Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

TCP Tuning

http://www.linux-admins.net/2010/09/linux-tcp-tuning.html

```
general config file:
$ cat /etc/sysctl.conf
```

The default maximum Linux TCP buffer sizes are way too small. TCP memory is calculated automatically based on system memory; you can find the actual values by typing the following commands:

```
$ cat /proc/sys/net/ipv4/tcp_mem
```

The default and maximum amount for the receive socket memory:

```
$ cat /proc/sys/net/core/rmem_default
$ cat /proc/sys/net/core/rmem_max
```

The default and maximum amount for the send socket memory:

```
$ cat /proc/sys/net/core/wmem_default
$ cat /proc/sys/net/core/wmem_max
```

The maximum amount of option memory buffers:

```
$ cat /proc/sys/net/core/optmem_max
```

Tune values

Set the max OS send buffer size (wmem) and receive buffer size (rmem) to 12 MB for queues on all protocols. In other words set the amount of memory that is allocated for each TCP socket when it is opened or created while transferring files:

WARNING! The default value of rmem_max and wmem_max is about 128 KB in most Linux distributions, which may be enough for a low-latency general purpose network environment or for apps such as DNS / Web server. However, if the latency is large, the default size might be too small. Please note that the following settings going to increase memory usage on your server.

```
# echo 'net.core.wmem_max=12582912' >> /etc/sysctl.conf
# echo 'net.core.rmem_max=12582912' >> /etc/sysctl.conf
```

You also need to set minimum size, initial size, and maximum size in bytes:

```
# echo 'net.ipv4.tcp_rmem= 10240 87380 12582912' >>
/etc/sysctl.conf
# echo 'net.ipv4.tcp_wmem= 10240 87380 12582912' >>
/etc/sysctl.conf
```

Turn on window scaling which can be an option to enlarge the transfer window:

```
# echo 'net.ipv4.tcp_window_scaling = 1' >> /etc/sysctl.conf
```

Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

Enable timestamps as defined in RFC1323:

```
# echo 'net.ipv4.tcp_timestamps = 1' >> /etc/sysctl.conf
```

Enable select acknowledgments:

```
# echo 'net.ipv4.tcp_sack = 1' >> /etc/sysctl.conf
```

By default, TCP saves various connection metrics in the route cache when the connection closes, so that connections established in the near future can use these to set initial conditions. Usually, this increases overall performance, but may sometimes cause performance degradation. If set, TCP will not cache metrics on closing connections.

```
# echo 'net.ipv4.tcp_no_metrics_save = 1' >> /etc/sysctl.conf
```

Set maximum number of packets, queued on the INPUT side, when the interface receives packets faster than kernel can process them.

```
# echo 'net.core.netdev_max_backlog = 5000' >> /etc/sysctl.conf
```

Now reload the changes:

```
# sysctl -p
```

Use tcpdump to view changes for eth0:

```
# tcpdump -ni eth0
```

```
In particular, the use of SYN cookies allows a server to avoid dropping
connections when the SYN queue fills up. Instead, the server behaves as if the
SYN queue had been enlarged.

# this gives the kernel more memory for tcp

# which you need with many (100k+) open socket connections



net.ipv4.tcp_syncookies = 1





net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216

net.ipv4.tcp_mem = 50576   64768   98152
```

```
!!!OBSOLETE!!!
But then how and why do these connection tracking happen. As the
KB explains, ip_conntrack is an iptables module that maintains a
list of connections through router. Each connection tracking entry
contains defined characteristics of the packet, including the
source and destination IP address and port numbers. Entries are
stored in a hash table with a fixed size.

net.ipv4.netfilter.ip_conntrack_max = 1048576
```

Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

```
sudo modprobe nf_conntrack

nf_conntrack_max

/proc/sys/net$ cat nf_conntrack_max
```

