Solutions Benchmarking

HAProxy Forwards Over 2 Million HTTP Requests per Second on a Single Arm-based AWS Graviton2 Instance HAProxy 在单个基于 Arm 的 AWS Graviton2 实例上每秒转发超过 200 万个 HTTP 请求

April 8th, 2021 16 min read

Willy Tarreau 威利·塔罗

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

软件负载均衡器在单个 Arm 实例上的吞吐量首次超过 200 万 RPS。

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!

几周前，当我在处理与线程锁定争用相关的 HAProxy 问题时，我发现自己在一台具有 8 核、16 线程 Intel Xeon W2145 处理器的服务器上运行了一些测试，该处理器是我们实验室中的处理器。虽然我的意图不是对代理进行基准测试，但我观察到 HAProxy 达到了每秒 103 万个 HTTP 请求。我突然想起了我告诉周围人的所有时间，"当我们跨越每秒百万个请求的障碍时，我会写下来。所以，我必须信守诺言！

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:

我想看看它如何在更多内核上扩展。我访问了一些基于 Arm 的新 AWS Graviton2 实例，这些实例提供多达 64 个内核。为了让您了解它们的设计，每个内核都使用自己的 L2 缓存，并且所有内核共享一个 L3 缓存。如果您在其中一台计算机上运行 `lscpu`，您可以自己看到这一点，它将显示内核数量和缓存的共享方式：

```
$ lscpu -e

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
```

view raw

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 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.

他们给我留下了深刻的印象，尤其是当我们受到 @AGSaidi 的鼓励切换到新的 Arm 大型系统扩展（LSE） 原子指令时，我们已经有一些可用的代码，但从未进行过如此大规模的测试，并且完全释放了这些机器的真正力量。所以这看起来是一个绝佳的机会，可以将所有内容结合起来，将我们的 HAProxy 基准测试推向一个新的水平！如果要使用 gcc 9.3.0 编译 HAProxy，请包含标志 `-march=armv8.1-a` 以启用 LSE 原子指令。在 gcc 版本 10.2.0 中，LSE 默认处于启用状态。

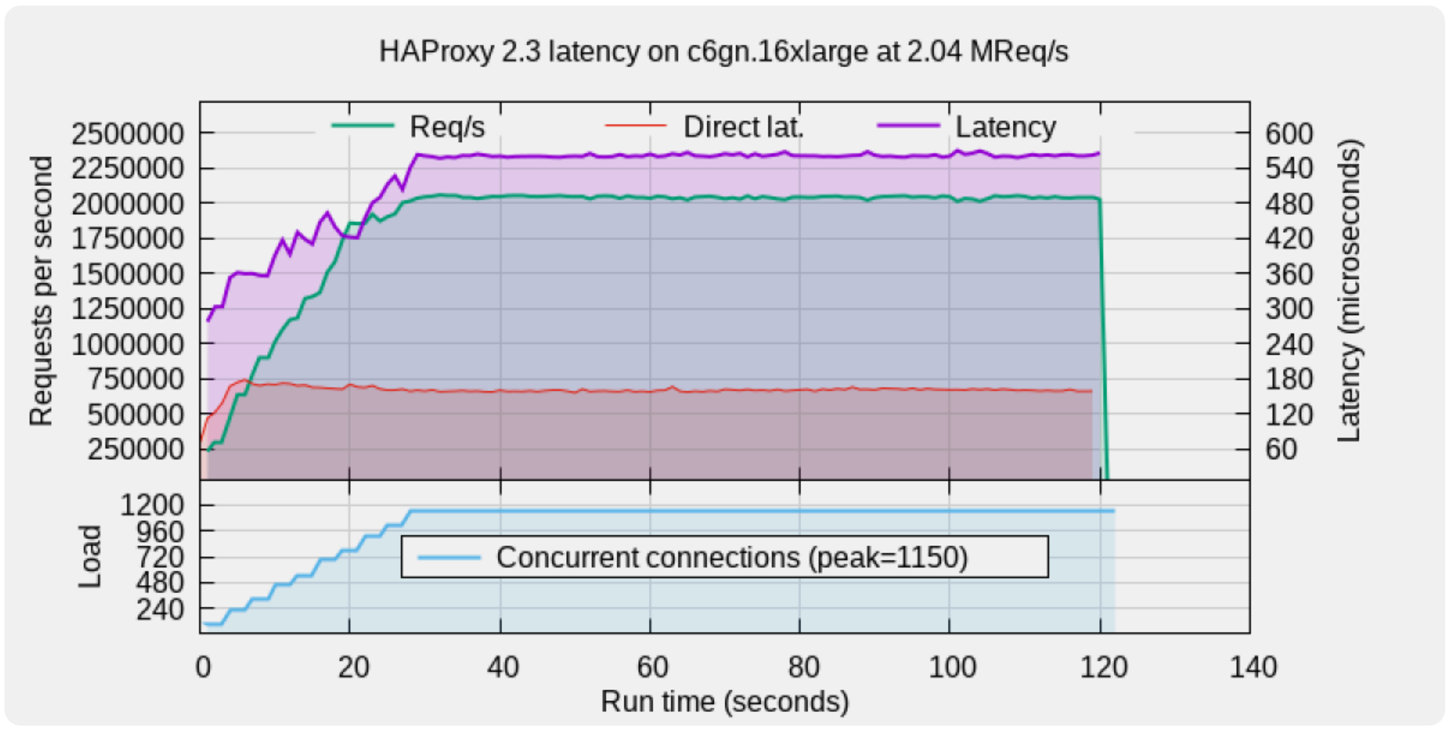
Synopsis 概要

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!

