

NAME

tcp – TCP protocol

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
```

```
tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
```

DESCRIPTION

This is an implementation of the TCP protocol defined in RFC 793, RFC 1122 and RFC 2001 with the NewReno and SACK extensions. It provides a reliable, stream-oriented, full-duplex connection between two sockets on top of **ip(7)**, for both v4 and v6 versions. TCP guarantees that the data arrives in order and retransmits lost packets. It generates and checks a per-packet checksum to catch transmission errors. TCP does not preserve record boundaries.

A newly created TCP socket has no remote or local address and is not fully specified. To create an outgoing TCP connection use **connect(2)** to establish a connection to another TCP socket. To receive new incoming connections, first **bind(2)** the socket to a local address and port and then call **listen(2)** to put the socket into the listening state. After that a new socket for each incoming connection can be accepted using **accept(2)**. A socket which has had **accept(2)** or **connect(2)** successfully called on it is fully specified and may transmit data. Data cannot be transmitted on listening or not yet connected sockets.

Linux supports RFC 1323 TCP high performance extensions. These include Protection Against Wrapped Sequence Numbers (PAWS), Window Scaling and Timestamps. Window scaling allows the use of large (> 64 kB) TCP windows in order to support links with high latency or bandwidth. To make use of them, the send and receive buffer sizes must be increased. They can be set globally with the `/proc/sys/net/ipv4/tcp_wmem` and `/proc/sys/net/ipv4/tcp_rmem` files, or on individual sockets by using the **SO_SNDBUF** and **SO_RCVBUF** socket options with the **setsockopt(2)** call.

The maximum sizes for socket buffers declared via the **SO_SNDBUF** and **SO_RCVBUF** mechanisms are limited by the values in the `/proc/sys/net/core/rmem_max` and `/proc/sys/net/core/wmem_max` files. Note that TCP actually allocates twice the size of the buffer requested in the **setsockopt(2)** call, and so a succeeding **getsockopt(2)** call will not return the same size of buffer as requested in the **setsockopt(2)** call. TCP uses the extra space for administrative purposes and internal kernel structures, and the `/proc` file values reflect the larger sizes compared to the actual TCP windows. On individual connections, the socket buffer size must be set prior to the **listen(2)** or **connect(2)** calls in order to have it take effect. See **socket(7)** for more information.

TCP supports urgent data. Urgent data is used to signal the receiver that some important message is part of the data stream and that it should be processed as soon as possible. To send urgent data specify the **MSG_OOB** option to **send(2)**. When urgent data is received, the kernel sends a **SIGURG** signal to the process or process group that has been set as the socket "owner" using the **SIOCSPGRP** or **FIOSETOWN** ioctls (or the POSIX.1-specified **fcntl(2)** **F_SETOWN** operation). When the **SO_OOBINLINE** socket option is enabled, urgent data is put into the normal data stream (a program can test for its location using the **SIOCATMARK** ioctl described below), otherwise it can be received only when the **MSG_OOB** flag is set for **recv(2)** or **recvmsg(2)**.

When out-of-band data is present, **select(2)** indicates the file descriptor as having an exceptional condition and `poll(2)` indicates a **POLLPRI** event.

Linux 2.4 introduced a number of changes for improved throughput and scaling, as well as enhanced functionality. Some of these features include support for zero-copy **sendfile(2)**, Explicit Congestion Notification, new management of **TIME_WAIT** sockets, keep-alive socket options and support for Duplicate SACK extensions.

Address formats

TCP is built on top of IP (see **ip(7)**). The address formats defined by **ip(7)** apply to TCP. TCP supports point-to-point communication only; broadcasting and multicasting are not supported.

/proc interfaces

System-wide TCP parameter settings can be accessed by files in the directory */proc/sys/net/ipv4/*. In addition, most IP */proc* interfaces also apply to TCP; see **ip(7)**. Variables described as *Boolean* take an integer value, with a nonzero value ("true") meaning that the corresponding option is enabled, and a zero value ("false") meaning that the option is disabled.

tcp_abc (Integer; default: 0; Linux 2.6.15 to Linux 3.8)

Control the Appropriate Byte Count (ABC), defined in RFC 3465. ABC is a way of increasing the congestion window (*cwnd*) more slowly in response to partial acknowledgements. Possible values are:

- 0** increase *cwnd* once per acknowledgement (no ABC)
- 1** increase *cwnd* once per acknowledgement of full sized segment
- 2** allow increase *cwnd* by two if acknowledgement is of two segments to compensate for delayed acknowledgements.

tcp_abort_on_overflow (Boolean; default: disabled; since Linux 2.4)

Enable resetting connections if the listening service is too slow and unable to keep up and accept them. It means that if overflow occurred due to a burst, the connection will recover. Enable this option *only* if you are really sure that the listening daemon cannot be tuned to accept connections faster. Enabling this option can harm the clients of your server.

