‘

HA Proxy 1.4:

From the Ground Up

Partial Spec Paper
written by:

David Raistrick
with Mary K Swanson

Contents

1 About HAProxy

HAProxy, written in C by Willy Tarreau, is open source software for TCP and HTTP load balancing. It's known for stability, efficiency and speed; chances are that this reputation is why you are installing it.

Load balancing, though, is a term that covers a broad range of functions, from basic TCP tunneling at Layer 4 to reverse proxy at Layer 7, and you may find more functionality in HAProxy than you expected—or, more than you bargained for.

In this chapter, you'll learn what HAProxy load balancing is and what it can do for your network, and learn the terms you'll need throughout the instructions. After that, you'll be ready to begin configuring HAProxy to support your network needs.

The current version of HAProxy is 1.4.

HAProxy Basics

Load balancers are not all the same. Some simply work at Layer 4, directing network traffic with no regard to payload. Others like HAProxy also operate at Layer 7, where requests can be routed based on content.

With HAProxy, you can have high availability, reverse proxy, and failover, use ACLs, generate status reports and monitor your system, but all of it depends on the load balancing function.

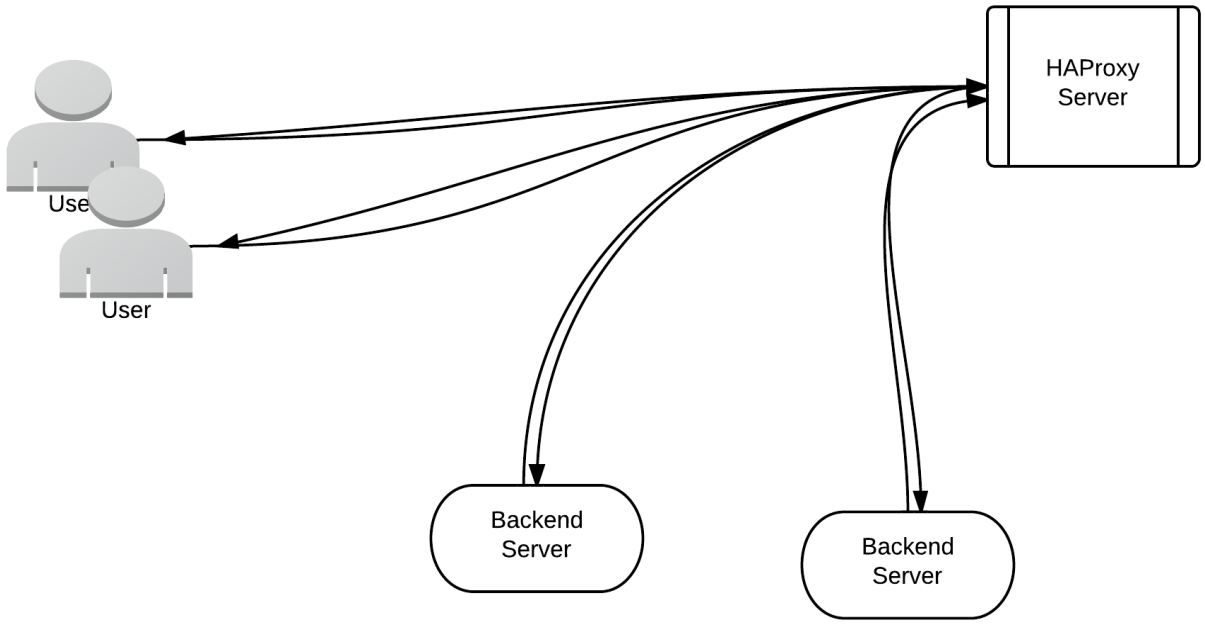
Load Balancing

A load balancer distributes network or application traffic across multiple servers, allowing more users to access the same resource simultaneously. HAProxy operates at Layer 4 (network and transport protocols, or tunneling) and Layer 7 (application layer protocols).

One Arm or Two?

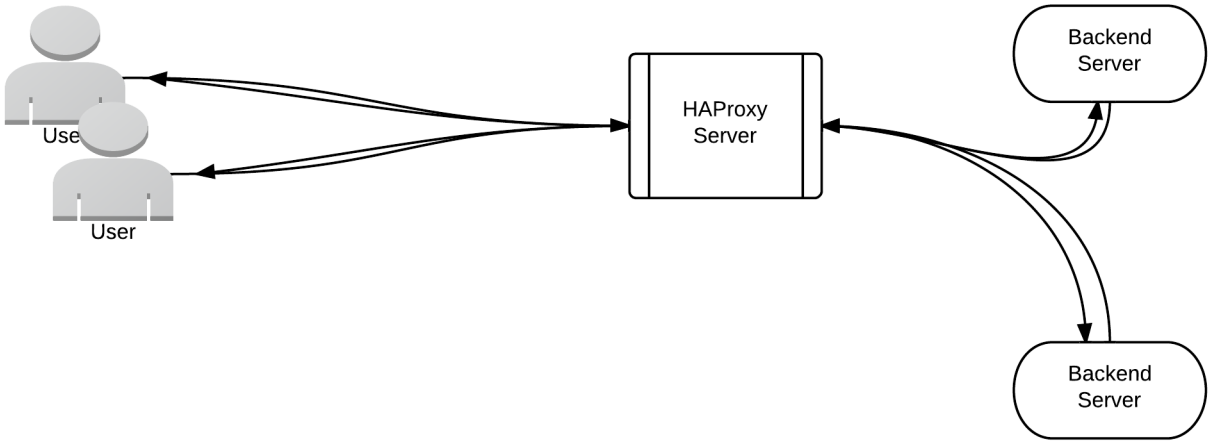
There are two common load balancing setups. In the simplest, the traffic is passed through one interface only— a request goes in one door to the load balancer, out the same door to the server resources, then the response travels back through the same interface into the load balancer and out

to the requestor. This is referred to as one arm. This is usually the easiest setup to use, but limits maximum available throughput.



One Arm Load Balancer

Most software load balancers use one arm, but there can be substantial performance benefits from using the other type of load balancer setup called two arm. In this type, two interfaces are used for traffic, with the request going through the first interface into the load balancer, out the second interface to the server resources, and the response moving back through each of the two interfaces on its way back to the requestor. In order to use two (or more) arm infrastructures, the network routing topology needs to be changed, and the server running the load balancer software needs to be reconfigured as a router.



Two Arm Load Balancer

Select Terms

These terms will be useful as you read this document and begin setting up your own HAProxy load balancer.

ACL

ACLs are access control lists. They provide the ability to allow or deny access based on request content. HAProxy allows you to create and administer Layer 7 ACLs for your application, content switching routing/directing traffic based on ACLs.

DoS and DDoS

DoS and DDoS stand for denial of service and distributed denial of service. A DoS is an assault on a network that floods it with so many additional requests that regular traffic is either slowed or completely interrupted. A DDoS uses multiple computers throughout the network. HAProxy defends against these attacks by decoupling the client connection and request from the servers, queuing requests in front of busy servers, allowing you to fine tune details like timeouts and connection limits, and adding/testing against custom cookies. HAProxy 1.5 will include even more features.

Failover

Failover is the process by which a system, detecting the failure of one server, automatically reroutes any traffic to the other remaining servers in the group. If the application is stateful, the system remembers the state of each connection, creating a seamless experience for users of the system. HAProxy's load balancing function allows you to set up automatic failover, ensuring that your network is always available.

High Availability

High availability is the term for a multiprocessing system that can quickly recover from a failure. There may a minute or two of downtime while one system switches over to another, but processing will continue. Another term for this is fault resilient.

One Arm and Two Arm

The terms 'one arm' and 'two arm' refer to the two common load balancing setups. One arm uses only one interface on the load balancer for passing traffic, where two arm uses two interfaces, one on the way in to the load balancer and one on the way out. HAProxy can be used with either setup.

HTTP Keep-alives

HTTP keep-alives are persistent connections between the client and the server, allowing you to perform multiple requests on the same connection. HAProxy 1.4 supports keep-alives between the client and HAProxy, but not between HAProxy and the backend server.

Reverse Proxy

Reverse proxy for HTTP is another way of saying a proxy for the server side of a system rather than for the client side. Some load balancers offer reverse proxies; some reverse proxy setups offer load balancing. The terms reverse proxy and load balancing are sometimes used interchangeably because of how often they are found together. You can use HAProxy to set up a reverse proxy.

Stickiness

Stickiness refers to keeping all of a particular client's traffic 'stuck' to a specific server. It is particularly useful for applications that are unable to share state information between multiple servers. Another term for stickiness is session persistence.

Tunnel

A tunnel for HAProxy simply passes all data for a connection transparently through the load balancer. For HTTP, this means that only the initial request will be inspected; all further requests on a connection will be passed through untouched. For TCP, only the source/destination addresses for the connection change.

Supported Platforms for HAProxy 1.4

HAProxy is known to function on the operating systems listed below. Other related platforms may be used, but are not supported by the developers.

Linux 2.4 on x86, x86_64, Alpha, SPARC, MIPS, PARISC; Linux 2.6 on x86, x86_64, ARM (ixp425), PPC64

Solaris 8/9 on UltraSPARC 2 and 3; Solaris 10 on Opteron and UltraSPARC

FreeBSD 4.10 to 8.4 on x86

OpenBSD 3.1 to 5.3 on i386, amd64, macppc, alpha, sparc64 and VAX (check the ports)

Summary

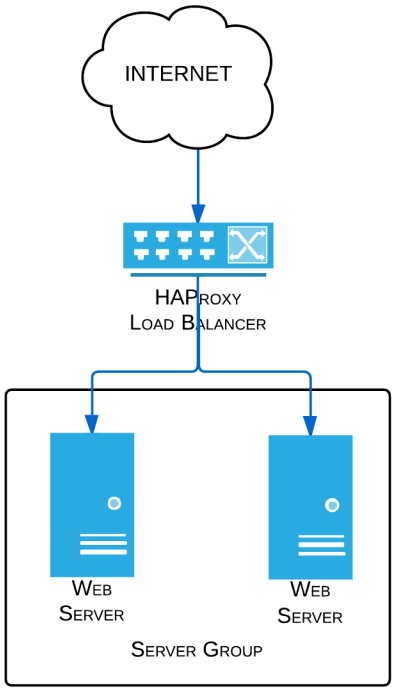
HAProxy is a load balancer, but load balancing doesn't cover the range of functions it can perform. It also offers reverse proxy, high availability, failover, tunneling, stickiness, access control using ACLs, protection against denial of service attacks, and many other functions.

