HAProxy Starter Guide

version 1.6

This document is an introduction to HAProxy for all those who don't know it, as well as for those who want to re-discover it when they know older versions. Its primary focus is to provide users with all the elements to decide if HAProxy is the product they're looking for or not. Advanced users may find here some parts of solutions to some ideas they had just because they were not aware of a given new feature. Some sizing information are also provided, the product's lifecycle is explained, and comparisons with partially overlapping products are provided.

This document doesn't provide any configuration help nor hint, but it explains where to find the relevant documents. The summary below is meant to help you search sections by name and navigate through the document.

Note to documentation contributors :
    This document is formatted with 80 columns per line, with even number of spaces for indentation and without tabs. Please follow these rules strictly so that it remains easily printable everywhere. If you add sections, please update the summary below for easier searching.

Summary

```
1. Available documentation

2. Quick introduction to load balancing and load balancers

3.    Introduction to HAProxy
3.1.      What HAProxy is and is not
3.2.      How HAProxy works
3.3.      Basic features
3.3.1.        Proxying
3.3.2.        SSL
3.3.3.        Monitoring
3.3.4.        High availability
3.3.5.        Load balancing
3.3.6.        Stickiness
3.3.7.        Sampling and converting information
3.3.8.        Maps
3.3.9.        ACLs and conditions
3.3.10.       Content switching
3.3.11.       Stick-tables
3.3.12.       Formated strings
3.3.13.       HTTP rewriting and redirection
3.3.14.       Server protection
3.3.15.       Logging
3.3.16.       Statistics
3.4.      Advanced features
3.4.1.        Management
3.4.2.        System-specific capabilities
3.4.3.        Scripting
3.5.      Sizing
3.6.      How to get HAProxy

4.    Companion products and alternatives
4.1.      Apache HTTP server
4.2.      NGINX
4.3.      Varnish
4.4.      Alternatives
```

1. Available documentation

The complete HAProxy documentation is contained in the following documents. Please ensure to consult the relevant documentation to save time and to get the most accurate response to your needs. Also please refrain from sending questions to the mailing list whose responses are present in these documents.

- intro.txt (this document) : it presents the basics of load balancing, HAProxy as a product, what it does, what it doesn't do, some known traps to avoid, some OS-specific limitations, how to get it, how it evolves, how to ensure you're running with all known fixes how to update it, complements and alternatives.

- management.txt : it explains how to start haproxy, how to manage it at runtime, how to manage it on multiple nodes, how to proceed with seamless upgrades.

- configuration.txt : the reference manual details all configuration keywords and their options. It is used when a configuration change is needed.

- architecture.txt : the architecture manual explains how to best architect a load-balanced infrastructure and how to interact with third party products.

- coding-style.txt : this is for developers who want to propose some code to the project. It explains the style to adopt for the code. It's not very strict and not all the code base completely respects it but contributions which diverge too much from it will be rejected.

- proxy-protocol.txt : this is the de-facto specification of the PROXY protocol which is implemented by HAProxy and a number of third party products.

- README : how to build haproxy from sources

2. Quick introduction to load balancing and load balancers

Load balancing consists in aggregating multiple components in order to achieve a total processing capacity above each component's individual capacity, without any intervention from the end user and in a scalable way. This results in more operations being performed simultaneously by the time it takes a component to perform only one. A single operation however will still be performed on a single component at a time and will not get faster than without load balancing. It always requires at least as many operations as available components and an efficient load balancing mechanism to make use of all components and to fully benefit from the load balancing. A good example of this is the number of lanes on a highway which allows as many cars to pass during the same time frame without increasing their individual speed.

Examples of load balancing :

- Process scheduling in multi-processor systems
- Link load balancing (eg: EtherChannel, Bonding)
- IP address load balancing (eg: ECMP, DNS roundrobin)
- Server load balancing (via load balancers)

The mechanism or component which performs the load balancing operation is called a load balancer. In web environments these components are called a "network load balancer", and more commonly a "load balancer" given that this activity is by far the best known case of load balancing.

A load balancer may act :

- at the link level : this is called link load balancing, and it consists in chosing what network link to send a packet to;

- at the network level : this is called network load balancing, and it consists in chosing what route a series of packets will follow;

- at the server level : this is called server load balancing and it consists in deciding what server will process a connection or request.

Two distinct technologies exist and address different needs, though with some overlapping. In each case it is important to keep in mind that load balancing consists in diverting the traffic from its natural flow and that doing so always requires a minimum of care to maintain the required level of consistency between all routing decisions.

The first one acts at the packet level and processes packets more or less individually. There is a 1-to-1 relation between input and output packets, so it is possible to follow the traffic on both sides of the load balancer using a regular network sniffer. This technology can be very cheap and extremely fast. It is usually implemented in hardware (ASICs) allowing to reach line rate, such as switches doing ECMP. Usually stateless, it can also be stateful (consider the session a packet belongs to and called layer4-LB or L4), may support DSR (direct server return, without passing through the LB again) if the packets were not modified, but provides almost no content awareness. This technology is very well suited to network-level load balancing, though it is sometimes used for very basic server load balancing at high speed.

The second one acts on session contents. It requires that the input streams is reassembled and processed as a whole. The contents may be modified, and the output stream is segmented into new packets. For this reason it is generally performed by proxies and they're often called layer 7 load balancers or L7. This implies that there are two distinct connections on each side, and that there is no relation between input and output packets sizes nor counts. Clients and servers are not required to use the same protocol (for example IPv4 vs IPv6, clear vs SSL). The operations are always stateful, and the return traffic must pass through the load balancer. The extra processing comes with a cost so it's not always possible to achieve line rate, especially with small packets. On the other hand, it offers wide possibilities and is generally achieved by pure software, even if embedded into hardware appliances. This technology is very well suited for server load balancing.

Packet-based load balancers are generally deployed in cut-through mode, so they are installed on the normal path of the traffic and divert it according to the configuration. The return traffic doesn't necessarily pass through the load balancer. Some modifications may be applied to the network destination address in order to direct the traffic to the proper destination. In this case, it is mandatory that the return traffic passes through the load balancer. If the routes doesn't make this possible, the load balancer may also replace the packets' source address with its own in order to force the return traffic to pass through it.

Proxy-based load balancers are deployed as a server with their own IP address and ports, without architecture changes. Sometimes this requires to perform some adaptations to the applications so that clients are properly directed to the load balancer's IP address and not directly to the server's. Some load balancers may have to adjust some servers' responses to make this possible (eg: the HTTP Location header field used in HTTP redirects). Some proxy-based load balancers may intercept traffic for an address they don't own, and spoof the client's address when connecting to the server. This allows them to be deployed as if they were a regular router or firewall, in a cut-through mode very similar to the packet based load balancers. This is particularly appreciated for products which combine both packet mode and proxy mode. In this case DSR is obviously still not possible and the return traffic still has to be routed back to the load balancer.