tcp_adv_win_scale (integer; default: 2; since Linux 2.4)

Count buffering overhead as $bytes/2^{tcp_adv_win_scale}$, if *tcp_adv_win_scale* is greater than 0; or $bytes - bytes/2^{-(tcp_adv_win_scale)}$, if *tcp_adv_win_scale* is less than or equal to zero.

The socket receive buffer space is shared between the application and kernel. TCP maintains part of the buffer as the TCP window, this is the size of the receive window advertised to the other end. The rest of the space is used as the "application" buffer, used to isolate the network from scheduling and application latencies. The *tcp_adv_win_scale* default value of 2 implies that the space used for the application buffer is one fourth that of the total.

tcp_allowed_congestion_control (String; default: see text; since Linux 2.4.20)

Show/set the congestion control algorithm choices available to unprivileged processes (see the description of the **TCP_CONGESTION** socket option). The items in the list are separated by white space and terminated by a newline character. The list is a subset of those listed in *tcp_available_congestion_control*. The default value for this list is "reno" plus the default setting of *tcp_congestion_control*.

tcp_autocorking (Boolean; default: enabled; since Linux 3.14)

If this option is enabled, the kernel tries to coalesce small writes (from consecutive **write(2)** and **sendmsg(2)** calls) as much as possible, in order to decrease the total number of sent packets. Coalescing is done if at least one prior packet for the flow is waiting in Qdisc queues or device transmit queue. Applications can still use the **TCP_CORK** socket option to obtain optimal behavior when they know how/when to uncork their sockets.

tcp_available_congestion_control (String; read-only; since Linux 2.4.20)

Show a list of the congestion-control algorithms that are registered. The items in the list are separated by white space and terminated by a newline character. This list is a limiting set for the list in *tcp_allowed_congestion_control*. More congestion-control algorithms may be available as modules, but not loaded.

tcp_app_win (integer; default: 31; since Linux 2.4)

This variable defines how many bytes of the TCP window are reserved for buffering overhead.

A maximum of $(window/2^{tcp_app_win}, mss)$ bytes in the window are reserved for the application buffer. A value of 0 implies that no amount is reserved.

tcp_base_mss (Integer; default: 512; since Linux 2.6.17)

The initial value of *search_low* to be used by the packetization layer Path MTU discovery (MTU probing). If MTU probing is enabled, this is the initial MSS used by the connection.

tcp_bic (Boolean; default: disabled; Linux 2.4.27/2.6.6 to Linux 2.6.13)

Enable BIC TCP congestion control algorithm. BIC-TCP is a sender-side-only change that ensures a linear RTT fairness under large windows while offering both scalability and bounded TCP-friendliness. The protocol combines two schemes called additive increase and binary search increase. When the congestion window is large, additive increase with a large increment ensures linear RTT fairness as well as good scalability. Under small congestion windows, binary search increase provides TCP friendliness.

tcp_bic_low_window (integer; default: 14; Linux 2.4.27/2.6.6 to Linux 2.6.13)

Set the threshold window (in packets) where BIC TCP starts to adjust the congestion window. Below this threshold BIC TCP behaves the same as the default TCP Reno.

tcp_bic_fast_convergence (Boolean; default: enabled; Linux 2.4.27/2.6.6 to Linux 2.6.13)

Force BIC TCP to more quickly respond to changes in congestion window. Allows two flows sharing the same connection to converge more rapidly.

tcp_congestion_control (String; default: see text; since Linux 2.4.13)

Set the default congestion-control algorithm to be used for new connections. The algorithm "reno" is always available, but additional choices may be available depending on kernel configuration. The default value for this file is set as part of kernel configuration.

tcp_dma_copybreak (integer; default: 4096; since Linux 2.6.24)

Lower limit, in bytes, of the size of socket reads that will be offloaded to a DMA copy engine, if one is present in the system and the kernel was configured with the **CONFIG_NET_DMA** option.

tcp_dsack (Boolean; default: enabled; since Linux 2.4)

Enable RFC 2883 TCP Duplicate SACK support.

tcp_fastopen (Bitmask; default: 0x1; since Linux 3.7)

Enables RFC 7413 Fast Open support. The flag is used as a bitmap with the following values:

- 0x1** Enables client side Fast Open support
- 0x2** Enables server side Fast Open support
- 0x4** Allows client side to transmit data in SYN without Fast Open option
- 0x200** Allows server side to accept SYN data without Fast Open option
- 0x400** Enables Fast Open on all listeners without **TCP_FASTOPEN** socket option

tcp_fastopen_key (since Linux 3.7)

Set server side RFC 7413 Fast Open key to generate Fast Open cookie when server side Fast Open support is enabled.

tcp_ecn (Integer; default: see below; since Linux 2.4)

Enable RFC 3168 Explicit Congestion Notification.

This file can have one of the following values:

- 0** Disable ECN. Neither initiate nor accept ECN. This was the default up to and including Linux 2.6.30.
