Pawel Chmielinski Follow
WebOps Engineer at @KainosSoftware
Feb 28, 2017 · 10 min read

Source: Unsplash.com

Tuning your Linux kernel and HAProxy instance for high loads

Configuration tips that enable serving a higher number of concurrent users

If you have ever configured a loadbalancer like HAProxy or a webserver like Nginx or Apache to handle a high number of concurrent users, then you might have discovered that there are quite a few tweaks required in order to achieve the desired effects. Could you recite all of them from the top of your head? If not, don't worry—this article has got you covered.

We will have a look at two types of tweaks. First ones are Linux kernel tweaks, which are common regardless if you're using HAProxy, Nginx, Apache or other webservers. The second type covers HAProxy specific configuration.

Kernel tweaks

Number of open files

Why should we care about open files when handling web traffic? It's simple—every incoming or outgoing connection needs to open a socket and each socket is a file on a Linux system. If you're configuring a webserver serving static content from the local filesystem, then each connection will result in one open socket. However, if you're configuring a loadbalancer serving content from backend servers then each incoming connections will open a minimum of two sockets, or even more, depending on the loadbalancing configuration.

It's important that you configure the maximum number of open files, as the default number is pretty low. On Ubuntu 16.04 it's up to 4096 open files per process, which is not an awful lot. You know that you have hit the limit if you see `Too many files open` lines in your logs.

Now, there are two ways to configure max open files, depending on whether your distribution uses systemd or not. Most tutorials found on Google assume systemd is not used, in which case the number of open files can be set by editing `/etc/security/limits.conf` (assuming pam_limits is used for daemon processes, see this answer for a more thorough explanation). A sample config to set both the soft and hard limits for every user on the system to 100k would look like this:

```
* soft nofile 100000
* hard nofile 100000
root soft nofile 100000
root hard nofile 100000
```

Afterwards restart your webserver/loadbalancer to apply the changes. You can check if it worked by issuing:

```
cat /proc/<PID_of_webserver>/limits
```

If the daemon process doesn't use pam_limits, it won't work. A bit hacky workaround is to use `ulimit 100000` directly in the init script or any of the files sourced inside it, like `/etc/default/<service_name>` on Ubuntu.

If you're on a system that uses systemd you will find that setting `limits.conf` doesn't work as well. That's because systemd doesn't use the `/etc/security/limits.conf` at all, but instead uses it's own configuration to determine the limits. However, keep in mind that even with systemd, `limits.conf` is still useful when running a long-running process from within a user shell, as the user limits still use the old config file. These can be displayed by issuing:

```
ulimit -a
```

Okay, how do we configure maximum open files for systemd? The answer is to override the configuration for a specific service. We do this by placing a file in `/etc/systemd/system/<service_name>.service.d/override.conf`. The file content could look like this to set a 100k max open files limit:

```
[Service]
LimitNOFILE=100000
```

After the change we have to reload systemd configuration and restart our service:

```
systemctl daemon-reload
systemctl restart <service_name>
```

To make sure that the override worked use the following:

```
systemctl cat <service_name>
cat /proc/<PID>/limits
```

This should work fine for Apache and Nginx, but if you're running HAProxy, you're in for a surprise. When we restart HAProxy, there are actually 3 processes spawned and only the top-level one (`/usr/sbin/haproxy-systemd-wrapper`) has our limits applied! That's because HAProxy configured it's open files limit automatically based on the `maxconn` value in `haproxy.conf`. We will have a look at this parameter further down the article.

There are two other values that relate to maximum open files—global values for the system. These can be checked by issuing:

```
sysctl fs.file-max
sysctl fs.nr_open
```

`fs.file-max` determines the maximum number of files in total that can be opened on the system. `fs.nr-open` determines the maximum value that `fs.file-max` can be configured to. On modern distributions both are configured to high values, but if you find that's not the case for your system, then feel free to tweak them as well. In any case make sure that they are configured much higher the the value used in `limits.conf` and/or `systemd`, because we don't want a single process to be able to block the operating system from opening files.