A very scalable layered approach would consist in having a front router which receives traffic from multiple load balanced links, and uses ECMP to distribute this traffic to a first layer of multiple stateful packet-based load balancers (L4). These L4 load balancers in turn pass the traffic to an even larger number of proxy-based load balancers (L7), which have to parse the contents to decide what server will ultimately receive the traffic.

The number of components and possible paths for the traffic increases the risk of failure; in very large environments, it is even normal to permanently have a few faulty components being fixed or replaced. Load balancing done without awareness of the whole stack's health significantly degrades availability. For this reason, any sane load balancer will verify that the components it intends to deliver the traffic to are still alive and reachable, and it will stop delivering traffic to faulty ones. This can be achieved using various methods.

The most common one consists in periodically sending probes to ensure the component is still operational. These probes are called "health checks". They must be representative of the type of failure to address. For example a ping-based check will not detect that a web server has crashed and doesn't listen to a port anymore, while a connection to the port will verify this, and a more advanced request may even validate that the server still works and that the database it relies on is still accessible. Health checks often involve a few retries to cover for occasional measuring errors. The period between checks must be small enough to ensure the faulty component is not used for too long after an error occurs.

Other methods consist in sampling the production traffic sent to a destination to observe if it is processed correctly or not, and to evince the components which return inappropriate responses. However this requires to sacrify a part of the production traffic and this is not always acceptable. A combination of these two mechanisms provides the best of both worlds, with both of them being used to detect a fault, and only health checks to detect the end of the fault. A last method involves centralized reporting : a central monitoring agent periodically updates all load balancers about all components' state. This gives a global view of the infrastructure to all components, though sometimes with less accuracy or responsiveness. It's best suited for environments with many load balancers and many servers.

Layer 7 load balancers also face another challenge known as stickiness or persistence. The principle is that they generally have to direct multiple subsequent requests or connections from a same origin (such as an end user) to the same target. The best known example is the shopping cart on an online store. If each click leads to a new connection, the user must always be sent to the server which holds his shopping cart. Content-awareness makes it easier to spot some elements in the request to identify the server to deliver it to, but that's not always enough. For example if the source address is used as a key to pick a server, it can be decided that a hash-based algorithm will be used and that a given IP address will always be sent to the same server based on a divide of the address by the number of available servers. But if one server fails, the result changes and all users are suddenly sent to a different server and lose their shopping cart. The solution against this issue consists in memorizing the chosen target so that each time the same visitor is seen, he's directed to the same server regardless of the number of available servers. The information may be stored in the load balancer's memory, in which case it may have to be replicated to other load balancers if it's not alone, or it may be stored in the client's memory using various methods provided that the client is able to present this information back with every request (cookie insertion, redirection to a sub-domain, etc). This mechanism provides the extra benefit of not having to rely on unstable or unevenly distributed information (such as the source IP address). This is in fact the strongest reason to adopt a layer 7 load balancer instead of a layer 4 one.

In order to extract information such as a cookie, a host header field, a URL or whatever, a load balancer may need to decrypt SSL/TLS traffic and even possibly to reencrypt it when passing it to the server. This expensive task explains why in some high-traffic infrastructures, sometimes there may be a lot of load balancers.

Since a layer 7 load balancer may perform a number of complex operations on the traffic (decrypt, parse, modify, match cookies, decide what server to send to, etc), it can definitely cause some trouble and will very commonly be accused of being responsible for a lot of trouble that it only revealed. Often it will be discovered that servers are unstable and periodically go up and down, or for web servers, that they deliver pages with some hard-coded links forcing the clients to connect directly to one specific server without passing via the load balancer, or that they take ages to respond under high load causing timeouts. That's why logging is an extremely important aspect of layer 7 load balancing. Once a trouble is reported, it is important to figure if the load balancer took a wrong decision and if so why so that it doesn't happen anymore.

3. Introduction to HAProxy

HAProxy is written "HAProxy" to designate the product, "haproxy" to designate the executable program, software package or a process, though both are commonly used for both purposes, and is pronounced H-A-Proxy. Very early it 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".

3.1. What HAProxy is and is not

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;

- 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.

- 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 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 (eg: 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 (eg: 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.

HAProxy is not :

- an explicit HTTP proxy, ie, the proxy that browsers use to reach the internet. There are excellent open-source software dedicated for this task, such as Squid. However HAProxy can be installed in front of such a proxy to provide load balancing and high availability.

- a caching proxy : it will return as-is the contents its received from the server and will not interfere with any caching policy. There are excellent open-source software for this task such as Varnish. HAProxy can be installed in front of such a cache to provide SSL offloading, and scalability through smart load balancing.

- a data scrubber : it will not modify the body of requests nor responses.

- a web server : during startup, it isolates itself inside a chroot jail and drops its privileges, so that it will not perform any single file-system access once started. As such it cannot be turned into a web server. There are excellent open-source software for this such as Apache or Nginx, and HAProxy can be installed in front of them to provide load balancing and high availability.

- a packet-based load balancer : it will not see IP packets nor UDP datagrams, will not perform NAT or even less DSR. These are tasks for lower layers. Some kernel-based components such as IPVS (Linux Virtual Server) already do this pretty well and complement perfectly with HAProxy.

3.2. How HAProxy works

HAProxy is a single-threaded, event-driven, non-blocking engine combining a very fast I/O layer with a priority-based 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. As such it implements a layered model offering bypass mechanisms at each level ensuring data don't reach higher levels when not 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 operation 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.

A single process can run many proxy instances; configurations as large as 300000 distinct proxies in a single process were reported to run fine. Thus there is usually no need to start more than one process for all instances.

It is possible to make HAProxy run over multiple processes, but it comes with a few limitations. In general it doesn't make sense in HTTP close or TCP modes because the kernel-side doesn't scale very well with some operations such as connect(). It scales pretty well for HTTP keep-alive mode but the performance that can be achieved out of a single process generaly outperforms common needs by an order of magnitude. It does however make sense when used as an SSL offloader, and this feature is well supported in multi-process mode.

HAProxy only requires the haproxy executable and a configuration file to run. For logging it is highly recommended to have a properly configured syslog daemon and log rotations in place. The configuration files are parsed before starting, then HAProxy tries to bind all listening sockets, and refuses to start if anything fails. Past this point it cannot fail anymore. This means that there are no runtime failures and that if it accepts to start, it will work until it is stopped.

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.

Processing incoming connections is by far the most complex task as it depends on a lot of configuration possibilities, but it can be summarized as the 9 steps below :

- accept incoming connections from listening sockets that belong to a configuration entity known as a "frontend", which references one or multiple listening addresses;

- apply the frontend-specific processing rules to these connections that may result in blocking them, modifying some headers, or intercepting them to execute some internal applets such as the statistics page or the CLI;

- pass these incoming connections to another configuration entity representing a server farm known as a "backend", which contains the list of servers and the load balancing strategy for this server farm;

- apply the backend-specific processing rules to these connections;

- decide which server to forward the connection to according to the load balancing strategy;

- apply the backend-specific processing rules to the response data;

- apply the frontend-specific processing rules to the response data;

- emit a log to report what happened in fine details;

- in HTTP, loop back to the second step to wait for a new request, otherwise close the connection.

