

Solutions Benchmarking

HAProxy Forwards Over 2 Million HTTP Requests per Second on a Single Arm-based AWS Graviton2 Instance

April 8th, 2021 16 min read



Willy Tarreau



For the first time, a software load balancer exceeds 2-million RPS on a single Arm instance.

A few weeks ago, while I was working on an HAProxy issue related to thread locking contention, I found myself running some tests on a server with an 8-core, 16-thread Intel Xeon W2145 processor that we have in our lab. Although my intention wasn't to benchmark the proxy, I observed HAProxy reach 1.03 million HTTP requests per second. I suddenly recalled all the times that I'd told people around me, "The day we cross the million-requests-per-second barrier, I'll write about it." So, I have to stand by my promise!

I wanted to see how that would scale on more cores. I had got access to some of the new Arm-based AWS [Graviton2 instances](#) which provide up to 64 cores. To give you an idea of their design, each core uses its own L2 cache and there's a single L3 cache shared by all cores. You can see this yourself if you run `lscpu` on one of these machines, which will show the number of cores and how caches are shared:

\$ lscpu -e

view raw

CPU	NODE	SOCKET	CORE	L1d:L1i:L2:L3	ONLINE
0	0	0	0	0:0:0:0	yes
1	0	0	1	1:1:1:0	yes
2	0	0	2	2:2:2:0	yes
3	0	0	3	3:3:3:0	yes
...					
60	0	0	60	60:60:60:0	yes
61	0	0	61	61:61:61:0	yes
62	0	0	62	62:62:62:0	yes
63	0	0	63	63:63:63:0	yes

I had been extremely impressed by them, especially when we were encouraged by [@AGSaidi](#) to switch to the new Arm Large System Extensions (LSE) atomic instructions, for which we already had some code available but never tested on such a large scale, and which had totally unlocked the true power of these machines. So that looked like a fantastic opportunity to combine everything and push our benchmarks of HAProxy to the next level! If

HAProxyConf 2025

Registrations are open! [Learn more](#)

✉ Subscribe to our blog

🏠 Blog

🔗 Share

HAProxy Forwards Over 2 Million HTTP Requests per Second on a Single Arm-based AWS Graviton2 Instance

Synopsis

Methodology

Results

TLS for Free

Analysis

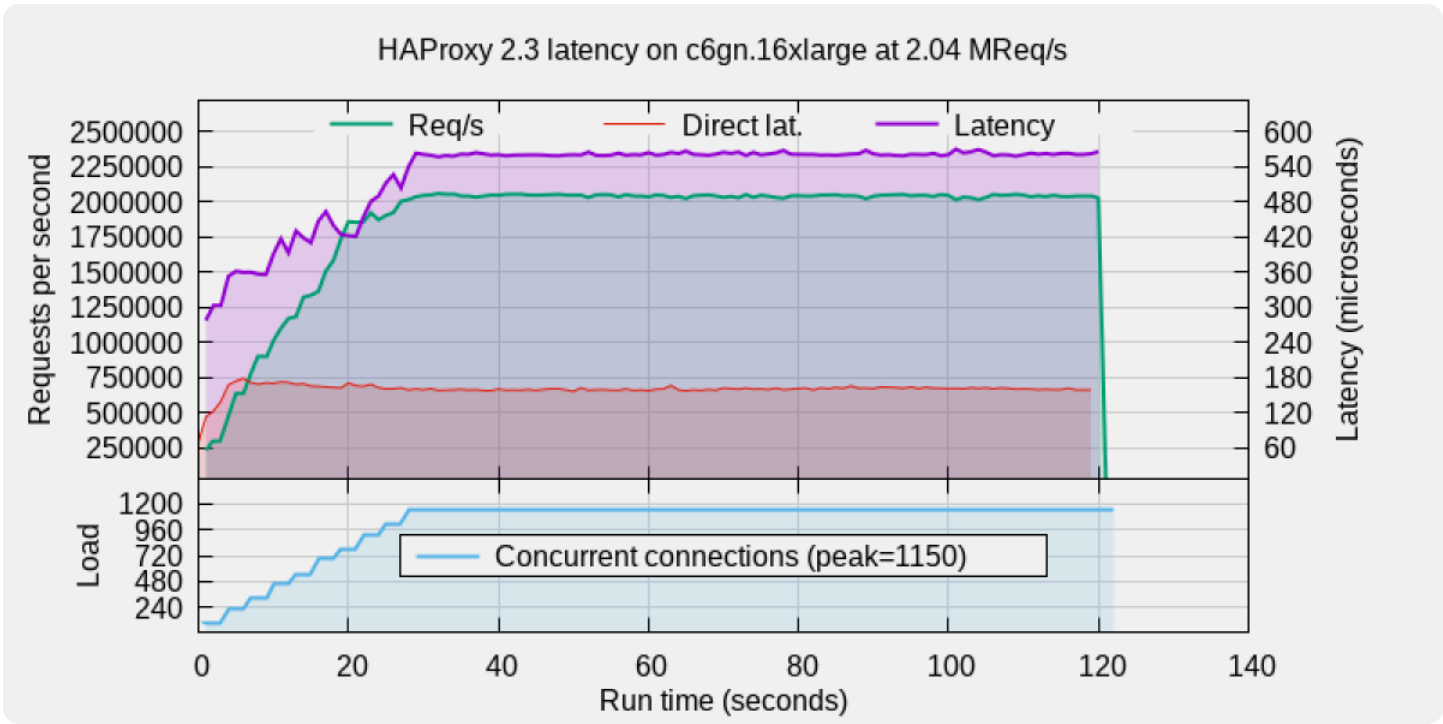
Conclusion

you are compiling HAProxy with gcc 9.3.0, include the flag `-march=armv8.1-a` to enable LSE atomic instructions. With gcc version 10.2.0, LSE is enabled by default.

Synopsis

Yes, you’ve read the title of this blog post right. HAProxy version 2.3, when tested on Arm-based AWS Graviton2 instances, reaches 2.04 million requests per second!

HAProxy 2.4, which is still under development, surpasses this, reaching between 2.07 and 2.08 million requests per second.



