ismecompany
Quality Solutions, Reliable Partner

Haproxy configuration review

Author:	Yaroslav Yarmoshyk
Customer	ISM Internal
Version:	2.0

© copyright 2011 ISM eCompany

Table of Contents

General haproxy configuration description:
By default haproxy has central and only configuration file, stored in /etc/haproxy/haproxy.cfg. But due to our needs, which are dialed with configs for each customer to be stored in separate files, I had to make changes in daemon starter (/etc/init.d/haproxy). That is why we have 4 files of global configuration and a directory for customer's config files. They are:

- /etc/haproxy/haproxy.cfg.def – this one stores global directives that are common for all customers;
- /etc/haproxy/haproxy.cfg – this file is being created each time haproxy starts, and comes as default file that haproxy uses for it's run. It is a combination of all client's configurations and haproxy.cfg.def.
- /etc/haproxy/haproxy.cfg.save – this file is being created and rewritten every time haproxy starts, it stores previous working config just in case if something goes wrong it can be used to start haproxy. I'm working on check config mechanism that will automaticaly bring back previous configuration if there are any errors in compiled new config.
- /etc/haproxy/haproxy.cfg.debug - Configuration that failed haproxy to start. It has cfg files entries, so the mistake in configuration can be tracked and fixed.
- /etc/haproxy/Clients – directory that has configurations for every server cluster that are served with current loadbalance system.

*names of files can be changed. They don't play big role.

Config file with default definitions looks like:

```
global
        log 127.0.0.1:514   local0
        log 127.0.0.1:514   local1
        user haproxy #user that daemon starts from
        group haproxy #group that daemon starts from
defaults
        log     global
        mode    http
        option  httplog
        option  dontlognull
        retries 3
        option redispatch
```

option redispatch - 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.

I has two sections global and defaults. Defaults section has options which are the same for each customer, but each frontend and backend can have it's own options, that will be local and global parametres will be ignored. Globals are only haproxy parameters.

There are 2 ways of configuring haproxy ballansing:

— Simple way, is used when cluster requires only one group of servers on backend to be used (one listen section).

```
listen <instance_name>
bind <ip_address>:<port>
mode <layer mode>
balance <ballance_mode>
option <option1>
option <option2>
…................
option <optionN>
server <server_name1> <node_ip_address>:<port> <option1> <option2> … <option N>
server <server_name2> <bind <option1> <option2> … <option N>
```

— More complicated way, used when cluster requires more than one group of servers on backend to be used, in this case each backend need to be configured in separate from frontend section. (Frontend that holds description for few backends)

```
frontend <instance_name>
bind <ip_address:port>
mode <layer mode>
option <option1>
option <option2>
…................
option <optionN>
acl <acl_name1> <acl_type> <acl_definition>

use_backend <backend_name> if <acl_name1>
default_backend <backend_name>

backend static <backend_name>
balance <ballance mode>
option <option1>
option <option2>
…................
option <optionN>
server <server_name1> <nod_ip_address>:<port> <option1> <option2> … <option N>
server <server_name2> <nod_ip_address>:<port> <option1> <option2> … <option N>

backend web<backend_name>
        server <server_name2> <nod_ip_address>:<port> <option1> <option2> …
<option N>
backend backoffice <backend_name>
         server <server_name2> <nod_ip_address>:<port> <option1> <option2> …
<option N>
```

Options and keywords can be found at: http://code.google.com/p/haproxy-docs/wiki/Keywords
Modes: http://code.google.com/p/haproxy-docs/wiki/mode

tcp – Is used for layer 3 balancing. 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 - Is used for layer 7 balancing. 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. Nothing will be logged. 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.