Frontends and backends are sometimes considered as half-proxies, since they only look at one side of an end-to-end connection; the frontend only cares about the clients while the backend only cares about the servers. HAProxy also supports full proxies which are exactly the union of a frontend and a backend. When HTTP processing is desired, the configuration will generally be split into frontends

and backends as they open a lot of possibilities since any frontend may pass a connection to any backend. With TCP-only proxies, using frontends and backends rarely provides a benefit and the configuration can be more readable with full proxies.

3.3. 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. Basic features : Proxying

Proxying is the action of transferring data between a client and a server over two independant 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 address and/or ports, even port ranges;

- Transparent accept : intercept traffic targetting 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 (eg: SACK), congestion control, and reduce RTT impacts;

- Support different protocol families on both sides (eg: 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. Basic features : SSL

HAProxy's SSL stack is recognized as one of the most featureful according to Google's engineers (http://istlsfastyet.com/). The most commonly used features making it quite complete are :

- SNI-based multi-hosting with no limit on sites count and focus on performance. At least one deployment is known for running 50000 domains with their respective certificates;
- support for wildcard certificates reduces the need for many certificates ;
- certificate-based client authentication with configurable policies on failure to present a valid certificate. This allows to present a different server farm to regenerate the client certificate for example;
- authentication of the backend server ensures the backend server is the real one and not a man in the middle;
- authentication with the backend server lets the backend server it's really the expected haproxy node that is connecting to it;
- TLS NPN and ALPN extensions make it possible to reliably offload SPDY/HTTP2 connections and pass them in clear text to backend servers;
- OCSP stapling further reduces first page load time by delivering inline an OCSP response when the client requests a Certificate Status Request;
- Dynamic record sizing provides both high performance and low latency, and significantly reduces page load time by letting the browser start to fetch new objects while packets are still in flight;
- permanent access to all relevant SSL/TLS layer information for logging, access control, reporting etc... These elements can be embedded into HTTP header or even as a PROXY protocol extension so that the offloaded server gets all the information it would have had if it performed the SSL termination itself.
- Detect, log and block certain known attacks even on vulnerable SSL libs, such as the Heartbleed attack affecting certain versions of OpenSSL.
- support for stateless session resumption (RFC 5077 TLS Ticket extension). TLS tickets can be updated from CLI which provides them means to implement Perfect Forward Secrecy by frequently rotating the tickets.

3.3.3. Basic features : Monitoring

HAProxy focuses a lot on availability. As such it cares about servers state, and about reporting its own state to other network components :

- Servers state is continuously monitored using per-server parameters. This ensures the path to the server is operational for regular traffic;
- Health checks support two hysteresis for up and down transitions in order to protect against state flapping;
- Checks can be sent to a different address/port/protocol : this makes it easy to check a single service that is considered representative of multiple ones, for example the HTTPS port for an HTTP+HTTPS server.
- Servers can track other servers and go down simultaneously : this ensures that servers hosting multiple services can fail atomically and that noone

will be sent to a partially failed server;

- Agents may be deployed on the server to monitor load and health : a server may be interested in reporting its load, operational status, administrative status independantly from what health checks can see. By running a simple agent on the server, it's possible to consider the server's view of its own health in addition to the health checks validating the whole path;
- Various check methods are available : TCP connect, HTTP request, SMTP hello, SSL hello, LDAP, SQL, Redis, send/expect scripts, all with/without SSL;
- State change is notified in the logs and stats page with the failure reason (eg: the HTTP response received at the moment the failure was detected). An e-mail can also be sent to a configurable address upon such a change ;
- Server state is also reported on the stats interface and can be used to take routing decisions so that traffic may be sent to different farms depending on their sizes and/or health (eg: loss of an inter-DC link);
- HAProxy can use health check requests to pass information to the servers, such as their names, weight, the number of other servers in the farm etc... so that servers can adjust their response and decisions based on this knowledge (eg: postpone backups to keep more CPU available);
- Servers can use health checks to report more detailed state than just on/off (eg: I would like to stop, please stop sending new visitors);
- HAProxy itself can report its state to external components such as routers or other load balancers, allowing to build very complete multi-path and multi-layer infrastructures.

3.3.4. Basic features : High availability

Just like any serious load balancer, HAProxy cares a lot about availability to ensure the best global service continuity :

- Only valid servers are used ; the other ones are automatically evinced from load balancing farms ; under certain conditions it is still possible to force to use them though;
- Support for a graceful shutdown so that it is possible to take servers out of a farm without affecting any connection;
- Backup servers are automatically used when active servers are down and replace them so that sessions are not lost when possible. This also allows to build multiple paths to reach the same server (eg: multiple interfaces);
- Ability to return a global failed status for a farm when too many servers are down. This, combined with the monitoring capabilities makes it possible for an upstream component to choose a different LB node for a given service;
- Stateless design makes it easy to build clusters : by design, HAProxy does its best to ensure the highest service continuity without having to store information that could be lost in the event of a failure. This ensures that a takeover is the most seamless possible;
- Integrates well with standard VRRP daemon keepalived : HAProxy easily tells keepalived about its state and copes very will with floating virtual IP addresses. Note: only use IP redundancy protocols (VRRP/CARP) over cluster-based solutions (Heartbeat, ...) as they're the ones offering the fastest, most seamless, and most reliable switchover.

3.3.5. Basic features : Load balancing

HAProxy offers a fairly complete set of load balancing features, most of which are unfortunately not available in a number of other load balancing products :

- no less than 9 load balancing algorithms are supported, some of which apply to input data to offer an infinite list of possibilities. The most common ones are round-robin (for short connections, pick each server in turn), leastconn (for long connections, pick the least recently used of the servers with the lowest connection count), source (for SSL farms or terminal server farms, the server directly depends on the client's source address), uri (for HTTP caches, the server directly depends on the HTTP URI), hdr (the server directly depends on the contents of a specific HTTP header field), first (for short-lived virtual machines, all connections are packed on the smallest possible subset of servers so that unused ones can be powered down);

- all algorithms above support per-server weights so that it is possible to accommodate from different server generations in a farm, or direct a small fraction of the traffic to specific servers (debug mode, running the next version of the software, etc);

- dynamic weights are supported for round-robin, leastconn and consistent hashing ; this allows server weights to be modified on the fly from the CLI or even by an agent running on the server;

- slow-start is supported whenever a dynamic weight is supported; this allows a server to progressively take the traffic. This is an important feature for fragile application servers which require to compile classes at runtime as well as cold caches which need to fill up before being run at full throttle;

- hashing can apply to various elements such as client's source address, URL components, query string element, header field values, POST parameter, RDP cookie;

- consistent hashing protects server farms against massive redistribution when adding or removing servers in a farm. That's very important in large cache farms and it allows slow-start to be used to refill cold caches;

- a number of internal metrics such as the number of connections per server, per backend, the amount of available connection slots in a backend etc makes it possible to build very advanced load balancing strategies.

