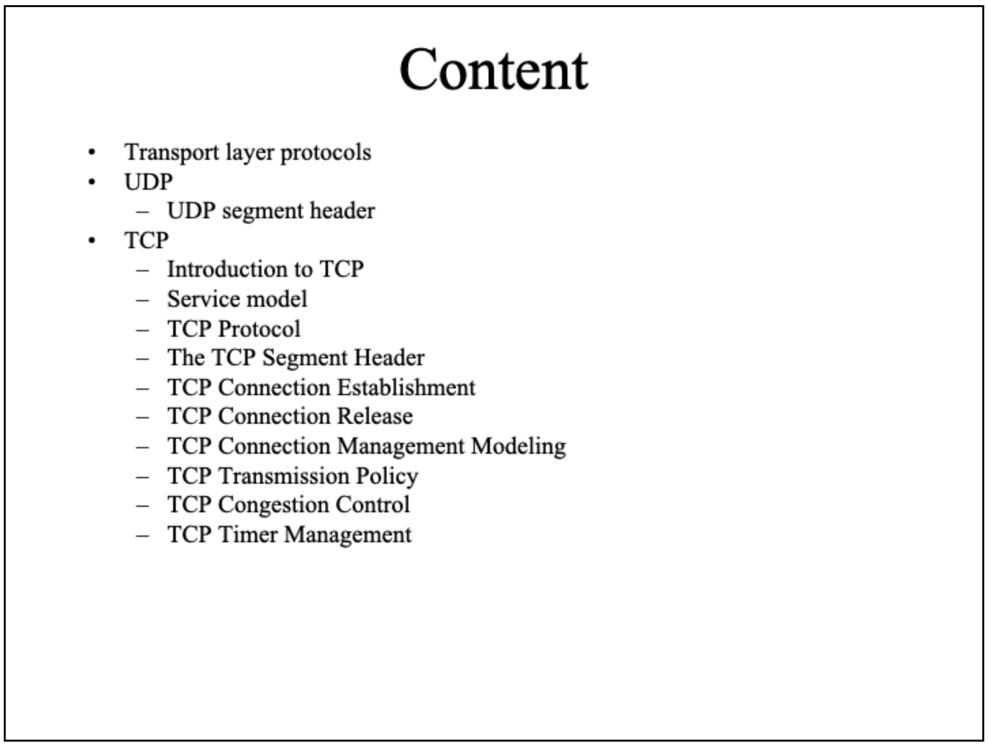
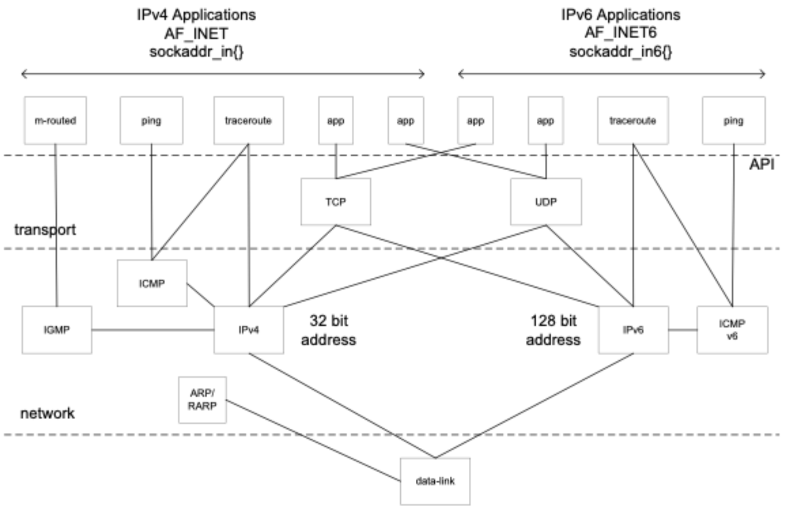
CT2108

Internet Transport Protocols
TCP and UDP



Content

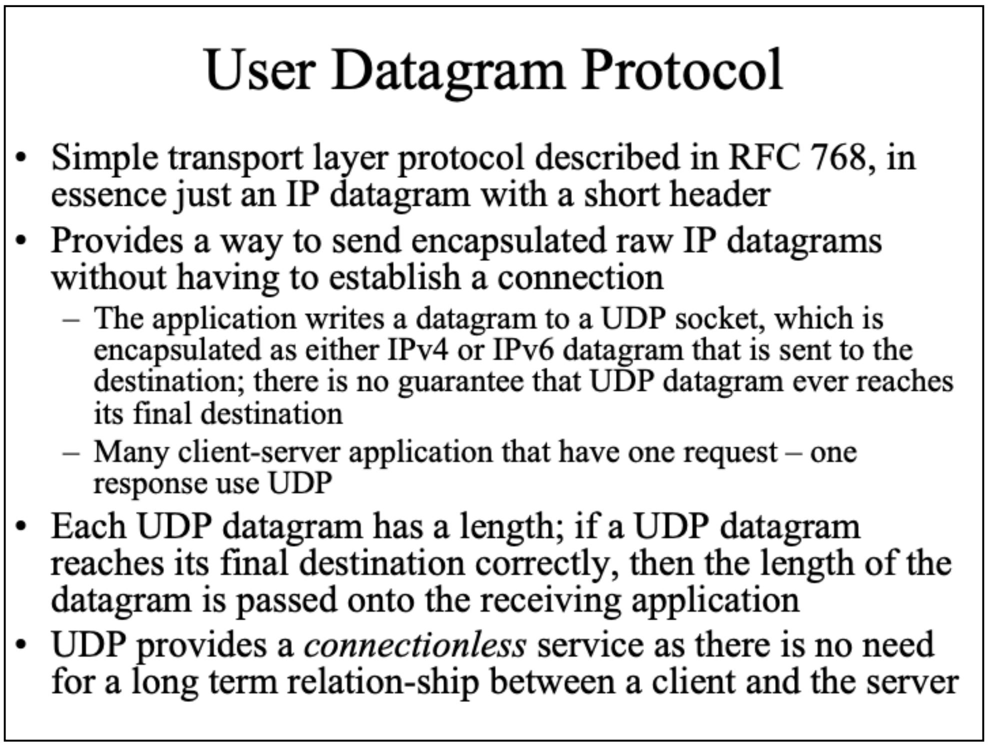
The big picture



- Internet has two main protocols at the transport layer
  - Connectionless protocol: UDP (User Datagram Protocol)
  - Connection oriented protocol: TCP (Transport Control Protocol)