If you’re curious about the testing methodology, the ways in which the systems were tuned, and the lessons learned, read on.

A new project called [dpbench](#) has been launched in collaboration with a few members of the NGINX team, which captures some of the best practices for benchmarking proxies. It offers guidance to anyone wanting to run similar experiments without falling prey to some common misconceptions. Note that the project welcomes contributors who could provide more tuning/reporting scripts, configs for various environments, results and tips.

Methodology

In these benchmarks, we’re testing two things:

- HTTP requests per second
- End-to-end request latency at and above the 99.99th percentile

Tail latencies—which are latencies that affect a small number of requests—are something extremely important to not overlook, especially in service meshes where one request from the user can result in many microservice requests on the backend. In these environments, a latency spike creates an amplification factor.

For example, if a user’s request translates to 20 backend requests, a bigger latency affecting only 1 in 10,000 of the backend requests will affect 1 in 500 of the user’s requests. If that user makes 50 requests during a visit, it’s even more likely to manifest. If the increase in latency is low, or if it’s extremely rare, then the problem isn’t dramatic, but when it’s both high and frequent it’s a concern. So, it’s important that we measure this during the benchmarks.

Instance size

We used AWS’s *c6gn.16xlarge* virtual machine instances, which are 64-core machines from the [c6gn series](#) with access to 100 Gbps of network bandwidth. They’re compute-optimized using Graviton2 processors and are built for applications that require high network bandwidth. This instance size is used for the client, proxy, and server.

I first started with a 16-core, then a 32-core, instance thinking this could have been enough. A direct test from the client to the server, without HAProxy in between, disappointed me a bit with only 800k packets per second, or a bit less than the regular *c6g* one. I then decided to go for the full one: 64 cores on a dedicated host. This time I got my 4.15 million packets per second and thought that it would be sufficient, since I really only needed 2 million to achieve 1 million requests per second, accounting for one packet for the request and one for the response.

All machines were installed with Ubuntu 20.04, which is among the installations proposed by default.

IRQ affinity

It took me a while to figure out how to completely stabilize the platform because while virtualized, there are still 32 interrupts (aka IRQs) assigned to the network queues, delivered to

32 cores! This could possibly explain the lower performance with a lower number of cores. 32 queues can sound like a lot to some readers, but we’re speaking about 100 Gbps networking here and millions of packets per second. At such packet rates, you definitely do not want any userland process to hit one of these cores during the middle of your test, and this is what was happening since these cores were more or less randomly assigned to processes.

Moving the interrupts to the 32 upper cores left the 32 lower ones unused and simplified the setup a lot. However, I wondered if the network stack would support running on only 16 cores with 2 interrupts per core, which would leave 48 cores for HAProxy and the rest of the system. I tried it and found it to be the optimal situation: the network saturates at around 4.15 million packets per second in each direction and the network-dedicated cores regularly appear at 100%, indicating interrupts are being forced to queue up (`ksoftirqd`).

At this level, whenever something happens on the system—for example, the *snaped* daemon was waking up every 60 seconds, *crond* was checking its files every 60 seconds—it has to share one core with either the network or HAProxy, which causes quite visible latency spikes of one millisecond or more. I preferred to stop these services to limit the risk of random stuff happening in the background during the tests:

```
$ sudo systemctl stop irqbalance
$ sudo systemctl stop snapd
$ sudo systemctl stop cron
```

view raw

This definitely is a good example of parasitic activities, justifying leaving a few cores available for remaining activities. Thus, I figured that I’d leave 2 cores available and bind every non-load related activity there.

In short, the test was run with the network interrupts bound to cores 48-63 on all machines (total 16 cores), HAProxy/client/server enabled on cores 2-47 (total 46 cores) and cores 0-1 left available for everything else (`sshd` and monitoring tools).

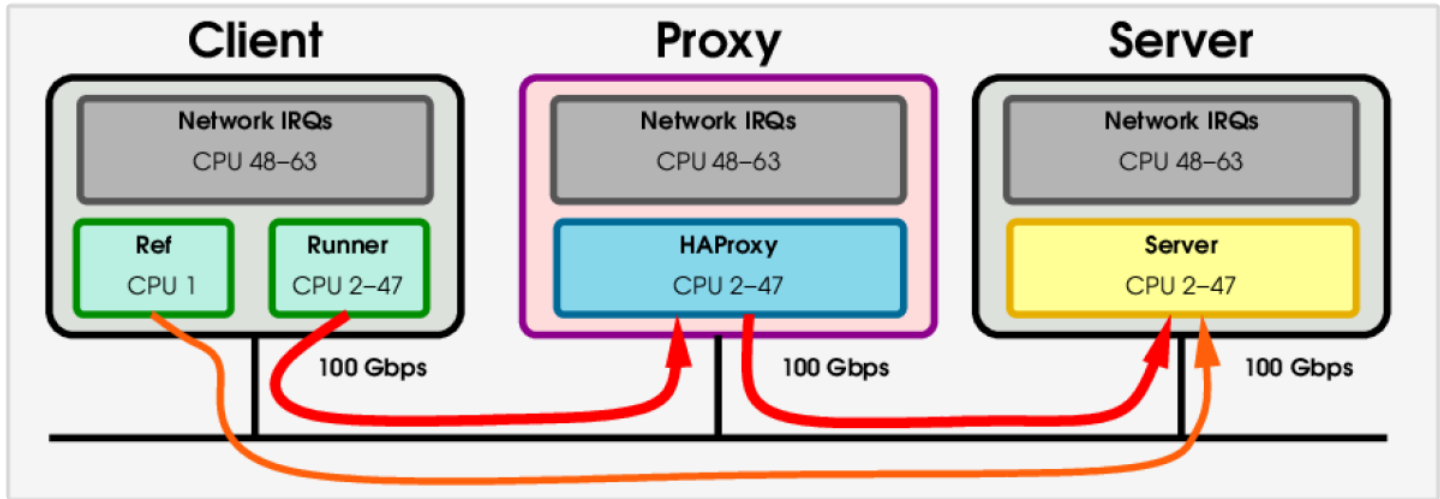
The IRQs are bound to the 16 upper cores with this command using the *set-irq.sh* tool from the *dpbench* project mentioned before:

```
$ sudo ~/dpbench/scripts/set-irq.sh ens5 16
```

view raw

Network topology

The whole setup was kept fairly simple and looks like this:



