3.3. TCP Variables

This section will take a brief look at the variables that changes the behaviour of the TCP variables. These variables are normally set to a pretty good value per default and most of them should never ever be touched, except when asked by authoritative developers! They are mainly described here, only for those who are curious about their basic meaning.

3.3.1. tcp_abort_on_overflow

The tcp_abort_on_overflow variable tells the kernel to reset new connections if the system is currently overflowed with new connection attempts that the daemon(s) can not handle. What this means, is that if the system is overflowed with 1000 large requests in a burst, connections may be reset since we can not handle them if this variable is turned on. If it is not set, the system will try to recover and handle all requests.

This variable takes an boolean value (ie, 1 or 0) and is per default set to 0 or FALSE. Avoid enabling this option except as a last resort since it most definitely harm your clients. Before considering using this variable you should try to tune up your daemons to accept connections faster.

3.3.2. tcp_adv_win_scale

This variable is used to tell the kernel how much of the socket buffer space should be used for TCP window size, and how much to save for an application buffer. If tcp_adv_win_scale is negative, the following equation is used to calculate the buffer overhead for window scaling:

Where bytes are the amount of bytes in the window. If the tcp_adv_win_scale value is positive, the following equation is used to calculate the buffer overhead:

The tcp_adv_win_scale variable takes an integer value and is per default set to 2. This in turn means that the application buffer is 1/4th of the total buffer space specified in the tcp_rmem variable.

3.3.3. tcp_app_win

This variable tells the kernel how many bytes to reserve for a specific TCP window in the TCP sockets memory buffer where the specific TCP window is transfered in. This value is used in a calculation that specifies how much of the buffer space to reserve that looks as the following:

As you may understand from the above calculation, the larger this value gets, the smaller will the buffer space be for the specific window. The only exception to this calculation is 0, which tells the kernel to reserve no space for this specific connection. The default value for this variable is 31 and should in

general be a good value. Do not change this value unless you know what you are doing.

3.3.4. tcp_dsack

This option is required to send duplicate SACKs which was briefly described in the tcp_sack variable explanation. This is described in detail within the RFC 2883. This RFC document explains in detail how to handle situations where a packet is received twice or out of order. D-SACK is an extension to standard SACK and is used to tell the sender when a packet was received twice (ie, it was duplicated). The D-SACK data can then be used by the transmitter to improve network settings and so on. This should be 100% backwards compatible with older implementations as long as the previous implementors have not tried to implement this into the old SACK option in their own fashion. This is extremely rare and should not be a problem for anyone.

The tcp_dsack variable uses a boolean value and is per default set to 1, or turned on. Of course, this behaviour is *only* used if tcp_sack is turned on since tcp_dsack is heavily dependant upon tcp_sack. In almost all cases this should be a good idea to have turned on.

3.3.5. tcp_ecn

The tcp_ecn variable turns on Explicit Congestion Notification in TCP connections. This is used to automatically tell the host when there are congestions in a route to a specific host or a network. This can be used to throttle the transmitters to send packets in a slower rate over that specific router or firewall. Explicit Congestion Notification (ECN) is explained in detail in the *RFC 3168 - The Addition of Explicit Congestion Notification (ECN) to IP* document and there is also a performance evaluation of the addition of ECN available in the *RFC 2884 - Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks* document.

Briefly, this document details how we could notify other hosts when we are congested or not, which in turn will make us able to choose other routes in preference over the currently used route, or to simply send less data until we no longer receive congestion messages.

There are still some old firewalls and routers out on the Internet that will filter away all IP packets that has the ECN bits set. They are fairly uncommon these days, but if you are unlucky, you may run into them. If you do experience connection problems to specific hosts, try turning ECN off and see how things go. If you find the actual host blocking the ECN packets, try getting in touch with the administrators and warn them about this. A deeper explanation of the problem, as well as a list of the most common hardware that causes this trouble is available, and can be found in the *Other resources* appendix, under the *ECN-under-Linux Unofficial Vendor Support Page* heading.

The tcp_ecn variable takes a boolean value and is per default set to 0, or turned off. If you want to turn this on in your kernel, you should set this variable to 1.

3.3.6. tcp_fack

The tcp_fack variable enables the Forward Acknowledgement system in Linux. Forward Acknowledgement is a special algorithm that works on top of the SACK options, and is geared at

congestion controlling.

