```
                         -------------------
                               HAProxy
                         Architecture  Guide
                         -------------------
                           version 1.2.18
                            willy tarreau
                             2008/05/25
```

This document provides real world examples with working configurations.
Please note that except stated otherwise, global configuration parameters
such as logging, chrooting, limits and time-outs are not described here.

```
===================================================
1. Simple HTTP load-balancing with cookie insertion
===================================================
```

A web application often saturates the front-end server with high CPU loads,
due to the scripting language involved. It also relies on a back-end database
which is not much loaded. User contexts are stored on the server itself, and
not in the database, so that simply adding another server with simple IP/TCP
load-balancing would not work.

```
                +-------+
                |clients|  clients and/or reverse-proxy
                +---+---+
                    |
                   -+-----+--------+----
                          |       _|_db
                       +--+--+   (___)
                       | web |   (___)
                       +-----+   (___)
                   192.168.1.1   192.168.1.2
```

Replacing the web server with a bigger SMP system would cost much more than
adding low-cost pizza boxes. The solution is to buy N cheap boxes and install
the application on them. Install haproxy on the old one which will spread the
load across the new boxes.

```
  192.168.1.1    192.168.1.11-192.168.1.14   192.168.1.2
 -------+-----------+-----+-----+-----+--------+----
        |           |     |     |     |       _|_db
     +--+--+      +-+-+ +-+-+ +-+-+ +-+-+    (___)
     | LB1 |      | A | | B | | C | | D |    (___)
     +-----+      +---+ +---+ +---+ +---+    (___)
     haproxy        4 cheap web servers
```

Config on haproxy (LB1) :

```
    listen webfarm 192.168.1.1:80
       mode http
       balance roundrobin
       cookie SERVERID insert indirect
       option httpchk HEAD /index.html HTTP/1.0
       server webA 192.168.1.11:80 cookie A check
       server webB 192.168.1.12:80 cookie B check
       server webC 192.168.1.13:80 cookie C check
       server webD 192.168.1.14:80 cookie D check
```

Description :

 - LB1 will receive clients requests.
 - if a request does not contain a cookie, it will be forwarded to a valid
   server
 - in return, a cookie "SERVERID" will be inserted in the response holding the

   server name (eg: "A").
 - when the client comes again with the cookie "SERVERID=A", LB1 will know that
   it must be forwarded to server A. The cookie will be removed so that the
   server does not see it.
 - if server "webA" dies, the requests will be sent to another valid server
   and a cookie will be reassigned.

Flows :

```
(client)                           (haproxy)                         (server A)
  >-- GET /URI1 HTTP/1.0 ------------> |
               ( no cookie, haproxy forwards in load-balancing mode. )
                                       | >-- GET /URI1 HTTP/1.0 ---------->
                                       | <-- HTTP/1.0 200 OK -------------<
               ( the proxy now adds the server cookie in return )
  <-- HTTP/1.0 200 OK ---------------< |
      Set-Cookie: SERVERID=A           |
  >-- GET /URI2 HTTP/1.0 ------------> |
      Cookie: SERVERID=A               |
      ( the proxy sees the cookie. it forwards to server A and deletes it )
                                       | >-- GET /URI2 HTTP/1.0 ---------->
                                       | <-- HTTP/1.0 200 OK -------------<
   ( the proxy does not add the cookie in return because the client knows it )
  <-- HTTP/1.0 200 OK ---------------< |
  >-- GET /URI3 HTTP/1.0 ------------> |
      Cookie: SERVERID=A               |
                                    ( ... )
```

Limits :

 - if clients use keep-alive (HTTP/1.1), only the first response will have
   a cookie inserted, and only the first request of each session will be
   analyzed. This does not cause trouble in insertion mode because the cookie
   is put immediately in the first response, and the session is maintained to
   the same server for all subsequent requests in the same session. However,
   the cookie will not be removed from the requests forwarded to the servers,
   so the server must not be sensitive to unknown cookies. If this causes
   trouble, you can disable keep-alive by adding the following option :

```
        option httpclose
```

 - if for some reason the clients cannot learn more than one cookie (eg: the
   clients are indeed some home-made applications or gateways), and the
   application already produces a cookie, you can use the "prefix" mode (see
   below).

 - LB1 becomes a very sensible server. If LB1 dies, nothing works anymore.
   => you can back it up using keepalived (see below)

 - if the application needs to log the original client's IP, use the
   "forwardfor" option which will add an "X-Forwarded-For" header with the
   original client's IP address. You must also use "httpclose" to ensure
   that you will rewrite every requests and not only the first one of each
   session :

```
        option httpclose
        option forwardfor
```

   The web server will have to be configured to use this header instead.
   For example, on apache, you can use LogFormat for this :

```
        LogFormat "%{X-Forwarded-For}i %l %u %t \"%r\" %>s %b " combined
        CustomLog /var/log/httpd/access_log combined
```

