# Transmission Control Protocol

From Wikipedia, the free encyclopedia

The **Transmission Control Protocol** (**TCP**) is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol (IP). Therefore, the entire suite is commonly referred to as *TCP/IP*. TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating by an IP network. Major Internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP. Applications that do not require reliable data stream service may use the User Datagram Protocol (UDP), which provides a connectionless datagram service that emphasizes reduced latency over reliability.

## Contents

- 1 Historical origin
- 2 Network function
- 3 TCP segment structure
- 4 Protocol operation
  - 4.1 Connection establishment
  - 4.2 Connection termination
  - 4.3 Resource usage
  - 4.4 Data transfer
    - 4.4.1 Reliable transmission
      - 4.4.1.1 Dupack based retransmission
      - 4.4.1.2 Timeout based retransmission
    - 4.4.2 Error detection
    - 4.4.3 Flow control
    - 4.4.4 Congestion control
  - 4.5 Maximum segment size
  - 4.6 Selective acknowledgments
  - 4.7 Window scaling
  - 4.8 TCP timestamps
  - 4.9 Out-of-band data
  - 4.10 Forcing data delivery
- 5 Vulnerabilities
  - 5.1 Denial of service
  - 5.2 Connection hijacking
  - 5.3 TCP veto
- 6 TCP ports
- 7 Development
- 8 TCP over wireless networks
- 9 Hardware implementations
- 10 Debugging
- 11 Alternatives
- 12 Checksum computation
  - 12.1 TCP checksum for IPv4
  - 12.2 TCP checksum for IPv6
  - 12.3 Checksum offload
- 13 See also
- 14 References
- 15 Further reading
- 16 External links
  - 16.1 RFC
  - 16.2 Others

## Historical origin

During May 1974, the Institute of Electrical and Electronic Engineers (IEEE) published a paper titled "*A Protocol for Packet Network Intercommunication.*"[1] The paper's authors, Vint Cerf and Bob Kahn, described an internetworking protocol for sharing resources using packet-switching among the nodes. A central control component of this model was the *Transmission Control Program* that incorporated both connection-oriented links and datagram services between hosts. The monolithic Transmission Control Program was later divided into a modular architecture consisting of the *Transmission Control Protocol* at the connection-oriented layer and the *Internet Protocol* at the internetworking (datagram) layer. The model became known informally as *TCP/IP*, although formally it was henceforth termed the *Internet Protocol Suite*.

## Network function

The Transmission Control Protocol provides a communication service at an intermediate level between an application program and the Internet Protocol. It provides host-to-host connectivity at the Transport Layer of the Internet model. An application does not need to know the particular mechanisms for sending data via a link to another host, such as the required packet fragmentation on the transmission medium. At the transport layer, the protocol handles all handshaking and transmission details and just presents an abstraction of the network connection to the application.

At the lower levels of the protocol stack, due to network congestion, traffic load balancing, or other unpredictable network behaviour, IP packets may be lost, duplicated, or delivered out of order. TCP detects these problems, requests re-transmission of lost data, rearranges out-of-order data and even helps minimise network congestion to reduce the occurrence of the other problems. If the data still remains undelivered, its source is notified of this failure. Once the TCP receiver has reassembled the sequence of octets originally transmitted, it passes them to the receiving application. Thus, TCP abstracts the application's communication from the underlying networking details.

TCP is used extensively by many applications available by internet, including the World Wide Web (WWW), E-mail, File Transfer Protocol, Secure Shell, peer-to-peer file sharing, and streaming media applications.

TCP is optimised for accurate delivery rather than timely delivery. Therefore, TCP sometimes incurs relatively long delays (on the order of seconds) while waiting for out-of-order messages or re-transmissions of lost messages. It is not particularly suitable for real-time applications such as Voice over IP. For such applications, protocols like the Real-time Transport Protocol (RTP) operating over the User Datagram Protocol (UDP) are usually recommended instead.[2]

TCP is a reliable stream delivery service which guarantees that all bytes received will be identical with bytes sent and in the correct order. Since packet transfer by many networks is not reliable, a technique known as 'positive acknowledgement with re-transmission' is used to guarantee reliability of packet transfers. This fundamental technique requires the receiver to respond with an acknowledgement message as it receives the data. The sender keeps a record of each packet it sends and maintains a timer from when the packet was sent. The sender re-transmits a packet if the timer expires before the message has been acknowledged. The timer is needed in case a packet gets lost or corrupted.[2]

While IP handles actual delivery of the data, TCP keeps track of 'segments' - the individual units of data transmission that a message is divided into for efficient routing through the network. For example, when an HTML file is sent from a web server, the TCP software layer of that server divides the sequence of file octets into segments and forwards them individually to the IP software layer (Internet Layer). The Internet Layer encapsulates each TCP segment into an IP packet by adding a header that includes (among other data) the destination IP address. When the client program on the destination computer receives them, the TCP layer (Transport Layer) re-assembles the individual segments and ensures they are correctly ordered and error-free as it streams them to an application.

## TCP segment structure

Transmission Control Protocol accepts data from a data stream, divides it into chunks, and adds a TCP header creating a TCP segment. The TCP segment is then encapsulated into an Internet Protocol (IP) datagram, and exchanged with peers.[3]

The term *TCP packet* appears in both informal and formal usage, whereas in more precise terminology *segment* refers to the TCP protocol data unit (PDU), *datagram*[4] to the IP PDU, and *frame* to the data link layer PDU:

> Processes transmit data by calling on the TCP and passing buffers of data as arguments. The TCP packages the data from these buffers into segments and calls on the internet module [e.g. IP] to transmit each segment to the destination TCP.[5]

A TCP segment consists of a segment *header* and a *data* section. The TCP header contains 10 mandatory fields, and an optional extension field (*Options*, pink background in table).

The data section follows the header. Its contents are the payload data carried for the application. The length of the data section is not specified in the TCP segment header. It can be calculated by subtracting the combined length of the TCP header and the encapsulating IP header from the total IP datagram length (specified in the IP header).

**TCP Header**

| ***Offsets*** | **Octet** | **0** | | | | | | | | **1** | | | | | | | | **2** | | | | | | | | **3** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Octet** | Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| **0** | 0 | Source port | | | | | | | | | | | | | | | | Destination port | | | | | | | | | | | | | | | |
| **4** | 32 | Sequence number | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **8** | 64 | Acknowledgment number (if ACK set) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **12** | 96 | Data offset | | | | Reserved<br>0 0 0 | | | NS | CWR | ECE | URG | ACK | PSH | RST | SYN | FIN | Window Size | | | | | | | | | | | | | | | |
| **16** | 128 | Checksum | | | | | | | | | | | | | | | | Urgent pointer (if URG set) | | | | | | | | | | | | | | | |
| **20**<br>**...** | 160<br>... | Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.)<br>... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

**Source port (16 bits)**
identifies the sending port

**Destination port (16 bits)**
identifies the receiving port

**Sequence number (32 bits)**
has a dual role:

- If the SYN flag is set (1), then this is the initial sequence number. The sequence number of the actual first data byte and the acknowledged number in the corresponding ACK are then this sequence number plus 1.
- If the SYN flag is clear (0), then this is the accumulated sequence number of the first data byte of this segment for the current session.

