Computer Networks
ICS 651

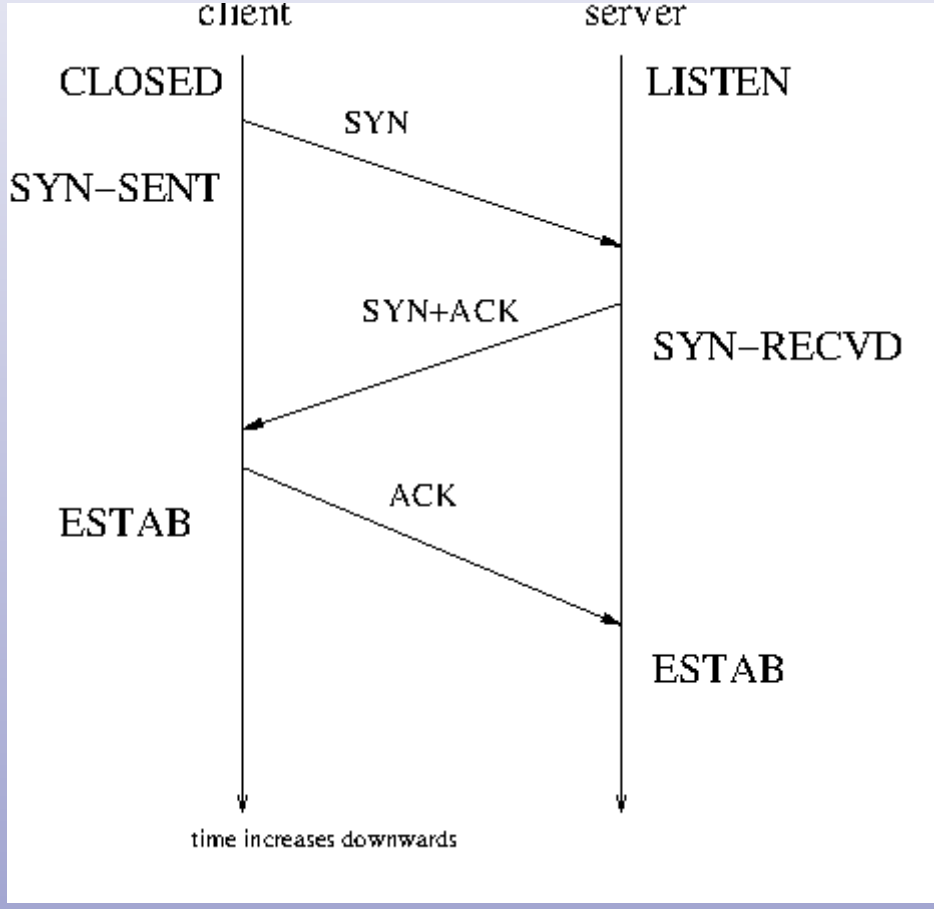
- TCP connection management
- TCP 3-way handshake
- TCP close
- TCP reset
- Reliable Transmission
- Sliding Window for Flow Control

TCP connection establishment

- when I receive a request to establish a connection, I must check:
  - that I don't already have this socket: one or more of the port numbers or IP numbers must differ from existing connections
  - that an application on my end desires to be connected
  - that I have sufficient resources to handle this connection
- the purpose of the connection establishment phase is to set up consistent connection state on the two peers