The main idea of FACK algorithm is to consider the most forward selective acknowledgement sequence number as a sign that all the previous un(selectively) acknowledged segments were lost. This observation allows to improve recovery of losses singificantly. This assumption breaks in presence of packet reordering, in which case the FACK algorithm is automatically turned off for that specific connection.

This algorithm was originally created by Matthew Mathis and co-authors. You can find the papers describing the algorithm more closely over at http://www.psc.edu/~mathis/.

The tcp_fack variable takes a boolean value, and is per default set to 1, or turned on. This behaviour is not used if tcp_sack is turned off since it is heavily dependant upon tcp_sack.

3.3.7. tcp_fin_timeout

The tcp_fin_timeout variable tells kernel how long to keep sockets in the state FIN-WAIT-2 if you were the one closing the socket. This is used if the other peer is broken for some reason and don't close its side, or the other peer may even crash unexpectedly. Each socket left in memory takes approximately 1.5Kb of memory, and hence this may eat a lot of memory if you have a moderate webserver or something alike.

This value takes an integer value which is per default set to 60 seconds. This used to be 180 seconds in 2.2 kernels, but was reduced due to the problems mentioned above with webservers and problems that arose from getting huge amounts of connections.

Also see the tcp_max_orphans and tcp_orphan_retries variables for more information.

3.3.8. tcp_keepalive_intvl

The tcp_keepalive_intvl variable tells the kernel how long to wait for a reply on each keepalive probe. This value is in other words extremely important when you try to calculate how long time will go before your connection will die a keepalive death.

The variable takes an integer value and the default value is 75 seconds. This is in the higher regions and should be concidered the higher threshold on what values should be concidered normal to use. The default values of the tcp_keepalive_probes and tcp_keepalive_intvl can be used to get the default time it will take before the connection is timed out because of keepalive.

With the default values of sending 9 probes with 75 seconds for each, it would take approximately 11 minutes before the connection is timed out, counting from when we start the probing which in turn will happen 2 hours from the time we last saw any traffic on the connection.

3.3.9. tcp_keepalive_probes

The tcp_keepalive_probes variable tells the kernel how many TCP keepalive probes to send out before it decides a specific connection is broken.

This variable takes an integer value, which should generally not be set higher than 50 depending on your tcp_keepalive_time value and the tcp_keepalive_interval. The default value is to send out 9 probes before telling the application that the connection is broken.

3.3.10. tcp_keepalive_time

The tcp_keepalive_time variable tells the TCP/IP stack how often to send TCP keepalive packets to keep an connection alive if it is currently unused. This value is only used when keepalive is enabled.

The tcp_keepalive_time variable takes an integer value which is counted in seconds. The default value is 7200 seconds, or 2 hours. This should be a good value for most hosts and will not take too much network resources from you. Do not set this value to low since it will then use up your network resources with unnecessary traffic.

3.3.11. tcp_max_orphans

The tcp_max_orphans variable tells the kernel how many TCP sockets that are not attached to any user file handle to maintain. In case this number is exceeded, orphaned connections are immediately reset and a warning is printed.

The only reason for this limit to exist is to prevent some simple DoS attacks. Generally you should not rely on this limit, nor should you lower it artificially. If need be, you should instead increase this limit if your network environment requires such an update. Increasing this limit may require that you get more memory installed to your system. If you hit this limit, you may also tune your network services a little bit to linger and kill sockets in this state more aggressively.

This variable takes an integer value and is per default set to 8192, but heavily depends upon how much memory you have. Each orphan that currently lives eats up 64Kb of unswappable memory, which means that one hell of a lot of data will be used up if problems arise.

If you run into this limit, you will get an error message via the syslog facility kern.info that looks something like this:

```
TCP: too many of orphaned sockets
```

If this shows up, either upgrade the box in question or look closer at the tcp_fin_timeout or tcp_orphans_retries which should give you some help with getting rid of huge amounts of orphaned sockets.

3.3.12. tcp_max_syn_backlog

The tcp_max_syn_backlog variable tells your box how many SYN requests to keep in memory that we have yet to get the third packet in a 3-way handshake from. The tcp_max_syn_backlog variable is overridden by the tcp_syncookies variable, which needs to be turned on for this variable to have any effect. If the server suffers from overloads at peak times, you may want to increase this value a little bit.

This variable takes an integer value and is per default set to different values depending on how much memory you have. If you have less than 128 Mb of RAM, it is set to a maximum of 128 SYN backlog

requests. If you have more than 128 Mb of RAM, it is set to 1024 SYN backlog requests.