Ballances: http://code.google.com/p/haproxy-docs/wiki/balance

1. **HTTP Load Balancing based on Layer 3 (sticky sessions)**
Config sample:

```
listen <listen_name> <ip_address_to_listen>:801
      mode tcp
      balance source
      option httpchk OPTIONS * HTTP/1.1\r\nHost:\ www
      server sv1 <srv1_ip>:80 check port 80
      server sv2 <srv2_ip>:80 check port 80
      .....................................................................
      server svN <srvN_ip>:80 check port 80
      server backup <srv_with_maintance_page>:80
```

2. **HTTP Load Balancing based on Layer 3 (random)**

Config sample:

```
listen <listen_name> <ip_address_to_listen>:801
        mode tcp
        balance roundrobin
        server sv1 <srv1_ip>:80 check port 80
        server sv2 <srv2_ip>:80 check port 80
        ....................................................................
        server svN <srvN_ip>:80 check port 80
        server backup <srv_with_maintance_page>:80
```

The difference is only in balance modes:

- first uses **source balance** mode where 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.
- Second uses **roundrobin balance** mode, where 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.

3. **HTTP Load Balancing based on Layer 7 (sticky sessions)**

This layer allow more options to use. Here is statistics for hosts, http life checking, cookie assignment and others.
This is config sample:

```
listen <listen_name> <ip_address_to_listen>:801
        mode http
        cookie SERVERID insert nocache indirect
        stats enable
        stats auth admin:123456
        stats uri /stats
        balance roundrobin
        option httpclose
        option forwardfor
        option httpchk OPTIONS * HTTP/1.1\r\nHost:\ www
        server sv1 <srv1_ip>:80 cookie S1 check port 80
        server sv2 <srv2_ip>:80 cookie S2 check port 80
        ....................................................................
        server svN <srvN_ip>:80 check port 80
        server backup <srv_with_maintance_page>:80
```

In this case each session will get cookie added (S1, S2, .. , Sn) that will send visitor to the same server in the cluster every time.

Description on options that are used:

cookie enables cookie-based persistence in a backend. The value of the cookie will be the value indicated after the cookie keyword in a server statement.
Stats enable enable http statistics, that will be available using web-url/stats and login+pass combination specified in stats auth field.
option forwardfor HAProxy works in reverse-proxy mode, the servers see its IP address as their client address. This header contains a value representing the client's IP address.
option httpchk a complete HTTP request is sent once the TCP connection is established, and responses 2xx and 3xx are considered valid.
option httpclose By default, when a client communicates with a server, HAProxy will only analyze, log, and process the first request of each connection. If *option httpclose* is set, it will 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.

In this case frontend+backend configuration, for more complicated cases, will have a view:

```
frontend <frontend_name>
        bind <ip_address_to_listen>:80
        mode http
        acl backoffice path_beg /backend
        acl backoffice2 path_beg /BackEnd
        use_backend  <back_end_name1> if backoffice
        use_backend  <back_end_name2> if backoffice2
        default_backend <dafault_back_end_name>

backend  <back_end_name1>
       server <server_name> <server_listen_ip>:80 cookie SC1 check inter 2000 fall 3

backend  <back_end_name2>
       server <server_name2> <server_listen_ip2>:80 cookie SC2 check inter 2000 fall 3

backend  <dafault_back_end_name>
        balance roundrobin
        stats enable
        stats realm LoadBalancer_statistics
        stats auth admin:adminpassword
        stats scope <frontend_name>
        stats scope <back_end_name1>
        stats scope <back_end_name2>
        stats scope <dafault_back_end_name>
        stats uri /stats
        server <server_name3> <server_listen_ip3>:80 cookie SC3 check inter 2000 fall 3
        server <server_name4> <server_listen_ip4>:80 cookie SC4 check inter 2000 fall 3
        server backup <backup_server_listen_ip>:80
```

In this example each server response gets sticky cookie (SC1, SC2, SC3, SC4), that will bring customer to the same server in cluster on next visit.

4. HTTP Load Balancing based on Layer 7 (random)

Config for random ballancing will look the same as configs in previous example, but without “cookie SCn” assignment. In this case extra “private” cookie will be added to header, to make it possible to get content if customers will be behind some proxy server, but that customer wouldn't be assigned to any of nodes in cluster, and next time visit the decision will be taken by roundrobin algorithm.

