WIKIPEDIA
The Free Encyclopedia
WIKIPEDIA

Transmission Control Protocol

Transmission Control Protocol

Protocol stack	
Abbreviation	TCP
Developer(s)	Vint Cerf and Bob Kahn
Introduction	1974
Based on	Transmission Control Program
OSI layer	Transport layer (4)
RFC(s)	9293 (https://datatracker.ietf.org/doc/html/rfc9293)

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 (bytes) between applications running on hosts communicating via an IP network. Major internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP, which is part of the Transport layer of the TCP/IP suite. SSL/TLS often runs on top of TCP.

TCP is connection-oriented, meaning that sender and receiver firstly need to establish a connection based on agreed parameters; they do this through three-way handshake procedure.[1] The server must be listening (passive open) for connection requests from clients before a connection is established. Three-way handshake (active open), retransmission, and error detection adds to reliability but lengthens latency. Applications that do not require reliable data stream service may use the User Datagram Protocol (UDP) instead, which provides a connectionless datagram service that prioritizes time over reliability. TCP employs network congestion avoidance. However, there are vulnerabilities in TCP, including denial of service, connection hijacking, TCP veto, and reset attack.

Historical origin

In May 1974, Vint Cerf and Bob Kahn described an internetworking protocol for sharing resources using packet switching among network nodes.[2] The authors had been working with Gérard Le Lann to incorporate concepts from the French CYCLADES project into the new network.[3] The specification of the resulting protocol, RFC 675 (*Specification of Internet Transmission Control Program*), was written by Vint Cerf, Yogen Dalal, and Carl Sunshine, and published in December 1974.[4] It contains the first attested use of the term *internet*, as a shorthand for *internetwork*.

The Transmission Control Program incorporated both connection-oriented links and datagram services between hosts. In version 4, the monolithic Transmission Control Program was divided into a modular architecture consisting of the *Transmission Control Protocol* and the *Internet Protocol*.[5][6] This resulted in a networking model that became known informally as *TCP/IP*, although formally it was variously referred to as the *DoD internet architecture model* (*DoD model* for short) or *DARPA model*.[7][8][9] Later, it became the part of, and synonymous with, the *Internet Protocol Suite*.

The following Internet Experiment Note (IEN) documents describe the evolution of TCP into the modern version:[10]

- IEN 5 (http://www.rfc-editor.org/ien/ien5.pdf) *Specification of Internet Transmission Control Program TCP Version 2 (*March 1977).
- IEN 21 (http://www.rfc-editor.org/ien/ien21.pdf) *Specification of Internetwork Transmission Control Program TCP Version 3 (*January 1978).
- IEN 27
- IEN 40
- IEN 44
- IEN 55
- IEN 81
- IEN 112
- IEN 124

TCP was standardized in January 1980 as RFC 761 (https://datatracker.ietf.org/doc/html/rfc761).

In 2004, Vint Cerf and Bob Kahn received the Turing Award for their foundational work on TCP/IP.[11][12]

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 IP fragmentation to accommodate the maximum transmission unit of the transmission medium. At the transport layer, TCP handles all handshaking and transmission details and presents an abstraction of the network connection to the application typically through a network socket interface.

At the lower levels of the protocol stack, due to network congestion, traffic load balancing, or unpredictable network behavior, 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 minimize network congestion to reduce the occurrence of the other problems. If the data still remains undelivered, the 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 internet applications, including the World Wide Web (WWW), email, File Transfer Protocol, Secure Shell, peer-to-peer file sharing, and streaming media.

TCP is optimized for accurate delivery rather than timely delivery and can incur relatively long delays (on the order of seconds) while waiting for out-of-order messages or re-transmissions of lost messages. Therefore, 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.[13]

TCP is a reliable byte stream delivery service that guarantees that all bytes received will be identical and in the same order as those sent. Since packet transfer by many networks is not reliable, TCP achieves this using a technique known as *positive acknowledgment with re-transmission*. This requires the receiver to respond with an acknowledgment 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 receiving the acknowledgment. The timer is needed in case a packet gets lost or corrupted.[13]

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 file into segments and forwards them individually to the internet layer in the network stack. The internet layer software 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 software in the transport layer re-assembles the segments and ensures they are correctly ordered and error-free as it streams the file contents to the receiving 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.[14]

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*[15] 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.[16]

A TCP segment consists of a segment *header* and a *data* section. The segment header contains 10 mandatory fields, and an optional extension field (*Options*, pink background in table). The data section follows the header and is the payload data carried for the application.[17] The length of the data section is not specified in the segment header; it can be calculated by subtracting the combined length of the segment header and IP header from the total IP datagram length specified in the IP header.

TCP header format[17]

Offset	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
0	0	*Source Port*																*Destination Port*													
4	32	*Sequence Number*																													
8	64	*Acknowledgement Number (meaningful when ACK bit set)*																													
12	96	*Data Offset*				*Reserved*				*CWR*	*ECE*	*URG*	*ACK*	*PSH*	*RST*	*SYN*	*FIN*	*Window*													
16	128	*Checksum*																*Urgent Pointer (meaningful when URG bit set)*[18]													
20	160	*(Options) If present, Data Offset will be greater than 5.* *Padded with zeroes to a multiple of 32 bits, since Data Offset counts words of 4 octets.*																													
⋮	⋮																														
56	448																														
60	480	*Data*																													
64	512																														
⋮	⋮																														

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 unset (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 of the ACK is expecting. This acknowledges receipt of all prior bytes (if any).[19] The first ACK sent by each end acknowledges the other end's initial sequence number itself, but no data.[20]
Data Offset (DOffset): 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 (Rsrvd): 4 bits
For future use and should be set to zero; senders should not set these and receivers should ignore them if set, in the absence of further specification and implementation.
From 2003 to 2017, the last bit (bit 103 of the header) was defined as the NS (Nonce Sum) flag by the experimental RFC 3540, ECN-nonce. ECN-nonce never gained widespread use and the RFC was moved to Historic status.[21]
Flags: 8 bits
Contains 8 1-bit flags (control bits) as follows:

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.[22][a]

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), the TCP peer is ECN capable.[23]
- If the SYN flag is unset (0), a packet with the Congestion Experienced flag set (ECN=11) in its IP header was received during normal transmission.[a] This serves as an indication of network congestion (or impending congestion) to the TCP sender.[24]

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.[25]

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 when it is set, and others when it is clear.

FIN: 1 bit
Last packet from sender

Window: 16 bits
The size of the *receive window*, which specifies the number of window size units[b] that the sender of this segment is currently willing to receive.[c] (See § Flow control and § Window scaling.)

Checksum: 16 bits
The 16-bit checksum field is used for error-checking of the TCP header, the payload and an IP pseudo-header. The pseudo-header consists of the source IP address, the destination IP address, the protocol number for the TCP protocol (6) and the length of the TCP headers and payload (in bytes).

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 (TCP Option): Variable 0–320 bits, in units of 32 bits; size(Options) == (DOffset - 5) * 32
The length of this field is determined by the *Data Offset* field. 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.[16]
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 Option-Kind value, the next two fields may be set. Option-Length indicates the total length of the option, and Option-Data contains data associated with the option, if applicable. For example, an Option-Kind byte of 1 indicates that this is a no operation option used only for padding, and does not have an Option-Length or Option-Data fields following it. An Option-Kind byte of 0 marks the end of options, and is also only one byte. An Option-Kind byte of 2 is used to indicate Maximum Segment Size option, and will be followed by an Option-Length byte specifying the length of the MSS field. Option-Length is the total length of the given options field, including Option-Kind and Option-Length fields. So while the MSS value is typically expressed in two bytes, Option-Length will be 4. As an example, an MSS option field with a value of `0x05B4` is coded 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).

Option-Kind	Option-Length	Option-Data	Purpose	Notes
0	—	—	End of options list	
1	—	—	No operation	This may be used to align option fields on 32-bit boundaries for better performance.
2	4	SS	Maximum segment size	See § Maximum segment size for details. `[SYN]`
3	3	S	Window scale	See § Window scaling for details.[26] `[SYN]`
4	2	—	Selective Acknowledgement permitted	See § Selective acknowledgments for details.[27] `[SYN]`
5	N (10, 18, 26, or 34)	BBBB, EEEE, ...	Selective ACKnowledgement (SACK)[28]	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	Timestamp and echo of previous timestamp	See § TCP timestamps for details.[26]
28	4	—	User Timeout Option	See RFC 5482 (https://datatracker.ietf.org/doc/html/rfc5482).
29	N	—	TCP Authentication Option (TCP-AO)	For message authentication, replacing MD5 authentication (option 19) originally designed to protect BGP sessions.[29] See RFC 5925 (https://datatracker.ietf.org/doc/html/rfc5925).
30	N	—	Multipath TCP (MPTCP)	See Multipath TCP for details.

The remaining Option-Kind values are historical, obsolete, experimental, not yet standardized, or unassigned. Option number assignments are maintained by the Internet Assigned Numbers Authority (IANA).[30]

Data: Variable
The payload of the TCP packet

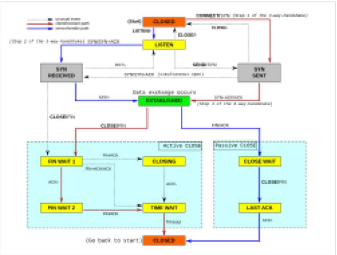
Protocol operation

TCP protocol operations may be divided into three phases. *Connection establishment* is a multi-step handshake process that establishes a connection before entering the *data transfer* phase. After data transfer is completed, the *connection termination* closes the connection and releases all allocated resources.