Hints :

```
Sometimes on the internet, you will find a few percent of the clients which
disable cookies on their browser. Obviously they have troubles everywhere on
the web, but you can still help them access your site by using the "source"
balancing algorithm instead of the "roundrobin". It ensures that a given IP
address always reaches the same server as long as the number of servers remains
unchanged. Never use this behind a proxy or in a small network, because the
distribution will be unfair. However, in large internal networks, and on the
internet, it works quite well. Clients which have a dynamic address will not
be affected as long as they accept the cookie, because the cookie always has
precedence over load balancing :

    listen webfarm 192.168.1.1:80
       mode http
       balance source
       cookie SERVERID insert indirect
       option httpchk HEAD /index.html HTTP/1.0
       server webA 192.168.1.11:80 cookie A check
       server webB 192.168.1.12:80 cookie B check
       server webC 192.168.1.13:80 cookie C check
       server webD 192.168.1.14:80 cookie D check


==================================================================
2. HTTP load-balancing with cookie prefixing and high availability
==================================================================

Now you don't want to add more cookies, but rather use existing ones. The
application already generates a "JSESSIONID" cookie which is enough to track
sessions, so we'll prefix this cookie with the server name when we see it.
Since the load-balancer becomes critical, it will be backed up with a second
one in VRRP mode using keepalived under Linux.

Download the latest version of keepalived from this site and install it
on each load-balancer LB1 and LB2 :

       http://www.keepalived.org/

You then have a shared IP between the two load-balancers (we will still use the
original IP). It is active only on one of them at any moment. To allow the
proxy to bind to the shared IP on Linux 2.4, you must enable it in /proc :

# echo 1 >/proc/sys/net/ipv4/ip_nonlocal_bind


    shared IP=192.168.1.1
  192.168.1.3  192.168.1.4    192.168.1.11-192.168.1.14   192.168.1.2
 -------+------------+-----------+-----+-----+-----+--------+----
        |            |           |     |     |     |       _|_db
     +--+--+      +--+--+      +-+-+ +-+-+ +-+-+ +-+-+    (___)
     | LB1 |      | LB2 |      | A | | B | | C | | D |    (___)
     +-----+      +-----+      +---+ +---+ +---+ +---+    (___)
     haproxy      haproxy        4 cheap web servers
     keepalived   keepalived


Config on both proxies (LB1 and LB2) :
--------------------------------------

    listen webfarm 192.168.1.1:80
       mode http
       balance roundrobin
       cookie JSESSIONID prefix
       option httpclose
       option forwardfor
       option httpchk HEAD /index.html HTTP/1.0
       server webA 192.168.1.11:80 cookie A check
       server webB 192.168.1.12:80 cookie B check
       server webC 192.168.1.13:80 cookie C check
       server webD 192.168.1.14:80 cookie D check
```

```
Notes: the proxy will modify EVERY cookie sent by the client and the server,
so it is important that it can access to ALL cookies in ALL requests for
each session. This implies that there is no keep-alive (HTTP/1.1), thus the
"httpclose" option. Only if you know for sure that the client(s) will never
use keep-alive (eg: Apache 1.3 in reverse-proxy mode), you can remove this
option.


Configuration for keepalived on LB1/LB2 :
-----------------------------------------

    vrrp_script chk_haproxy {           # Requires keepalived-1.1.13
        script "killall -0 haproxy"     # cheaper than pidof
        interval 2                      # check every 2 seconds
        weight 2                        # add 2 points of prio if OK
    }

    vrrp_instance VI_1 {
        interface eth0
        state MASTER
        virtual_router_id 51
        priority 101                    # 101 on master, 100 on backup
        virtual_ipaddress {
            192.168.1.1
        }
        track_script {
            chk_haproxy
        }
    }


Description :
-------------
 - LB1 is VRRP master (keepalived), LB2 is backup. Both monitor the haproxy
   process, and lower their prio if it fails, leading to a failover to the
   other node.
 - LB1 will receive clients requests on IP 192.168.1.1.
 - both load-balancers send their checks from their native IP.
 - if a request does not contain a cookie, it will be forwarded to a valid
   server
 - in return, if a JESSIONID cookie is seen, the server name will be prefixed
   into it, followed by a delimitor ('~')
 - when the client comes again with the cookie "JSESSIONID=A~xxx", LB1 will
   know that it must be forwarded to server A. The server name will then be
   extracted from cookie before it is sent to the server.
 - if server "webA" dies, the requests will be sent to another valid server
   and a cookie will be reassigned.


Flows :
-------

(client)                           (haproxy)                         (server A)
  >-- GET /URI1 HTTP/1.0 ------------> |
               ( no cookie, haproxy forwards in load-balancing mode. )
                                       | >-- GET /URI1 HTTP/1.0 ---------->
                                       |     X-Forwarded-For: 10.1.2.3
                                       | <-- HTTP/1.0 200 OK -------------<
                        ( no cookie, nothing changed )
  <-- HTTP/1.0 200 OK ---------------< |
  >-- GET /URI2 HTTP/1.0 ------------> |
    ( no cookie, haproxy forwards in lb mode, possibly to another server. )
                                       | >-- GET /URI2 HTTP/1.0 ---------->
                                       |     X-Forwarded-For: 10.1.2.3
                                       | <-- HTTP/1.0 200 OK -------------<
                                       |     Set-Cookie: JSESSIONID=123
    ( the cookie is identified, it will be prefixed with the server name )
```

```
  <-- HTTP/1.0 200 OK ---------------< |
      Set-Cookie: JSESSIONID=A~123     |
  >-- GET /URI3 HTTP/1.0 ------------> |
      Cookie: JSESSIONID=A~123         |
       ( the proxy sees the cookie, removes the server name and forwards
          to server A which sees the same cookie as it previously sent )
                                       | >-- GET /URI3 HTTP/1.0 ---------->
                                       |     Cookie: JSESSIONID=123
                                       |     X-Forwarded-For: 10.1.2.3
                                       | <-- HTTP/1.0 200 OK -------------<
                        ( no cookie, nothing changed )
  <-- HTTP/1.0 200 OK ---------------< |
                                    ( ... )
```

Hints :

Sometimes, there will be some powerful servers in the farm, and some smaller ones. In this situation, it may be desirable to tell haproxy to respect the difference in performance. Let's consider that WebA and WebB are two old P3-1.2 GHz while WebC and WebD are shiny new Opteron-2.6 GHz. If your application scales with CPU, you may assume a very rough 2.6/1.2 performance ratio between the servers. You can inform haproxy about this using the "weight" keyword, with values between 1 and 256. It will then spread the load the most smoothly possible respecting those ratios :

```
       server webA 192.168.1.11:80 cookie A weight 12 check
       server webB 192.168.1.12:80 cookie B weight 12 check
       server webC 192.168.1.13:80 cookie C weight 26 check
       server webD 192.168.1.14:80 cookie D weight 26 check
```

==
2.1 Variations involving external layer 4 load-balancers
==

Instead of using a VRRP-based active/backup solution for the proxies, they can also be load-balanced by a layer4 load-balancer (eg: Alteon) which will also check that the services run fine on both proxies :

```
              | VIP=192.168.1.1
         +----+----+
         | Alteon  |
         +----+----+
              |
 192.168.1.3  |  192.168.1.4  192.168.1.11-192.168.1.14   192.168.1.2
 -------+-----+------+-----------+-----+-----+-----+--------+----
        |            |           |     |     |     |       _|_db
     +--+--+      +--+--+      +-+-+ +-+-+ +-+-+ +-+-+    (___)
     | LB1 |      | LB2 |      | A | | B | | C | | D |    (___)
     +-----+      +-----+      +---+ +---+ +---+ +---+    (___)
     haproxy      haproxy        4 cheap web servers
```

Config on both proxies (LB1 and LB2) :

```
    listen webfarm 0.0.0.0:80
       mode http
       balance roundrobin
       cookie JSESSIONID prefix
       option httpclose
       option forwardfor
       option httplog
       option dontlognull
       option httpchk HEAD /index.html HTTP/1.0
       server webA 192.168.1.11:80 cookie A check
       server webB 192.168.1.12:80 cookie B check
       server webC 192.168.1.13:80 cookie C check
       server webD 192.168.1.14:80 cookie D check
```

The "dontlognull" option is used to prevent the proxy from logging the health checks from the Alteon. If a session exchanges no data, then it will not be logged.

