TCP Review

Carey Williamson
Department of Computer Science
University of Calgary

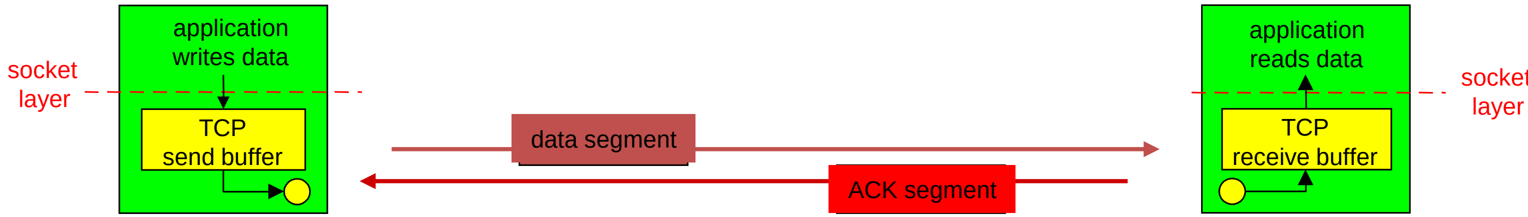
Credit: Most of this content was provided by Erich Nahum (IBM Research)

Transmission Control Protocol (TCP)

- Connection-oriented, point-to-point protocol:
  - Connection establishment and teardown phases
  - ‘Phone-like’ circuit abstraction (application-layer view)
  - One sender, one receiver
  - Called a “reliable byte stream” protocol
  - General purpose (for any network environment)

- Originally optimized for certain kinds of transfer:
  - Telnet (interactive remote login)
  - FTP (long, slow transfers)
  - Web is like neither of these!

TCP Protocol (cont'd)



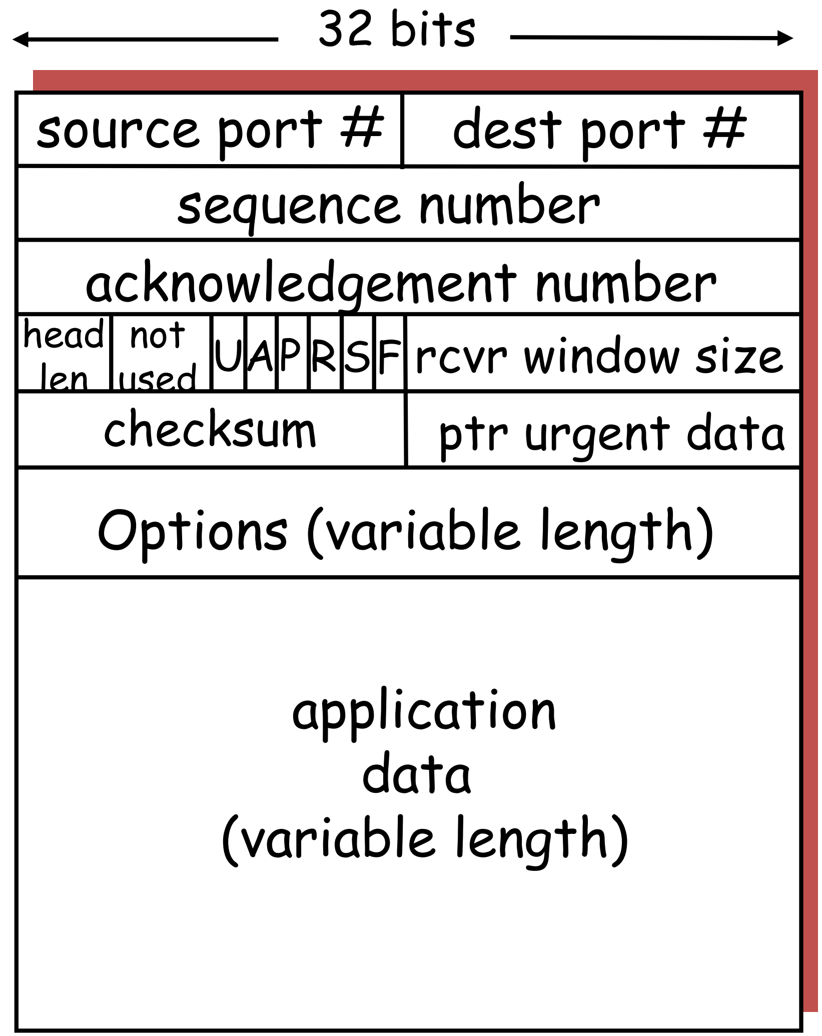
- Provides a reliable, in-order, byte stream abstraction:
  - Recover lost packets and detect/drop duplicates
  - Detect and drop corrupted packets
  - Preserve order in byte stream, no "message boundaries"
  - Full-duplex: bi-directional data flow in same connection
- Flow and congestion control:
  - Flow control: sender will not overwhelm receiver
  - Congestion control: sender will not overwhelm the network
  - Sliding window flow control
  - Send and receive buffers
  - Congestion control done via adaptive flow control window size