- 1** Enable ECN when requested by incoming connections and also request ECN on outgoing connection attempts.
- 2** Enable ECN when requested by incoming connections, but do not request ECN on outgoing connections. This value is supported, and is the default, since Linux 2.6.31.

When enabled, connectivity to some destinations could be affected due to older, misbehaving middle boxes along the path, causing connections to be dropped. However, to facilitate and encourage deployment with option 1, and to work around such buggy equipment, the **tcp_ecn_fallback**

option has been introduced.

tcp_ecn_fallback (Boolean; default: enabled; since Linux 4.1)

Enable RFC 3168, Section 6.1.1.1. fallback. When enabled, outgoing ECN-setup SYNs that time out within the normal SYN retransmission timeout will be resent with CWR and ECE cleared.

tcp_fack (Boolean; default: enabled; since Linux 2.2)

Enable TCP Forward Acknowledgement support.

tcp_fin_timeout (integer; default: 60; since Linux 2.2)

This specifies how many seconds to wait for a final FIN packet before the socket is forcibly closed. This is strictly a violation of the TCP specification, but required to prevent denial-of-service attacks. In Linux 2.2, the default value was 180.

tcp_frto (integer; default: see below; since Linux 2.4.21/2.6)

Enable F-RTO, an enhanced recovery algorithm for TCP retransmission timeouts (RTOs). It is particularly beneficial in wireless environments where packet loss is typically due to random radio interference rather than intermediate router congestion. See RFC 4138 for more details.

This file can have one of the following values:

- 0** Disabled. This was the default up to and including Linux 2.6.23.
- 1** The basic version F-RTO algorithm is enabled.
- 2** Enable SACK-enhanced F-RTO if flow uses SACK. The basic version can be used also when SACK is in use though in that case scenario(s) exists where F-RTO interacts badly with the packet counting of the SACK-enabled TCP flow. This value is the default since Linux 2.6.24.

Before Linux 2.6.22, this parameter was a Boolean value, supporting just values 0 and 1 above.

tcp_frto_response (integer; default: 0; since Linux 2.6.22)

When F-RTO has detected that a TCP retransmission timeout was spurious (i.e., the timeout would have been avoided had TCP set a longer retransmission timeout), TCP has several options concerning what to do next. Possible values are:

- 0** Rate halving based; a smooth and conservative response, results in halved congestion window (*cwnd*) and slow-start threshold (*ssthresh*) after one RTT.
- 1** Very conservative response; not recommended because even though being valid, it interacts poorly with the rest of Linux TCP; halves *cwnd* and *ssthresh* immediately.
- 2** Aggressive response; undoes congestion-control measures that are now known to be unnecessary (ignoring the possibility of a lost retransmission that would require TCP to be more cautious); *cwnd* and *ssthresh* are restored to the values prior to timeout.

tcp_keepalive_intvl (integer; default: 75; since Linux 2.4)

The number of seconds between TCP keep-alive probes.

tcp_keepalive_probes (integer; default: 9; since Linux 2.2)

The maximum number of TCP keep-alive probes to send before giving up and killing the connection if no response is obtained from the other end.

tcp_keepalive_time (integer; default: 7200; since Linux 2.2)

The number of seconds a connection needs to be idle before TCP begins sending out keep-alive probes. Keep-alives are sent only when the **SO_KEEPALIVE** socket option is enabled. The default value is 7200 seconds (2 hours). An idle connection is terminated after approximately an additional 11 minutes (9 probes an interval of 75 seconds apart) when keep-alive is enabled.

Note that underlying connection tracking mechanisms and application timeouts may be much shorter.

tcp_low_latency (Boolean; default: disabled; since Linux 2.4.21/2.6; obsolete since Linux 4.14)

If enabled, the TCP stack makes decisions that prefer lower latency as opposed to higher throughput. If this option is disabled, then higher throughput is preferred. An example of an application where this default should be changed would be a Beowulf compute cluster. Since Linux 4.14, this file still exists, but its value is ignored.

tcp_max_orphans (integer; default: see below; since Linux 2.4)

The maximum number of orphaned (not attached to any user file handle) TCP sockets allowed in the system. When this number is exceeded, the orphaned connection is reset and a warning is printed. This limit exists only to prevent simple denial-of-service attacks. Lowering this limit is not recommended. Network conditions might require you to increase the number of orphans allowed, but note that each orphan can eat up to ~64 kB of unswappable memory. The default initial value is set equal to the kernel parameter NR_FILE. This initial default is adjusted depending on the memory in the system.

tcp_max_syn_backlog (integer; default: see below; since Linux 2.2)

The maximum number of queued connection requests which have still not received an acknowledgement from the connecting client. If this number is exceeded, the kernel will begin dropping requests. The default value of 256 is increased to 1024 when the memory present in the system is adequate or greater (≥ 128 MB), and reduced to 128 for those systems with very low memory (≤ 32 MB).