Config on the Alteon :

```
    /c/slb/real  11
           ena
           name "LB1"
           rip 192.168.1.3
    /c/slb/real  12
           ena
           name "LB2"
           rip 192.168.1.4
    /c/slb/group 10
           name "LB1-2"
           metric roundrobin
           health tcp
           add 11
           add 12
    /c/slb/virt 10
           ena
           vip 192.168.1.1
    /c/slb/virt 10/service http
           group 10
```

Note: the health-check on the Alteon is set to "tcp" to prevent the proxy from forwarding the connections. It can also be set to "http", but for this the proxy must specify a "monitor-net" with the Alteons' addresses, so that the Alteon can really check that the proxies can talk HTTP but without forwarding the connections to the end servers. Check next section for an example on how to use monitor-net.

==
2.2 Generic TCP relaying and external layer 4 load-balancers
==

Sometimes it's useful to be able to relay generic TCP protocols (SMTP, TSE, VNC, etc...), for example to interconnect private networks. The problem comes when you use external load-balancers which need to send periodic health-checks to the proxies, because these health-checks get forwarded to the end servers. The solution is to specify a network which will be dedicated to monitoring systems and must not lead to a forwarding connection nor to any log, using the "monitor-net" keyword. Note: this feature expects a version of haproxy greater than or equal to 1.1.32 or 1.2.6.

```
                |  VIP=172.16.1.1   |
           +----+----+          +----+----+
           | Alteon1 |          | Alteon2 |
           +----+----+          +----+----+
 192.168.1.252  |  GW=192.168.1.254 |  192.168.1.253
                |                   |
          ------+---+------------+--+-----------------> TSE farm : 192.168.1.10
       192.168.1.1  |            | 192.168.1.2
                 +--+--+      +--+--+
                 | LB1 |      | LB2 |
                 +-----+      +-----+
                 haproxy      haproxy
```

Config on both proxies (LB1 and LB2) :

```
    listen tse-proxy
```

```
       bind :3389,:1494,:5900  # TSE, ICA and VNC at once.
       mode tcp
       balance roundrobin
       server tse-farm 192.168.1.10
       monitor-net 192.168.1.252/31
```

The "monitor-net" option instructs the proxies that any connection coming from 192.168.1.252 or 192.168.1.253 will not be logged nor forwarded and will be closed immediately. The Alteon load-balancers will then see the proxies alive without perturbating the service.

Config on the Alteon :

```
    /c/l3/if 1
           ena
           addr 192.168.1.252
           mask 255.255.255.0
    /c/slb/real  11
           ena
           name "LB1"
           rip 192.168.1.1
    /c/slb/real  12
           ena
           name "LB2"
           rip 192.168.1.2
    /c/slb/group 10
           name "LB1-2"
           metric roundrobin
           health tcp
           add 11
           add 12
    /c/slb/virt 10
           ena
           vip 172.16.1.1
    /c/slb/virt 10/service 1494
           group 10
    /c/slb/virt 10/service 3389
           group 10
    /c/slb/virt 10/service 5900
           group 10
```

Special handling of SSL :

Sometimes, you want to send health-checks to remote systems, even in TCP mode, in order to be able to failover to a backup server in case the first one is dead. Of course, you can simply enable TCP health-checks, but it sometimes happens that intermediate firewalls between the proxies and the remote servers acknowledge the TCP connection themselves, showing an always-up server. Since this is generally encountered on long-distance communications, which often involve SSL, an SSL health-check has been implemented to workaround this issue. It sends SSL Hello messages to the remote server, which in turns replies with SSL Hello messages. Setting it up is very easy :

```
    listen tcp-syslog-proxy
       bind :1514      # listen to TCP syslog traffic on this port (SSL)
       mode tcp
       balance roundrobin
       option ssl-hello-chk
       server syslog-prod-site 192.168.1.10 check
       server syslog-back-site 192.168.2.10 check backup
```

===
3. Simple HTTP/HTTPS load-balancing with cookie insertion
===

This is the same context as in example 1 above, but the web server uses HTTPS.

```
                +-------+
                |clients|  clients
                +---+---+
                    |
                   -+-----+--------+----
                          |       _|_db
                       +--+--+   (___)
                       | SSL |   (___)
                       | web |   (___)
                       +-----+
                   192.168.1.1   192.168.1.2
```

Since haproxy does not handle SSL, this part will have to be extracted from the servers (freeing even more ressources) and installed on the load-balancer itself. Install haproxy and apache+mod_ssl on the old box which will spread the load between the new boxes. Apache will work in SSL reverse-proxy-cache. If the application is correctly developped, it might even lower its load. However, since there now is a cache between the clients and haproxy, some security measures must be taken to ensure that inserted cookies will not be cached.

```
  192.168.1.1    192.168.1.11-192.168.1.14   192.168.1.2
 -------+-----------+-----+-----+-----+--------+----
        |           |     |     |     |       _|_db
     +--+--+      +-+-+ +-+-+ +-+-+ +-+-+    (___)
     | LB1 |      | A | | B | | C | | D |    (___)
     +-----+      +---+ +---+ +---+ +---+    (___)
     apache         4 cheap web servers
     mod_ssl
     haproxy
```

Config on haproxy (LB1) :

```
    listen 127.0.0.1:8000
       mode http
       balance roundrobin
       cookie SERVERID insert indirect nocache
       option httpchk HEAD /index.html HTTP/1.0
       server webA 192.168.1.11:80 cookie A check
       server webB 192.168.1.12:80 cookie B check
       server webC 192.168.1.13:80 cookie C check
       server webD 192.168.1.14:80 cookie D check
```

Description :

 - apache on LB1 will receive clients requests on port 443
 - it forwards it to haproxy bound to 127.0.0.1:8000
 - if a request does not contain a cookie, it will be forwarded to a valid server
 - in return, a cookie "SERVERID" will be inserted in the response holding the server name (eg: "A"), and a "Cache-control: private" header will be added so that the apache does not cache any page containing such cookie.
 - when the client comes again with the cookie "SERVERID=A", LB1 will know that it must be forwarded to server A. The cookie will be removed so that the server does not see it.
 - if server "webA" dies, the requests will be sent to another valid server and a cookie will be reassigned.