5. HTTP and HTTPS Load Balancing

Http load ballansing is used in regular haproxy way – the way it was described in previous examples. SSL content ballansing can be done on layer 3 in tcp mode. But in case when cluster hosts are unreachable – their certificates are unreachable too. So we need to store those in 2 places – on web server and at loadballancer itself. To show 503th error pages we'll use nginx. And everything is gonna work.

```
listen <listen_name_https> <ip_address_to_listen>:8011*1
      mode tcp
      balance source
      option httpchk OPTIONS * HTTP/1.1\r\nHost:\ www
      server sv1 <srv1_ip>:433 check port 433
      server sv2 <srv2_ip>:433 check port 433
      .....................................................................
      server svN <srvN_ip>:433 check port 433
      server backup 127.0.0.1:433
```

6.Server section description.

Normally server line at backend section looks like this:

```
server <server_name> <some_ip_address>[:port] [settings …]
```

Settings that we can use:
cookie <value> 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.
check option enables health checks on the server. By default, a server is always considered available. If "check" is set, the server will receive periodic health checks 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", and "ssl-hello-chk" options. Please refer to those options and parameters for more information.
inter <delay> sets the interval between two consecutive health checks to <delay> milliseconds. If left unspecified, the delay defaults to 2000 ms.
fall <count> 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.
weight <weight> 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.

Finally we'll get:

```
server <server_name1> <some_ip_address1>[:port] cookie SC1 weight 20 check inter 4000 fall 3
server <server_name2> <some_ip_address2>[:port] cookie SC2 weight 10 check inter 4000 fall 3
```

This is block of 2 servers that work for one backend each has its own name an ip address, responses from them have cookies SC1 and SC2 for sticky session, first has higher priority in cluster, each one is being checked every 4000 ms, and marks as offline after 3 errors in responses.

7. Load Balancing traffic to one server, with high load the extra server will be added to the Load Balance Pool.

In case if too many connections will come to servers and we'll need to add one more server in cluster it can be easily done by adding it into needed config file and after gentle restart (/etc/init.d/haproxy reload) haproxy will start proxying connections to it.
The only stuff needed to do is checking daemon starter parameters at reload section. Pay attention to -sf and -st parameters:

-sf <pidlist> Send FINISH signal to the pids in pidlist after startup. The processes which receive this signal will wait for all sessions to finish before exiting. This option must be specified last, followed by any number of PIDs. Technically speaking, SIGTTOU and SIGUSR1 are sent.

-st <pidlist> Send TERMINATE signal to the pids in pidlist after startup. The processes which receive this signal will wait immediately terminate, closing all active sessions. This option must be specified last, followed by any number of PIDs. Technically speaking, SIGTTOU and SIGTERM are sent.

We are interested in gentle connections completing, so that line should look like this:

```
$HAPROXY -f "$CONFIG" -p $PIDFILE -D $EXTRAOPTS -sf $(cat $PIDFILE)
```

None of connected customers will see that anything happened to the environment.

8. Maintenance page with error code 503 and Possibility to see the Maintenance Page per customer. So it is possible for the customers to see which maintenance page it is using but it can not see the pages of our other customers.

For this we will need to use some light web server (nginx). All maintenance pages will be stored in one place on loadballancer (/var/www/503_pages), and web server will refer to that location. Common maintenance page is

a combination of html message file and few jpg images. Viewing it as compiled web page will return “200 OK” http response in header. So all content should be uploaded with “503 Temporary unavailable” http response in header. To do that rewrite rule is required for nginx host.
Config sample for that:

```
server {
   listen <ip_address_to_listen>:80;
      server_name server-name.nl;
   location / {
      return 503;
      }
   error_page 503 @maint;
   location @maint {
      root /var/www/503_pages/failover-customer/;
      if (!-f $request_filename) {
      rewrite  (some\.jpg|some2\.gif)$ /$1 break;
      rewrite ^(.*)$ /index.html break;
            }
      }
}
```

For https content it is possible to store all certificates in one place (/etc/nginx/SSL/<customer_name>), and use the same nginx hosts to view 503 maintenance pages for https. For that it is required to change port in listen section (80 → 443), and add ssl configuration to server section in virtual host configuration:

```
ssl on;
ssl_protocols       SSLv3 TLSv1;
ssl_certificate /etc/nginx/SSL/<customer_name>/server.pem;
ssl_certificate_key /etc/nginx/SSL/<customer_name>/server.key;
```

All client's configarution could be stored in two files (for http and https content) or separate host's configs can be creaed fot each customer.