3.3.6. Basic features : Stickiness

Application load balancing would be useless without stickiness. HAProxy provides a fairly comprehensive set of possibilities to maintain a visitor on the same server even across various events such as server addition/removal, down/up cycles, and some methods are designed to be resistant to the distance between multiple load balancing nodes in that they don't require any replication :

- stickiness information can be individually matched and learned from different places if desired. For example a JSESSIONID cookie may be matched both in a cookie and in the URL. Up to 8 parallel sources can be learned at the same time and each of them may point to a different stick-table;

- stickiness information can come from anything that can be seen within a request or response, including source address, TCP payload offset and length, HTTTP query string elements, header field values, cookies, and so on...

- stick-tables are replicated between all nodes in a multi-master fashion ;

- commonly used elements such as SSL-ID or RDP cookies (for TSE farms) are directly accessible to ease manipulation;

- all sticking rules may be dynamically conditionned by ACLs;

- it is possible to decide not to stick to certain servers, such as backup servers, so that when the nominal server comes back, it automatically takes the load back. This is often used in multi-path environments;

- in HTTP it is often prefered not to learn anything and instead manipulate a cookie dedicated to stickiness. For this, it's possible to detect, rewrite, insert or prefix such a cookie to let the client remember what server was assigned;

- the server may decide to change or clean the stickiness cookie on logout, so that leaving visitors are automatically unbound from the server;

- using ACL-based rules it is also possible to selectively ignore or enforce stickiness regardless of the server's state; combined with advanced health checks, that helps admins verify that the server they're installing is up and running before presenting it to the whole world;

- an innovative mechanism to set a maximum idle time and duration on cookies ensures that stickiness can be smoothly stopped on devices which are never closed (smartphones, TVs, home appliances) without having to store them on persistent storage;

- multiple server entries may share the same stickiness keys so that stickiness is not lost in multi-path environments when one path goes down;

- soft-stop ensures that only users with stickiness information will continue to reach the server they've been assigned to but no new users will go there.

3.3.7. Basic features : Sampling and converting information

HAProxy supports information sampling using a wide set of "sample fetch functions". The principle is to extract pieces of information known as samples, for immediate use. This is used for stickiness, to build conditions, to produce information in logs or to enrich HTTP headers.

Samples can be fetched from various sources :

- constants : integers, strings, IP addresses, binary blocks;

- the process : date, environment variables, server/frontend/backend/process state, byte/connection counts/rates, queue length, random generator, ...

- variables : per-session, per-request, per-response variables;

- the client connection : source and destination addresses and ports, and all related statistics counters;

- the SSL client session : protocol, version, algorithm, cipher, key size, session ID, all client and server certificate fields, certificate serial, SNI, ALPN, NPN, client support for certain extensions;

- request and response buffers contents : arbitrary payload at offset/length, data length, RDP cookie, decoding of SSL hello type, decoding of TLS SNI;

- HTTP (request and response) : method, URI, path, query string arguments, status code, headers values, positionnal header value, cookies, captures, authentication, body elements;

A sample may then pass through a number of operators known as "converters" to experience some transformation. A converter consumes a sample and produces a new one, possibly of a completely different type. For example, a converter may be used to return only the integer length of the input string, or could turn a string to upper case. Any arbitrary number of converters may be applied in series to a sample before final use. Among all available sample converters, the following ones are the most commonly used :

- arithmetic and logic operators : they make it possible to perform advanced computation on input data, such as computing ratios, percentages or simply converting from one unit to another one;

- IP address masks are useful when some addresses need to be grouped by larger networks;

- data representation : url-decode, base64, hex, JSON strings, hashing;

- string conversion : extract substrings at fixed positions, fixed length, extract specific fields around certain delimiters, extract certain words, change case, apply regex-based substitution ;

- date conversion : convert to http date format, convert local to UTC and conversely, add or remove offset;

- lookup an entry in a stick table to find statistics or assigned server;

- map-based key-to-value conversion from a file (mostly used for geolocation).

3.3.8. Basic features : Maps

Maps are a powerful type of converter consisting in loading a two-columns file into memory at boot time, then looking up each input sample from the first column and either returning the corresponding pattern on the second column if the entry was found, or returning a default value. The output information also being a sample, it can in turn experience other transformations including other map lookups. Maps are most commonly used to translate the client's IP address to an AS number or country code since they support a longest match for network addresses but they can be used for various other purposes.

Part of their strength comes from being updatable on the fly either from the CLI or from certain actions using other samples, making them capable of storing and retrieving information between subsequent accesses. Another strength comes from the binary tree based indexation which makes them extremely fast event when they contain hundreds of thousands of entries, making geolocation very cheap and easy to set up.

3.3.9. Basic features : ACLs and conditions

Most operations in HAProxy can be made conditional. Conditions are built by combining multiple ACLs using logic operators (AND, OR, NOT). Each ACL is a series of tests based on the following elements :

- a sample fetch method to retrieve the element to test ;

- an optional series of converters to transform the element ;

- a list of patterns to match against ;

- a matching method to indicate how to compare the patterns with the sample

For example, the sample may be taken from the HTTP "Host" header, it could then be converted to lower case, then matched against a number of regex patterns using the regex matching method.

Technically, ACLs are built on the same core as the maps, they share the exact same internal structure, pattern matching methods and performance. The only real difference is that instead of returning a sample, they only return "found" or or "not found". In terms of usage, ACL patterns may be declared inline in the configuration file and do not require their own file. ACLs may be named for ease of use or to make configurations understandable. A named ACL may be declared multiple times and it will evaluate all definitions in turn until one matches.

About 13 different pattern matching methods are provided, among which IP address mask, integer ranges, substrings, regex. They work like functions, and just like with any programming language, only what is needed is evaluated, so when a condition involving an OR is already true, next ones are not evaluated, and similarly when a condition involving an AND is already false, the rest of the condition is not evaluated.

There is no practical limit to the number of declared ACLs, and a handful of commonly used ones are provided. However experience has shown that setups using a lot of named ACLs are quite hard to troubleshoot and that sometimes using anynmous ACLs inline is easier as it requires less references out of the scope being analysed.

3.3.10. Basic features : Content switching

HAProxy implements a mechanism known as content-based switching. The principle is that a connection or request arrives on a frontend, then the information carried with this request or connection are processed, and at this point it is possible to write ACLs-based conditions making use of these information to decide what backend will process the request. Thus the traffic is directed to one backend or another based on the request's contents. The most common example consists in using the Host header and/or elements from the path (sub-directories or file-name extensions) to decide whether an HTTP request targets a static object or the application, and to route static objects traffic to a backend made of fast and light servers, and all the remaining traffic to a more complex application server, thus constituting a fine-grained virtual hosting solution. This is quite convenient to make multiple technologies coexist as a more global solution.