The TCP Header

Fields enable the following:

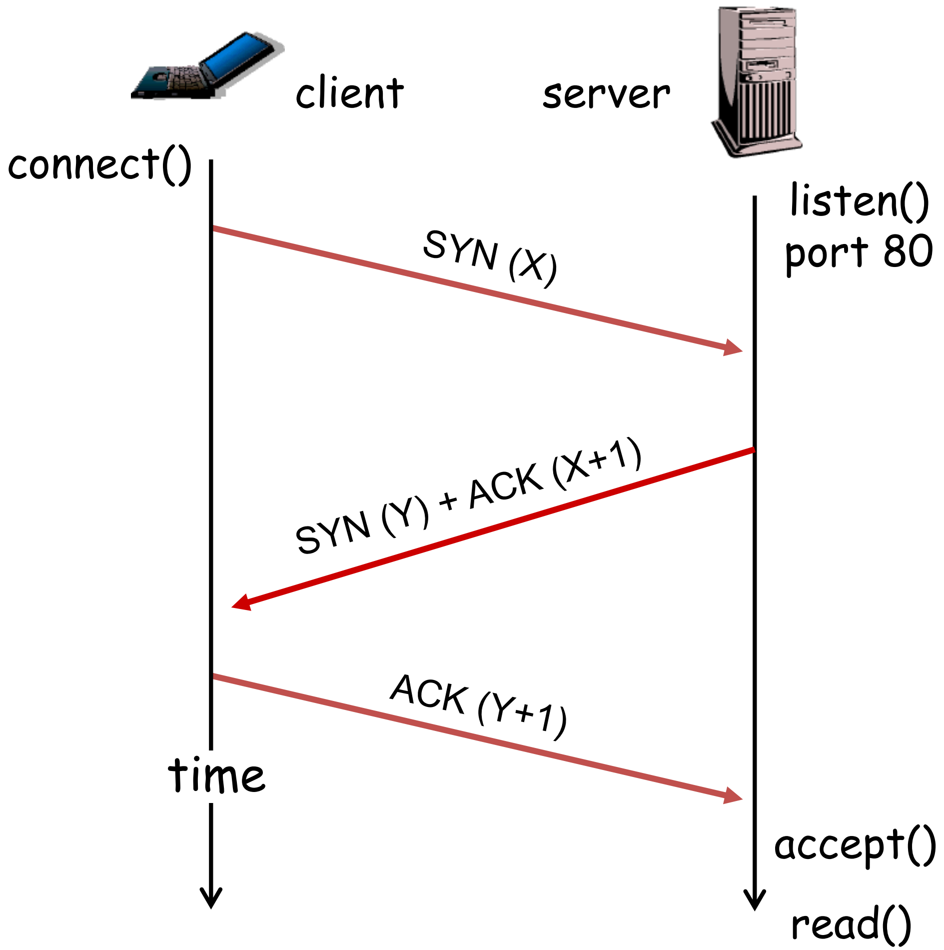
- Uniquely identifying each TCP connection

  (4-tuple: client IP and port, server IP and port)
- Identifying a byte range within that connection
- Checksum value to detect corruption
- Flags to identify protocol state transitions (SYN, FIN, RST)
- Informing other side of your state (ACK)