9. **One General Maintenance Page but each customer can use their own Maintenance Page**
It can be described in global configuration file (/etc/haproxy/haproxy.cfg.def).
For example:

```
errorfile 503 /etc/haproxy/errors/503.http
```

This line will point to error page, that will be used globally for all customers, who donn't have custom error pages.

10. **New visitors will see a heavy load page when it reach a to high amount of visitors**

Exmple:

```
        server <server_name> <some_ip_address>[:port] maxconn 1000 maxqueue 10
```

maxconn <maxconn> 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.
maxqueue <maxqueue> - 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.

In this case server will handle 1000 connection, next 10 will be queried, and others will be sent to next node in cluster. Next node will use the same algorithm and will continue till last server in cluster will be full. We can assign server with high load page to be the last in backend/listen section, that will be distpaleyd for extra connections when servers will reach their connection limit, that is established by haproxy.

11. Management tools. Overview of active sessions (real time and historical).

There is stats mechanism that allows to see cluster conditions in real time using web browser. To enable it add next lines to config file in backend/listen section:

```
stats enable
        stats realm LoadBalancer_statistics
        stats auth admin:adminpassword
        stats scope <frontend_name>
        stats scope <back_end_name1>
        stats scope <back_end_name2>
        stats scope <dafault_back_end_name>
        stats uri /stats
```

Statistics will be availiable using http://server-address.nl/stats using login admin and password adminpassword

There is no tool to show for historical ssessions overview exap haproxy logs.
log <address> <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).

        - 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).

  <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
```

There is log fasility described in globas section of default config:

```
        log 127.0.0.1:514   local0 #will send hold all info
        log 127.0.0.1:514   local1  notice  info #will hold notice and info level
        log 127.0.0.1:514   local2  emerg  alert #will hold light errors
        log 127.0.0.1:514   local3  crit warning #will hold hard errors
```

It points to empty space to make rsyslog write haproxy logs we need to create configuration file for haproxy logs:

```
nano /etc/rsyslog.d/haproxy.conf
```

Paste next there:

```
# Save HA-Proxy logs
$ModLoad imudp
$UDPServerRun 514
$UDPServerAddress 127.0.0.1

local0.*          -/var/log/haproxy/haproxy_all.log
local1.*          -/var/log/haproxy/haproxy_info.log
local2.*          -/var/log/haproxy/haproxy_lerror.log
local3.*          -/var/log/haproxy/haproxy_herror.log
&~
# & ~ means not to put what matched in the above line anywhere else for the rest of the
#rules
```

create directory for haproxy logs:

```
mkdir /var/log/haproxy/
```

12. **Config of different customers does not influence each other :**
By default haproxy can'nt use multiple configuration files, or/and include multiple files for work, like apache does. To make it possible to use separate configs per custome we'll need to build new configuration and start

haproxy with that configuration. To make that we'll need to edit daemon starter with confi builder cicle and some additional variables.

Variables:

```
#This is path to directory where client's configs are stored
CONFIG_DIR=/etc/haproxy/Clients
#File that has global and default haproxy parametres
CONFIG_DEF=/etc/haproxy/haproxy.cfg.def
#Saved working configuration
CONFIG_SAVE=/etc/haproxy/haproxy.cfg.save
#Configuration that failed haproxy to start
CONFIG_FAIL=/etc/haproxy/haproxy.cfg.debug
#Config file that is going to be build
```

Config builder itself:

