Cloud Onload® HAProxy Cookbook

SF-122383-CD

Issue 2

Table of Contents

1 Introduction

This chapter introduces you to this document. See:

- About this document on page 1
- Intended audience on page 2
- Registration and support on page 2
- Download access on page 2
- Further reading on page 2.

1.1 About this document

This document is the *HAProxy Cookbook* for Cloud Onload. It gives procedures for technical staff to configure and run tests, to benchmark HAProxy utilizing Solarflare's Cloud Onload and Solarflare NICs.

This document contains the following chapters:

- Introduction on page 1 (this chapter) introduces you to this document.
- Overview on page 3 gives an overviews of the software distributions used for this benchmarking.
- Summary of benchmarking on page 6 summarizes how the performance of HAProxy has been benchmarked, both with and without Cloud Onload, to determine what benefits might be seen.
- Evaluation on page 10 describes how the performance of the test system is evaluated.
- Benchmark results on page 16 presents the benchmark results that are achieved.

and the following appendixes:

- Cloud Onload profiles on page 22 contains the Cloud Onload profiles used for this benchmarking.
- Installation and configuration on page 30 describes how to install and configure the software distributions used for this benchmarking.

1.2 Intended audience

The intended audience for this *HAProxy Cookbook* are:

- software installation and configuration engineers responsible for commissioning and evaluating this system
- system administrators responsible for subsequently deploying this system for production use.

1.3 Registration and support

Support is available from support@solarflare.com.

1.4 Download access

Cloud Onload can be downloaded from: https://support.solarflare.com/.

Solarflare drivers, utilities packages, application software packages and user documentation can be downloaded from: https://support.solarflare.com/.

The scripts and Cloud Onload profiles used for this benchmarking are available on request from support@solarflare.com.

Please contact your Solarflare sales channel to obtain download site access.

1.5 Further reading

For advice on tuning the performance of Solarflare network adapters, see the following:

- *Solarflare Server Adapter User Guide* (SF-103837-CD).

  This is available from: https://support.solarflare.com/.

For more information about Cloud Onload, see the following:

- *Onload User Guide* (SF-104474-CD).

  This is available from: https://support.solarflare.com/.

2 Overview

This chapter gives an overview of the software distributions used for this benchmarking. See:

- HAProxy overview on page 3
- NGINX overview on page 3
- Wrk2 overview on page 4
- Cloud Onload overview on page 4.

2.1 HAProxy overview

HAProxy is a free, very fast and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications. It is particularly suited for very high traffic web sites and powers quite a number of the world's most visited ones. It is now shipped with most mainstream Linux distributions, and is often deployed in cloud platforms.

Its mode of operation makes its integration into existing architectures very easy and riskless, while still offering the possibility not to expose fragile web servers to the net.

HAProxy is heavily network dependent by design, so its performance can be significantly improved through enhancements to the underlying networking layer.

2.2 NGINX overview

Open source NGINX [engine x] is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server.

NGINX Plus is a software load balancer, web server, and content cache built on top of open source NGINX. NGINX has exclusive enterprise-grade features beyond what's available in the open source offering, including session persistence, configuration via API, and active health checks.

Open source NGINX is used for this benchmarking.

2.3 Wrk2 overview

Wrk is a modern HTTP benchmarking tool capable of generating significant load when run on a single multi-core CPU. It combines a multithreaded design with scalable event notification systems such as epoll and kqueue. An optional LuaJIT script can perform HTTP request generation, response processing, and custom reporting.

`Wrk2` is `wrk` modified to produce a constant throughput load, and accurate latency details to the high 9s (it can produce an accurate 99.9999 percentile when run long enough). In addition to `wrk`'s arguments, `wrk2` takes a required throughput argument (in total requests per second) via either the `--rate` or `-R` parameters.

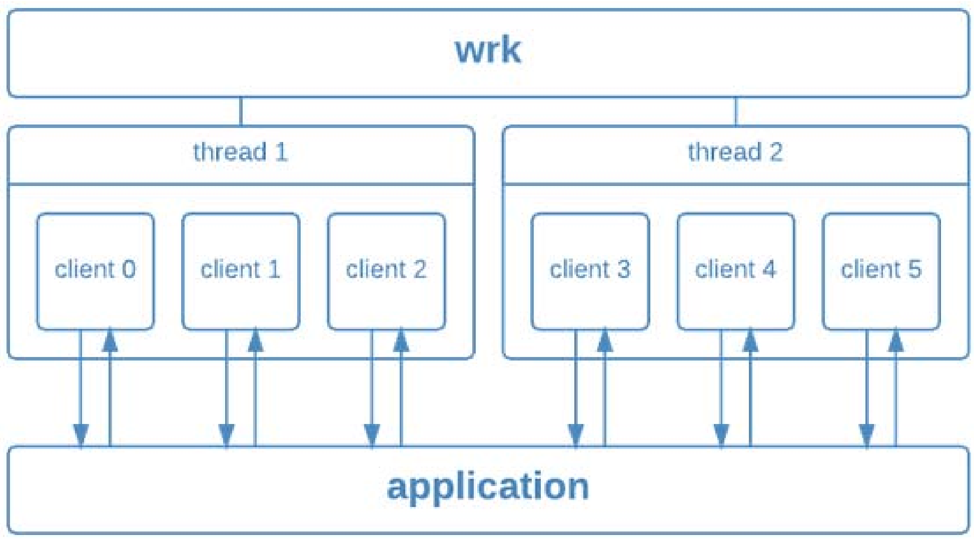

Figure 1: Wrk/wrk2 architecture

2.4 Cloud Onload overview

Cloud Onload is a high performance network stack from Solarflare (https://www.solarflare.com/) that dramatically reduces latency, improves CPU utilization, eliminates jitter, and increases both message rates and bandwidth. Cloud Onload runs on Linux and supports the TCP network protocol with a POSIX compliant sockets API and requires no application modifications to use. Cloud Onload achieves performance improvements in part by performing network processing at user-level, bypassing the OS kernel entirely on the data path.

Cloud Onload is a shared library implementation of TCP, which is dynamically linked into the address space of the application. Using Solarflare network adapters, Cloud Onload is granted direct (but safe) access to the network. The result is that the application can transmit and receive data directly to and from the network, without any involvement of the operating system. This technique is known as “kernel bypass”.

When an application is accelerated using Cloud Onload it sends or receives data without access to the operating system, and it can directly access a partition on the network adapter.

