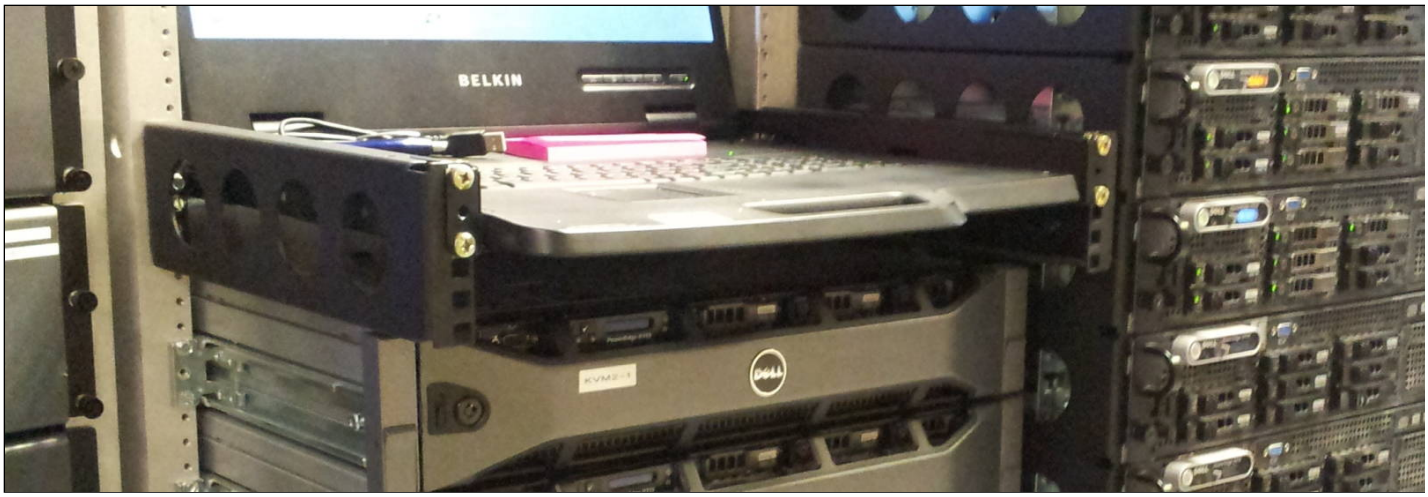

Performance Tuning on Linux — TCP

Tune TCP

We want to improve the performance of TCP applications. Some of what we do here can help our public services, predominantly HTTP/HTTPS, although performance across the Internet is limited by latency and congestion completely outside our control. But we do have complete control within our data center, and we must optimize TCP in order to optimize NFS and other internal services.

We will first provide adequate memory for TCP buffers. Then we will disable two options that are helpful on WAN links with high latency and variable bandwidth but degrade performance on fast LANs. Finally, we will increase the initial congestion window sizes.

Memory Management for TCP

Kernel parameters for TCP are set in `/proc/sys/net/core/[rw]mem*` and `/proc/sys/net/ipv4/tcp*`, those apply for both IPv4 and IPv6. As with all the categories we've seen so far, only a few of the available kernel parameters let us make useful changes. Leave the rest alone.

Parameters `core/rmem_default` and `core/wmem_default` are the default receive and send tcp buffer sizes, while `core/rmem_max` and `core/wmem_max` are the maximum receive and send buffer sizes that can be set using `setsockopt()`, in bytes.

Parameters `ipv4/tcp_rmem` and `ipv4/tcp_wmem` are the amount of memory in bytes for read (receive) and write (transmit) buffers per open socket. Each contains three numbers: the minimum, default, and maximum values.

Parameter `tcp_mem` is the amount of memory in 4096-byte pages totaled across all TCP applications. It contains three numbers: the minimum, pressure, and maximum. The pressure is the threshold at which TCP will start to reclaim buffer memory to move memory use down toward the minimum. You want to avoid hitting that threshold.

```
# grep . /proc/sys/net/ipv4/tcp*mem
/proc/sys/net/ipv4/tcp_mem:181419  241895  362838
/proc/sys/net/ipv4/tcp_rmem:4096    87380   6291456
/proc/sys/net/ipv4/tcp_wmem:4096    16384   4194304
```

Increase the default and maximum for `tcp_rmem` and `tcp_wmem` on servers and clients when they are on either a 10 Gbps LAN with latency under 1 millisecond, or communicating over high-latency low-speed WANs. In those cases their TCP buffers may fill and limit throughput, because the TCP window size can't be made large enough to handle the delay in receiving ACK's from the other end. IBM's High Performance Computing page recommends 4096 87380 16777216.

