UNIT - 5

The Internet Transport Protocols:

⇒ Internet has 2 main protocols: UDP (user datagram protocol) ⇒ connectionless.
TCP (trans control protocol) ⇒ connection oriented.

⇒ The protocols complement each other.

Introduction to UDP:

⇒ simple protocol.

⇒ light-weight protocol.

⇒ offers min functionality

⇒ focuses on fast delivery of data with little overhead.

⇒ Connectionless protocol: UDP does not establish a connection b/w the sender & receiver b4 data trans. It sends packets known as datagrams from one application to another.

⇒ No reliability: UDP does not gaurantee delivery, order or error checking. Once a packet is sent, there is no ack (confirmation of receipt).

⇒ minimal overhead: UDP does not manage flow control, congst control, retransmissions. This makes it very eff^n for apps where speed is most imp than reliability.

Eg: online gaming.
DNS (domain name sys) queries.
Video streaming.

UDP Segment Structure:

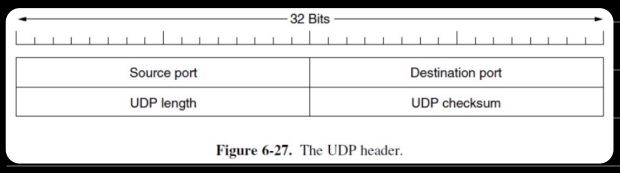


Figure 6-27. The UDP header.

⇒ Each UDP segment consists 8-byte header followed by the data payload.

⇒ it has a simple header, consisting the fields:

* Source Port: Identifies the sending application's port/ sending machine's port. 16-bit field, usually used when a reply needs to be sent back to the sender.
* Dest Port: Identifies the receiving application's port. 16-bit field, used to indicate where the data has to be delivered.
* Length: tot length of UDP segment, including the header and data. max value of this field is 65,515 bytes, slightly lower than the IP protocol's max packet size.
* Checksum: optional field for error-checking to ensure data integrity. 16-bit checksum, if checksum is computed, it is set to 1. If not, it is set to 0.

⇒ the header allows UDP to deliver data to the correct process on the receiving machine.

⇒ The port numbers in the header are crucial in identifying the specific application that should handle the data, as the transport layer uses these fields to direct the datagrams.

⇒ UDP pseudoheader is used in the checksum calculation to ensure the integrity of data. It includes:

* source and dest IP address.
* length of UDP segment.
* Protocol no. (17 for UDP)

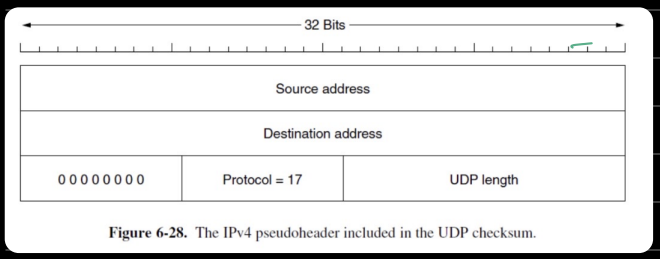


Figure 6-28. The IPv4 pseudoheader included in the UDP checksum.

⇒ Contains 32-bit IP add of source & dest machines, protocol no. for UDP i.e 17, byte count for UDP (including header).

⇒ If the same in IPv6, including the pseudoheader helps to detect misdelivered packets but also including it violates the protocol hierarchy since the IP addresses in it belong to the IP layer, not to the UDP layer.

⇒ TCP uses the same P.H for its checksum.

⇒ What does UDP not do?

* no flow control: does not prevent overwhelming the receiver, no flow regulation.
* no congestion control: does not manage network congestion, leading to packet loss during overload.
* no retransmissions: if a segment is lost/corrupted, UDP does not retransmit it. Handling lost data is the application's responsibility in UDP.

Introduction to TCP: (main workhorse of internet)

⇒ ensures data is delivered correctly in the right order.

⇒ conn oriented protocol: TCP establishes a connection b/w the sender & receiver b4 data trans.

⇒ reliability: TCP guarantees delivery, without errors, in correct sequence. If a packet is lost / corrupted, TCP retransmits it until it is correctly received, ack is provided by receiver.

⇒ provides flow control to prevent overwhelming the receiver with too much data.

⇒ provides congestion control: TCP can adjust the rate of trans for smooth flow.

Ex: web browsing (HTTP/HTTPS)
email (SMTP)
file Transfers (FTP)

Feature	UDP	TCP
Connection Type	Connectionless	Connection-Oriented
Reliability	No guarantees on delivery	Guarantees reliable delivery
Order of Packets	Packets may arrive out of order	Ensures packets are in correct order
Retransmission	No retransmission	Retransmits lost/corrupted packets
Flow Control	No flow control	Implements flow control
Congestion Control	No congestion control	Uses congestion control
Overhead	Minimal	Higher due to extra features
Use Cases	Real-time applications (e.g., streaming, gaming)	Reliable data transmission (e.g., web browsing, email)

working

⇒ each machine that uses TCP has a TCP transport entity, which is often part of the kernel.

⇒ TCP entity manages the TCP streams & interacts with the IP layer.

⇒ When an app wants to send data:

* Segmentation: TCP accepts a stream of data from an app and breaks it into pieces called segments.
  each seg is sent as a separate IP datagram.
  the size of each seg can be upto 64 KB.
* Establish conn: TCP establishes a connec b4 trans.
  this ensures both sender & receiver are ready to exchange data.

* Sending data: TCP begins transmitting the segments after establishing the connec.
  each seg includes seq no., allowing the receiver to reassemble the segments in right order.
* Ack: When receiver gets the seg, it sends an ack back to sender.
  if sender does not get ack, it assumes seg was lost and retransmits it.
* flow & conges control: TCP uses flow control to ensure sender does not overwhelm receiver.
  conges control mechanisms ensure that the network is not overloaded (by adjusting the trans speed
* Reassembling data: When TCP segments arrive at the receiver, the TCP entity at the receiving end reassembles the original byte stream by putting the segments back in order.

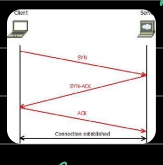
⇒ role of TCP in network reliability:
* Seq nos: ensures segments are processed in correct order.
* timeout & retrans: when segments are lost/corrupted, TCP retransmits them.
* flow control: TCP adjusts the rate at which data is sent, so that receiver can handle it.
* conges control: TCP dynamically adapts to network conditions to prevent overload & loss of packets.