A TCP connection is managed by an operating system through a resource 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:[31]

TCP socket states

State	Endpoint	Description
LISTEN	Server	Waiting for a connection request from any remote TCP end-point.
SYN-SENT	Client	Waiting for a matching connection request after having sent a connection request.
SYN-RECEIVED	Server	Waiting for a confirming connection request acknowledgment after having both received and sent a connection request.
ESTABLISHED	Server and client	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	Server and client	Waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.
FIN-WAIT-2	Server and client	Waiting for a connection termination request from the remote TCP.
CLOSE-WAIT	Server and client	Waiting for a connection termination request from the local user.
CLOSING	Server and client	Waiting for a connection termination request acknowledgment from the remote TCP.
LAST-ACK	Server and client	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	Server or client	Waiting for enough time to pass to be sure that all remaining packets on the connection have expired.
CLOSED	Server and client	No connection state at all.



A Simplified TCP State Diagram.

Connection establishment

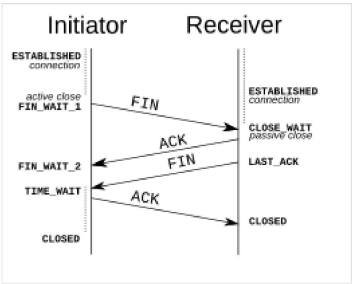
Before a client attempts to connect with a server, the server must first bind to and listen at a port to open it up for connections: this is called a passive open. Once the passive open is established, a client may establish a connection by initiating an active open using the three-way (or 3-step) handshake:

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 acknowledgment value i.e. A+1, and the acknowledgment number is set to one more than the received sequence number i.e. B+1.

Steps 1 and 2 establish and acknowledge the sequence number for one direction (client to server). Steps 2 and 3 establish and acknowledge the sequence number for the other direction (server to client). Following the completion of these steps, both the client and server have received acknowledgments and 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 state lets the TCP client resend the final acknowledgment to the server in case the ACK is lost in transit. The time duration is implementation-dependent, but some common values are 30 seconds, 1 minute, and 2 minutes. After the timeout, the client enters the CLOSED state and the local port becomes available for new connections.[32]



Connection termination

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 (combining two steps into one) and host A replies with an ACK.[33]

Some operating systems, such as Linux and HP-UX, implement a half-duplex close sequence. If the host actively closes a connection, while still having unread incoming data available, the host sends the signal RST (losing any received data) instead of FIN. This assures that a TCP application is aware there was a data loss.[34]

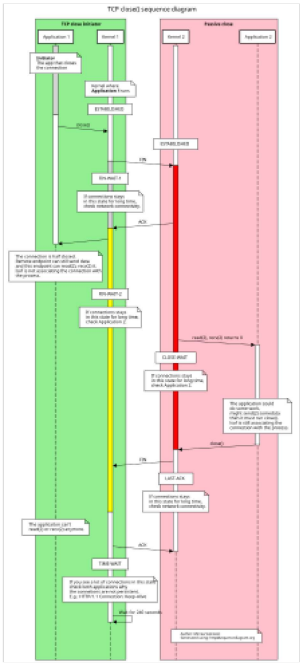
A connection can be in a half-open state, in which case one side has terminated the connection, but the other has not. The side that has terminated can no longer send any data into the connection, but the other side can. The terminating side should continue reading the data until the other side terminates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

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 an ephemeral 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.


Detailed TCP close() sequence diagram

Data transfer

The Transmission Control Protocol differs in several key features compared to the User Datagram Protocol:

- Ordered data transfer: the destination host rearranges segments according to a sequence number[13]
- Retransmission of lost packets: any cumulative stream not acknowledged is retransmitted[13]
- Error-free data transfer: corrupted packets are treated as lost and are retransmitted[14]
- 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. When the receiving host's buffer fills, the next acknowledgment suspends the transfer and allows the data in the buffer to be processed.[13]
- Congestion control: lost packets (presumed due to congestion) trigger a reduction in data delivery rate[13]

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 out-of-order delivery that may occur. The sequence number of the first byte is chosen by the transmitter for the first packet, which is flagged SYN. This number can be arbitrary, and should, in fact, be unpredictable to defend against TCP sequence prediction attacks.

Acknowledgments (ACKs) are sent with a sequence number by the receiver of data to tell the sender that data has been received to the specified byte. 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.

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

When a TCP segment is retransmitted, it retains the same sequence number as the original delivery attempt. This conflation of delivery and logical data ordering means that, when acknowledgment is received after a retransmission, the sender cannot tell whether the original transmission or the retransmission is being acknowledged, the so-called *retransmission ambiguity*.[35] TCP incurs complexity due to retransmission ambiguity.[36]

Dupack-based retransmission

If a single segment (say segment number 100) in a stream is lost, then the receiver cannot acknowledge packets above that segment number (100) because it uses cumulative ACKs. Hence the receiver acknowledges packet 99 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 acknowledgments, it retransmits the last unacknowledged packet. A threshold of three is used because the network may reorder segments causing duplicate acknowledgements. This threshold has been demonstrated to avoid spurious retransmissions due to reordering.[37] Some TCP implementations use selective acknowledgements (SACKs) to provide explicit feedback about the segments that have been received. This greatly improves TCP's ability to retransmit the right segments.

Retransmission ambiguity can cause spurious fast retransmissions and congestion avoidance if there is reordering beyond the duplicate acknowledgment threshold.[38] In the last two decades more packet reordering has been observed over the Internet [39] which led TCP implementations, such as the one in the Linux Kernel to adopt heuristic methods to scale the duplicate acknowledgment threshold. [40] Recently, there have been efforts to completely phase out dupack based fast-retransmissions and replace them with timer based ones.[41] (Not to be confused with the classic RTO discussed below). The time based loss detection algorithm called Recent Acknowledgment (RACK) [42] has been adopted as the default algorithm in Linux and Windows. [43]

Timeout-based retransmission

When a sender transmits a segment, it initializes a timer with a conservative estimate of the arrival time of the acknowledgment. The segment is retransmitted if the timer expires, with a new timeout threshold of twice the previous value, resulting in exponential backoff behavior. Typically, the initial timer value is $\textbf{smoothed RTT} + \textbf{max}(G, 4 \times \textbf{RTT variation})$, where G is the clock granularity.[44] This guards against excessive transmission traffic due to faulty or malicious actors, such as man-in-the-middle denial of service attackers.

Accurate RTT estimates are important for loss recovery, as it allows a sender to assume an unacknowledged packet to be lost after sufficient time elapses (i.e., determining the RTO time).[45] Retransmission ambiguity can lead a sender's estimate of RTT to be imprecise.[45] In an environment with variable RTTs, spurious timeouts can occur:[46] if the RTT is under-estimated, then the RTO fires and triggers a needless retransmit and slow-start. After a spurious retransmission, when the acknowledgments for the original transmissions arrive, the sender may believe them to be acknowledging the retransmission and conclude, incorrectly, that segments sent between the original transmission and retransmission have been lost, causing further needless retransmissions to the extent that the link truly becomes congested;[47][48] selective acknowledgement can reduce this effect.[49] RFC 6298 specifies that implementations must not use retransmitted segments when estimating RTT.[50] Karn's algorithm ensures that a good RTT estimate will be produced—eventually—by waiting until there is an unambiguous acknowledgment before adjusting the RTO.[51] After spurious retransmissions, however, it may take significant time before such an unambiguous acknowledgment arrives, degrading performance in the interim.[52] TCP timestamps also resolve the retransmission ambiguity problem in setting the RTO,[50] though they do not necessarily improve the RTT estimate.[53]

Error detection

Sequence numbers allow receivers to discard duplicate packets and properly sequence out-of-order packets. Acknowledgments allow senders to determine when to retransmit lost packets.

To assure correctness a checksum field is included; see § Checksum computation for details. The TCP checksum is a weak check by modern standards and is normally paired with a CRC integrity check at layer 2, below both TCP and IP, such as is used in PPP or the Ethernet frame. However, introduction of errors in packets between CRC-protected hops is common and the 16-bit TCP checksum catches most of these.[54]

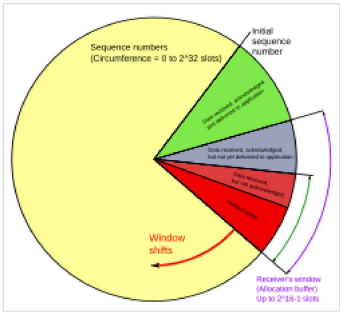
Flow control

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 be able to regulate the data flow so as not to be overwhelmed.[13]

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 receive window update from the receiving host.

When a receiver advertises a window size of 0, the sender stops sending data and starts its *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 acknowledgment containing the new window size.

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.



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.

Congestion control

The final main aspect of TCP is congestion control. TCP uses a number of mechanisms to achieve high performance and avoid congestive collapse, a gridlock situation where network performance is severely degraded. 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 the 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 or congestion avoidance.

Modern implementations of TCP contain four intertwined algorithms: slow start, congestion avoidance, fast retransmit, and fast recovery.[55]

In addition, senders employ a *retransmission timeout* (RTO) that is based on the estimated round-trip time (RTT) between the sender and receiver, as well as the variance in this round-trip time.[56] 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.[26] 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 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 accomplish this, typically the MSS is announced by each side using the MSS option when the TCP connection is established. The option value 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. 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 may also be called *MSS negotiation* but, strictly speaking, the MSS is not *negotiated*. Two completely independent values of MSS are permitted for the two directions of data flow in a TCP connection,[57][16] so there is no need to agree on a common MSS configuration for a bidirectional connection.