Notes :

 - if the cookie works in "prefix" mode, there is no need to add the "nocache" option because it is an application cookie which will be modified, and the application flags will be preserved.

```
  - if apache 1.3 is used as a front-end before haproxy, it always disables
    HTTP keep-alive on the back-end, so there is no need for the "httpclose"
    option on haproxy.
  - configure apache to set the X-Forwarded-For header itself, and do not do
    it on haproxy if you need the application to know about the client's IP.


Flows :
-------

(apache)                           (haproxy)                         (server A)
  >-- GET /URI1 HTTP/1.0 ------------> |
               ( no cookie, haproxy forwards in load-balancing mode. )
                                       | >-- GET /URI1 HTTP/1.0 ---------->
                                       | <-- HTTP/1.0 200 OK -------------<
               ( the proxy now adds the server cookie in return )
  <-- HTTP/1.0 200 OK ---------------< |
      Set-Cookie: SERVERID=A           |
      Cache-Control: private           |
  >-- GET /URI2 HTTP/1.0 ------------> |
      Cookie: SERVERID=A               |
      ( the proxy sees the cookie. it forwards to server A and deletes it )
                                       | >-- GET /URI2 HTTP/1.0 ---------->
                                       | <-- HTTP/1.0 200 OK -------------<
   ( the proxy does not add the cookie in return because the client knows it )
  <-- HTTP/1.0 200 OK ---------------< |
  >-- GET /URI3 HTTP/1.0 ------------> |
      Cookie: SERVERID=A               |
                                    ( ... )



========================================
3.1. Alternate solution using Stunnel
========================================

When only SSL is required and cache is not needed, stunnel is a cheaper
solution than Apache+mod_ssl. By default, stunnel does not process HTTP and
does not add any X-Forwarded-For header, but there is a patch on the official
haproxy site to provide this feature to recent stunnel versions.

This time, stunnel will only process HTTPS and not HTTP. This means that
haproxy will get all HTTP traffic, so haproxy will have to add the
X-Forwarded-For header for HTTP traffic, but not for HTTPS traffic since
stunnel will already have done it. We will use the "except" keyword to tell
haproxy that connections from local host already have a valid header.


  192.168.1.1    192.168.1.11-192.168.1.14   192.168.1.2
 -------+-----------+-----+-----+-----+--------+----
        |           |     |     |     |       _|_db
     +--+--+      +-+-+ +-+-+ +-+-+ +-+-+    (___)
     | LB1 |      | A | | B | | C | | D |    (___)
     +-----+      +---+ +---+ +---+ +---+    (___)
     stunnel        4 cheap web servers
     haproxy


Config on stunnel (LB1) :
-------------------------

    cert=/etc/stunnel/stunnel.pem
    setuid=stunnel
    setgid=proxy

    socket=l:TCP_NODELAY=1
    socket=r:TCP_NODELAY=1

    [https]
```

```
    accept=192.168.1.1:443
    connect=192.168.1.1:80
    xforwardedfor=yes
```

Config on haproxy (LB1) :

```
    listen 192.168.1.1:80
       mode http
       balance roundrobin
       option forwardfor except 192.168.1.1
       cookie SERVERID insert indirect nocache
       option httpchk HEAD /index.html HTTP/1.0
       server webA 192.168.1.11:80 cookie A check
       server webB 192.168.1.12:80 cookie B check
       server webC 192.168.1.13:80 cookie C check
       server webD 192.168.1.14:80 cookie D check
```

Description :

- stunnel on LB1 will receive clients requests on port 443
- it forwards them to haproxy bound to port 80
- haproxy will receive HTTP client requests on port 80 and decrypted SSL requests from Stunnel on the same port.
- stunnel will add the X-Forwarded-For header
- haproxy will add the X-Forwarded-For header for everyone except the local address (stunnel).

==
4. Soft-stop for application maintenance
==

When an application is spread across several servers, the time to update all instances increases, so the application seems jerky for a longer period.

HAproxy offers several solutions for this. Although it cannot be reconfigured without being stopped, nor does it offer any external command, there are other working solutions.

===
4.1 Soft-stop using a file on the servers
===

This trick is quite common and very simple: put a file on the server which will be checked by the proxy. When you want to stop the server, first remove this file. The proxy will see the server as failed, and will not send it any new session, only the old ones if the "persist" option is used. Wait a bit then stop the server when it does not receive anymore connections.

```
    listen 192.168.1.1:80
       mode http
       balance roundrobin
       cookie SERVERID insert indirect
       option httpchk HEAD /running HTTP/1.0
       server webA 192.168.1.11:80 cookie A check inter 2000 rise 2 fall 2
       server webB 192.168.1.12:80 cookie B check inter 2000 rise 2 fall 2
       server webC 192.168.1.13:80 cookie C check inter 2000 rise 2 fall 2
       server webD 192.168.1.14:80 cookie D check inter 2000 rise 2 fall 2
       option persist
       redispatch
       contimeout 5000
```

Description :

```
  - every 2 seconds, haproxy will try to access the file "/running" on the
    servers, and declare the server as down after 2 attempts (4 seconds).
  - only the servers which respond with a 200 or 3XX response will be used.
  - if a request does not contain a cookie, it will be forwarded to a valid
    server
  - if a request contains a cookie for a failed server, haproxy will insist
    on trying to reach the server anyway, to let the user finish what he was
    doing. ("persist" option)
  - if the server is totally stopped, the connection will fail and the proxy
    will rebalance the client to another server ("redispatch")

Usage on the web servers :
--------------------------
- to start the server :
    # /etc/init.d/httpd start
    # touch /home/httpd/www/running

- to soft-stop the server
    # rm -f /home/httpd/www/running

- to completely stop the server :
    # /etc/init.d/httpd stop

Limits
------
If the server is totally powered down, the proxy will still try to reach it
for those clients who still have a cookie referencing it, and the connection
attempt will expire after 5 seconds ("contimeout"), and only after that, the
client will be redispatched to another server. So this mode is only useful
for software updates where the server will suddenly refuse the connection
because the process is stopped. The problem is the same if the server suddenly
crashes. All of its users will be fairly perturbated.


==================================
4.2 Soft-stop using backup servers
==================================

A better solution which covers every situation is to use backup servers.
Version 1.1.30 fixed a bug which prevented a backup server from sharing
the same cookie as a standard server.


    listen 192.168.1.1:80
       mode http
       balance roundrobin
       redispatch
       cookie SERVERID insert indirect
       option httpchk HEAD / HTTP/1.0
       server webA 192.168.1.11:80 cookie A check port 81 inter 2000
       server webB 192.168.1.12:80 cookie B check port 81 inter 2000
       server webC 192.168.1.13:80 cookie C check port 81 inter 2000
       server webD 192.168.1.14:80 cookie D check port 81 inter 2000

       server bkpA 192.168.1.11:80 cookie A check port 80 inter 2000 backup
       server bkpB 192.168.1.12:80 cookie B check port 80 inter 2000 backup
       server bkpC 192.168.1.13:80 cookie C check port 80 inter 2000 backup
       server bkpD 192.168.1.14:80 cookie D check port 80 inter 2000 backup

Description
-----------
Four servers webA..D are checked on their port 81 every 2 seconds. The same
servers named bkpA..D are checked on the port 80, and share the exact same
cookies. Those servers will only be used when no other server is available
for the same cookie.

When the web servers are started, only the backup servers are seen as
available. On the web servers, you need to redirect port 81 to local
port 80, either with a local proxy (eg: a simple haproxy tcp instance),
```

or with iptables (linux) or pf (openbsd). This is because we want the real web server to reply on this port, and not a fake one. Eg, with iptables :

```
# /etc/init.d/httpd start
# iptables -t nat -A PREROUTING -p tcp --dport 81 -j REDIRECT --to-port 80
```

A few seconds later, the standard server is seen up and haproxy starts to send it new requests on its real port 80 (only new users with no cookie, of course).