```
#This part makes it possible to use separate configs for every customer
#Next command reads all cfg files from customer's dirrectory
#and combines them into one config file,
#Renaming current haproxy configuration file, just in case if something goes wrong
          mv $CONFIG $CONFIG_SAVE
#Creating default config
          cp $CONFIG_DEF $CONFIG
#Reading config files from Client's directory
        client=$(ls $CONFIG_DIR/*.cfg)
        cd $CONFIG_DIR
        for cfg in $client;
        do
#Make config file more comfortable:
        echo " " >> $CONFIG
        echo "#Using $cfg config" >> $CONFIG
        cat $cfg >>  $CONFIG
        done;
```

It is important to realize that we don't have any working static configuration file for our loadballance, because it is being compiled during haproxy start, so we can't check it in regular way after making any kind of changes.

After new file compilation it is highly important to check it on flow, and start haproxy only if it is fine, if it is not – restore working config, – we don't need our loadballancer to fail. For that one more sickle is used and some additional variables are needed:

```
#Check config file, if there are no errors - starts haproxy.
#If there are errors in configuration - shows them
#than brings back previously saved configuration file and starts haproxy with it
#
#Checking config file - we donn't want our live haproxy to fail. Steps to take:
# - check compiled config file for errors if any;
# - if everything is OK - goto last step;
# - show number of errors and display errors if any;
# - use previous config if errors found;
# - goto starting haproxy;
test_condition=$(haproxy -c -f $CONFIG 2>&1 |grep arsing |wc -l)
if [ $test_condition != "0" ]
        then
        echo " * There are $test_condition error(s) in configuration" && echo " "
        haproxy -c -f $CONFIG 2>&1 |grep parsing && echo " "
        echo " "
        echo " * Using previously saved config"
        mv $CONFIG_SAVE $CONFIG
```

```
      echo " "
      haproxy -c -f $CONFIG 2>&1 && echo " "
      else echo " " && echo " * Config file is OK" && echo " "
fi
```

Little discription of test_condition.

    test_condition=$(haproxy -c -f $CONFIG 2>&1 |grep arsing |wc -l)

it runs regular haproxy configuration check procedure after new config file is created:

    haproxy -c -f /etc/haproxy/haproxy.cfg

If any errors are found it brings back stderr(2) with messages about parsing config file, and a “Fatal errors found in configuration” message. The number of “parsing” will indicate the number of mistakes in configurations file, so they are captured with “grep”, and counted with “wc -l”.

But it is not he only check that is important. Due to bug in haproxy it doesn't check sockets that are going to be created (<listen_ip>:<>listen_port>, eg. 192.168.1.2:80) for each listen section in our configuration. This can create trouble on haproxy starting daemon step, stderr(2) will be shown and haproxy will fail to start.

Daemon starter is a regular shell script. It has:

- section that describes global variables;
- descriptions of start, stop, restart and reload procedures;
- and case condition on operations with daemon (CCOWD), depending on stdin (0);

CCOWD is responsible for proper functions call. It reads start, stop, restart and reload commands and makes daemon function call, after function is done it brings stderr or stdout, depending on stuff that happens on procedure run. And if there is any socket duplicating it will bring error with messages about parsing config file, here is one more place and one more condition where we'll need to catch errors, and bring back previous configuration for haprohy. There is no need to catch the error message the fact of stderr is condition itself.
So we'll need to find lines that are responsible for stderr (2) at start, restart and reload cases and put one more operation in it.
This is a section to find:

```
2)
 log_end_msg 1
```

This is a operation with configuration file. It needs to be placed to that section, before “;;” closing tag:

```
        mv $CONFIG $CONFIG_FAIL
        echo " " && echo " * Use $CONFIG_FAIL to debug"
        echo " " && echo " * Using previously saved config"
        cp $CONFIG_SAVE $CONFIG
        echo " "
        haproxy -c -f $CONFIG 2>&1
        log_daemon_msg "Trying to start haproxy one more" "haproxy"
        haproxy_start
              case "$?" in
                 0)
                       log_end_msg 0
                       ;;
                 1)
                       log_end_msg 1
                       ;;
                 2)
                       log_end_msg 1
                       ;;
                 esac
```