Selective acknowledgments

Relying purely on the cumulative acknowledgment scheme employed by the original TCP can lead to inefficiencies when packets are lost. For example, suppose bytes with sequence number 1,000 to 10,999 are sent in 10 different TCP segments of equal size, and the second segment (sequence numbers 2,000 to 2,999) is lost during transmission. In a pure cumulative acknowledgment protocol, the receiver can only send a cumulative ACK value of 2,000 (the sequence number immediately following the last sequence number of the received data) and cannot say that it received bytes 3,000 to 10,999 successfully. Thus the sender may then have to resend all data starting with sequence number 2,000.

To alleviate this issue TCP employs the *selective acknowledgment (SACK)* option, defined in 1996 in RFC 2018, which allows the receiver to acknowledge discontinuous blocks of packets that were received correctly, in addition to the sequence number immediately following the last sequence number of the last contiguous byte received successively, as in the basic TCP acknowledgment. The acknowledgment can include a number of *SACK blocks*, where each SACK block is conveyed by the *Left Edge of Block* (the first sequence number of the block) and the *Right Edge of Block* (the sequence number immediately following the last sequence number of the block), with a *Block* being a contiguous range that the receiver correctly received. In the example above, the receiver would send an ACK segment with a cumulative ACK value of 2,000 and a SACK option header with sequence numbers 3,000 and 11,000. The sender would accordingly retransmit only the second segment with sequence numbers 2,000 to 2,999.

A TCP sender may interpret an out-of-order segment delivery as a lost segment. If it does so, the TCP sender will retransmit the segment 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 May 2000 in RFC 2883, solves this problem. Once the TCP receiver detects a second duplicate packet, it sends a D-ACK to indicate that no segments were lost, allowing the TCP sender to 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 a TCP header option (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).

Selective acknowledgements can be 'reneged', where the receiver unilaterally discards the selectively acknowledged data. RFC 2018 discouraged such behavior, but did not prohibit it to allow receivers the option of reneging if they, for example, ran out of buffer space.[58] The possibility of reneging leads to implementation complexity for both senders and receivers, and also imposes memory costs on the sender.[59]

Window scaling

For more efficient use of high-bandwidth networks, a larger TCP window size may be used. A 16-bit TCP window size field controls the flow of data and its value is limited to 65,535 bytes. Since the size field cannot be expanded beyond this limit, 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 to 1 gigabyte. Scaling up to these larger window sizes 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 when interpreting it. 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.[60]

TCP timestamps

TCP timestamps, defined in RFC 1323 in 1992, 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*. 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 uses TCP timestamps to determine if retransmissions are occurring because packets are lost or simply out of order.[61]

TCP timestamps are enabled by default in Linux,[62] and disabled by default in Windows Server 2008, 2012 and 2016.[63]

Recent Statistics show that the level of TCP timestamp adoption has stagnated, at ~40%, owing to Windows Server dropping support since Windows Server 2008.[64]

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 marks the transmission as out-of-band data (OOB) and tells the receiving program to process it immediately. When finished, TCP informs the application and resumes the stream queue. An example is when TCP is used for a remote login session where the user can send a keyboard sequence that interrupts or aborts the remotely running program without waiting for the program to finish its current transfer.[13]

The *urgent* pointer only alters the processing on the remote host and doesn't expedite any processing on the network itself. The capability is implemented differently or poorly on different systems or may not be supported. Where it is available, it is prudent to assume only single bytes of OOB data will be reliably handled.[65][66] Since the feature is not frequently used, it is not well tested on some platforms and has been associated with vulnerabilities, WinNuke for instance.

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".[13] There is no way to indicate or control it in user space using Berkeley sockets; it is controlled by the protocol stack only.[67]

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,[68] and was pursued within the IETF through 2012.[69] Notable vulnerabilities include denial of service, connection hijacking, TCP veto and TCP reset attack.

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.[70] Sockstress is a similar attack, that might be mitigated with system resource management.[71] An advanced DoS attack involving the exploitation of the TCP *persist timer* was analyzed in Phrack No. 66.[72] PUSH and ACK floods are other variants.[73]

Connection hijacking

An attacker who is able to eavesdrop on 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. A simple hijack can result in one packet being erroneously accepted at one end. When the receiving host acknowledges the false segment, synchronization is lost.[74] Hijacking may be combined with ARP spoofing or other routing attacks that allow an attacker to take permanent control of the TCP connection.

Impersonating a different IP address was not difficult prior to RFC 1948 when the initial *sequence number* was easily guessable. The earlier implementations allowed an attacker to blindly send a sequence of packets that the receiver would believe came from a different IP address, without the need to intercept communication through 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 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.[75]

TCP ports

A TCP connection is identified by a four-tuple of the source address, source port, destination address, and destination port.[d][76][77] Port numbers are used to identify different services, and to allow multiple connections between hosts.[14] TCP uses 16-bit port numbers, providing 65,536 possible values for each of the source and destination ports.[17] The dependency of connection identity on addresses means that TCP connections are bound to a single network path; TCP cannot use other routes that multihomed hosts have available, and connections break if an endpoint's address changes.[78]

Port numbers are categorized into three basic categories: well-known, registered, and dynamic or private. The well-known ports are assigned by the Internet Assigned Numbers Authority (IANA) and are typically used by system-level 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).[e] 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 or private ports can also be used by end-user applications, however, these ports typically do not contain any meaning outside a particular TCP connection.

Network Address Translation (NAT), typically uses dynamic port numbers, on the public-facing 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, published in October 1989, 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).

Multipath TCP (MPTCP)[79][80] 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 enables the simultaneous use of different networks, which brings higher throughput and better handover capabilities. Multipath TCP also brings performance benefits in datacenter

environments.[81] The reference implementation[82] of Multipath TCP was developed in the Linux kernel.[83] Multipath TCP is used to support the Siri voice recognition application on iPhones, iPads and Macs.[84]

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. The tcpcrypt RFC was published by the IETF in May 2019.[85]

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.[86] TCP Fast Open was published as RFC 7413 in 2014.[87]

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.[88] The algorithm is designed to improve the speed of recovery and is the default congestion control algorithm in Linux 3.2+ kernels.[89]

Deprecated proposals

TCP Cookie Transactions (TCPCT) is an extension proposed in December 2009[90] 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. In 2016, TCPCT was deprecated in favor of TCP Fast Open. The status of the original RFC was changed to *historic*.[91]

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.

Wire image and ossification

The wire data of TCP provides significant information-gathering and modification opportunities to on-path observers, as the protocol metadata is transmitted in cleartext.[92][93] While this transparency is useful to network operators[94] and researchers,[95] information gathered from protocol metadata may reduce the end-user's privacy.[96] This visibility and malleability of metadata has led to TCP being difficult to extend—a case of protocol ossification—as any intermediate node (a 'middlebox') can make decisions based on that metadata or even modify it,[97][98] breaking the end-to-end principle.[99] One measurement found that a third of paths across the Internet encounter at least one intermediary that modifies TCP metadata, and 6.5% of paths encounter harmful ossifying effects from intermediaries.[100] Avoiding extensibility hazards from intermediaries placed significant constraints on the design of MPTCP,[101][102] and difficulties caused by intermediaries have hindered the deployment of TCP Fast Open in web browsers.[103] Another source of ossification is the difficulty of modification of TCP functions at the endpoints, typically in the operating system kernel[104] or in hardware with a TCP offload engine.[105]

Performance

As TCP provides applications with the abstraction of a reliable byte stream, it can suffer from head-of-line blocking: if packets are reordered or lost and need to be retransmitted (and thus are reordered), data from sequentially later parts of the stream may be received before sequentially earlier parts of the stream; however, the later data cannot typically be used until the earlier data has been received, incurring network latency. If multiple independent higher-level messages are encapsulated and multiplexed onto a single TCP connection, then head-of-line blocking can cause processing of a fully-received message that was sent later to wait for delivery of a message that was sent earlier.[106] Web browsers attempt to mitigate head-of-line blocking by opening multiple parallel connections. This incurs the cost of connection establishment repeatedly, as well as multiplying the resources needed to track those connections at the endpoints.[107] Parallel connections also have congestion control operating independently of each other, rather than being able to pool information together and respond more promptly to observed network conditions;[108] TCP's aggressive initial sending patterns can cause congestion if multiple parallel connections are opened; and the per-connection fairness model leads to a monopolization of resources by applications that take this approach.[109]

Connection establishment is a major contributor to latency as experienced by web users.[110][111] TCP's three-way handshake introduces one RTT of latency during connection establishment before data can be sent.[111] For short flows, these delays are very significant.[112] Transport Layer Security (TLS) requires a handshake of its own for key exchange at connection establishment. Because of the layered design, the TCP handshake and the TLS handshake proceed serially; the TLS handshake cannot begin until the TCP handshake has concluded.[113] Two RTTs are required for connection establishment with TLS 1.2 over TCP.[114] TLS 1.3 allows for zero RTT connection resumption in some circumstances, but, when layered over TCP, one RTT is still required for the TCP handshake, and this cannot assist the initial connection; zero RTT handshakes also present cryptographic challenges, as efficient, replay-safe and forward secure non-interactive key exchange is an open research topic.[115] TCP Fast Open allows the transmission of data in the initial (i.e., SYN and SYN-ACK) packets, removing one RTT of latency during connection establishment.[116] However, TCP Fast Open has been difficult to deploy due to protocol ossification; as of 2020, no Web browsers used it by default.[103]