Then, for `tcp_mem`, set it to twice the maximum value for `tcp_[rw]mem` multiplied by the maximum number of running network applications divided by 4096 bytes per page.

Increase `rmem_max` and `wmem_max` so they are at least as large as the third values of `tcp_rmem` and `tcp_wmem`.

Calculate the **bandwidth delay product,** the total amount of data in transit on the wire, as the product of the bandwidth in bytes per second multiplied by the round-trip delay time in seconds. A 1 Gbps LAN with 2 millisecond round-trip delay means 125 Mbytes per second times 0.002 seconds or 250 kbytes.

If you don't have buffers this large on the hosts, senders have to stop sending and wait for an acknowledgement, meaning that the network pipe isn't kept full and we're not using the full bandwidth. **Increase the buffer sizes as the bandwidth delay product increases.** However, **be careful.** The bandwidth delay product is the ideal, although you can't really

measure how it fluctuates. If you provide buffers significantly larger than the bandwidth delay product for connections outbound from your network edge, you are just contributing to buffer congestion across the Internet without making things any faster for yourself.

Internet congestion will get worse as Windows XP machines are retired. XP did no window scaling, so it contributed far less to WAN buffer saturation.

TCP Options

TCP Selective Acknowledgement (TCP SACK), controlled by the boolean `tcp_sack`, allows the receiving side to give the sender more detail about lost segments, reducing volume of retransmissions. This is useful on high latency networks, but disable this to improve throughput on high-speed LANs. Also disable `tcp_dsack`, if you aren't sending SACK you certainly don't want to send duplicates! Forward Acknowledgement works on top of SACK and will be disabled if SACK is.

There is an option `tcp_slow_start_after_idle` which causes communication to start or resume gradually. This is helpful *if* you are on a variable speed WAN like 3G or 4G (LTE) mobile network. But on a LAN or across a fixed-bandwidth WAN you want the connection to start out going as fast as it can.

There are several TCP congestion control algorithms, they are loaded as modules and `/proc/sys/net/ipv4/tcp_available_congestion_control` will list the currently loaded modules. They are designed to quickly recover from packet loss on high-speed WANs, so this may or may not be of interest to you. Reno is the TCP congestion control algorithm used by most operating systems. To learn more about some of the other choices:

BIC | CUBIC | High Speed TCP | Scalable TCP | TCP Low Priority | TCP Veno | YeAH TCP | TCP Illinois | DataCenter TCP (DCTCP)

Use `modprobe` to load the desired modules and then `echo` or `sysctl` to place the desired option into `tcp_congestion_control`.

Setting Kernel Parameters

The following file `/etc/sysctl.d/02-netIO.conf` will be used by `sysctl` at boot time.

```
### /etc/sysctl.d/02-netIO.conf
### Kernel settings for TCP

# Provide adequate buffer memory.
# rmem_max and wmem_max are TCP max buffer size
# settable with setsockopt(), in bytes
# tcp_rmem and tcp_wmem are per socket in bytes.
# tcp_mem is for all TCP streams, in 4096-byte pages.
# The following are suggested on IBM's
# High Performance Computing page
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.core.rmem_default = 16777216
net.core.wmem_default = 16777216
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 87380 16777216
# This server might have 200 clients simultaneously, so:
#   max(tcp_wmem) * 2 * 200 / 4096
net.ipv4.tcp_mem = 1638400 1638400 1638400

# Disable TCP SACK (TCP Selective Acknowledgement),
# DSACK (duplicate TCP SACK), and FACK (Forward Acknowledgement)
net.ipv4.tcp_sack = 0
net.ipv4.tcp_dsack = 0
net.ipv4.tcp_fack = 0

# Disable the gradual speed increase that's useful
# on variable-speed WANs but not for us
net.ipv4.tcp_slow_start_after_idle = 0
```

Tune Initial Window Size

The header on every TCP segment contains a 16-bit field advertising its current receive window size, 0 through 65,535, and a 32-bit field reporting the acknowledgement number. This means that everything up to the acknowledgement position in the stream has been received, and the receiver has enough TCP buffer memory for the sender to transmit up to the window size

in bytes beyond the latest acknowledgement.

