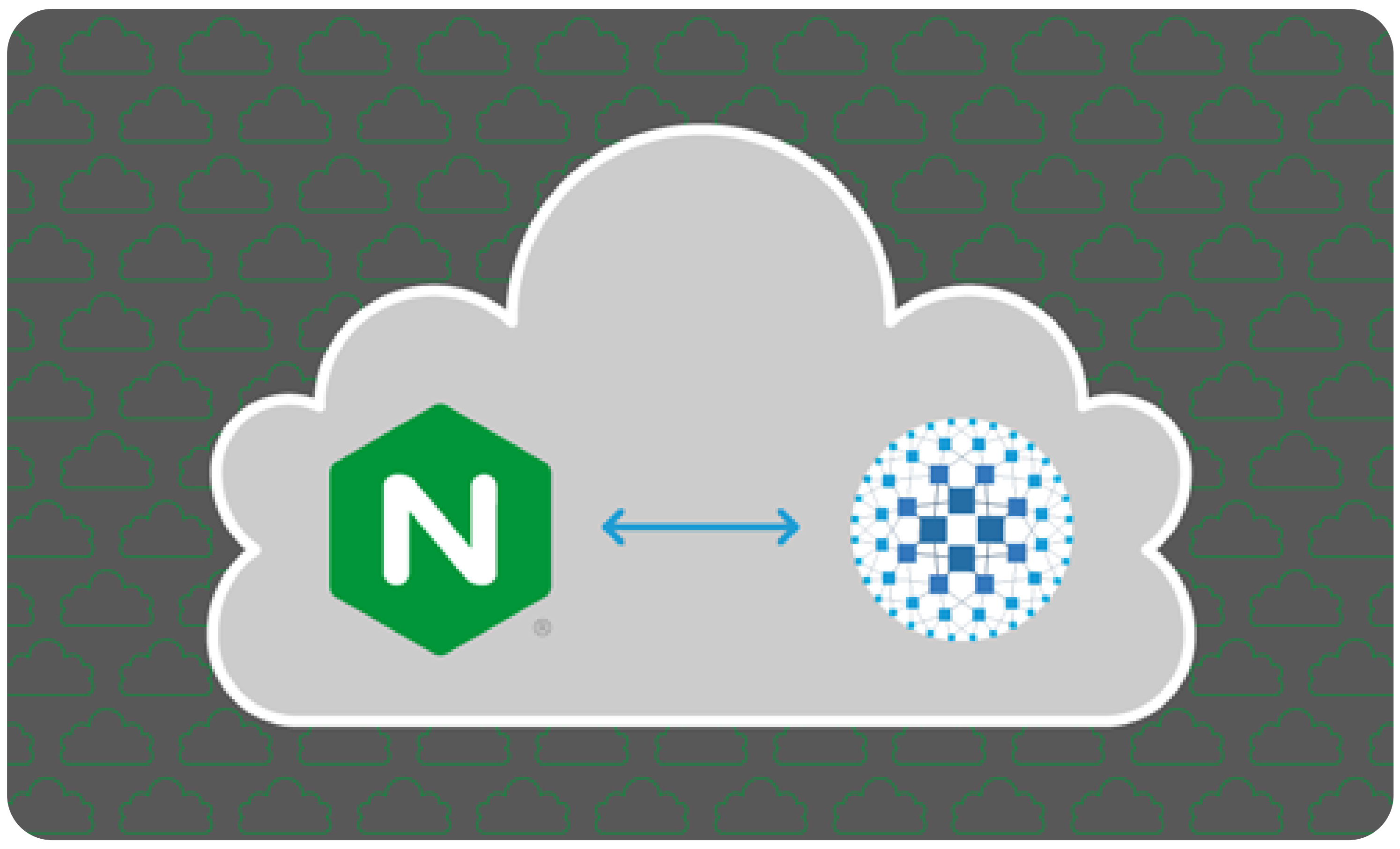
NGINX and HAProxy: Testing User Experience in the Cloud
NGINX 和 HAProxy：在云中测试用户体验

Jan 7, 2021 — by 由 Amir Rawdat 阿米尔·拉达特



Many performance benchmarks measure peak throughput or requests per second (RPS), but those metrics can oversimplify the performance story at real-world sites. Few organizations run their services at or near peak throughput, where a 10% change in performance either way can make a significant difference. The throughput or RPS a site requires is not infinite, but is fixed by external factors like the number of concurrent users they have to serve and the activity level of each user. In the end, what matters most is that your users receive the best level of service. End users don't care how many other people are visiting your site. They just care about the service they receive and don't excuse poor performance because the system is overloaded.