The server software was [httpterm](#), which started on 46 cores. The `ulimit` command increases the number of open file descriptors (i.e. slots for connections) available:

```
$ ulimit -n 100000
$ for i in {2..47}; do taskset -c 2-47 httpterm -D -L :8000;done
```

view raw

HAProxy is version 2.3:

```
$ ./haproxy -v
HA-Proxy version 2.3.7 2021/03/16 - https://haproxy.org/
Status: stable branch - will stop receiving fixes around Q1 2022.
Known bugs: http://www.haproxy.org/bugs/bugs-2.3.7.html
Running on: Linux 5.4.0-1038-aws #40-Ubuntu SMP Fri Feb 5 23:53:34 UTC 2021 aarch64
```

view raw

Its configuration file, test.cfg, is based on the dpbench project’s [basic forwarder example](#):

```
defaults
    mode http
    timeout client 60s
    timeout server 60s
    timeout connect 1s

listen px
    bind :8000
```

view raw


```
balance random
server s1 172.31.36.194:8000
```

We run it with the following commands:

```
$ ulimit -n 100000
$ taskset -c 2-47 haproxy -D -f test.cfg
```

view raw

The client software, shown as *Runner* in the diagram, was [h1load](#) with 46 threads, which provides live stats that allow you to validate that everything went as expected. The number of concurrent connections was set to 1150 because this is an integral multiple of 46, which matches the number of threads/processes along the chain. The HTTP response for these tests was configured to be a zero-byte body. This can be configured by changing the URL to specify the size of the response required by passing `/?s=<size>`.

```
$ ulimit -n 100000
$ taskset -c 2-47 h1load -e -ll -P -t 46 -s 30 -d 120 -c 1150 http://172.31.37.79:8000
```

view raw

A second client acted as the control for the experiment. As shown as *Ref* in the diagram, it ran on the same client machine on a dedicated core at a very low load of 1000 requests per second. Its purpose is to measure the direct communication between the client and the server, measuring the impact of the load on the network stacks all along the chain, to more accurately measure the cost of the load balancer's traversal. It will not measure the part experienced by the client software itself, but it already fixes a bottom value.

```
$ ulimit -n 100000
$ taskset -c 1 h1load -e -ll -P -t 1 -s 30 -d 120 -R 1000 -c 128 http://172.31.36.194
```

view raw

The tests were run for two minutes with a 30-second ramp-up. The ramp-up is not important for pure performance but is mandatory when collecting timing information because the operating systems on all machines take a lot of time to grow the structures used by the file descriptors and sockets. It's easy to see extremely long latencies on the client due to its own work. The ramp-up approach smooths this out and reduces the number of bad measurements on the client. Using stairs allows you to detect some unexpected boundaries too—typically a forgotten file-descriptor limitation (i.e. you forgot to run `ulimit`) that would cause sudden increases.

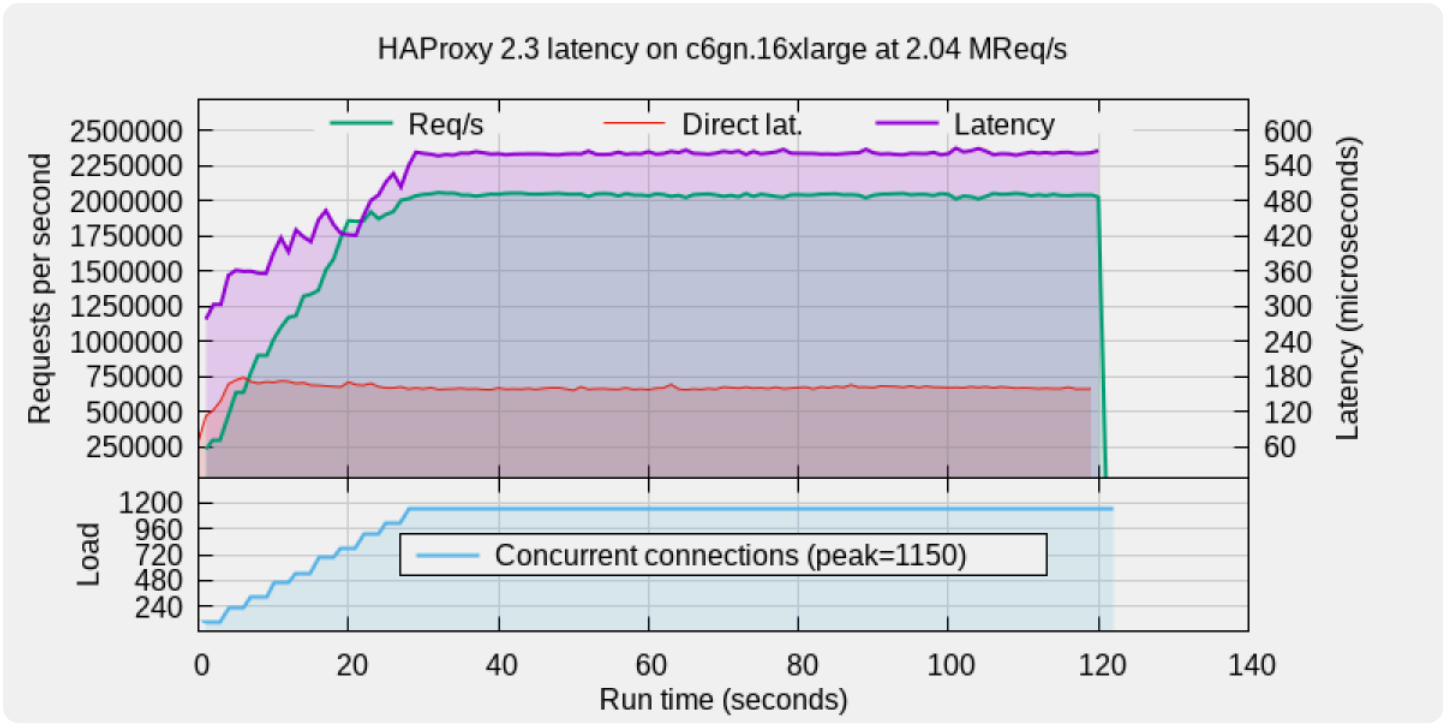
CPU usage and network traffic were collected as well on each machine, as documented in the [dpbench](#) project.

