Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Learn to edit
Community portal
Recent changes
Upload file

Tools
What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item

Print/export
Download as PDF
Printable version

In other projects
Wikimedia Commons

Languages
Deutsch
Español
Français
한국어
Italiano
Русский
Tagalog
Tiếng Việt
中文
61 more
Edit links

Transmission Control Protocol

From Wikipedia, the free encyclopedia

The **Transmission Control Protocol** (**TCP**) is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol (IP). Therefore, the entire suite is commonly referred to as *TCP/IP*. TCP provides reliable, ordered, and error-checked delivery of a stream of octets (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, and a connection between client and server is established before data can be sent. 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), which provides a connectionless datagram service that prioritizes time over reliability. TCP employs network congestion avoidance. However, there are vulnerabilities to TCP including denial of service, connection hijacking, TCP veto, and reset attack.

Internet protocol suite

Application layer
BGP · DHCP(v6) · DNS · FTP · HTTP · HTTPS · IMAP · LDAP · MGCP · MQTT · NNTP · NTP · POP · PTP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer
TCP · UDP · DCCP · SCTP · RSVP · *more...*

Internet layer
IP (IPv4 · IPv6) · ICMP(v6) · ECN · IGMP · IPsec · *more...*

Link layer
ARP · NDP · OSPF · Tunnels (L2TP) · PPP · MAC (Ethernet · Wi-Fi · DSL · ISDN · FDDI) *more...*

v · t · e

Contents [hide]

Historical origin [edit]

In May 1974, Vint Cerf and Bob Kahn described an internetworking protocol for sharing resources using packet switching among network nodes.[1] The authors had been working with Gérard Le Lann to incorporate concepts from the French CYCLADES project into the new network.[2] 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. It contains the first attested use of the term *Internet*, as a shorthand for *internetworking*.[3]

A central control component of this model was the *Transmission Control Program* that incorporated both connection-oriented links and datagram services between hosts. The monolithic Transmission Control Program was later divided into a modular architecture consisting of the *Transmission Control Protocol* and the *Internet Protocol*. This resulted in a networking model that became known informally as *TCP/IP*, although formally it was variously referred to as the Department of Defense (DOD) model, and ARPANET model, and eventually also as the *Internet Protocol Suite*.

In 2004, Vint Cerf and Bob Kahn received the Turing Award for their foundational work on TCP/IP.[4][5]

Network function [edit]

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 behaviour, IP packets may be lost, duplicated, or delivered out of order. TCP detects these problems, requests re-transmission of lost data, rearranges out-of-order data and even helps 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.[6]

TCP is a reliable stream delivery service which 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 acknowledgement with re-transmission*. This requires the receiver to respond with an acknowledgement message as it receives the data. The sender keeps a record of each packet it sends and maintains a timer from when the packet was sent. The sender re-transmits a packet if the timer expires before receiving the acknowledgement. The timer is needed in case a packet gets lost or corrupted.[6]

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 [edit]

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.[7]

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*[8] 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.[9]

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. 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 segment header

Offsets	Octet	0								1								2								3							
Octet	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	**0**	Source port																Destination port															
4	**32**	Sequence number																															
8	**64**	Acknowledgment number (if ACK set)																															
12	**96**	Data offset				Reserved **0 0 0**			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	**128**	Checksum																Urgent pointer (if URG set)															
20	**160**	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.)																															
⋮	**⋮**																																
60	**480**																																

Source port (16 bits)
: Identifies the sending port.

Destination port (16 bits)
: Identifies the receiving port.

Sequence number (32 bits)
: Has a dual role:

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

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

Data offset (4 bits)
: Specifies the size of the TCP header in 32-bit words. The minimum size header is 5 words and the maximum is 15 words thus giving the minimum size of 20 bytes and maximum of 60 bytes, allowing for up to 40 bytes of options in the header. This field gets its name from the fact that it is also the offset from the start of the TCP segment to the actual data.

Reserved (3 bits)
: For future use and should be set to zero.