许多性能基准测试衡量峰值吞吐量或每秒请求数 （RPS），但这些指标可能会过度简化实际站点的性能。很少有组织以峰值吞吐量或接近峰值吞吐量运行其服务，无论哪种方式，性能 10% 的变化都会产生重大影响。站点所需的吞吐量或 RPS 不是无限的，而是由外部因素固定的，例如它们必须服务的并发用户数和每个用户的活动级别。最后，最重要的是您的用户获得最佳级别的服务。最终用户并不关心有多少人正在访问您的网站。他们只关心他们收到的服务，不会因为系统过载而为性能不佳找借口。

This leads us to the observation that what matters most is that an organization deliver consistent, low-latency performance to all their users, even under high load. In comparing NGINX and HAProxy running on Amazon Elastic Compute Cloud (EC2) as reverse proxies, we set out to do two things:

这使我们观察到，最重要的是，组织能够为其所有用户提供一致、低延迟的性能，即使在高负载下也是如此。在将 NGINX 和在 Amazon Elastic Compute Cloud （EC2） 上运行的 HAProxy 作为反向代理进行比较时，我们着手做两件事：

1. Determine what level of load each proxy comfortably handles

   确定每个代理可以轻松处理的负载级别
2. Collect the latency percentile distribution, which we find is the metric most directly correlated with user experience

   收集延迟百分位分布，我们发现这是与用户体验最直接相关的指标

Testing Protocols and Metrics Collected
测试收集的协议和指标

We used the load-generation program `wrk2` to emulate a client, making continuous requests over HTTPS during a defined period. The system under test – HAProxy or NGINX – acted as a reverse proxy, establishing encrypted connections with the clients simulated by `wrk` threads, forwarding requests to a backend web server running NGINX Plus R22, and returning the response generated by the web server (a file) to the client.

我们使用负载生成程序 `wrk2` 来模拟客户端，在定义的时间段内通过 HTTPS 发出连续请求。被测系统（HAProxy 或 NGINX）充当反向代理，与 `wrk` 线程模拟的客户端建立加密连接，将请求转发到运行 NGINX Plus R22 的后端 Web 服务器，并将 Web 服务器生成的响应（文件）返回给客户端。

Each of the three components (client, reverse proxy, and web server) ran Ubuntu 20.04.1 LTS on a c5n.2xlarge Amazon Machine Image (AMI) in EC2.

这三个组件（客户端、反向代理和 Web 服务器）中的每一个都在 EC2 中的 c5n.2xlarge Amazon 系统映像 （AMI） 上运行 Ubuntu 20.04.1 LTS。

As mentioned, we collected the full latency percentile distribution from each test run. Latency is defined as the amount of time between the client generating the request and receiving the response. A latency percentile distribution sorts the latency measurements collected during the testing period from highest (most latency) to lowest.

如前所述，我们从每次测试运行中收集了完整的延迟百分位数分布。延迟定义为客户端生成请求和接收响应之间的时间量。延迟百分位数分布将测试期间收集的延迟测量值从最高 （最高延迟） 到最低进行排序。

Testing Methodology 测试方法

Client 客户

Using `wrk2` (version 4.0.0), we ran the following script on the Amazon EC2 instance:

使用 `wrk2`（版本 4.0.0），我们在 Amazon EC2 实例上运行以下脚本：

```
taskset -c 0-3 wrk -t 4 -c 100 -d 30s -R
requests_per_second
 —latency https://adc.domain.com:443/
```

To simulate many clients accessing a web application, 4 `wrk` threads were spawned that together established 100 connections to the reverse proxy. During the 30-second test run, the script generated a specified number of RPS. These parameters correspond to the following `wrk2` options:

为了模拟许多客户端访问 Web 应用程序，生成了 4 个 `wrk` 线程，它们共同建立了 100 个与反向代理的连接。在 30 秒的测试运行期间，该脚本生成了指定数量的 RPS。这些参数对应于以下 `wrk2` 选项：

- `-t` option – Number of threads to create (4)

  `-t` 选项 – 要创建的线程数 （4）
- `-c` option – Number of TCP connections to create (100)

  `-c` option – 要创建的 TCP 连接数 （100）
- `-d` option – Number of seconds in the testing period (30 seconds)

  `-d` 选项 – 测试期间的秒数 （30 秒）
- `-R` option – Number of RPS issued by the client

  `-R` option – 客户端颁发的 RPS 数量
- `--latency` option – Output includes corrected latency percentile information

  `--latency` 选项 – 输出包括更正后的延迟百分位数信息

We incrementally increased the number of RPS over the set of test runs until one of the proxies hit 100% CPU utilization. For further discussion, see Performance Results.