If a server completely crashes (even if it does not respond at the IP level), both the standard and backup servers will fail, so clients associated to this server will be redispatched to other live servers and will lose their sessions.

Now if you want to enter a server into maintenance, simply stop it from responding on port 81 so that its standard instance will be seen as failed, but the backup will still work. Users will not notice anything since the service is still operational :

```
# iptables -t nat -D PREROUTING -p tcp --dport 81 -j REDIRECT --to-port 80
```

The health checks on port 81 for this server will quickly fail, and the standard server will be seen as failed. No new session will be sent to this server, and existing clients with a valid cookie will still reach it because the backup server will still be up.

Now wait as long as you want for the old users to stop using the service, and once you see that the server does not receive any traffic, simply stop it :

```
# /etc/init.d/httpd stop
```

The associated backup server will in turn fail, and if any client still tries to access this particular server, he will be redispatched to any other valid server because of the "redispatch" option.

This method has an advantage : you never touch the proxy when doing server maintenance. The people managing the servers can make them disappear smoothly.

4.2.1 Variations for operating systems without any firewall software

The downside is that you need a redirection solution on the server just for the health-checks. If the server OS does not support any firewall software, this redirection can also be handled by a simple haproxy in tcp mode :

```
global
    daemon
    quiet
    pidfile /var/run/haproxy-checks.pid
listen 0.0.0.0:81
    mode tcp
    dispatch 127.0.0.1:80
    contimeout 1000
    clitimeout 10000
    srvtimeout 10000
```

To start the web service :

```
# /etc/init.d/httpd start
# haproxy -f /etc/haproxy/haproxy-checks.cfg
```

To soft-stop the service :

```
# kill $(</var/run/haproxy-checks.pid)
```

The port 81 will stop responding and the load-balancer will notice the failure.

```
4.2.2 Centralizing the server management
----------------------------------------

If one finds it preferable to manage the servers from the load-balancer itself,
the port redirector can be installed on the load-balancer itself. See the
example with iptables below.

Make the servers appear as operational :
  # iptables -t nat -A OUTPUT -d 192.168.1.11 -p tcp --dport 81 -j DNAT --to-dest :80
  # iptables -t nat -A OUTPUT -d 192.168.1.12 -p tcp --dport 81 -j DNAT --to-dest :80
  # iptables -t nat -A OUTPUT -d 192.168.1.13 -p tcp --dport 81 -j DNAT --to-dest :80
  # iptables -t nat -A OUTPUT -d 192.168.1.14 -p tcp --dport 81 -j DNAT --to-dest :80

Soft stop one server :
  # iptables -t nat -D OUTPUT -d 192.168.1.12 -p tcp --dport 81 -j DNAT --to-dest :80

Another solution is to use the "COMAFILE" patch provided by Alexander Lazic,
which is available for download here :

   http://w.ods.org/tools/haproxy/contrib/


4.2.3 Notes :
-------------
  - Never, ever, start a fake service on port 81 for the health-checks, because
    a real web service failure will not be detected as long as the fake service
    runs. You must really forward the check port to the real application.

  - health-checks will be sent twice as often, once for each standard server,
    and once for each backup server. All this will be multiplicated by the
    number of processes if you use multi-process mode. You will have to ensure
    that all the checks sent to the server do not overload it.

=======================
4.3 Hot reconfiguration
=======================

There are two types of haproxy users :
  - those who can never do anything in production out of maintenance periods ;
  - those who can do anything at any time provided that the consequences are
    limited.

The first ones have no problem stopping the server to change configuration
because they got some maintenance periods during which they can break anything.
So they will even prefer doing a clean stop/start sequence to ensure everything
will work fine upon next reload. Since those have represented the majority of
haproxy uses, there has been little effort trying to improve this.

However, the second category is a bit different. They like to be able to fix an
error in a configuration file without anyone noticing. This can sometimes also
be the case for the first category because humans are not failsafe.

For this reason, a new hot reconfiguration mechanism has been introduced in
version 1.1.34. Its usage is very simple and works even in chrooted
environments with lowered privileges. The principle is very simple : upon
reception of a SIGTTOU signal, the proxy will stop listening to all the ports.
This will release the ports so that a new instance can be started. Existing
connections will not be broken at all. If the new instance fails to start,
then sending a SIGTTIN signal back to the original processes will restore
the listening ports. This is possible without any special privileges because
the sockets will not have been closed, so the bind() is still valid. Otherwise,
if the new process starts successfully, then sending a SIGUSR1 signal to the
old one ensures that it will exit as soon as its last session ends.

A hot reconfiguration script would look like this :

  # save previous state
  mv /etc/haproxy/config /etc/haproxy/config.old
  mv /var/run/haproxy.pid /var/run/haproxy.pid.old
```

```
  mv /etc/haproxy/config.new /etc/haproxy/config
  kill -TTOU $(cat /var/run/haproxy.pid.old)
  if haproxy -p /var/run/haproxy.pid -f /etc/haproxy/config; then
    echo "New instance successfully loaded, stopping previous one."
    kill -USR1 $(cat /var/run/haproxy.pid.old)
    rm -f /var/run/haproxy.pid.old
    exit 1
  else
    echo "New instance failed to start, resuming previous one."
    kill -TTIN $(cat /var/run/haproxy.pid.old)
    rm -f /var/run/haproxy.pid
    mv /var/run/haproxy.pid.old /var/run/haproxy.pid
    mv /etc/haproxy/config /etc/haproxy/config.new
    mv /etc/haproxy/config.old /etc/haproxy/config
    exit 0
  fi
```

After this, you can still force old connections to end by sending
a SIGTERM to the old process if it still exists :

```
    kill $(cat /var/run/haproxy.pid.old)
    rm -f /var/run/haproxy.pid.old
```

Be careful with this as in multi-process mode, some pids might already
have been reallocated to completely different processes.