是的，您没看错这篇博文的标题。在基于 Arm 的 AWS Graviton2 实例上测试时，HAProxy 版本 2.3 达到每秒 204 万个请求！

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

HAProxy 2.4 仍在开发中，它超越了这个数字，达到了每秒 2.07 到 208 万个请求。



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

如果您对测试方法、系统的调整方式以及吸取的经验教训感到好奇，请继续阅读。

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.

与 NGINX 团队的一些成员合作启动了一个名为 dpbench 的新项目，该项目捕获了一些对代理进行基准测试的最佳实践。它为任何想要进行类似实验而不会成为一些常见误解的牺牲品的人提供指导。请注意，该项目欢迎可以提供更多调整/报告脚本、各种环境的配置、结果和提示的贡献者。

Methodology 方法论

In these benchmarks, we're testing two things:

在这些基准测试中，我们将测试两件事：

- HTTP requests per second 每秒 HTTP 请求数

- End-to-end request latency at and above the 99.99th percentile

  端到端请求延迟等于及以上 99.99 个百分位

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.

尾部延迟（影响少量请求的延迟）是不容忽视的极其重要的事情，尤其是在服务网格中，用户的一个请求可能会导致后端出现许多微服务请求。在这些环境中，延迟峰值会产生放大因子。

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.

例如，如果用户的请求转换为 20 个后端请求，则仅影响 10000 个后端请求的较大延迟将影响该用户请求的 500 个请求中的 1 个。如果该用户在一次访问期间发出 50 个请求，则更有可能出现。如果延迟的增加很低，或者非常罕见，那么问题并不严重，但是当它既高又频繁时，就是一个问题。因此，我们在基准测试期间衡量这一点非常重要。

Instance size 实例大小

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.

我们使用了 AWS 的 *c6gn.16xlarge* 虚拟机实例，这是 c6gn 系列的 64 核计算机，可访问 100Gbps 的网络带宽。它们使用 Graviton2 处理器进行计算优化，专为需要高网络带宽的应用程序而构建。此实例大小用于客户端、代理和服务器。

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.

我首先从一个 16 核开始，然后是一个 32 核实例，认为这可能就足够了。从客户端到服务器的直接测试，中间没有 HAProxy，我有点失望，每秒只有 800k 个数据包，或者比普通的 *c6g* 数据包少一点。然后，我决定选择完整的 1 个：专用主机上的 64 个内核。这一次，我得到了每秒 415 万个数据包，并认为这已经足够了，因为我真的只需要 200 万个数据包就可以实现每秒 100 万个请求，其中 1 个数据包用于请求，1 个数据包用于响应。

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

所有计算机都安装了 Ubuntu 20.04，这是默认建议的安装之一。

IRQ affinity IRQ 亲和性

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.

我花了一段时间才弄清楚如何完全稳定平台，因为在虚拟化时，仍有 32 个中断（又名 IRQ）分配给网络队列，交付到 32 个内核！这可能解释了内核数量较少时性能较低的原因。32 个队列对一些读者来说听起来很多，但我们在这里谈论的是 100 Gbps 网络和每秒数百万个数据包。在这样的数据包速率下，您绝对不希望任何用户空间进程在测试过程中命中这些内核之一，这就是发生的情况，因为这些内核或多或少是随机分配给进程的。

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`).

将中断移至 32 个较高核心，使 32 个较低核心未使用，并大大简化了设置。但是，我想知道网络堆栈是否支持仅在 16 个内核上运行，每个内核有 2 个中断，这将为 HAProxy 和系统的其余部分留下 48 个内核。我尝试了一下，发现这是最佳情况：网络在每个方向上以每秒约 415 万个数据包的速度饱和，网络专用核心经常以 100% 的速度出现，这表明中断被迫排队 （`ksoftirqd`）。

At this level, whenever something happens on the system—for example, the *snapd* 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:

在这个级别上，每当系统上发生某些事情时（例如，*snapd* 守护程序每 60 秒唤醒一次，*crond* 每 60 秒检查一次其文件），它必须与网络或 HAProxy 共享一个内核，这会导致非常明显的 1 毫秒或更长时间的延迟峰值。我更喜欢停止这些服务，以限制测试期间后台随机发生的风险：

view raw

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

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.

这绝对是寄生活动的一个很好的例子，证明为剩余的活动留下一些可用的内核是合理的。因此，我想我应该保留 2 个可用的内核，并将每个与负载无关的活动绑定到那里。

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).