In the next chapter, you'll learn what you need to consider when installing HAProxy, basic configuration steps, and how to set up global parameters.

2 Installation

This chapter covers installation considerations, serves as a step-by-step guide to create a basic configuration using global, default, listen and server parameters, and presents explanations of how to use selected other parameters. As the basic configuration is built, you'll learn how each selection and setting affects the load balancer.

For this basic configuration, you will build an HAProxy configuration for a load balancer in front of two identical web servers to distribute application traffic between the two.



Load Balancer with Two Identical Servers

Considerations

The steps to install HAProxy depend on the flavor of Linux or other UNIX-like operating system you use. Recent versions are available as packages in the software repositories for most common operating systems.

Here are a few examples of installation commands:

- Red Hat and other yum-based distributions:

```
yum install haproxy
```

Red Hat requires the EPEL repository.

- Debian, Ubuntu, and other distributions that use apt:

```
apt-get install haproxy
```

Be sure to update /etc/default/haproxy to set ENABLED=1 for debian.

- FreeBSD:

```
cd /usr/ports/net/haproxy
make install clean
```

Build from Source

If you need a newer version than your package provider offers, or there is no package available for your system, you can build from source.

Download and extract the source tarball from http://haproxy.1wt.eu/#down. Please see the README inside the tarball for more specific requirements and build instructions.

For Linux and Solaris SPARC, precompiled binaries are also available. Extract them and run them as root.

Command Line Arguments

At startup, you can make command line arguments to control parts of HAProxy's behavior in your environment.

HAProxy has only a few command line arguments you will need to use in a basic configuration, and most distribution packages already handle these for you in their startup scripts.

These arguments are used:

Argument	Mode	Description	Notes
-c		Validates the config file and exits.	
-D	daemon	Runs process in background.	The recommended mode.
-db		Disables daemon mode so that the process runs in the foreground.	
-d	debug	Enables debugging, writing all exchanges to stdout (standard output), and the process runs in the foreground.	Debug may prevent full system startup.
-n		Sets the maximum number of concurrent connections per process.	Like maxconn in the global section.
-N		Sets the maximum number of concurrent connections per proxy.	Like maxconn at the default level, which propagates down to any proxy (listen or frontend) if not specifically set.
-q	quiet	Does not display messages during startup.	
-V	verbose	Displays messages during startup even when -q is specified in the configuration.	

Command Line Arguments

If no command line argument or configuration parameter is set, the defaults are:

The process runs in the foreground (no daemon).

Messages are displayed during startup (verbose).

No debugging is written.

That Was Fast!

You now have the latest version of HAProxy installed. The next step is to set configuration parameters.

Basic Configuration

In this section, you will learn how to write a basic configuration for a load balancer and two web servers. The sections you will use in this basic configuration are:

- Global
- Default

- Listen

Parameters in other sections—global, default, listen, and frontend/backend—are sometimes referred in the HAProxy documentation to as global parameters.

Basic configuration can be as simple as three lines, or as complex as you want it to be. In this section, you'll learn about parameters in regular use throughout the industry. In succeeding chapters, you'll learn about other parameters used for special purposes, such as access control lists, logging, and, statistics.

Most parameters are specific to configuring the HAProxy application; many of these will be set during installation by the package installer or can be set at startup using a command line argument.

Some parameters, such as maxconn, also interact with the proxies defined by the application, setting maximums or minimums.

The basic configuration in this chapter uses application-specific parameters, except for maximum connections.

Global Section Parameters

The global section is not required to configure HAProxy; if you do not use the global section parameters, you do not need the section.

While the global section parameters are not required, if you start HAProxy as root:

It is strongly recommended that you set the user or uid, group or gid, and chroot.

It is also recommended that you set maximum connections (maxconn) so that HAProxy can automatically configure the open file limits (ulimit-n).

To create the basic configuration in the global section, you will set parameters that achieve these five goals:

- Set up a proxy that accepts no more than 500 connections.
- Change permissions so that the proxy runs as the user named haproxy.
- Change permissions so that the proxy runs as a group named haproxy.
- Restrict the proxy so that it sees the /home/haproxy subdirectory as root.
- Tell the proxy to run as a daemon (in the background).

Below is the global section of the configuration that defines the proxy as desired.

```
global
    maxconn 500
    user haproxy
    group haproxy
    chroot  /home/haproxy
    daemon
```

Global Section Example

To explain why these parameters are set, we'll look at each parameter more closely.

Maxconn

The maxconn parameter sets the maximum number of concurrent connections per process. The proxy will stop accepting connections when this limit it reached.

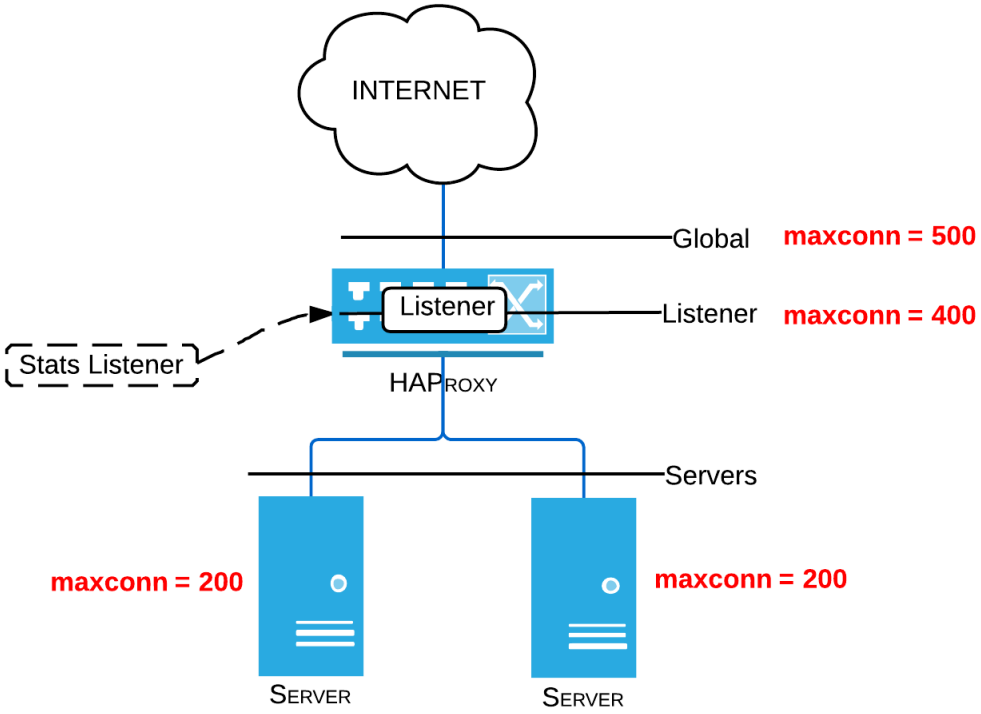
```
maxconn <number>
```

It is equivalent to the command-line argument `-n`.

Ulimit-n, the maximum number of file descriptors per process, is automatically computed according to the maxconn and other configuration factors, so using the separate ulimit-n <number> parameter is not recommended.

In the basic configuration, the maxconns are set to 500 because our example servers can only support a limited number of connections for their application. The global connection limit is set slightly higher than the maximum connections that the servers can handle to account for connections to other listeners, such as a stats listener.

The diagram below shows the simple listener/maxconn setup for the basic configuration.



**Simple Setup for Global,
Proxy (Listener), and Server Maxconn Parameters**

For a more advanced example, see the diagram in the Listen section.

User and Group

Before configuration, the user ID of the process is 0, and the name is root. The user parameter switches the user name to the name defined in the password file (/etc/passwd). This is referred to as a de-escalation of privileges. The uid parameter is the same, but identifies the user by user ID.

```
uid <number>
```

```
user <user name>
```

Group and gid function the same as user and uid, and are defined in the /etc/group file.

```
gid <number>
```

```
group <group name>
```

Chroot

Setting the chroot parameter changes the root directory for a specific application to a subdirectory, presenting the subdirectory as the root of the filesystem. This is called a chroot jail. Chroot effectively protects the larger system from malicious or accidental changes by displaying a restricted view of the filesystem with fewer system privileges.

chroot <directory>

The user name, group name, and chroot directory should be dedicated to HAProxy. Ensure that the chroot directory is empty and cannot be written to so that no content is accessible by the software.

The user name, group name, and chroot are set this way to limit software's access to the system by confining it to its own directory and user.

HAProxy must be started with root privileges in order to switch to another user/group name or chroot directory.

Daemon

Setting the daemon parameter forks the process, detaching it from the shell that started it, and making the process run in the background. The process can also be set to run in the background by using the command line -D argument. The daemon parameter set here can be disabled by using the command line -db argument.

```
daemon
```

Detaching the process from the shell and running it as a daemon allows the administrator to close the shell.

When the process runs in the foreground, you can find standard output and errors from the process, control and monitor it with an external tool, or debug it.