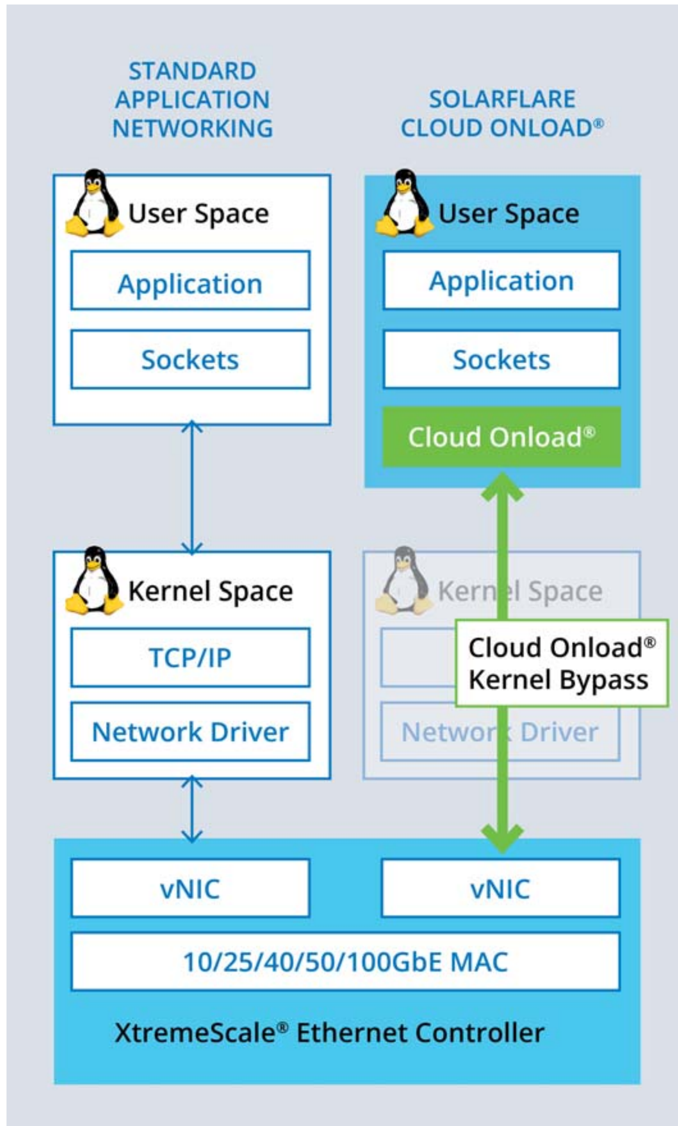

Figure 2: Cloud Onload architecture

3 Summary of benchmarking

This chapter summarizes how the performance of HAProxy has been benchmarked, both with and without Cloud Onload, to determine what benefits might be seen. See:

- Overview of HAProxy benchmarking on page 6
- Architecture for HAProxy benchmarking on page 7
- HAProxy benchmarking process on page 8.

3.1 Overview of HAProxy benchmarking

The HAProxy benchmarking uses two servers:

- The *load server* runs multiple instances of wrk2 to generate requests, and multiple instances of NGINX webservers to service requests.
- The *proxy server* runs multiple instances of HAProxy. It receives the requests that originate from wrk2 on the load server, and proxies those requests to an NGINX webserver on the load server.

Various benchmark tests are run, with HAProxy using the Linux kernel network stack.

The tests are then repeated, using Cloud Onload to accelerate HAProxy. Two different Cloud Onload profiles are used, that have different priorities:

- The balanced profile gives excellent throughput, with low latency. It has reduced CPU usage at lower traffic rates.
- The performance profile is latency focused. It constantly polls for network events to achieve the lowest latency possible, and so has higher CPU usage.

The results using the kernel network stack are compared with the results using the two different Cloud Onload profiles.

3.2 Architecture for HAProxy benchmarking

Benchmarking was performed with two Dell R640 servers, with the following specification:

Server	Dell R640
Memory	192GB
NICs	2 × X2541 (single port 100G): • Each NIC is affinitized to a separate NUMA node.
CPU	2 × Intel® Xeon® Gold 6148 CPU @ 2.40GHz: • Each CPU is on a separate NUMA node • There are 20 cores per CPU • Hyperthreading is enabled to give 40 hyperthreads per NUMA node
OS	Red Hat Enterprise Linux Server release 7.6 (Maipo)
Software	HAProxy 1.9.7 NGINX 1.17 wrk2 4.0.0

Each server is configured to leave as many CPUs as possible available for the application being benchmarked.

Each server has 2 NUMA nodes. 2 Solarflare NICs are fitted, each affinitized to a separate NUMA node, and connected directly to the corresponding NIC in the other server:

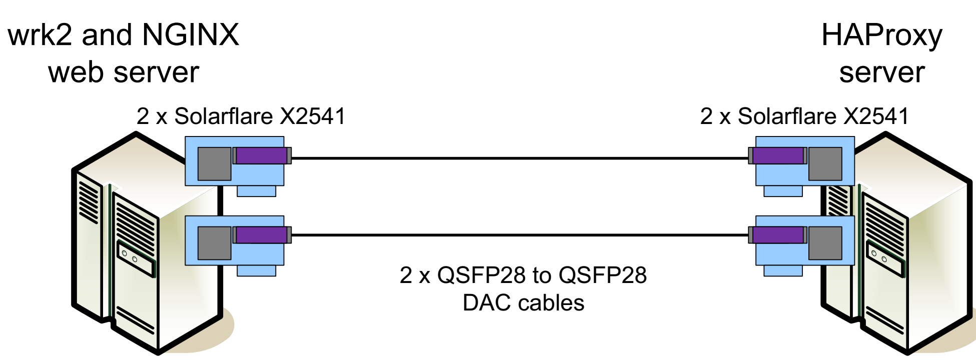

Figure 3: Architecture for HAProxy benchmarking

3.3 HAProxy benchmarking process

These are the high-level steps we followed to complete benchmarking with HAProxy:

- Install and test NGINX on the first server.
- Install wrk2 on the first server.
- Install HAProxy on the second server.
- Start NGINX web servers on the first server.

  All iterations of the test use the same configuration for consistency:
  - 40 NGINX web servers are used.
  - Each web server runs a single NGINX worker process.
  - Each NGINX worker process is assigned to a dedicated CPU, distributed across the NUMA nodes.
  - Each NGINX worker process uses the NIC that is affinitized to the local NUMA node for its CPU.
  - Each NGINX worker process uses a dedicated port.
  - Each NGINX web server is accelerated by Cloud Onload, to maximize the responsiveness of the proxied server.
- Start HAProxy servers on the other server:
  - One HAProxy server is used per NUMA node on the server.
    The setup used has 2 NUMA nodes, and so 2 HAProxy servers are started.
  - The first iteration of the test uses a single worker process per HAProxy server.
- Start `wrk2` on the first server to generate load.