Flags (9 bits)
: Contains 9 1-bit flags (control bits) as follows:

- NS (1 bit): ECN-nonce - concealment protection[a]
- 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.[b]
- ECE (1 bit): ECN-Echo has a dual role, depending on the value of the SYN flag. It indicates:
  - If the SYN flag is set (1), that the TCP peer is ECN capable.
  - If the SYN flag is clear (0), that a packet with Congestion Experienced flag set (ECN=11) in the IP header was received during normal transmission.[b] This serves as an indication of network congestion (or impending congestion) to the TCP sender.
- URG (1 bit): Indicates that the Urgent pointer field is significant
- ACK (1 bit): Indicates that the Acknowledgment field is significant. All packets after the initial SYN packet sent by the client should have this flag set.
- PSH (1 bit): Push function. Asks to push the buffered data to the receiving application.
- RST (1 bit): Reset the connection
- SYN (1 bit): Synchronize sequence numbers. Only the first packet sent from each end should have this flag set. Some other flags and fields change meaning based on this flag, and some are only valid when it is set, and others when it is clear.
- FIN (1 bit): Last packet from sender

Window size (16 bits)

The size of the *receive window*, which specifies the number of window size units[c] that the sender of this segment is currently willing to receive.[d] (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 (Variable 0–320 bits, in units of 32 bits)

The length of this field is determined by the *data offset* field. Options have up to three fields: Option-Kind (1 byte), Option-Length (1 byte), Option-Data (variable). The Option-Kind field indicates the type of option and is the only field that is not optional. Depending on 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	N/A	N/A	End of options list	
1	N/A	N/A	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 `[SYN]`
3	3	S	Window scale	See § Window scaling for details[10] `[SYN]`
4	2	N/A	Selective Acknowledgement permitted	See § Selective acknowledgments for details[11] `[SYN]`
5	N (10, 18, 26, or 34)	BBBB, EEEE, ...	Selective ACKnowledgement (SACK)[12]	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[13]

The remaining Option-Kind values are historical, obsolete, experimental, not yet standardized, or unassigned. Option number assignments are maintained by the IANA.[14]

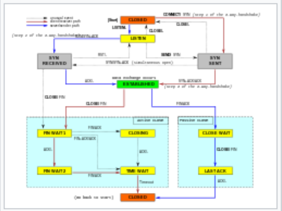
Padding

The TCP header padding is used to ensure that the TCP header ends, and data begins, on a 32-bit boundary. The padding is composed of zeros.[15]

Protocol operation [edit]

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:[16]



A Simplified TCP State Diagram. See TCP EFSM diagram for a more detailed state diagram including the states inside the ESTABLISHED state.

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 the remote TCP received the acknowledgment of its connection termination request.[e]
CLOSED	Server and client	No connection state at all.

Connection establishment [edit]

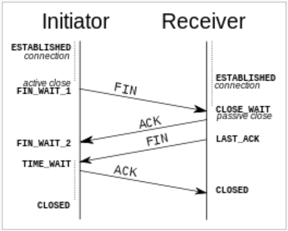
Before a client attempts to connect with a server, the server must first bind to and listen at a port to open it up for connections: this is called a passive open. Once the passive open is established, a client may establish a connection by initiating an active open using the three-way (or 3-step) handshake:

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. Steps 2 and 3 establish and acknowledge the sequence number for the other direction. Following the completion of these steps, both the client and server have received acknowledgments and a full-duplex communication is established.

Connection termination [edit]

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



Connection termination

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

It is also possible to terminate the connection by a 3-way handshake, when host A sends a FIN and host B replies with a FIN & ACK (merely combines 2 steps into one) and host A replies with an ACK.[17]

Some operating systems, such as Linux and HP-UX, implement a half-duplex close sequence in the TCP stack. 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.[18] This assures a TCP application that the remote process has read all the transmitted data by waiting for the signal FIN, before it actively closes the connection. The remote process cannot distinguish between an RST signal for *connection aborting* and *data loss*. Both cause the remote stack to lose all data received.

Some applications using the TCP open/close handshaking protocol may find the RST problem on active close. As an example:

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

For a program flow like above, a TCP/IP stack like that described above does not guarantee that all the data arrives to the other application if unread data has arrived at this end.