TCP throughput is affected by packet reordering. Reordered packets can cause duplicate acknowledgments to be sent, which, if they cross a threshold, will then trigger a spurious retransmission and congestion control. Transmission behavior can also become bursty, as large ranges are acknowledged all at once when a reordered packet at the range's start is received (in a manner similar to how head-of-line blocking affects applications).[117] Blanton & Allman (2002) found that throughput was inversely related to the amount of reordering, up to a threshold where all reordering triggers spurious retransmission.[118] Mitigating reordering depends on a sender's ability to determine that it has sent a spurious retransmission, and hence on resolving retransmission ambiguity.[119] Reducing reordering-induced spurious retransmissions may slow recovery from genuine loss.[120]

Selective acknowledgment can provide a significant benefit to throughput; Bruyeron, Hemon & Zhang (1998) measured gains of up to 45%.[121] An important factor in the improvement is that selective acknowledgment can more often avoid going into slow start after a loss and can hence better use available bandwidth.[122] However, TCP can only selectively acknowledge a maximum of three blocks of sequence numbers. This can limit the retransmission rate and hence loss recovery or cause needless retransmissions, especially in high-loss environments.[123][124]

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 underused. 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,[125] 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.[125][126]

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

Acceleration

The idea of a TCP accelerator is to terminate TCP connections inside the network processor and then relay the data to a second connection toward the end system. The data packets that originate from the sender are buffered at the accelerator node, which is responsible for performing local retransmissions in the event of packet loss. Thus, in case of losses, the feedback loop between the sender and the receiver is shortened to the one between the acceleration node and the receiver which guarantees a faster delivery of data to the receiver.[127]

Since TCP is a rate-adaptive protocol, the rate at which the TCP sender injects packets into the network is directly proportional to the prevailing load condition within the network as well as the processing capacity of the receiver. The prevalent conditions within the network are judged by the sender on the basis of the acknowledgments received by it. The acceleration node splits the feedback loop between the sender and the receiver and thus guarantees a shorter round trip time (RTT) per packet. A shorter RTT is beneficial as it ensures a quicker response time to any changes in the network and a faster adaptation by the sender to combat these changes.

Disadvantages of the method include the fact that the TCP session has to be directed through the accelerator; this means that if routing changes, so that the accelerator is no longer in the path, the connection will be broken. It also destroys the end-to-end property of the TCP ack mechanism; when the ACK is received by the sender, the packet has been stored by the accelerator, not delivered to the receiver.

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.[128]

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 as follows:[16]

> *The checksum field is the 16-bit ones' complement of the ones' complement sum of all 16-bit words in the header and text. The checksum computation needs to ensure the 16-bit alignment of the data being summed. If a segment contains an odd number of header and text octets, alignment can be achieved by padding the last octet with zeros on its right 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 ones' 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)

Offset	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		
0	0	*Source address*																															
4	32	*Destination address*																															
8	64	*Zeroes*								*Protocol (6)*								*TCP length*															
12	96	*Source port*																*Destination port*															
16	128	*Sequence number*																															
20	160	*Acknowledgement number*																															
24	192	*Data offset*				*Reserved*				*Flags*								*Window*															
28	224	*Checksum*																*Urgent pointer*															
32	256	*(Options)*																															
36	288	*Data*																															
40	320																																
⋮	⋮																																

The checksum is computed over the following fields:

Source address: 32 bits
The source address in the IPv4 header
Destination address: 32 bits
The destination address in the IPv4 header.
Zeroes: 8 bits; Zeroes == 0
All zeroes.
Protocol: 8 bits
The protocol value for TCP: 6.
TCP length: 16 bits
The length of the TCP header and data (measured in octets). For example, let's say we have IPv4 packet with Total Length of 200 bytes and IHL value of 5, which indicates a length of 5 × 32 bits = 160 bits = 20 bytes. We can compute the TCP length as **Total Length − IPv4 Header Length**, i.e. **200 − 20**, which results in **180** bytes.

TCP checksum for IPv6

When TCP runs over IPv6, the method used to compute the checksum is changed:[129]

> *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)

Offset	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	
0	0	*Source address*																														
4	32																															
8	64																															
12	96																															
16	128	*Destination address*																														
20	160																															
24	192																															
28	224																															
32	256	*TCP length*																														
36	288	*Zeroes*																								*Next header (6)*						
40	320	*Source port*																*Destination port*														
44	352	*Sequence number*																														
48	384	*Acknowledgement number*																														
52	416	*Data offset*				*Reserved*				*Flags*								*Window*														
56	448	*Checksum*																*Urgent pointer*														
60	480	*(Options)*																														
64	512	*Data*																														
68	544																															
⋮	⋮																															

The checksum is computed over the following fields:

Source address: 128 bits
The address in the IPv6 header.
Destination address: 128 bits
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: 32 bits
The length of the TCP header and data (measured in octets).
Zeroes: 24 bits; Zeroes == 0
All zeroes.
Next header: 8 bits
The protocol value for TCP: 6.

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 that are unaware or uncertain about the use of checksum offload to report invalid checksums in outbound packets that have not yet reached the network adapter.[130] This will only occur for packets that are intercepted before being transmitted by the network adapter; all packets transmitted by the network adaptor on the wire will have valid checksums.[131] This issue can also occur when monitoring packets being transmitted between virtual machines on the same host, where a virtual device driver may omit the checksum calculation (as an optimization), knowing that the checksum will be calculated later by the VM host kernel or its physical hardware.

See also

- Fault-tolerant messaging
- Micro-bursting (networking)
- TCP global synchronization
- TCP fusion
- TCP pacing
- TCP Stealth
- Transport layer § Comparison of transport layer protocols
- WTCP a proxy-based modification of TCP for wireless networks

Notes

a. Added to header by RFC 3168
b. Windows size units are, by default, bytes.
c. Window size is relative to the segment identified by the sequence number in the acknowledgment field.
d. Equivalently, a pair of network sockets for the source and destination, each of which is made up of an address and a port

e. As of the latest standard, HTTP/3, QUIC is used as a transport instead of TCP.