ICMP – Internet Control and Messaging Protocol

IGMP – Internet Group Management Protocol – used for multicast addressing

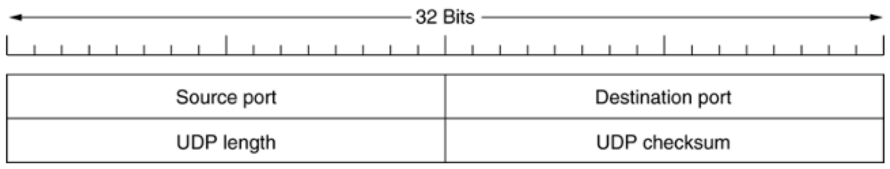


User Datagram Protocol

- Simple transport layer protocol described in RFC 768, in essence just an IP datagram with a short header
- Provides a way to send encapsulated raw IP datagrams without having to establish a connection
  - The application writes a datagram to a UDP socket, which is encapsulated as either IPv4 or IPv6 datagram that is sent to the destination; there is no guarantee that UDP datagram ever reaches its final destination
  - Many client-server application that have one request – one response use UDP
- Each UDP datagram has a length; if a UDP datagram reaches its final destination correctly, then the length of the datagram is passed onto the receiving application
- UDP provides a *connectionless* service as there is no need for a long term relation-ship between a client and the server

For example, a UDP client can create a socket and send a datagram to a given server and then immediately send another datagram on the same socket to a different server; similarly, UDP server can receive multiple datagrams from different sources on the same single UDP socket

UDP header



- UDP segment consists of 8 bytes header followed by the data
- The source port and destination port identify the end points within the source and destination machines; without the port fields, the transport would not know what to do with the packet; with them, it delivers the packets correctly to the application attached to the destination port
- The UDP length field includes the 8-byte header and the data
- The UDP checksum includes the 1 complement sum of the UDP data and header. It is optional, if not computed it should be stored as all bits “0”.

UDP

- Does not:
  - Flow control
  - Error control
  - Retransmission upon receipt of a bad segment
- Provides an interface to the IP protocol, with the added feature of demultiplexing multiple processes using the ports
- One area where UDP is useful is client server situations where the client sends a short request to the server and expects a short replay
  - If the request or reply is lost, the client can timeout and try again
- DNS (Domain Name System) is an application that uses UDP
  - shortly, the DNS is used to lookup the IP address of some host name; DNS sends and UDP packet containing the host name to a DNS server, the server replies with an UDP packet containing the host IP address. No setup is needed in advance and no release of a connection is required. Just two messages go over the network)
- Widely used in client server RPC
- Widely used in real time multimedia applications

Transmission Control Protocol

- Provides a reliable end to end byte stream over unreliable internetwork (different parts may have different topologies, bandwidths, delays, packet sizes, etc…)
- Defined in RFC793, with some clarifications and bug fixes in RFC1122 and extensions in RFC1323
- Each machine supporting TCP has a TCP transport entity (user process or part of the kernel) that manages TCP streams and interfaces to the IP layer
  - Accepts user data streams from local processes, breaks them into pieces (not larger than 64KB) and sends each piece as a separate IP datagram
  - At the receiving end, the IP datagrams that contains TCP packets, are delivered to the TCP transport entity, which reconstructs the original byte stream
  - The IP layer gives no guarantee that the datagrams will be delivered properly, so it is up to TCP to time out and retransmit them as the need arises.
  - Datagrams that arrive, may be in the wrong order; it is up to TCP to reassemble them into messages in the proper sequence

TCP Service Model (1)

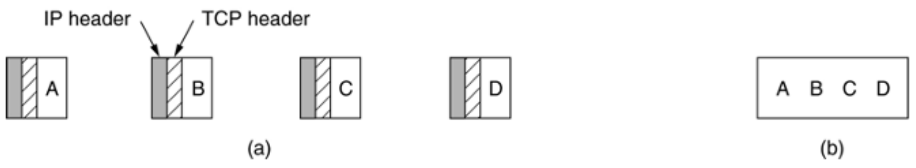
- Provides *connections* between clients and servers, both the client and the server create end points, called *sockets*
  - Each socket has a number (address) consisting of the IP address of the host and a 16 bits number, local to that host, called *port* (TCP name for TSAP)
  - To obtain a TCP service, a connection has to be explicitly established between the two end points
  - A socket may be used for multiple connections at the same time
  - Two or more connections can terminate in the same socket; connections are identified by socket identifiers at both ends (*socket1*, *socket2*)
- Provides *reliability*
  - When TCP sends data to the other end it requires acknowledgement in return; if ACK is not received, the TCP entity retransmits automatically the data and waits a longer amount of time

TCP Service Model (2)

- TCP contains algorithms to estimate *round-trip time* between a client and a server to know dynamically how long to wait for an acknowledgement
- TCP provides *flow control* – it tells exactly to its peer how many bytes of data is willing to accept from its peer.
- Port numbers below 1024 are called *well known ports* and are reserved for standard services e.g.
  - Port 21 used by FTP (File Transfer Protocol)
  - Port 22 used by SSH (Secure Remote Login)
  - Port 23 used by Telnet (Remote Login – insecure)
  - Port 25 used by SMTP (E-mail transport between servers)
  - Port 69 used by TFTP (Trivial File Transfer Protocol)
  - Port 80 used by HTTP (World Wide Web - insecure)
  - Port 443 used by TLS (Transport Layer Security – secure WWW)

TCP Service Model (3)

- All TCP connections are full duplex and point to point (it doesn't support multicasting or broadcasting)
- A TCP connection is a byte stream not a message stream (message boundaries are not preserved)
  - i.e. if a process is doing four writes of 512 bytes to a TCP stream, data may be delivered at the other end as four 512 bytes reads, two 1024 bytes reads or one 2048 read



(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.

TCP Service Model (4)

- When an application sends data to TCP, TCP entity may send it immediately or buffer it (in order to collect a larger amount of data to send at once)
  - Sometimes the application really wants the data to be sent immediately (i.e. after a command line has been finished); to force the data out, applications can use the PUSH flag (which tells TCP not to delay transmission)
- *Urgent data* – is a way of sending some urgent data from one application to another over TCP
  - i.e. when an interactive user hits DEL or CTRL-C key to break-off the remote computation; the sending app puts some control info in the TCP stream along with the URGENT flag; this causes the TCP to stop accumulating data and send everything it has for that connection immediately; when the urgent data is received at the destination, the receiving application is interrupted (by sending a break signal in Unix), so it can read the data stream to find the urgent data

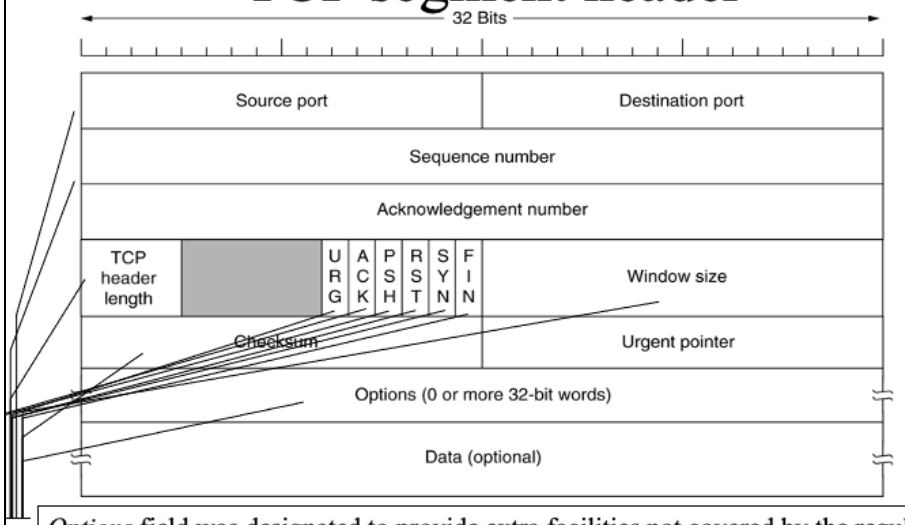
TCP Protocol Overview (1)