==
5. Multi-site load-balancing with local preference
==

5.1 Description of the problem
==============================

Consider a world-wide company with sites on several continents. There are two
production sites SITE1 and SITE2 which host identical applications. There are
many offices around the world. For speed and communication cost reasons, each
office uses the nearest site by default, but can switch to the backup site in
the event of a site or application failure. There also are users on the
production sites, which use their local sites by default, but can switch to the
other site in case of a local application failure.

The main constraints are :

  - application persistence : although the application is the same on both
    sites, there is no session synchronisation between the sites. A failure
    of one server or one site can cause a user to switch to another server
    or site, but when the server or site comes back, the user must not switch
    again.

  - communication costs : inter-site communication should be reduced to the
    minimum. Specifically, in case of a local application failure, every
    office should be able to switch to the other site without continuing to
    use the default site.

5.2 Solution
============
  - Each production site will have two haproxy load-balancers in front of its
    application servers to balance the load across them and provide local HA.
    We will call them "S1L1" and "S1L2" on site 1, and "S2L1" and "S2L2" on
    site 2. These proxies will extend the application's JSESSIONID cookie to
    put the server name as a prefix.

  - Each production site will have one front-end haproxy director to provide
    the service to local users and to remote offices. It will load-balance
    across the two local load-balancers, and will use the other site's
    load-balancers as backup servers. It will insert the local site identifier
    in a SITE cookie for the local load-balancers, and the remote site

identifier for the remote load-balancers. These front-end directors will be called "SD1" and "SD2" for "Site Director".

- Each office will have one haproxy near the border gateway which will direct local users to their preference site by default, or to the backup site in the event of a previous failure. It will also analyze the SITE cookie, and direct the users to the site referenced in the cookie. Thus, the preferred site will be declared as a normal server, and the backup site will be declared as a backup server only, which will only be used when the primary site is unreachable, or when the primary site's director has forwarded traffic to the second site. These proxies will be called "OP1".."OPXX" for "Office Proxy #XX".

5.3 Network diagram

Note : offices 1 and 2 are on the same continent as site 1, while office 3 is on the same continent as site 3. Each production site can reach the second one either through the WAN or through a dedicated link.

```
        Office1         Office2                          Office3
         users           users                            users
192.168  # # #   192.168 # # #                            # # #
.1.0/24  | | |   .2.0/24 | | |             192.168.3.0/24 | | |
  --+----+-+-+-   --+----+-+-+-                   ---+----+-+-+-
    |      | .1     |      | .1                      |      | .1
    |    +-+-+      |    +-+-+                       |    +-+-+
    |    |OP1|      |    |OP2|                       |    |OP3|  ...
  ,-:-.  +---+    ,-:-.  +---+                     ,-:-.  +---+
 (  X  )         (  X  )                          (  X  )
  `-:-'           `-:-'             ,---.          `-:-'
  --+---------------+------+----~~~(  X  )~~~~-------+---------+-
                           |        `---'                      |
                           |                                   |
                 +---+   ,-:-.                       +---+   ,-:-.
                 |SD1|  (  X  )                      |SD2|  (  X  )
   ( SITE 1 )    +-+-+   `-:-'         ( SITE 2 )    +-+-+   `-:-'
                   |.1     |                           |.1     |
   10.1.1.0/24     |       |     ,---. 10.2.1.0/24     |       |
        -+-+-+-+-+-+-+-----+-+--(  X  )------+-+-+-+-+-+-+-----+-+--
         | | | | |   |       |   `---'       | | | | |   |       |
      ...# # # # #   |.11    |.12         ...# # # # #   |.11    |.12
          Site 1   +-+--+  +-+--+             Site 2   +-+--+  +-+--+
          Local    |S1L1|  |S1L2|             Local    |S2L1|  |S2L2|
          users    +-+--+  +--+-+             users    +-+--+  +--+-+
                     |        |                          |        |
   10.1.2.0/24    -+-+-+--+--++--      10.2.2.0/24    -+-+-+--+--++--
                   |.1       |.4                       |.1       |.4
                 +-+-+     +-+-+                     +-+-+     +-+-+
                 |W11| ~~~ |W14|                     |W21| ~~~ |W24|
                 +---+     +---+                     +---+     +---+
              4 application servers               4 application servers
                    on site 1                           on site 2
```

5.4 Description

5.4.1 Local users

- Office 1 users connect to OP1 = 192.168.1.1
- Office 2 users connect to OP2 = 192.168.2.1
- Office 3 users connect to OP3 = 192.168.3.1
- Site 1 users connect to SD1 = 10.1.1.1
- Site 2 users connect to SD2 = 10.2.1.1

```
5.4.2 Office proxies
--------------------
 - Office 1 connects to site 1 by default and uses site 2 as a backup.
 - Office 2 connects to site 1 by default and uses site 2 as a backup.
 - Office 3 connects to site 2 by default and uses site 1 as a backup.

The offices check the local site's SD proxy every 30 seconds, and the
remote one every 60 seconds.


Configuration for Office Proxy OP1
----------------------------------

    listen 192.168.1.1:80
       mode http
       balance roundrobin
       redispatch
       cookie SITE
       option httpchk HEAD / HTTP/1.0
       server SD1 10.1.1.1:80 cookie SITE1 check inter 30000
       server SD2 10.2.1.1:80 cookie SITE2 check inter 60000 backup


Configuration for Office Proxy OP2
----------------------------------

    listen 192.168.2.1:80
       mode http
       balance roundrobin
       redispatch
       cookie SITE
       option httpchk HEAD / HTTP/1.0
       server SD1 10.1.1.1:80 cookie SITE1 check inter 30000
       server SD2 10.2.1.1:80 cookie SITE2 check inter 60000 backup


Configuration for Office Proxy OP3
----------------------------------

    listen 192.168.3.1:80
       mode http
       balance roundrobin
       redispatch
       cookie SITE
       option httpchk HEAD / HTTP/1.0
       server SD2 10.2.1.1:80 cookie SITE2 check inter 30000
       server SD1 10.1.1.1:80 cookie SITE1 check inter 60000 backup


5.4.3 Site directors ( SD1 and SD2 )
------------------------------------
The site directors forward traffic to the local load-balancers, and set a
cookie to identify the site. If no local load-balancer is available, or if
the local application servers are all down, it will redirect traffic to the
remote site, and report this in the SITE cookie. In order not to uselessly
load each site's WAN link, each SD will check the other site at a lower
rate. The site directors will also insert their client's address so that
the application server knows which local user or remote site accesses it.

The SITE cookie which is set by these directors will also be understood
by the office proxies. This is important because if SD1 decides to forward
traffic to site 2, it will write "SITE2" in the "SITE" cookie, and on next
request, the office proxy will automatically and directly talk to SITE2 if
it can reach it. If it cannot, it will still send the traffic to SITE1
where SD1 will in turn try to reach SITE2.

The load-balancers checks are performed on port 81. As we'll see further,
the load-balancers provide a health monitoring port 81 which reroutes to
```

port 80 but which allows them to tell the SD that they are going down soon
and that the SD must not use them anymore.