The TCP Service Model:

⇒ TCP offers a reliable, connec oriented service model
⇒ Both sender & receiver must create end-points called sockets. These are necessary for estab communication.
⇒ A socket is identified with a socket no. consisting of an IP add and 16-bit port no which is unique to every machine. (identifies spec process on machine) (identifies machine on which process runs)
⇒ Each TCP connec is estab b/w 2 sockets: one on source & one on dest.

Ex: Socket: 192.168.1.10:8080
(192.168.1.10 = IP add; 8080 = P.No.)

⇒ Port is also TSAP: Transport Service Access point.
⇒ The port no. is categorized as follows:
* well known ports ⇒ 0-1023: reserved for common services like HTTP (port 80), FTP (port 21), IMAP (port 143).
* registered ports: 1024-49151: can be registered by applications to avoid conflicts. These ports are available for general use.
* dynamic or private ports: 49152-65535: Typically used temporarily for client-side comm
⇒ Connec is explicitly established b/w two sockets on diff machines before any data can be transferred. This involves a three way handshake, which sets up the comm channel by exchanging messages b/w the two parties.



① SYN ⇒ Synchronize
② SYN-ACK ⇒ Sync-Acknowledge
③ ACK ⇒ Acknowledge

⇒ once estab, data can be sent & received. A single socket can handle multiple connections at once. But each connec is identified by 2 socket identifiers (source & dest). Ex: (socket 1, socket 2)

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Figure 6-34. Some assigned ports.

⇒ All TCP connections are full-duplex and point-to-point.

⇒ Full duplex: * data can flow in both directions simultaneously.
* A client and server can send data at same time without waiting for the other side finish.

⇒ Point-To-Point: Involve exactly 2 end points.
TCP does not support Multicasting and Broadcasting.

⇒ Byte-stream Comm: * TCP views data as a continuous byte-stream, rather than individual messages.
* This means that the boundaries b/w data chunks written by sender are not preserved when received.
* Ex: if sender writes (four) 512-byte segments, they might be received as:
→ four 512-byte chunks.
→ two 1024-byte chunks.
→ one 2048 byte chunk etc.

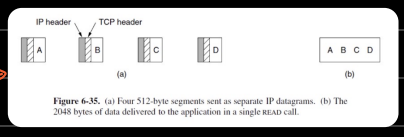


Figure 6-35. (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

* There is no indication of how the data was divided in the transmission, and the receiving app^ln processes the data as a continuous stream of bytes, just like reading from a file.

⇒ Delayed Trans and the Push Flag:
* TCP tries to optimize performance by buffering data b4 sending it, which allows it to send larger chunks, reducing overhead
* However, in some imp app^lns like gaming, it is imp to send data immediately. To handle such cases, TCP supports the notion of a PUSH flag.
* The push flag instructs the protocol to send any buffered data right away.
* App^lns cannot directly set the push flag, but diff O.Sys provide options like TCP_NODELAY (windows, linux) to force TCP to send data without delay.

⇒ Urgent data and urgent flag:
* TCP includes a mechanism for handling urgent data, the data that needs to be processed immediately by the receiving app^ln.
* The urgent flag marks data that req^s immediate attention.
* When the flag is set, TCP stops waiting for more data & sends everything it has.
* Once the urgent data arrives, the receiving process is interrupted to handle it.
* The end of urgent data is marked, but the start is not, so it is up to the receiving app^ln to figure out where the urgent data begins
* It is rarely used today due to its complexity & implementation differences b/w systems
* modern app^lns manage their own signalling mechanisms.

⇒ Well-known ports & Daemons:
* Many services on network have preassigned well known ports (0–1023) for com^mn. Ex:
→ Port 80: HTTP web traffic.
→ Port 143: IMAP email retrieval.

* Daemons (background processes that handle reqs) are often associated with spec ports
ex: inetd daemon in UNIX systems listens on multiple ports and forks a new process when a
connec req is received. It then assigns the appropriate daemon to handle the
req. This allows sys resources to be used efficiently, only activating daemons
when necessary.

The TCP Protocol:

TCP Protocol:

⇒ Each byte in a connec is assigned a 32-bit sequence no. which is crucial for tracking data trans. TCP exchanges data in segments each of which includes:
* fixed 20-byte header (with optional additional field)
* zero or more bytes data
⇒ TCP determines the size of segments based on data being trans.
⇒ It can neither combine data from multiple writes into one segment or split data from a single write across multiple segments.
⇒ There are 2 factors that limit the seg size:
* IP-Payload Size: Tot size of seg (including header) must fit within 65,535 byte limit set by IP.
* Max Transfer Unit (MTU): each network link has a specific MTU. Typically 1500 bytes for Ethernet.
TCP segments must fit within this size to avoid fragmentation.
⇒ If the MTU on any network path is smaller, it may cause IP fragmentation which can degrade perfo. To prev this, modern TCP implementations use Path MTU discovery, adjusting the seg size to avoid frag by finding the smallest MTU along the path.
⇒ Sliding Window Protocol:
* TCP uses sliding window protocol with a dynamic window size to manage data flow.
* When a seg is sent, a timer is started.
* The receiver sends an ack containing:
→ the next expected seq no
→ the remaining window size (amt of data it can receive)
* If the sender doesn't receive ACK b4 timer expires, it retransmits the segment.
⇒ Handling network challenges:
* TCP can handle various challenges such as:
→ Out of order segments: ex, bytes 3072-4095 may arrive b4 bytes 2048-3071, but TCP waits until all missing bytes are received b4 ack.
→ Delayed segments: if a seg is delayed, and sender times out, it may retrans the data with diff byte ranges.
TCP keeps track of which bytes have been correctly received.
* This protocol ensures reliable data trans, even when network conditions introduce delays, fragmentation, or out-of-order packets.

The TCP Segment Header:

⇒ A TCP seg starts with a 20-byte fixed header & optional fields followed by upto 65,495 data bytes.