Before Linux 2.6.20, it was recommended that if this needed to be increased above 1024, the size of the SYNACK hash table (**TCP_SYNQ_HSIZE**) in *include/net/tcp.h* should be modified to keep

```
TCP_SYNQ_HSIZE * 16 <= tcp_max_syn_backlog
```

and the kernel should be recompiled. In Linux 2.6.20, the fixed sized **TCP_SYNQ_HSIZE** was removed in favor of dynamic sizing.

tcp_max_tw_buckets (integer; default: see below; since Linux 2.4)

The maximum number of sockets in TIME_WAIT state allowed in the system. This limit exists only to prevent simple denial-of-service attacks. The default value of NR_FILE*2 is adjusted depending on the memory in the system. If this number is exceeded, the socket is closed and a warning is printed.

tcp_moderate_rcvbuf (Boolean; default: enabled; since Linux 2.4.17/2.6.7)

If enabled, TCP performs receive buffer auto-tuning, attempting to automatically size the buffer (no greater than *tcp_rmem[2]*) to match the size required by the path for full throughput.

tcp_mem (since Linux 2.4)

This is a vector of 3 integers: [low, pressure, high]. These bounds, measured in units of the system page size, are used by TCP to track its memory usage. The defaults are calculated at boot time from the amount of available memory. (TCP can only use *low memory* for this, which is limited to around 900 megabytes on 32-bit systems. 64-bit systems do not suffer this limitation.)

low TCP doesn't regulate its memory allocation when the number of pages it has allocated globally is below this number.

pressure

When the amount of memory allocated by TCP exceeds this number of pages, TCP moderates its memory consumption. This memory pressure state is exited once the number of pages allocated falls below the *low* mark.

high The maximum number of pages, globally, that TCP will allocate. This value overrides any other limits imposed by the kernel.

tcp_mtu_probing (integer; default: 0; since Linux 2.6.17)

This parameter controls TCP Packetization-Layer Path MTU Discovery. The following values may be assigned to the file:

- 0 Disabled
- 1 Disabled by default, enabled when an ICMP black hole detected
- 2 Always enabled, use initial MSS of *tcp_base_mss*.

tcp_no_metrics_save (Boolean; default: disabled; since Linux 2.6.6)

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 it may sometimes cause performance degradation. If *tcp_no_metrics_save* is enabled, TCP will not cache metrics on closing connections.

tcp_orphan_retries (integer; default: 8; since Linux 2.4)

The maximum number of attempts made to probe the other end of a connection which has been closed by our end.

tcp_reordering (integer; default: 3; since Linux 2.4)

The maximum a packet can be reordered in a TCP packet stream without TCP assuming packet loss and going into slow start. It is not advisable to change this number. This is a packet reordering detection metric designed to minimize unnecessary back off and retransmits provoked by reordering of packets on a connection.

tcp_retrans_collapse (Boolean; default: enabled; since Linux 2.2)

Try to send full-sized packets during retransmit.

tcp_retries1 (integer; default: 3; since Linux 2.2)

The number of times TCP will attempt to retransmit a packet on an established connection normally, without the extra effort of getting the network layers involved. Once we exceed this number of retransmits, we first have the network layer update the route if possible before each new retransmit. The default is the RFC specified minimum of 3.

tcp_retries2 (integer; default: 15; since Linux 2.2)

The maximum number of times a TCP packet is retransmitted in established state before giving up. The default value is 15, which corresponds to a duration of approximately between 13 to 30 minutes, depending on the retransmission timeout. The RFC 1122 specified minimum limit of 100 seconds is typically deemed too short.

tcp_rfc1337 (Boolean; default: disabled; since Linux 2.2)

Enable TCP behavior conformant with RFC 1337. When disabled, if a RST is received in TIME_WAIT state, we close the socket immediately without waiting for the end of the TIME_WAIT period.

tcp_rmem (since Linux 2.4)

This is a vector of 3 integers: [min, default, max]. These parameters are used by TCP to regulate receive buffer sizes. TCP dynamically adjusts the size of the receive buffer from the defaults listed below, in the range of these values, depending on memory available in the system.

min minimum size of the receive buffer used by each TCP socket. The default value is the system page size. (On Linux 2.4, the default value is 4 kB, lowered to **PAGE_SIZE** bytes in low-memory systems.) This value is used to ensure that in memory pressure mode, allocations below this size will still succeed. This is not used to bound the size of the receive buffer declared using **SO_RCVBUF** on a socket.

default the default size of the receive buffer for a TCP socket. This value overwrites the initial default buffer size from the generic global *net.core.rmem_default* defined for all protocols. The default value is 87380 bytes. (On Linux 2.4, this will be lowered to 43689 in low-memory systems.) If larger receive buffer sizes are desired, this value should be increased (to affect all sockets). To employ large TCP windows, the *net.ipv4.tcp_window_scaling* must be enabled (default).