Conntrack

In production systems there's a high probability that you're using iptables/ufw/firewalld firewall on your system. These services all use the `nf_conntrack` kernel module. The module is autoloaded when starting the service and adds some additional kernel parameters. The one we are most interested in is called `net.netfilter.nf_conntrack_max` and determines the maximum number of connections that the kernel module will track. If the number of connections exceed this number then new connections will be dropped and you will see the following log message:

```
nf_conntrack: table full, dropping packet
```

It's recommended not to tweak `nf_conntrack_max` manually, but indirectly, by setting `nf_conntrack_buckets`. The value for

`nf_conntrack_max` will be automatically set to 8* `nf_conntrack_buckets`. The standard way of configuring a kernel parameter would be to modify `/etc/sysctl.conf` or `/etc/sysctl.d/<config_name>.conf` and execute `sysctl -p`. However, in this case it might throw errors, since the parameter exists only if the `nf_conntrack` module is already loaded. A better way is to configure it on module load, by setting it in `/etc/modprobe.d/nf_conntrack.conf`. A sample configuration would look like this:

```
options nf_conntrack hashsize=100000
```

and would result in `net.netfilter.nf_conntrack_max` taking a value of 800k. This will only take effect after the module is reloaded. For example, on Ubuntu 16.04 with ufw, the commands to do this are:

```
systemctl stop ufw
modprobe -rv nf_conntrack
systemctl start ufw
```

You might get a `modprobe: FATAL: Module nf_conntrack is in use.` error on the second command. In order to fix this, find out what modules are using `nf_conntrack` with `lsmod | grep nf_conntrack` and unload all of them at once. In my case, the actual command looked like this: `modprobe -rv nf_nat_ftp nf_conntrack_netbios_ns nf_nat xt_conntrack nf_conntrack_broadcast nf_conntrack_ftp nf_conntrack_ipv4 nf_conntrack_ipv6 nf_conntrack`

TCP parameters

All the values below should be configured in `/etc/sysctl.conf` or `/etc/sysctl.d/<config_name>.conf` and enabled by executing `sysctl -p`. The values used assume the server has a lot of spare memory—in my case each server has 4GB RAM.

```
net.ipv4.tcp_max_syn_backlog = 100000
net.core.somaxconn = 100000
net.core.netdev_max_backlog = 100000
```

First, a very short reminder about how a TCP three-way handshake works:

1. Client sends a SYN packet to the server—this indicates an intention to initiate a TCP connection with the server
2. Server responds with a SYN-ACK packet—server accepts the connection
3. Client responds with an ACK packet—client acknowledges that the server accepted the connection. The connection is now established.

Now to the values:

1. `net.ipv4.tcp_max_syn_backlog` —how many half-open connections for which the client has not yet sent an ACK response can be kept in the queue. The default `net.ipv4.tcp_max_syn_backlog` is set to 128 on my Ubuntu 16.04. We set it to a higher value, because if the queue is full then legitimate clients won't be able to connect—they would get a *connection refused* error, just as if the target port was closed. This queue will be mostly filled up by clients, which are slow to send the ACK packet (or attackers conducting a SYN flood attack). However, there also other scenarios—specifically, if HAProxy's global `maxconn` value is reached, it will stop responding and the requests will wait in this queue until a socket is free. This will cause a delay on the client's side, but we assume it's better than a refused connection. Do note, however, that this might NOT be preferable if this server is behind a loadbalancer—in that case we might prefer to refuse the connection, so that the loadbalancer can immediately pick a different server.
2. `net.core.somaxconn` —the maximum value that `net.ipv4.tcp_max_syn_backlog` can take. Higher values are silently truncated to the value indicated by `somaxconn` . In older kernels this value could not be higher than 65535.
3. `net.core.netdev_max_backlog` —the maximum number of packets in the receive queue that passed through the network interface and are waiting to be processed by the kernel. The default is set to 1000 on Ubuntu 16.04 and probably should be fine on a 1G network. However, if you're using a 10G connection, then it's useful to increase this queue's size. If the queue is full the incoming packets will be silently dropped.

If you have read other tuning guides available on the Internet, you might have also seen recommendations to tweak `net.ipv4.tcp_rmem` and `net.ipv4.tcp_wmem`, which control the sizes of the receive and send buffers. However, these days the kernel does a good job to self-regulate these buffers, so it's unlikely the defaults need to be changed.