Resource usage [edit]

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

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

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

Data transfer [edit]

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

- Ordered data transfer: the destination host rearranges segments according to a sequence number[6]
- Retransmission of lost packets: any cumulative stream not acknowledged is retransmitted[6]
- Error-free data transfer[19]
- Flow control: limits the rate a sender transfers data to guarantee reliable delivery. The receiver continually hints the sender on how much data can be received (controlled by the sliding window). When the receiving host's buffer fills, the next acknowledgment contains a 0 in the window size, to stop transfer and allow the data in the buffer to be processed.[6]
- Congestion control[6]

Reliable transmission [edit]

TCP uses a *sequence number* to identify each byte of data. The sequence number identifies the order of the bytes sent from each computer so that the data can be reconstructed in order, regardless of any packet reordering, or packet loss that may occur during transmission. The sequence number of the first byte is 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.

Acknowledgements (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 (abbreviated as RTO) and duplicate cumulative acknowledgements (DupAcks).

Dupack-based retransmission [edit]

If a single segment (say segment 100) in a stream is lost, then the receiver cannot acknowledge packets above no. 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 acknowledgements, 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.[20] Sometimes selective acknowledgements (SACKs) are used to provide explicit feedback about the segments that have been received. This greatly improves TCP's ability to retransmit the right segments.

Timeout-based retransmission [edit]

When a sender transmits a segment, it initializes a timer with a conservative estimate of the arrival time of the acknowledgement. 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.[21] This guards against excessive transmission traffic due to faulty or malicious actors, such as man-in-the-middle denial of service attackers.

Error detection [edit]

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

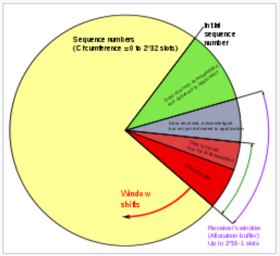
To assure correctness a checksum field is included; see checksum computation section for details on checksumming. The TCP checksum is a weak check by modern standards. Data Link Layers with high bit error rates may require additional link error correction/detection capabilities. The weak checksum is partially compensated for by the common use of a CRC or better integrity check at layer 2, below both TCP and IP, such as is used in PPP or the Ethernet frame. However, this does not mean that the 16-bit TCP checksum is redundant: remarkably, introduction of errors in packets between CRC-protected hops is common, but the end-to-end 16-bit TCP checksum catches most of these simple errors.[22] This is the end-to-end principle at work.

Flow control [edit]

TCP uses an end-to-end flow control protocol to avoid having the sender send data too fast for the TCP receiver to receive and process it reliably. Having a mechanism for flow control is essential in an environment where machines of diverse network speeds communicate. For example, if a PC sends data to a smartphone that is slowly processing received data, the smartphone must regulate the data flow so as not to be overwhelmed.[6]

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 acknowledgement and window update from the receiving host.

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



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

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

Congestion control [edit]

Main article: TCP congestion control

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

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

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

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

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

Maximum segment size [edit]

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

MSS announcement is also often called "MSS negotiation". Strictly speaking, the MSS is not "negotiated" between the originator and the receiver, because that would imply that both originator and receiver will negotiate and agree upon a single, unified MSS that applies to all communication in both directions of the connection. In fact, two completely independent values of MSS are permitted for the two directions of data flow in a TCP connection.[23] This situation may arise, for example, if one of the devices participating in a connection has an extremely limited amount of memory reserved (perhaps even smaller than the overall discovered Path MTU) for processing incoming TCP segments.

Selective acknowledgments [edit]

See also: SACK Panic

Relying purely on the cumulative acknowledgment scheme employed by the original TCP protocol 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 which 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 acknowledgement can specify 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 can 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. The TCP receiver sends a D-ACK to indicate that no segments were lost, and the TCP sender can then reinstate the higher transmission-rate.

The SACK option is not mandatory, and comes into operation only if both parties support it. This is negotiated when a connection is established. SACK uses 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).