max the maximum size of the receive buffer used by each TCP socket. This value does not override the global *net.core.rmem_max*. This is not used to limit the size of the receive buffer declared using **SO_RCVBUF** on a socket. The default value is calculated using the formula

$$\max(87380, \min(4 \text{ MB}, tcp_mem[1] * PAGE_SIZE / 128))$$

(On Linux 2.4, the default is 87380*2 bytes, lowered to 87380 in low-memory systems).

tcp_sack (Boolean; default: enabled; since Linux 2.2)

Enable RFC 2018 TCP Selective Acknowledgements.

tcp_slow_start_after_idle (Boolean; default: enabled; since Linux 2.6.18)

If enabled, provide RFC 2861 behavior and time out the congestion window after an idle period. An idle period is defined as the current RTO (retransmission timeout). If disabled, the congestion window will not be timed out after an idle period.

tcp_stdurg (Boolean; default: disabled; since Linux 2.2)

If this option is enabled, then use the RFC 1122 interpretation of the TCP urgent-pointer field. According to this interpretation, the urgent pointer points to the last byte of urgent data. If this option is disabled, then use the BSD-compatible interpretation of the urgent pointer: the urgent pointer points to the first byte after the urgent data. Enabling this option may lead to interoperability problems.

tcp_syn_retries (integer; default: 6; since Linux 2.2)

The maximum number of times initial SYNs for an active TCP connection attempt will be retransmitted. This value should not be higher than 255. The default value is 6, which corresponds to retrying for up to approximately 127 seconds. Before Linux 3.7, the default value was 5, which (in conjunction with calculation based on other kernel parameters) corresponded to approximately 180 seconds.

tcp_synack_retries (integer; default: 5; since Linux 2.2)

The maximum number of times a SYN/ACK segment for a passive TCP connection will be retransmitted. This number should not be higher than 255.

tcp_syncookies (integer; default: 1; since Linux 2.2)

Enable TCP syncookies. The kernel must be compiled with **CONFIG_SYN_COOKIES**. The syncookies feature attempts to protect a socket from a SYN flood attack. This should be used as a last resort, if at all. This is a violation of the TCP protocol, and conflicts with other areas of TCP such as TCP extensions. It can cause problems for clients and relays. It is not recommended as a tuning mechanism for heavily loaded servers to help with overloaded or misconfigured conditions. For recommended alternatives see *tcp_max_syn_backlog*, *tcp_synack_retries*, and *tcp_abort_on_overflow*. Set to one of the following values:

- 0** Disable TCP syncookies.
- 1** Send out syncookies when the syn backlog queue of a socket overflows.
- 2** (since Linux 3.12) Send out syncookies unconditionally. This can be useful for network testing.

tcp_timestamps (integer; default: 1; since Linux 2.2)

Set to one of the following values to enable or disable RFC 1323 TCP timestamps:

- 0** Disable timestamps.
- 1** Enable timestamps as defined in RFC1323 and use random offset for each connection rather than only using the current time.
- 2** As for the value 1, but without random offsets. Setting *tcp_timestamps* to this value is meaningful since Linux 4.10.

tcp_tso_win_divisor (integer; default: 3; since Linux 2.6.9)

This parameter controls what percentage of the congestion window can be consumed by a single TCP Segmentation Offload (TSO) frame. The setting of this parameter is a tradeoff between burstiness and building larger TSO frames.

tcp_tw_recycle (Boolean; default: disabled; Linux 2.4 to Linux 4.11)

Enable fast recycling of TIME_WAIT sockets. Enabling this option is not recommended as the remote IP may not use monotonically increasing timestamps (devices behind NAT, devices with per-connection timestamp offsets). See RFC 1323 (PAWS) and RFC 6191.

tcp_tw_reuse (Boolean; default: disabled; since Linux 2.4.19/2.6)

Allow to reuse TIME_WAIT sockets for new connections when it is safe from protocol viewpoint. It should not be changed without advice/request of technical experts.

tcp_vegas_cong_avoid (Boolean; default: disabled; Linux 2.2 to Linux 2.6.13)

Enable TCP Vegas congestion avoidance algorithm. TCP Vegas is a sender-side-only change to TCP that anticipates the onset of congestion by estimating the bandwidth. TCP Vegas adjusts the sending rate by modifying the congestion window. TCP Vegas should provide less packet loss, but it is not as aggressive as TCP Reno.

tcp_westwood (Boolean; default: disabled; Linux 2.4.26/2.6.3 to Linux 2.6.13)

Enable TCP Westwood+ congestion control algorithm. TCP Westwood+ is a sender-side-only modification of the TCP Reno protocol stack that optimizes the performance of TCP congestion control. It is based on end-to-end bandwidth estimation to set congestion window and slow start threshold after a congestion episode. Using this estimation, TCP Westwood+ adaptively sets a slow start threshold and a congestion window which takes into account the bandwidth used at the time congestion is experienced. TCP Westwood+ significantly increases fairness with respect to TCP Reno in wired networks and throughput over wireless links.