Results

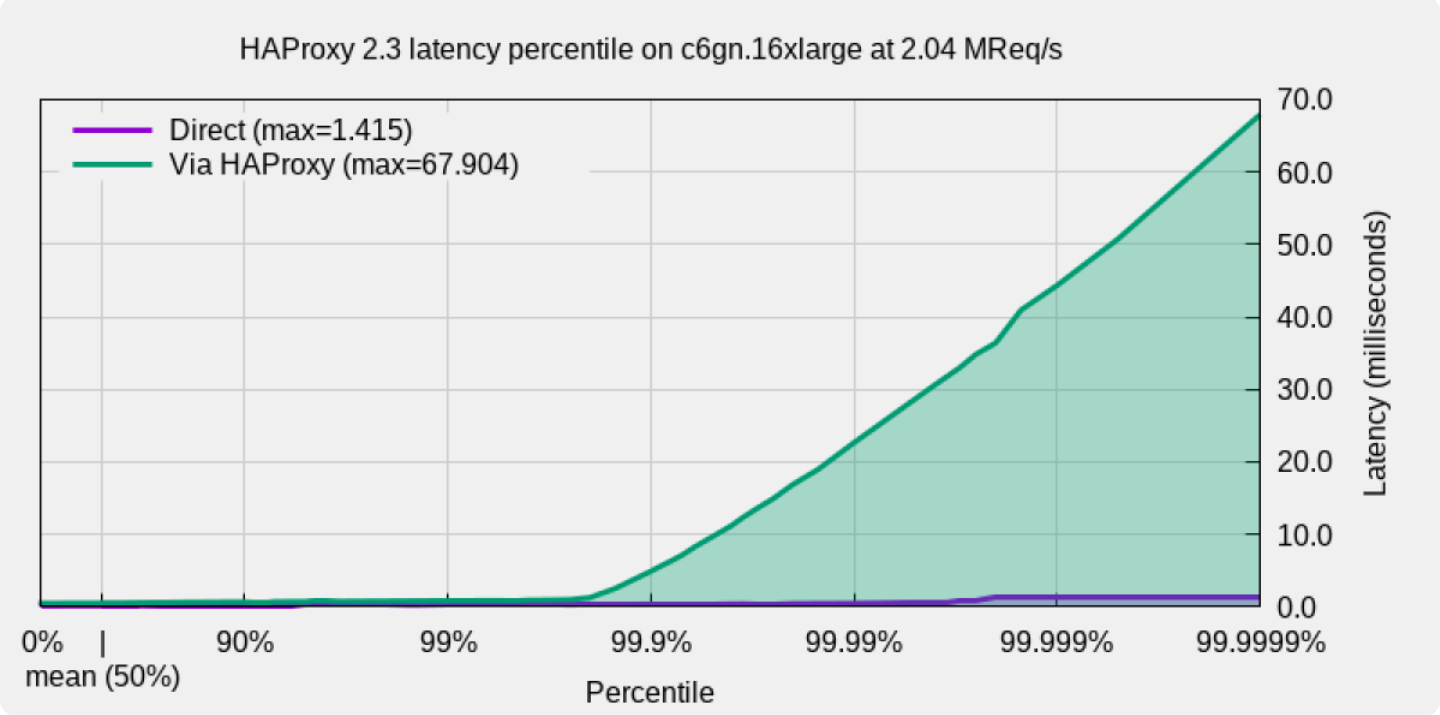
The results graphed below show curves that look quite clean and very stable, despite measurements being taken every second. With IRQs bound and the daemons above stopped on all the machines, almost all tests look as clean as this one.

HAProxy 2.3

Performance is almost flat at 2.04 to 2.05 million requests per second. The direct communication from the client to the server shows a 160 microseconds response time while adding HAProxy increases it to 560. So, the insertion of the load balancer, and the effect of adding an extra network hop, add at most 400 microseconds on average—less than half a millisecond.