⇒ Segments without data are valid and used for ack or control messages

⇒ layout:

* Source & Dest Port: * endpoints of a conn^{n} are where the TCP port and an IP add form a unique 48-bit endpoint.
  * Comb^{n} of source & dest addresses and ports creates the 5-tuple that uniquely identifies a conn^{n}.
* Seq no.: identifies the position of the first byte in the seg.
  Every byte in the TCP bytestream has a unique seq^{n} no.
* ACK no.: indicates next byte the receiver expects.
* Header length: Specifies length of TCP header in 32-bit words. It accounts for the optional fields, which makes the header variable in size.

⇒ Flags in TCP header:

* CWR and ECE: * used in explicit congestion notification (ECN) to manage network congestion.
  * ECE informs the sender of congestion.
  * CWR confirms the sender has reduced trans^{n} rate.
* URG: indicates urgent data within seg.
* ACK: confirms that the ack no. is valid.
* PSH: request the receiver to process data immediately rather than buffering it.
* RST: abruptly resets the conn^{n} due to an error or invalid segment.
* SYN: initiates a conn^{n}, with diff uses depending on the ACK flag (conn^{n} req or reply).
* FIN: closes the conn^{n}, indicating no more data will be sent.

⇒ Window size and flow control:

* Window Size: specifies how much data can be sent after the acknowledged data. A zero window size pauses data trans^{n} temporarily.
* Checksum: verifies the integrity of the header and data
* Options: allow for additional functionality, such as specifying max^{m} seg size (MSS). Options are encoded with Type-length-value fields and padded to a multiple of 32-bit.

⇒ Connection Termination: * TCP uses FIN segments to close connections.

* Each direction of comm^{n} is terminated independently, requiring 4 segments (two FIN, two ACK).
* Simultaneous termination can reduce this to 3 segments.
* Timers prevent indefinite waiting during termin^{n} closing the conn^{n} if no response is received.

⇒ This layout ensures reliable trans^{n} & effic^{t} flow control in the TCP protocol.

⇒ On a fast conn^{n}, the seq no. may wraparound quickly, leading to confusion b/w old and new data. So;

⇒ PAWS: 'Protection against Wrapped Sequence nos' is a scheme; discards arriving segments with old timestamps to prevent this problem.

⇒ SACK: 'Selective ACKnowledgment' option lets a receiver tell a sender the ranges of seq nos that it has received. So that the sender is aware of what data the receiver has and hence can determine what data should be retransmitted.

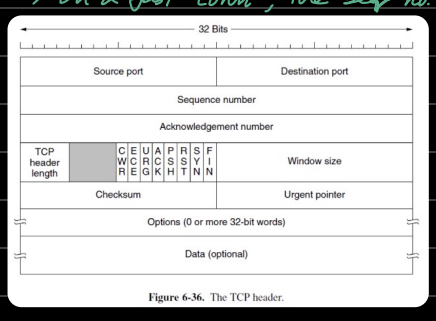


Figure 6-36. The TCP header.

TCP Connection Establishment:

⇒ TCP uses a three-way handshake to establish connections.

* Server * Passively waits for connections by using LISTEN and ACCEPT.
  * The server specifies a port it will listen on.
* Client (CONNECT): * Sends a CONNECT req with SYN bit set and ACK bit off, indicating
  * the device to connect.
* Server Response * If a process is listening on the requested port, it sends a TCP seg with both the SYN and ACK bits set, indicating that the connᶜ is accepted & acknowledging the client's SYN. The ACK no. sent refers to the client's initial seq no. + 1, completing the handshake.
  * if no process is listening, it sends a seg with the RST (reset) bit set, rejecting the connᶜ.
* Client Ack: * Upon receiving the server's SYN+ACK segment, the client sends an ACK segment back, confirming the server's seq no.
  * The data can be transferred again.

⇒ Simultaneous Connᶜ Requests: Both hosts may try to connect to each other simultaneously. In simultaneous connᶜ attempts, only 1 connᵈ is established, as connᵉᶜˢ are uniquely identified by their endpoints (IP & PORT).

⇒ Initial seq no.: * The initial seq no should vary to avoid delayed duplicate packets.
  * not always 0.
  * Historically, this was done using a clock-based scheme where the seq no. ticked every 4 microseconds.

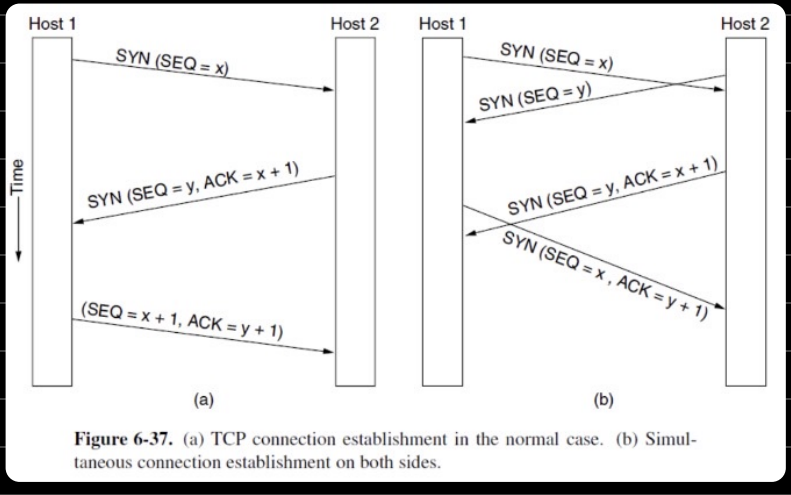


Figure 6-37. (a) TCP connection establishment in the normal case. (b) Simultaneous connection establishment on both sides.

⇒ SYN Flood Attack & Defense Mechanism (SYN Cookies):

* The 3-way handshake has a vulnerability known as the SYN-flood attack.
* In this attack, a malacious user sends a large no. of SYN segments to the server but never completes the handshake, causing the server to allocate resources (memory, connᵉᶜ table entries) unnecessarily. This can lead to a denial of service (DOS), as the server's resources are exhausted.
* SYN cookies are a technique used to defend against SYN floods.
* Instead of the server remembering the sequence no. when it sends SYN-ACK, it generates a cryptographically calculated seq no. (a SYN cookie) & sends it to the client

* The server does not store this seq no. When the client responds with an ack, the server recalculates the original seq no by reapplying the cryptographic func. If the calculated no matches, the connec is considered valid, and the handshake is completed.
* SYN cookies do have limitations, such as not supporting TCP options, so they are generally only used during a SYN flood attack.