简而言之，测试运行时，网络中断绑定到所有机器（总共 16 个内核）上的核心 48-63，在核心 2-47（总共 46 个核心）上启用 HAProxy/client/server，内核 0-1 留给其他所有设备（ `sshd` 和监控工具）。

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

使用前面提到的 *dpbench* 项目中的 *set-irq.sh* 工具，通过此命令将 IRQ 绑定到 16 个上层内核：

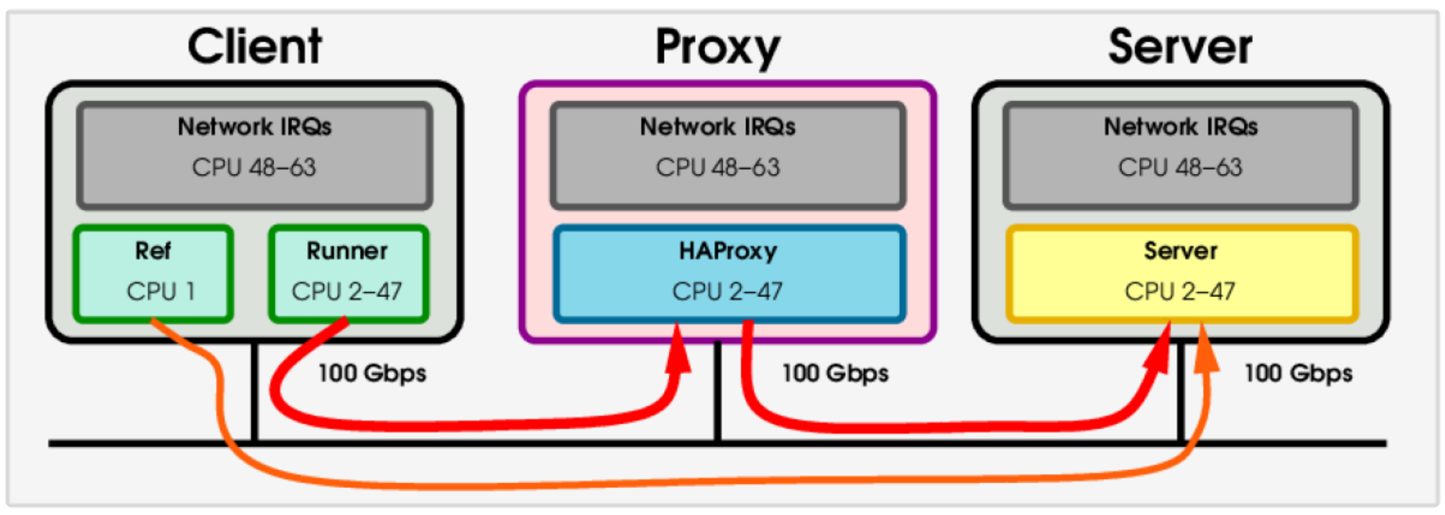
view raw

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

Network topology 网络拓扑

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

整个设置保持相当简单，如下所示：



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:

服务器软件是 httpterm，它从 46 个内核开始。 `ulimit` 命令增加了可用的打开文件描述符（即连接槽）的数量：

view raw

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

HAProxy is version 2.3: HAProxy 是 2.3 版：

view raw

```
$ ./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
```

Its configuration file, test.cfg, is based on the dpbench project's basic forwarder example:

它的配置文件 test.cfg 基于 dpbench 项目的基本转发器示例：

view raw

```
defaults
   mode http
   timeout client 60s
```

```
    timeout server 60s
    timeout connect 1s

listen px
    bind :8000
    balance random
    server s1 172.31.36.194:8000
```

We run it with the following commands:

我们使用以下命令运行它：

view raw

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

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>*.

客户端软件（在图中显示为 *Runner*）是具有 46 个线程的 h1load，它提供实时统计信息，允许您验证一切是否按预期进行。并发连接数设置为 1150，因为这是 46 的整数倍，这与链上的线程/进程数相匹配。这些测试的 HTTP 响应被配置为零字节正文。这可以通过更改 URL 来配置，以指定通过传递 */? s=<size>* 所需的响应大小。

view raw

```
$ ulimit -n 100000
$ taskset -c 2-47 h1load -e -ll -P -t 46 -s 30 -d 120 -c 1150 http://172.31.37.79:800
```

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.

第二个客户端充当实验的对照。 如图所示，它以每秒 1000 个请求的极低负载在专用内核上的同一客户端计算机上运行。其目的是测量客户端和服务器之间的直接通信，测量负载对整个链上网络堆栈的影响，以更准确地测量负载均衡器遍历的成本。它不会测量客户端软件本身经历的部分，但它已经固定了一个底部值。

view raw

```
$ 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
```

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.

测试运行了 2 分钟，并增加了 30 秒。ramp-up 对于纯粹的性能并不重要，但在收集 timing information 时是必需的，因为所有计算机上的 os 都需要大量时间来增加文件 descriptors 和 sockets 使用的结构。由于客户端自己的工作，很容易在客户端上看到极长的延迟。渐进方法消除了这种情况，并减少了客户端上错误测量的数量。使用楼梯也可以检测一些意外的边界 — 通常是忘记的文件描述符限制（即您忘记运行 `ulimit`），这会导致突然增加。

CPU usage and network traffic were collected as well on each machine, as documented in the dpbench project.