Additional Global Section Parameters

Other selected parameters are listed alphabetically in the following table. The command line equivalent is indicated. If a topic will be discussed later, the chapter number is provided.

Parameter	Description	Cmd	Ch
debug	Enables debug mode which lets the process happen in the foreground (disables daemon mode) and logs events to the stdout, a log file. Using debug in production could prevent system startup.	-d	3
log <address> <facility> [maximum level [minimum]]	Defines one of two possible global syslog servers. The global syslog is used to record startup, shutdown, and logs from any proxy set to log global messages. The <address> option can be: • IP address and port. The port is optional; 514 is the default. • A valid filesystem path to a UNIX domain socket. The <facility> option specifies one of the 24 syslog facilities, e.g., kern, cron, or local0. By default, all messages are sent, but you can set a maximum and minimum message level; the order is from most to least severe: emerg, alert, crit, err, warning, notice, info, and debug. If a minimum is set, log messages more severe than the minimum will be sent at this level.		6

nbproc <number>	Defines the number of processes created when in daemon mode. The receommended and default number is one.		
pidfile <name>	Writes process identifiers (pids) of all daemons into named pidfile. The pidfile must be accessible by the user starting and stopping processes.	-p	8
quiet	Log no messages.	-q	3
spread-checks <0-50>	Introduces randomness into the health check report interval. This helps avoid logging traffic jams, for instance, when you have many virtual servers located on the same physical box. The range is between -50 and +50 percent, and the default is 0. As an example, if you set: spread-checks <9>, your healthcheck interval would range from -9 percent to +9 percent of the interval you have specified .		7
stats maxconn <number>	Changes the value of the stats socket concurrent connections, which is 10 bu default.		7

stats socket <path> [{uid | user} <uid>] [{gid | group} <gid>] [mode <mode>]	Creates or backs up and replaces an existing UNIX socket. Requests to this socket can send these commands: • show stat to get a .csv file of the process stats • show info for more general information • show sess for a list of existing sessions		7
stats timeout <number >	Changes the timeout for stats socket, which is 10 seconds by default. In milliseconds, by default.		7

Global Section Parameters

Time in HAProxy is expressed in milliseconds (ms), unless otherwise indicated. If you wish to use another supported unit, add its one or two letter abbreviation as a suffix to the number. The abbreviations for other supported time units are `us, s, m. h,` and `d`.

Default Section Parameters

The default section parameters apply to the listeners and servers that follow in the configuration, unless a more specific parameter is set.

The default section is not required to configure HAProxy; if you do not use the default section parameters, you do not need the section.

The default values for some timeouts are set to infinity, so while the parameters are not required, it is recommended that you set default timeouts for your configuration so that you only have to set them in one place, and as a best practice, so you don't forget to set them on a listener.

To create the basic configuration in the default section, you will set parameters that achieve these goals:

- Set timeouts to values that reflect best practices.
- Disable logging of incomplete connections (which offer no relevant information and fills the logs).

Below is the default section of the configuration.

```
defaults
timeout connect 5s
timeout client 50s
timeout server 50s
option dontlognull
```

Defaults Section Example

Timeout Connect

Set the timeout connect parameter to five (5) seconds. This is considered a best practice, and a reasonable length of time to get a TCP connection established (handshake).

```
timeout connect <timeout>
```

Timeout Client and Server

The parameters timeout client and timeout server define how long the proxy waits for a client or server to send a request or send a response.

```
timeout client <timeout>
timeout server <timeout>
```

An infinite default timeout could use excessive resources, resulting in a denial of service.

Dontlognull

The dontlognull parameter disables the logging of incomplete connections.

option dontlognull

no option dontlognull

Listen Section Parameters

The listen section parameters define the proxy itself, including listening on a port, the servers behind the proxy and the parameters not set in defaults.

Either a listen section or a frontend/backend combination is required to configure HAProxy; if you use a listen section, you don't need a frontend/backend, and vice versa.

In this configuration, we are using a listen section because it's more commonly used. Frontend/backend was new in version 1.3, and fewer examples exist.

Listener vs. frontend/backend: a frontend describes a set of listeners accepting client connections, and a backend defines a server group, whereas listen defines a complete proxy with its frontend and backend combined in one section.

Use frontend/backend instead of listen to reuse backends (the defined set of servers) across multiple frontends (the listening proxies).

To create the basic configuration in the listen section, you will set parameters that achieve these goals:

- Listen on a port.
- Load balance traffic to the servers.

Below is the listen section of the basic configuration.

```
listen port80proxy
   bind 0.0.0.0:80
   mode http
   maxconn 400
   balance roundrobin

   option httpchk
   option forwardfor
   option httplog
   option redispatch

   retries 3  #default...

   server test2 10.1.1.2:80 check maxconn 200
   server test3 10.1.1.3:80 check maxconn 200
```

Listen Section Example

Listen

The listen parameter in the listen section establishes the proxy name.

Proxy names may only be composed of letters, digits, dashes '-' (a short dash or hyphen), underscores '_', dots '.', and colons ':'.

Bind

The bind parameter identifies an IP address and port to listen on.

```
bind [<address>]:<port_range> [interface,mss,
transparent, id, name, defer-accept]
```

In this configuration, you use '0.0.0.0' as the IP address in order to listen on all available addresses so that you don't need to know how the server is configured. You select port 80 because you are load balancing for web site content.

In order to listen on low ports (1024 and below), you must start HAProxy by logging in as root.

Some options for bind include:

- <address> Designates the address to listen on. The address can be a host name, IPv4 or IPv6 address. If not set, a wildcard (*), or the '0.0.0.0' address is used, all system IPv4 addresses will be listened on.
- <port range> Required. Either a unique TCP port or a port range for which the proxy will accept connections for the IP address specified. A port can be a numerical port, for example, '80' or a dash-delimited range, for example, '2000-2100'.

The numbers stated are included in the range.

Every <address:port> uses one socket, so ranges can quickly consume resources, and only one instance of a combination can be used.

To specify multiple address and port combinations, use commas to separate them. For example:

```
listen http_proxy
       bind :80,:443
       bind 10.0.0.1:11080,10.0.0.1:11443
```

Mode

The mode parameter tells the proxy whether you are proxying for TCP, layer 4, or HTTP, layer 7. At layer 4, HAProxy forwards traffic in two directions. At layer 7, HAProxy analyzes the protocol in order to allow or block, add or delete, or otherwise change content.

```
mode { tcp|http|health }
```

In this basic configuration, you set the mode to http because you are load balancing for web site content.

Maxconn

The proxy level maxconn parameter sets the maximum number of concurrent connections to this listener (proxy).

```
maxconn <conns>
```

You set 400 as the maxconn for the listener, which is less than the global maxconn of 500, in order to leave room for connections to a stats listener for statistics (discussed in the Statistics and Monitoring chapter).

Balance

The balance parameter defines the load balancing algorithm to be used.

```
balance <algorithm> [ <arguments> ]
```

HAProxy currently supports eight load balancing algorithms, or which four are more commonly used:

- roundrobin Requests are distributed evenly across each backend server (or, if weights are defined, unevenly according to weight).
- leastconn The server with the least connections at the moment of the request receives the next request. This may be particularly useful for certain types of TCP traffic that use long-lived connections.
- source The source IP address is used to determine which server receives a request with the goal of keeping all traffic for a specific source address stuck to a specific server. This is typically used for TCP; for HTTP traffic, stickiness is usually selected using a session cookie or header, instead of the source IP, since many clients may be behind the same firewall or proxy and appear to be from from the same address.
- hdr <name> The server is selected based on a header as the identifier. If the header isn't set, the request is distributed by roundrobin instead.

Httpchk

By default, server health checks try to establish a TCP connection. For an HTTP server, it's a good idea to make an HTTP request to verify the server can still respond. Setting the option httpchk tells HAProxy to use an HTTP test, and a complete HTTP request is sent once the TCP connection is established.

```
option httpchk

option httpchk <uri>

option httpchk <method> <uri>

option httpchk <method> <uri> <version>
```

By default, httpchk will cause an options / request to be sent. You can change the <method> (head, get) or specify a URL as well:

- <url> URL referred to in the HTTP requests.

- <version> HTTP version string. Default is HTTP/1.0.

In general, using the options or head method is recommended for health checks, since these types of requests don't cause the server to return the body in the response. If you wish to inspect the response for content (using http-check expect), you may need to change this method to a type that returns what you are checking for. Without http-check expect, httpchk expects a 2xx or 3xx response code from the request. Anything else is considered a failed request.

With http-check expect [!] <match> <pattern>, you can change the behavior of the tests performed by the check and httpchk combination. You can match specific status or response codes, or look for specific strings in the response body.

- <match> Keyword such as status, rstatus, string, or rstring.
- <pattern> String or a regex (regular expression).

You can use and exclamation mark (!) to search for items that do not match the pattern.

Forwardfor

Since HAProxy works as a reverse proxy, the servers see its IP address as their client address. The option forwardfor adds or updates the X-forwarded-for HTTP header with the source IP address of the connection so the server software can recognize where the connection came from.

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

- <network> Disable this option for matching sources.
- <name> Specify a different 'X-Forwarded-For' header name.

To disable the addition of the header for a known source address, add the except keyword and the address.

HAProxy works in tunnel mode and so will only add the header to the first request. To avoid that, set httpclose, forceclose, or http-server-close options.

Httplog

Standard log output contains the source and destination addresses, and the instance name. The option httplog adds the HTTP request, session state, timers, headers and cookies, and server names to the log in the HTTP format.

```
option httplog [ clf ]
```

If clf is specified, the output format will be CLF instead of HTTP.

Redispatch and Retries

When the option redispatch is set, if HAProxy attempts to send a request to a server and fails, it will send it to a new server after a number of retries.

```
option redispatch
```

```
no option redispatch
```

The retries parameter sets how many times HAProxy will try to send a request to a server.