我们在测试运行集中逐步增加了 RPS 的数量，直到其中一个代理达到 100% 的 CPU 利用率。有关进一步讨论，请参阅 性能结果 。

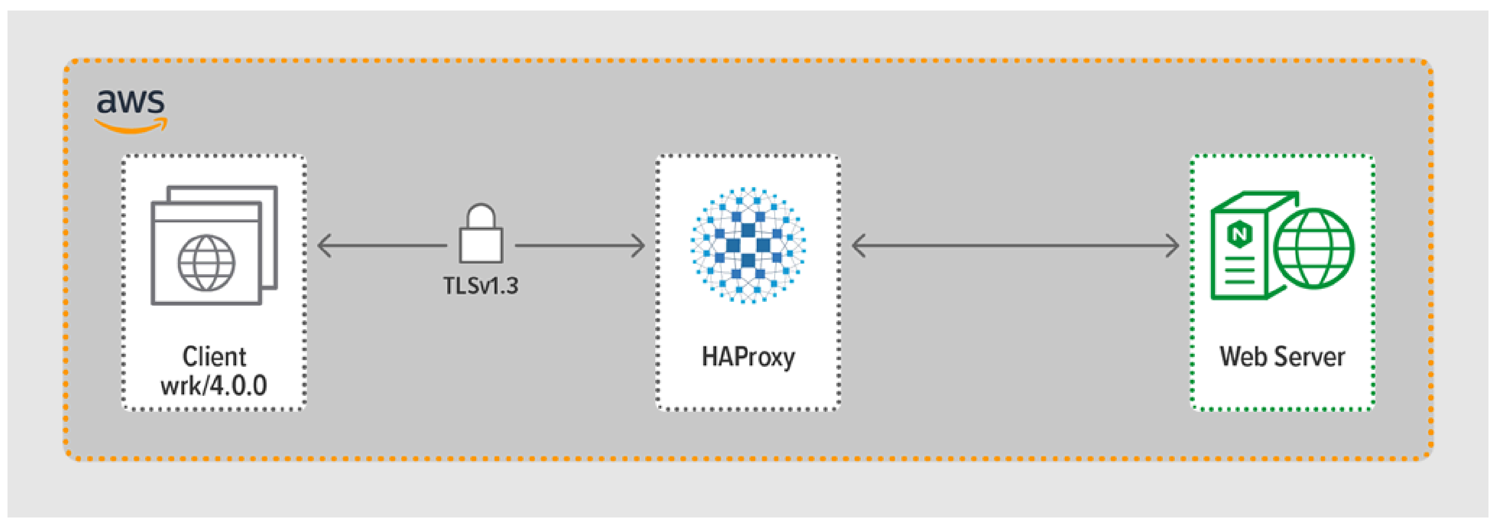
All connections between client and proxy were made over HTTPS with TLSv1.3. We used ECC with a 256-bit key size, Perfect Forward Secrecy, and the `TLS_AES_256_GCM_SHA384` cipher suite. (Because TLSv1.2 is still commonly used on the Internet, we re-ran the tests with it as well; the results were so similar to those for TLSv1.3 that we don't include them here.)

客户端和代理之间的所有连接都是通过 TLS v1.3 通过 HTTPS 建立的。我们使用了密钥大小为 256 位的 ECC、完全正向保密和 `TLS_AES_256_GCM_SHA384` 密码套件。（由于 TLSv1.2 在 Internet 上仍然普遍使用，因此我们也使用它重新运行了测试;结果与 TLSv1.3 的结果非常相似，因此我们在此处不列出它们。

HAProxy: Configuration and Versioning
HAProxy：配置和版本控制

We provisioned HAProxy version 2.3 (stable) as the reverse proxy.

