# [**https://upcloud.com/resources/tutorials/haproxy-load-balancer-centos**](https://upcloud.com/resources/tutorials/haproxy-load-balancer-centos)

**OR**

[**https://www.digitalocean.com/community/tutorials/how-to-use-haproxy-to-set-up-http-load-balancing-on-an-ubuntu-vps**](https://www.digitalocean.com/community/tutorials/how-to-use-haproxy-to-set-up-http-load-balancing-on-an-ubuntu-vps)

—----------------------------------------------------------------------------------------------------------------------------

# **Introduction to HAProxy**

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

## **What HAProxy is**

## HAProxy is :

- a TCP proxy : it can accept a TCP connection from a listening socket,

connect to a server and attach these sockets together allowing traffic to flow in both directions; IPv4, IPv6 and even UNIX sockets are supported on either side, so this can provide an easy way to translate addresses between different families.

- an HTTP reverse-proxy (called a "gateway" in HTTP terminology) : it presents itself as a server, receives HTTP requests over connections accepted on a listening TCP socket, and passes the requests from these connections to servers using different connections. It may use any combination of HTTP/1.x or HTTP/2 on any side and will even automatically detect the protocol spoken on each side when ALPN is used over TLS.

- an SSL terminator / initiator / offloader : SSL/TLS may be used on the

connection coming from the client, on the connection going to the server,

or even on both connections. A lot of settings can be applied per name

(SNI), and may be updated at runtime without restarting. Such setups are

extremely scalable and deployments involving tens to hundreds of thousands

of certificates were reported.

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

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

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

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

- a server load balancer : it can load balance TCP connections and HTTP

requests. In TCP mode, load balancing decisions are taken for the whole

connection. In HTTP mode, decisions are taken per request.

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

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

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

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

- a caching proxy : it may cache responses in RAM so that subsequent requests for the same object avoid the cost of another network transfer from the server as long as the object remains present and valid. It will however not store objects to any persistent storage. Please note that this caching feature is designed to be maintenance free and focuses solely on saving haproxy's precious resources and not save the server's resources. Caches designed to optimize servers require much more tuning and flexibility. If you instead need such an advanced cache, please use Varnish Cache, which integrates perfectly with haproxy, especially when SSL/TLS is needed on any side.

- a FastCGI gateway : FastCGI can be seen as a different representation of HTTP, and as such, HAProxy can directly load-balance a farm comprising any combination of FastCGI application servers without requiring to insert

another level of gateway between them. This results in resource savings and a reduction of maintenance costs.

## 

## **How HAProxy works**

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

keep-alive mode.

Once HAProxy is started, it does exactly 3 things :

- process incoming connections;

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

- exchange information with other haproxy nodes.

## **Basic features**

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

### [**3.3.1.**](https://www.haproxy.com/documentation/hapee/latest/onepage/intro/#3.3.1) **Basic features : Proxying**

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

- Provide the server with a clean connection to protect them against any

client-side defect or attack;

- Listen to multiple IP addresses and/or ports, even port ranges;

- Transparent accept : intercept traffic targeting any arbitrary IP address that doesn't even belong to the local system;

- Server port doesn't need to be related to listening port, and may even be translated by a fixed offset (useful with ranges);

- Transparent connect : spoof the client's (or any) IP address if needed

when connecting to the server;

- Provide a reliable return IP address to the servers in multi-site LBs;

- Offload the server thanks to buffers and possibly short-lived connections to reduce their concurrent connection count and their memory footprint;

- Optimize TCP stacks (e.g. SACK), congestion control, and reduce RTT impacts;

- Support different protocol families on both sides (e.g. IPv4/IPv6/Unix);

- Timeout enforcement : HAProxy supports multiple levels of timeouts depending on the stage the connection is, so that a dead client or server, or an attacker cannot be granted resources for too long;

- Protocol validation: HTTP, SSL, or payload are inspected and invalid

protocol elements are rejected, unless instructed to accept them anyway;

- Policy enforcement : ensure that only what is allowed may be forwarded;

- Both incoming and outgoing connections may be limited to certain network namespaces (Linux only), making it easy to build a cross-container,multi-tenant load balancer;

- PROXY protocol presents the client's IP address to the server even for

non-HTTP traffic. This is an HAProxy extension that was adopted by a number of third-party products by now, at least these ones at the time of writing :

- client : haproxy, stud, stunnel, exaproxy, ELB, squid

- server : haproxy, stud, postfix, exim, nginx, squid, node.js, varnish

### [**3.3.2.**](https://www.haproxy.com/documentation/hapee/latest/onepage/intro/#3.3.2) **Basic features : SSL**

HAProxy's SSL stack is recognized as one of the most featureful according to Google's engineers (http://istlsfastyet.com/).

