# Transport Layer

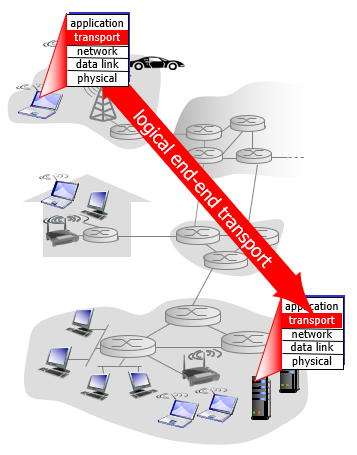
## Intro to Transport Layer

Our objectives:

* Understand principles behind transport layer services:
  + Multiplexing
  + Demultiplexing
* Learn about transport layer protocols:
  + TCP: connection-oriented reliable transport
  + UDP: connectionless transport

## Transport Layer: services and protocols

* The transport layer is layer 4 of the internet model.
  + runs on *end systems*, not network routers
  + provides *logical* (instead of physical) comms between app processes running on different hosts
  + responsible for moving messages from end to end in a network.
* The transport layer performs the following functions…
  + Basically, it links the app layer to the network.
  + On sender’s side: breaks app msgs into small packets for transmission (**segmenting**); passes to network layer
  + In between: responsible for establishing end-to-end connection between sender and receiver (**session management**)
  + On receiver’s side: reassembles segments into msgs, passes to app layer
* More than one transport protocol available to apps:
  + TCP: connection-oriented reliable transport
  + UDP: connectionless transport; not reliable; used for speedy data transfer where packet loss is not a problem
  + Internet uses TCP/IP, which is most widely used transport + network protocol

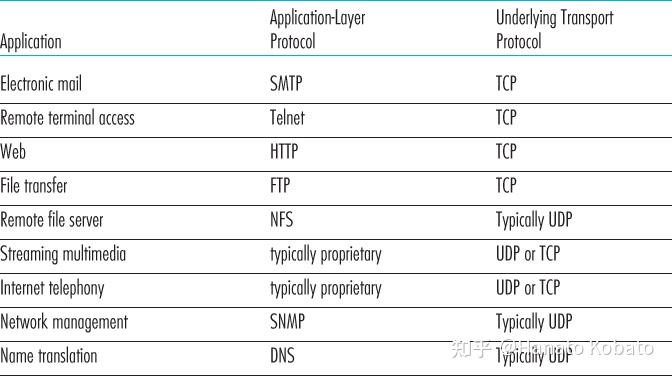


## Transport Layer vs Network Layer

* Network layer: logical communication between ***hosts***
* Transport layer: logical communication between ***processes***
  + Relies on and enhances network layer services
* Household analogy: 12 kids in Ann’s house sending letters to 12 kids in Bill’s house
  + Hosts = houses
  + Processes = kids
  + App messages = letters in envelopes
  + Transport protocol = Ann and Bill, the parents, who demux (or distribute the messages) to their kids
  + Network layer protocol = postal service

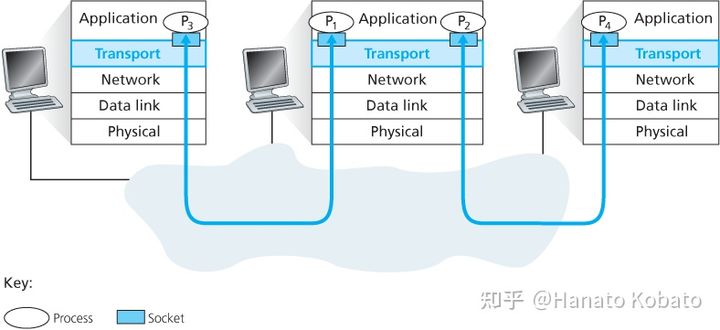
## Internet transport-layer protocols

* Internet makes two distinct transport-layer protocols available to the app layer
  + UDP (User Datagram Protocol)
    - Unreliable, unordered delivery (i.e. does not send segments in the right order and is unreliable; segments can get lost)
    - No-frills extension of “best-effort” IP (IP makes “best effort” to deliver segments between hosts, but makes *no guarantees*, as does UDP)
  + TCP (Transmission Control Protocol)
    - Reliable, in-order delivery (data is delivered from sending process to receiving process, correctly and in order)
    - Congestion control (tries to give each connection traversing a congested link an equal share of link bandwidth)
    - Connection setup