References

1. Labrador, Miguel A.; Perez, Alfredo J.; Wightman, Pedro M. (2010). *Location-Based Information Systems Developing Real-Time Tracking Applications*. CRC Press. ISBN 9781000556803.
2. Vinton G. Cerf; Robert E. Kahn (May 1974). "*A Protocol for Packet Network Intercommunication*" (https://web.archive.org/web/20160304150203/http://ece.ut.ac.ir/Classpages/F84/PrincipleofNetworkDesign/Papers/CK74.pdf) (PDF). *IEEE Transactions on Communications*. **22** (5): 637–648. doi:10.1109/tcom.1974.1092259 (https://doi.org/10.1109%2Ftcom.1974.1092259). Archived from the original (http://ece.ut.ac.ir/Classpages/F84/PrincipleofNetworkDesign/Papers/CK74.pdf) (PDF) on March 4, 2016.
3. Bennett, Richard (September 2009). "Designed for Change: End-to-End Arguments, Internet Innovation, and the Net Neutrality Debate" (https://www.itif.org/files/2009-designed-for-change.pdf) (PDF). Information Technology and Innovation Foundation. p. 11. Archived (https://web.archive.org/web/20190829092926/http://www.itif.org/files/2009-designed-for-change.pdf) (PDF) from the original on 29 August 2019. Retrieved 11 September 2017.
4. RFC 675.
5. Russell, Andrew Lawrence (2008). *'Industrial Legislatures': Consensus Standardization in the Second and Third Industrial Revolutions* (http://jhir.library.jhu.edu/handle/1774.2/32576) (Thesis). "See Abbate, *Inventing the Internet*, 129–30; Vinton G. Cerf (October 1980). "Protocols for Interconnected Packet Networks". *ACM SIGCOMM Computer Communication Review*. **10** (4): 10–11.; and *RFC 760* (https://datatracker.ietf.org/doc/html/rfc760). doi:10.17487/RFC0760 (https://doi.org/10.17487%2FRFC0760)."
6. Postel, Jon (15 August 1977), *Comments on Internet Protocol and TCP* (https://www.rfc-editor.org/ien/ien2.txt), IEN 2, archived (https://web.archive.org/web/20190516055704/http://www.rfc-editor.org/ien/ien2.txt) from the original on May 16, 2019, retrieved June 11, 2016, "We are screwing up in our design of internet protocols by violating the principle of layering. Specifically we are trying to use TCP to do two things: serve as a host level end to end protocol, and to serve as an internet packaging and routing protocol. These two things should be provided in a layered and modular way."
7. Cerf, Vinton G. (1 April 1980). "Final Report of the Stanford University TCP Project" (https://www.rfc-editor.org/ien/ien151.txt).
8. Cerf, Vinton G; Cain, Edward (October 1983). "The DoD internet architecture model". *Computer Networks*. **7** (5): 307–318. doi:10.1016/0376-5075(83)90042-9 (https://doi.org/10.1016%2F0376-5075%2883%2990042-9).
9. "The TCP/IP Guide – TCP/IP Architecture and the TCP/IP Model" (http://www.tcpipguide.com/free/t_TCPIPArchitectureandtheTCPIPModel.htm). *www.tcpipguide.com*. Retrieved 2020-02-11.
10. "Internet Experiment Note Index" (https://www.rfc-editor.org/ien/ien-index.html). *www.rfc-editor.org*. Retrieved 2024-01-21.
11. "Robert E Kahn – A.M. Turing Award Laureate" (https://amturing.acm.org/award_winners/kahn_4598637.cfm). *amturing.acm.org*. Archived (https://web.archive.org/web/20190713004804/https://amturing.acm.org/award_winners/kahn_4598637.cfm) from the original on 2019-07-13. Retrieved 2019-07-13.
12. "Vinton Cerf – A.M. Turing Award Laureate" (https://amturing.acm.org/award_winners/cerf_1083211.cfm). *amturing.acm.org*. Archived (https://web.archive.org/web/20211011080741/https://amturing.acm.org/award_winners/cerf_1083211.cfm) from the original on 2021-10-11. Retrieved 2019-07-13.
13. Comer, Douglas E. (2006). *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. Vol. 1 (5th ed.). Prentice Hall. ISBN 978-0-13-187671-2.
14. RFC 9293, 2.2. Key TCP Concepts.
15. RFC 791, pp. 5–6.
16. RFC 9293.
17. RFC 9293, 3.1. Header Format.
18. RFC 9293, 3.8.5 The Communication of Urgent Information.
19. RFC 9293, 3.4. Sequence Numbers.
20. RFC 9293, 3.4.1. Initial Sequence Number Selection.
21. "Change RFC 3540 "Robust Explicit Congestion Notification (ECN) Signaling with Nonces" to Historic" (https://datatracker.ietf.org/doc/status-change-ecn-signaling-with-nonces-to-historic/). *datatracker.ietf.org*. Retrieved 2023-04-18.
22. RFC 3168, p. 13-14.
23. RFC 3168, p. 15.
24. RFC 3168, p. 18-19.
25. RFC 793.
26. RFC 7323.
27. RFC 2018, 2. Sack-Permitted Option.
28. RFC 2018, 3. Sack Option Format.
29. Heffernan, Andy (August 1998). "Protection of BGP Sessions via the TCP MD5 Signature Option" (https://datatracker.ietf.org/doc/html/rfc2385). IETF. Retrieved 2023-12-30.
30. "Transmission Control Protocol (TCP) Parameters: TCP Option Kind Numbers" (https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml#tcp-parameters-1). IANA. Archived (https://web.archive.org/web/20171002210157/http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml#tcp-parameters-1) from the original on 2017-10-02. Retrieved 2017-10-19.
31. RFC 9293, 3.3.2. State Machine Overview.
32. Kurose, James F. (2017). *Computer networking : a top-down approach* (https://www.worldcat.org/oclc/936004518). Keith W. Ross (7th ed.). Harlow, England. p. 286. ISBN 978-0-13-359414-0. OCLC 936004518 (https://search.worldcat.org/oclc/936004518).
33. Tanenbaum, Andrew S. (2003-03-17). *Computer Networks* (https://archive.org/details/computernetworks00tane_2) (Fourth ed.). Prentice Hall. ISBN 978-0-13-066102-9.
34. RFC 1122, 4.2.2.13. Closing a Connection.
35. Karn & Partridge 1991, p. 364.
36. RFC 9002, 4.2. Monotonically Increasing Packet Numbers.
37. Mathis; Mathew; Semke; Mahdavi; Ott (1997). "The macroscopic behavior of the TCP congestion avoidance algorithm". *ACM SIGCOMM Computer Communication Review*. **27** (3): 67–82. CiteSeerX 10.1.1.40.7002 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.7002). doi:10.1145/263932.264023 (https://doi.org/10.1145%2F263932.264023). S2CID 1894993 (https://api.semanticscholar.org/CorpusID:1894993).
38. RFC 3522, p. 4.
39. Leung, Ka-cheong; Li, Victor O.k.; Yang, Daiqin (2007). "An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges" (https://ieeexplore.ieee.org/document/4118693). *IEEE Transactions on Parallel and Distributed Systems*. **18** (4): 522–535. doi:10.1109/TPDS.2007.1011 (https://doi.org/10.1109%2FTPDS.2007.1011).
40. Johannessen, Mads (2015). *Investigate reordering in Linux TCP* (http://urn.nb.no/URN:NBN:no-51662) (MSc thesis). University of Oslo.
41. Cheng, Yuchung (2015). *RACK: a time-based fast loss detection for TCP draft-cheng-tcpm-rack-00* (https://www.ietf.org/proceedings/94/slides/slides-94-tcpm-6.pdf) (PDF). IETF94. Yokohama: IETF.
42. RFC 8985.
43. Cheng, Yuchung; Cardwell, Neal; Dukkipati, Nandita; Jha, Priyaranjan (2017). *RACK: a time-based fast loss recovery draft-ietf-tcpm-rack-02* (https://datatracker.ietf.org/meeting/100/materials/slides-100-tcpm-draft-ietf-tcpm-rack-01.pdf) (PDF). IETF100. Yokohama: IETF.
44. RFC 6298, p. 2.
45. Zhang 1986, p. 399.
46. Karn & Partridge 1991, p. 365.
47. Ludwig & Katz 2000, p. 31-33.
48. Gurtov & Ludwig 2003, p. 2.
49. Gurtov & Floyd 2004, p. 1.
50. RFC 6298, p. 4.
51. Karn & Partridge 1991, p. 370-372.
52. Allman & Paxson 1999, p. 268.
53. RFC 7323, p. 7.
54. Stone; Partridge (2000). "When the CRC and TCP checksum disagree" (http://citeseer.ist.psu.edu/stone00when.html). *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. *ACM SIGCOMM Computer Communication Review*. pp. 309–319. CiteSeerX 10.1.1.27.7611 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.7611). doi:10.1145/347059.347561 (https://doi.org/10.1145%2F347059.347561). ISBN 978-1581132236. S2CID 9547018 (https://api.semanticscholar.org/CorpusID:9547018). Archived (https://web.archive.org/web/20080505024952/http://citeseer.ist.psu.edu/stone00when.html) from the original on 2008-05-05. Retrieved 2008-04-28.
55. RFC 5681.
56. RFC 6298.
57. RFC 1122.
58. RFC 2018, p. 10.
59. RFC 9002, 4.4. No Reneging.

