first requests from every user in "insert" mode, which makes it an easy way to count real users.

VN   A cookie was provided by the client, none was inserted in the response. This happens for most responses for which the client has already got a cookie.

VU   A cookie was provided by the client, with a last visit date which is not completely up-to-date, so an updated cookie was provided in response. This can also happen if there was no date at all, or if there was a date but the "maxidle" parameter was not set, so that the cookie can be switched to unlimited time.

EI   A cookie was provided by the client, with a last visit date which is too old for the "maxidle" parameter, so the cookie was ignored and a new cookie was inserted in the response.

OI   A cookie was provided by the client, with a first visit date which is too old for the "maxlife" parameter, so the cookie was ignored and a new cookie was inserted in the response.

DI   The server designated by the cookie was down, a new server was selected and a new cookie was emitted in the response.

VI   The server designated by the cookie was not marked dead but could not be reached. A redispatch happened and selected another one, which was then advertised in the response.

8.6. Non-printable characters

In order not to cause trouble to log analysis tools or terminals during log consulting, non-printable characters are not sent as-is into log files, but are converted to the two-digits hexadecimal representation of their ASCII code, prefixed by the character '#'. The only characters that can be logged without being escaped are comprised between 32 and 126 (inclusive). Obviously, the escape character '#' itself is also encoded to avoid any ambiguity ("#23"). It is the same for the character '"' which becomes "#22", as well as '{', '|' and '}' when logging headers.

Note that the space character (' ') is not encoded in headers, which can cause issues for tools relying on space count to locate fields. A typical header containing spaces is "User-Agent".

Last, it has been observed that some syslog daemons such as syslog-ng escape the quote ('"') with a backslash ('\'). The reverse operation can safely be performed since no quote may appear anywhere else in the logs.

8.7. Capturing HTTP cookies

Cookie capture simplifies the tracking a complete user session. This can be achieved using the "capture cookie" statement in the frontend. Please refer to section 4.2 for more details. Only one cookie can be captured, and the same cookie will simultaneously be checked in the request ("Cookie:" header) and in the response ("Set-Cookie:" header). The respective values will be reported in the HTTP logs at the "captured_request_cookie" and "captured_response_cookie" locations (see section 8.2.3 about HTTP log format). When either cookie is not seen, a dash ('-') replaces the value. This way, it's easy to detect when a user switches to a new session for example, because the server will reassign it

a new cookie. It is also possible to detect if a server unexpectedly sets a wrong cookie to a client, leading to session crossing.

```
  Examples :
        # capture the first cookie whose name starts with "ASPSESSION"
        capture cookie ASPSESSION len 32

        # capture the first cookie whose name is exactly "vgnvisitor"
        capture cookie vgnvisitor= len 32
```

8.8. Capturing HTTP headers

Header captures are useful to track unique request identifiers set by an upper proxy, virtual host names, user-agents, POST content-length, referrers, etc. In the response, one can search for information about the response length, how the server asked the cache to behave, or an object location during a redirection.

Header captures are performed using the "capture request header" and "capture response header" statements in the frontend. Please consult their definition in section 4.2 for more details.

It is possible to include both request headers and response headers at the same time. Non-existent headers are logged as empty strings, and if one header appears more than once, only its last occurrence will be logged. Request headers are grouped within braces '{' and '}' in the same order as they were declared, and delimited with a vertical bar '|' without any space. Response headers follow the same representation, but are displayed after a space following the request headers block. These blocks are displayed just before the HTTP request in the logs.

As a special case, it is possible to specify an HTTP header capture in a TCP frontend. The purpose is to enable logging of headers which will be parsed in an HTTP backend if the request is then switched to this HTTP backend.

```
  Example :
        # This instance chains to the outgoing proxy
        listen proxy-out
            mode http
            option httplog
            option logasap
            log global
            server cache1 192.168.1.1:3128

            # log the name of the virtual server
            capture request  header Host len 20

            # log the amount of data uploaded during a POST
            capture request  header Content-Length len 10

            # log the beginning of the referrer
            capture request  header Referer len 20

            # server name (useful for outgoing proxies only)
            capture response header Server len 20

            # logging the content-length is useful with "option logasap"
            capture response header Content-Length len 10

            # log the expected cache behaviour on the response
            capture response header Cache-Control len 8

            # the Via header will report the next proxy's name
            capture response header Via len 20
            # log the URL location during a redirection
            capture response header Location len 20

    >>> Aug  9 20:26:09 localhost \
          haproxy[2022]: 127.0.0.1:34014 [09/Aug/2004:20:26:09] proxy-out \
          proxy-out/cache1 0/0/0/162/+162 200 +350 - - ---- 0/0/0/0/0 0/0 \
          {fr.adserver.yahoo.co||http://fr.f416.mail.} {|864|private||} \
          "GET http://fr.adserver.yahoo.com/"

    >>> Aug  9 20:30:46 localhost \
          haproxy[2022]: 127.0.0.1:34020 [09/Aug/2004:20:30:46] proxy-out \
          proxy-out/cache1 0/0/0/182/+182 200 +279 - - ---- 0/0/0/0/0 0/0 \
          {w.ods.org||} {Formilux/0.1.8|3495|||} \
          "GET http://trafic.1wt.eu/ HTTP/1.1"

    >>> Aug  9 20:30:46 localhost \
          haproxy[2022]: 127.0.0.1:34028 [09/Aug/2004:20:30:46] proxy-out \
          proxy-out/cache1 0/0/2/126/+128 301 +223 - - ---- 0/0/0/0/0 0/0 \
          {www.sytadin.equipement.gouv.fr||http://trafic.1wt.eu/} \
          {Apache|230|||http://www.sytadin.} \
          "GET http://www.sytadin.equipement.gouv.fr/ HTTP/1.1"
```