TCP port range

This is relevant if you're configuring a loadbalancer or a reverse proxy. In this scenario you may run into an issue called *TCP source port exhaustion.* If you're not using some sort of connection pooling or multiplexing, then in general each connection from a client to the loadbalancer also opens a related connection to one of the backends. This will open a socket on the loadbalancer's system. Each socket is identified by the following 5-tuple:

- Protocol (we assume here, that this is always TCP)
- Source IP
- Source port
- Destination IP
- Destination port

You cannot have 2 sockets that are identified by the same 5-tuple on the system. The problem is TCP only has 65535 ports available. So in a scenario where the reverse proxy only has a single IP address and is proxying to a single backend on a single IP and port, we're looking at 1*1*65535*1*1 unique combinations. The actual number is actually lower, because by default Linux will only use range 32768-60999 as the source ports for outgoing connections. We can increase this, but the first 1024 are reserved, so in the end we set it to a range of 1024–65535. This is done with `sysctl`, using the same process as described before—writing the value to `/etc/sysctl.conf` or `/etc/sysctl.d/<config>.conf` and executing `sysctl -p`.

```
net.ipv4.ip_local_port_range=1024 65535
```

With an effective number of ports equal to 64511 ports, we have more breathing room, but in certain situations it might still not be enough. In that case you can look into increasing the number of other items in the 5-tuple:

- Configure more than one IP on the loadbalancer system. Make sure the loadbalancer is configured to also use these additional IPs
- Configure the destination backend to listen on multiple IPs and configure the loadbalancer to connect to these IPs
- Configure the destination to listen on multiple ports, if possible

Increasing the number of source IPs on the loadbalancer is most likely the easiest option available. In HAProxy this is possible by configuring the source option in the server line of the backend. If the backend is on the local system, it might be easier to tweak the destination IPs, as every request to any IP in the 127.0.0.0/8 pool will go to localhost without any additional configuration required.

HAProxy tweaks

Number of processes

By default HAProxy uses only a single process to serve requests. If running on a system with multiple cores, you might find out during high load periods that a single core has 100% CPU utilisation while others are mostly idle. This can be rectified by increasing the number of running processes. Assuming HAProxy is placed on a system with 4 cores, the config would look like the following:

```
global
   nbproc  4
   cpu-map  1 1
   cpu-map  2 2
   cpu-map  3 3
   cpu-map  4 4

(...)
```

`nbproc` directive will configure the number of processes, which in general should be equal to the number of cores on the system. `cpu-map` will assign a specific CPU to the running process, so that each process has it's own.

When increasing the number of HAProxy processing bear in mind this has certain consequences, which can be found in the documentation. The gist of it is that many configuration options work **per-process,** rather than on a global level, eg. maxconn, admin socket, stick-tables.

Maxconn

The default number of maximum connections is provided during HAProxy's compilation. The package available on my Ubuntu 16.04 was compiled with 2000 maximum connections. This is quite low, but can be changed in the configuration. First thing you need to know, however, is that there are actually three maxconn values in the config. A sample config might look like this:

```
global
    maxconn 100000
    (...)

frontend frontend1
    bind *:80
    mode http
    maxconn 100000
    default_backend backend1
    (...)

backend backend1
    server frontend-1 192.168.1.1:80 maxconn=200
    server frontend-2 192.168.1.2:80 maxconn=200
    balance roundrobin
    (...)
```

There's a very good answer on Stack Overflow illustrating how these values work, which I recommend reading. The short version is that both the global and the frontend maxconn values are by default equal to 2000, so you have to configure both, and they are configured **per-process.** So with `nbproc` configured to 4 and global `maxconn` set to 100k your HAProxy server will accept 400k connections. The remainder won't be rejected, but instead queued in the kernel.

There's also a per-server maxconn, which by default is not limited. This configuration option allows you to have a smaller number of active connections to each backend server. The rest gets queued in HAProxy internal queues. With a smaller number of connections reaching the servers, every request will get processed more quickly, so even though requests get queued on HAProxy their overall journey duration might actually be shorter. See this article to find some practical examples.

SSL cachesize

This is an important configuration value to tweak if your HAProxy instance is configured to serve SSL. By default the cache size is equal to

20k and the cache is shared between processes. Each SSL connection will create an entry in the cache. If you have more connections than the cache size, the oldest entry will get purged. In practice this means that if you have more connecting concurrent users than this value, they will all keep reissuing CPU-intensive SSL handshakes, possibly causing sudden performance degradation once you hit this number. It's best to configure this value to a much higher number, keeping in mind that each entry requires ~200 bytes of memory. Sample config:

```
global
    tune.ssl.cachesize 1000000
    (...)
```

With a value set to 1 million we thus need at least 200MB of memory to hold the cache.

Conclusion

The configuration options presented in this article remove the most common bottlenecks when serving high numbers of concurrent users. Using these recommendations it's most likely your next bottleneck will be the CPU or the network bandwidth, rather than a misconfigured configuration value. Thank you for reading and please let me know in the comments if I have missed anything.