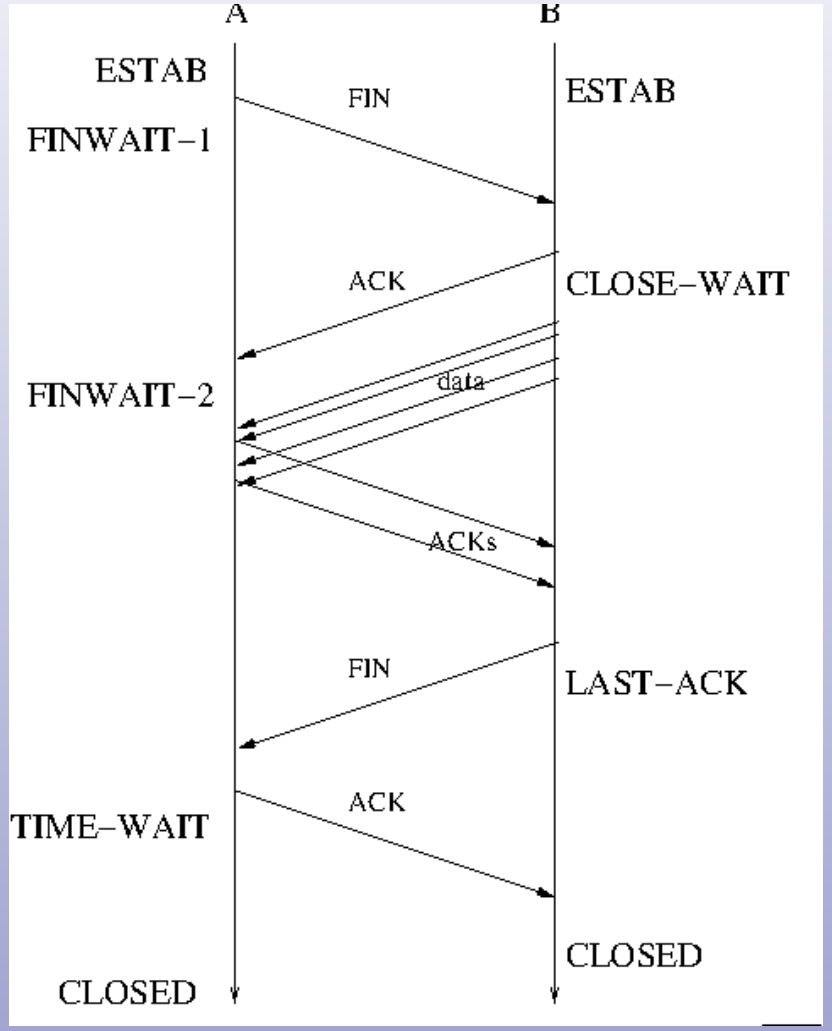
TCP 3-way handshake

- from state CLOSED:
  - send SYN, enter state SYN SENT
  - receive SYN and ACK, send ACK, enter state ESTAB, or
  - receive SYN, send ACK, enter state SYN RCVD, then proceed as below
- from state LISTEN:
  - receive SYN, send SYN and ACK, enter state SYN RCVD
  - receive ACK, enter state ESTAB
- retransmissions in case any of these are dropped
- see page 23 of RFC 793



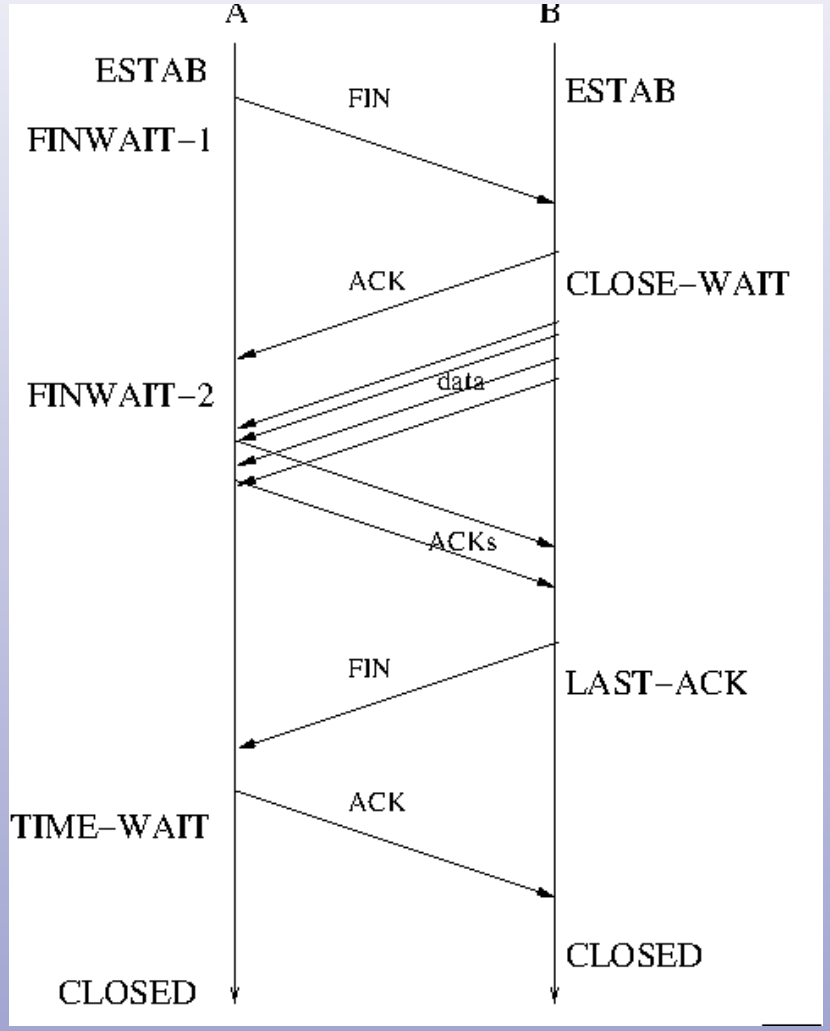
TCP close

- from state ESTAB:
  - receive FIN, send ACK, enter state CLOSE WAIT
  - application closes connection, send FIN, enter state LAST ACK
  - receive ACK, enter state CLOSED
- from state ESTAB:
  - application closes connection, send FIN, enter state FINWAIT-1
  - receive FIN, send ACK, enter state CLOSING
  - receive ACK, enter state TIME WAIT



