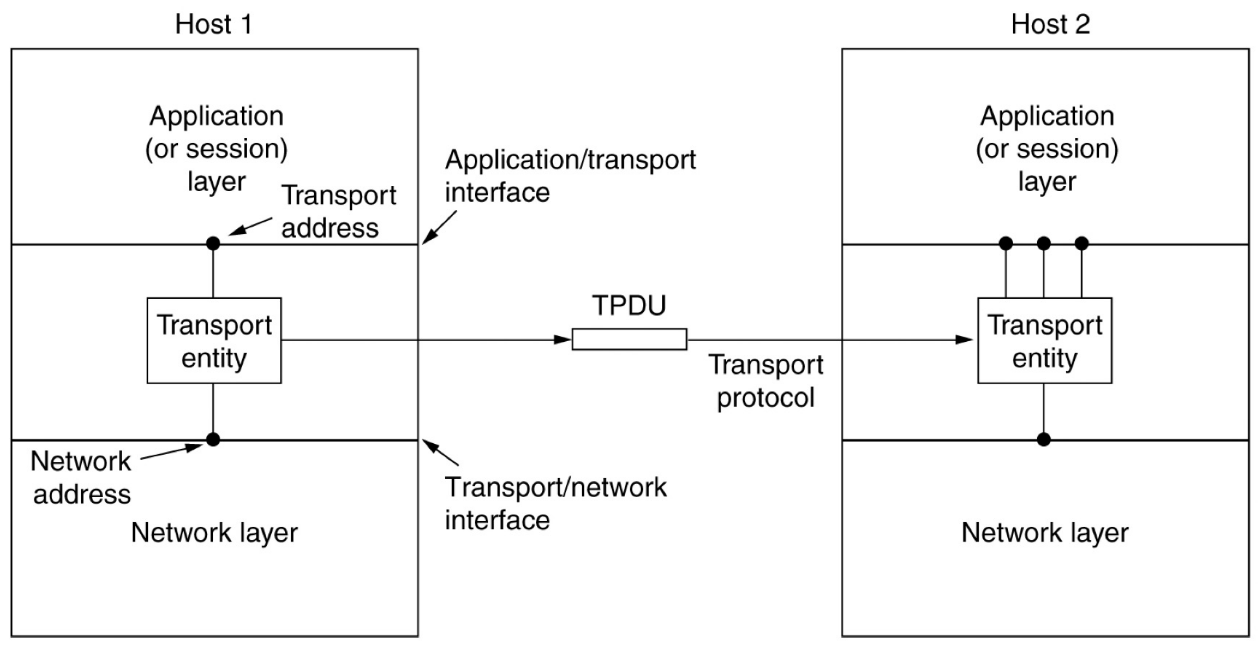
Transport layer

Responsible for delivering data across networks with the desired reliability or quality



The network, transport, and application layers.

The network layer provides end-to-end packet delivery using datagrams or virtual circuits.
The transport layer builds on the network layer to provide data transport from a process on a source machine to a process on a destination machine with a desired level of reliability that is independent of the physical networks currently in use.

The ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective data transmission service to its users, normally processes in the application layer.
To achieve this, the transport layer makes use of the services provided by the network layer. The software and/or hardware within the transport layer that does the work is called the transport entity.

Responsibilities of the transport layer include the following:

Service-point addressing: Computers often run several programs at the same time. For this reason, source-to-destination delivery means delivery not only from one computer to the next but also from a specific process (running program) on one computer to a specific process (running program) on the other. The transport layer header must therefore include a type of address called a service-point address (or port address). The network layer gets each packet to the correct computer; the transport layer gets the entire message to the correct process on that computer.

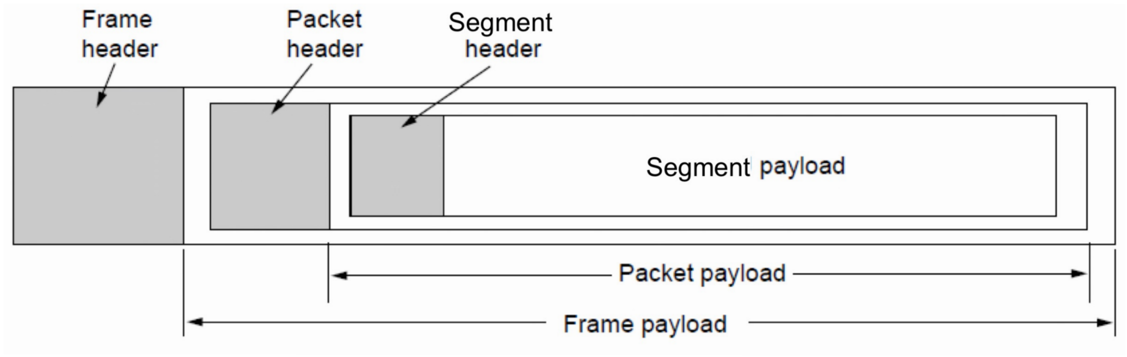
Segmentation and reassembly: A message is divided into transmittable segments, with each segment containing a sequence number. These numbers enable the trans- port layer to reassemble the message correctly upon arriving at the destination and to identify and replace packets that were lost in transmission.