⇒ Connec Termination:

* While TCP connections are full-duplex (allowing data to flow in both direcs simultaneously), each direc can be terminated independently. This process is often called 4-way handshake.
* FIN segment: * when a host (either client / server) wants to terminate, it sends a FIN segment.
  * indicates host has no more data to send.
  * consumes 1 byte of the sequence space.
* ACK segment: * other host acks the FIN by sending an ACK segment.
  * this confirms that the 1st host's data has been fin, but second host may continue sending data.
* Termination of other direc: data can continue to flow in the opp direc until that host also sends a FIN segment, which is then ackd by the 1st host.
* Basically 4 segments are exchanged (FIN & ACK for each direc) which is why the name ⇒ 4-way handshake arises for termination. However it is possible to combine the ACK of the 1st FIN and second FIN into one segment, reducing the tot no of segments to 3 aka Simultaneous SYN segments, both hosts may decide to send FIN segments at the same time, FIN segments are ackd as usual & connec is closed.

⇒ Timers and the two-army problem:

* After a FIN is sent, if no response is received (i.e. no ACK segment comes back), the sender of the FIN will eventually time-out & close the conn.
* The two-army prob refers to the theoretical impossibility of ensuring a perfect simultaneous connec termination but TCP handles this by using timers and retrans to ensure the connec is closed gracefully.
* The issue of connec closure failures is rare due to these timers

---

TCP Connection release:

⇒ The process of establishing and releasing a TCP connec is managed through a FSM: finite state machine, consisting of 11 states.

⇒ each state represents a diff phase of the connection's lifecycle, and certain events.

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

11 states with their funcs.

Figure 6-38. The states used in the TCP connection management finite state machine.

* LAST ACK: Sent a FIN after receiving one, waiting for an ACK.
* CLOSING: Both sides sent FINs simultaneously, waiting for an ACK.

⇒ TCP connections transition through these states based on the actions of the client & server during connᶜ estab and release.

⇒ events triggering transitions:
* Sys calls (actions initiated by user ex: connect, close, listen).
* Segment arrivals (TCP control statements like SYN, FIN, ACK, RST)
* Timeouts (ex: twice the maxᵐ packet lifetime to handle exceptions & packet loss).

⇒ client perspective:
* The client starts in a closed state. To initiate a connᶜᶜ, it issues a CONNECT request, creating a connᶜ record in TCP and moving to the SYN SENT state.
* In SYN sent state, the client sends a SYN seg to the server, asking to estab a connᵉᶜ.
* TCP waits for response from the server.
* When client receives SYN + ACK seg from server, it sends an ACK seg back to the server, completing the 3-way handshake.
* The connᶜ is now fully estabᵈ, and TCP moves into the ESTABLISHED state.
* In the ESTABLISHED state, data can now be exchanged b/w the client & server. Both dirᵉˢ of data flow are active.
* When the client finishes trans its data, it calls the CLOSE primitive. This sends a FIN segment to the server and moves TCP into the FIN WAIT 1 state.
* TCP waits for the server to ack the FIN.

* After the server acks the client's FIN, the client moves to the FIN WAIT 2 state. Now the client moves to the FIN WAIT 2 state. Now the client waits for the server to send its own FIN to close the other dirᵉᶜ of the connᵉ.
* The server sends a FIN to the client, closing its end of the connᵉ. The client acks the server's FIN, and now both dirᵉˢ are closed.
* TCP waits in the TIME WAIT state for a period (twice the max packet lifetime) to ensure no delayed or duplicate segments arrive from the connᵉ
* After TIME WAIT timer expires, TCP deletes the connᵉ record and the client returns to CLOSED state

⇒ Server's perspective:
* The server starts in the LISTEN state, waiting for an incoming connᵉ.
* When server receives a SYN from the client, it sends back a SYN+ACK seg & moves to the SYN RCVD state, waiting for the client's final ACK.
* When the server receives the final ACK from client, the 3 way handshake is complete and server moves to ESTABLISHED state. Data transfer can begin in both dirᵉˢ
* When the client finˢ trans data & sends a FIN seg, the server acks it & moves to the CLOSE WAIT state. It waits for the appˡⁿ to issue a CLOSE call.
* After the appˡⁿ closes the connᵉ, the server sends a FIN seg to the client, moving to the LAST ACK state.
* When server receives the client's ACK for its FIN, it deletes the connᵉ record & returns to CLOSED state.

⇒ Summary:

1) Client: CLOSED → SYN SENT → ESTABLISHED →
CLOSED ← TIME WAIT ← FIN WAIT 2 ← FIN WAIT 1

2) Server: LISTEN → SYN RCVD → ESTABLISHED
↓
CLOSED ← LAST ACK ← CLOSE WAIT

⇒ In some cases, both sides may send FIN at the same time. TCP enters CLOSING state on both sides. The connection will close after both sides receive ack for their FINs, moving to the TIME WAIT state