This is a distinctive feature of HAProxy.

```
retries <value>
```

In HTTP mode, if a server designated by a cookie is down, clients will be unable to access the service, due to persistence. This option allows the proxy to break persistence and redistribute requests to a working server.

Server

The server parameters establish a name for each server, the IP address or hostname of the server, the port on which the server is listening, make HAProxy test the server for functionality, and set the maximum concurrent connections for each server.

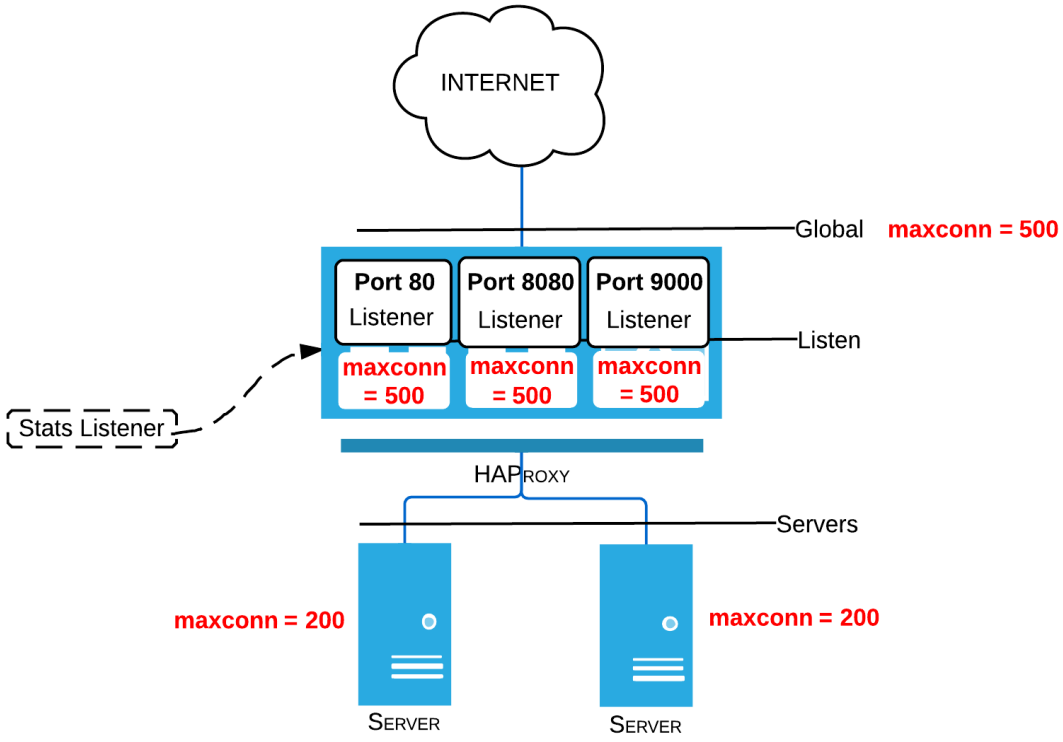
```
server <name> <address>[:port] [param*]
```

You set maxconn to 200 on each server in this basic configuration in order to limit the traffic on each server. These are set the same because our example servers are identical. You can see a diagram of your configuration's maxconn setup above in the global section.

For comparison, the diagram below shows a more advanced listener/maxconn setup, including three listeners and tuned maximum connections at the listener level that use the listeners' ability to queue connections when the maximum number of connections has been reached at the server level.

An example of another way to set up listeners, more elaborate that the basic configuration, you could create multiple listeners with maxconns set to the same as the global maximum connections. In this setup, the each listener is presumed to require the maximum number connections, but not concurrently. The statistics listener is not guaranteed to have connections if the maximum connections are reached.

HAProxy's queuing feature will help avoid dropped connections.



**Advanced Setup for Global,
Proxy (Listener), and Server Maxconn Parameters**

Server parameters include the following. Look for the default-server parameter in the section below.

- redir <prefix> Enables redirect mode for all get and head requests addressing the server, sending an 'HTTP 302.'

No trailing slash should be used after <prefix>.

All invalid requests will be rejected, and all requests not designated get or head will be behave normally.

Using a relative location will cause a loop between the client and HAProxy.

Additional Listen Section Parameters

Other selected parameters are listed alphabetically in the following table. If a topic will be discussed later, the chapter number is provided.

Parameter	Description	Ch
default-server [parameters]	Allows you to create a set of defaults that apply to all server entries. For example, check and maxconn could be set here to avoid repeating them for all servers. Not all server parameters can be set here. Others that can include:	

Parameter	Description	Ch
	error-limit <count>	
	Specifies the number of consecutive errors that trigger the event selected by on-error. Default is 10 errors.	
	fall <count>	
	Specifies that a server is ‘dead’ after this number of consecutive failed health checks. Default 3 if unspecified.	
	rise <count>	
	slowstart <start_time_in_ms>	
	port <port>	
errorfile <code> <file>	Returns file contents instead of HAProxy-generated errors.	
	<code> HTTP status code	
	<file> Full HTTP response	
errorloc <code> <url>	Returns an HTTP redirection to a URL instead of HAProxy-generated errors.	
	<code> HTTP status code	
	<url> Contents of the ‘location’ header	
option allbackups	By default, the first operational backup server gets all traffic when normal servers are all down. If you prefer to use multiple backups, enable this option to perform load balancing among all backup servers.	
	If this option is enabled in defaults, disable it in a specific instance by adding the keyword no before the option.	
reqadd <string> [{if | unless} <condition>]	Adds a header at the end of the HTTP request.	3
	<string> Complete line to be added.	
	<cond> Ignore reqadd when condition matches an ACLs	
	Header transformations apply to traffic passing through HAProxy, not to HAProxy-generated traffic like heatlh checks and error responses.	
backup	Allows the indicated server to be used in load balancing when all other non-backup servers are unavailable.	
rate-limit sessions <rate>	Limits the number of new sessions accepted per second.	7
	<rate> Maximum number of new sessions.	
	Once the limit is reached, pending sessions are kept in the socket's backlog. When applying a low limit, consider increasing the socket's backlog.	

Configuration examples

The examples below show the basic configuration and the multiple listener/advanced maxconn configuration presented in the listen section.

Basic Configuration Example

The example below shows the complete basic configuration presented in the chapter.

```
global
   maxconn 500
   user haproxy
   group haproxy
   chroot  /home/haproxy
   daemon

defaults
   timeout connect 5s
   timeout client 50s
   timeout server 50s
   option dontlognull

listen port80proxy
bind 0.0.0.0:80
   mode http
   maxconn 400
   balance roundrobin

   option httpchk
   option forwardfor
   option httplog
   option redispatch

   retries 3

   server test2 10.1.1.2:80 check maxconn 200
```

```
    server test3 10.1.1.3:80 check maxconn 200
```

Multiple Listener Configuration

The example below shows the multiple listener/advanced maxconn configuration presented in the listen section.

```
global
    maxconn 500
    user haproxy
    group haproxy
    chroot  /home/haproxy
    daemon

defaults
    timeout connect 5s
    timeout client 50s
    timeout server 50s
    option dontlognull

listen port80proxy
bind 0.0.0.0:80
    mode http
    maxconn 500
    balance roundrobin

    option httpchk
    option forwardfor
    option httplog
    option redispatch

listen port8080proxy
bind 0.0.0.0:8080
    mode http
    maxconn 500
    balance roundrobin
```

```
    option httpchk
    option forwardfor
    option httplog
    option redispatch


listen port9000proxy
bind 0.0.0.0:9000
    mode http
    maxconn 500
    balance roundrobin

    option httpchk
    option forwardfor
    option httplog
    option redispatch

    retries 3

    server test2 10.1.1.2:80 check maxconn 200
    server test3 10.1.1.3:80 check maxconn 200
```

Summary

Using the basic configuration, you have been able to build and execute your own simple, two server, one listener, load balancer on your Unix-based operating system using HAProxy.

In addition, you know why you've used the parameters presented, and have an idea how to implement some of HAProxy's other features such as debugging, statistics, default server parameters, backups, performance tuning with maximum connections, and multiple listeners.

You've also learned some of the basic ways to change forwarding, error handling, and health check headers.

In chapter 3, you'll learn how to make ACLs (access content lists) that will allow you to match header content against complex patterns and allow or block requests based on them.

3 Setting Up ACLs

In HAProxy, an access control list (ACL) defines a pattern based on IP address or port, request or response content, or an aspect of system status. ACLs are then used to create rules (or conditions) that allow or deny access to the system, or direct requests and responses to servers behind the HAProxy load balancer.

HAProxy's ACLs differ from ACLs in other systems by separating the definition of the pattern to be matched from the rule that uses it.

ACLs may identify an IP address that can be used in a rule to block requests from that source IP, pick out responses that use specific strings so that a rule can log them, or even search incoming requests for regular expressions, which contain literal characters but also wildcards and operators, in order to apply a rule which sends those requests to a selected server.

ACLs are not required in order to set up an HAProxy load balancer.

ACLs (as well as the rules that use them) can only be defined in the listen, frontend or backend sections of the configuration, not in the global, default, or server sections.

They are not available outside of the listen/frontend/backend in which they are defined, so you will need to duplicate any ACL you wish to use in another section or scope.

Because they cannot be reused in other scopes, you don't need to name ACLs uniquely in different sections.

In this chapter, you'll learn about constructing an ACL, along with all the options available to define it, and learn some of the uses of an ACL. In the chapter 4, you'll take what you've learned and apply it by creating rules that use your ACLs.