Connection control: The transport layer can be either connectionless or connection- oriented. A connectionless transport layer treats each segment as an independent packet and delivers it to the transport layer at the destination machine. A connection- oriented transport layer makes a connection with the transport layer at the destina- tion machine first before delivering the packets. After all the data are transferred, the connection is terminated.

Flow control: Like the data link layer, the transport layer is responsible for flow control. However, flow control at this layer is performed end to end rather than across a single link.

Error control: Like the data link layer, the transport layer is responsible for error control. However, error control at this layer is performed process-to- process rather than across a single link. The sending transport layer makes sure that the entire message arrives at the receiving transport layer without error (damage, loss, or duplication). Error correction is usually achieved through retransmission.

Transport layer sends segments in packets (in frames)



Segments (exchanged by the transport layer) are contained in packets (exchanged by the network layer). In turn, these packets are contained in frames (exchanged by the data link layer). When a frame arrives, the data link layer processes the frame header and, if the destination address matches for local deliv- ery, passes the contents of the frame payload field up to the network entity. The network entity similarly processes the packet header and then passes the contents of the packet payload up to the transport entity.

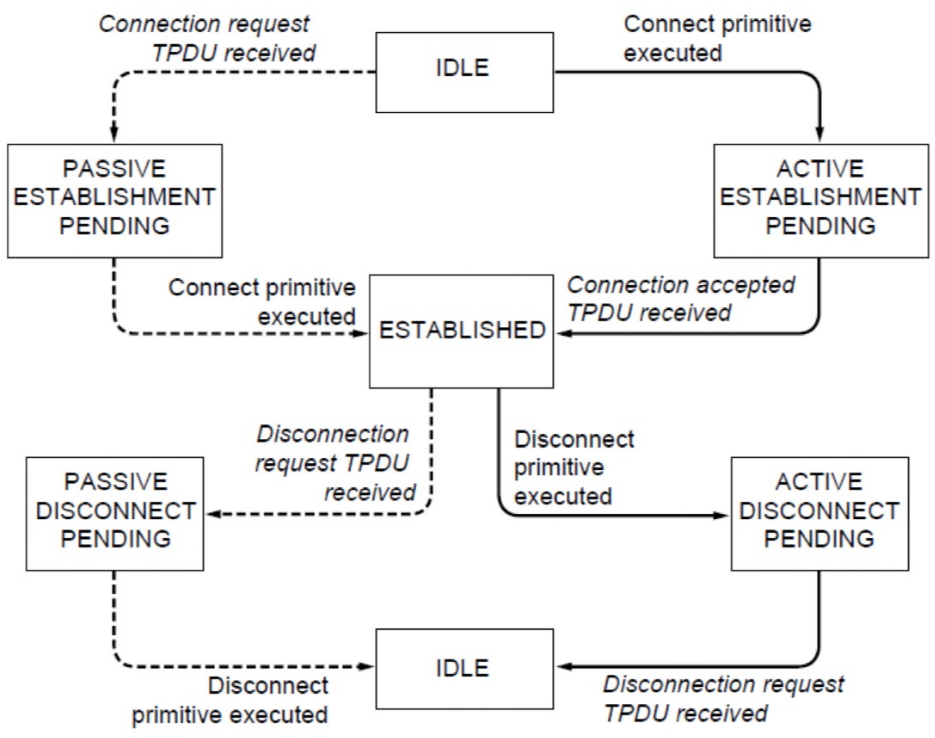
Transport Service Primitives

Primitives that applications might call to transport data for a simple connection-oriented service:

- Client calls CONNECT, SEND, RECEIVE, DISCONNECT
- Server calls LISTEN, RECEIVE, SEND, DISCONNECT

Primitive	Segment sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

State diagram for a simple connection-oriented service



IDLE
Connection request TPDU received
Connect primitive executed
PASSIVE ESTABLISHMENT PENDING
ACTIVE ESTABLISHMENT PENDING
Connect primitive executed
ESTABLISHED
Connection accepted TPDU received
Disconnection request TPDU received
Disconnect primitive executed
PASSIVE DISCONNECT PENDING
ACTIVE DISCONNECT PENDING
IDLE
Disconnect primitive executed
Disconnection request TPDU received

Solid lines (right) show client state sequence

Dashed lines (left) show server state sequence

Transitions in italics are due to segment arrivals.