**Acknowledgment number (32 bits)**
if the ACK flag is set then the value of this field is the next sequence number that the sender is expecting. This acknowledges receipt of all prior bytes (if any). The first ACK sent by each end acknowledges the other end's initial sequence number itself, but no data.

**Data offset (4 bits)**
specifies the size of the TCP header in 32-bit words. The minimum size header is 5 words and the maximum is 15 words thus giving the minimum size of 20 bytes and maximum of 60 bytes,

allowing for up to 40 bytes of options in the header. This field gets its name from the fact that it is also the offset from the start of the TCP segment to the actual data.

**Reserved (3 bits)**

for future use and should be set to zero

**Flags (9 bits) (aka Control bits)**

contains 9 1-bit flags

- NS (1 bit) – ECN-nonce concealment protection (experimental: see RFC 3540).
- CWR (1 bit) – Congestion Window Reduced (CWR) flag is set by the sending host to indicate that it received a TCP segment with the ECE flag set and had responded in congestion control mechanism (added to header by RFC 3168).
- ECE (1 bit) – ECN-Echo has a dual role, depending on the value of the SYN flag. It indicates:
  - If the SYN flag is set (1), that the TCP peer is ECN capable.
  - If the SYN flag is clear (0), that a packet with Congestion Experienced flag set (ECN=11) in IP header received during normal transmission (added to header by RFC 3168). This serves as an indication of network congestion (or impending congestion) to the TCP sender.
- URG (1 bit) – indicates that the Urgent pointer field is significant
- ACK (1 bit) – indicates that the Acknowledgment field is significant. All packets after the initial SYN packet sent by the client should have this flag set.
- PSH (1 bit) – Push function. Asks to push the buffered data to the receiving application.
- RST (1 bit) – Reset the connection
- SYN (1 bit) – Synchronize sequence numbers. Only the first packet sent from each end should have this flag set. Some other flags and fields change meaning based on this flag, and some are only valid for when it is set, and others when it is clear.
- FIN (1 bit) – No more data from sender

**Window size (16 bits)**

the size of the *receive window*, which specifies the number of window size units (by default, bytes) (beyond the segment identified by the sequence number in the acknowledgment field) that the sender of this segment is currently willing to receive (*see Flow control and Window Scaling*)

**Checksum (16 bits)**

The 16-bit checksum field is used for error-checking of the header and data

**Urgent pointer (16 bits)**

if the URG flag is set, then this 16-bit field is an offset from the sequence number indicating the last urgent data byte

**Options (Variable 0–320 bits, divisible by 32)**

The length of this field is determined by the data offset field. Options have up to three fields: Option-Kind (1 byte), Option-Length (1 byte), Option-Data (variable). The Option-Kind field indicates the type of option, and is the only field that is not optional. Depending on what kind of option we are dealing with, the next two fields may be set: the Option-Length field indicates the total length of the option, and the Option-Data field contains the value of the option, if applicable. For example, an Option-Kind byte of 0x01 indicates that this is a No-Op option used only for padding, and does not have an Option-Length or Option-Data byte following it. An Option-Kind byte of 0 is the End Of Options option, and is also only one byte. An Option-Kind byte of 0x02 indicates that this is the Maximum Segment Size option, and will be followed by a byte specifying the length of the MSS field (should be 0x04). This length is the total length of the given options field, including Option-Kind and Option-Length bytes. So while the MSS value is typically expressed in two bytes, the length of the field will be 4 bytes (+2 bytes of kind and length). In short, an MSS option field with a value of 0x05B4 will show up as (0x02 0x04 0x05B4) in the TCP options section.

Some options may only be sent when SYN is set; they are indicated below as [SYN]. Option-Kind and standard lengths given as (Option-Kind,Option-Length).

- 0 (8 bits) – End of options list
- 1 (8 bits) – No operation (NOP, Padding) This may be used to align option fields on 32-bit boundaries for better performance.
- 2,4,*SS* (32 bits) – Maximum segment size (*see maximum segment size*) [SYN]
- 3,3,*S* (24 bits) – Window scale (*see window scaling for details*) [SYN][6]
- 4,2 (16 bits) – Selective Acknowledgement permitted. [SYN] (*See selective acknowledgments for*

*details*)[7]

- 5,*N,BBBB,EEEE,...* (variable bits, *N* is either 10, 18, 26, or 34)- Selective ACKnowledgement (SACK)[8] These first two bytes are followed by a list of 1–4 blocks being selectively acknowledged, specified as 32-bit begin/end pointers.
- 8,10,*TTTT,EEEE* (80 bits)- Timestamp and echo of previous timestamp (*see TCP timestamps for details*)[9]

(The remaining options are historical, obsolete, experimental, not yet standardized, or unassigned)

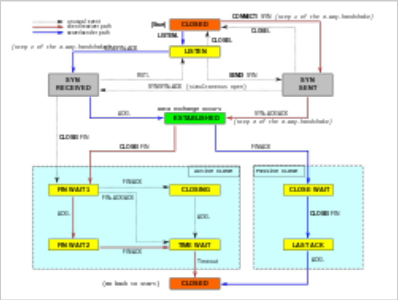
**Padding**

The TCP header padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros.[10]

# Protocol operation

TCP protocol operations may be divided into three phases. Connections must be properly established in a multi-step handshake process (*connection establishment*) before entering the *data transfer* phase. After data transmission is completed, the *connection termination* closes established virtual circuits and releases all allocated resources.

A TCP connection is managed by an operating system through a programming interface that represents the local end-point for communications, the *Internet socket*. During the lifetime of a TCP connection the local end-point undergoes a series of state changes:[11]



A Simplified TCP State Diagram. See TCP EFSM diagram (http://www.medianet.kent.edu/techreports/TR2005-07-22-tcp-EFSM.pdf) for a more detailed state diagram including the states inside the ESTABLISHED state.

**LISTEN**

(server) represents waiting for a connection request from any remote TCP and port.

**SYN-SENT**

(client) represents waiting for a matching connection request after having sent a connection request.

**SYN-RECEIVED**

(server) represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

**ESTABLISHED**

(both server and client) represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.

**FIN-WAIT-1**

(both server and client) represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

**FIN-WAIT-2**

(both server and client) represents waiting for a connection termination request from the remote TCP.

**CLOSE-WAIT**

(both server and client) represents waiting for a connection termination request from the local user.

**CLOSING**

(both server and client) represents waiting for a connection termination request acknowledgment from the remote TCP.

**LAST-ACK**

(both server and client) represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).

**TIME-WAIT**

(either server or client) represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request. [According to RFC 793 a connection can stay in TIME-WAIT for a maximum of four minutes known as two MSL (maximum segment lifetime).]

**CLOSED**

(both server and client) represents no connection state at all.