  
*Popular Internet applications and their underlying transport protocols*

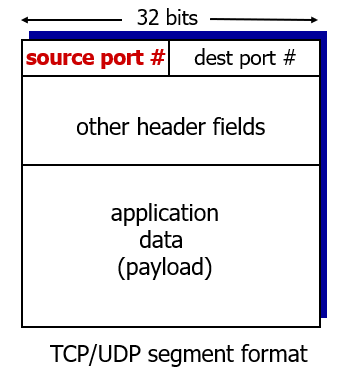
## Multiplexing and Demultiplexing

* Recall that a process can have one or more **sockets**, through which data passes from process to network and from network to process.
* **Multiplexing** is a process on the sender’s side that facilitates message transfer…
  + gathers data chunks at source host from different sockets
  + encapsulates each data chunk with header info (later used in demultiplexing) to create segments
  + passes segments to network layer
* Transport layer in receiving host does not deliver data directly to a process, but to one of potentially many intermediary sockets (hence each socket has unique identifier).
* **Demultiplexing** is the reverse process on the receiver’s side
  + Transport layer examines segment headers for socket info
  + Once info found, directs segment to that socket

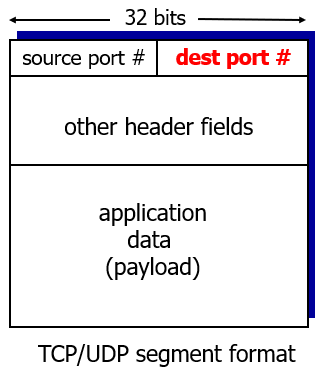


* Note that transport layer in middle host also has work to do…
  + needs to demux segments arriving from network layer below to either process P­1 or P­2 (by directing arriving segment data to corresponding process socket)
  + must also gather outgoing data from these sockets, form transport-layer segments, and pass them down to network layer for transmission to receiver

## How multiplexing works

* We already know that transport-layer multiplexing requires…
  + that sockets have unique identifiers (**source port number field**)
  + that each segment has special fields that indicate the destination socket (**destination port number field**)
* Each port is a 16 bit number, ranging from 0-65535.
  + The port numbers, 0-1023, are well-known and are restricted for use by well known application protocols such as HTTP (port 80) and FTP (port 21).

## How demultiplexing works

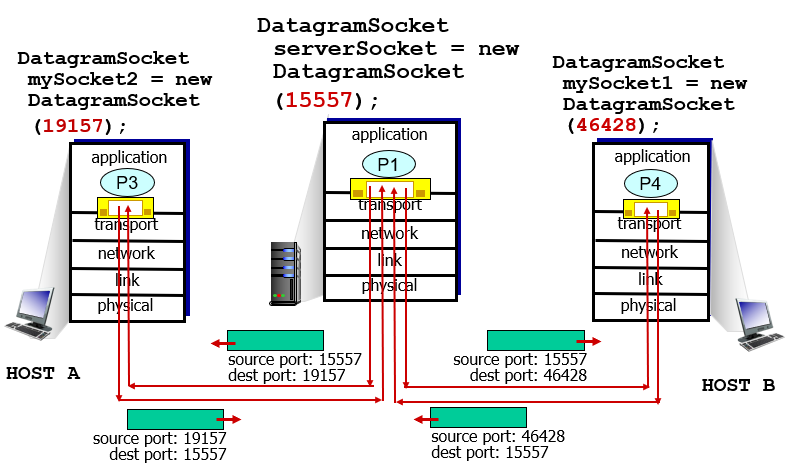
* Host receives IP datagrams…
  + each datagram has source and destination IP addresses
  + each datagram carries one transport-layer segment
  + each segment has source, destination port number
* Host uses *IP addresses* and *port numbers* to direct segment to correct socket.