socket primitives for TCP

Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

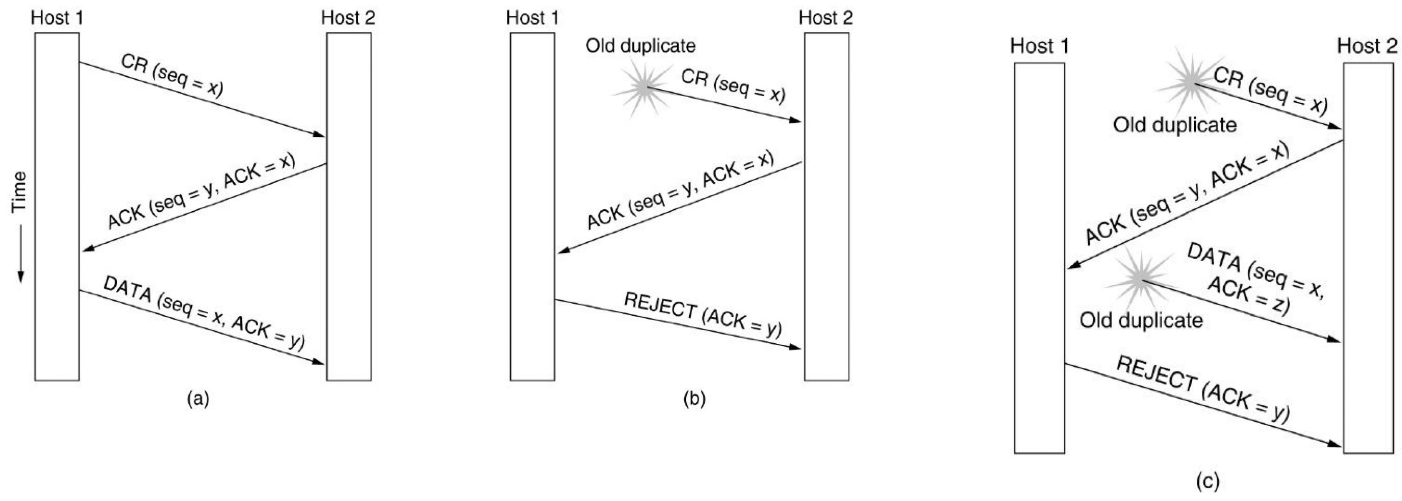
Connection Establishment

Establishing a connection sounds easy, but it is actually surprisingly tricky. At first glance, it would seem sufficient for one transport entity to just send a CONNECTION REQUEST segment to the destination and wait for a CONNECTION ACCEPTED reply. The problem occurs when the network can lose, delay, corrupt, and duplicate packets. This behaviour causes serious complications. Imagine a network that is so congested that acknowledgements hardly ever get back in time and each packet times out and is retransmitted two or three times. Suppose that the network uses datagrams inside and that every packet follows a different route. Some of the packets might get stuck in a traffic jam inside the network and take a long time to arrive. That is, they may be delayed in the network and pop out much later, when the sender thought that they had been lost. The worst possible nightmare is as follows. A user establishes a connection with a bank, sends messages telling the bank to transfer a large amount of money to the account of a not-entirely-trustworthy person. Unfortunately, the packets decide to take the scenic route to the destination and go off exploring a remote corner of the network. The sender then times out and sends them all again. This time the packets take the shortest route and are delivered quickly so the sender releases the connection.

Packet lifetime can be restricted to a known maximum using one (or more) of the following techniques:

1. Restricted network design.
2. Putting a hop counter in each packet.
3. Time stamping each packet.

The first technique includes any method that prevents packets from looping, combined with some way of bounding delay including congestion over the (now known) longest possible path. It is difficult, given that internets may range from a single city to international in scope. The second method consists of having the hop count initialized to some appropriate value and decremented each time the packet is forwarded. The network protocol simply discards any packet whose hop counter becomes zero. The third method requires each packet to bear the time it was created, with the routers agreeing to discard any packet older than some agreed-upon time. This latter method requires the router clocks to be synchronized, which itself is a nontrivial task, and in practice a hop counter is a close enough approximation to age.
TCP uses this three-way handshake to establish connections. Within a connection, a timestamp is used to extend the 32-bit sequence number so that it will not wrap within the maximum packet lifetime, even for gigabit-per-second connections. This mechanism is a fix to TCP that was needed as it was used on faster and faster links. It is described in RFC 1323 and called PAWS (Protection Against Wrapped Sequence numbers). Across connections, for the initial sequence numbers and before PAWS can come into play, TCP originally use the clock-based scheme just described. However, this turned out to have security vulnerability. The clock made it easy for an attacker to predict the next initial sequence number and send packets that tricked the three-way handshake and established a forged connection. To close this hole, pseudorandom initial sequence numbers are used for connections in practice.



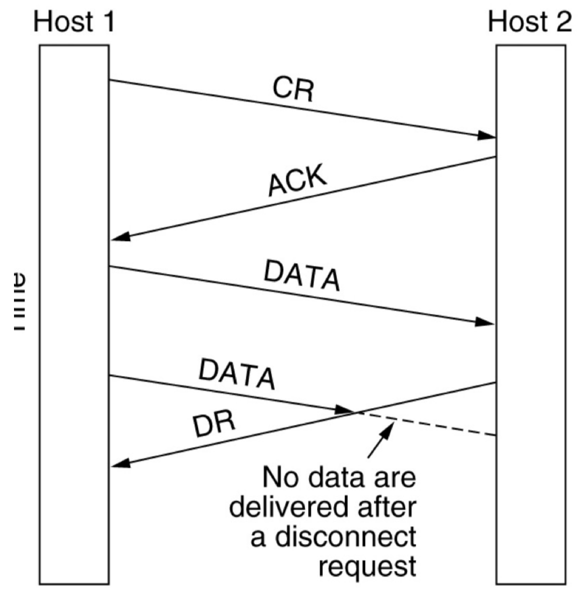
Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST.
(a) Normal operation,
(b) Old CONNECTION REQUEST appearing out of nowhere.
(c) Duplicate CONNECTION REQUEST and duplicate ACK.