Anatomy of an ACL

ACLs can be used one at a time or in combination; multi-line ACLs have an implicit OR. For example, this ACL would match traffic when the Host: header starts with 'www' or ends with 'com':

```
acl exampleacl hdr_beg(host) -i www

acl exampleacl hdr_end(host) -i com
```

So, redefining an ACL in the same section adds creates one named ACL with two parts connected by an implicit OR, and your ACLs do not need to be listed in any order.

To combine two ACLs with an AND, use them both in a rule.

An ACL can be constructed from a variety of parts, including a keyword, name, pattern (criteria), flags, operators, and values, but the only parts required for a simple, complete ACL are the keyword, (which indicates that this line is an ACL), the name, and a pattern.

The table below shows the parts of an ACL line, gives examples and notes for each part, and explains the potential purpose of the example in order to clarify the role of each part.

Line part	Keyword	Name	Pattern	Flag	Operator	Value
Notes		Case-sensitive	Layer 4 or 7	Three supported: -i , -f, --	Integers & decimals only	Numbers, strings, regexes & IP addresses
ACL example	acl	blockDOS	hdr_ip	N/A	N/A	183.10.10.xxx
Example code	acl blockDOS hdr_ip –i 183.10.10.124					
Explanation	This ACL (blockDOS) will match an IP address (183.10.10.xxx) found in a header (hdr_ip). This example assumes the ACL will be included in a rule that blocks requests causing a denial of service. Because the value is an IP address, flags that ignore letter case or load patterns from a file are not applicable.					

Parts of an ACL Line

Name

The name creates an identifier for the ACL. It is case-sensitive, and like most other names and IDs in HAProxy, may only contain upper and lower case letters, numbers, dashes (-), underscores (_), dots (.), and colons (:).

Pattern

The pattern is also called the criterion or match. Some patterns can stand alone, such as always_true, and do not require a value. Others like hdr_ip require a value to create a complete ACL. A pattern may have an additional identifier directly after it (no space) in parentheses:

```
be_conn(<backend name>)
```

Flags

The following flags are supported in HAProxy 1.4:

- -i directs the configuration to ignore case during matching of subsequent patterns on this line of the ACL.
- -f loads patterns from a file or from multiple files.

If this flag is used in conjunction with -i, then all lines of the loaded file would be examined without reference to letter case.

- —- ends the flag part of the line, which ensures that the configuration will not see code later in the line as a flag.

The flag to explicitly close the flag section is particularly useful if you're matching a string or regex that begins with '-'. For example, if you want to test a custom response header from and application that should end in '-finished':

```
acl exampleacl shdr_end(X-Application-Header) -finished
```

instead you would need to explicitly terminate the flag section before the value:

```
acl exampleacl shdr_end(X-Application-Header) -- -finished
```

Operators

Only integers, integer ranges and decimal-like numbers can use operators for matching a value against a pattern. Operators are:

- eq (=) Equal to one or more values.
- gt (>) Greater than one or more values.
- lt (<) Less than one or more values.
- ge (> or =) Greater than or equal to one or more values.
- le (< or =) Less than or equal to one or more values.

For example, to produce an ACL that says, 'If the number of current backend connections is greater than 100, match,' the usage statement is:

```
acl <name> be_conn <integer>
```

… and the line in the configuration would be:

```
acl currbackconnsacl be_conn gt 100
```

Values

The value or values that will be matched against the specified pattern can be integers, decimal-like numbers, or ranges of these; strings or regular expressions (regexes); or IP addresses or networks. Multiple values are separated by spaces.

IP Addresses and Networks

ACLs can use IP addresses, ports, and networks as values.

We don't recommend using host names in ACLs, because they will be resolved only at startup, when the configuration is first read. Therefore, if a host name's address changes, it will not be correctly resolved until restart.

To define a block of IP addresses, use an IP address and netmask in either CIDR (e.g., 192.168.100.0/22) or dot-decimal (192.168.100.0/255.255.255.0) notation. (This example includes addresses from 192.168.100.0 to 192.168.103.255.)

HAProxy cannot define an ACL using a range of IP addresses, e.g., 192.168.100.0-192.168.200.00.

Examples

To create an ACL matching a single network address in the header:

```
acl unfriendlysrcacl src_dst 193.168.100.1
```

Using IP addresses to block unwanted requests is safer that using them to allow requests from specific IP addresses because a malicious request can simply include the allowed IP address as part of the header.

To create an ACL matching a block of addresses in the header:

```
acl unfriendlysrcacl src_dst 193.168.100.0/16
```

Integers or Numbers

In HAProxy 1.4, an 'integer' refers only to positive natural numbers. An ACL value can include positive natural numbers using ranges and operators, and some values can also use a decimal-like number format of 'nn.nn,' which functions the same as an integer.

This decimal-like number format can be used for matching version numbers, e.g, HAProxy version 1.4 versus HAProxy version 1.5.

Integer/number values can use ranges and operators to further refine or extend the numbers matched.

To create a value using a numerical range, separate the two ends of the range with a colon. The range is inclusive of the numbers stated.

If the top or bottom end of the range is inherent, you can exclude that end of the range. For example, the lowest number possible port number is zero and the highest is 65535. To set a port range from the zero to 1024, you can enter '0:1024' or ':1024.' To set a range from 1024 to the highest port, you can enter '1024:65535' or '1024:' as notation for a valid port range.

To create a value using an operator, insert the two-letter operator before the value with a space between.

Examples

To create an ACL that matches all requests from a source using SSLv3.1, use the following:

```
acl ssl31reqacl req_ssl_ver 3.1
```

To create an ACL that matches destination ports 80 and 8080, use the following:

```
acl destport80n8080acl dst_port 80 8080
```

To create an ACL that matches a range of destination ports from 80 to 8080, use the following:

```
acl destport80rng8080acl dst_port 80:8080
```

To create an ACL that matches all URIs coming from a source port less than 80, but excluding 80, use the following:

```
acl srcportlt80acl url_port lt 80
```

Strings

A string is a value that is matched exactly as it is written. A backslash '\' can be used to escape (ignore) characters such as spaces, comments (#), or backslashes that should be treated literally. Flags can be used before string values so that they are case-insensitive, or load from a file.

To indicate a value in which some strings are case-insensitive and others are case-sensitive, enter the case-sensitive strings before the flag, or use the '--' flag to end the '-i' flag before the case-sensitive strings.

Examples

To create an ACL that matches if the path contains the string 'example.com', use the following:

```
acl iccstringacl path example.com
```

To create an ACL that matches if the path contains the string 'example.com,' without case sensitivity, so that it also matches 'EXample.com' and 'example.COM' use the following:

```
acl iccstringacl path -i example.com
```

To create an ACL that matches if the path contains the string ' icantclick' with a space at the beginning, use the following:

```
acl iccstringacl path \ example.com
```

Regular Expressions

In HAProxy, regular expression or regex is a value that matches a set of possibilities against an HAProxy pattern. Regex matching is similar to string matching, using the same backslash and operator flags, but is able to search for a string that includes multiple wildcard characters, as well as Boolean-type operators.

HAProxy regexes use POSIX Extended Regular Expressions (ERE), a standardized version of a search capability created to search text files (most famously, grep).

POSIX is the acronym for 'Portable Operating System Interface.'

Why Regex?

While you could specify a set of values as strings, if there are more than one or two permutations, it can become cumbersome. To capture many possibilities, you can use a regex, which substitutes a wildcard for letters, numbers, operators, spaces, and so on, allowing you, for example, to search for strings that represent many possible ways to spell or misspell a name (Smith, smyth, smythe, smithe), but not other words like Smithsonian by specifying a regex such as '^[Ss]m[iy]the?$'.

To produce an ACL using a regular expression, it helps to know a useful subset of the metacharacters that can be used.

The most commonly used regex operators are:

Metacharacter	Use	Example
^ caret	Beginning of line	
$ dollar sign	End of line	
. dot	Matches any single character	match tit, tat, tot, txt, t%t, tΩt, etc. t.t

? question mark	Matches zero or one of the preceding element	match both 1234 and 124 123?4
* asterisk, star	Match zero or more of the preceding element	match 123, 1234, 12344, 12344, etc.: 1234*
+ plus sign	Match one or more of the preceding element	match lose and loose (but also looose, loooose, etc.) lo+se
| pipe, vertical bar	Boolean operator 'or'	match either of the two elements: this|that
[ ] brackets	Match a list of characters	Match this and that, but also thit and thas: th[ia][st]
( ) parentheses	Match a group of expressions	Match only this and that: th(is|at)
\t	Tab	
\r	Carriage return (CR)	
\n	New line (LF)	
\	Indicate that a space is a space rather than a field delimiter	
\#	Indicate that a hash mark is a hash mark rather than an indicator for a commented line	
\\	Indicate that a backslash (\) is a backslash in a regex	
\\\\	Indicate that a backslash (\) is a backslash in the text	
\xXX	Write the ASCII hex code XX	

To learn more about regexes, see the IEEE standards pages.

Layer 4—TCP and Proxy Patterns

ACLs created for TCP/IP addresses and ports and proxies don't need to look inside the contents of a request or response for information. HAProxy allows you to match live status details about the proxy itself, as well as IP addresses and select TCP request data.

These ACLS will discern things like how many connections are being made at different locations in the system, whether the session rate of a server is going over a predetermined number, or which IP addresses are sending requests. They can also match destination ports or determine if a request is in HTTP or SSL.

In this section, you'll learn what these patterns are and how to indicate them in an ACL. Each pattern shows a the usage statement.