## Connectionless Demux (UDP)

* UDP utilizes above-described method of mux/demux to transfer messages, with port numbers assigned to UDP sockets.
* UDP socket is fully identified by a **two-tuple** (finite sequence of two elements), consisting of destination IP address and port number.
  + Thus, if 2 UDP segments have different *source* IP addresses and port numbers, but have the same *destination* IP address and port number, then the 2 segments will be directed to same destination process via same destination socket
* Server host may support many simultaneous UDP sockets, so each socket is identified by its own 2-tuple identifier
* What is the purpose of the source IP address and port number?
  + Acts as a “return address” for when receiver wants to send a segment back to the source.
  + In the example, when host B wants to send a segment back to host A, the destination port in the B-to-A segment will take its value from the source port value of the A-to-B segment. (The complete return address is A's IP address and the source port number.)

## Connectionless Demux: Example

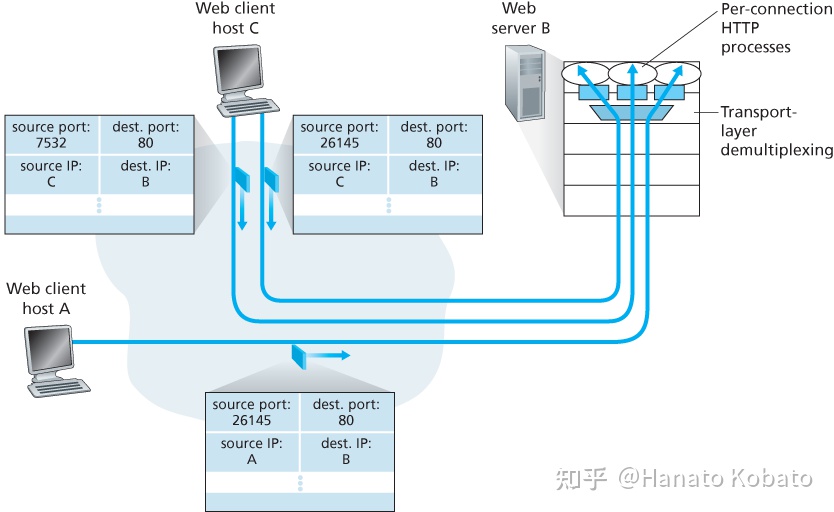
* Suppose a process in Host A (H-A), with UDP port 19157, wants to send a chunk of app data to a process with UDP port 46428 in Host B. How?
  + Transport layer in H-A creates a transport-layer segment that includes app data, source port #19157, destination port #46428, and two other values (length and checksum; described later)
  + Transport layer (in H-A) then passes the resulting segment to network layer
  + Network layer (H-A) encapsulates segment in IP datagram and makes a best-effort attempt to deliver segment to receiving host (H-B).
  + If segment arrives at receiving H-B, transport layer (H-B) examines destination port # (i.e. 46428) )in the segment and delivers segment to socket identified by port 46428

****

## Connection-oriented Demux

* TCP socket identified by 4-tuple
  + Source IP address
  + Source port #
  + Destination IP address
  + Destination port #
* Demux: receivers uses all 4 values to direct segment to correct socket
* Server host may support many simultaneous TCP sockets
  + each socket identified by its own 4-tuple
* Works a bit complexly for web servers…
  + web servers have different sockets for each connecting client
  + non-persistent HTTP will have different socket for each request
  + frequent creating and closing of sockets can impact server performance
* In the example, notice that Web server B opens up a new process for each connection: two separate sockets+processes for the two requests from web client host C, and one for the sole request from host A.