In Fig.(b), the first segment is a delayed duplicate CONNECTION REQUEST from an old connection. This segment arrives at host 2 without host 1's knowledge. Host 2 reacts to this segment by sending host 1 an ACK segment, in effect asking for verification that host 1 was indeed trying to set up a new connection. When host 1 rejects host 2's attempt to establish a connection, host 2 realizes that it was tricked by a delayed duplicate and abandons the connection. In this way, a delayed duplicate does no damage
The worst case is when both a delayed CONNECTION REQUEST and an ACK are floating around in the subnet. This case is shown in Fig. (c). As in the previous example, host 2 gets a delayed CONNECTION REQUEST and replies to it. At this point, it is crucial to realize that host 2 has proposed using y as the initial sequence number for host 2 to host 1 traffic, knowing full well that no segments containing sequence number y or acknowledgements to y are still in existence. When the second delayed segment arrives at host 2, the fact that z has been acknowledged rather than y tells host 2 that this, too, is an old duplicate. The important thing to realize here is that there is no combination of old segments that can cause the protocol to fail and have a connection set up by accident when no one wants it.

Connection Release

Releasing a connection is easier than establishing one. Nevertheless, there are more pitfalls than one might expect here. As we mentioned earlier, there are two styles of terminating a connection: asymmetric release and symmetric release Asymmetric release is the way the telephone system works: when one party hangs up, the connection is broken. Symmetric release treats the connection as two separate unidirectional connections and requires each one to be released separately.

Asymmetric release is abrupt and may result in data loss. Consider the scenario of Fig. After the connection is established, host 1 sends a segment that arrives properly at host 2. Then host 1 sends another segment. Unfortunately, host 2 issues a DISCONNECT before the second segment arrives. The result is that the connection is released and data are lost.

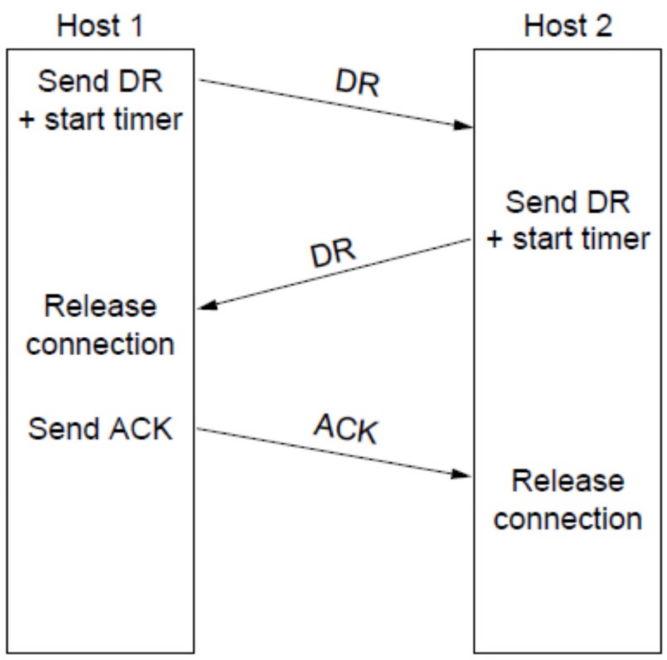


Clearly, a more sophisticated release protocol is needed to avoid data loss. One way is to use symmetric release, in which each direction is released independently of the other one. Here, a host can continue to receive data even after it has sent a DISCONNECT segment.

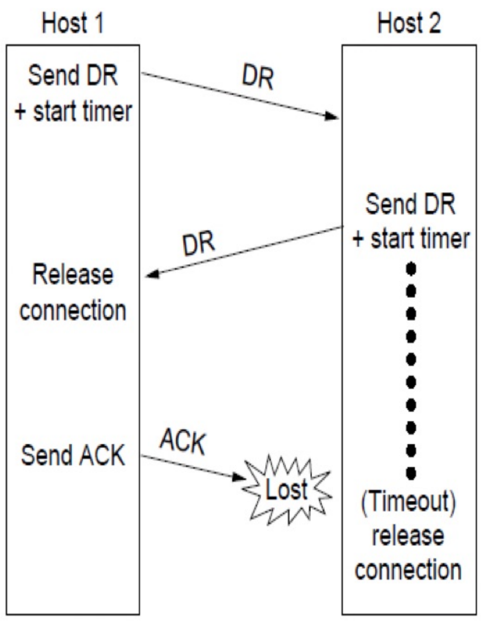
Symmetric release does the job when each process has a fixed amount of data to send and clearly knows when it has sent it. One can envision a protocol in which host 1 says “I am done. Are you done too?” If host 2 responds: “I am done too. Goodbye, the connection can be safely released.”