If this value is raised to a larger value than 1024 it would most probably be better to change the TCP_SYNQ_HSIZE value and recompile your kernel. The TCP_SYNQ_HSIZE variable is set in linux/include/tcp.h. This value should be set so to keep this formula true:

TCP_SYNQ_HSIZE*16<=tcp_max_syn_backlog

In other words, TCP_SYNQ_HSIZE times 16 should be smaller than or equal to tcp_max_syn_backlog.

3.3.13. tcp_max_tw_buckets

The tcp_max_tw_buckets variable tells the system the maximum number of sockets in TIME-WAIT to be held simultaneously. If this number is exceeded, the exceeding sockets are destroyed and a warning message is printed to you. The reason for this limit to exist is to get rid of really simple DoS attacks.

The tcp_max_tw_buckets variable takes an integer value which tells the system at which point to start destroying timewait sockets. The default value is set to 180000. This may sound much, but it is not. If anything, you should possibly need to increase this value if you start receiving errors due to this setting.

You should not lower this limit artificially. If you start receiving errors indicating this problem in normal operation, you should instead increase this value if your network requires so. This may lead to the requirement of more memory installed in the machine in question.

3.3.14. tcp_mem

The tcp_mem variable defines how the TCP stack should behave when it comes to memory usage. It consists of three values, just as the tcp_wmem and tcp_rmem variables. The values are measured in memory pages (in short, pages). The size of each memory page differs depending on hardware and configuration options in the kernel, but on standard i386 computers, this is 4 kilobyte or 4096 bytes. On some newer hardware, this is set to 16, 32 or even 64 kilobytes. All of these values have no real default value since it is calculated at boottime by the kernel, and should in most cases be good for you and most usages you may encounter.

The first value specified in the tcp_mem variable tells the kernel the low threshold. Below this point, the TCP stack do not bother at all about putting any pressure on the memory usage by different TCP sockets.

The second value tells the kernel at which point to start pressuring memory usage down. This so called memory pressure mode is continued until the memory usage enters the lower threshold again, and at which point it enters the default behaviour of the low threshold again. The memory pressure mode presses down the TCP receive and send buffers for all the sockets as much as possible, until the low mark is reached again.

The final value tells the kernel how many memory pages it may use maximally. If this value is reached, TCP streams and packets start getting dropped until we reach a lower memory usage again. This value includes all TCP sockets currently in use.

This variable may give tremenduous increase in throughput on high bandwidth networks, if used properly together with the tcp_rmem and tcp_wmem variable. The tcp_rmem variable doesn't need too much manual tuning however, since the Linux 2.4 kernels has very good autotuning handlings on this aspect, but the other two may be worth looking at. For more information about this, look at the *TCP Tuning Guide*.

3.3.15. tcp_orphan_retries

The tcp_orphan_retries variable tells the TCP/IP stack how many times to retry to kill connections on the other side before killing it on our own side. If your machine runs as a highly loaded http server it may be worth thinking about lowering this value. http sockets will consume large amounts of resources if not checked.

This variable takes an integer value. The default value for this variable is 7, which would approximately correspond to 50 seconds through 16 minutes depending on the Retransmission Timeout (RTO). For a complete explanation of the RTO, read the "3.7. Data Communication" section in RFC 793 - Transmission Control Protocol.

Also see the tcp_max_orphans variable for more information.

3.3.16. tcp_reordering

The tcp_reordering variable tells the kernel how much a TCP packet may be reordered in a stream without assuming that the packet was lost somewhere on the way. If the packet is assumed lost, the TCP stack will automatically go back into a slow start since it believes packets may have been lost due to congestion somewhere. The TCP stack will also fall back from using the FACK algorithm for this specific host in the future.

This variable takes an integer variable and is per default set to 3. This should in general be a good value and you should not touch it. If this value is lowered, it may result in bad network performance, especially if packets often get reordered in connections.

This variable is overridden by the **reordering** option in the **ip route** command starting with kernels 2.3.15 and higher. If **reordering** is not given to the **ip route** command, the default is taken from the sysctl tcp_reordering.

3.3.17. tcp_retrans_collapse

This variable implements a bug in the TCP protocol so it will be able to talk to certain other buggy TCP stacks. Without implementing this bug in the TCP stack, we would be unable to talk to certain printers that has this bug built in. This bug makes the TCP stack try to send bigger packets on retransmission of packets to work around bugs in those printers and other hardware implementations.