Configuration for SD1

```
    listen 10.1.1.1:80
       mode http
       balance roundrobin
       redispatch
       cookie SITE insert indirect
       option httpchk HEAD / HTTP/1.0
       option forwardfor
       server S1L1 10.1.1.11:80 cookie SITE1 check port 81 inter 4000
       server S1L2 10.1.1.12:80 cookie SITE1 check port 81 inter 4000
       server S2L1 10.2.1.11:80 cookie SITE2 check port 81 inter 8000 backup
       server S2L2 10.2.1.12:80 cookie SITE2 check port 81 inter 8000 backup
```

Configuration for SD2

```
    listen 10.2.1.1:80
       mode http
       balance roundrobin
       redispatch
       cookie SITE insert indirect
       option httpchk HEAD / HTTP/1.0
       option forwardfor
       server S2L1 10.2.1.11:80 cookie SITE2 check port 81 inter 4000
       server S2L2 10.2.1.12:80 cookie SITE2 check port 81 inter 4000
       server S1L1 10.1.1.11:80 cookie SITE1 check port 81 inter 8000 backup
       server S1L2 10.1.1.12:80 cookie SITE1 check port 81 inter 8000 backup
```

5.4.4 Local load-balancers S1L1, S1L2, S2L1, S2L2

Please first note that because SD1 and SD2 use the same cookie for both servers on a same site, the second load-balancer of each site will only receive load-balanced requests, but as soon as the SITE cookie will be set, only the first LB will receive the requests because it will be the first one to match the cookie.

The load-balancers will spread the load across 4 local web servers, and use the JSESSIONID provided by the application to provide server persistence using the new 'prefix' method. Soft-stop will also be implemented as described in section 4 above. Moreover, these proxies will provide their own maintenance soft-stop. Port 80 will be used for application traffic, while port 81 will only be used for health-checks and locally rerouted to port 80. A grace time will be specified to service on port 80, but not on port 81. This way, a soft kill (kill -USR1) on the proxy will only kill the health-check forwarder so that the site director knows it must not use this load-balancer anymore. But the service will still work for 20 seconds and as long as there are established sessions.

These proxies will also be the only ones to disable HTTP keep-alive in the chain, because it is enough to do it at one place, and it's necessary to do it with 'prefix' cookies.

Configuration for S1L1/S1L2

```
    listen 10.1.1.11:80 # 10.1.1.12:80 for S1L2
       grace 20000  # don't kill us until 20 seconds have elapsed
       mode http
       balance roundrobin
       cookie JSESSIONID prefix
       option httpclose
       option forwardfor
```

```
       option httpchk HEAD / HTTP/1.0
       server W11 10.1.2.1:80 cookie W11 check port 81 inter 2000
       server W12 10.1.2.2:80 cookie W12 check port 81 inter 2000
       server W13 10.1.2.3:80 cookie W13 check port 81 inter 2000
       server W14 10.1.2.4:80 cookie W14 check port 81 inter 2000

       server B11 10.1.2.1:80 cookie W11 check port 80 inter 4000 backup
       server B12 10.1.2.2:80 cookie W12 check port 80 inter 4000 backup
       server B13 10.1.2.3:80 cookie W13 check port 80 inter 4000 backup
       server B14 10.1.2.4:80 cookie W14 check port 80 inter 4000 backup

    listen 10.1.1.11:81 # 10.1.1.12:81 for S1L2
       mode tcp
       dispatch 10.1.1.11:80  # 10.1.1.12:80 for S1L2


Configuration for S2L1/S2L2
---------------------------

    listen 10.2.1.11:80 # 10.2.1.12:80 for S2L2
       grace 20000  # don't kill us until 20 seconds have elapsed
       mode http
       balance roundrobin
       cookie JSESSIONID prefix
       option httpclose
       option forwardfor
       option httpchk HEAD / HTTP/1.0
       server W21 10.2.2.1:80 cookie W21 check port 81 inter 2000
       server W22 10.2.2.2:80 cookie W22 check port 81 inter 2000
       server W23 10.2.2.3:80 cookie W23 check port 81 inter 2000
       server W24 10.2.2.4:80 cookie W24 check port 81 inter 2000

       server B21 10.2.2.1:80 cookie W21 check port 80 inter 4000 backup
       server B22 10.2.2.2:80 cookie W22 check port 80 inter 4000 backup
       server B23 10.2.2.3:80 cookie W23 check port 80 inter 4000 backup
       server B24 10.2.2.4:80 cookie W24 check port 80 inter 4000 backup

    listen 10.2.1.11:81 # 10.2.1.12:81 for S2L2
       mode tcp
       dispatch 10.2.1.11:80  # 10.2.1.12:80 for S2L2


5.5 Comments
------------
Since each site director sets a cookie identifying the site, remote office
users will have their office proxies direct them to the right site and stick
to this site as long as the user still uses the application and the site is
available. Users on production sites will be directed to the right site by the
site directors depending on the SITE cookie.

If the WAN link dies on a production site, the remote office users will not
see their site anymore, so they will redirect the traffic to the second site.
If there are dedicated inter-site links as on the diagram above, the second
SD will see the cookie and still be able to reach the original site. For
example :

Office 1 user sends the following to OP1 :
  GET / HTTP/1.0
  Cookie: SITE=SITE1; JSESSIONID=W14~123;

OP1 cannot reach site 1 because its external router is dead. So the SD1 server
is seen as dead, and OP1 will then forward the request to SD2 on site 2,
regardless of the SITE cookie.

SD2 on site 2 receives a SITE cookie containing "SITE1". Fortunately, it
can reach Site 1's load balancers S1L1 and S1L2. So it forwards the request
so S1L1 (the first one with the same cookie).

S1L1 (on site 1) finds "W14" in the JSESSIONID cookie, so it can forward the
```

request to the right server, and the user session will continue to work. Once the Site 1's WAN link comes back, OP1 will see SD1 again, and will not route through SITE 2 anymore.

