Basics

- The function of transport layer is to provide a reliable end-to-end communications service.
- It also provides data transfer service for the user layers above and shield the upper layers from the details of underlying network.
- This may include message error control and flow control functions. Optionally, it may provide for other transport services such as multiplexing (of multiple transport connection to a single network connectivity), inverse multiplexing (mapping a single transport connection to a multiple number of network connection, for performance purposes).
- The transport layer is responsible for process-to-process delivery—the delivery of a packet, part of a message, from one process to another.
- Two processes communicate in a client/server relationship.

Explain the basic five service primitives of the transport layer protocol.

- A PRIMITIVE means operations.
- A service in a computer network consists of a set of primitives.
- The primitives are to be used by the user to access the service.
- The primitive asks the service to do some action or to report on an action.
- The primitives are system calls.
- The primitive varies for different services.
- The following are some of the primitives used in a Transport Layer.

Primitive	TPDU sent	Meaning
LISTEN	None	Block until some process tries to connect
CONNECT	Connection Request	Actively attempt to establish connection
SEND	Data	Send data
RECEIVE	None	Block until a data TPDU arrives
DISCONNECT	Disconnect Request	Release the connection

- Consider an application with a server and a number of remote clients.
- To start with, the server executes a LISTEN primitive, typically by calling a library procedure that makes a system call to block the server until a client turns up.
- For lack of a better term, we will reluctantly use the somewhat ungainly acronym **TPDU (Transport Protocol Data Unit)** for messages sent from transport entity to transport entity.
- Thus, TPDUs (exchanged by the transport layer) are contained in packets (exchanged by the network layer).
- In turn, packets are contained in frames (exchanged by the data link layer).
- When a frame arrives, the data link layer processes the frame header and passes the contents of the frame payload field up to the network entity.

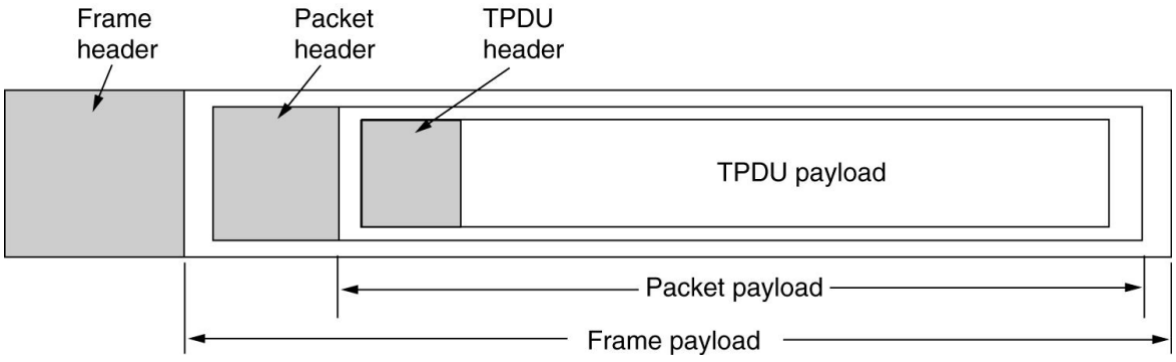


Figure: The nesting of TPDUs, packets, and frames

- When a client wants to talk to the server, it executes a CONNECT primitive.
- The transport entity carries out this primitive by blocking the caller and sending a packet to the server.
- Encapsulated in the payload of this packet is a transport layer message for the server's transport entity.
- The client's CONNECT call causes a CONNECTION REQUEST TPDU to be sent to the server.
- When it arrives, the transport entity checks to see that the server is blocked on a LISTEN (i.e., is interested in handling requests).
- It then unblocks the server and sends a CONNECTION ACCEPTED TPDU back to the client.
- When this TPDU arrives, the client is unblocked and the connection is established.
- Data can now be exchanged using the SEND and RECEIVE primitives.
- In the simplest form, either party can do a (blocking) RECEIVE to wait for the other party to do a SEND. When the TPDU arrives, the receiver is unblocked.
- It can then process the TPDU and send a reply.
- As long as both sides can keep track of whose turn it is to send, this scheme works fine.
- When a connection is no longer needed, it must be released to free up table space within the two transport entities.
- Disconnection has two variants: asymmetric and symmetric.

Q-2 Write about flow control, buffering mechanism in Transport protocols.