  All iterations of the test use the same configuration for consistency:
  - 20 `wrk2` processes are used.
  - Each `wrk2` process is assigned to a dedicated CPU, distributed across the NUMA nodes.
  - Each `wrk2` process uses the NIC that is affinitized to the local NUMA node for its CPU.
  - Each `wrk2` process is accelerated by Cloud Onload, to maximize the throughput of each connection going to the HAProxy server.
- Record the response rate of the proxied web server, as the number of requests per second.
- Increase the number of worker processes on each HAProxy server, and repeat the test.
  - Each worker process is assigned to a dedicated CPU, distributed across the NUMA nodes.

- Each worker process uses the NIC that is affinitized to the local NUMA node for its CPU.

Continue doing this until the number of HAProxy worker processes on the second server is the same as the number of NGINX worker processes on the first web server. For the setup used, this is 40 processes.

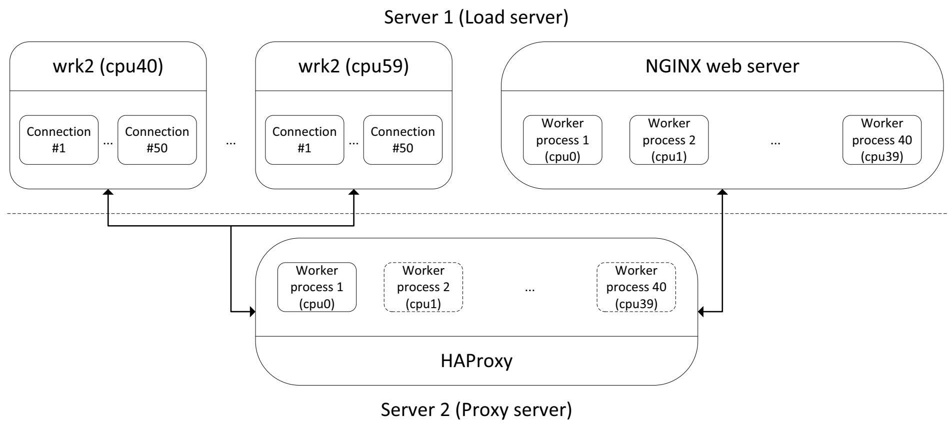

Figure 4: HAProxy software usage

- Repeat all tests, accelerating HAProxy with Cloud Onload.

These steps are detailed in the remaining chapters of this *Cookbook*.

The scripts and Cloud Onload profiles used for this benchmarking, that perform the above steps, are available on request from support@solarflare.com.

4 Evaluation

This chapter describes how the performance of the test system is evaluated. See:

- General server setup on page 10
- wrk2 client (on Load server) on page 11
- NGINX backend webservers (on Load server) on page 12
- HAProxy (on Proxy server) on page 13
- Graphing the benchmarking results on page 15.

4.1 General server setup

Each server is setup using a script that does the following:

1 Create a file that makes new module settings:

```
cat > /etc/modprobe.d/proxy.conf <<EOL
options sfc \\
  performance_profile=throughput \\
  rss_cpus=20 \\
  rx_irq_mod_usec=90 \\
  irq_adapt_enable=N \\
  rx_ring=512 \\
  piobuf_size=0
options nf_conntrack_ipv4 \\
  hashsize=524288
EOL
```

> **NOTE:** This script is required only when running HAProxy with the kernel network stack (i.e. without Cloud Onload).

2 Reload the drivers to pick up the new module settings:

```
onload_tool reload
```

3 Use the `network-throughput` tuned profile:

```
tuned-adm profile network-throughput
```

4 Stop various services:

```
systemctl stop irqbalance
systemctl stop iptables
systemctl stop firewalld
```

5 Increase the sizes of the OS receive and send buffers:

```
sysctl net.core.rmem_max=16777216 net.core.wmem_max=16777216
```

6 Configure huge pages:

```
sysctl vm.nr_hugepages=4096 > /dev/null
```

7 Ensure the connection tracking table is large enough:

```
sysctl net.netfilter.nf_conntrack_max=$(( $(sysctl --values
net.netfilter.nf_conntrack_buckets) * 4 )) > /dev/null
```

8 Increase the system-wide and per-process limits on the number of open files:

```
sysctl fs.file-max=8388608 > /dev/null
sysctl fs.nr_open=8388608 > /dev/null
```

9 Increase the range of local ports, so that the server can open lots of outgoing network connections:

```
sysctl -w net.ipv4.ip_local_port_range="2048 65535" > /dev/null
```

10 Increase the number of file descriptors that are available:

```
ulimit -n 8388608
```

11 Exclude from IRQ balancing the CPUs that are used for running HAProxy. For example, to exclude CPUs 0 to 39:

```
IRQBALANCE_BANNED_CPUS=ff,ffffffff irqbalance --oneshot
```

4.2 wrk2 client (on Load server)

Set up 20 instances of wrk2, running on cores 40 to 59, and start them all. An example command line for the first instance (core 40) is below.

```
EF_CLUSTER_SIZE=10 \
        taskset -c 40 \
        onload -p wrk-profile.opf \
        /opt/wrk2/wrk \
        -R 500000 \
        -c 100 \
        -d 60 \
        -t 1 \
        http://192.168.0.101:1080/1024.bin
```

This example runs a *Requests per second* test using a payload size of 1024 bytes (HTTP GET with keepalive).

- The `taskset -c` parameter is changed for each instance, to use cores 40 to 59.
- Instances on the even cores (NUMA node 0) use the IP address for the NIC that is affinitized to NUMA node 0 on the proxy server.
- Instances on the odd cores (NUMA node 1) use the IP address for the NIC that is affinitized to NUMA node 1 on the proxy server.
- The port number is fixed at `1080`. This is the port listened to by the proxy server.
- `EF_CLUSTER_SIZE` is set to the number of `wrk2` instances which share the same IP address (i.e. 10 per NUMA node in this case).