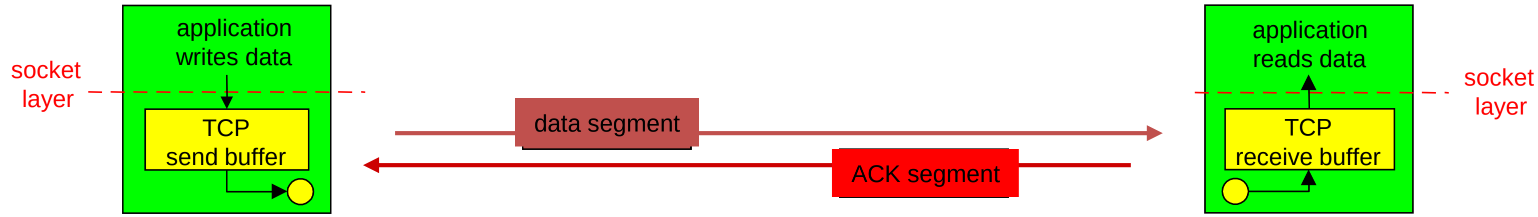


Establishing a TCP Connection

- Client sends SYN with initial sequence number (ISN = X)
- Server responds with its own SYN w/seq number Y and ACK of client ISN with X+1 (next expected byte)
- Client ACKs server's ISN with Y+1
- The ‘3-way handshake’
- X, Y randomly chosen
- All modulo 32-bit arithmetic



Sending Data



- Sender TCP passes segments to IP to transmit:
  - Keeps a copy in buffer at send side in case of loss
  - Called a “reliable byte stream” protocol
  - Sender must obey receiver advertised window
- Receiver sends acknowledgments (ACKs)
  - ACKs can be piggybacked on data going the other way
  - Protocol allows receiver to ACK every other packet in attempt to reduce ACK traffic (delayed ACKs)
  - Delay should not be more than 500 ms (typically 200 ms)
  - We’ll later see how this causes a few problems

Preventing Congestion

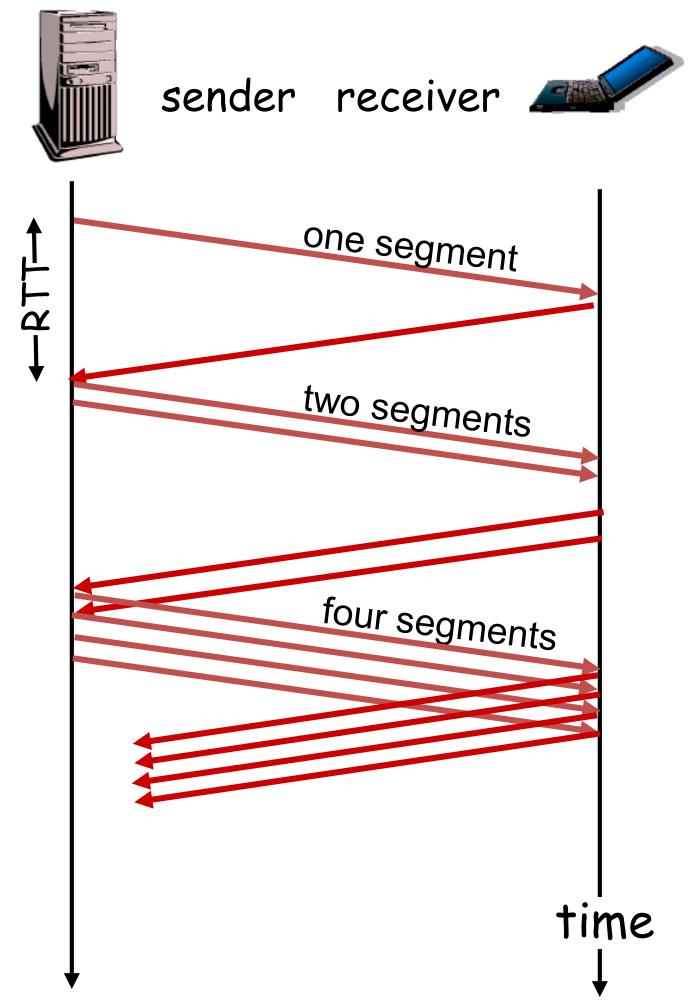
- Sender may not only overrun receiver, but may also overrun intermediate routers:
  - No way to explicitly know router buffer occupancy,
    so we need to infer it from packet losses
  - Assumption is that losses stem from congestion in the network (i.e., an intermediate router has no more buffers available)
- Sender maintains a congestion window (called cwnd or CW)
  - Never have more than CW of un-acknowledged data outstanding (or RWIN data; min of the two)
  - Successive ACKs from receiver cause CW to grow.
- How CW grows depends on which of 2 phases TCP is in:
  - Slow-start: initial state. Grows CW quickly (exponentially).
  - Congestion avoidance: steady-state. Grows CW slowly (linearly).
  - Switch between the two when CW > slow-start threshold