我们预置了 HAProxy 版本 2.3（稳定）作为反向代理。



The number of simultaneous users at a popular website can be huge. To handle the large volume of traffic, your reverse proxy needs to be able to scale to take advantage of multiple cores. There are two basic ways to scale: multi-processing and multi-threading. Both NGINX and HAProxy support multi-processing, but there is an important difference – in HAProxy's implementation, processes do not share memory (whereas in NGINX they do). The inability to share state across processes has several consequences for HAProxy:

热门网站的并发用户数量可能非常庞大。为了处理大量流量，您的反向代理需要能够扩展以利用多个内核。有两种基本的扩展方法：多处理和多线程。NGINX 和 HAProxy 都支持多处理，但有一个重要的区别——在 HAProxy 的实现中，进程不共享内存（而在 NGINX 中它们共享）。无法跨进程共享状态会对 HAProxy 产生以下几个后果：

- Configuration parameters – including limits, statistics, and rates – must be defined separately for each process.

  必须为每个进程单独定义配置参数（包括限制、统计数据和速率）。
- Performance metrics are collected per-process; combining them requires additional config, which can be quite complex.

  性能指标按进程收集;组合它们需要额外的配置，这可能非常复杂。
- Each process handles health checks separately, so target servers are probed per process rather than per server as expected.

  每个进程单独处理运行状况检查，因此目标服务器按进程探测，而不是按预期按服务器探测。
- Session persistence is not possible.

  会话持久性是不可能的。
- A dynamic configuration change made via the HAProxy Runtime API applies to a single process, so you must repeat the API call for each process.

  通过 HAProxy Runtime API 进行的动态配置更改适用于单个进程，因此您必须对每个进程重复 API 调用。

Because of these issues, HAProxy strongly discourages use of its multi-processing implementation. To quote directly from the HAProxy configuration manual: "USING MULTIPLE PROCESSES IS HARDER TO DEBUG AND IS REALLY DISCOURAGED."

由于这些问题，HAProxy 强烈建议不要使用其多处理实现。直接引用 HAProxy 配置手册中的代码：“使用多个进程更难调试，并且非常不鼓励使用。

HAProxy introduced multi-threading in version 1.8 as an alternative to multi-processing. Multi-threading mostly solves the state-sharing problem, but as we discuss in Performance Results, in multi-thread mode HAProxy does not perform as well as in multi-process mode.

HAProxy 在 1.8 版本中引入了多线程作为多处理的替代方案。多线程主要解决了状态共享问题，但正如我们在 性能结果中讨论的那样，在多线程模式下，HAProxy 的性能不如在多进程模式下。

Our HAProxy configuration included provisioning for both multi-thread mode (HAProxy MT) and multi-process mode (HAProxy MP). To alternate between modes at each RPS level during the testing, we commented and uncommented the appropriate set of lines and restarted HAProxy for the configuration to take effect:

我们的 HAProxy 配置包括多线程模式 （HAProxy MT） 和多进程模式 （HAProxy MP） 的配置。为了在测试期间在每个 RPS 级别之间切换模式，我们注释和取消注释了适当的行集，并重新启动了 HAProxy 以使配置生效：

```
$ sudo service haproxy restart
```

Here's the configuration with HAProxy MT provisioned: four threads are created under one process and each thread pinned to a CPU. For HAProxy MP (commented out here), there are four processes each pinned to a CPU.

以下是预置了 HAProxy MT 的配置：在一个进程下创建四个线程，每个线程都固定到一个 CPU。对于 HAProxy MP（此处注释掉），有四个进程，每个进程都固定到一个 CPU。