Normal release sequence, initiated by transport user on Host 1

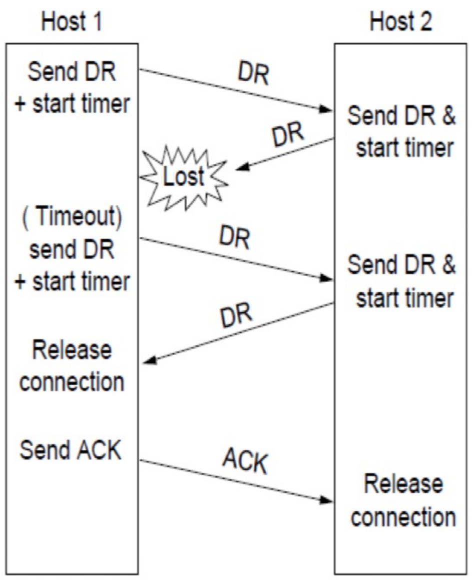
- DR=Disconnect Request
- Both DRs are ACKed by the other side



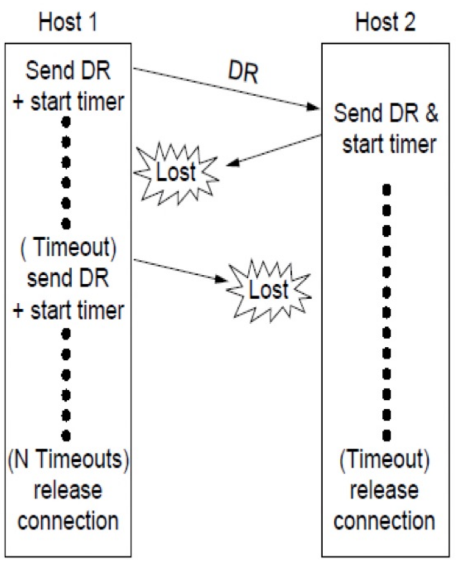
Error cases are handled with timer and retransmission



Final ACK lost, Host 2 times out



Lost DR causes retransmissions


Host 1
Host 2
Send DR + start timer
DR
Send DR & start timer
Lost
(Timeout) send DR + start timer
Lost
(N Timeouts) release connection
(Timeout) release connection

Extreme: Many lost DRs cause both hosts to timeout

Error Control and Flow Control

Foundation for error control is a sliding window (from Link layer) with checksums and retransmissions

Flow control manages buffering at sender/receiver

- Issue is that data goes to/from the network and applications at different times
- Window tells sender available buffering at receiver
- Makes a variable-size sliding window

Congestion Control

Two layers are responsible for congestion control:

— Transport layer, controls the offered load .

— Network layer, experiences congestion

• Desirable bandwidth allocation

• Regulating the sending rate

• Wireless issues

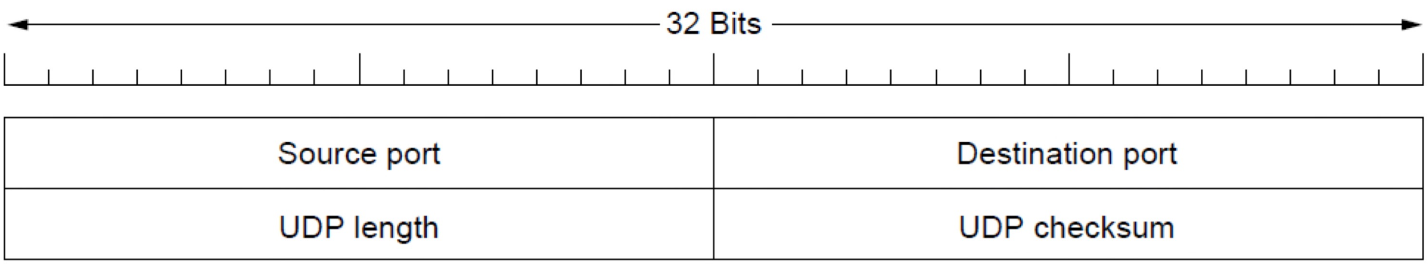
Crash Recovery

If hosts and routers are subject to crashes or connections are long-lived (e.g., large software or media downloads) recovery from these crashes becomes an issue. If the transport entity is entirely within the hosts, recovery from network and router crashes is straightforward. The transport entities expect lost segments all the time and know how to cope with them by using retransmissions. A more troublesome problem is how to recover from host crashes. In particular, it may be desirable for clients to be able to continue working when servers crash and quickly reboot. To illustrate the difficulty, let us assume that one host, the client, is sending a long file to another host, the file server, using a simple Stop-and-wait protocol. The transport layer on the server just passes the incoming segments to the transport user, one by one. Partway through the transmission, the server crashes. When it comes back up, its tables are reinitialized, so it no longer knows precisely where it was. In an attempt to recover its previous status, the server might send a broadcast segment to all other hosts, announcing that it has just crashed and requesting that its clients inform it of the status of all open connections. Each client can be in one of two states: one segment outstanding, S1, or no segments outstanding, S0. Based on only this state information, the client must decide whether to retransmit the most recent segment.