Congestion Control Principles

- Lack of congestion control would lead to congestion collapse (Jacobson 88).
- Idea is to be a “good network citizen”.
- Would like to transmit as fast as possible without loss.
- Probe network to find available bandwidth.
- In steady-state: linear increase in CW per RTT.
- After loss event: CW is halved.
- This general approach is called Additive Increase and Multiplicative Decrease (AIMD).
- Various papers on why AIMD leads to network stability.

Slow Start

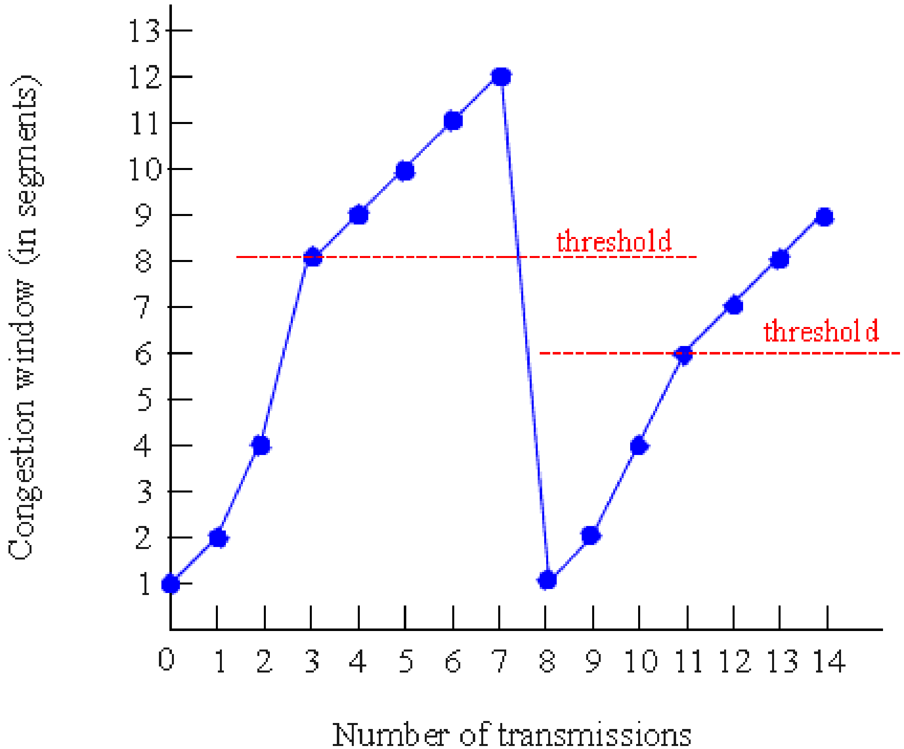
- Initial CW = 1.
- After each ACK, CW += 1;
- Continue until:
  - Loss occurs OR
  - CW > slow start threshold
- Then switch to congestion avoidance
- If we detect loss, cut CW in half
- Exponential increase in window size per RTT



Congestion Avoidance

Until (loss) {
 after CW packets ACKed:
  CW += 1;
}
ssthresh = CW/2;
Depending on loss type:
 SACK/Fast Retransmit:
  CW/= 2; continue;
 Course grained timeout:
  CW = 1; go to slow start.