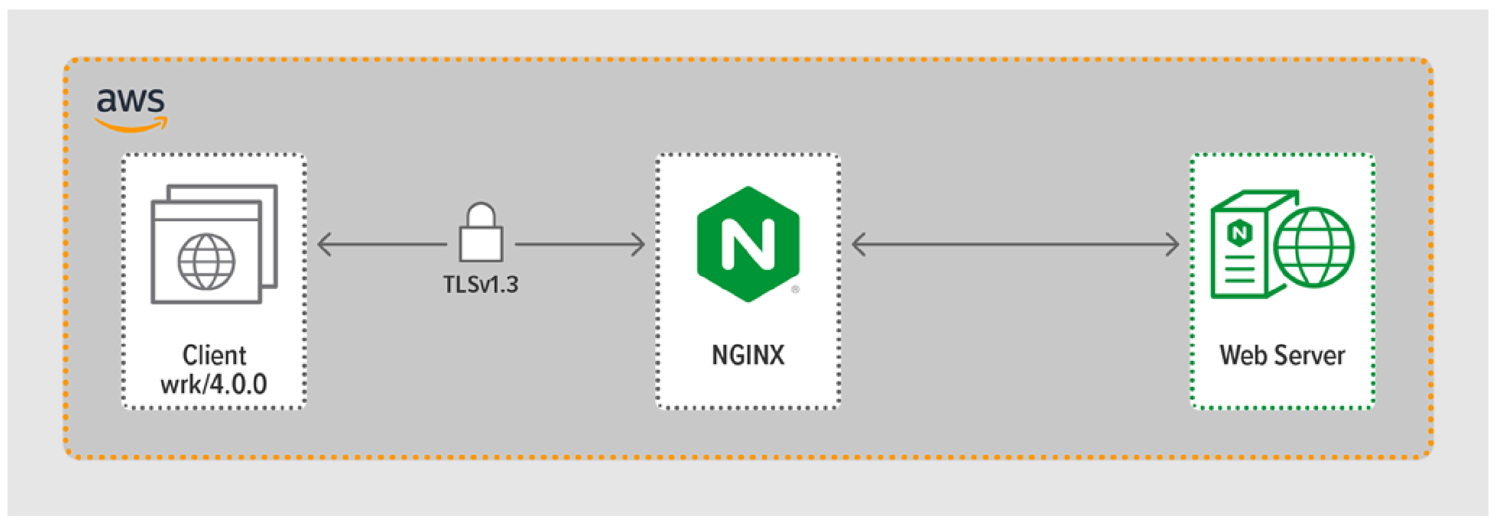
```
global

#Multi-thread mode

nbproc 1

nbthread 4

cpu-map auto:1/1-4 0-3


#Multi-process mode

#nbproc 4

#cpu-map 1 0

#cpu-map 2 1

#cpu-map 3 2
```

```
#cpu-map 4 3

ssl-server-verify none

log /dev/log local0

log /dev/log local1 notice

chroot /var/lib/haproxy

maxconn 4096


defaults

log global

option httplog

option http-keep-alive


frontend Local_Server

bind 172.31.12.25:80

bind 172.31.12.25:443 ssl crt /etc/ssl/certs/bundle-hapee.pem

redirect scheme https code 301 if !{ ssl_fc }

default_backend Web-Pool

http-request set-header Connection keep-alive


backend Web-Pool

mode http

server server1 backend.workload.1:80 check
```

NGINX: Configuration and Versioning
NGINX：配置和版本控制

We deployed NGINX Open Source version 1.18.0 as the reverse proxy.

我们部署了 NGINX Open Source 版本 1.18.0 作为反向代理。



To use all the cores available on the machine (four in this case), we included the `auto` parameter to the `worker_processes` directive, which is also the setting in the default **nginx.conf** file distributed from our repository. Additionally, the `worker_cpu_affinity` directive was included to pin each worker process to a CPU (each 1 in the second parameter denotes a CPU in the machine).

为了使用机器上所有可用的内核（在本例中为 4 个），我们在 `worker_processes` 指令中包含了 `auto` 参数，这也是从我们的存储库分发的默认 **nginx.conf** 文件中的设置。此外，还包括 `worker_cpu_affinity` 指令，用于将每个工作进程固定到 CPU（第二个参数中的每个 1 表示机器中的 CPU）。

```
user nginx;

worker_processes auto;

worker_cpu_affinity auto 1111;


error_log /var/log/nginx/error.log warn;

pid /var/run/nginx.pid;
```

```
events {

worker_connections 1024;

}


http {

include /etc/nginx/mime.types;

default_type application/octet-stream;


log_format main '$remote_addr – $remote_user [$time_local] "$request" '

'$status $body_bytes_sent "$http_referer" '

'"$http_user_agent" "$http_x_forwarded_for"';


access_log /var/log/nginx/access.log main;


sendfile on;

keepalive_timeout 65;

keepalive_requests 100000;


server {

listen 443 ssl reuseport;

ssl_certificate /etc/ssl/certs/hapee.pem;

ssl_certificate_key /etc/ssl/private/hapee.key;

ssl_protocols TLSv1.3;


location / {

proxy_set_header Connection ' ';

proxy_http_version 1.1;

proxy_pass http://backend;

}

}


upstream backend {

server backend.workload.1:80;

keepalive 100;

}

}
```