4.3 NGINX backend webservers (on Load server)

Create a set of 40 backend webservers, with similar configuration for each webserver, and start them all. An example command line to start the first webserver (port 1050 of the NIC that is affinitized to NUMA node 0) is below:

```
onload -p nginx-server.opf sbin/nginx -c nginx-server-node0_1050.conf
```

The corresponding `nginx-server-node0_1050.conf` configuration file is shown below.

```
cat >nginx-server-node0_1050.conf <<EOL
worker_processes       1;
worker_rlimit_nofile   8388608;
worker_cpu_affinity    auto 00000000000000000000000000000001;

pid                    /var/run/nginx-node0_1050.pid;

events {
    multi_accept            off;
    accept_mutex            off;
    use                     epoll;
    worker_connections      200000;
}

error_log logs/error-node0_1050.log debug;

http {
    default_type            application/octet-stream;

    access_log              off;
    error_log               /dev/null crit;

    keepalive_timeout  300s;
    keepalive_requests 1000000;

    server {
        listen       192.168.0.100:1050 reuseport;
        server_name  localhost;

        open_file_cache max=100000 inactive=20s;
        open_file_cache_valid 30s;
        open_file_cache_errors off;

        location = /0 {
          return 204;
        }
        location / {
            root   html-node0_1050;
            index  index.html;
        }
        location = /upload {
          return 200 'Thank you';
        }
    }
}
EOL
```


- The `worker_cpu_affinity` is changed for each instance, to use cores 0 to 39.
- Instances on the even cores (NUMA node 0) have the IP address in `http` → `server` → `listen` set to use the NIC that is affinitized to NUMA node 0, and the port address incrementing from `1050` upwards.
- Instances on the odd cores (NUMA node 1) have the IP address in `http` → `server` → `listen` set to use the NIC that is affinitized to NUMA node 1, and the port address also incrementing from `1050` upwards.
- The `pid` is changed for each instance.
- The `error_log` is changed for each instance
- The `server` → `location` → `root` is changed for each instance.

Static files for webservers

Each webserver serves static files from within the install directory, in a subdirectory that is configured by the `root` directive. Each webserver instance uses its own subdirectory, to avoid filesystem contention, and to model more closely a farm of separate servers.

The static files used range from 400B to 1MB. They were generated using `dd`. The example below creates the necessary files for the server that uses the above configuration file:

```
# mkdir -p /opt/nginx/html-node0_1050
# for payload in 400 1024 10240 32768 65536 102400 131072 262144 1024000
> do
> dd if=/dev/urandom of=/opt/nginx/html-node0_1050/$payload \
>    bs=$payload count=1 > /dev/null 2>&1
> done
```

4.4 HAProxy (on Proxy server)

Start various numbers of HAProxy worker processes (2, 8, 16, 24, 32 or 40), using either the kernel network stack, or one of two different Onload-accelerated network stacks. A total of 18 iterations are required.

Example command lines to start 16 worker processes are below:

- To start the proxy server with the kernel network stack, use the following:

  ```
  sbin/haproxy -c haproxy-node0_16.conf
  sbin/haproxy -c haproxy-node1_16.conf
  ```

- To start the proxy server with an Onload-accelerated network stack, use one of the following, for the two different Onload profiles under test:

  ```
  onload -p haproxy-balanced.opf sbin/haproxy -c haproxy-node0_16.conf
  onload -p haproxy-balanced.opf sbin/haproxy -c haproxy-node1_16.conf
  ```

  ```
  onload -p haproxy-performance.opf sbin/haproxy -c haproxy-node0_16.conf
  onload -p haproxy-performance.opf sbin/haproxy -c haproxy-node1_16.conf
  ```


The corresponding `haproxy-node0_16.conf` configuration file is shown below.

```
cat >haproxy-node0_16.conf <<EOL
global
  daemon
  log stdout  local0 notice
  maxconn 200000
  nbproc 8
  cpu-map 1 0
  cpu-map 2 2
  cpu-map 3 4
  cpu-map 4 6
  cpu-map 5 8
  cpu-map 6 10
  cpu-map 7 12
  cpu-map 8 14

defaults
  log global
  timeout client 30s
  timeout server 30s
  timeout connect 30s

frontend MyFrontend
  bind 192.168.0.101:1080 process 1
  bind 192.168.0.101:1080 process 2
  bind 192.168.0.101:1080 process 3
  bind 192.168.0.101:1080 process 4
  bind 192.168.0.101:1080 process 5
  bind 192.168.0.101:1080 process 6
  bind 192.168.0.101:1080 process 7
  bind 192.168.0.101:1080 process 8
  default_backend   MyBackend

backend MyBackend
  mode      http
  balance   static-rr
  server WebServer1050 192.168.0.100:1050
  server WebServer1051 192.168.0.100:1051
  server WebServer1052 192.168.0.100:1052
  server WebServer1053 192.168.0.100:1053
  server WebServer1054 192.168.0.100:1054
  server WebServer1055 192.168.0.100:1055
  server WebServer1056 192.168.0.100:1056
  server WebServer1057 192.168.0.100:1057
EOL
```

- The `nbproc` is set to the number of worker processes which share the same NUMA node (i.e. half the number of worker processes in the test).
- The `cpu-maps` are set to use one core per worker process, all on the same NUMA node (even core numbers in this case).

  For the corresponding `haproxy-node1_16.conf` configuration file, the odd core numbers are used. For example:

  ```
  cpu-map 1 1
  ```

- Instances on the even cores (NUMA node 0) have the IP addresses set as follows:
  - `frontend MyFrontend` → `bind` is set to use the NIC that is affinitized to NUMA node 0, with the port number set to `1080`.
  - `backend MyBackend` → `server` is set to use the NIC that is affinitized to NUMA node 0 on the load server, with all port numbers in the range `1050-1069`.
- Instances on the odd cores (NUMA node 1) have the IP addresses set as follows:
  - `frontend MyFrontend` → `bind` is set to use the NIC that is affinitized to NUMA node 1, with the port number set to `1080`.
  - `backend MyBackend` → `server` is set to use the NIC that is affinitized to NUMA node 1 on the load server, with all port numbers in the range `1050-1069`.

4.5 Graphing the benchmarking results

The results from each pass of `wrk2` are now gathered and summed, so that they can be further analyzed. They are then transferred into an Excel spreadsheet, to create graphs from the data.

5 Benchmark results

This chapter presents the benchmark results that are achieved. See:

- Results on page 17
- Analysis on page 21.

5.1 Results

Connections per second

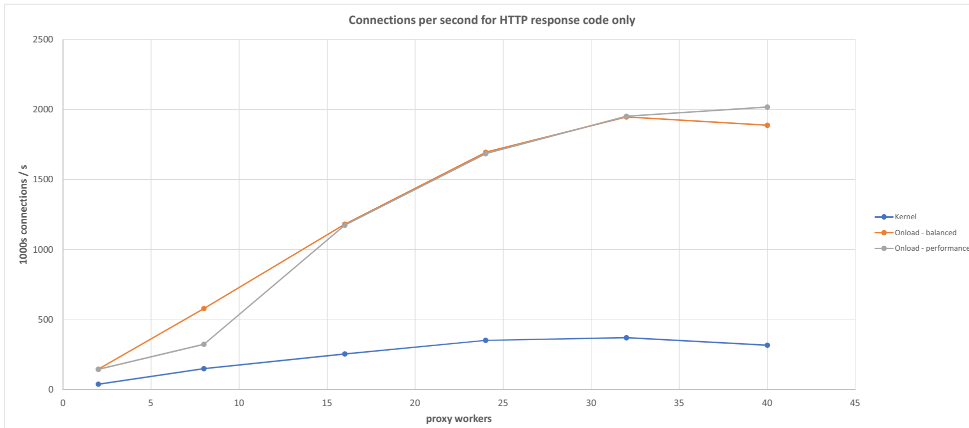

Figure 5: HAProxy connections per second

Table 1 below shows the results that were used to plot the graph in Figure 5 above.