(This is for TCP Reno/SACK: TCP Tahoe always sets CW=1 after a loss)



How are losses recovered?

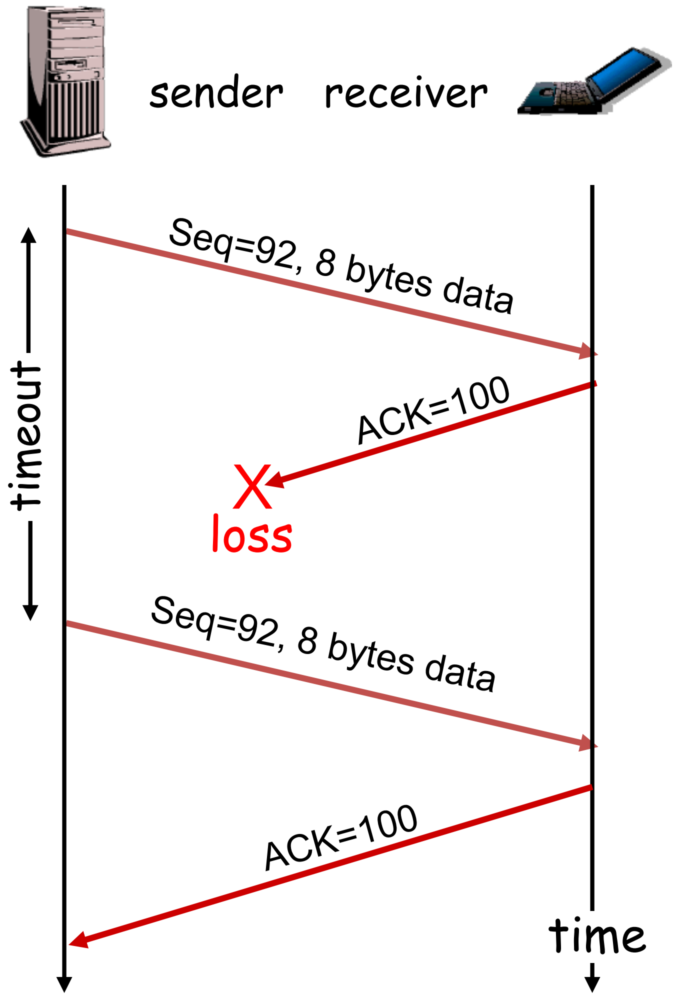
What if packet is lost (data or ACK!)

- Coarse-grained Timeout:
  - Sender does not receive ACK after some period of time
  - Event is called a retransmission time-out (RTO)
  - RTO value is based on estimated round-trip time (RTT)
  - RTT is adjusted over time using exponential weighted moving average:

    RTT = (1-x)*RTT + (x)*sample

    (x is typically 0.1)