TCP close, part 2

- from state FINWAIT-1, if we get an ACK:
  - receive ACK, enter state FINWAIT-2
  - receive FIN, send ACK, enter state TIME WAIT
- from state TIME WAIT, enter state closed after 4 minutes (2 maximum segment lifetimes):
  - last ack issue



TCP reset

- what should I do if I get a TCP segment for a connection that I have no record of?
- tell the sender to reset its connection
- If I am opening the connection and the segment I receive has an acknowledgement number I've never used, it might be an old segment. Again, reset the connection
- If the application program terminates, no sense in waiting for all the data to be delivered using the normal close

Reliable Transmission

- IP is not reliable: packets may be lost, reordered, or corrupted
- corruption is handled by the TCP checksum
- for packet loss or reordering, use acknowledgements:
  - each data packet must carry a sequence number
  - the sequence number tells us if data is out of order
  - for each data packet we receive, we send an empty (header only, no payload) acknowledgement packet stating that we got the data
  - if no ack is received within a certain time, we need to retransmit that packet

TCP sequence and acknowledgement numbers

- sequence and acknowledgement are 32-bit numbers
- carried in every packet (except only sequence number is valid in SYN)
- acks only sent in response to packets carrying data or SYN or FIN
- acks are cumulative: ack i acknowledges all the data up to sequence number i - 1
- sequence and ack numbers count bytes (+ SYN and FIN bits), not packets
  - useful e.g. with MTU discovery, retransmission of many small packets consolidated into one larger packet
- three duplicate acks are taken as a negative ack (NAK/NACK) inviting -- but not requiring -- retransmission

The need for flow control

- IP gives us ability to send end-to-end
- sequence and ack give us ability to reliably (modulo checksum) deliver data to application
- still missing: flow control, the ability to slow down the sender if the receiver is slow
- receiver buffers are limited
- why send if the receiver has no space?

TCP flow control

- with each ack, the receiver specifies how many more bytes past the ack the receiver is willing to accept
- the number of bytes is the **window**
- the window is **sliding** every time a new ack pushes its left edge to the right in sequence space
- the sender may have sent some of those bytes already -- because they were in the previous window
- when the sender exhaust the window, it must stop sending

TCP window

- the left edge of the TCP window is moved to the right by the receiver when it acknowledges new data
- the right edge of the TCP window is moved to the right by the receiver when it sends a window update
- the left edge of the TCP window is communicated to the sender by the acknowledgement number
- the right edge of the TCP window is communicated to the sender by the window field

TCP data transfer example

- A to B: seq 1, ack 1000000, window 100, 200 bytes of data
- A to B: seq 201, ack 1000000, window 100, 800 bytes of data
- B to A: seq 1000000, ack 201, window 1000, 0 bytes of data
- A to B: seq 1001, ack 1000000, window 100, 200 bytes of data

Zero windows

- suppose the window size is reduced to zero: receiver acks, and says it has a window size of zero, since all buffers are full
- later the receiver reopens the window by sending a new ack with a new window
- that ack is lost!
- deadlock: sender is waiting for window update, receiver is waiting for new data
- a sender with data to send, and a zero send window, must periodically send one byte past the window, to elicit an ack in response

Silly Window Syndrome

- receiver will move right edge of window by a small amount when it receives a small amount of data
- if sender is window-limited, it will then send a small segment
- this is undesirable, because small segments have more overhead, and because the receiver is likely to expand the window soon
- solution:
  - receiver must move right edge of window in increments of at least 1 MSS or 1/2 the actual window, whichever is less
  - sender increment its send window in increments of at least 1 MSS or 1/2 the *estimated* receiver's window, whichever is less
  - sender needs a timer so, when the timer expires, can send less than 1/2 the estimated window -- this avoids deadlock in case the estimate for the receiver's window is wrong (for example if the receiver's window got smaller)

Nagle algorithm

- two conflicting goals:
- send data as fast as possible, and
- don't send small packets, which have higher overhead (only significant if the network is busy/congested)
- solutions:
  - always send if the network is not busy -- send the first packet when all that was sent before has been acked
  - always send any maximum sized packet (either MSS, or at least 1/2 of the estimated receive window)
  - otherwise wait a little while to send, until either one of the above holds, or a timer expires

Nagle algorithm and sender SWS
from RFC 1122, section 4.2.3.4

- the usable window is the number of bytes I could send given the ack, window, and the next sequence number to send:

  U = ack + wnd - nxt

- D is the amount of new data I am ready to send
- send any time I can send a maximum-sized segment,

  min(D,U) >= MSS

- send if all sent data has been acked, i.e. there are no pending acks, and the window is large enough to send all the data I am ready to send

  nxt = ack and (D <= U or min(D, U) >= rcvbuff / 2)

  - the second condition (D <= U or min(D, U) >= rcvbuff / 2) avoids SWS on fast connections
  - because if D > U, the amount of data sent will be determined by the window size U, and that may lead to SWS

- or send if the override timeout expires
  - this timer may be combined with the timer for zero windows