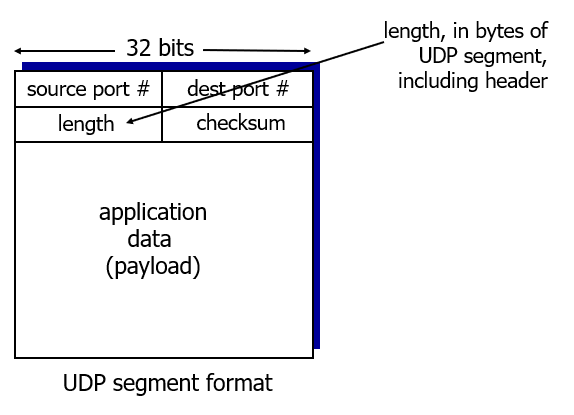
## Connection-oriented Demux: Example



## UDP: User Datagram Protocol

* “no frills,” “bare bones” Internet transport protocol
* “best effort” service, UDP segments may be: (1) lost, or (2) delivered out-of-order
* UDP said to be *“*connectionless*”*:
  + no handshaking between UDP sender, receiver
  + each UDP segment handled independently of others
* Why is there UDP then?
  + Finer app-layer control over what data is sent and when.
  + No connection establishment (which can add delay)
  + Simple: no connection state at sender, receiver
  + Small packet header overhead: 8 bits for UDP vs 20 bits for TCP
  + no congestion control: UDP can blast away as fast as desired
* Uses of UDP:
  + Streaming multimedia apps; e.g. vid conferencing (loss tolerant, rate sensitive)
  + domain querying; Domain Name Server (DNS)
  + network management; Simple Network Management Protocol (SNMP)
  + Control messages such as addressing; Dynamic Host Config Protocol (DHCP)
  + Routing control messages; Routing Info Protocol (RIP)
* How to make UDP reliable?
  + add reliability at application layer
  + application-specific error recovery!

## UDP: segment structure

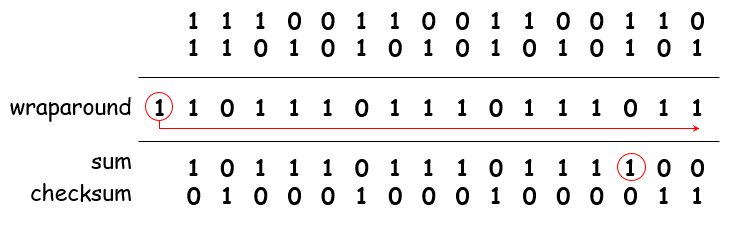


## UDP checksum

* Goal: detect “errors” (e.g., flipped bits) in transmitted segment
* Sender:
  + treat segment contents, including header fields, as sequence of 16-bit integers
  + checksum: addition (one’s complement sum) of segment contents
  + sender puts checksum value into UDP checksum field
* Receiver:
  + compute checksum of received segment
  + check if computed checksum equals checksum field value:
  + NO - error detected
  + YES - no error detected.
  + NOTE: *But maybe errors, nonetheless?* (remember the carryout 😊)