UDP Protocol

UDP provides connectionless, unreliable, datagram service. Connectionless service means that there is no logical connection between the two ends exchanging messages. Each message is an independent entity encapsulated in a datagram.

UDP does not see any relation (connection) between consequent datagram coming from the same source and going to the same destination.
UDP has an advantage: it is message-oriented. It gives boundaries to the messages exchanged. An application program may be designed to use UDP if it is sending small messages and the simplicity and speed is more important for the application than reliability.
User Datagram
UDP packets, called user datagram, have a fixed-size header of 8 bytes made of four fields, each of 2 bytes (16 bits).
. The 16 bits can define a total length of 0 to 65,535 bytes. However, the total length needs to be less because a UDP user datagram is stored in an IP datagram with the total length of 65,535 bytes. The last field can carry the optional checksum



UDP (User Datagram Protocol) is a shim over IP
• Header has ports (TSAPs), length and checksum.

Checksum covers UDP segment and IP pseudoheader
• Fields that change in the network are zeroed out
• Provides an end-to-end delivery check

UDP Services

Process-to-Process Communication

UDP provides process-to-process communication using socket addresses, a combination of IP addresses and port numbers.

Connectionless Services

As mentioned previously, UDP provides a connection less service. This means that each user datagram sent by UDP is an independent datagram. There is no relationship between the different user data grams even if they are coming from the same source process and going to the same destination program.

Flow Control

UDP is a very simple protocol. There is no flow control, and hence no window mechanism. The receiver may overflow with incoming messages.

Error Control

There is no error control mechanism in UDP except for the checksum. This means that the sender does not know if a message has been lost or duplicated.

Checksum

UDP checksum calculation includes three sections: a pseudo header, the UDP header, and the data coming from the application layer. The pseudo header is the part of the header of the IP packet in which the user datagram is to be encapsulated with some fields filled with 0s.

Transmission Control Protocol

Transmission Control Protocol (TCP) is a connection-oriented, reliable protocol. TCP explicitly defines connection establishment, data transfer, and connection teardown phases to provide a connection-oriented service.

TCP Services

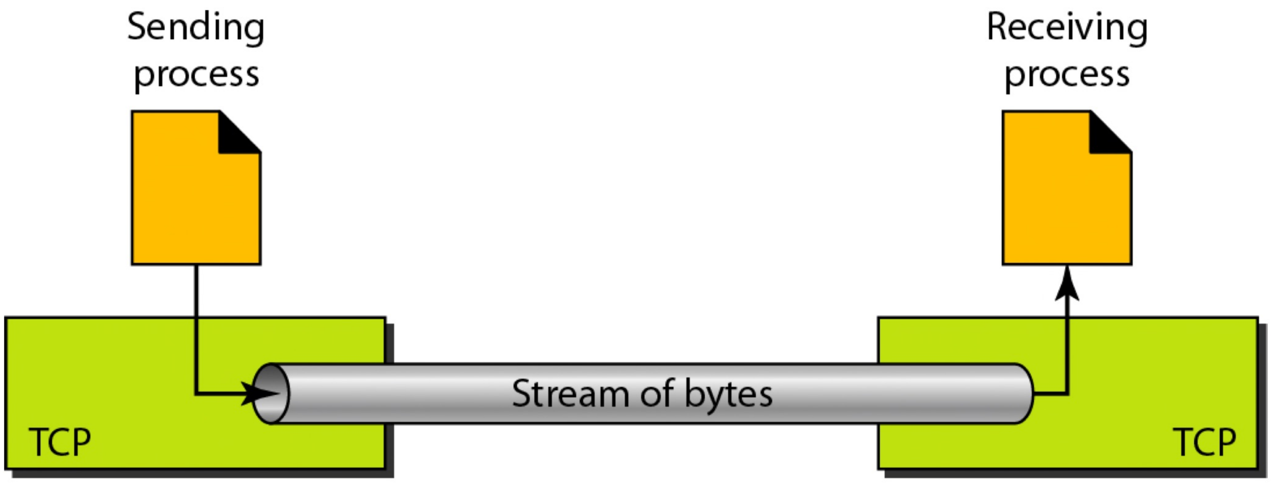
Process-to-Process Communication

As with UDP, TCP provides process-to-process communication using port numbers. We have already given some of the port numbers used by TCP.

Stream Delivery Service

In UDP, a process sends messages with predefined boundaries to UDP for delivery. UDP adds its own header to each of these messages and delivers it to IP for transmission.

TCP, on the other hand, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected by an imaginary "tube" that carries their bytes across the Internet.



Sending and Receiving Buffers

Because the sending and the receiving processes may not necessarily write or read data at the same rate, TCP needs buffers for storage.

There are two buffers, the sending buffer and the receiving buffer, one for each direction.