Proxy Status

These patterns extract status and state information from the defined proxies.

queue and avg_queue

The queue pattern matches the total connections in queue. The avg_queue patterns match the total connections in queue divided by the number of servers in use in the group. The current backend or frontend is used if none is named.

```
queue(<backend>) <integer>
avg_queue <integer>
avg_queue(<backend>) <integer>
```

be_id, fe_id, so_id, svr_id

The be_id, fe_id, so_id, and svr_id patterns identify the backend, frontend, socket, or server from which a call originated. The server ID can be used in frontends or backends.

```
be_id <integer>
fe_id <integer>
so_id <integer>
srv_id <integer>
```

For example, in a backend, to identify traffic from a specific frontend or listen (set with the 'id' keyword) where we can't reference the 'name' in an ACL:

```
acl used_frontend_for_ssl fe_id 8443
```

be_conn, fe_conn, dst_conn

The be_conn, fe_conn, and dst_conn patterns apply the number of current connections on the backend, frontend, or socket. The current backend or frontend is used if none is named. These

patterns can be used to specify another backend when the first is at a certain number of connections or give a warning when a frontend had reached a number of connections.

```
be_conn <integer>
be_conn(<backend>) <integer>
fe_conn <integer>
fe_conn(<frontend>) <integer>
dst_conn <integer>
```

be_sess_rate, fe_sess_rate

The be_sess_rate and fe_sess_rate patterns match the rate per second of session creation. They can be used to limit service use.

```
be_sess_rate <integer>
be_sess_rate(<backend>) <integer>
fe_sess_rate <integer>
fe_sess_rate(<frontend>) <integer>
```

srv_is_up, nbsrv

These are two patterns you can use as part of a health check of the servers or server group. The current backend or frontend is used if none is named.

- The srv_is_up pattern matches when the identified server or backend is functioning, and doesn't match when it is down or in maintenance mode.

```
srv_is_up(<server>)
srv_is_up(<backend>/<server>)
```

- The nbsrv pattern matches when the number of functioning servers is equal to the value.

```
nbsrv <integer>
nbsrv(<backend>) <integer>
```

Source & Destination IP addresses & Ports

These patterns extract source and destination IP addresses and ports at the transport layer.

Src, dst

The src and dst patterns apply to the client's TCP source IP address or the local address the client connects to.

```
src <ip_address>
dst <ip_address>
```

The TCP source address is used, and not the address of a client behind a proxy.

Src_port, dst_port

The src_port and dst_port patterns apply to the client's TCP source port or the local port the client connects to.

```
src_port <integer>
```

```
dst_port <integer>
```

For example, to match TCP traffic based on source IP, subnet, or port:

```
acl blacklistsrcacl src 0.0.0.0/7 224.0.0.0/3
```

```
acl blacklistsrcacl src_port 0:1023
```

Here, you match a few known source networks that should never be seen as a source of HTTP traffic, as well as any traffic originating from port 1023.

To match traffic targeted to a specific bind port:

```
acl viasslacl dst_port 81
```

For example, if you terminate SSL with SSL termination software, you could direct the SSL terminator to a specific bind port, and use this ACL to identify that traffic.

TCP requests

HAProxy provides a few patterns to match specific types of application data from the TCP request.

req_len

The req_len patterns matches when the data length in the request buffer matches the value. When the buffer is still changing within a session, a rule using this ACL will always match.

```
req_len <integer>
```

The HAProxy must verify that the buffer is receiving no more data before a rule using this ACL will not match.

req_proto_http

The req_proto_http pattern matches when request buffer data appears to be (and parses) as HTTP.

```
req_proto_http
```

req_rdp_cookie

The req_rdp_cookie pattern matches when the request buffer both has data that looks like RDP, and a cookie equal to the string value. Any cookie is checked unless a name is specified. As in the RDP specification, only the first cookie is examined. The cookie name is case-insensitive.

```
req_rdp_cookie <string>
```

```
req_rdp_cookie(<name>) <string>
```

req_rdp_cookie_cnt

The req_rdp_cookie_cnt pattern matches when the request buffer both has data that looks like RDP, and the number of RDP cookies matches the value. Any cookies are checked unless a name is specified.

```
req_rdp_cookie_cnt <integer>
```

```
req_rdp_cookie_cnt(<name>) <integer>
```

req_ssl_ver

The req_ssl_ver pattern matches when the request buffer has data that looks like SSL, and a protocol version that matches the value. Both SSLv2 and SSLv3 messages are supported. TLSv1 is announced as SSL version 3.1.

```
req_ssl_ver <decimal>
```

Layer 7—HTTP Patterns

These patterns examine the contents of a request or response at the HTTP or application layer. The request or response can only be evaluated after the full request or response has been received.

These patterns allow you to match the request method, request/response headers, the URI, and the path (a portion of the URI before any query parameters).

This is an example request:

```
GET /missingfile HTTP/1.1
User-Agent: curl/7.22.0 (x86_64-apple-darwin10.8.0)
libcurl/7.22.0 OpenSSL/1.0.0e zlib/1.2.5 libidn/1.22
Host: www.example.com
Accept: */*
```

This is an example response:

```
HTTP/1.1 404 Not Found
Date: Sat, 07 Sep 2013 02:03:17 GMT
Server: Apache/2.2.4 (FreeBSD) mod_ssl/2.2.4 OpenSSL/0.9.7e-p1
DAV/2 PHP/5.2.3
Content-Length: 202
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
```

```
<p>The requested URL /test was not found on this server.</p>
</body></html>
```

Headers

Header patterns using ‘hdr’ look at request headers; ‘shdr’ patterns (‘server header’) examine response headers.

Header matching in HAProxy is in compliance with RFC 2616, which delimits headers using commas.

The header name is not case-sensitive.

Hdr, Shdr

The hdr and shdr patterns match if any headers match any of the strings. This ACL pattern applies to all headers unless a header name is specified in the pattern.

```
hdr <string>
hdr(header) <string>
shdr <string>
shdr(header) <string>
```

For example, to match a specific host in the Host: header, use:

```
acl is.example.com hdr(host) -i example.com
```

The HTTP header ‘Host:’ is added to every request by the browser, and therefore contains the host name of the URL requested by the user.

As DNS is not sensitive to case, the host names are not case-sensitive.

Hdr_beg, shdr_beg

The hdr_beg and shdr_beg patterns match when one of the headers begins with one of the strings. See ‘hdr’ for more information on header matching.

```
hdr_beg <string>
hdr_beg(<header>) <string>
```

For example, to match any host beginning with ‘static’:

```
acl is.www hdr_beg(host) -i static
```

Hdr_cnt, shdr_cnt

The hdr_cnt and shdr_cnt patterns match when the number of occurrences of the header matches the value. You can use this pattern to call out when a specific header occurs more often than you expect.

```
hdr_cnt <integer>
hdr_cnt(<header>) <integer>
```

hdr_dir, shdr_dir

The hdr_dir and shdr_dir patterns match when a header contains the value behind or between slashes (/). They are used to search for filenames or directory names.

```
hdr_dir <string>
```

```
hdr_dir(<header>) <string>
```

hdr_dom, shdr_dom

The hdr_dom and shdr_dom patterns match when a header contains the value behind or between dots (.). They are used to search for domain names.

```
hdr_dom <string>
```

```
hdr_dom(<header>) <string>
```

For example, to identify any traffic targeting a specific website hosted behind your HAProxy, match the host header for a specific site using the following:

To match anything containing ‘example.com’ in the domain, (including strings such as ‘www.example.com’ and ‘random.example.com’), use:

```
acl is.example.com hdr_dom(host) -i example.com
```

hdr_end, shdr_end

The hdr_end and shdr_end patterns match when a header ends with the specified value.

```
hdr_end <string>
```

```
hdr_end(header) <string>
```

To match any host ending with ‘.com’:

```
acl is.dot.com hdr_end(host) -i .com
```

This type of match could be used to serve a maintenance or out-of-order page for a specific website site, or to route traffic to a specific backend, etc.

hdr_ip, shdr_ip

The hdr_ip and shdr_ip patterns match when a header contains an IP address value. This is mainly used with headers such as X-Forwarded-For or X-Client-IP.

```
hdr_ip <ip_address>
```

```
hdr_ip(<header>) <ip_address>
```

hdr_len, shdr_len

Returns true when at least one of the headers has a length which matches the values or ranges specified. This may be used to detect empty or too large headers.

```
hdr_len <integer>
```

```
hdr_len(<header>) <integer>
```

hdr_reg, shdr_reg

Returns true when one of the headers matches of the regular expressions. It can be used at any time, but it is important to remember that regex matching is slower than other methods.

```
hdr_reg <regex>
```

```
hdr_reg(header) <regex>
```

hdr_sub, shdr_sub

sub-string Returns true when one of the headers contains one of the strings.

```
hdr_sub <string>
```

```
hdr_sub(header) <string>
```

hdr_val, shdr_val

Returns true when one of the headers starts with a number which matches the values or ranges specified. This may be used to limit content-length to acceptable values for example.

```
hdr_val <integer>
```

```
hdr_val(header) <integer>
```

To identify HTTP server responses with no content (empty pages, for example):

```
acl is.empty.response shdr_val(content-length) lt 0
```

URLs

For the purposes of HAProxy ACLs, the URL is defined as the string located after the method in the HTTP request, inclusive of query parameters, if present.

For example, the request for:

http://www.example.com/index.html?value=true

would be:

```
GET /index.html?value=true
```

```
GET is the method, /index.html?value=true is the URL.
```

The URL does not include the ‘http://’, nor the ‘www.example.com’ hostname. To match the hostname, you'll need to inspect the Host: header.

url

The url pattern matches the whole URL, (as defined by HAProxy), passed in the request.

```
url <string>
```

url_beg

The url_beg pattern matches when the URL begins with a specific value.

```
url_beg <string>
```

To match a specific URL:

```
acl is.store url /store
```

To match URLs that start with '/store':

```
acl is.store url_beg /store
```

url_port, url_ip

The url_port pattern matches the port or IP specified in the absolute URI in an HTTP request and is only used with option http_proxy.