还收集了每台计算机上的 CPU 使用率和网络流量，如 dpbench 项目中所述。

Results 结果

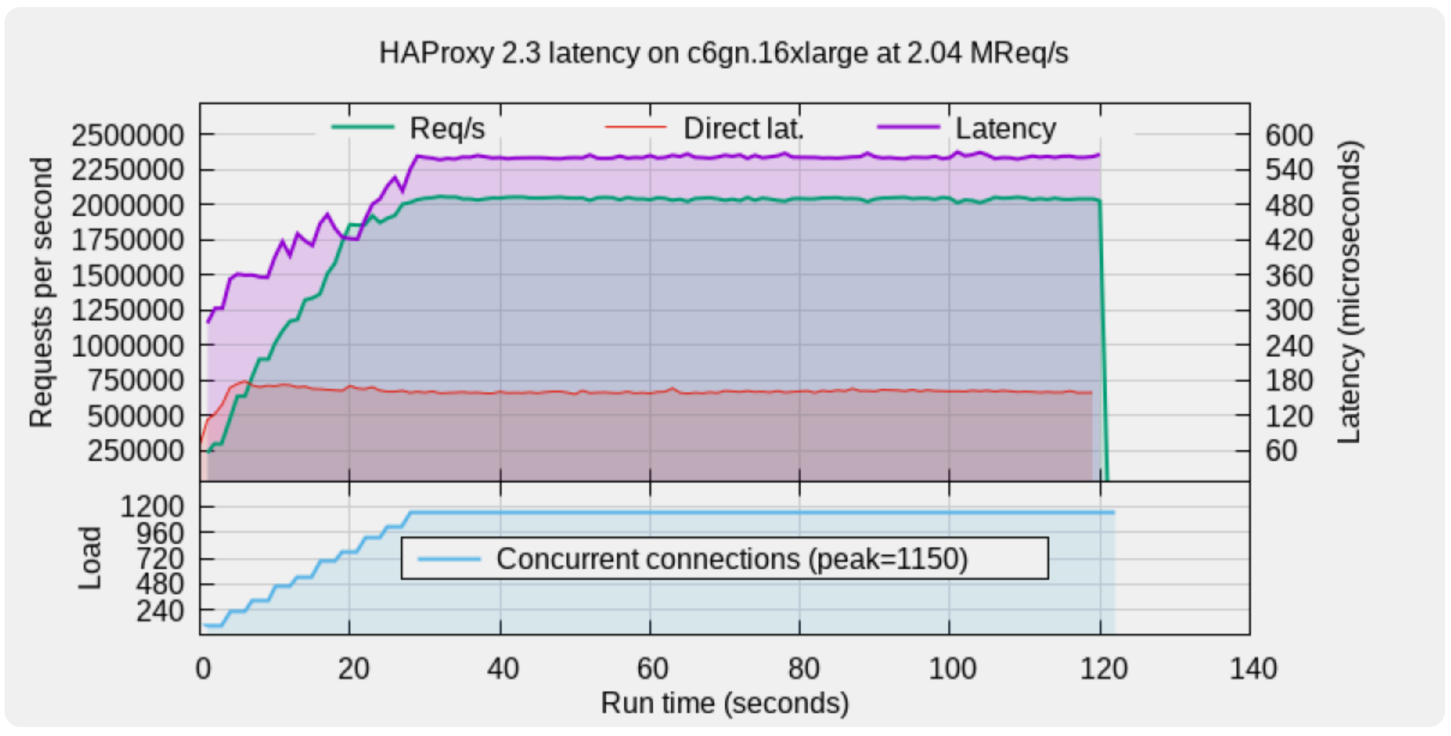
The results graphed below show curves that look quite clean and very stable, despite measurements being taken every second. With IRQs bound and the daemons above stopped on all the machines, almost all tests look as clean as this one.