This variable takes a boolean value and is normally set to 1, or on. Implementing this bug workaround will not break compatibility from our host to others, but it will make it possible to speak to those printers. In general, it should not be a dangerous workaround, but you may turn it off if you receive weird error messages.

3.3.18. tcp_retries1

The tcp_retries1 variable tells the kernel how many times it should retry to get to a host before reaching a decision that something is wrong and that it should report the suspected problem to the network layer. The minimal value here specified by RFC ???? is 3, which is also the default. This corresponds to 3 seconds through 8 minutes depending on your Retransmission timeout (RTO). For a good explanation of the Retransmission timeout, read the "3.7. Data Communication" section in RFC 793 - Transmission Control Protocol.

This variable takes an integer, which is per default set to 3 as explained above. The lower limit is 3 if you want to follow standards, and the upper bound should be lower than 100 or so since timeouts could be worse than horrible if this high.

3.3.19. tcp_retries2

The tcp_retries2 value tells the kernel how many times to retry before killing an alive TCP connection. This limit is specified to a minimum of 100 seconds in RFC 1122, but is normally way to short.

The variable takes an integer value and is set to 15 per default. This value corresponds to 13-30 minutes depending on the Retransmission timeout (RTO). Generally this should be a good timeout, you may bring it down but not necessarily.

3.3.20. tcp_rfc1337

The tcp_rfc1337 variable implements the solution found in *RFC 1337 - TIME-WAIT Assassination Hazards in TCP* to TIME-WAIT Assassination. In short, the problem is that old duplicate packets may interfer with new connections, and lead to three different problems. The first one is that old duplicate data may be accepted erroneously in new connections, leading to the sent data becoming corrupt. The second problem is that connections may become desynchronized and get into an ACK loop because of old duplicate packets entering new connections, which will become desynchronized. The third and last problem is that old duplicate packets may enter newly established connections erroneously and kill the new connection.

There are three possible solutions to this according to the mentioned RFC, however, one solution is only partial and not a long term solution, while the last requires heavy modifications of the TCP protocol, and is hence not a viable option.

The final solution that the linux kernel implements with this option, is to simply ignore RST packets sent to a socket while it is in the TIME-WAIT state. In use together with 2 minute Maximum Segment Life (MSL), this should eliminate all three problems discussed in RFC 1337.

3.3.21. tcp_rmem

The tcp_rmem variable is pretty much the same as the tcp_wmem, except in one large area. It tells the kernel the TCP receive memory buffers instead of the transmit buffer which is defined in tcp_wmem. This variable takes 3 different values, just the same as the tcp_wmem variable.

The first value tells the kernel the minimum receive buffer for each TCP connection, and this buffer is always allocated to a TCP socket, even under high pressure on the system. This value is set to 4096 bytes, or 4 kilobytes, in newer kernels, but was in previous kernels set to 8192 bytes or 8 kilobytes. This should generally be a good value, and you should avoid raising this value if you are sporadically experiencing large bursts and high network loads since the system may get into even worse problems then.

The second value specified tells the kernel the default receive buffer allocated for each TCP socket. This value overrides the /proc/sys/net/core/rmem_default value used by other protocols. The default value here is 87380 bytes, or 85 kilobytes. This value is used together with tcp_adv_win_scale and tcp_app_win to calculate the TCP window size, which is discussed within the explanations of those variables. This value should under normal circumstances not be touched either since it may result in similar problems as with the first value in this variable.

This variable may give tremenduous increase in throughput on high bandwidth networks, if used properly together with the tcp_mem and tcp_wmem variable. The tcp_rmem variable doesn't need too much manual tuning however, since the Linux 2.4 kernels has very good autotuning handlings on this aspect, but the other two may be worth looking at. For more information about this, look at the *TCP Tuning Guide*.

The third and last value specified in this variable specifies the maximum receive buffer that can be allocated for a TCP socket. This value is overridden by the /proc/sys/net/core/rmem_max if the ipv4 value is larger than the core value. You need to look at the core value before you do any changes to the ipv4 value in other words. The default value here is a double up of the second value specified. In other words, 87380 * 2 bytes, or 174760 bytes (170 kilobytes). Generally this should be a good value and should not need to be changed.