—-------------------------------------------------------------------------

# **Install and Configure HAProxy**

Perform the following procedure on your two HAProxy nodes:

1. Install haproxy.  
   # **yum install haproxy**

**2**. Configure haproxy for SELinux and HTTP.

# **vim /etc/firewalld/services/haproxy-http.xml**

Add the following lines:

<?xml version="1.0" encoding="utf-8"?>

<service>

<short>HAProxy-HTTP</short>

<description>HAProxy load-balancer</description>

<port protocol="tcp" port="80"/>

</service>

As root, assign the correct SELinux context and file permissions to the haproxy-http.xml file.

# **cd /etc/firewalld/services**

# **restorecon haproxy-http.xml**

# **chmod 640 haproxy-http.xml**

**3**. If you intend to use HTTPS, configure haproxy for SELinux and HTTPS.

# **vim /etc/firewalld/services/haproxy-https.xml**

Add the following lines:

<?xml version="1.0" encoding="utf-8"?>

<service>

<short>HAProxy-HTTPS</short>

<description>HAProxy load-balancer</description>

<port protocol="tcp" port="443"/>

</service>

As root, assign the correct SELinux context and file permissions to the haproxy-https.xml file.

# **cd /etc/firewalld/services**

# **restorecon haproxy-https.xml**

# **chmod 640 haproxy-https.xml**

**4**. If you intend to use HTTPS, generate keys for SSL. If you do not have a certificate, you may use a self-signed certificate. For information on generating keys and on self-signed certificates, see the Red Hat Enterprise Linux *System Administrator's Guide*.

Finally, put the certificate and key into a PEM file.

# **cat example.com.crt example.com.key > example.com.pem**

# **cp example.com.pem /etc/ssl/private/**

**5**. Configure HAProxy.

# **vim /etc/haproxy/haproxy.cfg**

The global and defaults sections of haproxy.cfg may remain unchanged. After the defaults sections, you will need to configure frontend and backend sections, as in the following example:

frontend http\_web \*:80

mode http

default\_backend rgw

frontend rgw­-https

bind <insert vip ipv4>:443 ssl crt /etc/ssl/private/example.com.pem

default\_backend rgw

backend rgw

balance roundrobin

mode http

server rgw1 10.0.0.71:80 check

server rgw2 10.0.0.80:80 check

**6**. Enable/start haproxy

# **systemctl enable haproxy**

# **systemctl start haproxy**

—-----------------------------------------------------------------------------------------------------

## **Installing and Configuring HAProxy**

To install HAProxy:

1. Install the haproxy package on each front-end server:  
   # **yum install haproxy**
2. Edit /etc/haproxy/haproxy.cfg to configure HAProxy on each server. See [Section 17.2.1, “About the HAProxy Configuration File”](https://docs.oracle.com/en/operating-systems/oracle-linux/6/admin/section_uqs_5mb_nr.html).

3. Enable IP forwarding and binding to non-local IP addresses:  
# **echo "net.ipv4.ip\_forward = 1" >> /etc/sysctl.conf**

# **echo "net.ipv4.ip\_nonlocal\_bind = 1" >> /etc/sysctl.conf**

# **sysctl -p**

net.ipv4.ip\_forward = 1

net.ipv4.ip\_nonlocal\_bind = 1

4. Enable access to the services or ports that you want HAProxy to handle.

For example, to enable access to HTTP and make this rule persist across reboots, enter the following commands:

# **iptables -I INPUT -p tcp -m state --state NEW -m tcp --dport 80 -j ACCEPT**

# **service iptables save**

5. Enable and start the haproxy service on each server:  
# **chkconfig haproxy on**

# **service haproxy start**

If you change the HAProxy configuration, reload the haproxy service:  
# **service haproxy reload**