- In some ways the flow control problem in the transport layer is the same as in the data link layer, but in other ways it is different.
- The basic similarity is that in both layers a sliding window or other scheme is needed on each connection to keep a fast transmitter from overrunning a slow receiver.
- The main difference is that a router usually has relatively few lines, whereas a host may have numerous connections.
- In the data link layer, the sending side must buffer outgoing frames because they might have to be retransmitted.
- If the subnet provides datagram service, the sending transport entity must also buffer, and for the same reason.
- If the receiver knows that the sender buffers all TPDUs until they are acknowledged, the receiver may or may not dedicate specific buffers to specific connections, as it sees fit. The receiver may, for example, maintain a single buffer pool shared by all connections.
- When a TPDU comes in, an attempt is made to dynamically acquire a new buffer.
- If one is available, the TPDU is accepted; otherwise, it is discarded. Since the sender is prepared to retransmit TPDUs lost by the subnet, no harm is done by having the receiver drop TPDUs, although some resources are wasted.
- The sender just keeps trying until it gets an acknowledgement.
- If the network service is unreliable, the sender must buffer all TPDUs sent, just as in the data link layer.
- However, with reliable network service, other trade-offs become possible.
- In particular, if the sender knows that the receiver always has buffer space, it need not retain copies of the TPDUs it sends.
- However, if the receiver cannot guarantee that every incoming TPDU will be accepted, the sender will have to buffer anyway.
- In the latter case, the sender cannot trust the network layer's acknowledgement, because the acknowledgement means only that the TPDU arrived, not that it was accepted.
- We will come back to this important point later.
- Even if the receiver has agreed to do the buffering, there still remains the question of the buffer size.
- If most TPDUs are nearly the same size, it is natural to organize the buffers as a pool of identically-sized buffers, with one TPDU per buffer, as in Figure (a).
- However, if there is wide variation in TPDU size, from a few characters typed at a terminal to thousands of characters from file transfers, a pool of fixed-sized buffers presents problems.
- If the buffer size is chosen equal to the largest possible TPDU, space will be wasted whenever a short

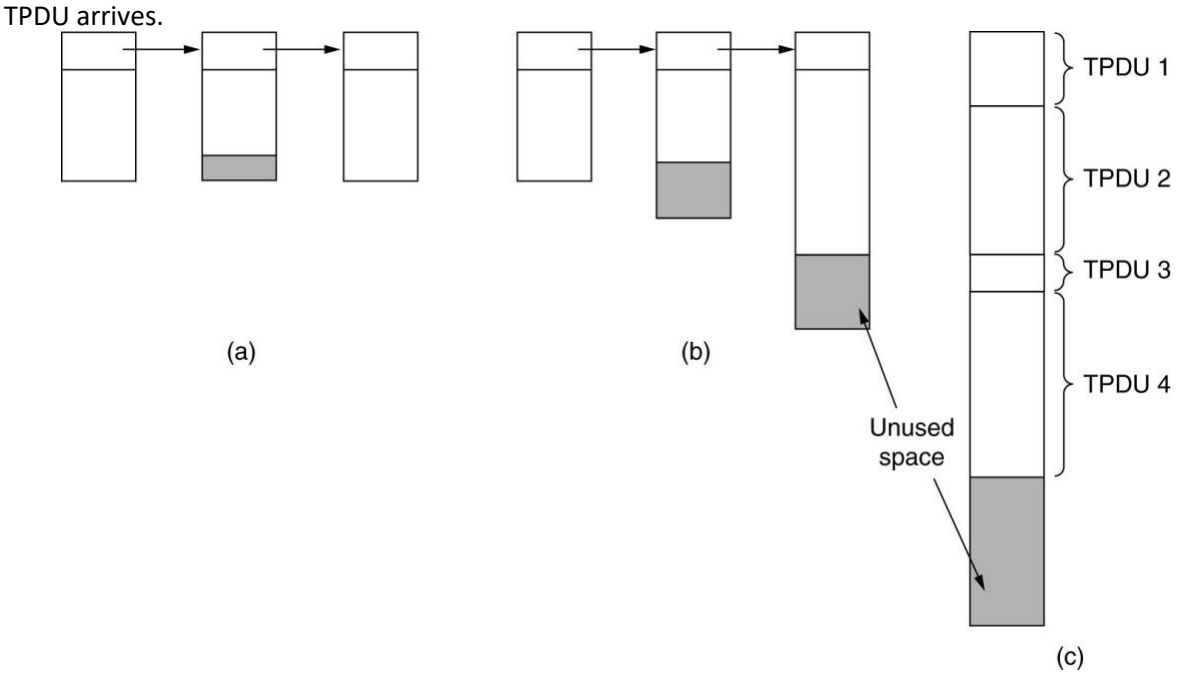


Figure: (a) Chained fixed-size buffers. (b) Chained variable-sized buffers.
(c) One large circular buffer per connection.

- If the buffer size is chosen less than the maximum TPDU size, multiple buffers will be needed for long TPDUs, with the attendant complexity.
- Another approach to the buffer size problem is to use variable-sized buffers, as in Figure (b).
- The advantage here is better memory utilization, at the price of more complicated buffer management.
- A third possibility is to dedicate a single large circular buffer per connection, as in Figure (c).
- This system also makes good use of memory, provided that all connections are heavily loaded, but is poor if some connections are lightly loaded.
- The optimum trade-off between source buffering and destination buffering depends on the type of traffic carried by the connection.

Q-3 Explain crash recovery of transport protocol.