## Connection establishment

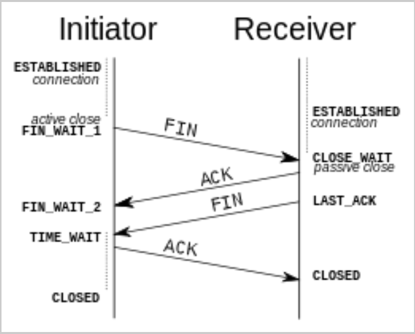
To establish a connection, TCP uses a three-way handshake. Before a client attempts to connect with a server, the server must first bind to and listen at a port to open it up for connections: this is called a passive open. Once the passive open is established, a client may initiate an active open. To establish a connection, the three-way (or 3-step) handshake occurs:

1. **SYN**: The active open is performed by the client sending a SYN to the server. The client sets the segment's sequence number to a random value A.
2. **SYN-ACK**: In response, the server replies with a SYN-ACK. The acknowledgment number is set to one more than the received sequence number i.e. A+1, and the sequence number that the server chooses for the packet is another random number, B.
3. **ACK**: Finally, the client sends an ACK back to the server. The sequence number is set to the received acknowledgement value i.e. A+1, and the acknowledgement number is set to one more than the received sequence number i.e. B+1.

At this point, both the client and server have received an acknowledgment of the connection. The steps 1, 2 establish the connection parameter (sequence number) for one direction and it is acknowledged. The steps 2, 3 establish the connection parameter (sequence number) for the other direction and it is acknowledged. With these, a full-duplex communication is established.

## Connection termination

The connection termination phase uses a four-way handshake, with each side of the connection terminating independently. When an endpoint wishes to stop its half of the connection, it transmits a FIN packet, which the other end acknowledges with an ACK. Therefore, a typical tear-down requires a pair of FIN and ACK segments from each TCP endpoint. After the side that sent the first FIN has responded with the final ACK, it waits for a timeout before finally closing the connection, during which time the local port is unavailable for new connections; this prevents confusion due to delayed packets being delivered during subsequent connections.



Connection termination

A connection can be "half-open", in which case one side has terminated its end, but the other has not. The side that has terminated can no longer send any data into the connection, but the other side can. The terminating side should continue reading the data until the other side terminates as well.

It is also possible to terminate the connection by a 3-way handshake, when host A sends a FIN and host B replies with a FIN & ACK (merely combines 2 steps into one) and host A replies with an ACK.[12]

Some host TCP stacks may implement a half-duplex close sequence, as Linux or HP-UX do. If such a host actively closes a connection but still has not read all the incoming data the stack already received from the link, this host sends a RST instead of a FIN (Section 4.2.2.13 in RFC 1122). This allows a TCP application to be sure the remote application has read all the data the former sent—waiting the FIN from the remote side, when it actively closes the connection. But the remote TCP stack cannot distinguish between a *Connection Aborting RST* and *Data Loss RST*. Both cause the remote stack to lose all the data received.

Some application protocols may violate the OSI model layers, using the TCP open/close handshaking for the application protocol open/close handshaking — these may find the RST problem on active close.

As an example:

```
s = connect(remote);
send(s, data);
close(s);
```

For a usual program flow like above, a TCP/IP stack like that described above does not guarantee that all the data arrives to the other application.

## Resource usage

Most implementations allocate an entry in a table that maps a session to a running operating system process. Because TCP packets do not include a session identifier, both endpoints identify the session using the client's address and port. Whenever a packet is received, the TCP implementation must perform a lookup on this table to find the destination process. Each entry in the table is known as a Transmission Control Block or TCB. It contains information about the endpoints (IP and port), status of the connection, running data about the packets that are being exchanged and buffers for sending and receiving data.

The number of sessions in the server side is limited only by memory and can grow as new connections arrive, but the client must allocate a random port before sending the first SYN to the server. This port remains allocated during the whole conversation, and effectively limits the number of outgoing connections from each of the client's IP addresses. If an application fails to properly close unrequired connections, a client can run out of resources and become unable to establish new TCP connections, even from other applications.

Both endpoints must also allocate space for unacknowledged packets and received (but unread) data.

## Data transfer

There are a few key features that set TCP apart from User Datagram Protocol:

- Ordered data transfer — the destination host rearranges according to sequence number[2]
- Retransmission of lost packets — any cumulative stream not acknowledged is retransmitted[2]
- Error-free data transfer[13]
- Flow control — limits the rate a sender transfers data to guarantee reliable delivery. The receiver continually hints the sender on how much data can be received (controlled by the sliding window). When the receiving host's buffer fills, the next acknowledgment contains a 0 in the window size, to stop transfer and allow the data in the buffer to be processed.[2]
- Congestion control[2]

### Reliable transmission

TCP uses a *sequence number* to identify each byte of data. The sequence number identifies the order of the bytes sent from each computer so that the data can be reconstructed in order, regardless of any packet reordering, or packet loss that may occur during transmission. The sequence number of the first byte is decided during the initial 3-way handshake. This number can be arbitrary, and should in fact be unpredictable to defend against TCP sequence prediction attacks.

Acknowledgements (Acks) are sent by the receiver of data to tell the sender that data has been received. Acks do not imply that the data has been delivered to the application. They merely signify that it is now the receiver's responsibility to deliver the data. In TCP Acks are cumulative. That is, if the $i^{th}$ byte is acknowledged, it means that all previous bytes have been received too.

Reliability is achieved by the sender detecting lost data and retransmitting it. TCP uses two primary techniques to identify loss. Retransmission timeout (abbreviated as RTO) and duplicate cumulative acknowledgements (DupAcks).

#### Dupack based retransmission

If a single packet (say packet 100) in a stream is lost, then the receiver cannot acknowledge packets above 100 because it uses cumulative acks. Hence the receiver acknowledges packet 100 again on the receipt of another data packet. This duplicate acknowledgement is used as a signal for packet loss. That is, if the sender receives three duplicate acknowledgements, it retransmits the last unacknowledged packet. A threshold of three is used because the network may reorder packets causing duplicate acknowledgements. This threshold has been demonstrated to avoid spurious retransmissions due to reordering.[14] Sometimes selective acknowledgements (SACKs) are used to give more explicit feedback on which packets have been received. This greatly improves TCP's ability to retransmit the right packets.

#### Timeout based retransmission

Whenever a packet is sent, the sender sets a timer that is a conservative estimate of when that packet will be acked. If the sender does not receive an ack by then, it transmits that packet again. The timer is reset every time the sender receives an acknowledgement. This means that the retransmit timer fires only when the sender has received *no* acknowledgement for a long time. Typically the timer value is set to $smoothedRTT + max(G, 4 * RTTvariation)$ where G is the clock granularity.[15] Further, in case a retransmit timer has fired and still no acknowledgement is received, the next timer is set to twice the previous value (up to a certain threshold). Among other things, this helps defend against a man-in-the-middle denial of service attack that tries to fool the sender into making so many retransmissions that the receiver is overwhelmed.