- Each byte on a TCP connection has its own 32-bit sequence number, used for various purposes (re-arrangement of out of sequence segments; identification of duplicate segments, etc…)
- Sending and receiving TCP entities exchange data in the form of *segments*. A segment consists of a fixed 20-byte fixed header (plus an optional part) followed by zero or more data bytes
- The TCP software decide how big segments should be; it can accumulate data from several writes to one segment or split data from one write over multiple segments.
  - Two limits restrict the segment size:
    - Each segment including the TCP header must fit the 65535-byte IP payload
    - Each network has a maximum transfer unit (*MTU*) and each segment must fit in the MTU (in practice the MTU is a few thousand bytes and therefore sets the upper bound on the segment size)

TCP protocol Overview (2)

- The basic protocol 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 sends back a segment (with data, if any data is to be carried) bearing an acknowledgement number equal to the next sequence number it expects to receive;
  - If sender's timer goes off, before an acknowledgement has been received, the segment is sent again
- Problems with the TCP protocol
  - Segments can be fragmented on the way; parts of the segment can arrive, but some could get lost
  - Segments can be delayed, and duplicates can arrive at the receiving end
  - Segments may hit a congested or broken network along its path

TCP segment header



Options field was designated to provide extra facilities not covered by the regular header; the most important one is the one that allows each host to specify the maximum TCP payload is willing to accept; all Internet hosts are required to accept at least 536 +20 TCP segments

Source and *destination* ports identify the local endpoints for the connection; each host may decide for itself how to allocate the its own ports starting at 1024; a port number plus its host IP form a 48 bits unique TSAP

Sequence Number is associated with every byte that is sent. It is used for a number of different purposes: it is used to re-arrange the data at the receiving end, before passing it to the application; it is used to detect duplicate data and

Acknowledgement number fields specifies the next byte expected

TCP header length tells how many 32-bit words are contained in the TCP header; this is required because the *Options* field is of variable length; technically it indicates the start of data within the segment, measured in 32 bits words.

URG flag is set to 1 if urgent pointer is in use; the urgent pointer is used to indicate a byte offset from the current sequence number at which urgent data is to be found; this facilitates interrupt messages without getting the TCP itself involved in carrying such message types

ACK flag is set to 1 to indicate that the acknowledgement number is valid; if set to 0, then the packet doesn't contain an acknowledgement, so the appropriate field is ignored (the Acknowledgement number field is ignored)

PSH flag indicates pushed data, so the receiver is requested to deliver the received data to the application upon arrival, without buffering it to form a full buffer has been received

RST flag is used to restart a connection that has become confused due to a host crash or any other reasons; it is also used to reject an invalid segment or refuse an attempt to open a connection

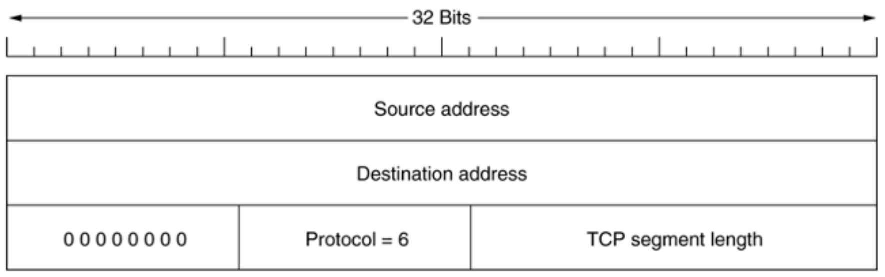
SYN flag is used to establish connections; the connection requests have SYN = 1 and ACK = 0 to indicate that the piggyback acknowledgement field is not in use; the connection response does bear an acknowledgment, so it has SYN = 1 and ACK = 1; In essence SYN bit is used to denote a CONNECTION REQUEST and a CONNECTION ACCEPTED with ACK field used to distinguish between those two possibilities

Flow control in TCP is done using variable size sliding window; the *Window size* field tells how many bytes may be sent starting at the byte acknowledged; a window size field with value 0 is legal and means that bytes up to (*Acknowledgement number* -1) have been received, and no more accepted; to resume receiving data, the receiver releases another segment with a window size different than 0 and same acknowledge number

Checksum is provided for extreme reliability. It checksums the header, the data and the conceptual pseudo-header shown on the next slide; when performing the computation, the data field is padded with an additional zero byte if its length is an odd number; the checksum is simply the sum in 1's complement; as consequence, when the receiver performs the calculation on the entire segment, including the checksum field, the result should be 0

Options field was designated to provide extra facilities not covered by the regular header; the most important one is the one that allows each host to specify the maximum TCP payload is willing to accept; all Internet hosts are required to accept at least 536 +20 TCP segments