tcp_window_scaling (Boolean; default: enabled; since Linux 2.2)

Enable RFC 1323 TCP window scaling. This feature allows the use of a large window (> 64 kB) on a TCP connection, should the other end support it. Normally, the 16 bit window length field in the TCP header limits the window size to less than 64 kB. If larger windows are desired, applications can increase the size of their socket buffers and the window scaling option will be employed. If *tcp_window_scaling* is disabled, TCP will not negotiate the use of window scaling with the other end during connection setup.

tcp_wmem (since Linux 2.4)

This is a vector of 3 integers: [min, default, max]. These parameters are used by TCP to regulate send buffer sizes. TCP dynamically adjusts the size of the send buffer from the default values listed below, in the range of these values, depending on memory available.

min Minimum size of the send buffer used by each TCP socket. The default value is the system page size. (On Linux 2.4, the default value is 4 kB.) This value is used to ensure that in memory pressure mode, allocations below this size will still succeed. This is not used to bound the size of the send buffer declared using **SO_SNDBUF** on a socket.

default The default size of the send buffer for a TCP socket. This value overwrites the initial default buffer size from the generic global */proc/sys/net/core/wmem_default* defined for all protocols. The default value is 16 kB. If larger send buffer sizes are desired, this value should be increased (to affect all sockets). To employ large TCP windows, the */proc/sys/net/ipv4/tcp_window_scaling* must be set to a nonzero value (default).

max The maximum size of the send buffer used by each TCP socket. This value does not override the value in */proc/sys/net/core/wmem_max*. This is not used to limit the size of the send buffer declared using **SO_SNDBUF** on a socket. The default value is calculated using the formula

$$\max(65536, \min(4 \text{ MB}, tcp_mem[1] * \text{PAGE_SIZE} / 128))$$

(On Linux 2.4, the default value is 128 kB, lowered 64 kB depending on low-memory systems.)

tcp_workaround_signed_windows (Boolean; default: disabled; since Linux 2.6.26)

If enabled, assume that no receipt of a window-scaling option means that the remote TCP is broken and treats the window as a signed quantity. If disabled, assume that the remote TCP is not broken even if we do not receive a window scaling option from it.

Socket options

To set or get a TCP socket option, call **getsockopt(2)** to read or **setsockopt(2)** to write the option with the option level argument set to **IPPROTO_TCP**. Unless otherwise noted, *optval* is a pointer to an *int*. In addition, most **IPPROTO_IP** socket options are valid on TCP sockets. For more information see **ip(7)**.

Following is a list of TCP-specific socket options. For details of some other socket options that are also applicable for TCP sockets, see **socket(7)**.

TCP_CONGESTION (since Linux 2.6.13)

The argument for this option is a string. This option allows the caller to set the TCP congestion control algorithm to be used, on a per-socket basis. Unprivileged processes are restricted to choosing one of the algorithms in *tcp_allowed_congestion_control* (described above). Privileged processes (**CAP_NET_ADMIN**) can choose from any of the available congestion-control algorithms (see the description of *tcp_available_congestion_control* above).

TCP_CORK (since Linux 2.2)

If set, don't send out partial frames. All queued partial frames are sent when the option is cleared again. This is useful for prepending headers before calling **sendfile(2)**, or for throughput optimization. As currently implemented, there is a 200 millisecond ceiling on the time for which output is corked by **TCP_CORK**. If this ceiling is reached, then queued data is automatically transmitted. This option can be combined with **TCP_NODELAY** only since Linux 2.5.71. This option should not be used in code intended to be portable.

TCP_DEFER_ACCEPT (since Linux 2.4)

Allow a listener to be awakened only when data arrives on the socket. Takes an integer value (seconds), this can bound the maximum number of attempts TCP will make to complete the connection. This option should not be used in code intended to be portable.

TCP_INFO (since Linux 2.4)

Used to collect information about this socket. The kernel returns a *struct tcp_info* as defined in the file */usr/include/linux/tcp.h*. This option should not be used in code intended to be portable.

TCP_KEEPCNT (since Linux 2.4)

The maximum number of keepalive probes TCP should send before dropping the connection. This option should not be used in code intended to be portable.

TCP_KEEPIDL (since Linux 2.4)

The time (in seconds) the connection needs to remain idle before TCP starts sending keepalive probes, if the socket option **SO_KEEPALIVE** has been set on this socket. This option should not be used in code intended to be portable.

TCP_KEEPIIDL (since Linux 2.4)

The time (in seconds) between individual keepalive probes. This option should not be used in code intended to be portable.

TCP_LINGER2 (since Linux 2.4)

The lifetime of orphaned **FIN_WAIT2** state sockets. This option can be used to override the system-wide setting in the file */proc/sys/net/ipv4/tcp_fin_timeout* for this socket. This is not to be confused with the **socket(7)** level option **SO_LINGER**. This option should not be used in code intended to be portable.