If the sender infers that data has been lost in the network using one of the two techniques described above, it retransmits the data.

### Error detection

Sequence numbers allow receivers to discard duplicate packets and properly sequence reordered packets. Acknowledgments allow senders to determine when to retransmit lost packets.

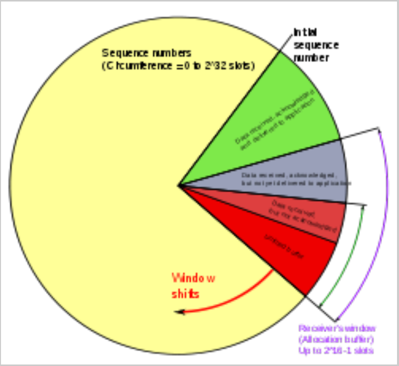
To assure correctness a checksum field is included; see checksum computation section for details on checksumming. The TCP checksum is a weak check by modern standards. Data Link Layers with high bit error rates may require additional link error correction/detection capabilities. The weak checksum is partially compensated for by the common use of a CRC or better integrity check at layer 2, below both TCP and IP, such as is used in PPP or the Ethernet frame. However, this does not mean that the 16-bit TCP checksum is redundant: remarkably, introduction of errors in packets between CRC-protected hops is common, but the end-to-end 16-bit TCP checksum catches most of these simple errors.[16] This is the end-to-end principle at work.

### Flow control

TCP uses an end-to-end flow control protocol to avoid having the sender send data too fast for the TCP receiver to receive and process it reliably. Having a mechanism for flow control is essential in an environment where machines of diverse network speeds communicate. For example, if a PC sends data to a smartphone that is slowly processing received data, the smartphone must regulate the data flow so as not to be overwhelmed.[2]

TCP uses a sliding window flow control protocol. In each TCP segment, the receiver specifies in the *receive window* field the amount of additionally received data (in bytes) that it is willing to buffer for the connection. The sending host can send only up to that amount of data before it must wait for an acknowledgment and window update from the receiving host.

When a receiver advertises a window size of 0, the sender stops sending data and starts the *persist timer*. The persist timer is used to protect TCP from a deadlock situation that could arise if a subsequent window size update from the receiver is lost, and the sender cannot send more data until receiving a new window size update from the receiver. When the persist timer expires, the TCP sender attempts recovery by sending a small packet so that the receiver responds by sending another acknowledgement containing the new window size.


Sequence numbers
(Circumference =0 to 2^32 slots)
Initial sequence number
Data received, acknowledged and delivered to application
Data received, acknowledged, but not yet delivered to application
Data received but not acknowledged
Unfilled buffer
Window shifts
Receiver's window
(Allocation buffer)
Up to 2^16-1 slots

TCP sequence numbers and receive windows behave very much like a clock. The receive window shifts each time the receiver receives and acknowledges a new segment of data. Once it runs out of sequence numbers, the sequence number loops back to 0.

If a receiver is processing incoming data in small increments, it may repeatedly advertise a small receive window. This is referred to as the silly window syndrome, since it is inefficient to send only a few bytes of data in a TCP segment, given the relatively large overhead of the TCP header.

### Congestion control

The final main aspect of TCP is congestion control. TCP uses a number of mechanisms to achieve high performance and avoid congestion collapse, where network performance can fall by several orders of magnitude. These mechanisms control the rate of data entering the network, keeping the data flow below a rate that would trigger collapse. They also yield an approximately max-min fair allocation between flows.

Acknowledgments for data sent, or lack of acknowledgments, are used by senders to infer network conditions between the TCP sender and receiver. Coupled with timers, TCP senders and receivers can alter the behavior of the flow of data. This is more generally referred to as congestion control and/or network congestion avoidance.

Modern implementations of TCP contain four intertwined algorithms: slow-start, congestion avoidance, fast retransmit, and fast recovery (RFC 5681).

In addition, senders employ a *retransmission timeout* (RTO) that is based on the estimated round-trip time (or RTT) between the sender and receiver, as well as the variance in this round trip time. The behavior of this timer is specified in RFC 6298. There are subtleties in the estimation of RTT. For example, senders must be careful when calculating RTT samples for retransmitted packets; typically they use Karn's Algorithm or TCP timestamps (see RFC 1323). These individual RTT samples are then averaged over time to create a Smoothed Round Trip Time (SRTT) using Jacobson's algorithm. This SRTT value is what is finally used as the round-trip time estimate.

Enhancing TCP to reliably handle loss, minimize errors, manage congestion and go fast in very high-speed environments are ongoing areas of research and standards development. As a result, there are a number of TCP congestion avoidance algorithm variations.

## Maximum segment size

The maximum segment size (MSS) is the largest amount of data, specified in bytes, that TCP is willing to receive in a single segment. For best performance, the MSS should be set small enough to avoid IP fragmentation, which can lead to packet loss and excessive retransmissions. To try to accomplish this, typically the MSS is announced by each side using the MSS option when the TCP connection is established, in which case it is derived from the maximum transmission unit (MTU) size of the data link layer of the networks to which the sender and receiver are directly attached. Furthermore, TCP senders can use path MTU discovery to infer the minimum MTU along the network path between the sender and receiver, and use this to dynamically adjust the MSS to avoid IP fragmentation within the network.

MSS announcement is also often called "MSS negotiation". Strictly speaking, the MSS is not "negotiated" between the originator and the receiver, because that would imply that both originator and receiver will negotiate and agree upon a single, unified MSS that applies to all communication in both directions of the connection. In fact, two completely independent values of MSS are permitted for the two directions of data flow in a TCP connection.[17] This situation may arise, for example, if one of the devices participating in a connection has an extremely limited amount of memory reserved (perhaps even smaller than the overall discovered Path MTU) for processing incoming TCP segments.

## Selective acknowledgments

Relying purely on the cumulative acknowledgment scheme employed by the original TCP protocol can lead to inefficiencies when packets are lost. For example, suppose 10,000 bytes are sent in 10 different TCP packets, and the first packet is lost during transmission. In a pure cumulative acknowledgment protocol, the receiver cannot say that it received bytes 1,000 to 9,999 successfully, but failed to receive the first packet, containing bytes 0 to 999. Thus the sender may then have to resend all 10,000 bytes.

To alleviate this issue TCP employs the *selective acknowledgment (SACK)* option, defined in RFC 2018, which **allows the receiver to acknowledge discontinuous blocks of packets which were received correctly**, in addition to the sequence number of the last contiguous byte received successively, as in the basic TCP acknowledgment. The acknowledgement can specify a number of *SACK blocks*, where each SACK block is conveyed by the starting and ending sequence numbers of a contiguous range that the receiver correctly received. In the example above, the receiver would send SACK with sequence numbers 1000 and 9999. The sender would accordingly retransmit only the first packet (bytes 0 to 999).