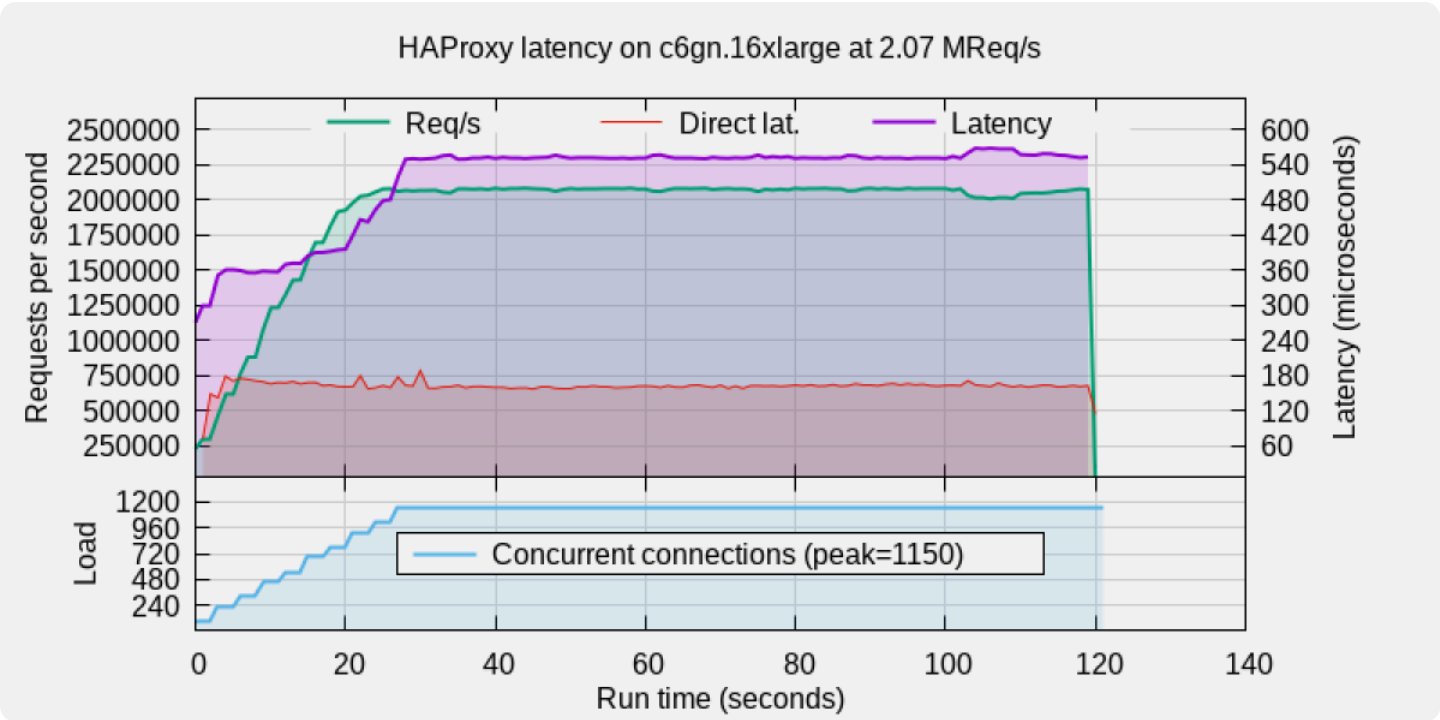


The percentile graph shows that the latency at 99.9% is at 5ms, indicating that 1 in 1,000 of the requests takes more than 5ms. However, it continues to grow and reaches 68ms for 1 request per million. The fact is that the machine is totally saturated and due to this we're measuring the effect of queuing at every stage in the chain (network buffers and HAProxy).

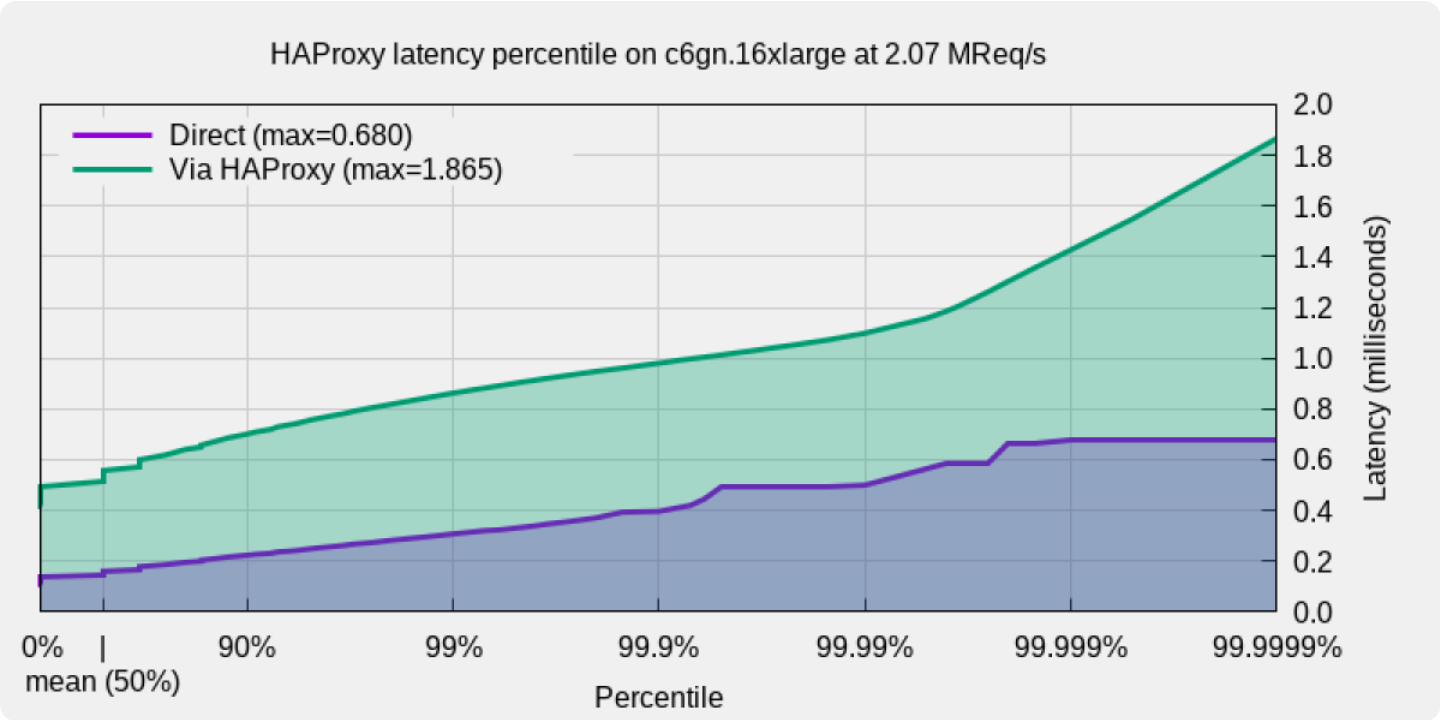


HAProxy 2.4

These results are for HAProxy version 2.3. The 2.4 development branch has received a lot of improvements over time and running the same tests on it gave even better results. HAProxy 2.4 oscillates between 2.07 and 2.08 million requests per second. The direct communication from the client to the server shows 163 microseconds response time while adding HAProxy, and the extra network hop increases it to 552. Therefore, it adds up to 389 microseconds on average, or slightly more than a third of a millisecond.



The maximum latency here reaches 1.87 milliseconds, out of which 0.68 are already experienced in direct communication. So, HAProxy never adds more than 1.19 milliseconds.