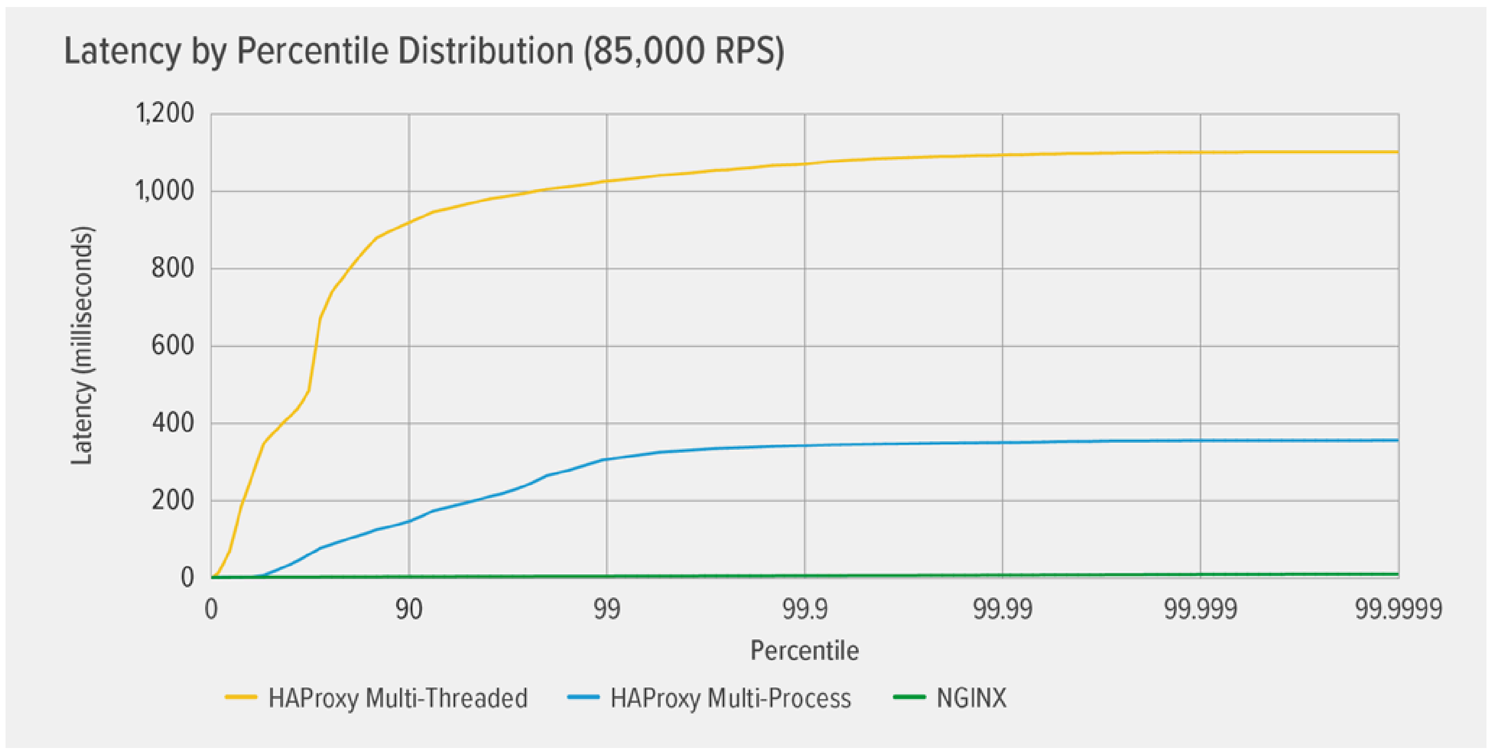
Performance Results 性能结果

With your reverse proxy acting as the front end to your application, its performance is critical.
由于反向代理充当应用程序的前端，因此其性能至关重要。

We tested each reverse proxy (NGINX, HAProxy MP, and HAProxy MT) at increasing numbers of RPS until one of them reached 100% CPU utilization. All three performed similarly at the RPS levels where CPU was not exhausted.
我们在不断增加的 RPS 数量下测试了每个反向代理（NGINX、HAProxy MP 和 HAProxy MT），直到其中一个达到 100% 的 CPU 利用率。这三者在 CPU 未耗尽的 RPS 级别上表现相似。

Reaching 100% CPU utilization occurred first for HAProxy MT, at 85,000 RPS, and at that point performance worsened dramatically for both HAProxy MT and HAProxy MP. Here we present the latency percentile distribution of each reverse proxy at that load level. The chart was plotted from the output of the `wrk` script using the HdrHistogram program available on GitHub.