下图显示的曲线看起来非常干净且非常稳定，尽管每秒进行一次测量。绑定了 IRQ 并在所有机器上停止了上面的守护进程后，几乎所有测试看起来都和这个一样干净。

HAProxy 2.3

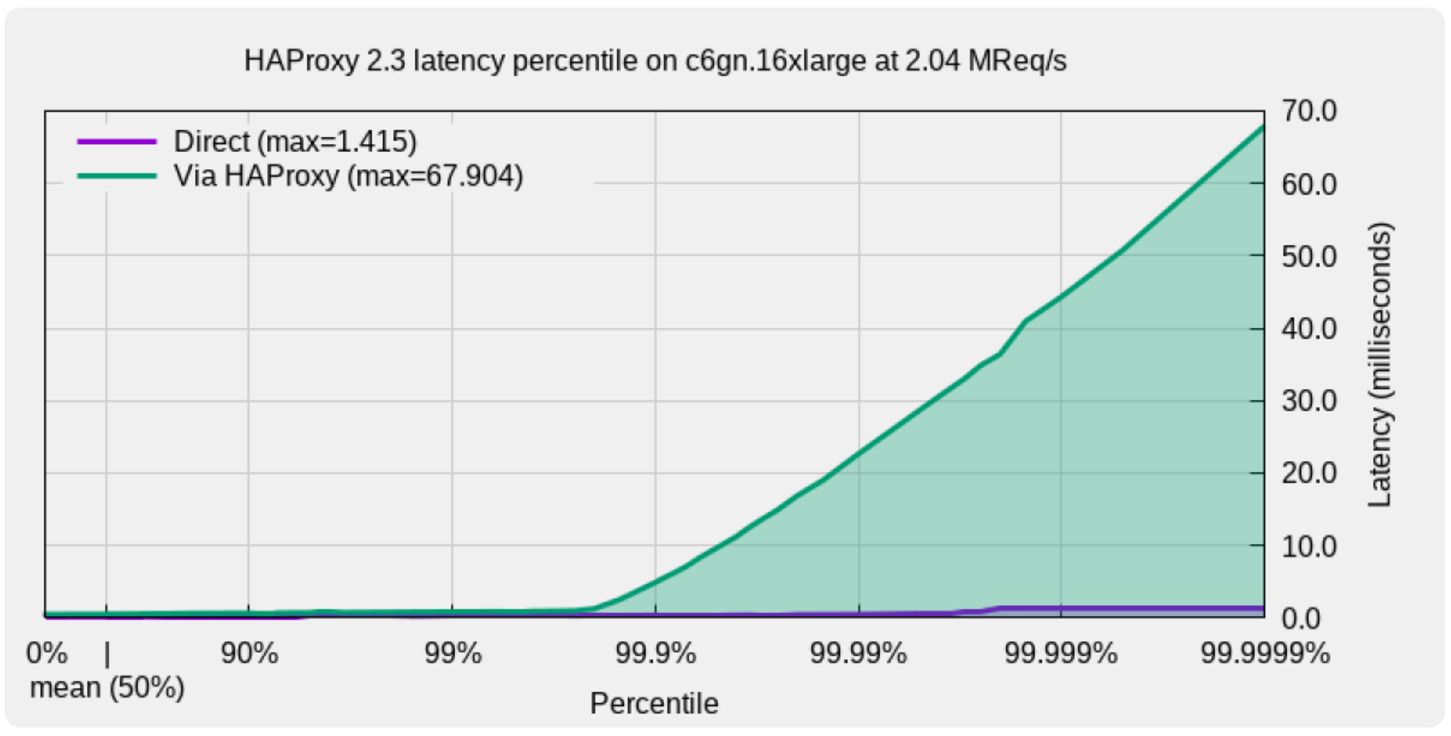
Performance is almost flat at 2.04 to 2.05 million requests per second. The direct communication from the client to the server shows a 160 microseconds response time while adding HAProxy increases it to 560. So, the insertion of the load balancer, and the effect of adding an extra network hop, add at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

性能几乎持平，每秒 2.04 到 205 万个请求。从客户端到服务器的直接通信显示 160 微秒的响应时间，而添加 HAProxy 则将其增加到 560 微秒。因此，负载均衡器的插入以及添加额外网络跃点的效果平均最多增加 400 微秒，不到半毫秒。



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).