TCP_FIN_TIMEOUT - This setting determines the time that must elapse before TCP/IP can release a closed connection and reuse its resources. During this TIME_WAIT state, reopening the connection to the client costs less than establishing a new connection. By reducing the value of this entry, TCP/IP can release closed connections faster, making more resources available for new connections. Addjust this in the presense of many connections sitting in the TIME_WAIT state:

```
[root@server:~]# echo 30 > /proc/sys/net/ipv4/tcp_fin_timeout
```

TCP_KEEPALIVE_INTERVAL - This determines the wait time between isAlive interval probes. To set:

```
[root@server:~]# echo 30 > /proc/sys/net/ipv4/tcp_keepalive_intvl
```

TCP_KEEPALIVE_PROBES - This determines the number of probes before timing out. To set:

```
[root@server:~]# echo 5 > /proc/sys/net/ipv4/tcp_keepalive_probes
```

TCP_TW_RECYCLE - This enables fast recycling of TIME_WAIT sockets. The default value is 0 (disabled). Should be used with caution with loadbalancers.

```
[root@server:~]# echo 1 > /proc/sys/net/ipv4/tcp_tw_recycle
```

TCP_TW_REUSE - This allows reusing sockets in TIME_WAIT state for new connections when it is safe from protocol viewpoint. Default value is 0 (disabled). It is generally a safer alternative to tcp_tw_recycle

```
[root@server:~]# echo 1 > /proc/sys/net/ipv4/tcp_tw_reuse
```

Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

The parameter on line 1 is the maximum number of remembered connection requests, which still have not received an acknowledgment from connecting clients.
The parameter on line 2 determines the number of SYN+ACK packets sent before the kernel gives up on the connection. To open the other side of the connection, the kernel sends a SYN with a piggybacked ACK on it, to acknowledge the earlier received SYN. This is part 2 of the three-way handshake.
And lastly on line 3 is the maximum number of TCP sockets not attached to any user file handle, held by system. If this number is exceeded orphaned connections are reset immediately and warning is printed.

If you are often dealing with SYN floods the following tunning can be helpful:

```
[root@host1 ~]# sysctl -w net.ipv4.tcp_max_syn_backlog="16384"
[root@host1 ~]# sysctl -w net.ipv4.tcp_synack_retries="1"
[root@host1 ~]# sysctl -w net.ipv4.tcp_max_orphans="400000"
```

//TO READ
Warning on Large MTUs: If you have configured your Linux host to use 9K MTUs, but the connection is using 1500 byte packets, then you actually need 9/1.5 = 6 times more buffer space in order to fill the pipe. In fact some device drivers only allocate memory in power of two sizes, so you may even need 16/1.5 = 11 times more buffer space!

And finally a warning for both 2.4 and 2.6: for very large BDP paths where the TCP window is > 20 MB, you are likely to hit the Linux SACK implementation problem. If Linux has too many packets in flight when it gets a SACK event, it takes too long to located the SACKed packet, and you get a TCP timeout and CWND goes back to 1 packet. Restricting the TCP buffer size to about 12 MB seems to avoid this problem, but clearly limits your total throughput. Another solution is to disable SACK.

Starting with Linux 2.4, Linux implemented a sender-side autotuning mechanism, so that setting the optimal buffer size on the sender is not needed. This assumes you have set large buffers on the receive side, as the sending buffer will not grow beyond the size of the receive buffer.

However, Linux 2.4 has some other strange behavior that one needs to be aware of. For example: The value for ssthresh for a given path is cached in the routing table. This means that if a connection has has a retransmission and reduces its window, then all connections to that host for the next 10 minutes will use a reduced window size, and not even try to increase its window. The only way to disable this behavior is to do the following before all new connections (you must be root):

Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

```
[root@server1 ~] # sysctl -w net.ipv4.route.flush=1
```

```
[root@host1 ~]# sysctl fs.file-nr
fs.file-nr = 197600 0 3624009
```

The first value (197600) is the number of allocated file handles.