Another use case of content-switching consists in using different load balancing algorithms depending on various criteria. A cache may use a URI hash while an application would use round robin.

Last but not least, it allows multiple customers to use a small share of a common resource by enforcing per-backend (thus per-customer connection limits).

Content switching rules scale very well, though their performance may depend on the number and complexity of the ACLs in use. But it is also possible to write dynamic content switching rules where a sample value directly turns into a backend name and without making use of ACLs at all. Such configurations have been reported to work fine at least with 300000 backends in production.

3.3.11. Basic features : Stick-tables

Stick-tables are commonly used to store stickiness information, that is, to keep a reference to the server a certain visitor was directed to. The key is then the identifier associated with the visitor (its source address, the SSL ID of the connection, an HTTP or RDP cookie, the customer number extracted from the URL or from the payload, ...) and the stored value is then the server's identifier.

Stick tables may use 3 different types of samples for their keys : integers, strings and addresses. Only one stick-table may be referenced in a proxy, and it is designated everywhere with the proxy name. Up to 8 key may be tracked in parallel. The server identifier is committed during request or response processing once both the key and the server are known.

Stick-table contents may be replicated in active-active mode with other HAProxy nodes known as "peers" as well as with the new process during a reload operation so that all load balancing nodes share the same information and take the same routing decision if a client's requests are spread over multiple nodes.

Since stick-tables are indexed on what allows to recognize a client, they are often also used to store extra information such as per-client statistics. The extra statistics take some extra space and need to be explicitly declared. The type of statistics that may be stored includes the input and output bandwidth, the number of concurrent connections, the connection rate and count over a period, the amount and frequency of errors, some specific tags and counters, etc... In order to support keeping such information without being forced to stick to a given server, a special "tracking" feature is implemented and allows to track up to 3 simultaneous keys from different tables at the same time regardless of stickiness rules. Each stored statistics may be searched, dumped and cleared from the CLI and adds to the live troubleshooting capabilities.

While this mechanism can be used to surclass a returning visitor or to adjust the delivered quality of service depending on good or bad behaviour, it is mostly used to fight against service abuse and more generally DDoS as it allows to build complex models to detect certain bad behaviours at a high processing speed.

3.3.12. Basic features : Formated strings

There are many places where HAProxy needs to manipulate character strings, such as logs, redirects, header additions, and so on. In order to provide the greatest flexibility, the notion of formated strings was introduced, initially for logging purposes, which explains why it's still called "log-format". These strings contain escape characters allowing to introduce various dynamic data including variables and sample fetch expressions into strings, and even to adjust the encoding while the result is being turned into a string (for example, adding quotes). This provides a powerful way to build header contents or to customize log lines. Additionally, in order to remain simple to build most common strings, about 50 special tags are provided as shortcuts for information commonly used in logs.

3.3.13. Basic features : HTTP rewriting and redirection

Installing a load balancer in front of an application that was never designed for this can be a challenging task without the proper tools. One of the most commonly requested operation in this case is to adjust requests and response headers to make the load balancer appear as the origin server and to fix hard coded information. This comes with changing the path in requests (which is strongly advised against), modifying Host header field, modifying the Location response header field for redirects, modifying the path and domain attribute for cookies, and so on. It also happens that a number of servers are somewhat verbose and tend to leak too much information in the response, making them more vulnerable to targetted attacks. While it's theorically not the role of a load balancer to clean this up, in practice it's located at the best place in the infrastructure to guarantee that everything is cleaned up.

Similarly, sometimes the load balancer will have to intercept some requests and

respond with a redirect to a new target URL. While some people tend to confuse redirects and rewriting, these are two completely different concepts, since the rewriting makes the client and the server see different things (and disagree on the location of the page being visited) while redirects ask the client to visit the new URL so that it sees the same location as the server.

In order to do this, HAProxy supports various possibilities for rewriting and redirect, among which :

- regex-based URL and header rewriting in requests and responses. Regex are the most commonly used tool to modify header values since they're easy to manipulate and well understood;

- headers may also be appended, deleted or replaced based on formated strings so that it is possible to pass information there (eg: client side TLS algorithm and cipher);

- HTTP redirects can use any 3xx code to a relative, absolute, or completely dynamic (formated string) URI;

- HTTP redirects also support some extra options such as setting or clearing a specific cookie, dropping the query string, appending a slash if missing, and so on;

- all operations support ACL-based conditions;

3.3.14. Basic features : Server protection

HAProxy does a lot to maximize service availability, and for this it deploys large efforts to protect servers against overloading and attacks. The first and most important point is that only complete and valid requests are forwarded to the servers. The initial reason is that HAProxy needs to find the protocol elements it needs to stay synchronized with the byte stream, and the second reason is that until the request is complete, there is no way to know if some elements will change its semantics. The direct benefit from this is that servers are not exposed to invalid or incomplete requests. This is a very effective protection against slowloris attacks, which have almost no impact on HAProxy.

Another important point is that HAProxy contains buffers to store requests and responses, and that by only sending a request to a server when it's complete and by reading the whole response very quickly from the local network, the server side connection is used for a very short time and this preserves server resources as much as possible.

A direct extension to this is that HAProxy can artificially limit the number of concurrent connections or outstanding requests to a server, which guarantees that the server will never be overloaded even if it continuously runs at 100% of its capacity during traffic spikes. All excess requests will simply be queued to be processed when one slot is released. In the end, this huge resource savings most often ensures so much better server response times that it ends up actually being faster than by overloading the server. Queued requests may be redispatched to other servers, or even aborted in queue when the client aborts, which also protects the servers against the "reload effect", where each click on "reload" by a visitor on a slow-loading page usually induces a new request and maintains the server in an overloaded state.

The slow-start mechanism also protects restarting servers against high traffic levels while they're still finalizing their startup or compiling some classes.

Regarding the protocol-level protection, it is possible to relax the HTTP parser to accept non stardard-compliant but harmless requests or responses and even to fix them. This allows bogus applications to be accessible while a fix is being developped. In parallel, offending messages are completely captured with a detailed report that help developers spot the issue in the application. The most dangerous protocol violations are properly detected and dealt with and fixed. For example malformed requests or responses with two Content-length headers are either fixed if the values are exactly the same, or rejected if they differ, since it becomes a security problem. Protocol inspection is not limited to HTTP, it is also available for other protocols like TLS or RDP.

When a protocol violation or attack is detected, there are various options to respond to the user, such as returning the common "HTTP 400 bad request", closing the connection with a TCP reset, faking an error after a long delay ("tarpit") to confuse the attacker. All of these contribute to protecting the servers by discouraging the offending client from pursuing an attack that becomes very expensive to maintain.