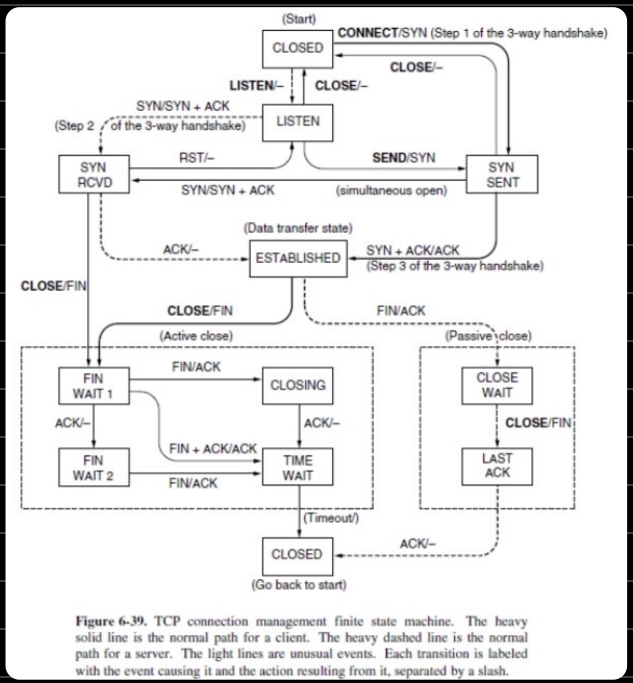
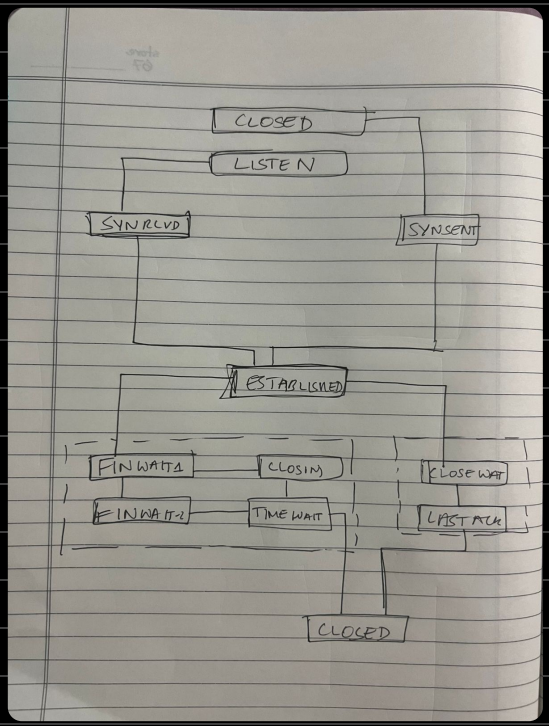


Figure 6-39. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled with the event causing it and the action resulting from it, separated by a slash.

Combined TCP State Transitions for Client and Server:

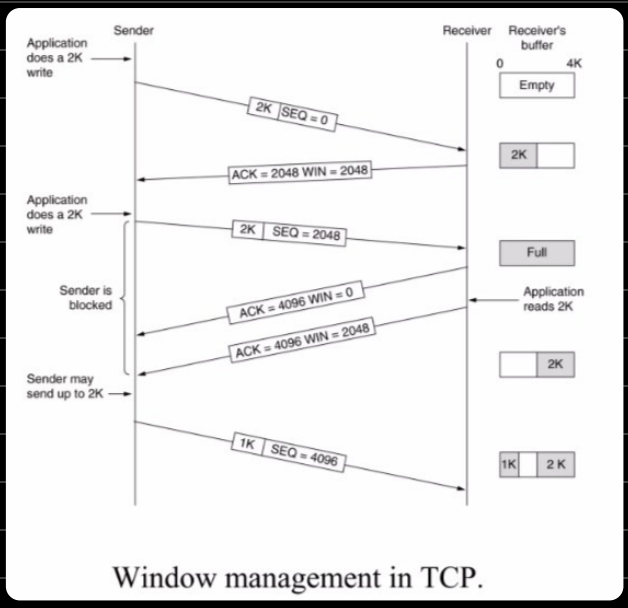
1. **Initial State (Before Connection):**
   - **Client:** CLOSED
   - **Server:** LISTEN
2. **Connection Establishment (Three-Way Handshake):**
   - **Client** sends a **SYN** segment and moves to **SYN SENT**.
     - **Client:** CLOSED → SYN SENT
   - **Server** receives the SYN, sends a **SYN + ACK**, and moves to **SYN RCVD**.
     - **Server:** LISTEN → SYN RCVD
   - **Client** receives the **SYN + ACK**, sends an **ACK** and moves to **ESTABLISHED**.
     - **Client:** SYN SENT → ESTABLISHED
   - **Server** receives the **ACK** and moves to **ESTABLISHED**.
     - **Server:** SYN RCVD → ESTABLISHED
3. **Data Transmission (Connection Open):**
   - **Client** and **Server** are now both in the **ESTABLISHED** state, where data can flow in both directions.
     - **Client:** ESTABLISHED
     - **Server:** ESTABLISHED
4. **Connection Release (Four-Way Handshake):**
   - **Client** initiates connection termination by sending a **FIN** and moving to **FIN WAIT 1**.
     - **Client:** ESTABLISHED → FIN WAIT 1
   - **Server** receives the FIN, sends an **ACK**, and moves to **CLOSE WAIT**.
     - **Server:** ESTABLISHED → CLOSE WAIT
   - **Client** receives the ACK and moves to **FIN WAIT 2**.
     - **Client:** FIN WAIT 1 → FIN WAIT 2
   - **Server** sends its own **FIN** after the application closes and moves to **LAST ACK**.
     - **Server:** CLOSE WAIT → LAST ACK
   - **Client** receives the server's **FIN**, sends an **ACK**, and moves to **TIME WAIT**.
     - **Client:** FIN WAIT 2 → TIME WAIT
   - **Server** receives the **ACK** and moves to **CLOSED**.
     - **Server:** LAST ACK → CLOSED
   - **Client** waits in the **TIME WAIT** state for twice the maximum packet lifetime to ensure all segments have been cleared, then moves to **CLOSED**.
     - **Client:** TIME WAIT → CLOSED



easier FSM

TCP Trans policy:

⇒ T.P plays a critical role controlling the flow of data b/w sender & receiver.
⇒ ensures receiver is not overwhelmed.
⇒ This is managed through window management mechanisms & addresses issues like the silly window syndrome.
⇒ TCP Window Management:
* TCP uses a sliding window protocol to manage data transmission.
* The window is a buffer space that controls how much data can be sent b4 needing an ack from the receiver.
* This window size determines how much data the sender can transmit b4 it waits for an ack from receiver.



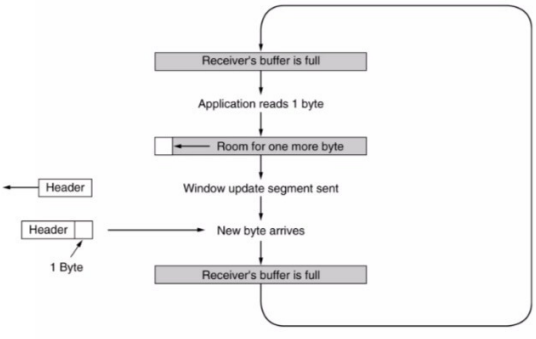
Window management in TCP.

* 2 Buffers & 1 window } to control flow of data coming from sender

* Segments sent → buffer → receives segments → buffer.
* Size of window can be increased, decreased by the dest.