Table 1: Thousands of connections per second

Worker processes	Kernel	Onload balanced	Onload performance	Onload balanced gain	Onload performance gain
2	39	146	145	271%	268%
8	150	579	324	287%	116%
16	255	1181	1174	363%	360%
24	352	1695	1684	381%	378%
32	371	1947	1952	425%	427%
40	317	1887	2017	496%	537%

Requests per second

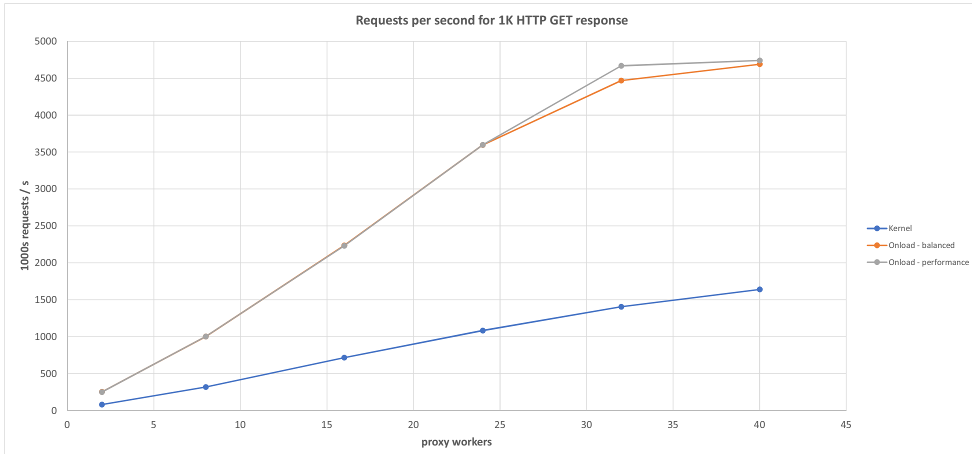

Figure 6: HAProxy requests per second

Table 2 below shows the results that were used to plot the graph in Figure 6 above.

Table 2: Thousands of requests per second for 1KB

Worker processes	Kernel	Onload balanced	Onload performance	Onload balanced gain	Onload performance gain
2	80	253	250	218%	214%
8	318	1002	1002	215%	215%
16	716	2236	2231	212%	212%
24	1082	3595	3597	232%	232%
32	1405	4468	4669	218%	232%
40	1640	4690	4741	186%	189%

Throughput

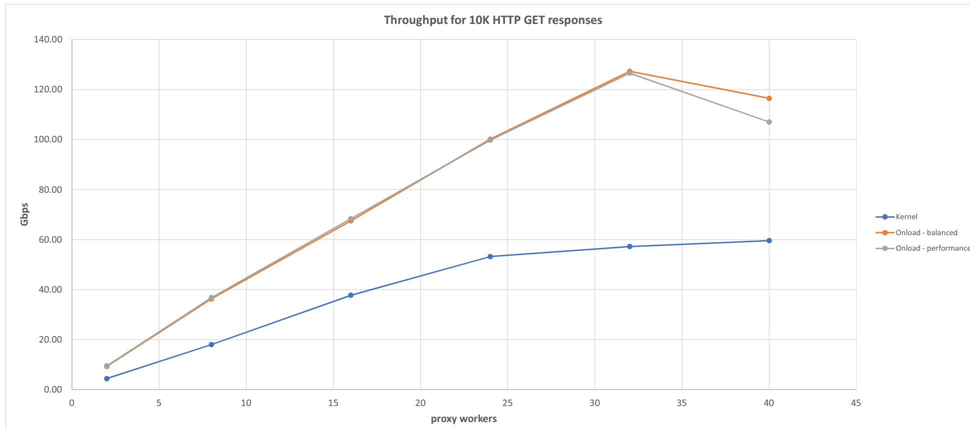

Figure 7: HAProxy throughput

Table 3 below shows the results that were used to plot the graph in Figure 7 above.

Table 3: Throughput for 10K in Gbps

Worker processes	Kernel	Onload balanced	Onload performance	Onload balanced gain	Onload performance gain
2	4.41	9.27	9.50	110%	115%
8	18.03	36.28	36.78	101%	104%
16	37.74	67.51	68.32	79%	81%
24	53.18	100.13	99.77	88%	88%
32	57.21	127.28	126.51	122%	121%
40	59.60	116.46	107.01	95%	80%

Latency

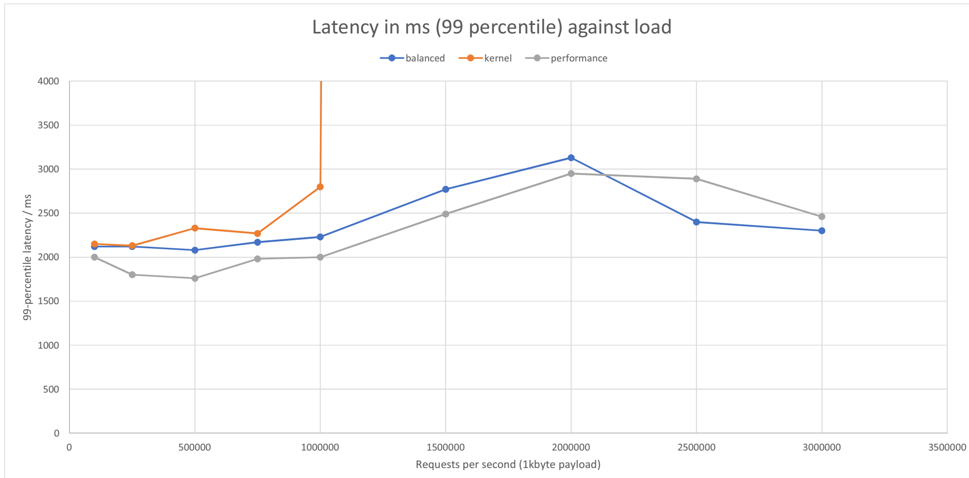

Figure 8: HAProxy latency

Table 4 below shows the results that were used to plot the graph in Figure 8 above.