What is visible on the CPU usage report shown below is that HAProxy is not saturated anymore. It uses on average 42 of the 46 allocated CPUs. However, the network layer is visible under it and the *ksoftirq* threads regularly pop at 100% CPU.

top - 05:43:31 up 29 min, 4 users, load average: 16.62, 12.76, 8.62

Tasks: 639 total, 2 running, 637 sleeping, 0 stopped, 0 zombie

%Cpu(s): 44.5 us, 33.2 sy, 0.0 ni, 22.2 id, 0.0 wa, 0.0 hi, 0.1 si, 0.0 st

MiB Mem : 126544.6 total, 125538.5 free, 568.9 used, 437.1 buff/cache

MiB Swap: 0.0 total, 0.0 free, 0.0 used. 124966.6 avail Mem

view raw

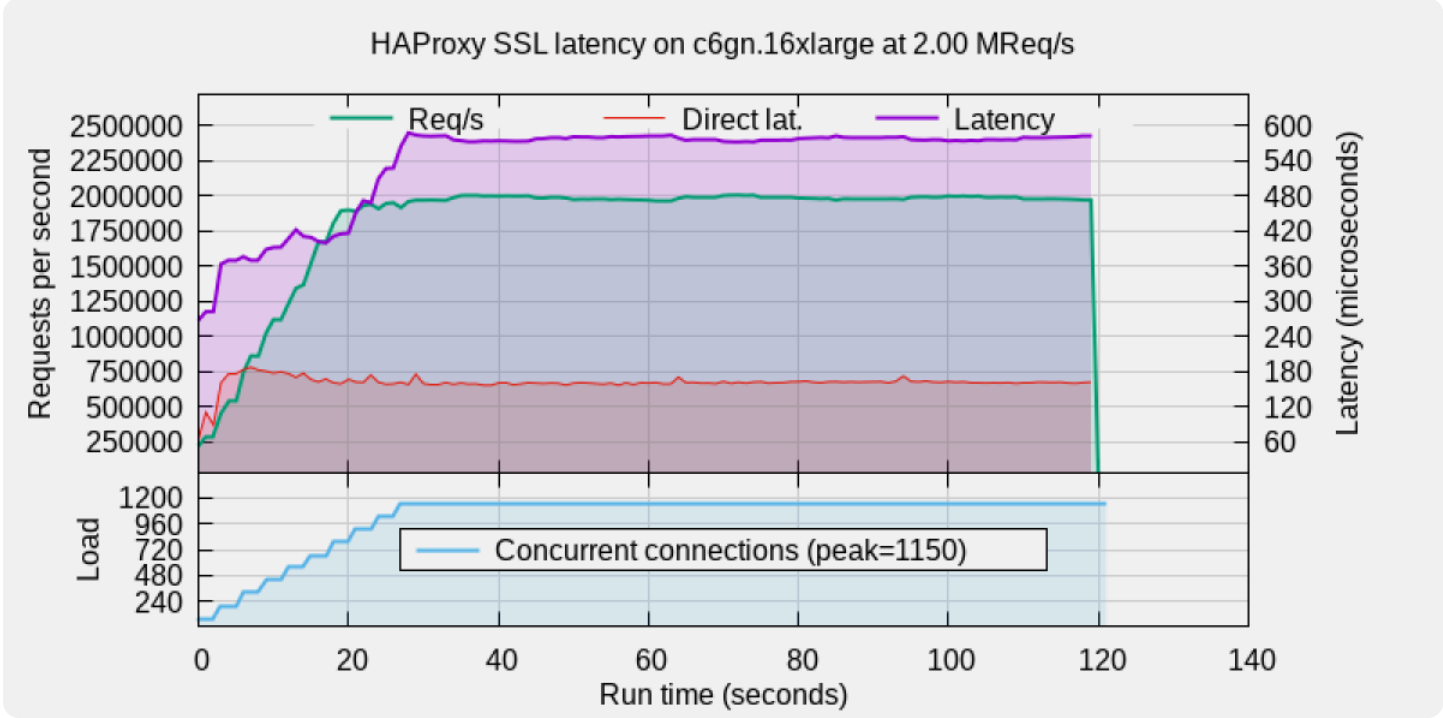
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2044	ubuntu	20	0	3356916	27312	6260	R	4194	0.0	20:09.40	haproxy+
336	root	20	0	0	0	0	S	8.9	0.0	0:24.27	ksoftir+
355	root	20	0	0	0	0	S	5.9	0.0	0:23.12	ksoftir+
361	root	20	0	0	0	0	S	5.9	0.0	0:19.64	ksoftir+
409	root	20	0	0	0	0	S	5.9	0.0	0:23.90	ksoftir+
373	root	20	0	0	0	0	S	5.0	0.0	0:22.96	ksoftir+
379	root	20	0	0	0	0	S	5.0	0.0	0:18.61	ksoftir+
391	root	20	0	0	0	0	S	5.0	0.0	0:22.93	ksoftir+
415	root	20	0	0	0	0	S	5.0	0.0	0:17.42	ksoftir+
385	root	20	0	0	0	0	S	4.0	0.0	0:18.74	ksoftir+
397	root	20	0	0	0	0	S	4.0	0.0	0:15.33	ksoftir+
432	root	20	0	0	0	0	S	4.0	0.0	0:15.12	ksoftir+
450	root	20	0	0	0	0	S	4.0	0.0	0:17.04	ksoftir+
367	root	20	0	0	0	0	S	3.0	0.0	0:16.48	ksoftir+
403	root	20	0	0	0	0	S	3.0	0.0	0:17.66	ksoftir+
444	root	20	0	0	0	0	S	3.0	0.0	0:12.70	ksoftir+

This is an ideal combination of resources, with HAProxy having exactly what it needs and the network stack having exactly what it needs as well. It's not possible to go beyond this because we've reached the limit of 4.15 million packets per second in both directions that we previously measured as the hard limit on the network interface.