A TCP sender can interpret an out-of-order packet delivery as a lost packet. If it does so, the TCP sender will retransmit the packet previous to the out-of-order packet and slow its data delivery rate for that connection. The duplicate-SACK option, an extension to the SACK option that was defined in RFC 2883, solves this problem. The TCP receiver sends a D-ACK to indicate that no packets were lost, and the TCP sender can then reinstate the higher transmission-rate.

The SACK option is not mandatory, and comes into operation only if both parties support it. This is negotiated when a connection is established. SACK uses the optional part of the TCP header (see TCP segment structure for details). The use of SACK has become widespread — all popular TCP stacks support it. Selective acknowledgment is also used in Stream Control Transmission Protocol (SCTP).

## Window scaling

For more efficient use of high-bandwidth networks, a larger TCP window size may be used. The TCP window size field controls the flow of data and its value is limited to between 2 and 65,535 bytes.

Since the size field cannot be expanded, a scaling factor is used. The TCP window scale option, as defined in RFC 1323, is an option used to increase the maximum window size from 65,535 bytes to 1 gigabyte. Scaling up to larger window sizes is a part of what is necessary for TCP tuning.

The window scale option is used only during the TCP 3-way handshake. The window scale value represents the number of bits to left-shift the 16-bit window size field. The window scale value can be set from 0 (no shift) to 14 for each direction independently. Both sides must send the option in their SYN segments to enable window scaling in either direction.

Some routers and packet firewalls rewrite the window scaling factor during a transmission. This causes sending and receiving sides to assume different TCP window sizes. The result is non-stable traffic that may be very slow. The problem is visible on some sites behind a defective router.[18]

## TCP timestamps

TCP timestamps, defined in RFC 1323, can help TCP determine in which order packets were sent. TCP timestamps are not normally aligned to the system clock and start at some random value. Many operating systems will increment the timestamp for every elapsed millisecond; however the RFC only states that the ticks should be proportional.

There are two timestamp fields:

a 4-byte sender timestamp value (my timestamp)
a 4-byte echo reply timestamp value (the most recent timestamp received from you).

TCP timestamps are used in an algorithm known as *Protection Against Wrapped Sequence* numbers, or *PAWS* (see RFC 1323 for details). PAWS is used when the receive window crosses the sequence number wraparound boundary. In the case where a packet was potentially retransmitted it answers the question: "Is this sequence number in the first 4 GB or the second?" And the timestamp is used to break the tie.

Also, the Eifel detection algorithm (RFC 3522) uses TCP timestamps to determine if retransmissions are occurring because packets are lost or simply out of order.

### Out-of-band data

It is possible to interrupt or abort the queued stream instead of waiting for the stream to finish. This is done by specifying the data as *urgent*. This tells the receiving program to process it immediately, along with the rest of the urgent data. When finished, TCP informs the application and resumes back to the stream queue. An example is when TCP is used for a remote login session, the user can send a keyboard sequence that interrupts or aborts the program at the other end. These signals are most often needed when a program on the remote machine fails to operate correctly. The signals must be sent without waiting for the program to finish its current transfer.[2]

TCP OOB data was not designed for the modern Internet. The *urgent* pointer only alters the processing on the remote host and doesn't expedite any processing on the network itself. When it gets to the remote host there are two slightly different interpretations of the protocol, which means only single bytes of OOB data are reliable. This is assuming it is reliable at all as it is one of the least commonly used protocol elements and tends to be poorly implemented. [19][20]

### Forcing data delivery

Normally, TCP waits for 200 ms for a full packet of data to send (Nagle's Algorithm tries to group small messages into a single packet). This wait creates small, but potentially serious delays if repeated constantly during a file transfer. For example, a typical send block would be 4 KB, a typical MSS is 1460, so 2 packets go out on a 10 Mbit/s ethernet taking ~1.2 ms each followed by a third carrying the remaining 1176 after a 197 ms pause because TCP is waiting for a full buffer.

In the case of telnet, each user keystroke is echoed back by the server before the user can see it on the screen. This delay would become very annoying.

Setting the socket option `TCP_NODELAY` overrides the default 200 ms send delay. Application programs use this socket option to force output to be sent after writing a character or line of characters.

The RFC defines the `PSH` push bit as "a message to the receiving TCP stack to send this data immediately up to the receiving application".[2] There is no way to indicate or control it in user space using Berkeley sockets and it is controlled by protocol stack only.[21]

## Vulnerabilities

TCP may be attacked in a variety of ways. The results of a thorough security assessment of TCP, along with possible mitigations for the identified issues, were published in 2009,[22] and is currently being pursued within the IETF.[23]

### Denial of service

By using a spoofed IP address and repeatedly sending purposely assembled SYN packets, followed by many ACK packets, attackers can cause the server to consume large amounts of resources keeping track of the bogus connections. This is known as a SYN flood attack. Proposed solutions to this problem include SYN cookies and cryptographic puzzles, though SYN cookies come with their own set of vulnerabilities.[24] Sockstress is a similar attack, that might be mitigated with system resource management.[25] An advanced DoS attack involving the exploitation of the TCP Persist Timer was analyzed in Phrack #66.[26]

### Connection hijacking

An attacker who is able to eavesdrop a TCP session and redirect packets can hijack a TCP connection. To do so, the attacker learns the sequence number from the ongoing communication and forges a false segment that looks like the next segment in the stream. Such a simple hijack can result in one packet being erroneously accepted at one end. When the receiving host acknowledges the extra segment to the other side of the connection, synchronization is lost. Hijacking might be combined with Address Resolution Protocol (ARP) or routing attacks that allow taking control of the packet flow, so as to get permanent control of the hijacked TCP connection.[27]

Impersonating a different IP address was not difficult prior to RFC 1948, when the initial *sequence number* was easily guessable. That allowed an attacker to blindly send a sequence of packets that the receiver would believe to come from a different IP address, without the need to deploy ARP or routing attacks: it is enough to ensure that the legitimate host of the impersonated IP address is down, or bring it to that condition using denial-of-service attacks. This is why the initial sequence number is now chosen at random.

### TCP veto

An attacker who can eavesdrop and predict the size of the next packet to be sent can cause the receiver to accept a malicious payload without disrupting the existing connection. The attacker injects a malicious packet with the sequence number and a payload size of the next expected packet. When the legitimate packet is ultimately received, it is found to have the same sequence number and length as a packet already received and is silently dropped as a normal duplicate packet—the legitimate packet is "vetoed" by the malicious packet. Unlike in connection hijacking, the connection is never desynchronized and communication continues as normal after the malicious payload is accepted. TCP veto gives the attacker less control over the communication, but makes the attack particularly resistant to detection. The large increase in network traffic from the ACK storm is avoided. The only evidence to the receiver that something is amiss is a single duplicate packet, a normal occurrence in an IP network. The sender of the vetoed packet never sees any evidence of an attack.[28]

Another vulnerability is TCP reset attack.

## TCP ports

TCP and UDP use port numbers to identify sending and receiving application end-points on a host, often called Internet sockets. Each side of a TCP connection has an associated 16-bit unsigned port number (0-65535) reserved by the sending or receiving application. Arriving TCP packets are identified as belonging to a specific TCP connection by its sockets, that is, the combination of source host address, source port, destination host address, and destination port. This means that a server computer can provide several clients with several services simultaneously, as long as a client takes care of initiating any simultaneous connections to one destination port from different source ports.