Table 4: Latency for 1KB

<table>
<tr><th>Requests per second</th><th>Kernel</th><th>Onload balanced</th><th>Onload performance</th><th>Onload balanced gain</th><th>Onload performance gain</th></tr>
<tr><td>100000</td><td>2150</td><td>2120</td><td>2000</td><td>1%</td><td>8%</td></tr>
<tr><td>250000</td><td>2130</td><td>2120</td><td>1800</td><td>0%</td><td>18%</td></tr>
<tr><td>500000</td><td>2330</td><td>2080</td><td>1760</td><td>12%</td><td>32%</td></tr>
<tr><td>750000</td><td>2270</td><td>2170</td><td>1980</td><td>5%</td><td>15%</td></tr>
<tr><td>1000000</td><td>2800</td><td>2230</td><td>2000</td><td>26%</td><td>40%</td></tr>
<tr><td>1500000</td><td>8830000</td><td>2770</td><td>2490</td><td rowspan="4" colspan="2">Kernel cannot maintain requested packet rate. Gain is meaningless.</td></tr>
<tr><td>2000000</td><td>20790000</td><td>3130</td><td>2950</td></tr>
<tr><td>2500000</td><td>25870000</td><td>2400</td><td>2890</td></tr>
<tr><td>3000000</td><td>34210000</td><td>2300</td><td>2460</td></tr>
</table>

5.2 Analysis

When compared with the kernel stack, Cloud Onload delivers significant improvements to all metrics.

Connections per second

The connections per second shows great improvement with Cloud Onload, peaking at an improvement of 537% over the kernel stack. With large numbers of proxy workers (32 to 40) the Cloud Onload performance levels out. This is most likely because the load server is unable to generate and measure any more traffic, but might be because the proxy server itself is saturated.

Requests per second

The requests per second also shows great improvement with Cloud Onload, peaking at an improvement of 232% over the kernel stack. With 40 worker processes, results continue to improve, indicating that further performance is available from Cloud Onload.

Throughput

The throughput shows significant improvement with Cloud Onload, peaking at an improvement of 122% over the kernel stack. With large numbers of proxy workers (32 to 40) the Cloud Onload performance slightly decreases, either because the load server cannot generate any more traffic, or because the proxy server itself is saturated.

Latency

The latency figures are as output by `wrk2`, and show the time from when the *should* have been sent (according to the configured packet rate), until when the packet was actually received. The 99 percentile figure is reported.

When the kernel stack packet rate is raised above 1 million requests per second, it can no longer maintain this rate. Jitter increases, the number of outliers exceeds 1%, and so the reported latency suddenly and dramatically increases. Any further small increase in load would make the server appear completely unresponsive to an end user.

In contrast, Cloud Onload continues to deliver low latency with 3 million requests per second, and is actually trending towards even lower latency. The stable and low value for the 99th percentile of latency indicates low jitter and predictable performance.

A Cloud Onload profiles

This appendix contains the Cloud Onload profiles used for this benchmarking. See:

- The wrk-profile Cloud Onload profile on page 22
- The nginx-server Cloud Onload profile on page 23
- The haproxy Cloud Onload profiles on page 24.

These profiles, along with the scripts used for this benchmarking, are available on request from support@solarflare.com.

A.1 The wrk-profile Cloud Onload profile

The `wrk-profile.opf` Cloud Onload profile is as follows:

```
onload_set EF_SOCKET_CACHE_MAX 40000
onload_set EF_TCP_TCONST_MSL 1
onload_set EF_TCP_FIN_TIMEOUT 15
onload_set EF_HIGH_THROUGHPUT_MODE 1
onload_set EF_LOG_VIA_IOCTL 1
onload_set EF_NO_FAIL 1
onload_set EF_UDP 0

#ensure sufficient resources
onload_set EF_MAX_PACKETS 205000
onload_set EF_MAX_ENDPOINTS 400000
onload_set EF_FDTABLE_SIZE 8388608
onload_set EF_USE_HUGE_PAGES 2
onload_set EF_MIN_FREE_PACKETS 50000

#environment variable can overwrite
onload_set EF_LOAD_ENV 1

#spinning configuration
onload_set EF_POLL_USEC 100000
onload_set EF_SLEEP_SPIN_USEC 50
onload_set EF_EPOLL_SPIN 1

#scalable filters with clustering for outgoing connections
onload_set EF_SCALABLE_FILTERS 'any=rss:active'
onload_set EF_SCALABLE_FILTERS_ENABLE 1
onload_set EF_CLUSTER_NAME 'load'
onload_set EF_CLUSTER_SIZE 12 #needs to overwritten by environment

#shared local ports to improve rate of socket recycling
onload_set EF_TCP_SHARED_LOCAL_PORTS_MAX 28000
onload_set EF_TCP_SHARED_LOCAL_PORTS 28000
onload_set EF_TCP_SHARED_LOCAL_PORTS_PER_IP 1
```



```
onload_set EF_TCP_SHARED_LOCAL_PORTS_REUSE_FAST 1
onload_set EF_TCP_SHARED_LOCAL_PORTS_NO_FALLBACK 1

#epoll configuration
onload_set EF_UL_EPOLL 3
onload_set EF_EPOLL_MT_SAFE 1

#reduce transmit CPU load
onload_set EF_TX_PUSH 0
onload_set EF_PIO 0
onload_set EF_CTPIO 0

# Adjustments for potentially-lossy network environment
onload_set EF_TCP_INITIAL_CWND 14600
onload_set EF_DYNAMIC_ACK_THRESH 4
onload_set EF_TAIL_DROP_PROBE 1
onload_set EF_TCP_RCVBUF_MODE 1
```

A.2 The nginx-server Cloud Onload profile

The `nginx-server.opf` Cloud Onload profile is as follows:

```
onload_set EF_ACCEPTQ_MIN_BACKLOG 400
onload_set EF_SOCKET_CACHE_MAX 40000
onload_set EF_TCP_TCONST_MSL 1
onload_set EF_TCP_FIN_TIMEOUT 15
onload_set EF_TCP_SYNRECV_MAX 90000
onload_set EF_TCP_BACKLOG_MAX 400
onload_set EF_HIGH_THROUGHPUT_MODE 1
onload_set EF_LOG_VIA_IOCTL 1
onload_set EF_NO_FAIL 1
onload_set EF_UDP 0

#ensure sufficient resources
onload_set EF_MAX_PACKETS 205000
onload_set EF_MAX_ENDPOINTS 400000
onload_set EF_USE_HUGE_PAGES 2
onload_set EF_MIN_FREE_PACKETS 50000

#epoll configuration
onload_set EF_UL_EPOLL 3
onload_set EF_EPOLL_MT_SAFE 1

#don't use clustering when SO_REUSEPORT is set
onload_set EF_CLUSTER_IGNORE 1

#environment variable can overwrite
onload_set EF_LOAD_ENV 1

#spinning configuration
onload_set EF_POLL_USEC 100000
onload_set EF_SLEEP_SPIN_USEC 50
onload_set EF_EPOLL_SPIN 1

#reduce transmit CPU load
onload_set EF_TX_PUSH 0
```



```
onload_set EF_PIO 0
onload_set EF_CTPIO 0

# Adjustments for potentially-lossy network environment
onload_set EF_TCP_INITIAL_CWND 14600
onload_set EF_DYNAMIC_ACK_THRESH 4
onload_set EF_TAIL_DROP_PROBE 1
onload_set EF_TCP_RCVBUF_MODE 1
```