However, when a new user on Office 1 connects to the application during a site 1 failure, it does not contain any cookie. Since OP1 does not see SD1 because of the network failure, it will direct the request to SD2 on site 2, which will by default direct the traffic to the local load-balancers, S2L1 and S2L2. So only initial users will load the inter-site link, not the new ones.

6. Source balancing

Sometimes it may reveal useful to access servers from a pool of IP addresses instead of only one or two. Some equipments (NAT firewalls, load-balancers) are sensible to source address, and often need many sources to distribute the load evenly amongst their internal hash buckets.

To do this, you simply have to use several times the same server with a different source. Example :

```
    listen 0.0.0.0:80
       mode tcp
       balance roundrobin
       server from1to1 10.1.1.1:80 source 10.1.2.1
       server from2to1 10.1.1.1:80 source 10.1.2.2
       server from3to1 10.1.1.1:80 source 10.1.2.3
       server from4to1 10.1.1.1:80 source 10.1.2.4
       server from5to1 10.1.1.1:80 source 10.1.2.5
       server from6to1 10.1.1.1:80 source 10.1.2.6
       server from7to1 10.1.1.1:80 source 10.1.2.7
       server from8to1 10.1.1.1:80 source 10.1.2.8
```

7. Managing high loads on application servers

One of the roles often expected from a load balancer is to mitigate the load on the servers during traffic peaks. More and more often, we see heavy frameworks used to deliver flexible and evolutive web designs, at the cost of high loads on the servers, or very low concurrency. Sometimes, response times are also rather high. People developing web sites relying on such frameworks very often look for a load balancer which is able to distribute the load in the most evenly fashion and which will be nice with the servers.

There is a powerful feature in haproxy which achieves exactly this : request queueing associated with concurrent connections limit.

Let's say you have an application server which supports at most 20 concurrent requests. You have 3 servers, so you can accept up to 60 concurrent HTTP connections, which often means 30 concurrent users in case of keep-alive (2 persistent connections per user).

Even if you disable keep-alive, if the server takes a long time to respond, you still have a high risk of multiple users clicking at the same time and having their requests unserved because of server saturation. To workaround the problem, you increase the concurrent connection limit on the servers, but their performance stalls under higher loads.

The solution is to limit the number of connections between the clients and the servers. You set haproxy to limit the number of connections on a per-server basis, and you let all the users you want connect to it. It will then fill all the servers up to the configured connection limit, and will put the remaining connections in a queue, waiting for a connection to be released on a server.

This ensures five essential principles :

- all clients can be served whatever their number without crashing the servers, the only impact it that the response time can be delayed.

- the servers can be used at full throttle without the risk of stalling, and fine tuning can lead to optimal performance.

- response times can be reduced by making the servers work below the congestion point, effectively leading to shorter response times even under moderate loads.

- no domino effect when a server goes down or starts up. Requests will be queued more or less, always respecting servers limits.

- it's easy to achieve high performance even on memory-limited hardware. Indeed, heavy frameworks often consume huge amounts of RAM and not always all the CPU available. In case of wrong sizing, reducing the number of concurrent connections will protect against memory shortages while still ensuring optimal CPU usage.

Example :

Haproxy is installed in front of an application servers farm. It will limit the concurrent connections to 4 per server (one thread per CPU), thus ensuring very fast response times.

```
  192.168.1.1   192.168.1.11-192.168.1.13   192.168.1.2
 -------+-------------+-----+-----+------------+----
        |             |     |     |           _|_db
     +--+--+        +-+-+ +-+-+ +-+-+        (___)
     | LB1 |        | A | | B | | C |        (___)
     +-----+        +---+ +---+ +---+        (___)
     haproxy       3 application servers
                   with heavy frameworks
```

Config on haproxy (LB1) :

```
    listen appfarm 192.168.1.1:80
       mode http
       maxconn 10000
       option httpclose
       option forwardfor
       balance roundrobin
       cookie SERVERID insert indirect
       option httpchk HEAD /index.html HTTP/1.0
       server railsA 192.168.1.11:80 cookie A maxconn 4 check
       server railsB 192.168.1.12:80 cookie B maxconn 4 check
       server railsC 192.168.1.13:80 cookie C maxconn 4 check
       contimeout 60000
```

Description :

The proxy listens on IP 192.168.1.1, port 80, and expects HTTP requests. It can accept up to 10000 concurrent connections on this socket. It follows the roundrobin algorithm to assign servers to connections as long as servers are not saturated.

It allows up to 4 concurrent connections per server, and will queue the requests above this value. The "contimeout" parameter is used to set the maximum time a connection may take to establish on a server, but here it is also used to set the maximum time a connection may stay unserved in the queue (1 minute here).

If the servers can each process 4 requests in 10 ms on average, then at 3000 connections, response times will be delayed by at most :

```
3000 / 3 servers / 4 conns * 10 ms = 2.5 seconds
```

Which is not that dramatic considering the huge number of users for such a low number of servers.

When connection queues fill up and application servers are starving, response times will grow and users might abort by clicking on the "Stop" button. It is very undesirable to send aborted requests to servers, because they will eat CPU cycles for nothing.

An option has been added to handle this specific case : "option abortonclose". By specifying it, you tell haproxy that if an input channel is closed on the client side AND the request is still waiting in the queue, then it is highly likely that the user has stopped, so we remove the request from the queue before it will get served.

Managing unfair response times

Sometimes, the application server will be very slow for some requests (eg: login page) and faster for other requests. This may cause excessive queueing of expectedly fast requests when all threads on the server are blocked on a request to the database. Then the only solution is to increase the number of concurrent connections, so that the server can handle a large average number of slow connections with threads left to handle faster connections.

But as we have seen, increasing the number of connections on the servers can be detrimental to performance (eg: Apache processes fighting for the accept() lock). To improve this situation, the "minconn" parameter has been introduced. When it is set, the maximum connection concurrency on the server will be bound by this value, and the limit will increase with the number of clients waiting in queue, till the clients connected to haproxy reach the proxy's maxconn, in which case the connections per server will reach the server's maxconn. It means that during low-to-medium loads, the minconn will be applied, and during surges the maxconn will be applied. It ensures both optimal response times under normal loads, and availability under very high loads.

Example :

```
    listen appfarm 192.168.1.1:80
       mode http
       maxconn 10000
       option httpclose
       option abortonclose
       option forwardfor
       balance roundrobin
       # The servers will get 4 concurrent connections under low
       # loads, and 12 when there will be 10000 clients.
       server railsA 192.168.1.11:80 minconn 4 maxconn 12 check
       server railsB 192.168.1.12:80 minconn 4 maxconn 12 check
       server railsC 192.168.1.13:80 minconn 4 maxconn 12 check
       contimeout 60000
```