- If hosts and routers are subject to crashes, recovery from these crashes becomes an issue.
- If the transport entity is entirely within the hosts, recovery from network and router crashes is straightforward.
- If the network layer provides datagram service, the transport entities expect lost TPDUs all the time and know how to cope with them.
- If the network layer provides connection-oriented service, then loss of a virtual circuit is handled by establishing a new one and then probing the remote transport entity to ask it which TPDUs it has received and which ones it has not received.
- In particular, it may be desirable for clients to be able to continue working when servers crash and then quickly reboot.
- No matter how the client and server are programmed, there are always situations where the protocol fails to recover properly.
- The server can be programmed in one of two ways: acknowledge first or write first.
- The client can be programmed in one of four ways: always retransmit the last TPDU, never retransmit the last TPDU, retransmit only in state S0, or retransmit only in state S1.
- This gives eight combinations, but as we shall see, for each combination there is some set of events that makes the protocol fail.
- Three events are possible at the server: sending an acknowledgement writing to the output process (W), and crashing (C).

- The three events can occur in six different orderings: AC(W), AWC, C(AW), C(WA), WAC, and WC(A), where the parentheses are used to indicate that neither A nor W can follow C (i.e., once it has crashed, it has crashed).

Strategy used by sending host	Strategy used by receiving host: First ACK, then write			First write, then ACK		
	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

Figure: Different combinations of client and server strategy.

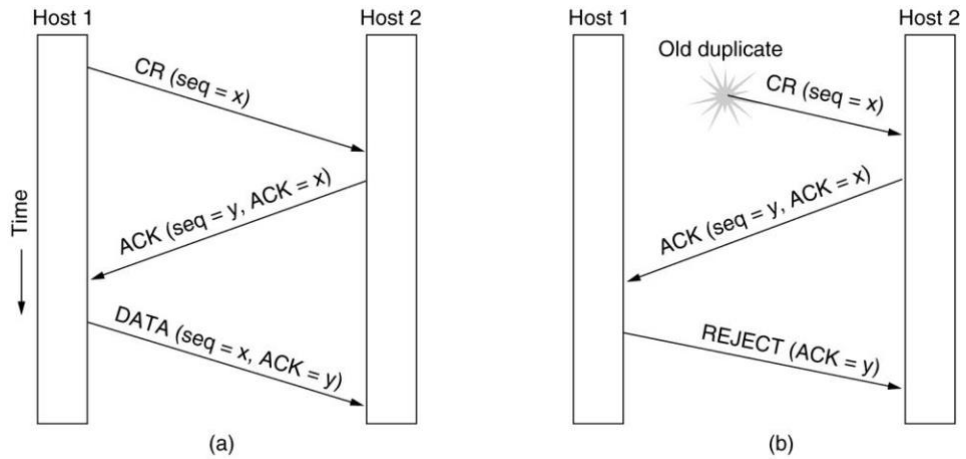
- Figure shows all eight combinations of client and server strategy and the valid event sequences for each one.
- Notice that for each strategy there is some sequence of events that causes the protocol to fail.

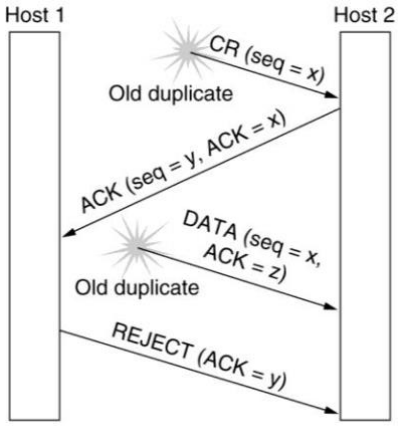
For example, if the client always retransmits, the AWC event will generate an undetected duplicate, even though the other two events work properly.

Q-4 Explain connection establishment and connection release in Transport protocols.

Connection Establishment

- Transport protocols uses three way handshake for connection establishment.
- This establishment protocol does not require both sides to begin sending with the same sequence number, so it can be used with synchronization methods other than the global clock method.
- The normal setup procedure when host 1 initiates are shown in Figure (a).
- Host 1 chooses a sequence number, x, and sends a CONNECTION REQUEST TPDU containing it to host 2. Host 2 replies with an ACK TPDU acknowledging x and announcing its own initial sequence number, y. Finally, host 1 acknowledges host 2's choice of an initial sequence number in the first data TPDU that it sends.





(c)

Figure: (a) Normal operation, (b) Old CONNECTION REQUEST appearing out of nowhere.
(c) Duplicate CONNECTION REQUEST and duplicate ACK.

- Now let us see how the three-way handshake works in the presence of delayed duplicate control TPDUs.
- In Figure (b), the first TPDU is a delayed duplicate CONNECTION REQUEST from an old connection.
- This TPDU arrives at host 2 without host 1's knowledge. Host 2 reacts to this TPDU by sending host 1 an
- ACK TPDU, 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.
- The worst case is when both a delayed CONNECTION REQUEST and an ACK are floating around in the subnet.
- This case is shown in Figure (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 TPDUs containing sequence number y or acknowledgements to y are still in existence.
- When the second delayed TPDU 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 TPDUs that can cause the protocol to fail and have a connection set up by accident when no one wants it.