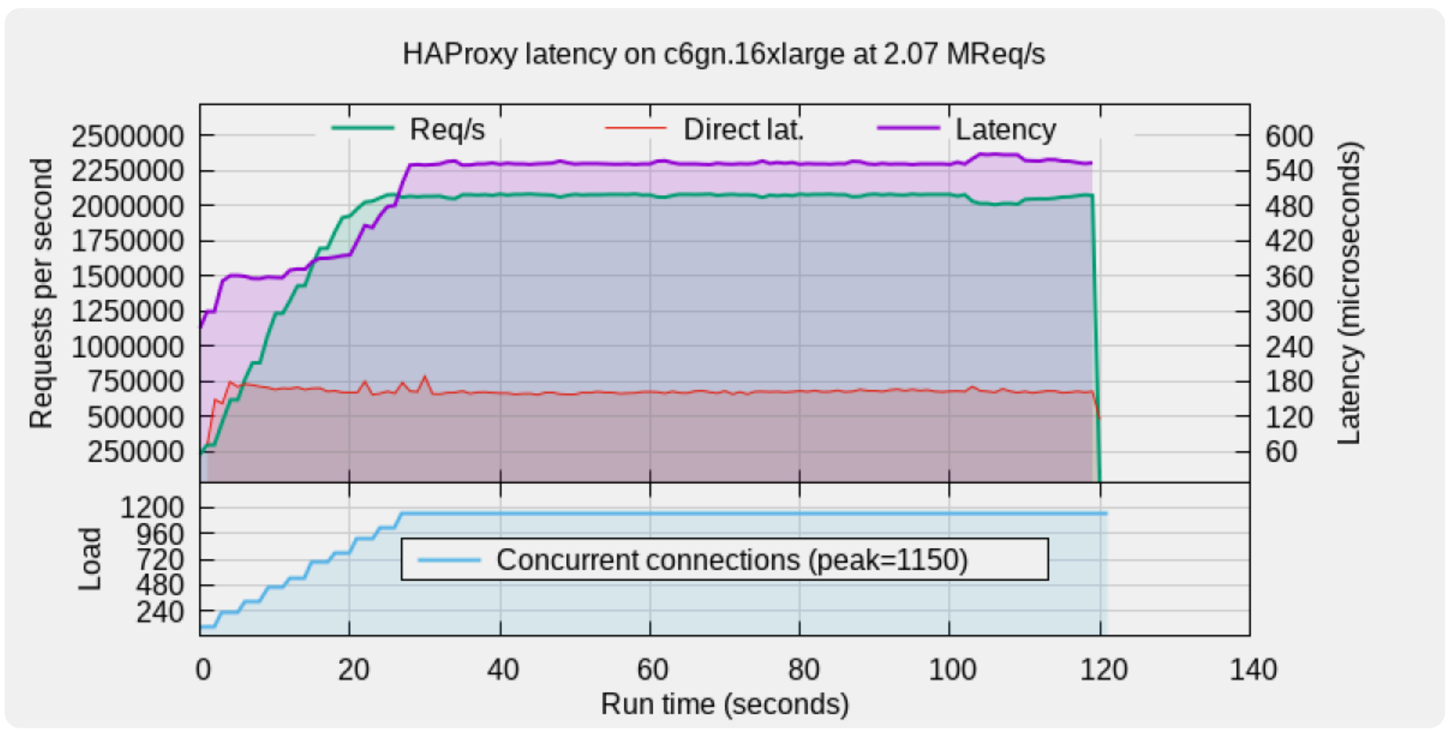
百分位数图显示 99.9% 的延迟为 5 毫秒，表示每 1000 个请求中就有 1 个请求花费的时间超过 5 毫秒。但是，它继续增长并达到 68 毫秒（每百万个请求 1 个）。事实是机器已经完全饱和，因此我们正在测量排队在链的每个阶段（网络缓冲区和 HAProxy）的影响。



HAProxy 2.4

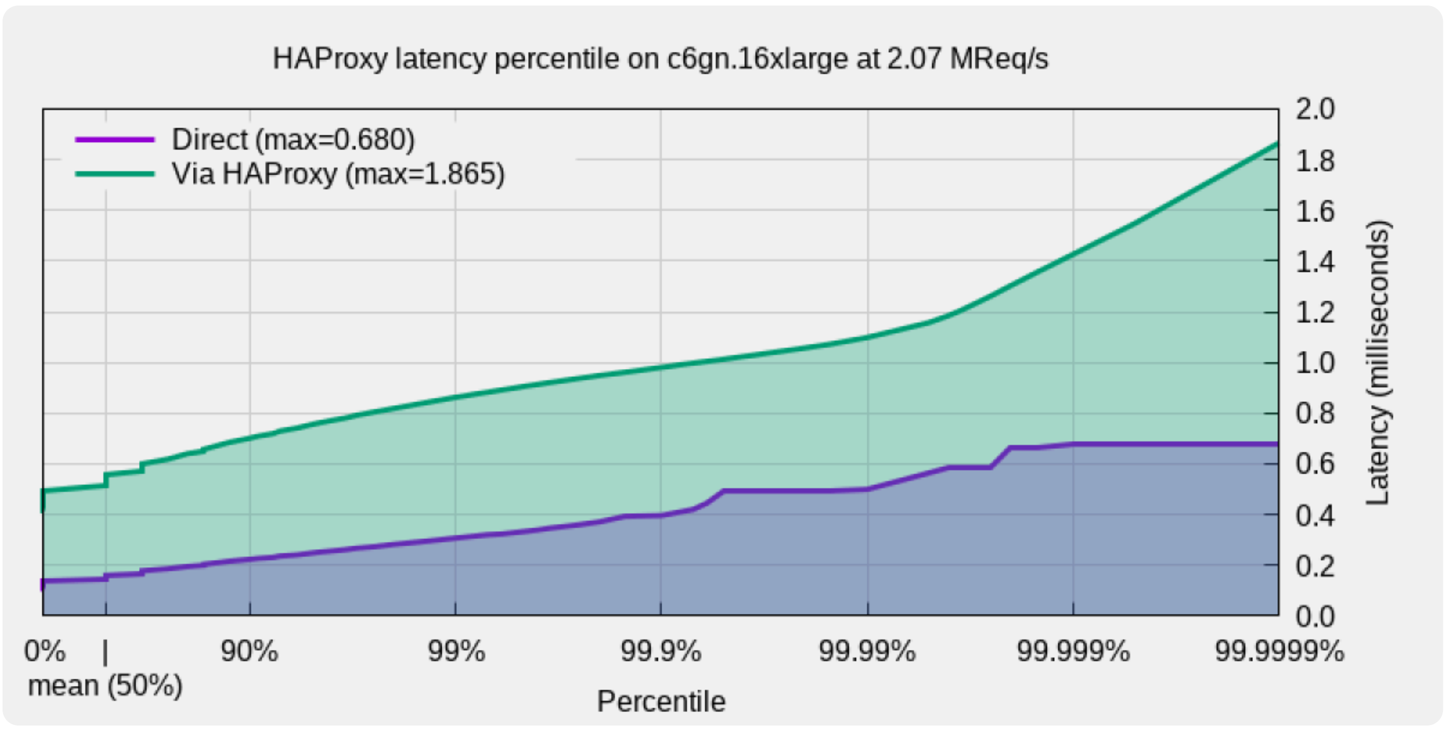
These results are for HAProxy version 2.3. The 2.4 development branch has received a lot of improvements over time and running the same tests on it gave even better results. HAProxy 2.4 oscillates between 2.07 and 2.08 million requests per second. The direct communication from the client to the server 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.