Window scaling [edit]

Main article: TCP window scale option

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

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

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

Some routers and packet firewalls rewrite the window scaling factor during a transmission. This causes sending and receiving sides to assume different TCP window sizes. The result is non-stable traffic that may be very slow. The problem is visible on some sites behind a defective router.[24]

TCP timestamps [edit]

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* (see RFC 1323 for details). PAWS is used when the receive window crosses the sequence number wraparound boundary. In the case where a packet was potentially retransmitted it answers the question: "Is this sequence number in the first 4 GB or the second?" And the timestamp is used to break the tie.

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

Recent Statistics show that the level of Timestamp adoption has stagnated, at ~40%, owing to Windows server dropping support since Windows Server 2008.[25]

TCP timestamps are enabled by default In Linux kernel.,[26] and disabled by default in Windows Server 2008, 2012 and 2016.[27]

Out-of-band data [edit]

It is possible to interrupt or abort the queued stream instead of waiting for the stream to finish. This is done by specifying the data as *urgent*. This tells the receiving program to process it immediately, along with the rest of the urgent data. When finished, TCP informs the application and resumes back to the stream queue. An example is when TCP is used for a remote login session, the user can send a keyboard sequence that interrupts or aborts the program at the other end. These signals are most often needed when a program on the remote machine fails to operate correctly. The signals must be sent without waiting for the program to finish its current transfer.[6]

TCP out-of-band data was not designed for the modern Internet. The *urgent* pointer only alters the processing on the remote host and doesn't expedite any processing on the network itself. When it gets to the remote host there are two slightly different interpretations of the protocol, which means only single bytes of OOB data are reliable. This is assuming it is reliable at all as it is one of the least commonly used protocol elements and tends to be poorly implemented.[28][29]

Forcing data delivery [edit]

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".[6] There is no way to indicate or control it in user space using Berkeley sockets and it is controlled by protocol stack only.[30]

Vulnerabilities [edit]

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,[31] and is currently being pursued within the IETF.[32]

Denial of service [edit]

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.[33] Sockstress is a similar attack, that might be mitigated with system resource management.[34] An advanced DoS attack involving the exploitation of the TCP Persist Timer was analyzed in Phrack #66.[35] PUSH and ACK floods are other variants.[36]

Connection hijacking [edit]

Main article: TCP sequence prediction attack

An attacker who is able to eavesdrop a TCP session and redirect packets can hijack a TCP connection. To do so, the attacker learns the sequence number from the ongoing communication and forges a false segment that looks like the next segment in the stream. Such a simple hijack can result in one packet being erroneously accepted at one end. When the receiving host acknowledges the extra segment to the other side of the connection, synchronization is lost. Hijacking might be combined with Address Resolution Protocol (ARP) or routing attacks that allow taking control of the packet flow, so as to get permanent control of the hijacked TCP connection.[37]

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

TCP veto [edit]

An attacker who can eavesdrop and predict the size of the next packet to be sent can cause the receiver to accept a malicious payload without disrupting the existing connection. The attacker injects a malicious packet with the sequence number and a payload size of the next expected packet. When the legitimate packet is ultimately received, it is found to have the same sequence number and length as a packet already received and is silently dropped as a normal duplicate packet—the legitimate packet is "vetoed" by the malicious packet. Unlike in connection hijacking, the connection is never desynchronized and communication continues as normal after the malicious payload is accepted. TCP veto gives the attacker less control over the communication, but makes the attack particularly resistant to detection. The large increase in network traffic from the ACK storm is avoided. The only evidence to the receiver that something is amiss is a single duplicate packet, a normal occurrence in an IP network. The sender of the vetoed packet never sees any evidence of an attack.[38]

Another vulnerability is TCP reset attack.

TCP ports [edit]

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

Port numbers are categorized into three basic categories: well-known, registered, and dynamic/private. The well-known ports are assigned by the Internet Assigned Numbers Authority (IANA) and are typically used by system-level or root processes. Well-known applications running as servers and passively listening for connections typically use these ports. Some examples include: FTP (20 and 21), SSH (22), TELNET (23), SMTP (25), HTTP over SSL/TLS (443), and HTTP (80). Note, as of the latest standard, HTTP/3, QUIC is used as a transport instead of TCP. Registered ports are typically used by end user applications as ephemeral source ports when contacting servers, but they can also identify named services that have been registered by a third party. Dynamic/private ports can also be used by end user applications, but are less commonly so. Dynamic/private ports do not contain any meaning outside of any particular TCP connection.