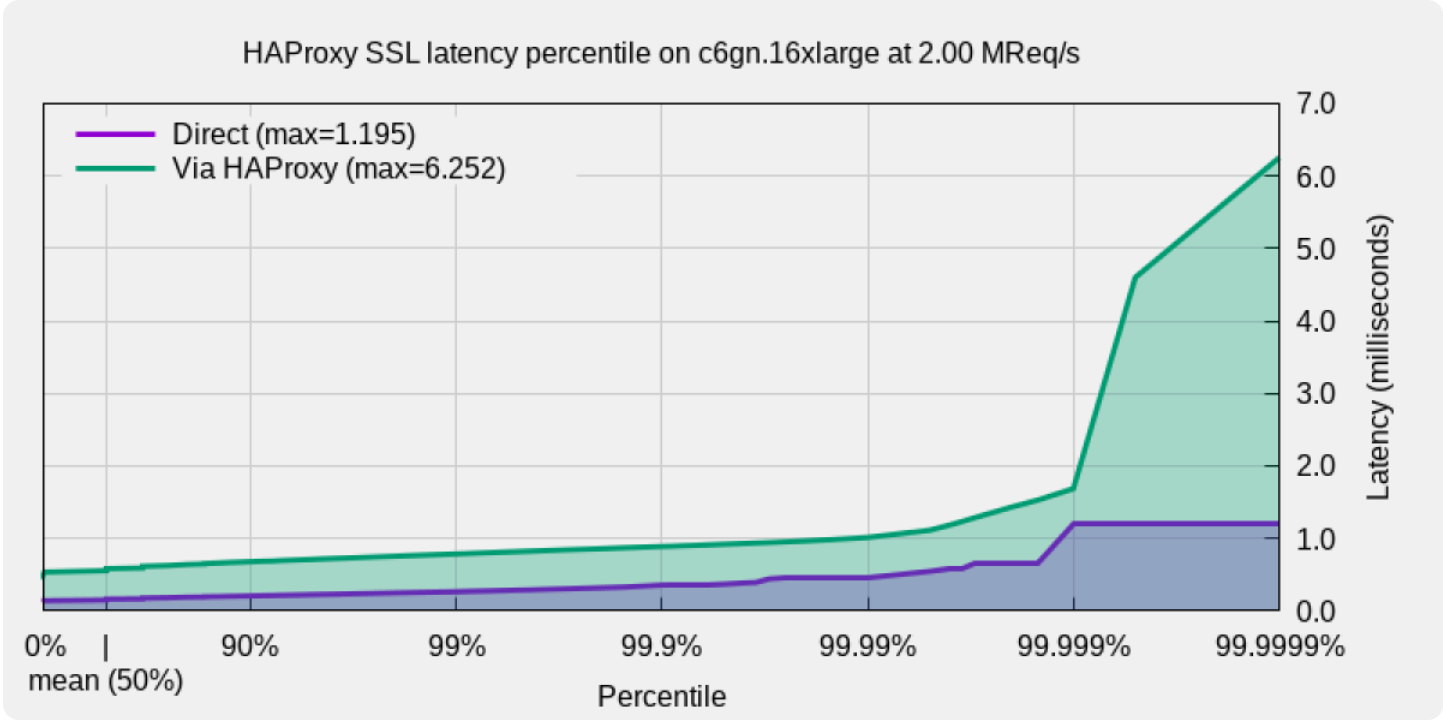
TLS for Free

Some readers are probably wondering if we couldn't use these spare CPU cycles in HAProxy to do a bit of [TLS termination](#) for free. Let's try it.

Results show that we're just between 1.99 and 2.01 million requests per second over TLS. The average response time is 575 microseconds for 162 direct, so HAProxy over TLS adds at most 413 microseconds to the chain. For these tests, I used an RSA 2048-bit certificate and the connections were made over TLSv1.3 with TLS_AES_256_GCM_SHA384.



The latency percentiles show that less than 1 request in 100,000 is above 1.6 milliseconds and that they never exceed 6.25 milliseconds, for 1.2 max direct. HAProxy never adds more than 5 milliseconds when terminating TLS.



Now we've saturated the CPUs:

```
top - 06:42:08 up 1:28, 4 users, load average: 40.05, 20.99, 12.57
Tasks: 639 total, 2 running, 637 sleeping, 0 stopped, 0 zombie
%Cpu(s): 54.9 us, 31.3 sy, 0.0 ni, 13.7 id, 0.0 wa, 0.0 hi, 0.1 si, 0.0 st
MiB Mem : 126544.6 total, 125279.7 free, 626.9 used, 638.0 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used. 124893.0 avail Mem
```

view raw

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
5433	ubuntu	20	0	3357232	39948	6428	R	4597	0.0	260:36.56	haproxy
373	root	20	0	0	0	0	S	8.9	0.0	1:32.24	ksoftir+
391	root	20	0	0	0	0	S	6.9	0.0	1:29.67	ksoftir+
409	root	20	0	0	0	0	S	5.9	0.0	1:26.27	ksoftir+
379	root	20	0	0	0	0	S	5.0	0.0	1:34.07	ksoftir+
415	root	20	0	0	0	0	S	5.0	0.0	1:20.57	ksoftir+
438	root	20	0	0	0	0	S	5.0	0.0	1:16.82	ksoftir+
355	root	20	0	0	0	0	S	4.0	0.0	1:27.25	ksoftir+
361	root	20	0	0	0	0	S	4.0	0.0	1:24.02	ksoftir+
385	root	20	0	0	0	0	S	4.0	0.0	1:19.53	ksoftir+
432	root	20	0	0	0	0	S	4.0	0.0	1:08.21	ksoftir+
450	root	20	0	0	0	0	S	4.0	0.0	1:13.43	ksoftir+
336	root	20	0	0	0	0	S	3.0	0.0	1:31.32	ksoftir+
367	root	20	0	0	0	0	S	3.0	0.0	1:20.86	ksoftir+
403	root	20	0	0	0	0	S	3.0	0.0	1:12.48	ksoftir+
397	root	20	0	0	0	0	S	2.0	0.0	1:10.06	ksoftir+

Analysis

Why assign so many cores if the network is the limitation? Just to show that it's possible. Also, some users might want to run heavy processing in HAProxy with lots of rules, multi-threaded Lua modules (2.4 only), or heavy TLS traffic. In addition, figuring out how many cores are needed for the network part to handle a given load is extremely useful, because once these

cores are considered “reserved”, it is easier to assign the remaining ones to various tasks (which was done for the load generators by the way).