这些结果适用于 HAProxy 版本 2.3。随着时间的推移，2.4 开发分支得到了很多改进，在它上运行相同的测试可以得到更好的结果。HAProxy 2.4 每秒在 2.07 到 208 万个请求之间波动。在添加 HAProxy 时，从客户端到服务器的直接通信显示 163 微秒的响应时间，额外的网络跃点将其增加到 552 微秒。因此，它平均加起来为 389 微秒，或略高于三分之一毫秒。



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.

这里的最大延迟达到 1.87 毫秒，其中 0.68 毫秒在直接通信中已经经历过。因此，HAProxy 添加的时间永远不会超过 1.19 毫秒。



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.

在下面显示的 CPU 使用率报告中可以看到的是 HAProxy 不再饱和。它平均使用分配的 46 个 CPU 中的 42 个。但是，网络层在它下面可见，并且 *ksoftirq* 线程经常以 100% CPU 的速度弹出。

view raw

```
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

    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.

这是一个理想的资源组合，HAProxy 恰好有它需要的东西，而网络堆栈也恰好有它需要的东西。不可能超过这个范围，因为我们已经达到了每秒 415 万个数据包的双向限制，我们之前将其测量为网络接口的硬限制。

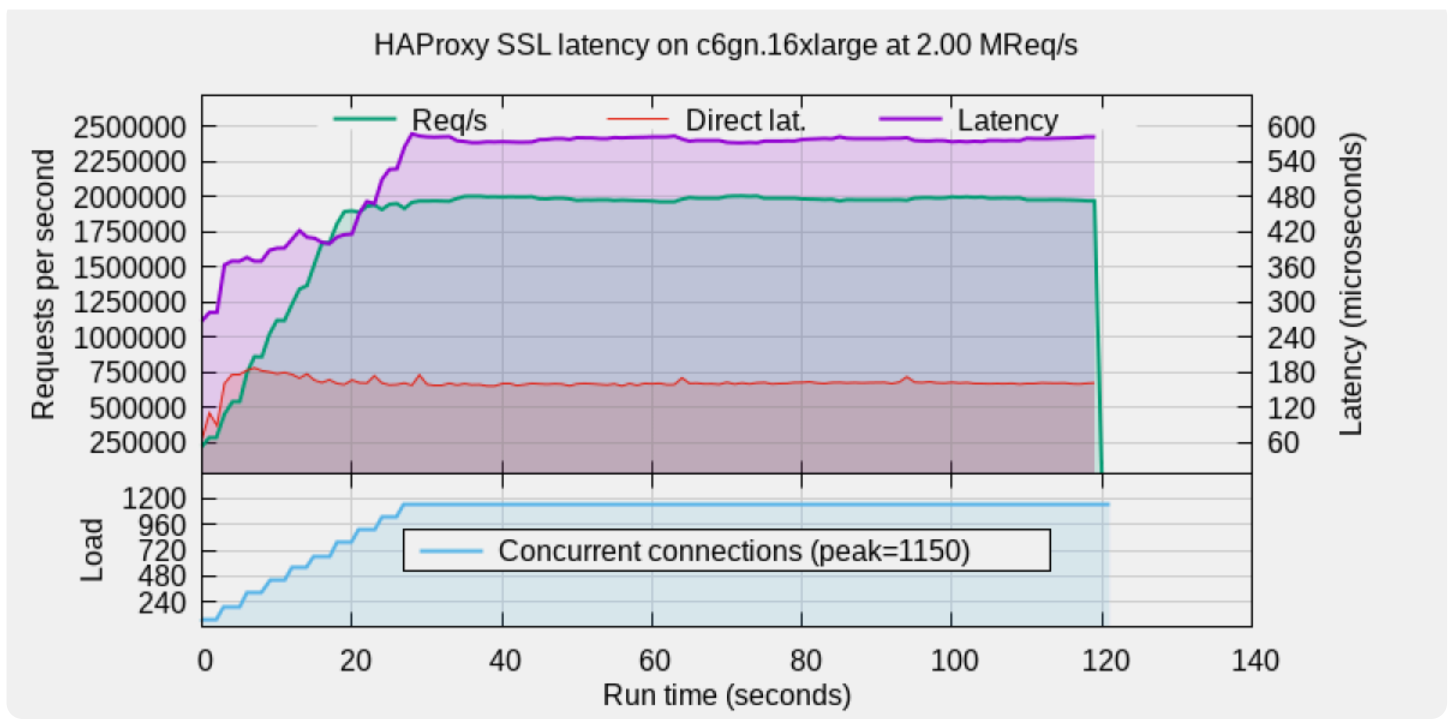
TLS for Free TLS 免费

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.

一些读者可能想知道我们是否可以在 HAProxy 中使用这些空闲的 CPU 周期来免费进行一些 TLS 终止。让我们试试吧。

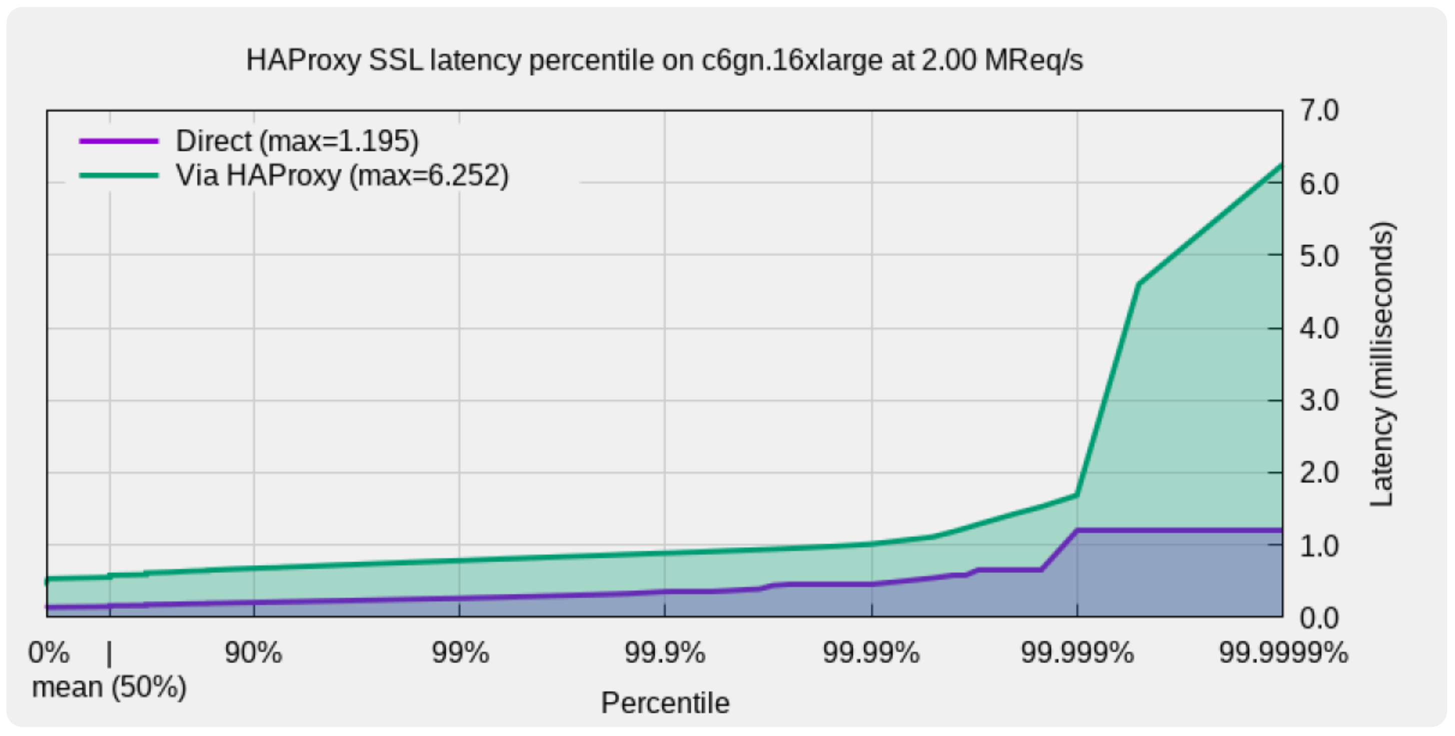
Results show that we're just between 1.99 and 2.01 million requests per second over TLS. The average response time is 575 microseconds for 162 direct, so HAProxy over TLS adds at most 413 microseconds to the chain. For these tests, I used an RSA 2048-bit certificate and the connections were made over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

结果显示，通过 TLS 每秒处理的请求量仅在 1.99 到 201 万个之间。162 次直接的平均响应时间为 575 微秒，因此基于 TLS 的 HAProxy 最多向链中添加 413 微秒。在这些测试中，我使用了 RSA 2048 位证书，并且通过 TLSv1.3 与 TLS_AES_256_GCM_SHA384 建立连接。



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.

延迟百分位数显示，100000 个请求中不到 1 个请求超过 1.6 毫秒，并且它们永远不会超过 6.25 毫秒，对于最大 1.2 毫秒的直接请求。HAProxy 在终止 TLS 时添加的时间永远不会超过 5 毫秒。



Now we've saturated the CPUs:

现在我们已经使 CPU 饱和：

view raw

```
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

    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:

view raw

```
> show profiling
  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.

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.