The idea is to more efficiently stream data by using buffers along the way to store segments and acknowledgements "in flight" in either direction. Across a fast switched LAN the improvement comes from allowing the receiver to keep filling the buffer while doing something else, putting off an acknowledgement packet until the receive buffer starts to fill.

Increases in the initial TCP window parameters can significantly improve performance. The *initial receive window size* or **initrwnd** specifies the receive window size advertised at the beginning of the connection, in segments, or zero to force Slow Start.

The *initial congestion window size* or **initcwnd** limits the number of TCP segments the server is willing to send at the beginning of the connection before receiving the first ACK from the client, regardless of the window size the client advertise. The server might be overly cautious, only sending a few kbytes and then waiting for an ACK because its initial congestion window is too small. It will send the smaller of the receiver's window size and the server's initcwnd. We have to put up with whatever the client says, but we can crank up the initcwnd value on the server and usually make for a much faster start.

Researchers at Google studied this. Browsers routinely open many connections to load a single page and its components, each of which will otherwise start slow. They recommend increasing initcwnd to at least 10.

CDN Planet has an interesting and much simpler article showing that increasing initcwnd to 10 cuts the total load time for a 14 kbyte file to about half the original time. They also found that many content delivery networks use an initcwnd of 10 and some set it even higher.

The initrwnd and initcwnd are specified in the routing table, so you can tune each route individually. If specified, they apply to all TCP connections made via that route.

First, look at the routing table. Let's use this simple example. This server has an Internet-facing connection on enp0s2 and is connected to an internal LAN through enp0s3. Let's say it's an HTTP/HTTPS server on the Internet side, and an client of NFSv4 over TCP on the internal side.

```
# ip route show
default via 24.13.158.1 dev enp0s2
10.1.1.0/24 dev enp0s3  proto kernel  scope link  src 10.1.1.100
24.13.158.0/23 dev enp0s2  proto kernel  scope link  src 24.13.159.33
```

Now we will modify the routes to specify both initcwnd and initrwnd of 10 segments:

```
# ip route change default via 24.13.158.1 dev enp0s2 initcwnd 10 initrwnd 10
# ip route change 10.1.1.0/24 dev enp0s3 proto kernel scope link src 10.1.1.100 initcwnd
10 initrwnd 10
# ip route change 24.13.158.0/23 dev enp0s2 proto kernel scope link src 24.13.159.33
initcwnd 10 initrwnd 10
# ip route show
default via 24.13.158.1 dev enp0s2  initcwnd 10  initrwnd 10
10.1.1.0/24 dev enp0s3  proto kernel  scope link  src 10.1.1.100  initcwnd 10  initrwnd 10
24.13.158.0/23 dev enp0s2  proto kernel  scope link  src 24.13.159.33  initcwnd 10
initrwnd 10
```

The appropriate commands could be added to `/etc/rc.d/rc.local` for application at boot time.

Changes to the TCP window size also affect UDP buffering. On nets faster than 1 Gbps make sure that your applications use `setsockopt()` to request larger `SO_SNDBUF` and `SO_RCVBUF`.

Going Further

Calomel has a great page on performance tuning, although it is *specific to FreeBSD.* But read through their descriptions of how to tune BSD kernel parameters, and apply what you can to an analysis and tuning of your server.

And next...

Now that we have tuned the networking protocols, we can tune NFS file service running on top of TCP.

RAM and disk storage

RAM I/O speeds, disk controllers and interfaces, rotating versus solid-state disks, performance versus power sa

Disk I/O

Elevator sorting algorithms, tuning the scheduler, disk memory management, measuring disk I/O

File Systems