  First done in TCP Tahoe

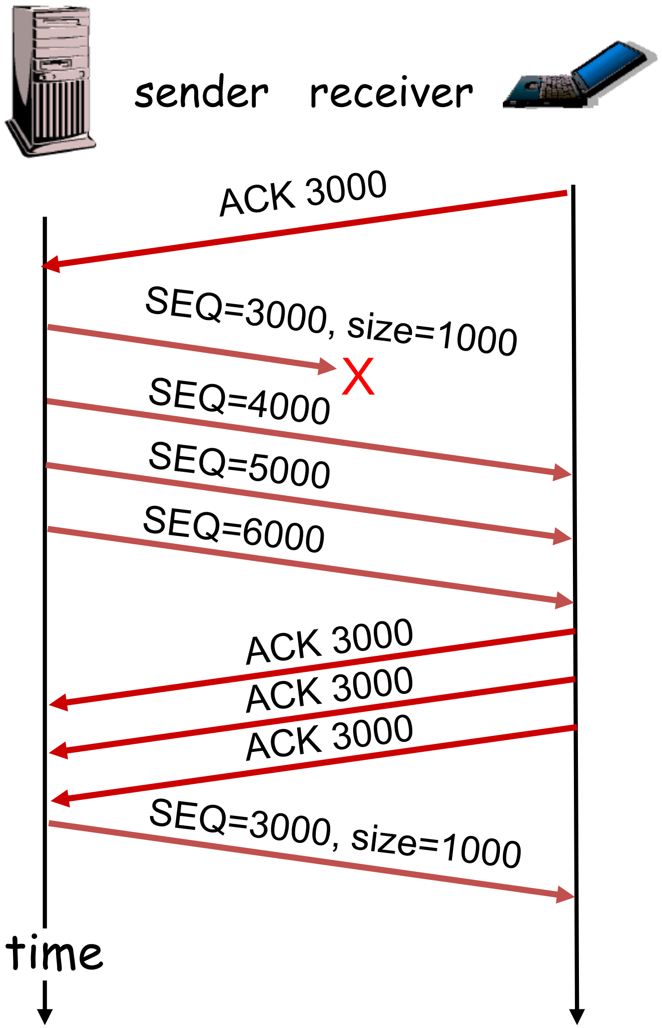


lost ACK scenario

Fast Retransmit

- Receiver expects N, gets N+1:
  - Immediately sends ACK(N)
  - This is called a duplicate ACK
  - Does NOT delay ACKs here!
  - Continue sending dup ACKs for each subsequent packet (not N)
- Sender gets 3 duplicate ACKs:
  - Infers N is lost and resends
  - 3 chosen so out-of-order packets don’t trigger Fast Retransmit accidentally
  - Called “fast” since we don’t need to wait for a full RTT