A.3 The haproxy Cloud Onload profiles

There are two haproxy Cloud Onload profiles.

- The balanced profile gives excellent throughput, with low latency. It has reduced CPU usage at lower traffic rates.
- The performance profile is latency focused. It constantly polls for network events to achieve the lowest latency possible, and so has higher CPU usage.

The differences between these profiles are minor, and are in the profile files. See:

- The haproxy-balanced profile on page 24
- The haproxy-performance profile on page 24.

The majority of the settings are common to both profiles, and are in separate shared files that each profile sources or includes. See:

- The haproxy-config profile fragment on page 25.
- The reverse-proxy-throughput profile fragment on page 27.

The haproxy-balanced profile

The `haproxy-balanced.opf` Cloud Onload profile is as follows:

```
. ${PROXY_CONFIG_DIR}/haproxy-config.opf-fragment
onload_import ${PROXY_CONFIG_DIR}/reverse-proxy-throughput.opf-fragment
```

The haproxy-performance profile

The `haproxy-performance.opf` Cloud Onload profile is as follows:

```
. ${PROXY_CONFIG_DIR}/haproxy-config.opf-fragment
onload_set EF_TX_PUSH 1
onload_set EF_SLEEP_SPIN_USEC 0
onload_import ${PROXY_CONFIG_DIR}/reverse-proxy-throughput.opf-fragment
```


The haproxy-config profile fragment

The haproxy-config`.opf-fragment` file, sourced by both the above profiles, is as follows:

```
## Tuning profile for haproxy with OpenOnload acceleration.
#
# User may supply the following environment variables:
#
#   PROXY_WORKERS        - the number of worker processes that haproxy is
#                          configured to use. Overrides value automatically
#                          detected from haproxy configuration
#

# For diagnostic output
module="haproxy-balanced profile"

# Parse the config file
read_file() {
  local setting
  eval "local worker_values=$(perl -ne 'print "$1 " if'"/^\s*nbproc\s+(\S+)/" $1)" # need
eval as the value can use shell variables

  for workers in $worker_values
  do
    setting=$workers
  done
  echo $setting
}

# Parse the config file or directory
read_file_or_dir() {
  local name="$1"
  local setting
  if [ -f $name ]
  then
    echo $(read_file "$name")
  elif [ -d $name ]
  then
    for file in $name/*.cfg
    do
      if [ -f $file ]
      then
        local possible=$(read_file "$file")
        if [ -n "$possible" ]
        then
          setting=$possible
        fi
      fi
    done
    echo $setting
  fi
}


# Scan all the config files which haproxy would use
determine_worker_processes() {
```

```
  local file
  local num=1 # use one as the default to match haproxy without nbproc configured

  # Look for a -f, -- or -C option
  local state="IDLE"
  for option in "$@"
  do
    if [ "$state" = "MINUS_f" ]
    then
      file=$option
      num=$(read_file_or_dir "$file")
      state="IDLE"
    elif [ "$state" = "MINUS_MINUS" ]
    then
      file=$option
      num=$(read_file_or_dir "$file")
    elif [ "$state" = "MINUS_C" ]
    then
      cd $option
      state="IDLE"
    elif [ "$option" = "-f" ]
    then
      state="MINUS_f"
    elif [ "$option" = "--" ]
    then
      state="MINUS_MINUS"
    elif [ "$option" = "-C" ]
    then
      state="MINUS_C"
    fi
  done

  echo $num
}

# Define the number of workers
calc_workers=$(determine_worker_processes "$@")
num_workers=${PROXY_WORKERS:-$calc_workers}
if ! [ -n "$num_workers" ]; then
  fail "ERROR: Environment variable PROXY_WORKERS is not set and worker count cannot be
determined from haproxy configuration"
fi
log "$module: configuring for $num_workers workers (from config appear to be
$calc_workers)"


onload_set EF_PIPE 0
```