Port numbers are categorized into three basic categories: well-known, registered, and dynamic/private. The well-known ports are assigned by the Internet Assigned Numbers Authority (IANA) and are typically used by system-level or root processes. Well-known applications running as servers and passively listening for connections typically use these ports. Some examples include: FTP (20 and 21), SSH (22), TELNET (23), SMTP (25), HTTP over SSL/TLS (443), and HTTP (80). Registered ports are typically used by end user applications as ephemeral source ports when contacting servers, but they can also identify named services that have been registered by a third party. Dynamic/private ports can also be used by end user applications, but are less commonly so. Dynamic/private ports do not contain any meaning outside of any particular TCP connection.

Network Address Translation (NAT), typically uses dynamic port numbers, on the ("Internet-facing") public side, to disambiguate the flow of traffic that is passing between a public network and a private subnetwork, thereby allowing many IP addresses (and their ports) on the subnet to be serviced by a single public-facing address.

## Development

TCP is a complex protocol. However, while significant enhancements have been made and proposed over the years, its most basic operation has not changed significantly since its first specification RFC 675 in 1974, and the v4 specification RFC 793, published in September 1981. RFC 1122, Host Requirements for Internet Hosts, clarified a number of TCP protocol implementation requirements. A list of the 8 required specifications and over 20 strongly encouraged enhancements is available in RFC 7414. Among this list is RFC 2581, TCP Congestion Control, one of the most important TCP-related RFCs in recent years, describes updated algorithms that avoid undue congestion. In 2001, RFC 3168 was written to describe Explicit Congestion Notification (ECN), a congestion avoidance signaling mechanism.

The original TCP congestion avoidance algorithm was known as "TCP Tahoe", but many alternative algorithms have since been proposed (including TCP Reno, TCP Vegas, FAST TCP, TCP New Reno, and TCP Hybla).

TCP Interactive (iTCP) [29] is a research effort into TCP extensions that allows applications to subscribe to TCP events and register handler components that can launch applications for various purposes, including application-assisted congestion control.

Multipath TCP (MPTCP) [30][31] is an ongoing effort within the IETF that aims at allowing a TCP connection to use multiple paths to maximize resource usage and increase redundancy. The redundancy offered by Multipath TCP in the context of wireless networks [32] enables statistical multiplexing of resources, and thus increases TCP throughput dramatically. Multipath TCP also brings performance benefits in datacenter environments.[33] The reference implementation[34] of Multipath TCP is being developed in the Linux kernel.[35][36]

TCP Cookie Transactions (TCPCT) is an extension proposed in December 2009 to secure servers against denial-of-service attacks. Unlike SYN cookies, TCPCT does not conflict with other TCP extensions such as window scaling. TCPCT was designed due to necessities of DNSSEC, where servers have to handle large numbers of short-lived TCP connections.

tcpcrypt is an extension proposed in July 2010 to provide transport-level encryption directly in TCP itself. It is designed to work transparently and not require any configuration. Unlike TLS (SSL), tcpcrypt itself does not provide authentication, but provides simple primitives down to the application to do that. As of 2010, the first tcpcrypt IETF draft has been published and implementations exist for several major platforms.

TCP Fast Open is an extension to speed up the opening of successive TCP connections between two endpoints. It works by skipping the three-way handshake using a cryptographic "cookie". It is similar to an earlier proposal called T/TCP, which was not widely adopted due to security issues.[37] As of July 2012, it is an IETF Internet draft.[38]

Proposed in May 2013, Proportional Rate Reduction (PRR) is a TCP extension developed by Google engineers. PRR ensures that the TCP window size after recovery is as close to the Slow-start threshold as possible.[39] The algorithm is designed to improve the speed of recovery and is the default congestion control algorithm in Linux 3.2+ kernels.[40]

## TCP over wireless networks

TCP was originally designed for wired networks. Packet loss is considered to be the result of network congestion and the congestion window size is reduced dramatically as a precaution. However, wireless links are known to experience sporadic and usually temporary losses due to fading, shadowing, hand off, interference, and other radio effects, that are not strictly congestion. After the (erroneous) back-off of the congestion window size, due to wireless packet loss, there may be a congestion avoidance phase with a conservative decrease in window size. This causes the radio link to be underutilized. Extensive research on combating these harmful effects has been conducted. Suggested solutions can be categorized as end-to-end solutions, which require modifications at the client or server,[41] link layer solutions, such as Radio Link Protocol (RLP) in cellular networks, or proxy-based solutions which require some changes in the network without modifying end nodes.[41][42]

A number of alternative congestion control algorithms, such as Vegas, Westwood, Veno, and Santa Cruz, have been proposed to help solve the wireless problem.

## Hardware implementations

One way to overcome the processing power requirements of TCP is to build hardware implementations of it, widely known as TCP offload engines (TOE). The main problem of TOEs is that they are hard to integrate into computing systems, requiring extensive changes in the operating system of the computer or device. One company to develop such a device was Alacritech.

## Debugging

A packet sniffer, which intercepts TCP traffic on a network link, can be useful in debugging networks, network stacks, and applications that use TCP by showing the user what packets are passing through a link. Some networking stacks support the SO_DEBUG socket option, which can be enabled on the socket using setsockopt. That option dumps all the packets, TCP states, and events on that socket, which is helpful in debugging. Netstat is another utility that can be used for debugging.

## Alternatives

For many applications TCP is not appropriate. One problem (at least with normal implementations) is that the application cannot access the packets coming after a lost packet until the retransmitted copy of the lost packet is received. This causes problems for real-time applications such as streaming media, real-time multiplayer games and voice over IP (VoIP) where it is generally more useful to get most of the data in a timely fashion than it is to get all of the data in order.

For historical and performance reasons, most storage area networks (SANs) use Fibre Channel Protocol (FCP) over Fibre Channel connections.

Also, for embedded systems, network booting, and servers that serve simple requests from huge numbers of clients (e.g. DNS servers) the complexity of TCP can be a problem. Finally, some tricks such as transmitting data between two hosts that are both behind NAT (using STUN or similar systems) are far simpler without a relatively complex protocol like TCP in the way.

Generally, where TCP is unsuitable, the User Datagram Protocol (UDP) is used. This provides the application multiplexing and checksums that TCP does, but does not handle streams or retransmission, giving the application developer the ability to code them in a way suitable for the situation, or to replace them with other methods like forward error correction or interpolation.

Stream Control Transmission Protocol (SCTP) is another protocol that provides reliable stream oriented services similar to TCP. It is newer and considerably more complex than TCP, and has not yet seen widespread deployment. However, it is especially designed to be used in situations where reliability and near-real-time considerations are important.

Venturi Transport Protocol (VTP) is a patented proprietary protocol that is designed to replace TCP transparently to overcome perceived inefficiencies related to wireless data transport.