⇒ The window size changes dynamically based on network conditions the receiver's avail buff space, and the RTT (round-trip time) b/w the sender and receiver. This ensures that the sender does not overwhelm the receiver and that both sides can handle the amt of data being trans.
⇒ TCP uses window mangment as a part of its flow & congs control mechanisms:
→ flow control: ensures sender does not send more data than the receiver can handle, based on receiver's advertised window size.
→ Congs control: prev sender from flooding the n/w with data, even if the recvr has a large buff space. This is managed by a separate window space (CWND), which limits the data in flight based on network congs.

⇒ Silly window syndrome:



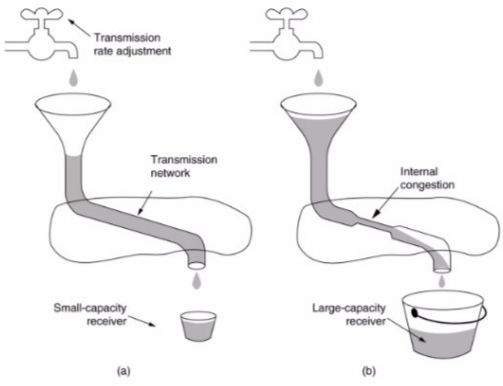
Silly window syndrome.

If receiver buff is full then no data will receive. The app reads 1 byte then 1 byte space becomes empty causing the window to update. This leads to an arrival of another byte and agn the receiver is full.... repeats....

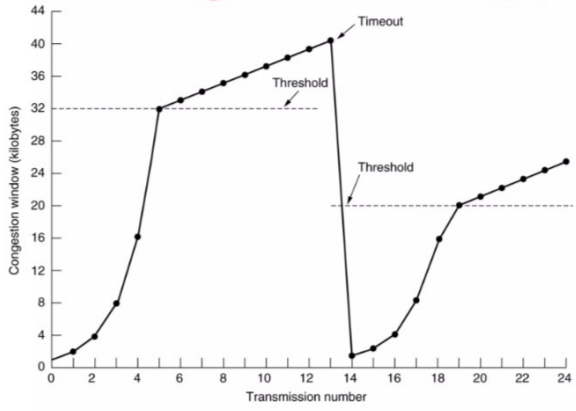
This decreases the working of TCP.

This mostly occurs when sender sends data in large blocks.

TCP Congestion Control:



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



An example of the Internet congestion algorithm.

TCP Timer Management:

⇒ TCP uses multiple Timers.

⇒ The most imp one is the RTO (retrans timeout):

* most crucial timer in TCP.
* responsible for retrans lost or delayed segments.
* The RTO is started when a seg is sent, and if the corresponding Ack arrives b4 the timer expires, the timer is stopped.
* If timer expires b4 receiving Ack, the seg is retrans.
* Ensures that lost packets are retrans after a resonable amt of time.
* determining appropriate timeout interval is difficult because the delay for an Ack can vary significantly depending on the n/w conditions like conges.

⇒ If timeout is too short (T1) it leads to unnecessary retrans, clogging the network. If the timeout is too long (T2), perfom suffers as the sender waits too long b4 retrans lost segments.

⇒ Therefore, TCP needs a dynamic alg to adapt the timeout interval continuously.

⇒ Dynamic calⁿ of Retrans timeout:

* TCP uses an alg proposed by Jacobson that dynamically adjusts the RTO based on network perf
  - → SRTT : Smoothed round-trip time (smoothed estimate of the R.T b/w sender & receiver)
  - → RTTVAR : round-trip time variation (variation in the R.T.T which accounts for fluctuations in n/w delays

* Calculating SRTT:

→ When an Ack is received, TCP measures the round-trip time (R) for the corresponding seg and updates the SRTT using an Exponentially Weighted Moving Avg (EWMA).

→ $SRTT = \alpha \times SRTT + (1-\alpha) \times R$.

→ where α is the smoothing factor. Typically $\alpha = \frac{7}{8}$. This formula ensures that newer measurements of RTT are given more weight while older vals decay gradually

* Calculating RTTVAR:
  → to account for variability, the RTTVAR is also updated.
  → RTTVAR = β × RTTVAR + (1 − β) × |SRTT − R|
  → where β = 3/4. This gives an estimate of how much the RTT is fluctuating over time.
* Calculating RTO:
  → RTO = SRTT + 4 × RTTVAR

⇒ EWMA: This is smoothing technique that discards random noise in RTT measurements, making the estimates more stable.

⇒ Karn's Algorithm:
* One challenge in cal the RTO is handling retrans signals. If a seg times out and is retransed, it is unclear whether the received Ack corresponds to the original or retrans seg.
* This uncertainty can corrupt RTT measurements. Karn's alg solves this.
* ① Ack to Karns Alg, when a seg is retransed, TCP does not update the RTT estimate using the Ack for that segment. This avoids contaminating the RTO calculation. ② And for each retrans, the RTO is doubled, ensuring that TCP doesn't flood the network with repeated retrans.

⇒ Other TCP Timers:
* Persistence Timer: * ensures comm doesn't get stuck when receiv's window is closed.
  * prevents deadlock when receiv's window size is 0. (no data can be accepted)
  * If window size remains 0 for an extended period, sender sends a "probe" seg to check if window size has increased. If the size is still 0, the persistent timer is reset & the cycle repeats.
* Keep-alive Timer: * checks if idle connections are still alive.
  * if no data has been exchanged for a long period, the timer triggers a keepalive message.
  * If the other side does not respond, connec is terminated.
* TIME-WAIT Timer: * ensures old packets are flushed from n/w after a connec is closed.
  * used during connec termination.
  * after closing a connec TCP keeps the conn in the TIME WAIT state for twice the max^m seg lifetime (MSL) to ensure that any delayed packets in the n/w are discarded. This prevents old duplicate packets from being mistaken for part of a new connec.