At the sender, the buffer has three types of chambers. The white section contains empty chambers that can be filled by the sending process (producer). The colored area holds bytes that have been sent but not yet acknowledged. The TCP sender keeps these bytes in the buffer until it receives an acknowledgment. The shaded area contains bytes to be sent by the sending TCP.

The operation of the buffer at the receiver is simpler. The circular buffer is divided into two areas (shown as white and colored).

The white area contains empty chambers to be filled by bytes received from the network.

The colored sections contain received bytes that can be read by the receiving process. When a byte is read by the receiving process, the chamber is recycled and added to the pool of empty chambers.

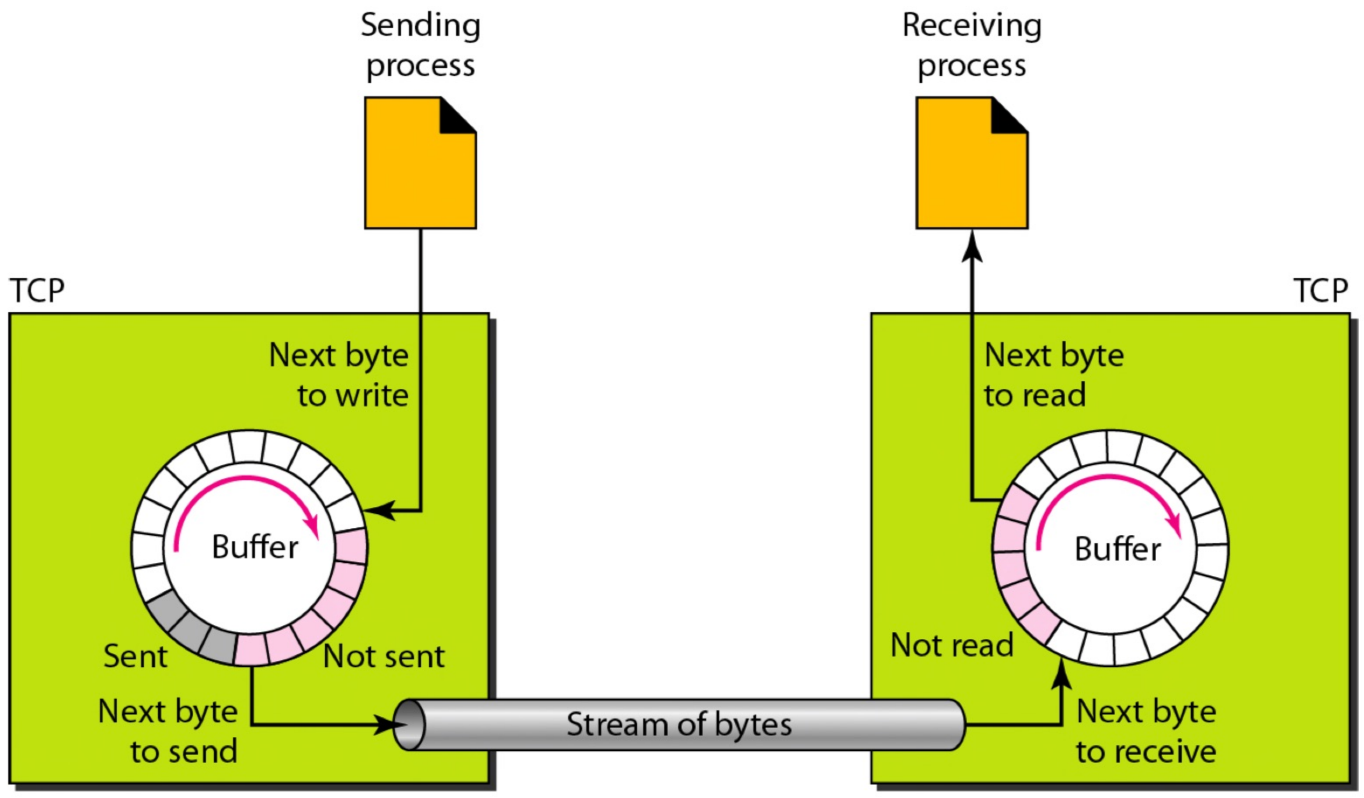


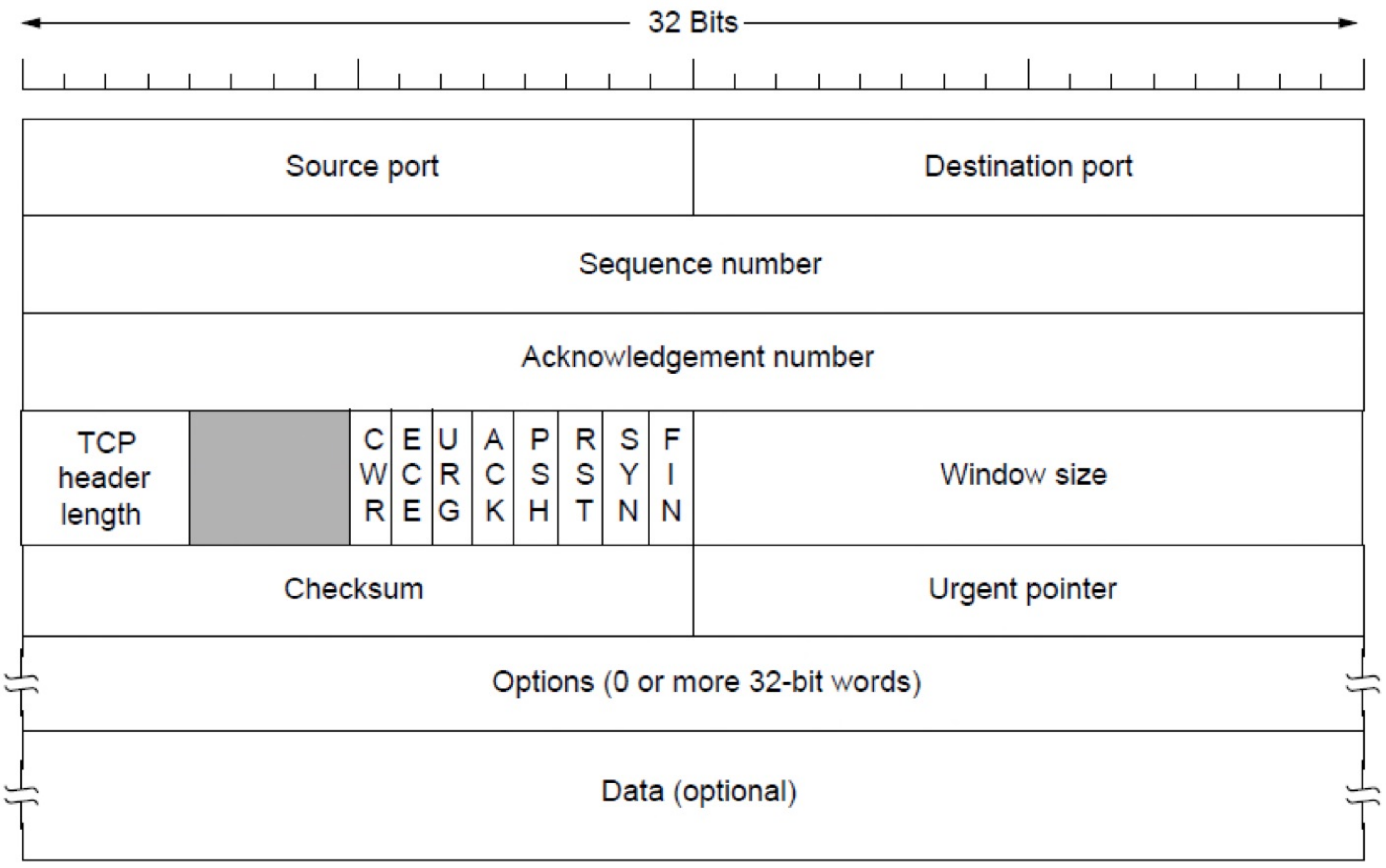
Figure shows the movement of the data in one direction. At the sending site, the buffer has three types of chambers. The white section contains empty chambers that can be filled by the sending process (producer). The gray area holds bytes that have been sent but not yet acknowledged. TCP keeps these bytes in the buffer until it receives an acknowledgment. The colored area contains bytes to be sent by the sending TCP.

Segments Although buffering handles the disparity between the speed of the producing and consuming Processes, we need one more step before we can send data.
The network layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes. At the transport layer, TCP groups a number of bytes together into a packet called a segment.
The segments are encapsulated in an IP datagram and transmitted. This entire operation is transparent to the receiving process.