Connection Release

- We see the normal case in which one of the users sends a DR (DISCONNECTION REQUEST) TPDU to initiate the connection release.
- When it arrives, the recipient sends back a DR TPDU, too, and starts a timer, just in case its DR is lost.
- When this DR arrives, the original sender sends back an ACK TPDU and releases the connection. Finally, when the ACK TPDU arrives, the receiver also releases the connection.
- Releasing a connection means that the transport entity removes the information about the connection from its table of currently open connections and signals the connection's owner (the transport user) somehow.
- This action is different from a transport user issuing a DISCONNECT primitive.
- If the final ACK TPDU is lost, as shown in Figure (b), the situation is saved by the timer.
- When the timer expires, the connection is released anyway.
- Now consider the case of the second DR being lost. The user initiating the disconnection will not receive the expected response, will time out, and will start all over again.
- In Figure (c) we see how this works, assuming that the second time no TPDUs are lost and all TPDUs are delivered correctly and on time.
- Figure (d), is the same as Figure (c) except that now we assume all the repeated attempts to

retransmit the DR also fail due to lost TPDUs.

- After N retries, the sender just gives up and releases the connection. Meanwhile, the receiver times out and also exits.
- While this protocol usually suffices, in theory it can fail if the initial DR and N retransmissions are all lost.
- The sender will give up and release the connection, while the other side knows nothing at all about the attempts to disconnect and is still fully active.
- This situation results in a half-open connection.
- We could have avoided this problem by not allowing the sender to give up after N retries but forcing it to go on forever until it gets a response.
- However, if the other side is allowed to time out, then the sender will indeed go on forever, because no response will ever be forthcoming. If we do not allow the receiving side to time out, then the protocol hangs in Figure (d).