HAProxy MT 首先达到 100% 的 CPU 利用率，为 85,000 RPS，此时 HAProxy MT 和 HAProxy MP 的性能急剧恶化。在这里，我们展示了每个反向代理在该负载级别的延迟百分位分布。该图表是使用 GitHub 上提供的 HdrHistogram 程序根据 `wrk` 脚本的输出绘制的。



At 85,000 RPS, latency with HAProxy MT climbs abruptly until the 90th percentile, then gradually levels off at approximately 1100 milliseconds (ms).

在 85000 RPS 时，HAProxy MT 的延迟会突然攀升，直到第 90 个百分位，然后在
大约 1100 毫秒 （ms） 时逐渐趋于平稳。

HAProxy MP performs better than HAProxy MT – latency climbs at a slower rate until the 99th percentile, at which point it starts to level off at roughly 400ms. (As a confirmation that HAProxy MP is more efficient, we observed that HAProxy MT used slightly more CPU than HAProxy MP at every RPS level.)

HAProxy MP 的性能优于 HAProxy MT – 延迟以较慢的速度攀升，直到第 99 个百分位，此时它在大约 400 毫秒时开始趋于平稳。（为了确认 HAProxy MP 更高效，我们观察到 HAProxy MT 在每个 RPS 级别使用的 CPU 都比 HAProxy MP 略多。

NGINX suffers virtually no latency at any percentile. The highest latency that any significant number of users might experience (at the 99.9999th percentile) is roughly 8ms.

NGINX 在任何百分位上几乎没有延迟。任何大量用户可能遇到的最高延迟（在第 99.9999 个百分位） 约为 8 毫秒 。

What do these results tell us about user experience? As mentioned in the introduction, the metric that really matters is response time from the end-user perspective, and not the service time of the system under test.

这些结果告诉我们关于用户体验的什么信息？如简介中所述，真正重要的指标是最终用户视角的响应时间，而不是被测系统的服务时间。

It's a common misconception that the median latency in a distribution best represents user experience. In fact, the median is the number that about half of response times are worse than! Users typically issue many requests and access many resources per page load, so several of their requests are bound to experience latencies at the upper percentiles in the chart (99th through 99.9999th). Because users are so intolerant of poor performance, latencies at the high percentiles are the ones they're most likely to notice.

Think of it this way: your experience of checking out at a grocery store is determined by how long it takes to leave the store from the moment you got in the checkout line, not just how long it took for the cashier to ring up your items. If, for example, a customer in front of you disputes the price of an item and the cashier has to get someone to verify it, your overall checkout time is much longer than usual.

To take this into account in our latency results, we need to correct for something called
coordinated omission
, in which (as explained in a note at the end of the `wrk2` README) "high latency responses result in the load generator coordinating with the server to avoid measurement during high latency periods". Luckily `wrk2` corrects for coordinated omission by default (for more details about coordinated omission, see the README).

When HAProxy MT exhausts the CPU at 85,000 RPS, many requests experience high latency. They are rightfully included in the data because we are correcting for coordinated omission. It just takes one or two high-latency requests to delay a page load and result in the perception of poor performance. Given that a real system is serving multiple users at a time, even if only 1% of requests have high latency (the value at the 99th percentile), a large proportion of users are potentially affected.

Conclusion

One of the main points of performance benchmarking is determining whether your app is responsive enough to satisfy users and keep them coming back.

Both NGINX and HAProxy are software-based and have event-driven architectures. While HAProxy MP delivers better performance than HAProxy MT, the lack of state sharing among the processes makes management more complex, as we detailed in HAProxy: Configuration and Versioning. HAProxy MT addresses these limitations, but at the expense of lower performance as demonstrated in the results.

With NGINX, there are no tradeoffs – because processes share state, there’s no need for a multi-threading mode. You get the superior performance of multi-processing without the limitations that make HAProxy discourage its use.

To get started with NGINX Open Source, download binaries or source.

← Previous: Updates to NGINX Unit for Autumn 2020 Next: Hello, New API →