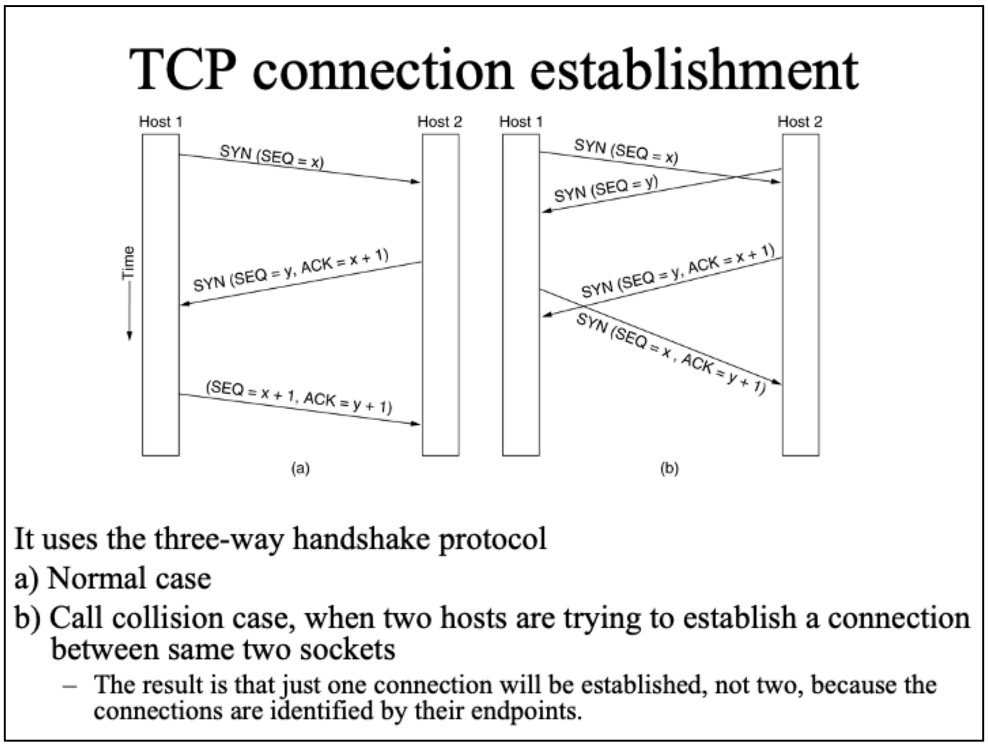
TCP pseudoheader



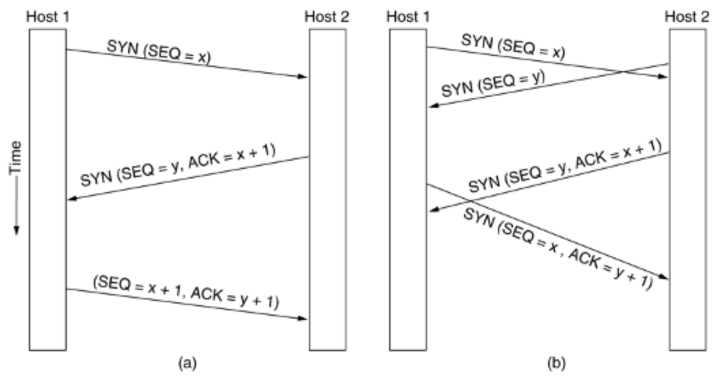
- The pseudoheader contains the 32 bits IP addresses of the source and the destination machines, the protocol number (6 for TCP) and the byte count for the TCP segment (including the header)
- Including this pseudoheader in the TCP checksum calculation helps detect miss-delivered packets, but doing so violates the protocol hierarchy (since IP addresses belong to the network layer, not to the TCP layer)

TCP extra options

- For lines with high bandwidth and high delay, the 64KB widow size is often a problem
  - i.e., on a T3 line (44.736 Mb/s) it takes only about 12ms to output a full 64KB window. If the round-trip propagation delay is 50ms (typical for a transcontinental fiber), then the sender will be idle ¾ of the time, waiting for acknowledgements
  - In RFC 1323 a window scale option was proposed, allowing the sender and receiver to negotiate a window scale factor. This number allows both sides to shift the window size up to 14 bits to the left, thus allowing for window size up to 2^{30} bytes; most TCP implementations support this option
- The use of "*selective repeat*" instead of "*go and back n*" protocol described in RFC 1106
  - If the receiver gets a bad segment and then a large number of good ones, the normal TCP protocol eventually time out and retransmit all the unacknowledged segments, including the ones that were received correctly
  - RFC 1106 introduces NACs to allow the receiver to ask for a specific segment (or segments). After it gets those, it can acknowledge all the buffered data, thus reducing the amount of data retransmitted.



TCP connection establishment



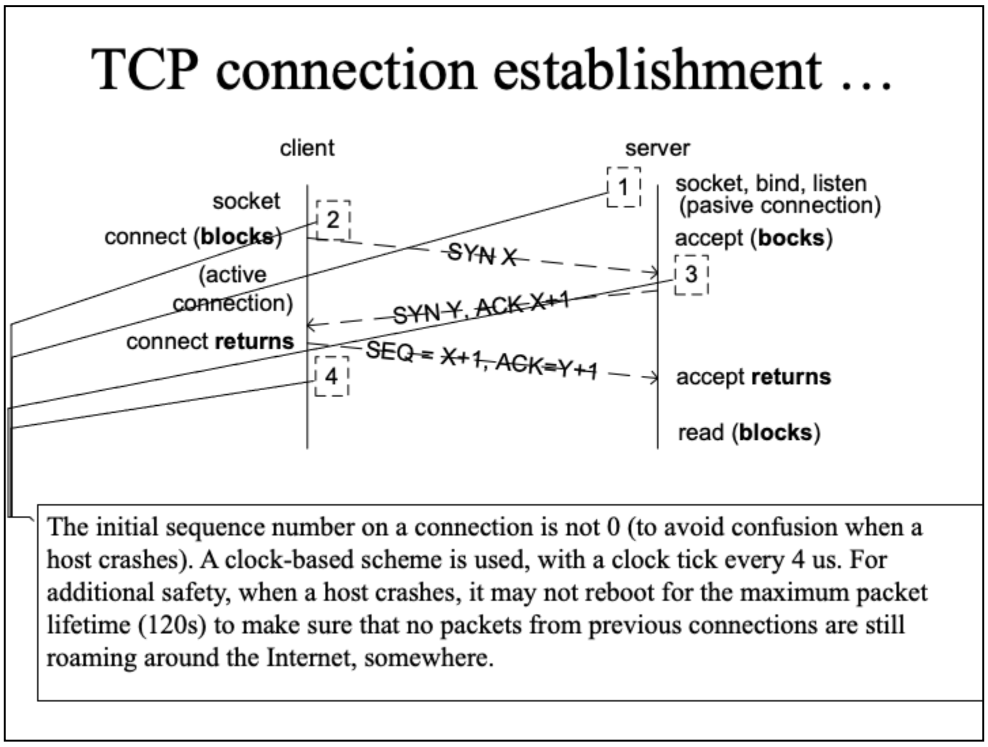
Host 1
Host 2
SYN (SEQ = x)
Time
SYN (SEQ = y, ACK = x + 1)
(SEQ = x + 1, ACK = y + 1)
(a)
Host 1
Host 2
SYN (SEQ = x)
SYN (SEQ = y)
SYN (SEQ = y, ACK = x + 1)
SYN (SEQ = x , ACK = y + 1)
(b)

It uses the three-way handshake protocol

a) Normal case

b) Call collision case, when two hosts are trying to establish a connection between same two sockets

  – The result is that just one connection will be established, not two, because the connections are identified by their endpoints.



TCP connection establishment …

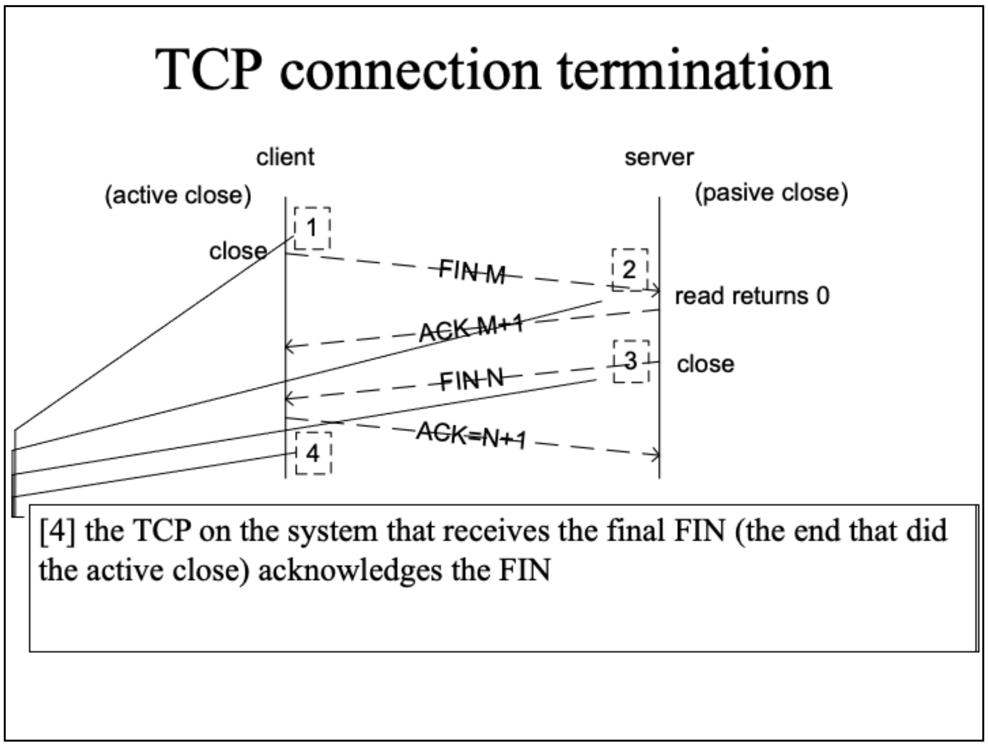
The initial sequence number on a connection is not 0 (to avoid confusion when a host crashes). A clock-based scheme is used, with a clock tick every 4 us. For additional safety, when a host crashes, it may not reboot for the maximum packet lifetime (120s) to make sure that no packets from previous connections are still roaming around the Internet, somewhere.