Port 80 is assumed if the port is not specified in the request.

```
url_port <integer>
```

Other url patterns

The path patterns url_dir, url_dom, url_end, url_len, url_reg, and url_sub use the same logic as the header (hdr) patterns of the same type.

Paths

The path is the portion of the URI that that precedes the optional query parameters.

In this URL, http://www.example.com/index?list=true&mode=http
which has two query parameters, the path is /index .

The query parameters are separated from the path by a '?'. Using path matching lets you easily constrain your match.

path

Returns true when the path part of the request, which starts at the first slash and ends before the question mark, equals one of the strings. It may be used to match known files, such as /favicon.ico.

```
path <string>
```

Other Path Patterns

The path patterns path_beg, path_dir, path_dom, path_end, path_len, path_reg, and path_sub use the same logic as the header (hdr) and URL (url) patterns of the same type.

Other Layer 7 Patterns

HAProxy allows you to match the request method, identify the first request in a multi-request connection, as well as perform basic HTTP authentication.

http_auth, http_auth_group

The http_auth pattern matches when authentication data username and password from the userlist is equal to the specified value. http_auth_group matches if the user is assigned to a group.

```
http_auth(userlist)
http_auth_group(userlist) <group> [<group>]*
```

In HAProxy 1.4, only http basic auth is supported.

http_first_req

The http_first_req pattern matches when the first request of the connection is processed. You can use this pattern to apply some rules only on the first request.

```
http_first_req
```

method

The method pattern matches the method in the HTTP request, e.g,, GET, HEAD, POST PUT, OPTIONS, or any other method supported by HTTP. (Hard-coded ACLs, discussed below, check for most common methods.)

```
method <string>
```

To identify HTTP requests that use the HEAD or OPTIONS requests (frequently used for testing http connectivity, and for checking cache expirations):

```
acl is_head_or_options method HEAD OPTIONS
```

req_ver

The req_ver pattern matches the version in the HTTP request.

```
req_ver <string>
```

status

The status pattern matches the HTTP status code in the response.

```
status <integer>
```

For example, the status code 404 is found in the response example from the beginning of the section:

```
HTTP/1.1 404 Not Found
```

This ACL will match that status code:

```
acl statusacl status 404
```

Other ACLs

HAProxy provides a few predefined ACLs, and also allows you to define your ACLs in-line when you need them (referred to as anonymous ACLs), instead of creating a named ACL. A few additional patterns assist you in debugging your configuration.

Hard-coded ACLs

Some predefined ACLs are hard-coded so that they do not have to be declared in advance. Hard-coded ACLs use upper case. Some of the most useful are:

- FALSE always_false
- HTTP req_proto_http
- HTTP_1.0 req_ver 1.0
- LOCALHOST src 127.0.0.1/8
- RDP_COOKIE req_rdp_cookie_cnt gt 0
- TRUE always_true

Anonymous ACLS

Anonymous ACLs are created in-line, instead of needing to be names and defined. Enclose anonymous ACLs in curly braces { }, with a space before and after the expression.

For example, for an ACL in this rule:

```
acl is.get method GET
allow if is.get
```

can be defined anonymously in this way:

```
allow if { method GET }
```

Debugging patterns

Three patterns that you can use to help debug your TCP and proxy ACLs are:

- always_false Never match, ignore all values and flags set in the ACL.
- always_true Always match, but ignore all values and flags set in the ACL.
- wait_end Wait for the end of the analysis period to match.

Since wait_end either stops the evaluation or immediately matches, this ACL should be listed as the last one in a rule.

Summary

Now you are able to set up ACLs to match against all parts of a TCP or HTTP request header or HTTP response header, and you know many of the potential options for designing an ACL for status and different parts of a header

Next, you'll take what you've learned about ACLs and begin using it to create rules that will allow you to take more control of your HAProxy load balancer.

4 Modifying HTTP Traffic

In HAProxy, an ACL defines a pattern based on IP address or port, request or response content, or an aspect of system status to allow and block connections, restrict users and applications, and route traffic to specific servers at layer 7 (HTTP).

In this chapter, you'll learn how to modify HTTP traffic using rules that consume ACLs, and what keywords you can use. You'll also learn how to use ACLs and rules in manipulating request/response and header lines and in routing traffic to various servers.

Finally, you'll see examples of full configurations in HAProxy that take advantage of ACLs, rules, header manipulation and HTTP routing.

Rule Design Tips

Some information to keep in mind as you use this chapter:

Rules Using Multiple ACLs

Multiple ACLs are joined by the rule operator 'AND' implicitly, so without an explicit OR (optional) or NOT) (negative) operator, all of the ACLs indicated must be matched in order for the rule to be executed.

The operators OR and NOT are stated as follows:

- OR — Double vertical bar or pipe, '||', which can also be stated as 'OR'.
- NOT — Exclamation point, '!'.

Implicit AND

When you create a rule that uses multiple ACLs, the ACLs are joined by an implicit AND unless otherwise specified. You can only specify AND implicitly, by leaving it out.

In contrast, the parts of one ACL are linked by an implicit ‘OR’.

Here’s an example of a rule that uses only the implicit ‘AND’:

To block POST requests, but only when sourced from localhost (using hardcoded ACLs):

```
block if LOCALHOST METH_POST
```

Explicit OR (||)

To join multiple ACLs in a rule with an or, use OR, or the ||.

The following example shows a rule in which the operator OR is used:

To block any request that sources from localhost, as well as any request with a method of POST:

```
block if LOCALHOST or METH_POST
```

alternatively,

```
block if LOCALHOST || METH_POST
```

Explicit NOT (!)

To reverse the behavior of an ACL in a rule, (an ACL that would ‘match’ will now ‘not match’), use the ‘not’ operator, ‘!’.

The following example shows you a rule in which the operator NOT is used.

To only allow requests from localhost if they use any method except POST:

```
block if LOCALHOST NOT METH_POST
```

Common Arguments

Arguments used repeatedly throughout rules are explained here. Those that are used once, as well as any caveats or additions, are explained in each section.

- <regex> As previously defined in Chapter 3, Setting Up Acccess Control Lists, a regex, or regular expression, is a value that matches a set of possibilities against a pattern. Parenthetical grouping is supported; the backslash before the expression is not required. (In HAProxy documentation, the argument is referred to as ‘<search>’.)
- <acl> In some rules, the use of an ACL is optional, indicated by placing the construction within brackets: [{ if | unless } <acl>]
  (In HAProxy documentation, the ACL argument is referred to as ‘<cond>’ or ‘<condition>’.)
- <string> As previously defined in Chapter 3, Setting Up Access Control Lists, a string is a value that is applied exactly as it is written.

A backslash ‘\’ must be used to escape (ignore) characters such as spaces, comments (#), or backslashes that should be treated literally.

Ignore Case

To ignores case when matching, use the form of the keyword with the ‘i’’ in the middle. An example is that ‘reqdel’ becomes ‘reqidel’. Any keyword with this option is indicated in its section.

Rule Order

Rules and rule statements are applied using the following order:

- For requests, frontend rules are applied first, then backend rules.
- For responses, backend rules are applied first, then frontend rules.
- Block rules are applied before request rules.
- The use_backend keyword is used after request rules so headers and request lines can be changed before routing traffic to a specific backend.

Matching Whole Lines

For response and request rules, HAProxy only evaluates whole lines. Header names and header values cannot be matched separately.

The first line of the request, the ‘request-line’, is treated as a header. This makes it possible to rewrite request methods, URLs, or response codes using req or rsp rules.

Helpful regexs for matching headers or request-lines:

- ^[^\ \t]*[\ \t] applies to any method, space, and prefix.
- ^[^ \t:]*: applies to any header name and colon.

Controlling Access

The block keyword is used to control access using keywords, allowing you to block or allow traffic access to specific resources.

The keyword reqtarpit is a special case of blocking for use with requests so that the requestor is not made aware that the request has been blocked.

block

The block keyword is used to block (deny) or to explicitly allow certain requests, while blocking all other traffic.

```
block { if | unless } <acl>
```

You can also block (deny or explicitly allow) a request or a response using keywords such as reqdeny and reqallow, but it is not recommended, as ‘block’ rules using ACLs are more effective and easier to construct.

Block a Request

To block a request at layer 7 and return a 403 error to the requestor, use the keyword block with the conditional term ‘if’.

```
block unless <acl>
```

Examples:

At the TCP request level, you would like to block a selection of networks and addresses that you’ve determined to be sources of traffic you don't want:

```
These are RFC5737 reserved for docu.
acl blacklisted_ips src 192.0.2.0/24
acl blacklisted_ips src 198.51.100.35
acl blacklisted_ips src 198.51.100.40
acl blacklisted_ips src 203.0.113.93
block if blacklisted_ips
```

At the HTTP request level, block requests for static content if the referrer header doesn't match your website's URL. This helps to prevent other websites from cross-linking to your images.

Specifically, if the URL contains /static/, check for a referrer header for your domain (example.com).

```
acl our_referer hdr_dom(referer) -i example.com
acl is_static url_dir /static/
block if is_static our_referer
```

Block for Each Case until an ACL is Matched

To block traffic for multiple cases and allow specific traffic to pass, you can use the block keyword with ‘unless’.

For example:

```
acl is_example hdr_dom(host) -i example.com
acl is_local hdr_dom(host) -i local.example.com
acl is_www hdr_dom(host) -i www.example.com


block unless is_local block unless.....
block if is_example
```

Explicitly Allow a Request

To allow content, use the keyword block with the conditional term ‘unless’. This keyword allows you to create a rule in which you can explicitly allow content while blocking any other content. Blocked content returns a 403 error to the requestor.

```
block unless <acl>
```

For example, in the following rule the ACL matches the source IP address block 224.0.0.0/4 on any port and destined for any backend, but allows all other traffic to pass.

```
acl invalid.src src 224.0.0.0/3
```

```
block unless invalid.src
```

For example:

If you want to deny all traffic except traffic sourced from a specific address range, use block with the conditional term ‘unless’:

```
acl our_network src 192.0.2.0/24
```

```
block unless our_network
```

reqtarpit, reqitarpit

Use reqtarpit to tarpit an HTTP request containing a line matching a regular expression.