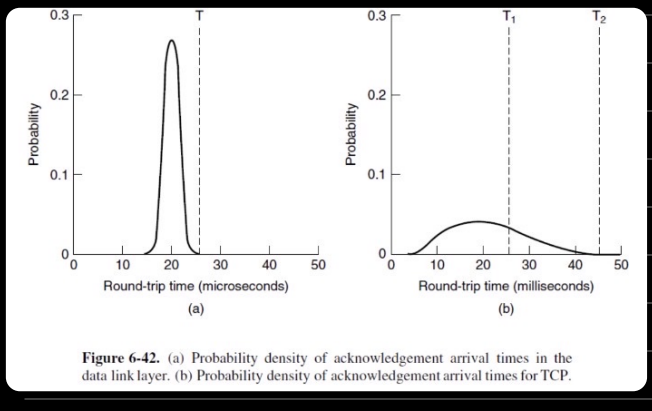


Figure 6-42. (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

Q

The following is a partial dump of a *TCP* header in hexadecimal format:

0532001700000001 00000000 500207FF 00000000

i) What is the source port number?
ii) What is the application being used?
iii) What is the sequence number?
iv) What is the ack number?

i) 0532 ⇒ 0x0532

⇒ $(0\times16^3)+(5\times16^2)+(3\times16^1)+(2\times16^0)$

⇒ 1330

ii) 0017 ⇒ 0x0017 ⇒ 23//

iii) 00000001 ⇒ $(0\times16^7)+\ldots+(1\times16^0)$

⇒ 1//

iv) 00000000 ⇒ 0x00000000

⇒ 0//

Application layer:

World Wide Web (WWW)

The **World Wide Web (WWW)** is an information system that allows documents and other web resources to be accessed over the internet using a web browser. It was invented by Tim Berners-Lee in 1989 and is based on three core technologies:

1. **HTML (HyperText Markup Language)**: The standard language for creating web pages.
2. **HTTP (HyperText Transfer Protocol)**: The protocol used for transferring web pages from a server to a browser.
3. **URLs (Uniform Resource Locators)**: The addresses used to locate web pages and other resources on the internet.

The WWW operates on the principle of **hypertext**, which allows users to navigate between web pages via hyperlinks. The content on the web is stored on web servers and is accessible by clients (browsers) using the HTTP protocol.

Static Web Pages

A **static web page** is a web page that displays fixed content. Each time a user visits the page, they see the same information unless the HTML code of the page is manually updated by a developer. Static web pages are usually written in plain HTML, CSS, and sometimes JavaScript.

Characteristics of Static Web Pages:

1. **Fixed Content**: The content doesn't change unless manually updated.
2. **Simple and Fast**: Since there is no server-side processing, static web pages load quickly.
3. **No Interaction**: Static pages do not interact with databases or change based on user input.
4. **Easier to Create**: Static pages are simpler to create and maintain, making them suitable for smaller websites, blogs, or portfolio pages.

Example: A company's "About Us" page, which presents the same information to all visitors.

Dynamic Web Pages

A **dynamic web page** is a web page that changes its content automatically based on user interaction, time, or other factors. These pages are generated on the fly by a web server, often using server-side programming languages like PHP, Python, Ruby, or JavaScript with frameworks like Node.js.

Characteristics of Dynamic Web Pages:

1. **Interactive Content**: Dynamic pages can display different content to different users or update content without reloading the page.
2. **Database-Driven**: Dynamic web pages often pull content from a database. For example, an e-commerce site fetches product details from a database based on a user's search query.
3. **Personalization**: Pages can be personalized for each user, such as showing a user's account information, preferences, or location-based content.
4. **Server-Side Processing**: Dynamic web pages require server-side processing to generate new content in real time.

Example: Social media sites like Facebook, where the content is unique to each user based on their interactions and preferences.

Working of HTTP (HyperText Transfer Protocol)

HTTP (HyperText Transfer Protocol) is the foundation of communication on the World Wide Web. It defines how web browsers (clients) request resources from web servers and how servers respond to those requests. HTTP follows a request-response model where the client sends a request to the server, and the server returns a response.

How HTTP Works:

1. **Client Request**: A client (usually a web browser) sends an HTTP request to the server for a specific resource (e.g., a web page, image, or file). This request contains a method, the requested URL, and additional information like headers.
2. **Server Processing**: The server processes the request. It may access a database, fetch files, or perform other operations depending on the resource.
3. **Server Response**: The server returns an HTTP response, which includes a status code (indicating success or error), response headers (like content type and caching rules), and the requested resource or an error message.
4. **Client Rendering**: The client receives the response and renders the content for the user (e.g., displaying a webpage or playing a video).

HTTP Methods

HTTP defines various methods that specify the type of request being made. These methods are used to perform different actions on the server.

Common HTTP Methods:

1. **GET**: Requests data from the server without changing its state.
   - **Example**: Loading a webpage or fetching an image.
   - **Characteristics**: Safe and idempotent (multiple requests result in the same outcome).
2. **POST**: Sends data to the server to create or update resources.
   - **Example**: Submitting a form or uploading a file.
   - **Characteristics**: Not idempotent, changes the server's state.
3. **PUT**: Updates or replaces an existing resource on the server.
   - **Example**: Updating a user profile.
   - **Characteristics**: Idempotent (multiple requests have the same effect).
4. **DELETE**: Deletes a specified resource on the server.
   - **Example**: Removing a user or deleting a post.
   - **Characteristics**: Idempotent.
5. **PATCH**: Partially updates an existing resource.
   - **Example**: Updating only the email field of a user profile.
   - **Characteristics**: Not idempotent.
6. **HEAD**: Similar to GET, but only requests the headers without the body of the resource.
   - **Example**: Checking if a file exists without downloading it.
7. **OPTIONS**: Describes the communication options for the target resource.
   - **Example**: Pre-flight checks in CORS (Cross-Origin Resource Sharing) to determine allowed methods.
8. **CONNECT**: Establishes a tunnel to the server, usually for secure SSL connections.
   - **Example**: Used in HTTPS for encrypted communication.
9. **TRACE**: Echoes the received request, used for debugging.

HTTP Caching

Caching in HTTP is the mechanism by which web content is temporarily stored to improve performance, reduce server load, and minimize network traffic. When a resource is cached, future requests for the same resource are served faster by the cache instead of making a full request to the server.

Types of HTTP Caches:

1. **Browser Cache:** Stores resources locally on the user's device.
2. **Proxy Cache:** Located between the client and the server, typically at an ISP or network level.
3. **Server Cache:** Stored on the server to speed up responses for repeated requests.