[1] the server must be prepared to accept an incoming connection; this is normally done by calling socket, bind and listen and it is called *passive open*
[2] the client, after the creation of a new socket, issues an *active open* by calling connect. This causes the client TCP to send a SYN segment (which stands for synchronize) to tell the server the client's initial sequence number for the data that the client will send on the connection; normally there is no data sent with SYN: it just contains an IP header, a TCP header and possible TCP options
[3] the server must acknowledge the client's SYN and the server must also send its own SYN containing the initial sequence number for the data that the server will send on the connection. The server sends SYN and the ACK of the client's SYN in a single segment
[4]the client must acknowledge the server's SYN

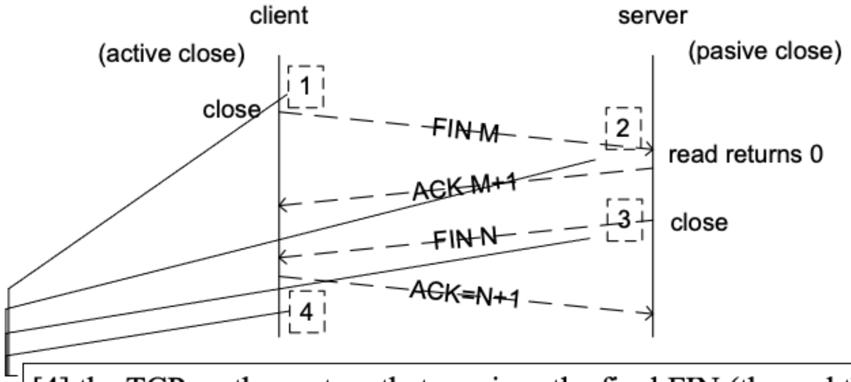
The initial sequence number on a connection is not 0 (to avoid confusion when a host crashes). A clock-based scheme is used, with a clock tick every 4 us. For additional safety, when a host crashes, it may not reboot for the maximum packet life time (120s) to make sure that no packets from previous connections are still roaming around the Internet, somewhere.

TCP connection termination

- The connection is full duplex, each simple connection is released independently
- To release a connection, either party can send a TCP segment with FIN bit set, which means that there is no more data to transmit
- When FIN is acknowledged, that direction is shut down for new data; however, data may continue to flow indefinitely in the other direction
- When both directions have been shutdown, the connection is released
- Normally, four TCP segments are used to shutdown the connection (one FIN and one ACK for each direction)
- To avoid complications when segments are lost, timers are used; if the ACK for a FIN packet is not arriving in two packet lifetimes, the sender of the FIN releases the connection; the other side will eventually realize that nobody seem to listen to it anymore and times out as well

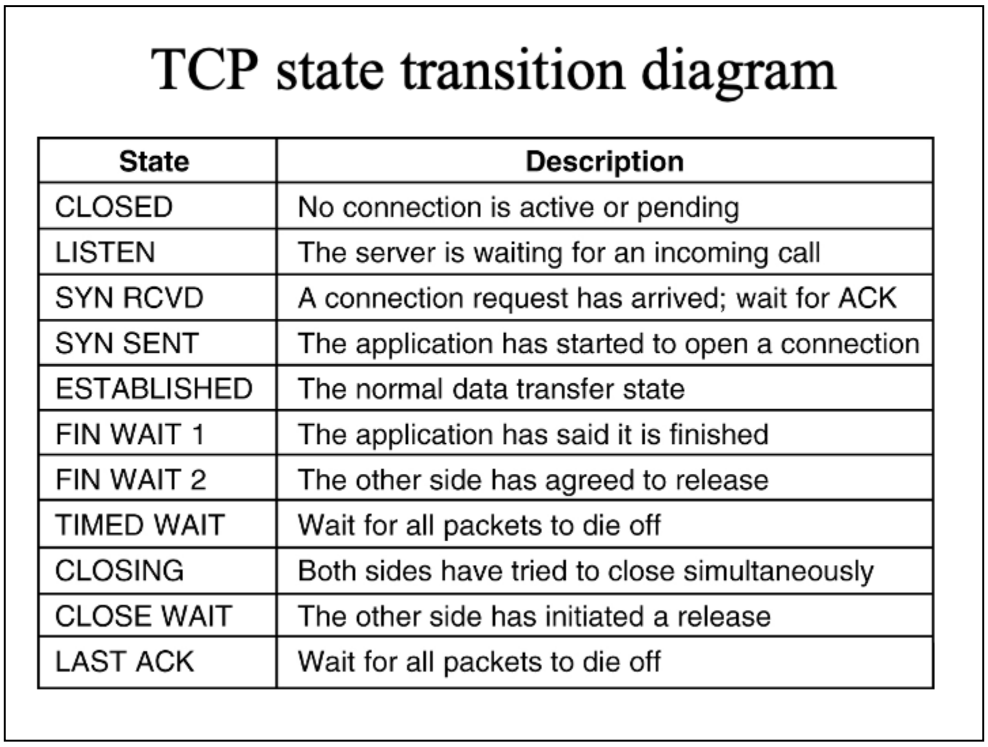


TCP connection termination



[4] the TCP on the system that receives the final FIN (the end that did the active close) acknowledges the FIN

[1] one application calls close first, and we say that this end performs the active close. This end's TCP sends a FIN segment, which means it is finished sending data
[1] one application calls close first, and we say that this end performs the active close. This end's TCP sends a FIN segment, which means it is finished sending data
[3] sometime latter, the application that received the end-of-file will close the socket; this will cause its TCP to send a FIN packet
[4] the TCP on the system that receives the final FIN (the end that did the active close) acknowledges the FIN