The second value (0) is the number of unused but allocated file handles. And the third value (3624009) is the system-wide maximum number of file handles. It can be increased by tuning the following kernel parameter:

```
[root@host1 ~]# echo 10000000 > /proc/sys/fs/file-max
```

To see how many file descriptors are being used by a process you can use one of the following:

```
[root@host1 ~]# lsof -a -p 28290
[root@host1 ~]# ls -l /proc/28290/fd | wc -l
```

The 28290 number is the process id.

An finally if you are using stateful iptable rules the nf_conntrack kernel module might run out of memory for connection tracking and an error will be logged: nf_conntrack: table full, dropping packet

In order to raise that limit, therefore allocate more memory, you need to calculate how much RAM each connection uses. You can get that information from the proc file /proc/slabinfo.

The nf_conntrack entry show the active entries, and how big each object is, and how many fit in a slab (each slab fits in one or more kernel page, usually 4K if not using hugepages). Accounting for the overhead of the kernel page size you can see from the slabinfo that each nf_conntrack object takes about 316 bytes (this will differ on different systems). So to track 1M connections you'll need to allocate roughly 316 MB of memory.

```
[root@host1 ~]# sysctl net.netfilter.nf_conntrack_count
[root@host1 ~]# sysctl net.netfilter.nf_conntrack_max
[root@host1 ~]# sysctl -w net.netfilter.nf_conntrack_max=1000000
[root@host1 ~]# echo 250000 > /sys/module/nf_conntrack/parameters/hashsize # hashsize = nf_conntrack_max / 4 #there is no other way!
```

```
# recommended for CentOS7/Debian8 hosts
net.core.default_qdisc = fq
```

Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

//to do : what is jumbo frames !

http://phoenixframework.org/blog/the-road-to-2-million-websocket-connections

The First 1k Connections Again!

I set up a machine remotely and attempted benchmarking again. This time I was getting 1k connections, but at least Tsung didn't crash. The reason for this was the system-wide resource limit was being reached. To verify this I ran `ulimit -n` which returned `1024` which would explain why I could only get 1k connections.

From this point onwards the following configuration was used. This configuration took us all the way to 2 million connections.

```
sysctl -w fs.file-max=12000500
sysctl -w fs.nr_open=20000500
ulimit -n 20000000
sysctl -w net.ipv4.tcp_mem='10000000 10000000 10000000'
sysctl -w net.ipv4.tcp_rmem='1024 4096 16384'
sysctl -w net.ipv4.tcp_wmem='1024 4096 16384'
sysctl -w net.core.rmem_max=16384
sysctl -w net.core.wmem_max=16384
```

Setting up ulimit:

http://posidev.com/blog/2009/06/04/set-ulimit-parameters-on-ubuntu/

```
# recommended for hosts with jumbo frames enabled
net.ipv4.tcp_mtu_probing=1
```

`http://www.lognormal.com/blog/2012/09/27/linux-tcpip-tuning/`

TIME_WAIT state

TCP connections go through various states during their lifetime. There's the handshake that goes through multiple states, then the `ESTABLISHED` state, and then a whole bunch of states for either end to terminate the connection, and finally a `TIME_WAIT` state that lasts a really long time. If you're interested in all the states, read through the netstat man page, but right now the only one we care about is the `TIME_WAIT` state, and we care about it mainly because it's so long.

By default, a connection is supposed to stay in the `TIME_WAIT` state for twice the . Its purpose is to make sure any lost packets that arrive after a connection is closed do not confuse the TCP subsystem (the full details of this are beyond the scope of this article, but ask me if you'd like

Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

details). The default `msl` is 60 seconds, which puts the default `TIME_WAIT timeout` value at 2 minutes. Which means you'll run out of available ports if you receive more than about 400 requests a second, or if we look back to how nginx does proxies, this actually translates to 200 requests per second. Not good for scaling.

We fixed this by setting the timeout value to 1 second.

I'll let that sink in a bit. Essentially we reduced the timeout value by 99.16%. This is a huge reduction, and not to be taken lightly. Any documentation you read will recommend against it, but here's why we did it.