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

Development [edit]

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

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

TCP Interactive (iTCP) [39] is a research effort into TCP extensions that allows applications to subscribe to TCP events and register handler components that can launch applications for various purposes, including application-assisted congestion control.

Multipath TCP (MPTCP) [40][41] 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 utilization of different networks, which brings higher throughput and better handover capabilities. Multipath TCP also brings performance benefits in datacenter environments.[42] The reference implementation[43] of Multipath TCP is being developed in the Linux kernel.[44] Multipath TCP is used to support the Siri voice recognition application on iPhones, iPads and Macs [45]

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

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

TCP Fast Open is an extension to speed up the opening of successive TCP connections between two endpoints. It works by skipping the three-way handshake using a cryptographic "cookie". It is similar to an earlier proposal called T/TCP, which was not widely adopted due to security issues.[46] TCP Fast Open was published as RFC 7413 in 2014.[47]

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.[48] The algorithm is designed to improve the speed of recovery and is the default congestion control algorithm in Linux 3.2+ kernels.[49]

TCP over wireless networks [edit]

TCP was originally designed for wired networks. Packet loss is considered to be the result of network congestion and the congestion window size is reduced dramatically as a precaution. However, wireless links are known to experience sporadic and usually temporary losses due to fading, shadowing, hand off, interference, and other radio effects, that are not strictly congestion. After the (erroneous) back-off of the congestion window size, due to wireless packet loss, there may be a congestion avoidance phase with a conservative decrease in window size. This causes the radio link to be underutilized. Extensive research on combating these harmful effects has been conducted. Suggested solutions can be categorized as end-to-end solutions, which require modifications at the client or server,[50] 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.[50][51]

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

Hardware implementations [edit]

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

Debugging [edit]

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 [edit]

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.[52]

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 [edit]

TCP checksum for IPv4 [edit]

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

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

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

TCP pseudo-header for checksum computation (IPv4)

Bit offset	0–3	4–7	8–15	16–31
0	Source address			
32	Destination address			
64	Zeros		Protocol	TCP length
96	Source port			Destination port
128	Sequence number			

160	Acknowledgement number			
192	Data offset	Reserved	Flags	Window
224	Checksum			Urgent pointer
256	Options (optional)			
256/288+	Data			

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

TCP checksum for IPv6 [edit]

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

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

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

TCP pseudo-header for checksum computation (IPv6)

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

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

Checksum offload [edit]

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.[53] 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.[54] 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.

RFC documents [edit]

- RFC 675 – Specification of Internet Transmission Control Program, December 1974 Version
- RFC 793 – TCP v4
- STD 7 – Transmission Control Protocol, Protocol specification
- RFC 1122 – includes some error corrections for TCP
- RFC 1323 – TCP Extensions for High Performance [Obsoleted by RFC 7323]
- RFC 1379 – Extending TCP for Transactions—Concepts [Obsoleted by RFC 6247]
- RFC 1948 – Defending Against Sequence Number Attacks
- RFC 2018 – TCP Selective Acknowledgment Options
- RFC 5681 – TCP Congestion Control
- RFC 6247 – Moving the Undeployed TCP Extensions RFC 1072, RFC 1106, RFC 1110, RFC 1145, RFC 1146, RFC 1379, RFC 1644, and RFC 1693 to Historic Status
- RFC 6298 – Computing TCP's Retransmission Timer
- RFC 6824 – TCP Extensions for Multipath Operation with Multiple Addresses
- RFC 7323 – TCP Extensions for High Performance
- RFC 7414 – A Roadmap for TCP Specification Documents

See also [edit]

- Connection-oriented communication
- List of TCP and UDP port numbers (a long list of ports and services)
- Micro-bursting (networking)
- T/TCP variant of TCP

- TCP global synchronization
- TCP pacing
- Transport layer § Comparison of transport layer protocols
- WTCP a proxy-based modification of TCP for wireless networks

Notes [edit]

a. ^ Experimental: see RFC 3540
b. ^ a b Added to header by RFC 3168
c. ^ Windows size units are, by default, bytes.
d. ^ Window size is relative to the segment identified by the sequence number in the acknowledgment field.
e. ^ According to RFC 793 a connection can stay in TIME-WAIT for a maximum of four minutes known as two maximum segment lifetime (MSL).