Introduced in TCP Reno

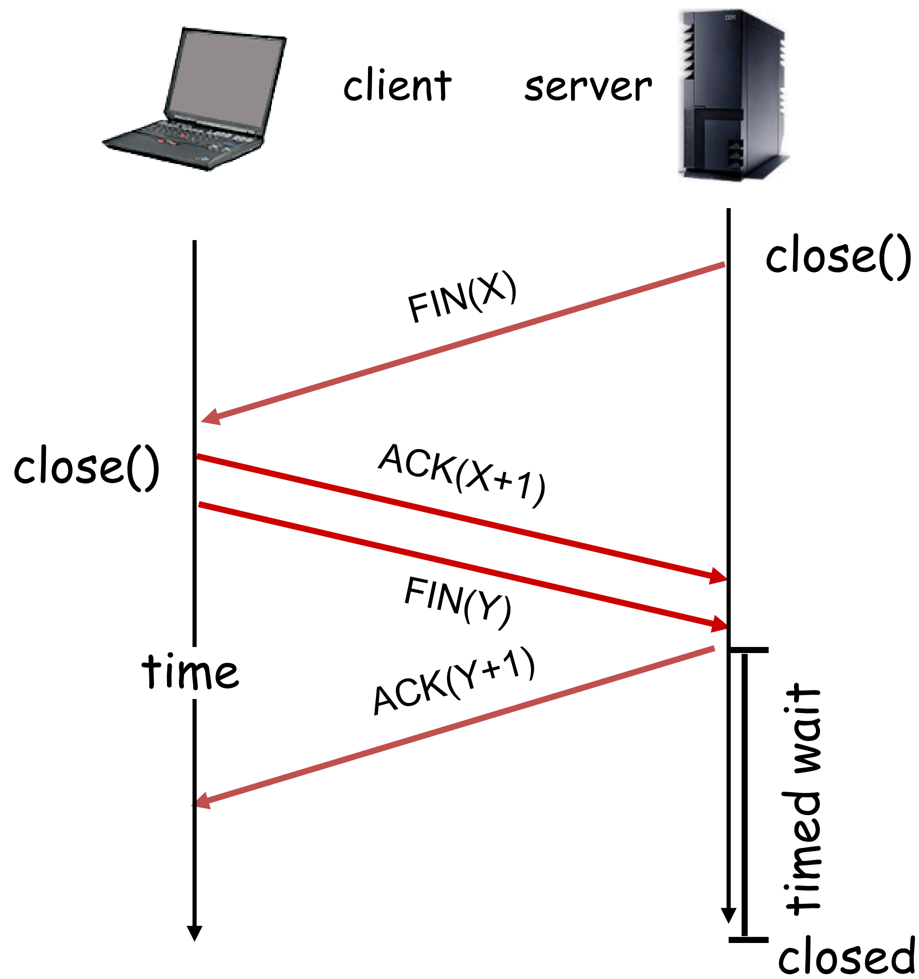


Other Loss Recovery Methods

- Selective Acknowledgements (SACK):
  - Returned ACKs contain option w/SACK block
  - Block says, "got up N-1 AND got N+1 through N+3"
  - A single ACK can generate a retransmission
- New Reno partial ACKs:
  - New ACK during fast retransmit may not ACK all outstanding data. Ex:
    - Have ACK of 1, waiting for 2-6, get 3 dup acks of 1
    - Retransmit 2, get ACK of 3, can now infer 4 lost as well
- Other schemes exist (e.g., Vegas)
- Reno has been prevalent; SACK now catching on

Connection Termination

- Either side may terminate a connection. (In fact, connection can stay half-closed.) Let's say the server closes (typical in WWW)
- Server sends FIN with seq Number (SN+1) (i.e., FIN is a byte in sequence)
- Client ACK's the FIN with SN+2 ("next expected")
- Client sends it's own FIN when ready
- Server ACK's client FIN as well with SN+1.

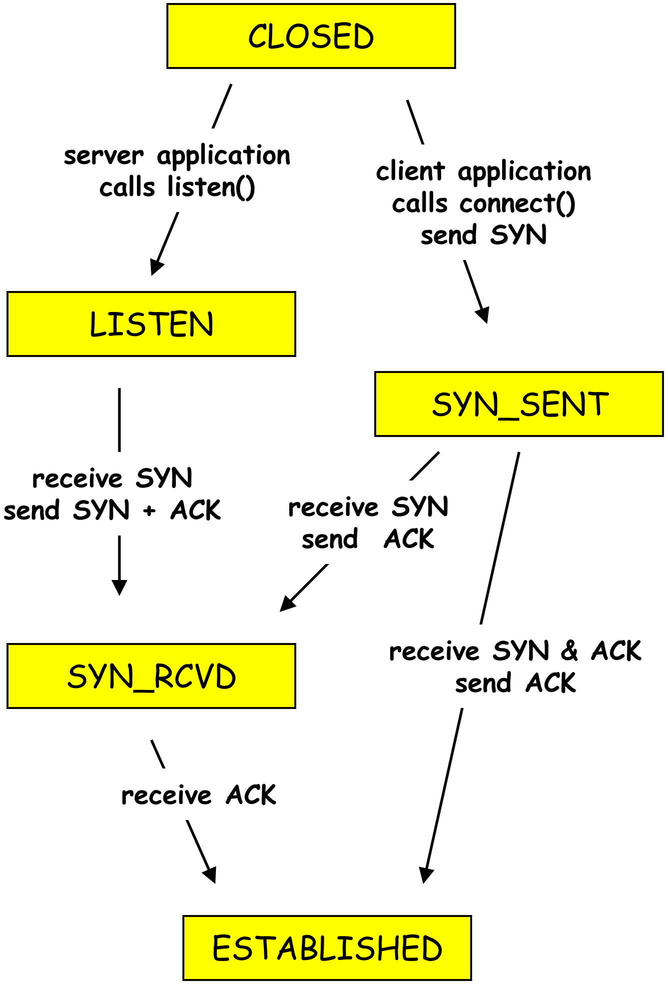


The TCP State Machine

- TCP uses a Finite State Machine, kept by each side of a connection, to keep track of what state a connection is in.
- State transitions reflect inherent races that can happen in the network, e.g., two FIN's passing each other in the network.
- Certain things can go wrong along the way, i.e., packets can be dropped or corrupted. In fact, machine is not perfect; certain problems can arise not anticipated in the original RFC.
- This is where timers will come in, which we will discuss more later.