HTTP Caching Mechanisms:

1. **Cache-Control Headers:** Control caching behavior with various directives.
   - `max-age`: Specifies how long a resource can be cached (in seconds).
   - `no-cache`: Forces the client to revalidate the resource before using the cached version.
   - `no-store`: Prevents caching entirely.
   - `public`: Allows caching by any cache (browser, proxy).
   - `private`: Allows caching only by the browser, not shared caches.
2. **ETag (Entity Tag):** A unique identifier assigned to a resource. The client sends the ETag with future requests, and the server responds with a status code (like `304 Not Modified`) if the resource hasn't changed.
3. **Last-Modified:** A timestamp indicating the last time the resource was modified. The client can send this timestamp in an `If-Modified-Since` header, and the server will respond with a `304 Not Modified` if the resource hasn't changed.
4. **Expires:** Specifies an absolute date and time after which the resource is considered stale.

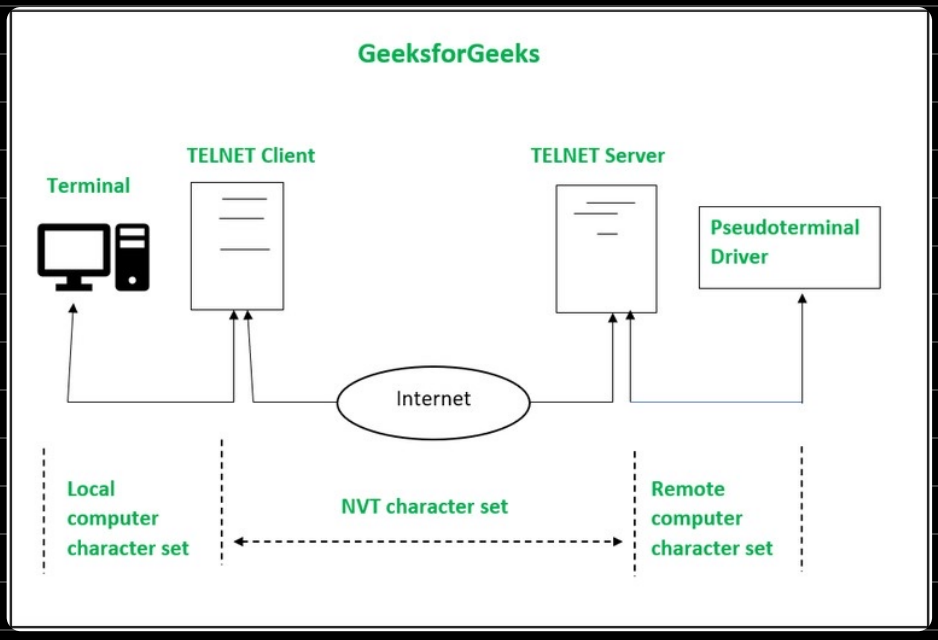
Caching Process:

1. **Initial Request:** The client requests a resource, and if the resource is not in the cache, the server provides it along with caching instructions.
2. **Subsequent Requests:** The client checks the cache to see if the resource is still valid. If so, the cached version is used, saving time and bandwidth.
3. **Revalidation:** If the cached resource has expired or needs revalidation (e.g., using `ETag` or `Last-Modified`), the client sends a conditional request. If the resource hasn't changed, the server responds with `304 Not Modified`.

Benefits of Caching:

- **Improved Performance:** Cached resources load faster as they are fetched locally.
- **Reduced Bandwidth Usage:** Less data is transmitted over the network, saving bandwidth.
- **Reduced Server Load:** Fewer requests reach the server, which can reduce the strain on server resources.

Telnet



Telnet Overview

Telnet (TELecommunication NETwork) is a protocol used for accessing remote computers over a network. It allows users to connect to a server or device and control it as if they were sitting in front of it, using a command-line interface (CLI). Telnet operates on **port 23** by default.

How Telnet Works

1. **Client-Server Communication**:
   - **Client**: The computer initiating the connection (e.g., your personal computer).
   - **Server**: The remote computer you want to access (e.g., a web server or router).
   - The client uses Telnet software to send commands to the server over the network, and the server responds to these commands.
2. **Establishing a Telnet Session**:
   - When you initiate a Telnet connection (e.g., by typing `telnet [server-ip]` in a terminal), the client sends a **TCP** connection request to the server on **port 23**.
   - The server accepts the connection, and a session is created between the client and server.
3. **Command Execution**:
   - Once connected, the client is presented with a **remote command-line interface** (CLI).
   - The user can type commands, which are sent over the network to the server.
   - The server executes the commands and sends back the output to the client, which is displayed in the client's terminal.
4. **Data Transmission**:
   - Telnet uses the **TCP/IP** protocol to ensure reliable delivery of data.
   - It is a text-based protocol, meaning all communication between the client and server is sent as plain text.

5. **Closing the Connection**:
    - Once the user is done, they can terminate the session by typing `exit` or `logout`.
    - The Telnet connection is closed, and control is returned to the client machine.

Key Characteristics of Telnet

- **Unencrypted**: Telnet transmits all data, including usernames and passwords, in plain text. This makes it insecure, as attackers can intercept and read the data.
    - As a result, Telnet has been largely replaced by **SSH** (Secure Shell), which provides encrypted communication.
- **Text-Based Protocol**: Telnet is primarily used for text-based communication between computers. It doesn't support graphical interfaces.
- **Interactive**: Telnet allows for interactive command execution, where the client sends commands to the server and the server responds immediately.

Common Uses of Telnet

1. **Remote Management**:
    - Telnet was historically used by network administrators to remotely manage devices such as routers, switches, and servers.
2. **Testing Network Services**:
    - Telnet can be used to check if a service (such as a web server) is reachable on a specific port. For example, you can use `telnet [server-ip] 80` to test if a web server is responding on port 80.
3. **Access to Legacy Systems**:
    - Some older or legacy systems that haven't upgraded to more secure protocols like SSH may still use Telnet for remote management.

Security Concerns

- **No Encryption**: Telnet sends data in plain text, making it vulnerable to **eavesdropping** and **man-in-the-middle attacks**.
- **Password Exposure**: Since Telnet does not encrypt passwords, anyone intercepting the data can easily obtain login credentials.

Because of these issues, Telnet has been mostly replaced by more secure alternatives like **SSH**, which provides encrypted communication for remote access.