TCP also has issues in high-bandwidth environments. The TCP congestion avoidance algorithm works very well for ad-hoc environments where the data sender is not known in advance. If the environment is predictable, a timing based protocol such as Asynchronous Transfer Mode (ATM) can avoid TCP's retransmits overhead.

UDP-based Data Transfer Protocol (UDT) has better efficiency and fairness than TCP in networks that have high bandwidth-delay product.[43]

Multipurpose Transaction Protocol (MTP/IP) is patented proprietary software that is designed to adaptively achieve high throughput and transaction performance in a wide variety of network conditions, particularly those where TCP is perceived to be inefficient.

## Checksum computation

### TCP checksum for IPv4

When TCP runs over IPv4, the method used to compute the checksum is defined in RFC 793:

> *The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16-bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.*

In other words, after appropriate padding, all 16-bit words are added using one's complement arithmetic. The sum is then bitwise complemented and inserted as the checksum field. A pseudo-header that mimics the IPv4 packet header used in the checksum computation is shown in the table below.

**TCP pseudo-header for checksum computation (IPv4)**

| Bit offset | 0–3 | 4–7 | 8–15 | 16–31 |
|---|---|---|---|---|
| **0** | Source address | | | |
| **32** | Destination address | | | |
| **64** | Zeros | | Protocol | TCP length |
| **96** | Source port | | | Destination port |
| **128** | Sequence number | | | |
| **160** | Acknowledgement number | | | |
| **192** | Data offset | Reserved | Flags | Window |
| **224** | Checksum | | | Urgent pointer |
| **256** | Options (optional) | | | |
| **256/288+** | Data | | | |

The source and destination addresses are those of the IPv4 header. The protocol value is 6 for TCP (cf. List of IP protocol numbers). The TCP length field is the length of the TCP header and data (measured in octets).

### TCP checksum for IPv6

When TCP runs over IPv6, the method used to compute the checksum is changed, as per RFC 2460:

> *Any transport or other upper-layer protocol that includes the addresses from the IP header in its checksum computation must be modified for use over IPv6, to include the 128-bit IPv6 addresses instead of 32-bit IPv4 addresses.*

A pseudo-header that mimics the IPv6 header for computation of the checksum is shown below.

**TCP pseudo-header for checksum computation (IPv6)**

| Bit offset | 0–7 | 8–15 | 16–23 | 24–31 |
|---|---|---|---|---|
| **0** | Source address | | | |
| **32** | | | | |
| **64** | | | | |
| **96** | | | | |
| **128** | Destination address | | | |
| **160** | | | | |
| **192** | | | | |
| **224** | | | | |
| **256** | TCP length | | | |
| **288** | Zeros | | | Next header |
| **320** | Source port | | Destination port | |
| **352** | Sequence number | | | |
| **384** | Acknowledgement number | | | |
| **416** | Data offset<br>Reserved | Flags | Window | |
| **448** | Checksum | | Urgent pointer | |
| **480** | Options (optional) | | | |
| **480/512+** | Data | | | |

- Source address – the one in the IPv6 header
- Destination address – the final destination; if the IPv6 packet doesn't contain a Routing header, TCP uses the destination address in the IPv6 header, otherwise, at the originating node, it uses the address in the last element of the Routing header, and, at the receiving node, it uses the destination address in the IPv6 header.
- TCP length – the length of the TCP header and data
- Next Header – the protocol value for TCP

### Checksum offload

Many TCP/IP software stack implementations provide options to use hardware assistance to automatically compute the checksum in the network adapter prior to transmission onto the network or upon reception from the network for validation. This may relieve the OS from using precious CPU cycles calculating the checksum. Hence, overall network performance is increased.

This feature may cause packet analyzers detecting outbound network traffic upstream of the network adapter that are unaware or uncertain about the use of checksum offload to report invalid checksum in outbound packets.

## See also

- Connection-oriented communication
- Karn's algorithm
- List of TCP and UDP port numbers (a long list of ports and services)
- Maximum segment lifetime
- Maximum transmission unit
- Micro-bursting (networking)
- Nagle's algorithm
- Port (computer networking)
- T/TCP variant of TCP
- TCP congestion avoidance algorithms
- TCP global synchronization
- TCP pacing
- TCP segment
- TCP sequence prediction attack
- TCP tuning for high performance networks
- WTCP a proxy-based modification of TCP for wireless networks
- Transport Layer § Comparison of transport layer protocols

## References

1. Vinton G. Cerf; Robert E. Kahn (May 1974). "*A Protocol for Packet Network Intercommunication*" (PDF). *IEEE Transactions on Communications*. **22** (5): 637–648. doi:10.1109/tcom.1974.1092259. Archived from the original (PDF) on March 4, 2016.
2. Comer, Douglas E. (2006). *Internetworking with TCP/IP:Principles, Protocols, and Architecture*. **1** (5th ed.). Prentice Hall. ISBN 0-13-187671-6.
3. "TCP (Linktionary term)".
4. "RFC 791 – section 2.1".
5. "RFC 793".
6. "RFC 1323, TCP Extensions for High Performance, Section 2.2".
7. "RFC 2018, TCP Selective Acknowledgement Options, Section 2".
8. "RFC 2018, TCP Selective Acknowledgement Options, Section 3".
9. "RFC 1323, TCP Extensions for High Performance, Section 3.2".
10. RFC 793 section 3.1
11. RFC 793 Section 3.2
12. Tanenbaum, Andrew S. (2003-03-17). *Computer Networks* (Fourth ed.). Prentice Hall. ISBN 0-13-066102-3.
13. "TCP Definition". Retrieved 2011-03-12.
14. Mathis; Mathew; Semke; Mahdavi; Ott (1997). "The macroscopic behavior of the TCP congestion avoidance algorithm". *ACM SIGCOMM Computer Communication Review*. **27.3**: 67–82.
15. Paxson, V.; Allman, M.; Chu, J.; Sargent, M. (June 2011). "The Basic Algorithm" (https://tools.ietf.org/html/rfc6298#section-2). *Computing TCP's Retransmission Timer* (https://tools.ietf.org/html/rfc6298). IETF. p. 2. sec. 2. RFC 6298. https://tools.ietf.org/html/rfc6298#section-2. Retrieved October 24, 2015.
16. Stone; Partridge (2000). "When The CRC and TCP Checksum Disagree". *Sigcomm*.
17. "RFC 879".
18. "TCP window scaling and broken routers [LWN.net]".
19. Gont, Fernando (November 2008). "On the implementation of TCP urgent data". 73rd IETF meeting. Retrieved 2009-01-04.
20. Peterson, Larry (2003). *Computer Networks*. Morgan Kaufmann. p. 401. ISBN 1-55860-832-X.
21. Richard W. Stevens (2006). November 2011 *TCP/IP Illustrated. Vol. 1, The protocols* Check |url= value (help). Addison-Wesley. pp. Chapter 20. ISBN 978-0-201-63346-7.
22. Security Assessment of the Transmission Control Protocol (TCP) (https://web.archive.org/web/20090306052826/http://www.cpni.gov.uk/Docs/tn-03-09-security-assessment-TCP.pdf) at the Wayback Machine (archived March 6, 2009)
23. Security Assessment of the Transmission Control Protocol (TCP) (https://tools.ietf.org/html/draft-ietf-tcpm-tcp-security)
24. Jakob Lell. "Quick Blind TCP Connection Spoofing with SYN Cookies". Retrieved 2014-02-05.
25. Some insights about the recent TCP DoS (Denial of Service) vulnerabilities (http://www.gont.com.ar/talks/hacklu2009/fgont-hacklu2009-tcp-security.pdf)
26. "Exploiting TCP and the Persist Timer Infiniteness".
27. "Laurent Joncheray, *Simple Active Attack Against TCP*, 1995".
28. John T. Hagen; Barry E. Mullins (2013). "TCP veto: A novel network attack and its application to SCADA protocols". *Innovative Smart Grid Technologies (ISGT), 2013 IEEE PES*.