8.9. Examples of logs

These are real-world examples of logs accompanied with an explanation. Some of them have been made up by hand. The syslog part has been removed for better reading. Their sole purpose is to explain how to decipher them.

```
    >>> haproxy[674]: 127.0.0.1:33318 [15/Oct/2003:08:31:57.130] px-http \
          px-http/srv1 6559/0/7/147/6723 200 243 - - ---- 5/3/3/1/0 0/0 \
          "HEAD / HTTP/1.0"

    => long request (6.5s) entered by hand through 'telnet'. The server replied
       in 147 ms, and the session ended normally ('----')

    >>> haproxy[674]: 127.0.0.1:33319 [15/Oct/2003:08:31:57.149] px-http \
          px-http/srv1 6559/1230/7/147/6870 200 243 - - ---- 324/239/239/99/0 \
          0/9 "HEAD / HTTP/1.0"

    => Idem, but the request was queued in the global queue behind 9 other
       requests, and waited there for 1230 ms.

    >>> haproxy[674]: 127.0.0.1:33320 [15/Oct/2003:08:32:17.654] px-http \
          px-http/srv1 9/0/7/14/+30 200 +243 - - ---- 3/3/3/1/0 0/0 \
          "GET /image.iso HTTP/1.0"

    => request for a long data transfer. The "logasap" option was specified, so
       the log was produced just before transferring data. The server replied in
       14 ms, 243 bytes of headers were sent to the client, and total time from
       accept to first data byte is 30 ms.

    >>> haproxy[674]: 127.0.0.1:33320 [15/Oct/2003:08:32:17.925] px-http \
          px-http/srv1 9/0/7/14/30 502 243 - - PH-- 3/2/2/0/0 0/0 \
          "GET /cgi-bin/bug.cgi? HTTP/1.0"

    => the proxy blocked a server response either because of an "rspdeny" or
       "rspideny" filter, or because the response was improperly formatted and
       not HTTP-compliant, or because it blocked sensitive information which
       risked being cached. In this case, the response is replaced with a "502
       bad gateway". The flags ("PH--") tell us that it was haproxy who decided
       to return the 502 and not the server.
```

```
    >>> haproxy[18113]: 127.0.0.1:34548 [15/Oct/2003:15:18:55.798] px-http \
          px-http/<NOSRV> -1/-1/-1/-1/8490 -1 0 - - CR-- 2/2/2/0/0 0/0 ""

    => the client never completed its request and aborted itself ("C---") after
       8.5s, while the proxy was waiting for the request headers ("-R--").
       Nothing was sent to any server.

    >>> haproxy[18113]: 127.0.0.1:34549 [15/Oct/2003:15:19:06.103] px-http \
         px-http/<NOSRV> -1/-1/-1/-1/50001 408 0 - - cR-- 2/2/2/0/0 0/0 ""

    => The client never completed its request, which was aborted by the
       time-out ("c---") after 50s, while the proxy was waiting for the request
       headers ("-R--").  Nothing was sent to any server, but the proxy could
       send a 408 return code to the client.

    >>> haproxy[18989]: 127.0.0.1:34550 [15/Oct/2003:15:24:28.312] px-tcp \
          px-tcp/srv1 0/0/5007 0 cD 0/0/0/0/0 0/0

    => This log was produced with "option tcplog". The client timed out after
       5 seconds ("c----").

    >>> haproxy[18989]: 10.0.0.1:34552 [15/Oct/2003:15:26:31.462] px-http \
          px-http/srv1 3183/-1/-1/-1/11215 503 0 - - SC-- 205/202/202/115/3 \
          0/0 "HEAD / HTTP/1.0"

    => The request took 3s to complete (probably a network problem), and the
       connection to the server failed ('SC--') after 4 attempts of 2 seconds
       (config says 'retries 3'), and no redispatch (otherwise we would have
       seen "/+3"). Status code 503 was returned to the client. There were 115
       connections on this server, 202 connections on this proxy, and 205 on
       the global process. It is possible that the server refused the
       connection because of too many already established.


/*
 * Local variables:
 *  fill-column: 79
 * End:
 */
```