HAProxy also proposes some more advanced options to protect against accidental data leaks and session crossing. Not only it can log suspicious server responses but it will also log and optionally block a response which might affect a given visitors' confidentiality. One such example is a cacheable cookie appearing in a cacheable response and which may result in an intermediary cache to deliver it to another visitor, causing an accidental session sharing.

3.3.15. Basic features : Logging

Logging is an extremely important feature for a load balancer, first because a load balancer is often accused of the trouble it reveals, and second because it is placed at a critical point in an infrastructure where all normal and abnormal activity needs to be analysed and correlated with other components.

HAProxy provides very detailed logs, with millisecond accuracy and the exact connection accept time that can be searched in firewalls logs (eg: for NAT correlation). By default, TCP and HTTP logs are quite detailed an contain everything needed for troubleshooting, such as source IP address and port, frontend, backend, server, timers (request receipt duration, queue duration, connection setup time, response headers time, data transfer time), global process state, connection counts, queue status, retries count, detailed stickiness actions and disconnect reasons, header captures with a safe output encoding. It is then possible to extend or replace this format to include any sampled data, variables, captures, resulting in very detailed information. For example it is possible to log the number cumulated requests for this client or the number of different URLs for the client.

The log level may be adjusted per request using standard ACLs, so it is possible to automatically silent some logs considered as pollution and instead raise warnings when some abnormal behaviour happen for a small part of the traffic (eg: too many URLs or HTTP errors for a source address). Administrative logs are also emitted with their own levels to inform about the loss or recovery of a server for example.

Each frontend and backend may use multiple independant log outputs, which eases multi-tenancy. Logs are preferably sent over UDP, maybe JSON-encoded, and are truncated after a configurable line length in order to guarantee delivery.

3.3.16. Basic features : Statistics

HAProxy provides a web-based statistics reporting interface with authentication, security levels and scopes. It is thus possible to provide each hosted customer with his own page showing only his own instances. This page can be located in a hidden URL part of the regular web site so that no new port needs to be opened. This page may also report the availability of other HAProxy nodes so that it is easy to spot if everything works as expected at a glance. The view is synthetic with a lot of details accessible (such as error causes, last access and last

change duration, etc), which are also accessible as a CSV table that other tools may import to draw graphs. The page may self-refresh to be used as a monitoring page on a large display. In administration mode, the page also allows to change server state to ease maintenance operations.

3.4. Advanced features

3.4.1. Advanced features : Management

HAProxy is designed to remain extremely stable and safe to manage in a regular production environment. It is provided as a single executable file which doesn't require any installation process. Multiple versions can easily coexist, meaning that it's possible (and recommended) to upgrade instances progressively by order of criticity instead of migrating all of them at once. Configuration files are easily versionned. Configuration checking is done off-line so it doesn't require to restart a service that will possibly fail. During configuration checks, a number of advanced mistakes may be detected (eg: for example, a rule hiding another one, or stickiness that will not work) and detailed warnings and configuration hints are proposed to fix them. Backwards configuration file compatibility goes very far away in time, with version 1.5 still fully supporting configurations for versions 1.1 written 13 years before, and 1.6 only dropping support for almost unused, obsolete keywords that can be done differently. The configuration and software upgrade mechanism is smooth and non disruptive in that it allows old and new processes to coexist on the system, each handling its own connections. System status, build options and library compatibility are reported on startup.

Some advanced features allow an application administrator to smoothly stop a server, detect when there's no activity on it anymore, then take it off-line, stop it, upgrade it and ensure it doesn't take any traffic while being upgraded, then test it again through the normal path without opening it to the public, and all of this without touching HAProxy at all. This ensures that even complicated production operations may be done during opening hours with all technical resources available.

The process tries to save resources as much as possible, uses memory pools to save on allocation time and limit memory fragmentation, releases payload buffers as soon as their contents are sent, and supports enforcing strong memory limits above which connections have to wait for a buffer to become available instead of allocating more memory. This system helps guarantee memory usage in certain strict environments.

A command line interface (CLI) is available as a UNIX or TCP socket, to perform a number of operations and to retrieve troubleshooting information. Everything done on this socket doesn't require a configuration change, so it is mostly used for temporary changes. Using this interface it is possible to change a server's address, weight and status, to consult statistics and clear counters, dump and clear stickiness tables, possibly selectively by key criteria, dump and kill client-side and server-side connections, dump captured errors with a detailed analysis of the exact cause and location of the error, dump, add and remove entries from ACLs and maps, update TLS shared secrets, apply connection limits and rate limits on the fly to arbitrary frontends (useful in shared hosting environments), and disable a specific frontend to release a listening port (useful when daytime operations are forbidden and a fix is needed nonetheless).

For environments where SNMP is mandatory, at least two agents exist, one is provided with the HAProxy sources and relies on the Net-SNMP perl module. Another one is provided with the commercial packages and doesn't require Perl. Both are roughly equivalent in terms of coverage.

It is often recommended to install 4 utilities on the machine where HAProxy is deployed :

- socat (in order to connect to the CLI, though certain forks of netcat can also do it to some extents);

- halog from the latest HAProxy version : this is the log analysis tool, it parses native TCP and HTTP logs extremely fast (1 to 2 GB per second) and extracts useful information and statistics such as requests per URL, per source address, URLs sorted by response time or error rate, termination codes etc... It was designed to be deployed on the production servers to help troubleshoot live issues so it has to be there ready to be used;

- tcpdump : this is highly recommended to take the network traces needed to troubleshoot an issue that was made visible in the logs. There is a moment where application and haproxy's analysis will diverge and the network traces are the only way to say who's right and who's wrong. It's also fairly common to detect bugs in network stacks and hypervisors thanks to tcpdump;

- strace : it is tcpdump's companion. It will report what HAProxy really sees and will help sort out the issues the operating system is responsible for from the ones HAProxy is responsible for. Strace is often requested when a bug in HAProxy is suspected;

3.4.2. Advanced features : System-specific capabilities

Depending on the operating system HAProxy is deployed on, certain extra features may be available or needed. While it is supported on a number of platforms, HAProxy is primarily developped on Linux, which explains why some features are only available on this platform.

The transparent bind and connect features, the support for binding connections to a specific network interface, as well as the ability to bind multiple processes to the same IP address and ports are only available on Linux and BSD systems, though only Linux performs a kernel-side load balancing of the incoming requests between the available processes.

On Linux, there are also a number of extra features and optimizations including support for network namespaces (also known as "containers") allowing HAProxy to be a gateway between all containers, the ability to set the MSS, Netfilter marks and IP TOS field on the client side connection, support for TCP FastOpen on the listening side, TCP user timeouts to let the kernel quickly kill connections when it detects the client has disappeared before the configured timeouts, TCP splicing to let the kernel forward data between the two sides of a connections thus avoiding multiple memory copies, the ability to enable the "defer-accept" bind option to only get notified of an incoming connection once data become available in the kernel buffers, and the ability to send the request with the ACK confirming a connect (sometimes called "biggy-back") which is enabled with the "tcp-smart-connect" option. On Linux, HAProxy also takes great care of manipulating the TCP delayed ACKs to save as many packets as possible on the network.