60. "TCP window scaling and broken routers" (https://lwn.net/Articles/92727/). *LWN.net*. Archived (https://web.archive.org/web/20200331213612/https://lwn.net/Articles/92727/) from the original on 2020-03-31. Retrieved 2016-07-21.
61. RFC 3522.
62. "IP sysctl" (https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt). *Linux Kernel Documentation*. Archived (https://web.archive.org/web/20160305080444/https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt) from the original on 5 March 2016. Retrieved 15 December 2018.
63. Wang, Eve. "TCP timestamp is disabled" (https://web.archive.org/web/20181215225201/https://social.technet.microsoft.com/Forums/office/en-US/6b1e4653-320f-4dbf-8b1a-64d27d8464fc/tcp-timestamp-is-disabled). *Technet – Windows Server 2012 Essentials*. Microsoft. Archived from the original (https://social.technet.microsoft.com/Forums/office/en-US/6b1e4653-320f-4dbf-8b1a-64d27d8464fc/tcp-timestamp-is-disabled) on 2018-12-15. Retrieved 2018-12-15.
64. David Murray; Terry Koziniec; Sebastian Zander; Michael Dixon; Polychronis Koutsakis (2017). "An Analysis of Changing Enterprise Network Traffic Characteristics" (http://profiles.murdoch.edu.au/myprofile/david-murray/files/2012/06/An_Analysis_of_Changing_Enterprise_Network_Traffic_Characteristics-22.pdf) (PDF). The 23rd Asia-Pacific Conference on Communications (APCC 2017). Archived (https://web.archive.org/web/20171003124654/http://profiles.murdoch.edu.au/myprofile/david-murray/files/2012/06/An_Analysis_of_Changing_Enterprise_Network_Traffic_Characteristics-22.pdf) (PDF) from the original on 3 October 2017. Retrieved 3 October 2017.
65. Gont, Fernando (November 2008). "On the implementation of TCP urgent data" (http://www.gont.com.ar/talks/IETF73/ietf73-tcpm-urgent-data.ppt). 73rd IETF meeting. Archived (https://web.archive.org/web/20190516181338/https://www.gont.com.ar/talks/IETF73/ietf73-tcpm-urgent-data.ppt) from the original on 2019-05-16. Retrieved 2009-01-04.
66. Peterson, Larry (2003). *Computer Networks* (https://archive.org/details/computernetworks00pete_974). Morgan Kaufmann. p. 401 (https://archive.org/details/computernetworks00pete_974/page/n419). ISBN 978-1-55860-832-0.
67. Richard W. Stevens (November 2011). *TCP/IP Illustrated. Vol. 1, The protocols* (https://archive.org/details/tcpipillustrated00stev). Addison-Wesley. pp. Chapter 20. ISBN 978-0-201-63346-7.
68. "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) (PDF). Archived from the original on March 6, 2009. Retrieved 2010-12-23.
69. Survey of Security Hardening Methods for Transmission Control Protocol (TCP) Implementations (https://tools.ietf.org/html/draft-ietf-tcpm-tcp-security)
70. Jakob Lell (13 August 2013). "Quick Blind TCP Connection Spoofing with SYN Cookies" (http://www.jakoblell.com/blog/2013/08/13/quick-blind-tcp-connection-spoofing-with-syn-cookies/). Archived (https://web.archive.org/web/20140222101226/http://www.jakoblell.com/blog/2013/08/13/quick-blind-tcp-connection-spoofing-with-syn-cookies/) from the original on 2014-02-22. Retrieved 2014-02-05.
71. "Some insights about the recent TCP DoS (Denial of Service) vulnerabilities" (https://web.archive.org/web/20130618235445/http://www.gont.com.ar/talks/hacklu2009/fgont-hacklu2009-tcp-security.pdf) (PDF). Archived from the original (http://www.gont.com.ar/talks/hacklu2009/fgont-hacklu2009-tcp-security.pdf) (PDF) on 2013-06-18. Retrieved 2010-12-23.
72. "Exploiting TCP and the Persist Timer Infiniteness" (http://phrack.org/issues.html?issue=66&id=9#article). Archived (https://web.archive.org/web/20100122131412/http://www.phrack.org/issues.html?issue=66&id=9#article) from the original on 2010-01-22. Retrieved 2010-01-22.
73. "PUSH and ACK Flood" (https://f5.com/glossary/push-and-ack-flood). *f5.com*. Archived (https://web.archive.org/web/20170928005428/https://f5.com/glossary/push-and-ack-flood) from the original on 2017-09-28. Retrieved 2017-09-27.
74. Laurent Joncheray (1995). "Simple Active Attack Against TCP" (https://www.usenix.org/legacy/publications/library/proceedings/security95/full_papers/joncheray.pdf) (PDF). Retrieved 2023-06-04.
75. John T. Hagen; Barry E. Mullins (2013). "TCP veto: A novel network attack and its Application to SCADA protocols". *2013 IEEE PES Innovative Smart Grid Technologies Conference (ISGT)*. *Innovative Smart Grid Technologies (ISGT), 2013 IEEE PES*. pp. 1–6. doi:10.1109/ISGT.2013.6497785 (https://doi.org/10.1109%2FISGT.2013.6497785). ISBN 978-1-4673-4896-6. S2CID 25353177 (https://api.semanticscholar.org/CorpusID:25353177).
76. RFC 9293, 4. Glossary.
77. RFC 8095, p. 6.
78. Paasch & Bonaventure 2014, p. 51.
79. RFC 6182.
80. RFC 6824.
81. Raiciu; Barre; Pluntke; Greenhalgh; Wischik; Handley (2011). "Improving datacenter performance and robustness with multipath TCP" (https://web.archive.org/web/20200404105843/https://inl.info.ucl.ac.be/publications/improving-datacenter-performance-and-robustness-multipath-tcp). *ACM SIGCOMM Computer Communication Review*. **41** (4): 266. CiteSeerX 10.1.1.306.3863 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.306.3863). doi:10.1145/2043164.2018467 (https://doi.org/10.1145%2F2043164.2018467). Archived from the original (http://inl.info.ucl.ac.be/publications/improving-datacenter-performance-and-robustness-multipath-tcp) on 2020-04-04. Retrieved 2011-06-29.
82. "MultiPath TCP – Linux Kernel implementation" (http://www.multipath-tcp.org/). Archived (https://web.archive.org/web/20130327041817/http://www.multipath-tcp.org/) from the original on 2013-03-27. Retrieved 2013-03-24.
83. Raiciu; Paasch; Barre; Ford; Honda; Duchene; Bonaventure; Handley (2012). "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP" (https://www.usenix.org/conference/nsdi12/how-hard-can-it-be-designing-and-implementing-deployable-multipath-tcp). *Usenix NSDI*: 399–412. Archived (https://web.archive.org/web/20130603045638/https://www.usenix.org/conference/nsdi12/how-hard-can-it-be-designing-and-implementing-deployable-multipath-tcp) from the original on 2013-06-03. Retrieved 2013-03-24.
84. Bonaventure; Seo (2016). "Multipath TCP Deployments" (https://www.ietfjournal.org/multipath-tcp-deployments/). *IETF Journal*. Archived (https://web.archive.org/web/20200223070325/https://www.ietfjournal.org/multipath-tcp-deployments/) from the original on 2020-02-23. Retrieved 2017-01-03.
85. *Cryptographic Protection of TCP Streams (tcpcrypt)* (https://datatracker.ietf.org/doc/html/rfc8548). May 2019. doi:10.17487/RFC8548 (https://doi.org/10.17487%2FRFC8548). RFC 8548 (https://datatracker.ietf.org/doc/html/rfc8548).
86. Michael Kerrisk (2012-08-01). "TCP Fast Open: expediting web services" (https://lwn.net/Articles/508865/). LWN.net. Archived (https://web.archive.org/web/20140803234830/http://lwn.net/Articles/508865/) from the original on 2014-08-03. Retrieved 2014-07-21.
87. RFC 7413.
88. RFC 6937.
89. Grigorik, Ilya (2013). *High-performance browser networking* (1. ed.). Beijing: O'Reilly. ISBN 978-1449344764.
90. RFC 6013.
91. RFC 7805.
92. RFC 8546, p. 6.
93. RFC 8558, p. 3.
94. RFC 9065, 2. Current Uses of Transport Headers within the Network.
95. RFC 9065, 3. Research, Development, and Deployment.
96. RFC 8558, p. 8.
97. RFC 9170, 2.3. Multi-party Interactions and Middleboxes.
98. RFC 9170, A.5. TCP.
99. Papastergiou et al. 2017, p. 620.
100. Edeline & Donnet 2019, p. 175-176.
101. Raiciu et al. 2012, p. 1.
102. Hesmans et al. 2013, p. 1.
103. Rybczyńska 2020.
104. Papastergiou et al. 2017, p. 621.
105. Corbet 2015.
106. Briscoe et al. 2016, pp. 29–30.
107. Marx 2020, HOL blocking in HTTP/1.1.
108. Marx 2020, Bonus: Transport Congestion Control.
109. IETF HTTP Working Group, Why just one TCP connection?.
110. Corbet 2018.
111. RFC 7413, p. 3.
112. Sy et al. 2020, p. 271.
113. Chen et al. 2021, p. 8-9.
114. Ghedini 2018.
115. Chen et al. 2021, p. 3-4.
116. RFC 7413, p. 1.
117. Blanton & Allman 2002, p. 1-2.
118. Blanton & Allman 2002, p. 4-5.
119. Blanton & Allman 2002, p. 3-4.
120. Blanton & Allman 2002, p. 6-8.
121. Bruyeron, Hemon & Zhang 1998, p. 67.
122. Bruyeron, Hemon & Zhang 1998, p. 72.
123. Bhat, Rizk & Zink 2017, p. 14.
124. RFC 9002, 4.5. More ACK Ranges.

125. "TCP performance over CDMA2000 RLP" (https://web.archive.org/web/20110503193100/http://academic.research.microsoft.com/Paper/3352358.aspx). Archived from the original (http://academic.research.microsoft.com/Paper/3352358.aspx) on 2011-05-03. Retrieved 2010-08-30.
126. Muhammad Adeel & Ahmad Ali Iqbal (2007). "TCP Congestion Window Optimization for CDMA2000 Packet Data Networks". *Fourth International Conference on Information Technology (ITNG'07)*. pp. 31–35. doi:10.1109/ITNG.2007.190 (https://doi.org/10.1109%2FITNG.2007.190). ISBN 978-0-7695-2776-5. S2CID 8717768 (https://api.semanticscholar.org/CorpusID:8717768).
127. "TCP Acceleration" (https://web.archive.org/web/20240422182258/https://www.frame.ie/use-cases/understanding-tcp-and-the-need-for-tcp-acceleration/). Archived from the original (https://www.frame.ie/use-cases/understanding-tcp-and-the-need-for-tcp-acceleration/) on 2024-04-22. Retrieved 2024-04-18.
128. Yunhong Gu, Xinwei Hong, and Robert L. Grossman. "An Analysis of AIMD Algorithm with Decreasing Increases" (http://udt.sourceforge.net/doc/gridnet-v8.pdf) Archived (https://web.archive.org/web/20160305003043/http://udt.sourceforge.net/doc/gridnet-v8.pdf) 2016-03-05 at the Wayback Machine. 2004.
129. RFC 8200.
130. "Wireshark: Offloading" (https://wiki.wireshark.org/CaptureSetup/Offloading). Archived (https://web.archive.org/web/20170131220028/https://wiki.wireshark.org/CaptureSetup/Offloading/) from the original on 2017-01-31. Retrieved 2017-02-24. "Wireshark captures packets before they are sent to the network adapter. It won't see the correct checksum because it has not been calculated yet. Even worse, most OSes don't bother initialize this data so you're probably seeing little chunks of memory that you shouldn't. New installations of Wireshark 1.2 and above disable IP, TCP, and UDP checksum validation by default. You can disable checksum validation in each of those dissectors by hand if needed."
131. "Wireshark: Checksums" (https://www.wireshark.org/docs/wsug_html_chunked/ChAdvChecksums.html). Archived (https://web.archive.org/web/20161022133751/https://www.wireshark.org/docs/wsug_html_chunked/ChAdvChecksums.html) from the original on 2016-10-22. Retrieved 2017-02-24. "Checksum offloading often causes confusion as the network packets to be transmitted are handed over to Wireshark before the checksums are actually calculated. Wireshark gets these "empty" checksums and displays them as invalid, even though the packets will contain valid checksums when they leave the network hardware later."