Format The segment consists of a header of 20 to 60 bytes, followed by data from the application program.The header is 20 bytes if there are no options and up to 60 bytes if it contains options.

Source port address This is a 16-bit field that defines the port number of the application program in the host that is sending the segment.
Destination port address This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment.
Sequence number This 32-bit field defines the number assigned to the first byte of data contained in this segment.
Acknowledgment number This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party.
Header length This 4-bit field indicates the number of 4-byte words in the TCP header. The length of the header can be between 20 and 60 bytes.

TCP Segment Header

TCP header includes addressing (ports), sliding window (seq. / ack. number), flow control (window), error control (checksum) and more.



NS	ECN-nonce concealment protection (experimental)
CWR	Congestion Window Reduced. Set by the sender to indicate that it received a TCP segment with the ECE flag set and had responded in the congestion control mechanism
ECE	ECN-Echo. Has a dual-mode depending on the value of the SYN flag: ∘ SYN set (1) → the TCP peer is ECN capable ∘ SYN clear (0) → a packet with Congestion Experienced flag set (ECN=11) in IP header received during normal transmission. This serves as an indication of network congestion to the TCP sender
URG	Indicates that the Urgent pointer field is significant
ACK	Indicates that the Acknowledgment field is significant. All packets after the initial SYN packet sent by the client should have this flag set
PSH	Push function. Asks to push the buffered data to the receiving application
RST	Reset the connection
SYN	Synchronize sequence numbers. Only the first packet sent from each end should have this flag set
FIN	Last package from the sender