Some systems have an unreliable clock which jumps back and forth in the past and in the future. This used to happen with some NUMA systems where multiple processors didn't see the exact same time of day, and recently it became more common in virtualized environments where the virtual clock has no relation with the real clock, resulting in huge time jumps (sometimes up to 30 seconds have been observed). This causes a lot of trouble with respect to timeout enforcement in general. Due to this flaw of these systems, HAProxy maintains its own monotonic clock which is based on the system's clock but where drift is measured and compensated for. This ensures that even with a very bad system clock, timers remain reasonably accurate and timeouts continue to work. Note that this problem affects all the software running on such systems and is not specific to HAProxy. The common effects are spurious timeouts or application freezes. Thus if this behaviour is detected on a system, it must be fixed, regardless of the fact that

HAProxy protects itself against it.

3.4.3. Advanced features : Scripting

HAProxy can be built with support for the Lua embedded language, which opens a wide area of new possibilities related to complex manipulation of requests or responses, routing decisions, statistics processing and so on. Using Lua it is even possible to establish parallel connections to other servers to exchange information. This way it becomes possible (though complex) to develop an authentication system for example. Please refer to the documentation in the file "doc/lua-api/index.rst" for more information on how to use Lua.

3.5. Sizing

Typical CPU usage 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. This means that the operating system and its tuning have a strong impact on the global performance.

Usages vary a lot between users, some focus on bandwidth, other ones on request rate, others on connection concurrency, others on SSL performance. this section aims at providing a few elements to help in this task.

It is important to keep in mind that every operation comes with a cost, so each individual operation adds its overhead on top of the other ones, which may be negligible in certain circumstances, and which may dominate in other cases.

When processing the requests from a connection, we can say that :

- forwarding data costs less than parsing request or response headers;

- parsing request or response headers cost less than establishing then closing a connection to a server;

- establishing an closing a connection costs less than a TLS resume operation;

- a TLS resume operation costs less than a full TLS handshake with a key computation;

- an idle connection costs less CPU than a connection whose buffers hold data;

- a TLS context costs even more memory than a connection with data;

So in practice, it is cheaper to process payload bytes than header bytes, thus it is easier to achieve high network bandwidth with large objects (few requests per volume unit) than with small objects (many requests per volume unit). This explains why maximum bandwidth is always measured with large objects, while request rate or connection rates are measured with small objects.

Some operations scale well on multiple process spread over multiple processors, and others don't scale as well. Network bandwidth doesn't scale very far because the CPU is rarely the bottleneck for large objects, it's mostly the network bandwidth and data busses to reach the network interfaces. The connection rate doesn't scale well over multiple processors due to a few locks in the system when dealing with the local ports table. The request rate over persistent connections scales very well as it doesn't involve much memory nor network bandwidth and doesn't require to access locked structures. TLS key computation scales very well as it's totally CPU-bound. TLS resume scales moderately well, but reaches its limits around 4 processes where the overhead of accessing the shared table offsets the small gains expected from more power.

The performance numbers one can expect from a very well tuned system are in the following range. It is important to take them as orders of magnitude and to expect significant variations in any direction based on the processor, IRQ setting, memory type, network interface type, operating system tuning and so on.

The following numbers were found on a Core i7 running at 3.7 GHz equiped with a dual-port 10 Gbps NICs running Linux kernel 3.10, HAProxy 1.6 and OpenSSL 1.0.2. HAProxy was running as a single process on a single dedicated CPU core, and two extra cores were dedicated to network interrupts :

- 20 Gbps of maximum network bandwidth in clear text for objects 256 kB or higher, 10 Gbps for 41kB or higher;

- 4.6 Gbps of TLS traffic using AES256-GCM cipher with large objects;

- 83000 TCP connections per second from client to server;

- 82000 HTTP connections per second from client to server;

- 97000 HTTP requests per second in server-close mode (keep-alive with the client, close with the server);

- 243000 HTTP requests per second in end-to-end keep-alive mode;

- 300000 filtered TCP connections per second (anti-DDoS)

- 160000 HTTPS requests per second in keep-alive mode over persistent TLS connections;

- 13100 HTTPS requests per second using TLS resumed connections;

- 1300 HTTPS connections per second using TLS connections renegociated with RSA2048;

- 20000 concurrent saturated connections per GB of RAM, including the memory required for system buffers; it is possible to do better with careful tuning but this setting it easy to achieve.

- about 8000 concurrent TLS connections (client-side only) per GB of RAM, including the memory required for system buffers;

- about 5000 concurrent end-to-end TLS connections (both sides) per GB of RAM including the memory required for system buffers;

Thus a good rule of thumb to keep in mind is that the request rate is divided by 10 between TLS keep-alive and TLS resume, and between TLS resume and TLS renegociation, while it's only divided by 3 between HTTP keep-alive and HTTP close. Another good rule of thumb is to remember that a high frequency core with AES instructions can do around 5 Gbps of AES-GCM per core.

Having more core rarely helps (except for TLS) and is even counter-productive due to the lower frequency. In general a small number of high frequency cores is better.

Another good rule of thumb is to consider that on the same server, HAProxy will be able to saturate :

- about 5-10 static file servers or caching proxies;

- about 100 anti-virus proxies;

- and about 100-1000 application servers depending on the technology in use.

3.6. How to get HAProxy

-----------------------

HAProxy is an opensource project covered by the GPLv2 license, meaning that everyone is allowed to redistribute it provided that access to the sources is also provided upon request, especially if any modifications were made.

HAProxy evolves as a main development branch called "master" or "mainline", from which new branches are derived once the code is considered stable. A lot of web sites run some development branches in production on a voluntarily basis, either to participate to the project or because they need a bleeding edge feature, and their feedback is highly valuable to fix bugs and judge the overall quality and stability of the version being developped.

The new branches that are created when the code is stable enough constitute a stable version and are generally maintained for several years, so that there is no emergency to migrate to a newer branch even when you're not on the latest. Once a stable branch is issued, it may only receive bug fixes, and very rarely minor feature updates when that makes users' life easier. All fixes that go into a stable branch necessarily come from the master branch. This guarantees that no fix will be lost after an upgrade. For this reason, if you fix a bug, please make the patch against the master branch, not the stable branch. You may even discover it was already fixed. This process also ensures that regressions in a stable branch are extremely rare, so there is never any excuse for not upgrading to the latest version in your current branch.

Branches are numberred with two digits delimited with a dot, such as "1.6". A complete version includes one or two sub-version numbers indicating the level of fix. For example, version 1.5.14 is the 14th fix release in branch 1.5 after version 1.5.0 was issued. It contains 126 fixes for individual bugs, 24 updates on the documentation, and 75 other backported patches, most of which were needed to fix the aforementionned 126 bugs. An existing feature may never be modified nor removed in a stable branch, in order to guarantee that upgrades within the same branch will always be harmless.