Using a tarpit is designed to slow an attacks on a server. Robot attacks often keep a connection open, waiting for a reply. A tarpitted request will not connect, but will be kept open for the amount of time defined in ‘timeout tarpit’ or, if not set, ‘timeout connect’. Eventually it will return a 500 error (unrecoverable internal error) to the requestor, so that the robot attacker does not suspect a rejection.

Logs will reflect a 500 error, but also indicate the request was tarpitted with the PT flag.

This rule cannot be used in the defaults section.

```
reqtarpit <regex> [{if | unless} <acl>]
```

```
reqitarpit <regex> [{if | unless} <acl>] (ignore case)
```

For example:

You are getting abusive requests from a crawler bot that identifies itself by the User-Agent header of "BadRobot"

Instead of simply blocking the requests quickly, you’d prefer to tarpit them. reqtarpit works with or without an ACL, so we'll show both examples:

Without an ACL:

```
reqtarpit ^User-Agent:\ .*[Bb][Aa][Dd][Rr][Oo][Bb][Oo[Tt].*
```

With an ACL:

Using case-insensitive substring matching with an ACL is much more straightforward. Note that even with an ACL, you still need to provide a regex for reqtarpit, but since the ACL handles the specifics, you use a simple header name match:

```
acl is_badrobot hdr_sub(User-Agent) -i BadRobot
```

```
reqtarpit ^User-Agent: if is_badrobot
```

Manipulating Header Content

You can change header and request and response line content in order to affect traffic, monitoring and logging.

No more than 4 kBs of characters can be added to a request or response header in order to prevent performance slowdowns.

reqadd

Use the rspadd keyword to add new header line <string>, plus an extra line, after the last header of the request. This only applies to requests passing through the proxy, not to requests generated by it.

These keywords may not be used in the defaults section.

```
reqadd <string> [{if | unless} <acl>]
```

For example, to add "thisisanewheader" to requests through port 80, use the following construction:

```
acl new.header.acl dst_port 80
reqadd thisisanewheader if new.header.acl
```

rspadd

Use the rspadd keyword to add new header line <string>, plus an extra line, after the last header of the response. This only applies to responses passing through the proxy, not to responses generated by it, such as health checks and errors.

This keyword may not be used in the defaults section.

```
rspadd <string> [{if | unless} <acl>]
```

For example, to add "thisisaresponseheader" to responses made from a backend, use the following construction:

```
acl new.resheader.acl
rspadd thisisaresponseheader if new.resheader.acl
```

reqdel, reqidel

Use the rspadd keyword to delete all headers in the request matching the regex defined. You can use this to remove unwanted or extraneous headers. This only applies to responses passing through the proxy, not to responses generated by it, such as health checks and errors.

This keyword may not be used in the defaults section.

```
reqdel <regex> [{if | unless} <acl>]
reqidel <regex> [{if | unless} <acl>] (ignore case)
```

For example, if you would like to remove the header ‘pollywantacookie’, use a configuration line similar to the following:

```
reqidel ^pollywantacookie.*
```

rspdel, rspidel

Use the rspdel keyword to delete all headers in the response matching the regex defined. You can use this to remove unwanted or extraneous headers. This only applies to responses passing through the proxy, not to responses generated by it, such as health checks and errors.

These keywords may not be used in the defaults section.

```
rspdel <regex> [{if | unless} <acl>]
rspidel <regex> [{if | unless} <acl>] (ignore case)
```

For example, to remove the server from the response, using the following:

```
reqidel ^server:.*
```

reqrep, reqirep

Use the reqrep keyword to replace all headers in the request matching the regex defined with a given string. You can use this to replace unwanted or extraneous headers. This only applies to responses passing through the proxy, not to responses generated by it, such as health checks and errors.

Using this keyword, ‘string’ is the complete line to be added.

```
reqrep <regex> <string> [{if | unless} <acl>]
reqirep <regex> <string> [{if | unless} <acl>] (ignore case)
```

For example, in the following, to replace ‘/static/’ with ‘/’:

```
reqrep ^([^\ :]*)\ /static/(.*) \1\ /\2
```

To replace ‘www.example.com’ with ‘www’ in the host name:

```
reqirep ^Host:\ www.example.com Host:\ www
```

rsprep, rspirep

Use the reqrep keyword to replace all headers in the response matching the regex defined with a given string. You can use this to replace unwanted or extraneous headers. This only applies to responses passing through the proxy, not to responses generated by it, such as health checks and errors.

Using this keyword, ‘string’ is the complete line to be added.

```
rsprep <regex> <string> [{if | unless} <acl>]
```

```
rspirep <regex> <string> [{if | unless} <acl>] (ignore case)
```

For example, to replace ‘Location: 127.0.0.1:80’ with "Location: www.example.com’, use the following construction:

```
rspirep ^Location:\ 127.0.0.1:80 Location:\www.example.com
```

Other ACL-using Rules

These keywords will be covered in the chapters on statistics and monitoring and special topics.

- stats admin
- stats http-request
- monitor fail
- http-request
- force-persist
- ignore-persist

Routing Traffic

Routing (or switching) traffic primarily uses the redirect keywords and the use_backend keyword to send requests or responses.

redirect location, redirect prefix

A redirect is an HTTP response requesting the sender issue a new request to a different URL. A redirect contains a redirect code and a target URL.

The redirect location keyword generates a redirect, unless an ACL is specified using{ if | unless }. The redirect prefix keyword adds a prefix to the URL and generates a redirect, unless an ACL is specified using{ if | unless }.

redirect location <url> [code <code>] <option> [{if | unless} <acl>]

redirect prefix <prefix> [code <code>] <option> [{if | unless} <acl>]

These keywords may not be used in the defaults section.

Options include:

- <code> indicates the redirect code to return: 301: Permanently moved, cache the new location; 302: Permanently moved, do not cache the new location. Default.; 303: Permanently moved, do not cache the new location, fetch location using GET.; 307: Permanently moved, do not cache the new location, fetch location using the same

method.; 308: Permanently moved, cache the new location, fetch location using the same method.

- ‘set-cookie NAME[=value]’ adds a session cookie header with a name or a value to the response to indicate that this user has been seen before.
- ‘clear-cookie NAME[=]’ adds a session cookie header with a name or an ‘=’ sign to the response, but with ‘max-age’ set to zero to direct the browser to delete the cookie.

Additional options for redirect prefix include:

- ‘drop-query’ sets the location without any query-string.
- ‘append-slash’ is used with ‘drop-query’ for requests sent without an ending ‘/’, asking them to resend the same URL with a final ‘/’.

If the <prefix> is a backslash ‘/’, nothing is added to the URL, allowing the redirect to ask for a request to the same URL.

For example, to require users to log in using a secure method, you can use the following rules to put the URL for login on an HTTPS.

```
acl nonsecure.acl dst_port 80
acl secure.acl dst_port 8080
acl login.page url_beg /login
acl logout.acl url_beg /logout
acl userid.acl url_reg /login?userid=[^&]+
acl cookie.set.acl hdr_sub(cookie) SEEN=1

redirect prefix https://example.com set-cookie SEEN=1 if
!cookie_set.acl
redirect prefix https://example.com if login_page !secure.acl
redirect prefix http://example.com drop-query if login_page
!userid.acl
redirect location http://example.com/ if !login_page secure.acl
redirect location / clear-cookie USERID= if logout.acl
```

use_backend

The keyword use_backend routes requests to a specific backend when an ACL applies. Rules are applied in order, so the first backend matched will be assigned.

If no ACL-defined backend is valid, the default backend will be used. If no default is defined and the section is ‘listen’, servers defined in the section are used. If the section is ‘frontend’, a 503 error (service unavailable) is returned to the requestor.

This keyword may not be used in the defaults or backend sections.

The keyword can also be used with TCP so that it is possible to route from a TCP frontend to an HTTP backend. In this case, either the frontend has already checked that the protocol is HTTP,

and backend processing will immediately follow, or the backend will wait for a complete HTTP request to get in. This feature is useful when a frontend must decode several protocols on a unique port, one of them being HTTP.

```
use_backend <backend> if <acl>
use_backend <backend> unless <acl>
```

There is no built-in limit to the number of use_backend rules.

To learn more about using routing rules, see the HTTP routing section.

Configuration Examples

In this section, you'll be able to see several common routing situations and how you can use HAProxy's ACL-based rules to appropriately move your traffic from listener to server, frontend to backend.

More configuration examples and more information on HTTP routing will be added as time allows.

Route Traffic to Three Servers

To configure HAProxy to route requests to three different servers, use a configuration based on the following example:

```
global
    maxconn 1000

defaults
    mode http
    timeout connect 5000ms

frontend http-in
    bind *:80
    default_backend big.backend

backend big.backend
    server www 10.0.1.20:7575 maxconn 350
    server store 10.0.1.21:7575 maxconn 350
    server image 10.0.1.22:7575 maxconn 350
```

```
listen admin
    bind *:8080
    stats enable
```

You may want to define your defaults more strictly to suit your needs or set up logging that shows you the volume of traffic to these servers. This will help you to tune your configuration to make the best used of your servers.

Note that in this configuration, the server maxconn is set higher than the global maximum connections. This works well when you do not expect all three of these servers to be equally busy at any one time.

Summary

At the end of this chapter, you should have a grasp of the very basics of configuring HAProxy using ACLs in rules to manipulate headers and route HTTP traffic to servers on the backend, giving you the ability to police traffic at layer 7.