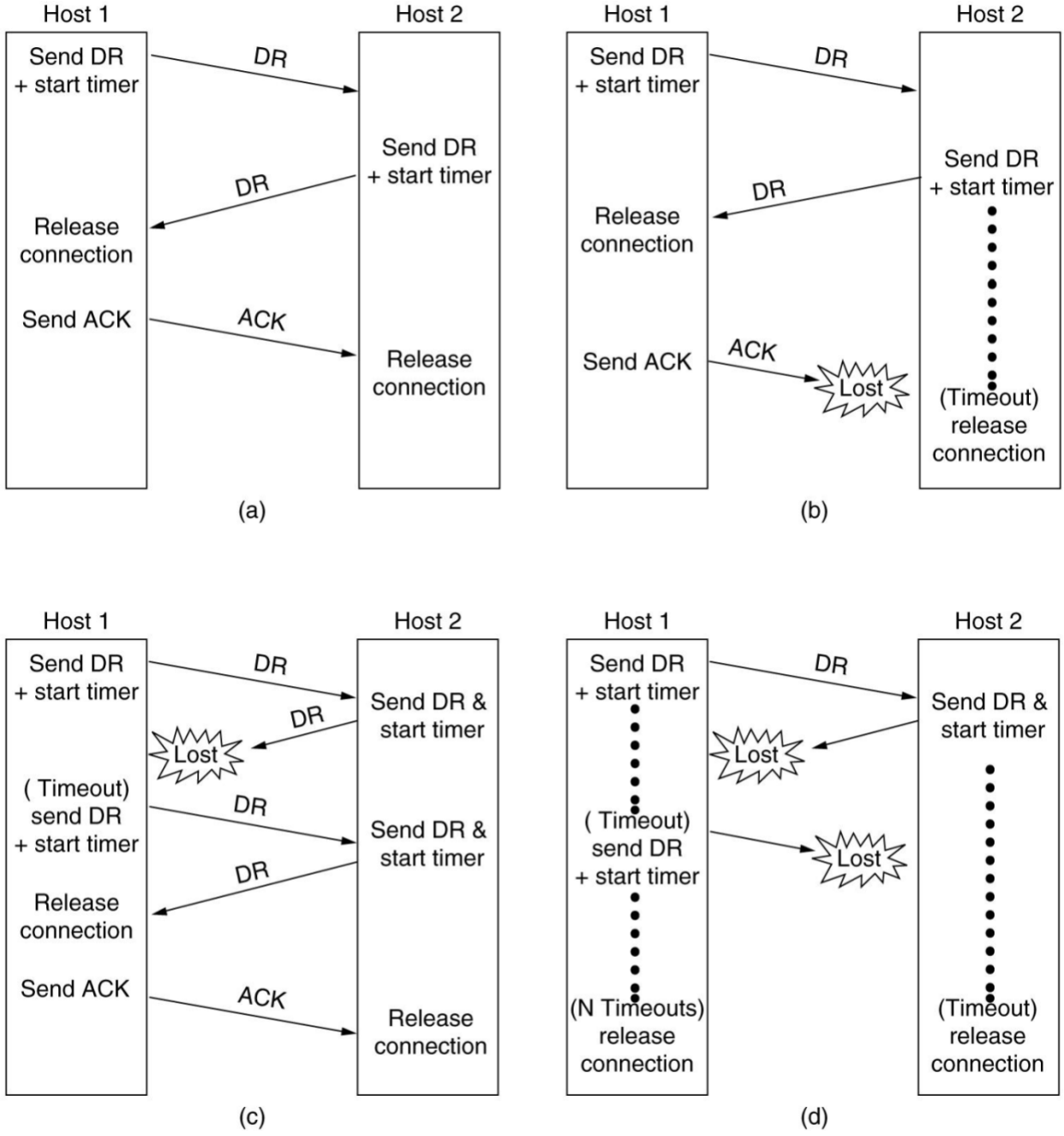


Figure: Four protocol scenarios for releasing a connection. (a) Normal case of a three-way handshake. (b) final ACK lost.(c) Response lost. (d) Response lost and subsequent DRs lost.

Q-5 Discuss the working principle of TCP.

- TCP (Transmission Control Protocol) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork.
- An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters.
- TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.
- TCP service is obtained by both the sender and receiver creating end points, called sockets.
- Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a port.
- A port is the TCP name for a TSAP.
- For TCP service to be obtained, a connection must be explicitly established between a socket on the sending machine and a socket on the receiving machine.
- The basic protocol used by TCP entities is the sliding window protocol.
- When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exist, otherwise without data) bearing an acknowledgement number equal to the next sequence number it expects to receive.
- If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again.

TCP Connection Establishment

- To establish a connection, one side, say, the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives, either specifying a specific source or nobody in particular.
- The other side, say, the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password).
- The CONNECT primitive sends a TCP segment with the SYN bit on and ACK bit off and waits for a response.
- When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the Destination port field.
- If not, it sends a reply with the RST bit on to reject the connection.
- If some process is listening to the port, that process is given the incoming TCP segment.
- It can then either accept or reject the connection.

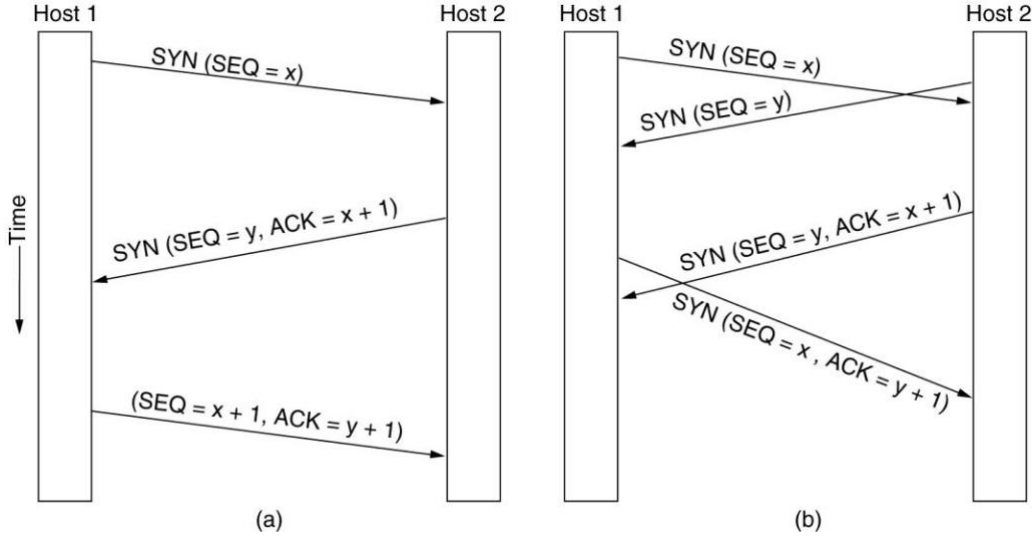


Figure: (a) TCP connection establishment in the normal case.
(b) Call collision.

- If it accepts, an acknowledgement segment is sent back. The sequence of TCP segments sent in the

normal case is shown in Figure (a).

- Note that a SYN segment consumes 1 byte of sequence space so that it can be acknowledged unambiguously.
- In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the sequence of events is as illustrated in Figure (b).
- The result of these events is that just one connection is established, not two because connections are identified by their end points.