The reverse-proxy-throughput profile fragment

The `reverse-proxy-throughput.opf-fragment` file, included by both the above profiles, is as follows:

```
# Enable epoll implementation that scales with large numbers of fds.
onload_set EF_UL_EPOLL 3

# Assert application use of epoll is multithread safe.
onload_set EF_EPOLL_MT_SAFE 1

# Enable clustering to spread connections over workers.
# Name will need to be overridden for each cluster if
# wanting to use multiple clusters.
onload_set EF_CLUSTER_SIZE "$num_workers"
onload_set EF_CLUSTER_NAME prox

# Force termination of orphaned stacks on restart.
onload_set EF_CLUSTER_RESTART 1

# Allow sharing of stacks in cluster by two processes
# to allow hot/seamless restart.
onload_set EF_CLUSTER_HOT_RESTART 1

# Enable scalable filters to avoid using a separate filter
# for each connection.  A proxy both accepts passive connections
# and makes active connections.
# This will need to be overridden to not use 'any' interface if
# wanting separate clusters on different interfaces.
onload_set EF_SCALABLE_FILTERS "any=rss:active:passive"

# Scalable filters mode for applications using master/worker
# hierarchy.
onload_set EF_SCALABLE_FILTERS_ENABLE 2

# Connections not accepted through scalable filters interface
# are refused.
onload_set EF_SCALABLE_LISTEN_MODE 1

# Enable shared local ports which allows Onload to recycle resources
# for active open connections more efficiently.
# A large number of shared local ports are created and the maximum
# is set to the same value.  I.e. all shared local ports are
# allocated on stack creation and not allocated later.
onload_set EF_TCP_SHARED_LOCAL_PORTS 570000
onload_set EF_TCP_SHARED_LOCAL_PORTS_MAX $EF_TCP_SHARED_LOCAL_PORTS

# EF_TCP_SHARED_LOCAL_PORTS_REUSE_FAST allows recycling ports
# immediately when CLOSED state is reached via LAST-ACK (i.e.
# when socket received FIN from server rather than sent FIN via
# close())
onload_set EF_TCP_SHARED_LOCAL_PORTS_REUSE_FAST 1

# Validate shared local ports are used by setting
# EF_TCP_SHARED_LOCAL_PORTS_NO_FALLBACK=1 which causes connect()
# to fail when shared local ports are not used.
onload_set EF_TCP_SHARED_LOCAL_PORTS_NO_FALLBACK 1
```



```
# Use a separate pool of shared local ports per local IP
onload_set EF_TCP_SHARED_LOCAL_PORTS_PER_IP 1
# Set the limit of the shared ports pool per IP/cluster
onload_set EF_TCP_SHARED_LOCAL_PORTS_PER_IP_MAX 32000

# How many more shared local ports to allocate if current pool
# exhausted.  Unused when all allocated at startup.
onload_set EF_TCP_SHARED_LOCAL_PORTS_STEP 2048

# Support lots of sockets and enable socket caching.
onload_set EF_MAX_ENDPOINTS 1000000
onload_set EF_SOCKET_CACHE_MAX $(( $EF_MAX_ENDPOINTS / 4 ))

# Enable spinning with sleep spin to reduce CPU load
onload_set EF_POLL_USEC 1000000
onload_set EF_SLEEP_SPIN_USEC 50

# Allocate plenty of packet memory and force hugepages.
onload_set EF_MAX_PACKETS $(( (180000*16) / $num_workers ))
onload_set EF_PREALLOC_PACKETS 1
onload_set EF_USE_HUGE_PAGES 2

# Tune TCP socket parameters.
onload_set EF_TCP_SYNRECV_MAX 1000000

# Disable low-latency sends to minimise CPU overheads.
onload_set EF_TX_PUSH 0
onload_set EF_PIO 0
onload_set EF_CTPIO 0

# Prevent spinning inside socket calls.  (Spinning will take place
# on epoll_wait polling).
onload_set EF_PKT_WAIT_SPIN 0
onload_set EF_TCP_RECV_SPIN 0
onload_set EF_TCP_SEND_SPIN 0
onload_set EF_TCP_CONNECT_SPIN 0
onload_set EF_TCP_ACCEPT_SPIN 0
onload_set EF_UDP_RECV_SPIN 0
onload_set EF_UDP_SEND_SPIN 0

# Forward packets arriving via scalable filter to the kernel
# when required (e.g. IGMP)
onload_set EF_KERNEL_PACKETS_BATCH_SIZE 1

# Adjustments for potentially-lossy network environment
# Use a minimum congestion window of 10 MSS
onload_set EF_TCP_INITIAL_CWND 14600

# Set how many unacked segments force ACK.  Increasing this
# will reduce network load but could result in peer needing to
```

```
# retransmit more data if network is lossy.
onload_set EF_DYNAMIC_ACK_THRESH 4

# Force enable tail drop probe to retransmit faster.
# (If not set, uses /proc/sys/net/ipv4/tcp_early_retrans
# which is also on by default)
onload_set EF_TAIL_DROP_PROBE 1

# Enable dynamically sized TCP receive buffers.
onload_set EF_TCP_RCVBUF_MODE 1
```


B Installation and configuration

This appendix describes how to install and configure the software distributions used for this benchmarking. See:

- Installing HAProxy on page 30
- Installing NGINX on page 31
- Installing wrk2 on page 32
- Installing Cloud Onload on page 33.

B.1 Installing HAProxy

This section describes how to install and configure HAProxy.

Installation

NOTE: For a reference description of how to install HAProxy, see the README file in the distribution, and the documentation at http://www.haproxy.org.

In summary:

1. If you already have an old HAProxy installation on your system, remove the old installation:

```
# rm -f /opt/haproxy
# rm -rf /opt/haproxy-1.9.7
```

2. Change to /opt (the parent directory of the HAProxy installation):

```
# cd /opt
```

3. Download the HAProxy tarball and unpack it:

```
# curl -s http://www.haproxy.org/download/1.9/src/haproxy-1.9.7.tar.gz | \
     tar xzf -
```

4. Make and install HAProxy:

```
# make -C /opt/haproxy-1.9.7 TARGET=linux2628
```

5. Create a soft link to the installed version of HAProxy

```
# ln -s /opt/haproxy-1.9.7 /opt/haproxy
```


B.2 Installing NGINX

This section describes how to install and configure NGINX.

Installation

NOTE: For a reference description of how to install NGINX, see https://docs.nginx.com/nginx/admin-guide/installing-nginx/installing-nginx-open-source/.

In summary:

1 If you already have an old NGINX installation on your system:

a) Back up your configs and logs:

```
# cp -a /etc/nginx /etc/nginx-plus-backup
# cp -a /var/log/nginx /var/log/nginx-plus-backup
```

b) Remove the old installation:

```
# rm -rf /opt/nginx
```

2 Create a new NGINX directory:

```
mkdir -p /opt/nginx
```

3 Change to a temporary directory:

```
cd $( mktemp -d )
```

4 Clone NGINX from its git repository:

```
git clone https://github.com/nginx/nginx .
```

5 Configure NGINX:

```
./auto/configure --prefix=/opt/nginx
```

6 Make and install NGINX:

```
make install
```

7 Check the NGINX binary version to ensure that you have NGINX installed correctly:

```
# nginx -v
nginx version: nginx/1.17
```

8 Start NGINX:

```
# systemctl start nginx
```

or just:

```
# nginx
```


9 Verify access to Web Server

B.3 Installing wrk2

This section describes how to install and configure `wrk2`.

Installation

NOTE: For a reference description of how to install `wrk2`, see: https://github.com/giltene/wrk2/wiki/Installing-wrk2-on-Linux.

In summary:

1 If the build tools are not already installed, install them:

```
# yum groupinstall 'Development Tools'
```

2 If the OpenSSL dev libs are not already installed, install them:

```
# yum install -y openssl-devel
```

3 If git is not already installed, install it:

```
# yum install -y git
```

4 Create a directory to hold wrk2:

```
# mkdir -p Onload_Testing/WRK2
# cd Onload_Testing/WRK2
```

5 Use git to download wrk2:

```
# git clone https://github.com/giltene/wrk2.git
```

6 Build wrk2:

```
# cd wrk2
# make
```

7 Copy the wrk2 executable to a location on your PATH. For example:

```
# cp wrk2 /usr/local/bin
```

B.4 Installing Cloud Onload

For instructions on how to install and configure Cloud Onload, refer to the *Onload User Guide* (SF-104474-CD). This is available from https://support.solarflare.com/.