TCP_MAXSEG

The maximum segment size for outgoing TCP packets. In Linux 2.2 and earlier, and in Linux 2.6.28 and later, if this option is set before connection establishment, it also changes the MSS

value announced to the other end in the initial packet. Values greater than the (eventual) interface MTU have no effect. TCP will also impose its minimum and maximum bounds over the value provided.

TCP_NODELAY

If set, disable the Nagle algorithm. This means that segments are always sent as soon as possible, even if there is only a small amount of data. When not set, data is buffered until there is a sufficient amount to send out, thereby avoiding the frequent sending of small packets, which results in poor utilization of the network. This option is overridden by **TCP_CORK**; however, setting this option forces an explicit flush of pending output, even if **TCP_CORK** is currently set.

TCP_QUICKACK (since Linux 2.4.4)

Enable quickack mode if set or disable quickack mode if cleared. In quickack mode, acks are sent immediately, rather than delayed if needed in accordance to normal TCP operation. This flag is not permanent, it only enables a switch to or from quickack mode. Subsequent operation of the TCP protocol will once again enter/leave quickack mode depending on internal protocol processing and factors such as delayed ack timeouts occurring and data transfer. This option should not be used in code intended to be portable.

TCP_SYNCNT (since Linux 2.4)

Set the number of SYN retransmits that TCP should send before aborting the attempt to connect. It cannot exceed 255. This option should not be used in code intended to be portable.

TCP_USER_TIMEOUT (since Linux 2.6.37)

This option takes an *unsigned int* as an argument. When the value is greater than 0, it specifies the maximum amount of time in milliseconds that transmitted data may remain unacknowledged, or buffered data may remain untransmitted (due to zero window size) before TCP will forcibly close the corresponding connection and return **ETIMEDOUT** to the application. If the option value is specified as 0, TCP will use the system default.

Increasing user timeouts allows a TCP connection to survive extended periods without end-to-end connectivity. Decreasing user timeouts allows applications to "fail fast", if so desired. Otherwise, failure may take up to 20 minutes with the current system defaults in a normal WAN environment.

This option can be set during any state of a TCP connection, but is effective only during the synchronized states of a connection (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, and LAST-ACK). Moreover, when used with the TCP keepalive (**SO_KEEPALIVE**) option, **TCP_USER_TIMEOUT** will override keepalive to determine when to close a connection due to keepalive failure.

The option has no effect on when TCP retransmits a packet, nor when a keepalive probe is sent.

This option, like many others, will be inherited by the socket returned by **accept(2)**, if it was set on the listening socket.

Further details on the user timeout feature can be found in RFC 793 and RFC 5482 ("TCP User Timeout Option").

TCP_WINDOW_CLAMP (since Linux 2.4)

Bound the size of the advertised window to this value. The kernel imposes a minimum size of `SOCK_MIN_RCVBUF/2`. This option should not be used in code intended to be portable.

TCP_FASTOPEN (since Linux 3.6)

This option enables Fast Open (RFC 7413) on the listener socket. The value specifies the maximum length of pending SYNs (similar to the backlog argument in **listen(2)**). Once enabled, the listener socket grants the TCP Fast Open cookie on incoming SYN with TCP Fast Open option.

More importantly it accepts the data in SYN with a valid Fast Open cookie and responds SYN-ACK acknowledging both the data and the SYN sequence. **accept(2)** returns a socket that is available for read and write when the handshake has not completed yet. Thus the data exchange can commence before the handshake completes. This option requires enabling the server-side support on `sysctl net.ipv4.tcp_fastopen` (see above). For TCP Fast Open client-side support, see **send(2)**

MSG_FASTOPEN or **TCP_FASTOPEN_CONNECT** below.

TCP_FASTOPEN_CONNECT (since Linux 4.11)

This option enables an alternative way to perform Fast Open on the active side (client). When this option is enabled, **connect(2)** would behave differently depending on if a Fast Open cookie is available for the destination.

If a cookie is not available (i.e. first contact to the destination), **connect(2)** behaves as usual by sending a SYN immediately, except the SYN would include an empty Fast Open cookie option to solicit a cookie.

If a cookie is available, **connect(2)** would return 0 immediately but the SYN transmission is deferred. A subsequent **write(2)** or **sendmsg(2)** would trigger a SYN with data plus cookie in the Fast Open option. In other words, the actual connect operation is deferred until data is supplied.

Note: While this option is designed for convenience, enabling it does change the behaviors and certain system calls might set different *errno* values. With cookie present, **write(2)** or **sendmsg(2)** must be called right after **connect(2)** in order to send out SYN+data to complete 3WHS and establish connection. Calling **read(2)** right after **connect(2)** without **write(2)** will cause the blocking socket to be blocked forever.

The application should either set **TCP_FASTOPEN_CONNECT** socket option before **write(2)** or **sendmsg(2)**, or call **write(2)** or **sendmsg(2)** with **MSG_FASTOPEN** flag directly, instead of both on the same connection.