3.3.22. tcp_sack

The tcp_sack variable enables Selective Acknowledgements (SACK) as they are defined in *RFC 2883 - An Extension to Selective Acknowledgement (SACK) Option for TCP* and *RFC 2883 - An Extension to Selective Acknowledgement (SACK) Option for TCP*. These RFC documents contain information on an TCP option that was especially developed to handle lossy connections.

If this variable is turned on, our host will set the SACK option in the TCP option field in the TCP header when it sends out a SYN packet. This tells the server we are connecting to that we are able to handle SACK. In the future, if the server knows how to handle SACK, it will then send ACK packets with the SACK option turned on. This option selectively acknowledges each segment in a TCP window. This is especially good on very lossy connections (connections that loose a lot of data in the transfer) since this makes it possible to only retransmit specific parts of the TCP window which lost data and not the whole TCP window as the old standards told us to do. This means that if a certain segment of a TCP window is not received, the receiver will not return a SACK for that segment. The sender will then know which packets where not received by the receiver, and will hence retransmit that packet. For redundancy, this option will fill up all space possibly within the option space, 40 bytes per segment. Each SACK'ed packet takes up 2 32-bit unsigned integers and hence the option space can contain 4 SACK'ed segments. However, normally the timestamp option is used in conjunction with this option. The timestamp option takes up 10 bytes of data, and hence only 3 segments may be SACK'ed in each packet in normal operation.

If you know that you will be sending data over an extremely lossy connection such as a bad internet connection at one point or another, this variable is recommended to turn on. However, if you will only send data over an internal network consisting of a perfect condition 2 feet cat-5 cable and both machines being able to keep up with maximum speed without any problems, you should not need it. This option is not required, but it is definitely a good idea to have it turned on. Note that the SACK option is 100% backwards compatible, so you should not run into any problems talking to any other hosts on the internet who do not support it.

The tcp_sack option takes a boolean value. This is per default set to 1, or turned on. This is generally a good idea and should cause no problems.

3.3.23. tcp_stdurg

This variable enables or disables RFC 1122 compliance. The default behaviour is to be BSD 4.2 compliant, which follows the RFC 793 explanation of the URG flag. If this variable is turned on, we may be unable to communicate properly with certain hosts on the internet, or more specifically, those hosts on the internet that are BSD 4.2 compliant. For more information on the changes, read the *RFC 1122 - Requirements for Internet Hosts -- Communication Layers* under the section "4.2.2.4 Urgent Pointer: RFC 793 Section 3.1 explanation" which refers back to *RFC 793 - Transmission Control Protocol* as can be seen in the name of the section mentioned.

The tcp_stdurg variable takes a boolean value and is per default set to 0, or FALSE. If this is turned on, your box may be unable to talk to certain hosts as described above.

3.3.24. tcp_syn_retries

The tcp_syn_retries variable tells the kernel how many times to try to retransmit the initial SYN packet for an active TCP connection attempt.

This variable takes an integer value, but should not be set higher than 255 since each retransmission will consume huge amounts of time as well as some amounts of bandwidth. Each connection retransmission takes aproximately 30-40 seconds. The default setting is 5, which would lead to an aproximate of 180 seconds delay before the connection times out.

3.3.25. tcp_synack_retries

The tcp_synack_retries setting tells the kernel how many times to retransmit the SYN,ACK reply to an SYN request. In other words, this tells the system how many times to try to establish a passive TCP connection that was started by another host.

This variable takes an integer value, but should under no circumstances be larger than 255 for the same reasons as for the tcp_syn_retries variable. Each retransmission will take aproximately 30-40 seconds. The default value of the tcp_synack_retries variable is 5, and hence the default timeout of passive TCP connections is aproximately 180 seconds.

3.3.26. tcp_syncookies

The tcp_syncookies variable is used to send out so called syncookies to hosts when the kernels syn backlog queue for a specific socket is overflowed. This means that if our host is flooded with several SYN packets from different hosts, the syn backlog queue may overflow, and hence this function starts sending out cookies to see if the SYN packets are really legit.

This variable is used to prevent an extremely common attack that is called a "syn flood attack". The tcp_syncookies variable takes an boolean value which can either be set to 0 or 1, where 0 means off. The default setting is to turn this function off.

There has been a lot of discussions about the problems and flaws with syncookies in the past. Personally, I choose to look on SYN cookies as something fairly usefull, and since it is not causing any strangeness under normal operation, it should not be very dangerous. However, it may be dangerous, and you may want to see below.