TCP Connection Release

- TCP connections are full duplex.
- Each simplex connection is released independently of its sibling.
- To release a connection, either party can send a TCP segment with the FIN bit set, which means that it has no more data to transmit.
- When the FIN is acknowledged, that direction is shut down for new data.
- Data may continue to flow indefinitely in the other direction, however.
- When both directions have been shut down, the connection is released. Normally, four TCP segments are needed to release a connection, one FIN and one ACK for each direction.
- However, it is possible for the first ACK and the second FIN to be contained in the same segment, reducing the total count to three.
- To avoid the two-army problem, timers are used.
- If a response to a FIN is not forthcoming within two maximum packet lifetimes, the sender of the FIN releases the connection.
- The other side will eventually notice that nobody seems to be listening to it anymore and will time out as well.
- While this solution is not perfect, given the fact that a perfect solution is theoretically impossible, it will have to do. In practice, problems rarely arise.

Q-6 Explain TCP header fields.

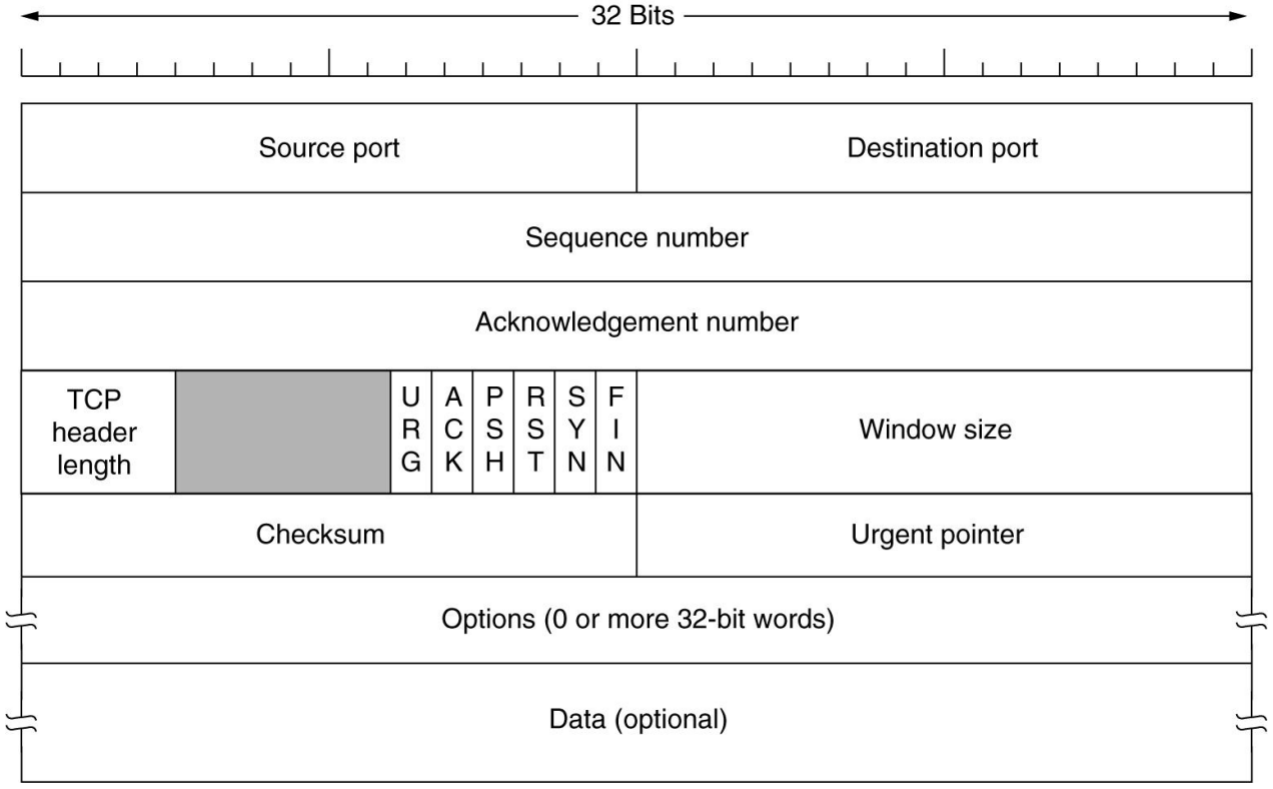


Figure: TCP Header

- The **Source port** and **Destination port** fields identify the local end points of the connection.
- A port plus its host's IP address forms a 48-bit unique end point. The source and destination end points together identify the connection.
- The **Sequence number** and **Acknowledgement number** fields perform their usual functions.
- Both are 32 bits long because every byte of data is numbered in a TCP stream.
- The **TCP header length** tells how many 32-bit words are contained in the TCP header.
- This field really indicates the start of the data within the segment, measured in 32-bit words, but that number is just the header length in words, so the effect is the same.
- Next comes a 6-bit field that is not used.
- Now come six 1-bit flags. **URG** is set to 1 if the Urgent pointer is in use.
- The **Urgent pointer** is used to indicate a byte offset from the current sequence number at which urgent data are to be found.
- The **ACK** bit is set to 1 to indicate that the Acknowledgement number is valid.
- If ACK is 0, the segment does not contain an acknowledgement so the Acknowledgement number field is ignored.
- The **PSH** bit indicates pushed data.
- The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency).
- The **RST** bit is used to reset a connection that has become confused due to a host crash or some other reason.
- It is also used to reject an invalid segment or refuse an attempt to open a connection.
- The **SYN** bit is used to establish connections.
- The connection request has SYN = 1 and ACK = 0 to indicate that the piggyback acknowledgement field is not in use.
- The connection reply does bear an acknowledgement, so it has SYN = 1 and ACK = 1. In essence the SYN bit is used to denote CONNECTION REQUEST and CONNECTION ACCEPTED, with the ACK bit used to distinguish between those two possibilities.
- The **FIN** bit is used to release a connection. It specifies that the sender has no more data to transmit. However, after closing a connection, the closing process may continue to receive data indefinitely.
- Both SYN and FIN segments have sequence numbers and are thus guaranteed to be processed in the correct order.
- The **Window size** field tells how many bytes may be sent starting at the byte acknowledged.
- A Window size field of 0 is legal and says that the bytes up to and including Acknowledgement number - 1 have been received, but that the receiver is currently badly in need of a rest and would like no more data for the moment, thank you.