Bibliography

Requests for Comments

- Cerf, Vint; Dalal, Yogen; Sunshine, Carl (December 1974). *Specification of Internet Transmission Control Program, December 1974 Version* (https://datatracker.ietf.org/doc/html/rfc675). doi:10.17487/RFC0675 (https://doi.org/10.17487%2FRFC0675). RFC 675 (https://datatracker.ietf.org/doc/html/rfc675).
- Postel, Jon (September 1981). *Internet Protocol* (https://datatracker.ietf.org/doc/html/rfc791). doi:10.17487/RFC0791 (https://doi.org/10.17487%2FRFC0791). RFC 791 (https://datatracker.ietf.org/doc/html/rfc791).
- Postel, Jon (September 1981). *Transmission Control Protocol* (https://datatracker.ietf.org/doc/html/rfc793). doi:10.17487/RFC0793 (https://doi.org/10.17487%2FRFC0793). RFC 793 (https://datatracker.ietf.org/doc/html/rfc793).
- Braden, Robert, ed. (October 1989). *Requirements for Internet Hosts – Communication Layers* (https://datatracker.ietf.org/doc/html/rfc1122). doi:10.17487/RFC1122 (https://doi.org/10.17487%2FRFC1122). RFC 1122 (https://datatracker.ietf.org/doc/html/rfc1122).
- Jacobson, Van; Braden, Bob; Borman, Dave (May 1992). *TCP Extensions for High Performance* (https://datatracker.ietf.org/doc/html/rfc1323). doi:10.17487/RFC1323 (https://doi.org/10.17487%2FRFC1323). RFC 1323 (https://datatracker.ietf.org/doc/html/rfc1323).
- Bellovin, Steven M. (May 1996). *Defending Against Sequence Number Attacks* (https://datatracker.ietf.org/doc/html/rfc1948). doi:10.17487/RFC1948 (https://doi.org/10.17487%2FRFC1948). RFC 1948 (https://datatracker.ietf.org/doc/html/rfc1948).
- Mathis, Matt; Mahdavi, Jamshid; Floyd, Sally; Romanow, Allyn (October 1996). *TCP Selective Acknowledgment Options* (https://datatracker.ietf.org/doc/html/rfc2018). doi:10.17487/RFC2018 (https://doi.org/10.17487%2FRFC2018). RFC 2018 (https://datatracker.ietf.org/doc/html/rfc2018).
- Allman, Mark; Paxson, Vern; Stevens, W. Richard (April 1999). *TCP Congestion Control* (https://datatracker.ietf.org/doc/html/rfc2581). doi:10.17487/RFC2581 (https://doi.org/10.17487%2FRFC2581). RFC 2581 (https://datatracker.ietf.org/doc/html/rfc2581).
- Floyd, Sally; Mahdavi, Jamshid; Mathis, Matt; Podolsky, Matthew (July 2000). *An Extension to the Selective Acknowledgement (SACK) Option for TCP* (https://datatracker.ietf.org/doc/html/rfc2883). doi:10.17487/RFC2883 (https://doi.org/10.17487%2FRFC2883). RFC 2883 (https://datatracker.ietf.org/doc/html/rfc2883).
- Ramakrishnan, K. K.; Floyd, Sally; Black, David (September 2001). *The Addition of Explicit Congestion Notification (ECN) to IP* (https://datatracker.ietf.org/doc/html/rfc3168). doi:10.17487/RFC3168 (https://doi.org/10.17487%2FRFC3168). RFC 3168 (https://datatracker.ietf.org/doc/html/rfc3168).
- Ludwig, Reiner; Meyer, Michael (April 2003). *The Eifel Detection Algorithm for TCP* (https://datatracker.ietf.org/doc/html/rfc3522). doi:10.17487/RFC3522 (https://doi.org/10.17487%2FRFC3522). RFC 3522 (https://datatracker.ietf.org/doc/html/rfc3522).
- Spring, Neil; Weatherall, David; Ely, David (June 2003). *Robust Explicit Congestion Notification (ECN) Signaling with Nonces* (https://datatracker.ietf.org/doc/html/rfc3540). doi:10.17487/RFC3540 (https://doi.org/10.17487%2FRFC3540). RFC 3540 (https://datatracker.ietf.org/doc/html/rfc3540).
- Allman, Mark; Paxson, Vern; Blanton, Ethan (September 2009). *TCP Congestion Control* (https://datatracker.ietf.org/doc/html/rfc5681). doi:10.17487/RFC5681 (https://doi.org/10.17487%2FRFC5681). RFC 5681 (https://datatracker.ietf.org/doc/html/rfc5681).
- Simpson, William Allen (January 2011). *TCP Cookie Transactions (TCPCT)* (https://datatracker.ietf.org/doc/html/rfc6013). doi:10.17487/RFC6013 (https://doi.org/10.17487%2FRFC6013). RFC 6013 (https://datatracker.ietf.org/doc/html/rfc6013).
- Ford, Alan; Raiciu, Costin; Handley, Mark; Barre, Sebastien; Iyengar, Janardhan (March 2011). *Architectural Guidelines for Multipath TCP Development* (https://datatracker.ietf.org/doc/html/rfc6182). doi:10.17487/RFC6182 (https://doi.org/10.17487%2FRFC6182). RFC 6182 (https://datatracker.ietf.org/doc/html/rfc6182).
- Paxson, Vern; Allman, Mark; Chu, H.K. Jerry; Sargent, Matt (June 2011). *Computing TCP's Retransmission Timer* (https://datatracker.ietf.org/doc/html/rfc6298). doi:10.17487/RFC6298 (https://doi.org/10.17487%2FRFC6298). RFC 6298 (https://datatracker.ietf.org/doc/html/rfc6298).
- Ford, Alan; Raiciu, Costin; Handley, Mark; Bonaventure, Olivier (January 2013). *TCP Extensions for Multipath Operation with Multiple Addresses* (https://datatracker.ietf.org/doc/html/rfc6824). doi:10.17487/RFC6824 (https://doi.org/10.17487%2FRFC6824). RFC 6824 (https://datatracker.ietf.org/doc/html/rfc6824).
- Mathis, Matt; Dukkipati, Nandita; Cheng, Yuchung (May 2013). *Proportional Rate Reduction for TCP* (https://datatracker.ietf.org/doc/html/rfc6937). doi:10.17487/RFC6937 (https://doi.org/10.17487%2FRFC6937). RFC 6937 (https://datatracker.ietf.org/doc/html/rfc6937).
- Borman, David; Braden, Bob; Jacobson, Van (September 2014). Scheffenegger, Richard (ed.). *TCP Extensions for High Performance* (https://datatracker.ietf.org/doc/html/rfc7323). doi:10.17487/RFC7323 (https://doi.org/10.17487%2FRFC7323). RFC 7323 (https://datatracker.ietf.org/doc/html/rfc7323).
- Duke, Martin; Braden, Robert; Eddy, Wesley M.; Blanton, Ethan; Zimmermann, Alexander (February 2015). *A Roadmap for Transmission Control Protocol (TCP) Specification Documents* (https://datatracker.ietf.org/doc/html/rfc7414). doi:10.17487/RFC7414 (https://doi.org/10.17487%2FRFC7414). RFC 7414 (https://datatracker.ietf.org/doc/html/rfc7414).
- Cheng, Yuchung; Chu, Jerry; Radhakrishnan, Sivasankar; Jain, Arvind (December 2014). *TCP Fast Open* (https://datatracker.ietf.org/doc/html/rfc7413). doi:10.17487/RFC7413 (https://doi.org/10.17487%2FRFC7413). RFC 7413 (https://datatracker.ietf.org/doc/html/rfc7413).
- Zimmermann, Alexander; Eddy, Wesley M.; Eggert, Lars (April 2016). *Moving Outdated TCP Extensions and TCP-Related Documents to Historic or Informational Status* (https://datatracker.ietf.org/doc/html/rfc7805). doi:10.17487/RFC7805 (https://doi.org/10.17487%2FRFC7805). RFC 7805

(https://datatracker.ietf.org/doc/html/rfc7805).
- Fairhurst, Gorry; Trammell, Brian; Kuehlewind, Mirja, eds. (March 2017). *Services Provided by IETF Transport Protocols and Congestion Control Mechanisms* (https://datatracker.ietf.org/doc/html/rfc8095). doi:10.17487/RFC8095 (https://doi.org/10.17487%2FRFC8095). RFC 8095 (https://datatracker.ietf.org/doc/html/rfc8095).
- Cheng, Yuchung; Cardwell, Neal; Dukkipati, Nandita; Jha, Priyaranjan, eds. (February 2021). *The RACK-TLP Loss Detection Algorithm for TCP* (https://datatracker.ietf.org/doc/html/rfc8985). doi:10.17487/RFC8985 (https://doi.org/10.17487%2FRFC8985). RFC 8985 (https://datatracker.ietf.org/doc/html/rfc8985).
- Deering, Stephen E.; Hinden, Robert M. (July 2017). *Internet Protocol, Version 6 (IPv6) Specification* (https://datatracker.ietf.org/doc/html/rfc8200). doi:10.17487/RFC8200 (https://doi.org/10.17487%2FRFC8200). RFC 8200 (https://datatracker.ietf.org/doc/html/rfc8200).
- Trammell, Brian; Kuehlewind, Mirja (April 2019). *The Wire Image of a Network Protocol* (https://datatracker.ietf.org/doc/html/rfc8546). doi:10.17487/RFC8546 (https://doi.org/10.17487%2FRFC8546). RFC 8546 (https://datatracker.ietf.org/doc/html/rfc8546).
- Hardie, Ted, ed. (April 2019). *Transport Protocol Path Signals* (https://datatracker.ietf.org/doc/html/rfc8558). doi:10.17487/RFC8558 (https://doi.org/10.17487%2FRFC8558). RFC 8558 (https://datatracker.ietf.org/doc/html/rfc8558).
- Iyengar, Jana; Swett, Ian, eds. (May 2021). *QUIC Loss Detection and Congestion Control* (https://datatracker.ietf.org/doc/html/rfc9002). doi:10.17487/RFC9002 (https://doi.org/10.17487%2FRFC9002). RFC 9002 (https://datatracker.ietf.org/doc/html/rfc9002).
- Fairhurst, Gorry; Perkins, Colin (July 2021). *Considerations around Transport Header Confidentiality, Network Operations, and the Evolution of Internet Transport Protocols* (https://datatracker.ietf.org/doc/html/rfc9065). doi:10.17487/RFC9065 (https://doi.org/10.17487%2FRFC9065). RFC 9065 (https://datatracker.ietf.org/doc/html/rfc9065).
- Thomson, Martin; Pauly, Tommy (December 2021). *Long-Term Viability of Protocol Extension Mechanisms* (https://datatracker.ietf.org/doc/html/rfc9170). doi:10.17487/RFC9170 (https://doi.org/10.17487%2FRFC9170). RFC 9170 (https://datatracker.ietf.org/doc/html/rfc9170).
- Eddy, Wesley M., ed. (August 2022). *Transmission Control Protocol (TCP)* (https://datatracker.ietf.org/doc/html/rfc9293). doi:10.17487/RFC9293 (https://doi.org/10.17487%2FRFC9293). RFC 9293 (https://datatracker.ietf.org/doc/html/rfc9293).