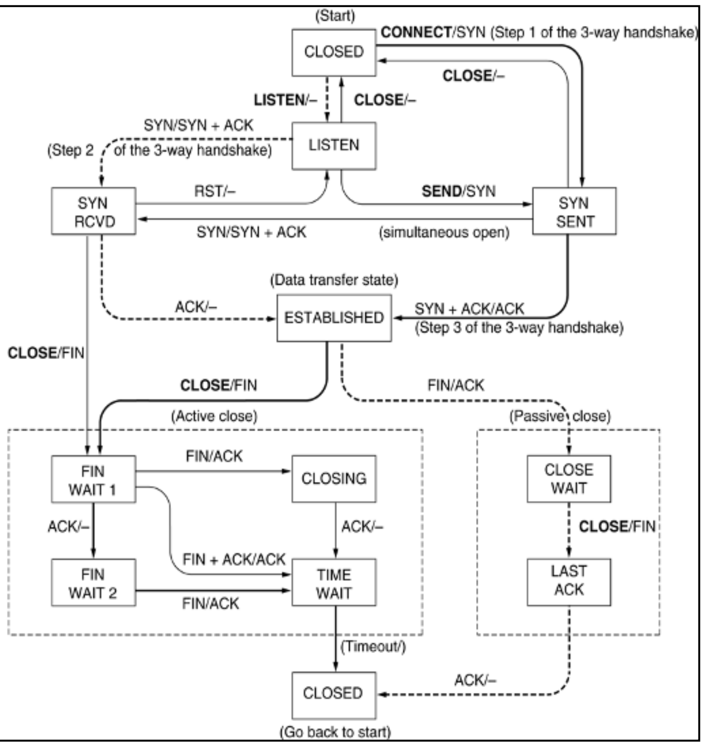
TCP state transition diagram

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
TIMED 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

The steps involved in establishing and releasing connections can be described/modeled using a Finite State Machine model. TCP can be represented as a FSM with 11 states.

Each connection starts in a CLOSED state. It leaves that state when it does either a passive open (LISTEN) or an active open (CONNECT). If the other side does the opposite one, a connection is established, and the state becomes ESTABLISHED. Connection release can be initiated by either side. When it is complete, the state returns to CLOSED

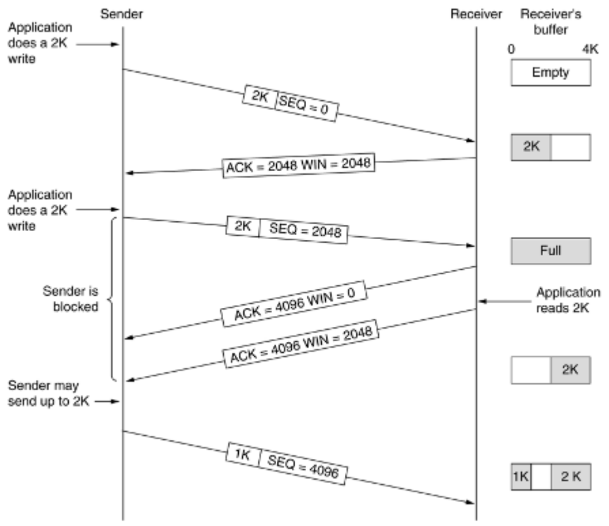
- The heavy solid line – client
- The heavy dashed line – server
- The light lines - unusual events
- Each transition is labeled by Event causing it/Action



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 by the event causing it and the action resulting from it, separated by a slash.

The event can either be a user-initiated system call (CONNECT, LISTEN, SEND or CLOSE), a segment arrival (SYN, FYN, ACK or RST) or in one case a timeout. The action is the sending of a control segment (SYN, FIN or RST) or nothing, indicated by “-”

TCP transmission policy



- Window management in TCP, starting with the client having a 4096 bytes buffer

TCP transmission policy

- When window size is 0 the sender can't send segments with two exceptions
  - Urgent data may be sent (i.e. to allow the user to kill the process running on the remote machine)
  - The sender may send 1 byte segment to make the receiver re-announce the next byte expected and window size
- Senders are not required to send data as soon as they get it from the application layer;
  - i.e. when the first 2KB of data came in from the application, TCP may have decided to buffer it until the next 2KB of data would have arrived, and send at once a 4KB segment (knowing that the receiver can accept 4KB buffer)
  - This leaves space for improvements
- Receivers are not required to send acknowledgements as soon as possible

TCP performance issues (1)

- Consider a telnet session to an interactive editor that reacts to every keystroke, we will have the worst-case scenario:
  - when a character arrives at the sending TCP entity, TCP creates a 21 bytes segment, which is given to IP to be sent as a 41 bytes datagram;
  - at the receiving side, TCP immediately sends a 40 bytes acknowledgement (20 bytes TCP segment headers and 20 bytes IP headers)
  - Latter, at the receiving side, when the editor (application) has read the character, TCP sends a window update, moving the window 1 byte to the right; this packet is also 40 bytes
  - Finally, when the editor has interpreted the character, it will echo it as a 41-byte character
- We will have 162 bytes of bandwidth are used and four segments are sent for each character typed.

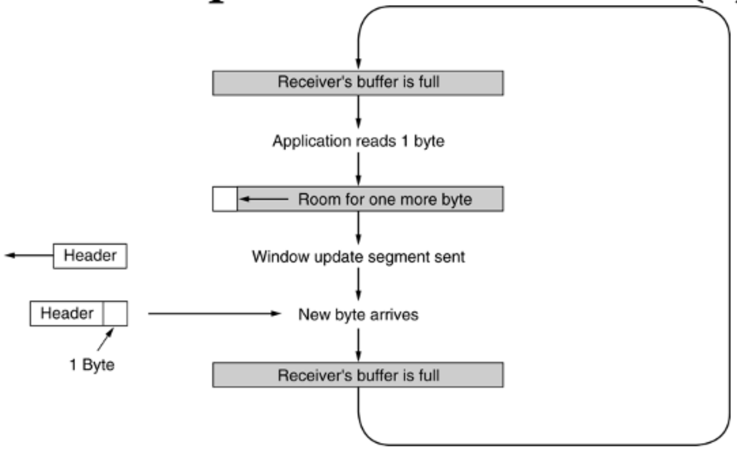
TCP optimizations – delayed ACK

- One solution that many TCP implementation use to optimize this situation is to delay acknowledgements and window updates for 500ms
  - The idea is the hope to acquire some data that will be bundled in the ACK or window update segment
    - i.e. in the editor case, assuming that the editor sends the echo within 500 ms from the character read, the window update and the actual byte of data will be sent back as a 41 bytes packet
  - This solution deals with the problem at the receiver end, it doesn' t solve the inefficiency at the sending end

TCP optimizations – Nagle's algorithm

- Operation:
  - When data come into the sender TCP one byte at a time, just send the first byte as a single TCP segment and buffer all the subsequent ones until the first byte is acknowledged
  - Then send all the buffered characters in one TCP segment and start buffering again until they are all acknowledged
  - The algorithm additionally allows a new segment to be sent if enough data has accumulated to fill half the window or a new maximum segment
- If the user is typing quickly and the network is slow, then a substantial number of characters may go in each segment, greatly reducing the usage of the bandwidth
- Nagle's algorithm is widely used in TCP implementations; there are times when it is better to disable it:
  - i.e. when an X-Window is run over internet, mouse movements have to be sent to remote computer; gathering them and send them in bursts, make the cursor move erratically at the other end.