HAProxy is available from multiple sources, at different release rhythms :

- The official community web site : http://www.haproxy.org/ : this site provides the sources of the latest development release, all stable releases, as well as nightly snapshots for each branch. The release cycle is not fast, several months between stable releases, or between development snapshots. Very old versions are still supported there. Everything is provided as sources only, so whatever comes from there needs to be rebuilt and/or repackaged;

- A number of operating systems such as Linux distributions and BSD ports. These systems generally provide long-term maintained versions which do not always contain all the fixes from the official ones, but which at least contain the critical fixes. It often is a good option for most users who do not seek advanced configurations and just want to keep updates easy;

- Commercial versions from http://www.haproxy.com/ : these are supported professional packages built for various operating systems or provided as appliances, based on the latest stable versions and including a number of features backported from the next release for which there is a strong demand. It is the best option for users seeking the latest features with the reliability of a stable branch, the fastest response time to fix bugs, or simply support contracts on top of an opensource product;

In order to ensure that the version you're using is the latest one in your branch, you need to proceed this way :

- verify which HAProxy executable you're running : some systems ship it by default and administrators install their versions somewhere else on the system, so it is important to verify in the startup scripts which one is used;

- determine which source your HAProxy version comes from. For this, it's generally sufficient to type "haproxy -v". A development version will appear like this, with the "dev" word after the branch number :

  ```
  HA-Proxy version 1.6-dev3-385ecc-68 2015/08/18
  ```

  A stable version will appear like this, as well as unmodified stable versions provided by operating system vendors :

  ```
  HA-Proxy version 1.5.14 2015/07/02
  ```

  And a nightly snapshot of a stable version will appear like this with an hexadecimal sequence after the version, and with the date of the snapshot instead of the date of the release :

  ```
  HA-Proxy version 1.5.14-e4766ba 2015/07/29
  ```

  Any other format may indicate a system-specific package with its own patch set. For example HAProxy Enterprise versions will appear with the following format (<branch>-<latest commit>-<revision>) :

  ```
  HA-Proxy version 1.5.0-994126-357 2015/07/02
  ```

- for system-specific packages, you have to check with your vendor's package repository or update system to ensure that your system is still supported, and that fixes are still provided for your branch. For community versions coming from haproxy.org, just visit the site, verify the status of your branch and compare the latest version with yours to see if you're on the latest one. If not you can upgrade. If your branch is not maintained anymore, you're definitely very late and will have to consider an upgrade to a more recent branch (carefully read the README when doing so).

HAProxy will have to be updated according to the source it came from. Usually it follows the system vendor's way of upgrading a package. If it was taken from sources, please read the README file in the sources directory after extracting the sources and follow the instructions for your operating system.

4. Companion products and alternatives

HAProxy integrates fairly well with certain products listed below, which is why they are mentionned here even if not directly related to HAProxy.

4.1. Apache HTTP server

Apache is the de-facto standard HTTP server. It's a very complete and modular project supporting both file serving and dynamic contents. It can serve as a frontend for some application servers. In can even proxy requests and cache responses. In all of these use cases, a front load balancer is commonly needed. Apache can work in various modes, certain being heavier than other ones. Certain modules still require the heavier pre-forked model and will prevent Apache from scaling well with a high number of connections. In this case HAProxy can provide a tremendous help by enforcing the per-server connection limits to a safe value and will significantly speed up the server and preserve its resources that will be better used by the application.

Apache can extract the client's address from the X-Forwarded-For header by using the "mod_rpaf" extension. HAProxy will automatically feed this header when "option forwardfor" is specified in its configuration. HAProxy may also offer a nice protection to Apache when exposed to the internet, where it will better

resist to a wide number of types of DoS.

4.2. NGINX

NGINX is the second de-facto standard HTTP server. Just like Apache, it covers a wide range of features. NGINX is built on a similar model as HAProxy so it has no problem dealing with tens of thousands of concurrent connections. When used as a gateway to some applications (eg: using the included PHP FPM), it can often be beneficial to set up some frontend connection limiting to reduce the load on the PHP application. HAProxy will clearly be useful there both as a regular load balancer and as the traffic regulator to speed up PHP by decongestionning it. Also since both products use very little CPU thanks to their event-driven architecture, it's often easy to install both of them on the same system. NGINX implements HAProxy's PROXY protocol, thus it is easy for HAProxy to pass the client's connection information to NGINX so that the application gets all the relevant information. Some benchmarks have also shown that for large static file serving, implementing consistent hash on HAProxy in front of NGINX can be beneficial by optimizing the OS' cache hit ratio, which is basically multiplied by the number of server nodes.

4.3. Varnish

Varnish is a smart caching reverse-proxy, probably best described as a web application accelerator. Varnish doesn't implement SSL/TLS and wants to dedicate all of its CPU cycles to what it does best. Varnish also implements HAProxy's PROXY protocol so that HAProxy can very easily be deployed in front of Varnish as an SSL offloader as well as a load balancer and pass it all relevant client information. Also, Varnish naturally supports decompression from the cache when a server has provided a compressed object, but doesn't compress however. HAProxy can then be used to compress outgoing data when backend servers do not implement compression, though it's rarely a good idea to compress on the load balancer unless the traffic is low.

When building large caching farms across multiple nodes, HAProxy can make use of consistent URL hashing to intelligently distribute the load to the caching nodes and avoid cache duplication, resulting in a total cache size which is the sum of all caching nodes.

4.4. Alternatives

Linux Virtual Server (LVS or IPVS) is the layer 4 load balancer included within the Linux kernel. It works at the packet level and handles TCP and UDP. In most cases it's more a complement than an alternative since it doesn't have layer 7 knowledge at all.

Pound is another well-known load balancer. It's much simpler and has much less features than HAProxy but for many very basic setups both can be used. Its author has always focused on code auditability first and wants to maintain the set of features low. Its thread-based architecture scales less well with high connection counts, but it's a good product.

Pen is a quite light load balancer. It supports SSL, maintains persistence using a fixed-size table of its clients' IP addresses. It supports a packet-oriented mode allowing it to support direct server return and UDP to some extents. It is meant for small loads (the persistence table only has 2048 entries).

NGINX can do some load balancing to some extents, though it's clearly not its primary function. Production traffic is used to detect server failures, the load balancing algorithms are more limited, and the stickiness is very limited.

But it can make sense in some simple deployment scenarios where it is already present. The good thing is that since it integrates very well with HAProxy, there's nothing wrong with adding HAProxy later when its limits have been faced.

Varnish also does some load balancing of its backend servers and does support real health checks. It doesn't implement stickiness however, so just like with NGINX, as long as stickiness is not needed that can be enough to start with. And similarly, since HAProxy and Varnish integrate so well together, it's easy to add it later into the mix to complement the feature set.