References [edit]

1. ^ Vinton G. Cerf; Robert E. Kahn (May 1974). *"A Protocol for Packet Network Intercommunication"* (PDF). *IEEE Transactions on Communications*. **22** (5): 637–648. doi:10.1109/tcom.1974.1092259. Archived from the original (PDF) on March 4, 2016.
2. ^ Bennett, Richard (September 2009). "Designed for Change: End-to-End Arguments, Internet Innovation, and the Net Neutrality Debate" (PDF). Information Technology and Innovation Foundation. p. 11. Retrieved 11 September 2017.
3. ^ Cerf, Vinton; Dalal, Yogen; Sunshine, Carl (December 1974), RFC 675, *Specification of Internet Transmission Control Protocol*
4. ^ "Robert E Kahn - A.M. Turing Award Laureate". *amturing.acm.org*.
5. ^ "Vinton Cerf - A.M. Turing Award Laureate". *amturing.acm.org*.
6. ^ a b c d e f g h i Comer, Douglas E. (2006). *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. **1** (5th ed.). Prentice Hall. ISBN 978-0-13-187671-2.
7. ^ "TCP (Transmission Control Protocol)". Retrieved 2019-06-26.
8. ^ "RFC 791 – section 2.2".
9. ^ *Transmission Control Protocol*. IETF. September 1981. doi:10.17487/RFC0793. RFC 793.
10. ^ *TCP Extensions for High Performance*. sec. 2.2. RFC 1323.
11. ^ "RFC 2018, TCP Selective Acknowledgement Options, Section 2".
12. ^ "RFC 2018, TCP Selective Acknowledgement Options, Section 3".
13. ^ "RFC 1323, TCP Extensions for High Performance, Section 3.2".
14. ^ "Transmission Control Protocol (TCP) Parameters: TCP Option Kind Numbers". IANA.
15. ^ RFC 793 section 3.1
16. ^ RFC 793 Section 3.2
17. ^ Tanenbaum, Andrew S. (2003-03-17). *Computer Networks* (Fourth ed.). Prentice Hall. ISBN 978-0-13-066102-9.
18. ^ RFC 1122, Section 4.2.2.13
19. ^ "TCP Definition". Retrieved 2011-03-12.
20. ^ 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. doi:10.1145/263932.264023.
21. ^ Paxson, V.; Allman, M.; Chu, J.; Sargent, M. (June 2011). "The Basic Algorithm". *Computing TCP's Retransmission Timer*. IETF. p. 2. sec. 2. doi:10.17487/RFC6298. RFC 6298. Retrieved October 24, 2015.
22. ^ Stone; Partridge (2000). "When The CRC and TCP Checksum Disagree". *ACM SIGCOMM Computer Communication Review*: 309–319. CiteSeerX 10.1.1.27.7611. doi:10.1145/347059.347561. ISBN 978-1581132236.
23. ^ "RFC 879".
24. ^ "TCP window scaling and broken routers [LWN.net]".
25. ^ David Murray; Terry Koziniec; Sebastian Zander; Michael Dixon; Polychronis Koutsakis (2017). "An Analysis of Changing Enterprise Network Traffic Characteristics" (PDF). The 23rd Asia-Pacific Conference on Communications (APCC 2017). Retrieved 3 October 2017.
26. ^ "IP sysctl". *Linux Kernel Documentation*. Retrieved 15 December 2018.
27. ^ Wang, Eve. "TCP timestamp is disabled". *Technet - Windows Server 2012 Essentials*. Microsoft. Archived from the original on 2018-12-15. Retrieved 2018-12-15.
28. ^ Gont, Fernando (November 2008). "On the implementation of TCP urgent data". 73rd IETF meeting. Retrieved 2009-01-04.
29. ^ Peterson, Larry (2003). *Computer Networks*. Morgan Kaufmann. p. 401. ISBN 978-1-55860-832-0.
30. ^ Richard W. Stevens (November 2011). *TCP/IP Illustrated. Vol. 1, The protocols*. Addison-Wesley. pp. Chapter 20. ISBN 978-0-201-63346-7.