- A **Checksum** is also provided for extra reliability.
- **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
- The **Options** field provides a way to add extra facilities not covered by the regular header.
- The most important option is the one that allows each host to specify the maximum TCP payload it is willing to accept.
- Using large segments is more efficient than using small ones because the 20-byte header can then be amortized over more data, but small hosts may not be able to handle big segments.

Q-7 Discuss the working principle of UDP.

- The Internet protocol suite supports a connectionless transport protocol, UDP (User Datagram Protocol). UDP provides a way for applications to send encapsulated IP datagrams and send them without having to establish a connection.
- UDP transmits segments consisting of an 8-byte header followed by the payload. The header is shown in Figure.

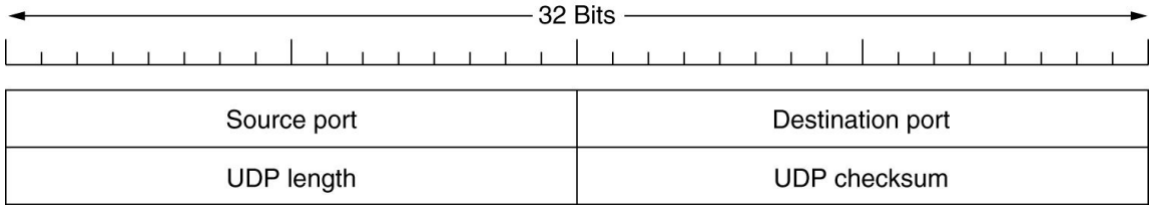


Figure: The UDP header.

- The two ports serve to identify the end points within the source and destination machines.
- When a UDP packet arrives, its payload is handed to the process attached to the destination port. This attachment occurs when BIND primitive or something similar is used, as we know in for TCP (the binding process is the same for UDP).
- In fact, the main value of having UDP over just using raw IP is the addition of the source and destination ports.
- Without the port fields, the transport layer would not know what to do with the packet. With them, it delivers segments correctly.
- The source port is primarily needed when a reply must be sent back to the source.
- By copying the source port field from the incoming segment into the destination port field of the outgoing segment, the process sending the reply can specify which process on the sending machine is to get it.
- The UDP length field includes the 8-byte header and the data.
- The UDP checksum is optional and stored as 0 if not computed (a true computed 0 is stored as all 1s).
- Turning it off is foolish unless the quality of the data does not matter (e.g., digitized speech).
- It is probably worth mentioning explicitly some of the things that UDP does not do.
- It does not do flow control, error control, or retransmission upon receipt of a bad segment. All of that is up to the user processes.
- What it does do is provide an interface to the IP protocol with the added feature of de multiplexing multiple processes using the ports.
- That is all it does. For applications that need to have precise control over the packet flow, error control, or timing, UDP provides just what the doctor ordered.
- One area where UDP is especially useful is in client-server situations.
- Often, the client sends a short request to the server and expects a short reply back.
- If either the request or reply is lost, the client can just time out and try again.

Q-8 Explain how congestion control is achieved in TCP?

- When a connection is established, a suitable window size has to be chosen.
- The receiver can specify a window based on its buffer size.
- If the sender sticks to this window size, problems will not occur due to buffer overflow at the receiving end, but they may still occur due to internal congestion within the network.
- In Figure (a), we see a thick pipe leading to a small-capacity receiver.
- As long as the sender does not send more water than the bucket can contain, no water will be lost.
- In Figure (b), the limiting factor is not the bucket capacity, but the internal carrying capacity of the network.