TCP performance issues (2)



- Silly window syndrome (Clark, 1982)
  - Data is passed to the sending TCP entity in large blocks
  - Data is read at the receiving side in small chucks (1 byte)

TCP optimizations – Clark's solution

- Clark's solution:
  - Prevent the receiver from sending a window update for 1 byte
  - Instead have the receiver to advertise a decent amount of space available; specifically, the receiver should not send a window update unless it has space to handle the maximum segment size (that has been advertised when the connection was established) or its receiving buffer its half empty, which ever is smaller
  - Furthermore, the sender can help by not sending small segments; instead it should wait until it has accumulated enough space in the window to send a full segment or at least one containing half of the receiver's buffer size (which can be estimated from the pattern of window updates it has received in the past)

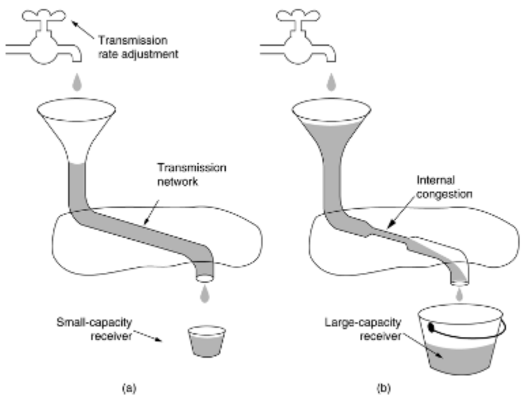
Nagle's algorithm vs. Clark's solution

- Clark's solution to the silly window syndrome and Nagle's algorithm are complementary
  - Nagle was trying to solve the problem caused by the sending application deliver data to TCP one byte at a time
  - Clark was trying to solve the problem caused by the receiving application reading data from TCP one byte at a time
  - Both solutions are valid and can work together. The goal is for the sender not to send small segments and the receiver not to ask for them
- The receiving TCP can also improve performance by blocking a READ request from the application until it has a large chunk of data to provide:
  - However, this can increase the response time.
  - But, for non-interactive applications (e.g. file transfer) efficiency may outweigh the response time to individual requests.

TCP congestion control

- TCP deals with congestion by dynamically manipulating the window size
- First step in managing the congestion is to detect it
  - A timeout caused by a lost packet can be caused by
    - Noise on the transmission line (not really an issue for modern infrastructure)
    - Packet discard at a congested router
  - Most transmission timeouts are due congestion
- All the Internet TCP algorithms assume that timeouts are due to congestion and monitor timeouts to detect congestion

TCP congestion control



- Two types of problems can occur:
  - Network capacity
  - Receiver capacity
- When the load offered to a network is more than it can handle, congestion builds up
  - (a) A fast network feeding a low-capacity receiver.
  - (b) A slow network feeding a high-capacity receiver.

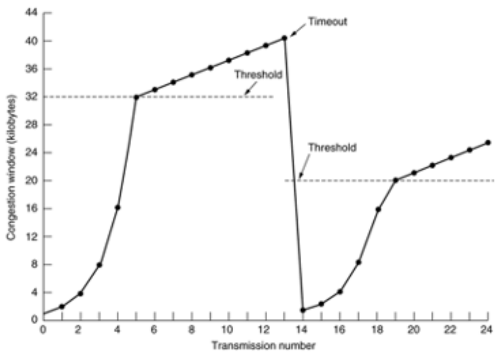
TCP congestion control

- TCP deals with network capacity congestion and receiver capacity congestion separately; the sender maintains two windows
  - The window that the receiver has guaranteed
  - The *congestion window*
- Both of the windows reflect the number of bytes that the sender may transmit; the number that can be transmitted is the minimum of the two windows
  - If the receiver says “send 8K” but the sender knows that more than 4K will congest the network, it sends 4K
  - On the other hand, if the receiver says “send 8K” and the sender knows that the network can handle 32K, then it sends the full 8K
  - Therefore, the effective window is the minimum between what the sender thinks is all right and the receiver thinks is all right

Slow start algorithm (Jacobson, 1988)

- When a connection is established, the sender initializes the congestion window to the size of the maximum segment in use; it then sends a maximum segment
  - If the segment is acknowledged in time, the sender doubles the size of the congestion window (making it twice the size of a segment) and sends two segments, that have to be acknowledged separately
  - As each of those segments is acknowledged in time, the size of the congestion window is increased by one maximum segment size (in effect, each burst successfully acknowledged doubles the congestion window)
- The congestion window keeps growing until either a timeout occurs, or the receivers window is reached
- The idea is that if bursts of size, say 1024, 2048, 4096 bytes work fine, but burst of 8192 bytes timeouts, congestion window remains at 4096 to avoid congestion; as long as the congestion window remains at 4096, no larger bursts than that will be sent, no matter how much space the receiver grants

Slow start algorithm (Jacobson, 1988)



• Max segment size is 1024 bytes
• Initially the threshold was 64KB, but a timeout occurred, and the threshold is set to 32KB and the congestion window to 1024 at transmission time 0

- The internet congestion algorithm uses a third parameter, the **threshold**, initially 64K, 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 size.
- Each burst successfully acknowledged doubles the congestion window.
  - It grows exponentially until the threshold value is reached.
  - It then grows linearly until the receivers window value is reached.

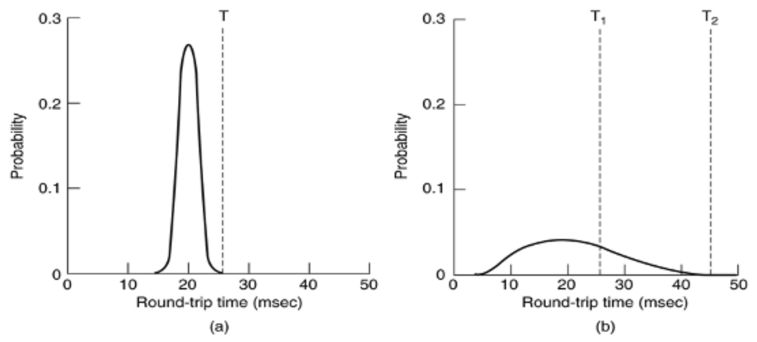
TCP timer management

- TCP uses multiple timers to do its work
  - The most important is the *retransmission timer*
    - When a segment is sent, a retransmission timer is started
    - If the segment is acknowledged before this timer expires, the timer is stopped
    - If the timer goes off before the segment is acknowledged, then the segment gets retransmitted (and the timer restarted)
    - The big question is how long this timer interval should be?
  - *Keepalive timer* is designed to check for connection integrity
    - When goes off (because a long time of inactivity), causing one side to check if the other side is still there

TCP timer management

- TCP uses multiple timers to do its work
  - *Persistence timer* is designed to prevent deadlock
    - Receiver sends a packet with window size 0
    - Latter, it sends another packet with larger window size, letting the sender know that it can send data, but this segment gets lost
    - Both the receiver and transmitter are waiting for the other
    - Solution: persistence timer on the sender end, that goes off and produces a probe packet to go to the receiver and make it to advertise again its window
  - *TIMED WAIT state timer* used when a connection is closed; it runs for twice the maximum packet lifetime to make sure that when a connection is closed, all packets belonging to this connection have died off.