Here is the typical call flow with this new option:

```
s = socket();
setsockopt(s, IPPROTO_TCP, TCP_FASTOPEN_CONNECT, 1, ...);
connect(s);
write(s); /* write() should always follow connect()
          * in order to trigger SYN to go out. */
read(s)/write(s);
/* ... */
close(s);
```

Sockets API

TCP provides limited support for out-of-band data, in the form of (a single byte of) urgent data. In Linux this means if the other end sends newer out-of-band data the older urgent data is inserted as normal data into the stream (even when **SO_OOBINLINE** is not set). This differs from BSD-based stacks.

Linux uses the BSD compatible interpretation of the urgent pointer field by default. This violates RFC 1122, but is required for interoperability with other stacks. It can be changed via */proc/sys/net/ipv4/tcp_stdurg*.

It is possible to peek at out-of-band data using the **recv(2)** **MSG_PEEK** flag.

Since Linux 2.4, Linux supports the use of **MSG_TRUNC** in the *flags* argument of **recv(2)** (and **recvmsg(2)**). This flag causes the received bytes of data to be discarded, rather than passed back in a caller-supplied buffer. Since Linux 2.4.4, **MSG_TRUNC** also has this effect when used in conjunction with **MSG_OOB** to receive out-of-band data.

Ioctls

The following **ioctl(2)** calls return information in *value*. The correct syntax is:

```
int value;
error = ioctl(tcp_socket, ioctl_type, &value);
```

ioctl_type is one of the following:

SIOCINQ

Returns the amount of queued unread data in the receive buffer. The socket must not be in LISTEN state, otherwise an error (**EINVAL**) is returned. **SIOCINQ** is defined in *<linux/sockios.h>*.

Alternatively, you can use the synonymous **FIONREAD**, defined in `<sys/ioctl.h>`.

SIOCATMARK

Returns true (i.e., *value* is nonzero) if the inbound data stream is at the urgent mark.

If the **SO_OOBINLINE** socket option is set, and **SIOCATMARK** returns true, then the next read from the socket will return the urgent data. If the **SO_OOBINLINE** socket option is not set, and **SIOCATMARK** returns true, then the next read from the socket will return the bytes following the urgent data (to actually read the urgent data requires the **recv(MSG_OOB)** flag).

Note that a read never reads across the urgent mark. If an application is informed of the presence of urgent data via **select(2)** (using the *exceptfds* argument) or through delivery of a **SIGURG** signal, then it can advance up to the mark using a loop which repeatedly tests **SIOCATMARK** and performs a read (requesting any number of bytes) as long as **SIOCATMARK** returns false.

SIOCOUTQ

Returns the amount of unsent data in the socket send queue. The socket must not be in LISTEN state, otherwise an error (**EINVAL**) is returned. **SIOCOUTQ** is defined in `<linux/sockios.h>`. Alternatively, you can use the synonymous **TIOCOUTQ**, defined in `<sys/ioctl.h>`.

Error handling

When a network error occurs, TCP tries to resend the packet. If it doesn't succeed after some time, either **ETIMEDOUT** or the last received error on this connection is reported.

Some applications require a quicker error notification. This can be enabled with the **IPPROTO_IP** level **IP_RECVERR** socket option. When this option is enabled, all incoming errors are immediately passed to the user program. Use this option with care — it makes TCP less tolerant to routing changes and other normal network conditions.

ERRORS

EAFNOTSUPPORT

Passed socket address type in *sin_family* was not **AF_INET**.

EPIPE The other end closed the socket unexpectedly or a read is executed on a shut down socket.

ETIMEDOUT

The other end didn't acknowledge retransmitted data after some time.

Any errors defined for **ip(7)** or the generic socket layer may also be returned for TCP.

VERSIONS

Support for Explicit Congestion Notification, zero-copy **sendfile(2)**, reordering support and some SACK extensions (DSACK) were introduced in Linux 2.4. Support for forward acknowledgement (FACK), **TIME_WAIT** recycling, and per-connection keepalive socket options were introduced in Linux 2.3.

BUGS

Not all errors are documented.

IPv6 is not described.

SEE ALSO

accept(2), **bind(2)**, **connect(2)**, **getsockopt(2)**, **listen(2)**, **recvmsg(2)**, **sendfile(2)**, **sendmsg(2)**, **socket(2)**, **ip(7)**, **socket(7)**

The kernel source file *Documentation/networking/ip-sysctl.txt*.

RFC 793 for the TCP specification.

RFC 1122 for the TCP requirements and a description of the Nagle algorithm.

RFC 1323 for TCP timestamp and window scaling options.

RFC 1337 for a description of **TIME_WAIT** assassination hazards.

RFC 3168 for a description of Explicit Congestion Notification.

RFC 2581 for TCP congestion control algorithms.

RFC 2018 and RFC 2883 for SACK and extensions to SACK.