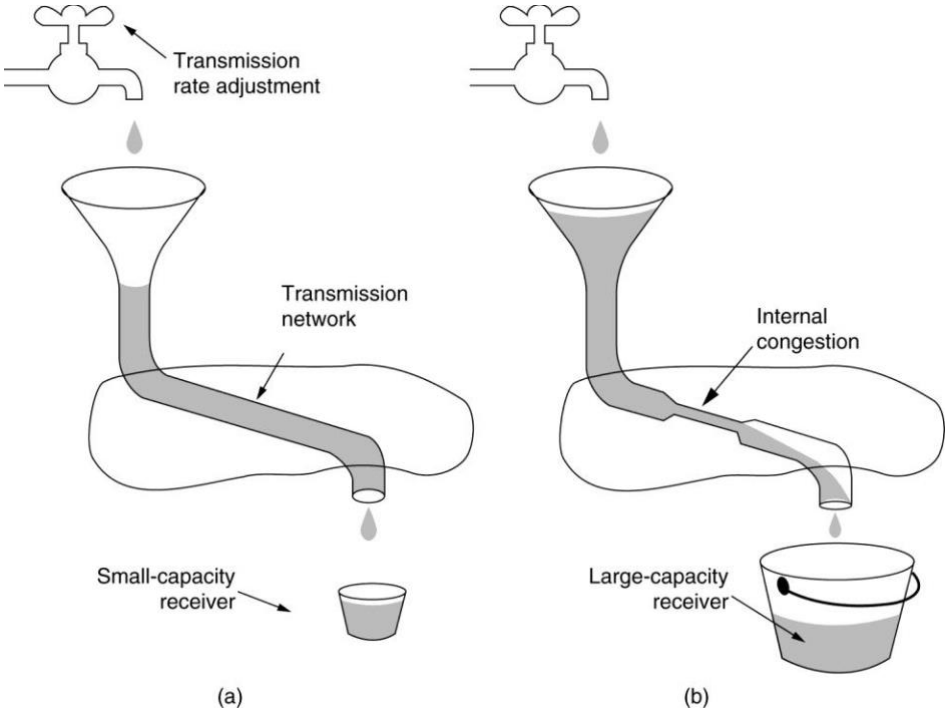


Figure: (a) A fast network feeding a low capacity receiver.
(b) A slow network feeding a high-capacity receiver.

- If too much water comes in too fast, it will back up and some will be lost (in this case by overflowing the funnel).
- Each sender maintains two windows: the window the receiver has granted and a second window, the congestion window.
- Each reflects the number of bytes the sender may transmit.
- The number of bytes that may be sent is the minimum of the two windows.
- Thus, the effective window is the minimum of what the sender thinks is all right and what the receiver thinks is all right.
- When a connection is established, the sender initializes the congestion window to the size of the maximum segment in use on the connection.
- It then sends one maximum segment.
- If this segment is acknowledged before the timer goes off, it adds one segment's worth of bytes to the congestion window to make it two maximum size segments and sends two segments.
- As each of these segments is acknowledged, the congestion window is increased by one maximum segment size.
- When the congestion window is n segments, if all n are acknowledged on time, the congestion window is increased by the byte count corresponding to n segments.
- In effect, each burst acknowledged doubles the congestion window.
- The congestion window keeps growing exponentially until either a timeout occurs or the receiver's window is reached.
- Now let us look at the Internet congestion control algorithm. I
- t uses a third parameter, the threshold, initially 64 KB, in addition to the receiver and congestion windows.
- When a timeout occurs, the threshold is set to half of the current congestion window, and the congestion window is reset to one maximum segment.
- Slow start is then used to determine what the network can handle, except that exponential growth stops when the threshold is hit.
- From that point on, successful transmissions grow the congestion window linearly (by one maximum segment for each burst) instead of one per segment.

Q-9 Compare UDP and TCP.

Parameter	TCP	UDP
Name	Transmission Control Protocol	User Datagram Protocol or Universal Datagram Protocol
Connection	TCP is a connection-oriented protocol.	UDP is a connectionless protocol.
Usage	TCP is suited for applications that require high reliability, and transmission time is relatively less critical.	UDP is suitable for applications that need fast, efficient transmission, such as games. UDP's stateless nature is also useful for servers that answer small queries from huge numbers of clients.
Use by other protocols	HTTP, HTTPs, FTP, SMTP, Telnet	DNS, DHCP, TFTP, SNMP, RIP, VOIP.
Ordering of data packets	TCP rearranges data packets in the order specified.	UDP has no inherent order as all packets are independent of each other. If ordering is required, it has to be managed by the application layer.
Speed of transfer	The speed for TCP is slower than UDP.	UDP is faster because there is no error-checking for packets.
Header Size	TCP header size is 20 bytes	UDP Header size is 8 bytes.
Common Header Fields	Source port, Destination port, Check Sum	Source port, Destination port, Check Sum
Error Checking	TCP does error checking	UDP does error checking, but no recovery options.
Data Flow Control	TCP does Flow Control. TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control.	UDP does not have an option for flow control
Reliability	There is absolute guarantee that the data transferred remains intact and arrives in the same order in which it was sent.	There is no guarantee that the messages or packets sent would reach at all.