TCP Retransmission Timer

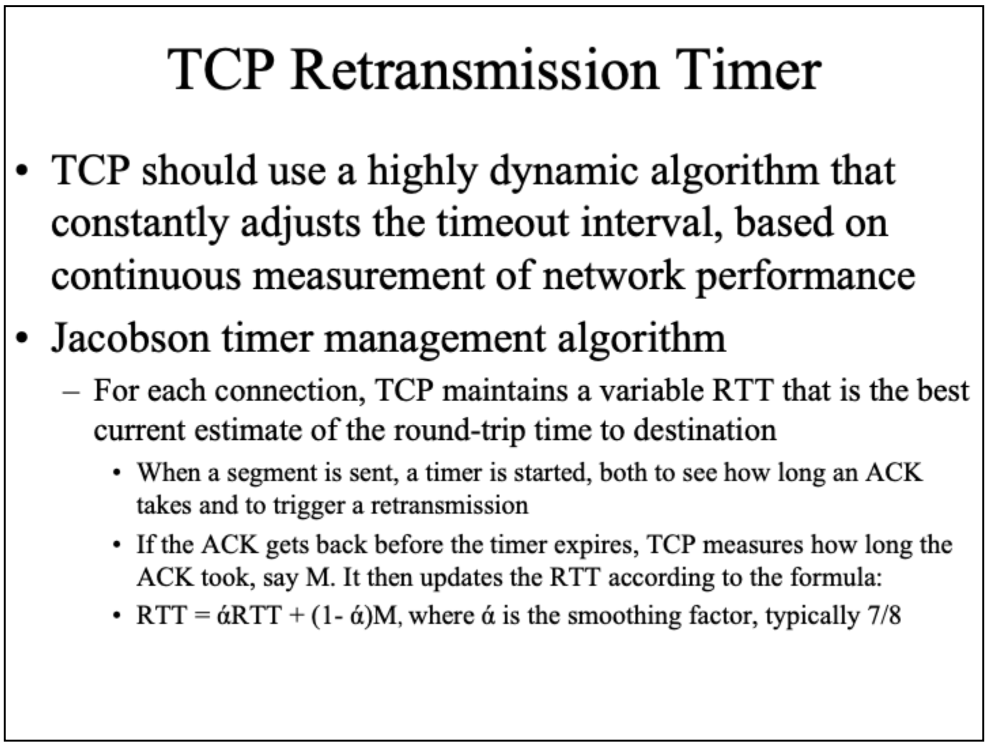


(a) Probability density of ACK arrival times in the data link layer.

- Expected delay is highly predictable, so the timer can be set to go off slightly after the acknowledgement is expected; since ACK are rarely delayed in data link layer, the absence of it means that it has been lost

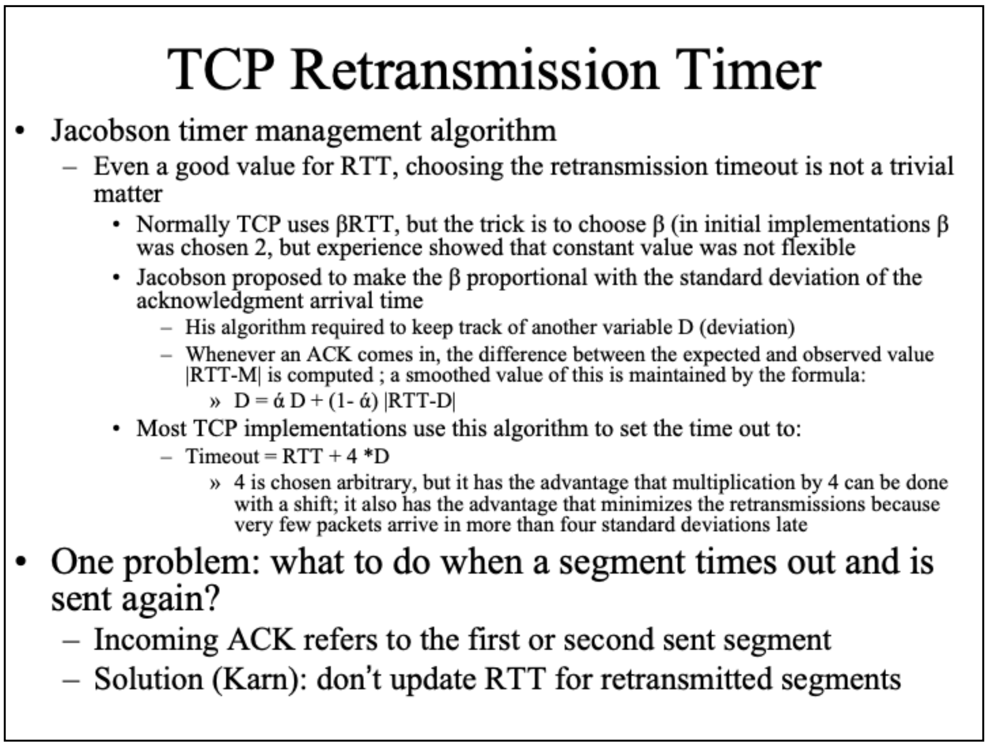
(b) Probability density of ACK arrival times for TCP.

- Determining the round-trip time to the destination is tricky; even if RRT is known, determining the timeout interval is difficult
  - Too small, unnecessary transmissions will occur
  - Too large, performance will be affected



TCP Retransmission Timer

- TCP should use a highly dynamic algorithm that constantly adjusts the timeout interval, based on continuous measurement of network performance
- Jacobson timer management algorithm
  - For each connection, TCP maintains a variable RTT that is the best current estimate of the round-trip time to destination
    - When a segment is sent, a timer is started, both to see how long an ACK takes and to trigger a retransmission
    - If the ACK gets back before the timer expires, TCP measures how long the ACK took, say M. It then updates the RTT according to the formula:
    - RTT = άRTT + (1- ά)M, where ά is the smoothing factor, typically 7/8



TCP Retransmission Timer

- Jacobson timer management algorithm
  - Even a good value for RTT, choosing the retransmission timeout is not a trivial matter
    - Normally TCP uses βRTT, but the trick is to choose β (in initial implementations β was chosen 2, but experience showed that constant value was not flexible
    - Jacobson proposed to make the β proportional with the standard deviation of the acknowledgment arrival time
      - His algorithm required to keep track of another variable D (deviation)
      - Whenever an ACK comes in, the difference between the expected and observed value |RTT-M| is computed ; a smoothed value of this is maintained by the formula:
        - » D = ά D + (1- ά) |RTT-D|
    - Most TCP implementations use this algorithm to set the time out to:
      - Timeout = RTT + 4 *D
        - » 4 is chosen arbitrary, but it has the advantage that multiplication by 4 can be done with a shift; it also has the advantage that minimizes the retransmissions because very few packets arrive in more than four standard deviations late
- One problem: what to do when a segment times out and is sent again?
  - Incoming ACK refers to the first or second sent segment
  - Solution (Karn): don’t update RTT for retransmitted segments