Again, remember the point of the `TIME_WAIT` state is to avoid confusing the transport layer. The transport layer will get confused if it receives an out of order packet on a currently established socket, and send a reset packet in response. The key here is the term *established socket*. A socket is a tuple of 4 terms. The source and destination IPs and ports. Now for our purposes, our server IP is constant, so that leaves 3 variables.

Our port numbers are recycled, and we have 47535 of them. That leaves the other end of the connection.

In order for a collision to take place, we'd have to get a new connection from an existing client, AND that client would have to use the same port number that it used for the earlier connection, AND our server would have to assign the same port number to this connection as it did before. Given that we use persistent HTTP connections between clients and nginx, the probability of this happening is so low that we can ignore it. 1 second is a long enough `TIME_WAIT` timeout.

The two TCP tuning parameters were set using `sysctl` by putting a file into `/etc/sysctl.d/` with the following:

```
net.ipv4.ip_local_port_range = 1024    65535
net.ipv4.netfilter.ip_conntrack_tcp_timeout_time_wait = 1
```

nf_conntrack_tcp_timeout_established

It turns out that there's another timeout value you need to be concerned with. The established connection timeout. Technically this should only apply to connections that are in the `ESTABLISHED` state, and a connection should get out of this state when a `FIN` packet goes through in either direction. This doesn't appear to happen and I'm not entirely sure why.

So how long do connections stay in this table then? The default value for `nf_conntrack_tcp_timeout_established` is 432000 seconds. I'll wait for you to do the long division…

Fun times.

Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

I changed the timeout value to 10 minutes (600 seconds) and in a few days time I noticed `conntrack_count` go down steadily until it sat at a very manageable level of a few thousand.

We did this by adding another line to the sysctl file:

```
net.netfilter.nf_conntrack_tcp_timeout_established=600
```

Starting with version 2.6.13, Linux supports pluggable congestion control algorithms . The congestion control algorithm used is set using the sysctl variable net.ipv4.tcp_congestion_control, which is set to bic/cubic or reno by default, depending on which version of the 2.6 kernel you are using.
To get a list of congestion control algorithms that are available in your kernel (if you are running 2.6.20 or higher), run:

```
[root@server1 ~] # sysctl net.ipv4.tcp_available_congestion_control
```

view raw gistfile1.sh hosted with ❤ by GitHub
The choice of congestion control options is selected when you build the kernel. The following are some of the options are available in the 2.6.23 kernel:
* reno: Traditional TCP used by almost all other OSes. (default)
* cubic: CUBIC-TCP (NOTE: There is a cubic bug in the Linux 2.6.18 kernel used by Redhat Enterprise Linux 5.3 and Scientific Linux 5.3. Use 2.6.18.2 or higher!)
* bic: BIC-TCP
* htcp: Hamilton TCP
* vegas: TCP Vegas
* westwood: optimized for lossy networks

If cubic and/or htcp are not listed when you do 'sysctl net.ipv4.tcp_available_congestion_control', try the following, as most distributions include them as loadable kernel modules:

```
[root@server1 ~] # /sbin/modprobe tcp_htcp

[root@server1 ~] # /sbin/modprobe tcp_cubic
```

view raw gistfile1.sh hosted with ❤ by GitHub
For long fast paths, I highly recommend using cubic or htcp. Cubic is the default for a number of Linux distributions, but if is not the default on your system, you can do the following:

Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

[root@server1 ~] # sysctl -w net.ipv4.tcp_congestion_control=cubic

Window size after idle

Related to the above is the `sysctl` setting `net.ipv4.tcp_slow_start_after_idle`. This tells the system whether it should start at the default window size only for new TCP connections or also for existing TCP connections that have been idle for too long (on 3.5, too long is 1 second, but see `net.sctp.rto_initial` for its current value on your system). If you're using persistent HTTP connections, you're likely to end up in this state, so set `net.ipv4.tcp_slow_start_after_idle=0` (just put it into the sysctl config file mentioned above).