But, quite frankly, nobody needs such insane performance levels on a single machine. It’s possible to downscale this setup considerably to very inexpensive machines and still get excellent response times.

For much lower loads, such as 100 concurrent connections—while still high for many users—even a free *t4g.micro* instance, which comes with 2 vCPUs and 1 GB of RAM, reaches 102,000 requests per second for non-TLS traffic and 75,000 for requests for TLS. Although, this modest machine instance drops to about 56,000 non-TLS and 45,000 TLS at 1,000 concurrent connections.

However, the 2 vCPUs need to be shared between HAProxy, the network stack, and the tools, which causes visible latency spikes at 99.9999% above 25,000 TLS requests per second. But we’re talking about a free instance and high loads already! For even lower loads it is also possible to install HAProxy for free on any existing machine you have without even noticing its presence!

Conclusion

With the Arm-based AWS Graviton2 instances, it’s possible to have both computational *and* network performance. The Graviton2 machines are absolutely awesome with their 64 true cores and their impressive scalability. Their support of the LSE atomic instructions has totally solved the scalability issues that were plaguing the pre-armv8.1-a CPUs. I don’t understand why we’re still not seeing boards with such CPUs. They’re fantastic for developers, they provide totally uniform memory access to all 64 cores, and they’ve already helped us leverage some contention points on HAProxy 2.4. Even just a scaled-down one with 16 cores and lower frequency would be great... Please 😊

It is also clearly visible that there’s still a little bit of headroom, but that we’ve hit the 4-million packets-per-second limit of the current virtualization layer. However, reaching 100 Gbps is trivial there. I could verify that HAProxy reaches 92 Gbps of HTTP payload bandwidth with requests as small as 30 kB! I’m really wondering what such machines could do on bare metal, but most likely we’re observing one of the most inexpensive bit-movers nowadays! I wouldn’t be surprised if Amazon was already using this hardware for its own infrastructure.

On the HAProxy front, the current 2.4 development branch received a lot of scalability optimizations recently that managed to further reduce the peak latencies and increase the fairness between requests. We went as far as measuring both the per-function CPU time and per-function latency at runtime to spot offenders and focus our efforts on certain areas in the code. The Runtime API command `show profiling displays` these metrics for us:


> show profiling [view raw](#)

Tasks activity:

function	calls	cpu_tot	cpu_avg	lat_tot	lat_avg
h1_io_cb	3667960	9.089s	2.477us	12.88s	3.510us
si_cs_io_cb	3487273	4.971s	1.425us	12.44m	214.0us
process_stream	2330329	10.20s	4.378us	21.52m	554.1us
ssl_sock_io_cb	2134538	2.612m	73.43us	6.315h	10.65ms
h1_timeout_task	1553472	-	-	6.881m	265.8us
accept_queue_process	320615	13.72m	2.568ms	8.094m	1.515ms
task_run_applet	58	3.897ms	67.19us	3.658ms	63.07us
session_expire_embryonic	1	-	-	294.0ns	294.0ns

That command is part of the long-term instrumentation effort that allows developers to continuously verify that they’re not degrading anything in the codebase. It’s also very useful to understand what could be a likely cause of suboptimal processing in the field. The “perf” tool was used a lot on these machines, and a few small contention points were identified which, once addressed, could allow us to gain a few more percentage points.

We are approaching an era where not only can you get the world’s fastest load balancer for free, but, with the ability to run it on cheap machine instances and still get amazing results, its operating cost is pushed to nearly zero too.



Subscribe to our blog.
Get the latest release updates, tutorials, and deep-dives from HAProxy experts.

Tags: [Benchmarking](#)

Authors



Willy Tarreau

Willy released the first version of HAProxy in 2001, welcomed the first contribution in 2004, and became a Linux kernel maintainer in 2006. His focus has always been on the lower layers where efficiency can still be improved after everything was squeezed at visible layers, and on reliability, probably because he hates revisiting complex code.

[GitHub](#)

Related Posts

August 25th, 2012

How to Use HAProxy & Varnish Together on a Single Domain Name

In this blog post, we'll explain how to use both HAProxy and Varnish on a web application hosted on a single domain name.



January 2nd, 2014

HAProxy Advanced Redis Health Check

In this blog post, we demonstrate how to build a simple Redis infrastructure thanks to the HAProxy advanced send/expect health checks feature.



February 15th, 2019

Test Driving “Power of Two Random Choices” Load Balancing

The Power of Two Random Choices load-balancing algorithm made us curious. See how it stacks up against other modern-day algorithms available in HAProxy.



February 15th, 2015

A HTTP Monitor Which Matches Multiple Conditions in HAProxy

Health checking is one of the most important features of a load balancer. Here we show how to match multiple conditions for HTTP health checking in HAProxy.



Products

- HAProxy One
- HAProxy Enterprise
- HAProxy Fusion Control Plane
- HAProxy Edge
- HAProxy Enterprise Kubernetes Ingress Controller
- HAProxy ALOHA

Solutions

- Load balancing
- UDP load balancing
- API gateway
- AI gateway
- High availability
- Security
- SSL/TLS processing
- DDoS protection and rate limiting
- Bot management
- Web application firewall
- Kubernetes
- Automation and self-service
- HAProxy GUI
- Application acceleration
- Public sector

Resources

- HAProxy Enterprise documentation
- HAProxy ALOHA documentation
- HAProxy Kubernetes Ingress Controller documentation
- Compare Community with Enterprise
- Certified integrations
- User spotlight series
- Content library
- Blog
- Success stories

Partners

- Partner program
- Certified integration program
- Find a partner
- Partner deal registration

Support

- Customer support portal
- Support options
- Professional services
- Community mailing list

Company

- About us
- Contact us
- Events
- Careers
- News

+1 (844) 222-4340