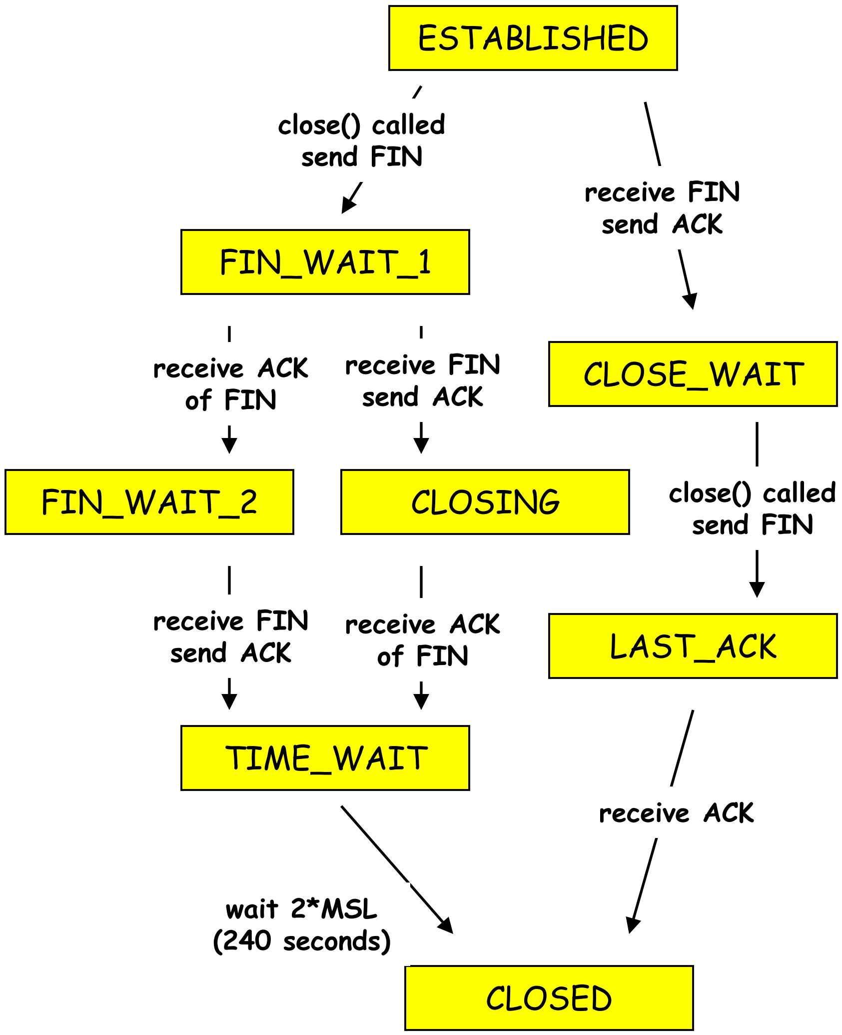
TCP Connection Establishment

- CLOSED: more implied than actual, i.e., no connection
- LISTEN: willing to receive connections (accept call)
- SYN-SENT: sent a SYN, waiting for SYN-ACK
- SYN-RECEIVED: received a SYN, waiting for an ACK of our SYN
- ESTABLISHED: connection ready for data transfer



TCP Connection Termination

- FIN-WAIT-1: we closed first, waiting for ACK of our FIN (active close)
- FIN-WAIT-2: we closed first, other side has ACKED our FIN, but not yet FIN'ed
- CLOSING: other side closed before it received our FIN
- TIME-WAIT: we closed, other side closed, got ACK of our FIN
- CLOSE-WAIT: other side sent FIN first, not us (passive close)
- LAST-ACK: other side sent FIN, then we did, now waiting for ACK



Summary: TCP Protocol

- Protocol provides reliability in face of complex and unpredictable network behavior
- Tries to trade off efficiency with being "good network citizen“ (i.e., fairness)
- Vast majority of bytes transferred on Internet today are TCP-based:
  - Web
  - Email
  - Peer-to-peer (Napster, Gnutella, FreeNet, KaZaa, BitTorrent)
  - Video streaming applications (Netflix, YouTube)
  - Online social networks (Facebook, Twitter)
  - Other emerging network applications