29. TCP Interactive (iTCP) (http://www.medianet.kent.edu/projects_files/projectITCP.html)
30. RFC 6182
31. RFC 6824
32. "TCP with feed-forward source coding for wireless downlink networks".
33. Raiciu; Barre; Pluntke; Greenhalgh; Wischik; Handley (2011). "Improving datacenter performance and robustness with multipath TCP". *Sigcomm*.
34. "MultiPath TCP - Linux Kernel implementation".
35. Barre; Paasch; Bonaventure (2011). "MultiPath TCP: From Theory to Practice". *IFIP Networking*.
36. Raiciu; Paasch; Barre; Ford; Honda; Duchene; Bonaventure; Handley (2012). "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP". *USENIX NSDI*.
37. Michael Kerrisk (2012-08-01). "TCP Fast Open: expediting web services". LWN.net.
38. Y. Cheng, J. Chu, S. Radhakrishnan, A. Jain (2012-07-16). *TCP Fast Open* (https://tools.ietf.org/html/draft-ietf-tcpm-fastopen-01). IETF. I-D draft-ietf-tcpm-fastopen-01. https://tools.ietf.org/html/draft-ietf-tcpm-fastopen-01.
39. "RFC 6937 - Proportional Rate Reduction for TCP". *http://tools.ietf.org/html/rfc6937*. External link in `|website=` (help);
40. Grigorik, Ilya (2013). *High-performance browser networking* (1. ed.). Beijing: O'Reilly. ISBN 1449344763.
41. "TCP performance over CDMA2000 RLP". Retrieved 2010-08-30
42. Muhammad Adeel & Ahmad Ali Iqbal (2004). "TCP Congestion Window Optimization for CDMA2000 Packet Data Networks". *International Conference on Information Technology (ITNG'07)*: 31–35. doi:10.1109/ITNG.2007.190. ISBN 978-0-7695-2776-5.
43. Yunhong Gu, Xinwei Hong, and Robert L. Grossman. "An Analysis of AIMD Algorithm with Decreasing Increases" (http://udt.sourceforge.net/doc/gridnet-v8.pdf). 2004.

# Further reading

- Stevens, W. Richard. *TCP/IP Illustrated, Volume 1: The Protocols*. ISBN 0-201-63346-9.
- Stevens, W. Richard; Wright, Gary R. *TCP/IP Illustrated, Volume 2: The Implementation*. ISBN 0-201-63354-X.
- Stevens, W. Richard. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. ISBN 0-201-63495-3.**

# External links

## RFC

- RFC 675 – Specification of Internet Transmission Control Program, December 1974 Version
- RFC 793 – TCP v4
- RFC 1122 – includes some error corrections for TCP
- RFC 1323 – TCP Extensions for High Performance [Obsoleted by RFC 7323]
- RFC 1379 – Extending TCP for Transactions—Concepts [Obsoleted by RFC 6247]
- RFC 1948 – Defending Against Sequence Number Attacks
- RFC 2018 – TCP Selective Acknowledgment Options
- RFC 5681 – TCP Congestion Control
- RFC 6247 - Moving the Undeployed TCP Extensions RFC 1072, RFC 1106, RFC 1110, RFC 1145, RFC 1146, RFC 1379, RFC 1644, and RFC 1693 to Historic Status
- RFC 6298 – Computing TCP's Retransmission Timer
- RFC 6824 - TCP Extensions for Multipath Operation with Multiple Addresses
- RFC 7323 - TCP Extensions for High Performance
- RFC 7414 – A Roadmap for TCP Specification Documents

Wikiversity has learning materials about ***Transmission Control Protocol***

Wikimedia Commons has media related to ***Transmission Control Protocol***.

## Others

- Oral history interview with Robert E. Kahn (http://purl.umn.edu/107387), Charles Babbage Institute, University of Minnesota, Minneapolis. Focuses on Kahn's role in the development of computer networking from 1967 through the early 1980s. Beginning with his work at Bolt Beranek

and Newman (BBN), Kahn discusses his involvement as the ARPANET proposal was being written, his decision to become active in its implementation, and his role in the public demonstration of the ARPANET. The interview continues into Kahn's involvement with networking when he moves to IPTO in 1972, where he was responsible for the administrative and technical evolution of the ARPANET, including programs in packet radio, the development of a new network protocol (TCP/IP), and the switch to TCP/IP to connect multiple networks.
- IANA Port Assignments (http://www.iana.org/assignments/port-numbers)
- John Kristoff's Overview of TCP (Fundamental concepts behind TCP and how it is used to transport data between two endpoints) (http://condor.depaul.edu/~jkristof/technotes/tcp.html)
- TCP fast retransmit simulation animated: slow start, sliding window, duplicated Ack, congestion window (http://www.visualland.net/tcp_histrory.php?simu=tcp_fast_retransmit&protocol=TCP&title=4.%20Fast%20transmit&ctype=1)
- TCP, Transmission Control Protocol (http://www.networksorcery.com/enp/protocol/tcp.htm)
- Checksum example (http://mathforum.org/library/drmath/view/54379.html)
- Engineer Francesco Buffa's page about Transmission Control Protocol (https://web.archive.org/web/20110312114242/http://www.ilmondodelletelecomunicazioni.it/english/telematics/protocols.html)
- TCP tutorial (http://www.ssfnet.org/Exchange/tcp/tcpTutorialNotes.html)
- Linktionary on TCP segments (http://www.linktionary.com/s/segment_tcp.html)
- TCP Sliding Window simulation animated (ns2) (http://www.visualland.net/tcp_histrory.php?simu=tcp_swnd&protocol=TCP&title=2.Sliding%20Window&ctype=1)
- Multipath TCP (http://www.multipath-tcp.org)
- TCP Technology and Testing methodologies (http://www.exfo.com/library/technical-resources/application-notes-case-studies-white-papers/bu2-tcp-technology-testing-methodologies-white-paper)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Transmission_Control_Protocol&oldid=750054935"

Categories: Transmission Control Protocol | Transport layer protocols