31. ^ "Security Assessment of the Transmission Control Protocol (TCP)" (PDF). Archived from the original on March 6, 2009. Retrieved 2010-12-23.
32. ^ Security Assessment of the Transmission Control Protocol (TCP)
33. ^ Jakob Lell. "Quick Blind TCP Connection Spoofing with SYN Cookies". Retrieved 2014-02-05.
34. ^ "Some insights about the recent TCP DoS (Denial of Service) vulnerabilities" (PDF).
35. ^ "Exploiting TCP and the Persist Timer Infiniteness".
36. ^ "PUSH and ACK Flood". *f5.com*.
37. ^ "Laurent Joncheray, *Simple Active Attack Against TCP*, 1995".
38. ^ John T. Hagen; Barry E. Mullins (2013). *TCP veto: A novel network attack and its application to SCADA protocols. Innovative Smart Grid Technologies (ISGT), 2013 IEEE PES*. pp. 1–6. doi:10.1109/ISGT.2013.6497785. ISBN 978-1-4673-4896-6.
39. ^ "TCP Interactive". *www.medianet.kent.edu*.
40. ^ RFC 6182
41. ^ RFC 6824
42. ^ Raiciu; Barre; Pluntke; Greenhalgh; Wischik; Handley (2011). "Improving datacenter performance and robustness with multipath TCP". *ACM SIGCOMM Computer Communication Review*. **41** (4): 266. CiteSeerX 10.1.1.306.3863. doi:10.1145/2043164.2018467.
43. ^ "MultiPath TCP - Linux Kernel implementation".
44. ^ Raiciu; Paasch; Barre; Ford; Honda; Duchene; Bonaventure; Handley (2012). "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP". *Usenix NSDI*: 399–412.
45. ^ Bonaventure; Seo (2016). "Multipath TCP Deployments". *IETF Journal*.
46. ^ Michael Kerrisk (2012-08-01). "TCP Fast Open: expediting web services". LWN.net.
47. ^ Yuchung Cheng; Jerry Chu; Sivasankar Radhakrishnan & Arvind Jain (December 2014). "TCP Fast Open". IETF. Retrieved 10 January 2015.
48. ^ "RFC 6937 - Proportional Rate Reduction for TCP". Retrieved 6 June 2014.
49. ^ Grigorik, Ilya (2013). *High-performance browser networking* (1. ed.). Beijing: O'Reilly. ISBN 978-1449344764.
50. ^ a b "TCP performance over CDMA2000 RLP". Archived from the original on 2011-05-03. Retrieved 2010-08-30.
51. ^ Muhammad Adeel & Ahmad Ali Iqbal (2004). *TCP Congestion Window Optimization for CDMA2000 Packet Data Networks. International Conference on Information Technology (ITNG'07)*. pp. 31–35. doi:10.1109/ITNG.2007.190. ISBN 978-0-7695-2776-5.
52. ^ Yunhong Gu, Xinwei Hong, and Robert L. Grossman. "An Analysis of AIMD Algorithm with Decreasing Increases". 2004.
53. ^ "Wireshark: Offloading". "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."
54. ^ "Wireshark: Checksums". "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."

Further reading [edit]

- Stevens, W. Richard (1994-01-10). *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley Pub. Co. ISBN 978-0-201-63346-7.
- Stevens, W. Richard; Wright, Gary R (1994). *TCP/IP Illustrated, Volume 2: The Implementation*. 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*. ISBN 978-0-201-63495-2.**

External links [edit]

- Oral history interview with Robert E. Kahn
- IANA Port Assignments
- IANA TCP Parameters
- John Kristoff's Overview of TCP (Fundamental concepts behind TCP and how it is used to transport data between two endpoints)
- Checksum example
- TCP tutorial

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

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

Categories: Transmission Control Protocol | Transport layer protocols