InitialWindow For TCP initial Slow Start and Download File
https://www.cdnplanet.com/blog/tune-tcp-initcwnd-for-optimum-performance/

```
net.ipv4.tcp_low_latency = 1
#very low impact http://www.linuxvox.com/post/what-is-the-linux-kernel-parameter-tcp_low_latency/

net.core.optmem_max = 33554432

# Increase max number of PIDs
kernel.pid_max = 4194303

# The maximum number of "backlogged sockets".  Default is 128.
net.core.somaxconn = 1024 --> 50000
```

net.ipv4.tcp_syn_retries=2

```
net.ipv4.tcp_orphan_retries=2

net.ipv4.tcp_retries2=8


# Increase max number of sockets allowed in TIME_WAIT
net.ipv4.tcp_max_tw_buckets = 1440000
# Number of packets to keep in the backlog before the kernel starts dropping them
# A sane value is net.ipv4.tcp_max_syn_backlog = 3240000
net.ipv4.tcp_max_syn_backlog = 3240000
```

net.ipv4.tcp_keepalive_time = 600

```
# Disable TCP SACK (TCP Selective Acknowledgement), DSACK (duplicate TCP SACK), and FACK (Forward Acknowledgement)
# SACK requires enabling tcp_timestamps and adds some packet overhead
# Only advised in cases of packet loss on the network
net.ipv4.tcp_sack = 0
net.ipv4.tcp_dsack = 0
net.ipv4.tcp_fack = 0

# Disable TCP timestamps
```

Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

```
# Can have a performance overhead and is only advised in cases where sack is
needed (see tcp_sack)
net.ipv4.tcp_timestamps=0
```

http://fibrevillage.com/sysadmin/91-tcp-performance-tuning-10g-nic-with-high-rtt-in-linux
good refrences

https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Welcome%20to%20High%20Performance%20Computing%20(HPC)%20Central/page/Linux%20System%20Tuning%20Recommendations

Linux : How to tune up receive (TX) and transmit (RX) buffers on network interface

By Kaven Gagnon | May 5, 2015
0 Comment

Modern and performance/server grade network interface have the capability of using transmit and receive buffer description ring into the main memory. They use direct memory access (DMA) to transfer packets from the main memory to carry packets independently from the CPU.

The usual default buffering values for regular desktop NICs are 256 or 512 bytes. High performances NICs can achieve up to 4096 and/or 8192 bytes.

To view the capability and the current values of your interface, you'll need "ethtool". Simply do the following command :

```
ethtool -g eth0
```

This will output something like this :

```
Ring parameters for eth0:
Pre-set maximums:
RX: 4096
RX Mini: 0
RX Jumbo: 0
TX: 4096
Current hardware settings:
RX: 256
RX Mini: 0
RX Jumbo: 0
TX: 256
```

We can see here that both RX and TX values are set to 256 but the interface have the capability of 4096 bytes.

Navid Malek
navidmalekedu@gmail.com
navidmalek.blog.ir

To increase the buffers, do the following :

```
ethtool -G eth0 rx 4096 tx 4096
```

//TODO ON SERVER
Another thing you can do to help increase TCP throughput with 1GB NICs is to increase the size of the interface queue. For paths with more than 50 ms RTT, a value of 5000-10000 is recommended. To increase txqueuelen, do the following:

```
[root@server1 ~] ifconfig eth0 queuelen 5000
```

Good Tuning advices for server and process
http://www.tweaked.io/guide/kernel/

Commands :

```
 gedit /etc/sysctl.conf
  120  modprobe nf_conntrack
  121  sysctl -p
  122  echo "262144" > /sys/module/nf_conntrack/parameters/hashsize
sudo ethtool -G ens33 rx 4096 tx 4096

echo "262144" > /sys/module/nf_conntrack/parameters/hashsize
```