Other documents

- Allman, Mark; Paxson, Vern (October 1999). "On estimating end-to-end network path properties" (https://doi.org/10.1145%2F316194.316230). *ACM SIGCOMM Computer Communication Review*. **29** (4): 263–274. doi:10.1145/316194.316230 (https://doi.org/10.1145%2F316194.316230). hdl:2060/20000004338 (https://hdl.handle.net/2060%2F20000004338).
- Bhat, Divyashri; Rizk, Amr; Zink, Michael (June 2017). "Not so QUIC: A Performance Study of DASH over QUIC". *NOSSDAV'17: Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video*. pp. 13–18. doi:10.1145/3083165.3083175 (https://doi.org/10.1145%2F3083165.3083175). S2CID 32671949 (https://api.semanticscholar.org/CorpusID:32671949).
- Blanton, Ethan; Allman, Mark (January 2002). "On making TCP more robust to packet reordering" (https://www.icir.org/mallman/pubs/BA02/BA02.pdf) (PDF). *ACM SIGCOMM Computer Communication Review*. **32**: 20–30. doi:10.1145/510726.510728 (https://doi.org/10.1145%2F510726.510728). S2CID 15305731 (https://api.semanticscholar.org/CorpusID:15305731).
- Briscoe, Bob; Brunstrom, Anna; Petlund, Andreas; Hayes, David; Ros, David; Tsang, Ing-Jyh; Gjessing, Stein; Fairhurst, Gorry; Griwodz, Carsten; Welzl, Michael (2016). "Reducing Internet Latency: A Survey of Techniques and Their Merits". *IEEE Communications Surveys & Tutorials*. **18** (3): 2149–2196. doi:10.1109/COMST.2014.2375213 (https://doi.org/10.1109%2FCOMST.2014.2375213). hdl:2164/8018 (https://hdl.handle.net/2164%2F8018). S2CID 206576469 (https://api.semanticscholar.org/CorpusID:206576469).
- Bruyeron, Renaud; Hemon, Bruno; Zhang, Lixa (April 1998). "Experimentations with TCP selective acknowledgment". *ACM SIGCOMM Computer Communication Review*. **28** (2): 54–77. doi:10.1145/279345.279350 (https://doi.org/10.1145%2F279345.279350). S2CID 15954837 (https://api.semanticscholar.org/CorpusID:15954837).
- Chen, Shan; Jero, Samuel; Jagielski, Matthew; Boldyreva, Alexandra; Nita-Rotaru, Cristina (2021). "Secure Communication Channel Establishment: TLS 1.3 (Over TCP Fast Open) versus QUIC" (https://doi.org/10.1007%2Fs00145-021-09389-w). *Journal of Cryptology*. **34** (3). doi:10.1007/s00145-021-09389-w (https://doi.org/10.1007%2Fs00145-021-09389-w). S2CID 235174220 (https://api.semanticscholar.org/CorpusID:235174220).
- Corbet, Jonathan (8 December 2015). "Checksum offloads and protocol ossification" (https://lwn.net/Articles/667059/). *LWN.net*.
- Corbet, Jonathan (29 January 2018). "QUIC as a solution to protocol ossification" (https://lwn.net/Articles/745590/). *LWN.net*.
- Edeline, Korian; Donnet, Benoit (2019). *A Bottom-Up Investigation of the Transport-Layer Ossification*. 2019 Network Traffic Measurement and Analysis Conference (TMA). doi:10.23919/TMA.2019.8784690 (https://doi.org/10.23919%2FTMA.2019.8784690).
- Ghedini, Alessandro (26 July 2018). "The Road to QUIC" (https://blog.cloudflare.com/the-road-to-quic/). Cloudflare.
- Gurtov, Andrei; Floyd, Sally (February 2004). *Resolving Acknowledgment Ambiguity in non-SACK TCP* (https://www.cs.helsinki.fi/u/gurtov/papers/heuristics.pdf) (PDF). Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN'04).
- Gurtov, Andrei; Ludwig, Reiner (2003). *Responding to Spurious Timeouts in TCP* (https://www.cs.helsinki.fi/u/gurtov/papers/infocom03.pdf) (PDF). IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies. doi:10.1109/INFCOM.2003.1209251 (https://doi.org/10.1109%2FINFCOM.2003.1209251).
- Hesmans, Benjamin; Duchene, Fabien; Paasch, Christoph; Detal, Gregory; Bonaventure, Olivier (2013). *Are TCP extensions middlebox-proof?*. HotMiddlebox '13. CiteSeerX 10.1.1.679.6364 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.679.6364). doi:10.1145/2535828.2535830 (https://doi.org/10.1145%2F2535828.2535830).
- IETF HTTP Working Group. "HTTP/2 Frequently Asked Questions" (https://http2.github.io/faq/).
- Karn, Phil; Partridge, Craig (November 1991). "Improving round-trip time estimates in reliable transport protocols" (https://doi.org/10.1145%2F118544.118549). *ACM Transactions on Computer Systems*. **9** (4): 364–373. doi:10.1145/118544.118549 (https://doi.org/10.1145%2F118544.118549).
- Ludwig, Reiner; Katz, Randy Howard (January 2000). "The Eifel algorithm: making TCP robust against spurious retransmissions" (https://doi.org/10.1145%2F505688.505692). *ACM SIGCOMM Computer Communication Review*. doi:10.1145/505688.505692 (https://doi.org/10.1145%2F505688.505692).
- Marx, Robin (3 December 2020). "Head-of-Line Blocking in QUIC and HTTP/3: The Details" (https://calendar.perfplanet.com/2020/head-of-line-blocking-in-quic-and-http-3-the-details/).
- Paasch, Christoph; Bonaventure, Olivier (1 April 2014). "Multipath TCP". *Communications of the ACM*. **57** (4): 51–57. doi:10.1145/2578901 (https://doi.org/10.1145%2F2578901). S2CID 17581886 (https://api.semanticscholar.org/CorpusID:17581886).
- Papastergiou, Giorgos; Fairhurst, Gorry; Ros, David; Brunstrom, Anna; Grinnemo, Karl-Johan; Hurtig, Per; Khademi, Naeem; Tüxen, Michael; Welzl, Michael; Damjanovic, Dragana; Mangiante, Simone (2017). "De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives". *IEEE Communications Surveys & Tutorials*. **19**: 619–639. doi:10.1109/COMST.2016.2626780 (https://doi.org/10.1109%2FCOMST.2016.2626780). hdl:2164/8317 (https://hdl.handle.net/2164%2F8317). S2CID 1846371 (https://api.semanticscholar.org/CorpusID:1846371).
- Rybczyńska, Marta (13 March 2020). "A QUIC look at HTTP/3" (https://lwn.net/Articles/814522/). *LWN.net*.
- Sy, Erik; Mueller, Tobias; Burkert, Christian; Federrath, Hannes; Fischer, Mathias (2020). "Enhanced Performance and Privacy for TLS over TCP Fast Open" (https://doi.org/10.2478%2Fpopets-2020-0027). *Proceedings on Privacy Enhancing Technologies*. **2020** (2): 271–287. arXiv:1905.03518 (https://arxiv.org/abs/1905.03518). doi:10.2478/popets-2020-0027 (https://doi.org/10.2478%2Fpopets-2020-0027).

- Zhang, Lixia (5 August 1986). "Why TCP timers don't work well". *ACM SIGCOMM Computer Communication Review*. **16** (3): 397–405. doi:10.1145/1013812.18216 (https://doi.org/10.1145%2F1013812.18216).

Further reading

- Stevens, W. Richard (1994-01-10). *TCP/IP Illustrated, Volume 1: The Protocols* (https://archive.org/details/tcpipillustrated00stev). Addison-Wesley Pub. Co. ISBN 978-0-201-63346-7.
- Stevens, W. Richard; Wright, Gary R (1994). *TCP/IP Illustrated, Volume 2: The Implementation* (https://archive.org/details/tcpipillustrated00stev_1). Addison-Wesley. ISBN 978-0-201-63354-2.
- Stevens, W. Richard (1996). *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. Addison-Wesley. ISBN 978-0-201-63495-2.**

External links

- Oral history interview with Robert E. Kahn (http://purl.umn.edu/107387)
- IANA Port Assignments (https://www.iana.org/assignments/port-numbers)
- IANA TCP Parameters (https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml)
- 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)
- Checksum example (http://mathforum.org/library/drmath/view/54379.html)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Transmission_Control_Protocol&oldid=1255521330"