The tcp_syncookies option means that under high load the system will make new connections without advanced features like ECN or SACK being used. If syncookies are being triggered during normal load rather than an attack you should tune the tcp queue length and the servers handling the load.

You must not use this facility to help a highly loaded server to stand down from legal connections. If you start to see syn flood warnings in your logs, and they show out to be legit connections, you may tune the tcp_max_syn_backlog, tcp_synack_retries and tcp_abort_on_overflow variables.

3.3.27. tcp_timestamps

The tcp_timestamps variable tells the kernel to use timestamps as defined in RFC 1323. In short, this is an TCP option that can be used to calculate the Round Trip Measurement in a better way than the retransmission timeout method can. This should be backwards compatible in almost all circumstances so you could very well turn this on if your host lives on a high speed network. If you only use up to an 10mbps connection of some sort(LAN or Internet or anything for that matter), you should manage fairly well without this option, and at really low speeds, you may even be better off with this variable turned off.

This variable takes a boolean value and is per default set to 1, or enabled. Generally this should be a good idea to have turned on. The only exception would be if you live on an extremely slow connection such as a 56 kbps modem connection to the Internet.

For more technical information about this option read section 4 of the *RFC 1323 - TCP Extensions for High Performance*. This document discusses the technical and theoretical introduction of these options and how it should work.

3.3.28. tcp_tw_recycle

This variable enables the fast recycling function of TIME-WAIT sockets. Unless you know what you are doing you should not touch this function at all.

The tcp_tw_recycle variable takes an integer value and the default value is 0 from my experience and

my understanding of the source code of linux. In other words, the statement in the linux/Documentation /ip-sysctl.txt file is wrong unless I am mistaken.

Do not reset this from its default value unless you know what you are doing and/or have gotten the advice or request from an technical expert or kernel coder.

3.3.29. tcp_window_scaling

The tcp_window_scaling variable enables window scaling as it is defined in RFC 1323. This RFC specifies how we can scale TCP windows if we are sending them over Large Fat Pipes (LFP). When sending TCP packets over these large pipes, we experience heavy bandwidth loss due to the channels not being fully filled while waiting for ACK's for our previous TCP windows. The main problem is that a TCP Window can not be larger than 2**16 bytes, or 65Kb large. Enabling tcp_window_scaling enables a special TCP option which makes it possible to scale these windows to a larger size, and hence reduces bandwidth losses due to not utilizing the whole connection.

This variable takes a boolean value and is per default set to 1, or true. If you want to turn this off, set it to 0.

For more information about TCP window scaling, read the *RFC 1323 - TCP Extensions for High Performance*.

3.3.30. tcp_wmem

This variable takes 3 different values which holds information on how much TCP sendbuffer memory space each TCP socket has to use. Every TCP socket has this much buffer space to use before the buffer is filled up. Each of the three values are used under different conditions.

The first value in this variable tells the minimum TCP send buffer space available for a single TCP socket. This space is always allocated for a specific TCP socket opened by a program as soon as it is opened. This value is normally set to 4096 bytes, or 4 kilobytes.

The second value in the variable tells us the default buffer space allowed for a single TCP socket to use. If the buffer tries to grow larger than this, it may get hampered if the system is currently under heavy load and don't have a lot of memory space available. It may even have to drop packets if the system is so heavily loaded that it can not give more memory than this limit. The default value set here is 16384 bytes, or 16 kilobytes of memory. It is not very wise to raise this value since the system is most probably already under heavy memory load and usage, and this would hence lead to even more problems for the rest of the system. This value overrides the /proc/sys/net/core/wmem_default value that is used by other protocols, and is usually set to a lower value than the core value.

The third value tells the kernel the maximum TCP send buffer space. This defines the maximum amount of memory a single TCP socket may use. Per default this value is set to 131072, or 128 kilobytes. This should be a reasonable value for most circumstances, and you will most probably never need to change these values. However, if you ever do need to change it, you should keep in mind that the /proc/sys /net/core/wmem_max value overrides this value, and hence this value should always be smaller than that value.

This variable may give tremenduous increase in throughput on high bandwidth networks, if used properly together with the tcp_mem and tcp_rmem variable. The tcp_wmem variable is the variable of the three which may give the most gain from this kind of tweaking. Do note that you will see almost no gain on slower networks than giga ethernet networks. For more information about this, look at the *TCP Tuning Guide*.

Prev Home Next
Inet peer storage Up ICMP Variables