## Internet checksum: example

* Example of internet checksum: add two 16-bit integers

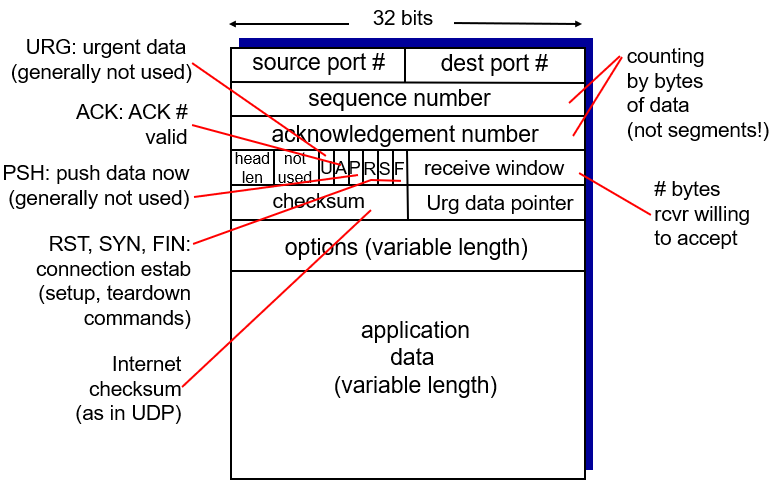


* Note: When adding binary numbers, a carryout from the most significant bit needs to be added to the result‼
* Checksum is the complement/opposite of the sum (0’s to 1’s and 1’s to 0’s)

## TCP: Overview

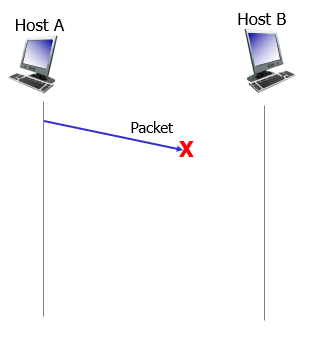
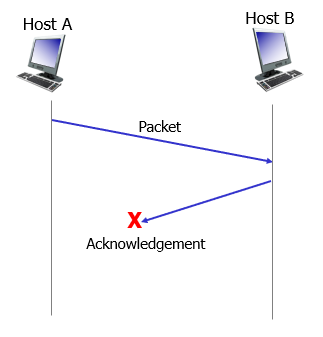
* point-to-point: one sender, one receiver
* reliable, in-order *byte stream:* no “message boundaries”
* pipelined: TCP congestion and flow control set window size
* full duplex data:
  + bi-directional data flow in same connection
  + MSS: maximum segment size
* connection-oriented: handshaking (exchange of control msgs) initializes sender, receiver state before data exchange
* flow controlled: sender will not overwhelm receiver

**TCP segment structure**



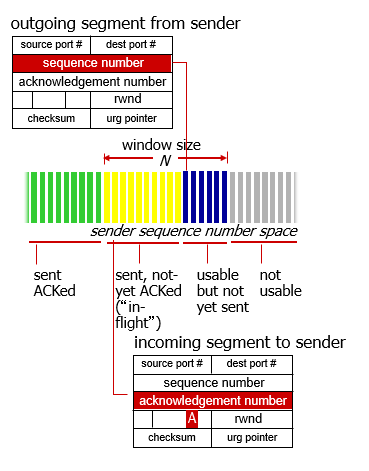
## Reliable data transfer (RDT)

* One of the biggest issues in networking
  + Packets can get lost
  + Data can become corrupted
* TCP counteracts these problems with the following features…
  + Sequence number
  + Acknowledgment
  + Retransmission
  + Timer

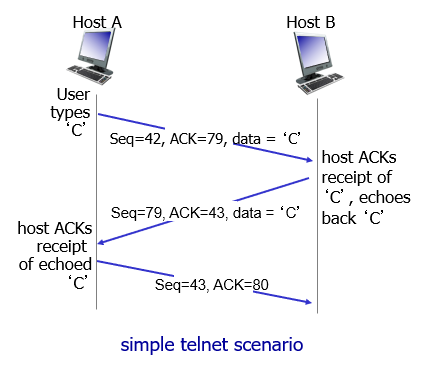
 

* Simplest model of rdt is that underlying channel is supposedly reliable (**rdt 1.0**)
  + Sender: waits for call from above (i.e. app layer) to send data to network layer in packets
  + Receiver: waits for call from below (i.e. network layer) to extract received packet and deliver data to app layer
  + All packet flow from sender to receiver; no differences in data and packets
* A more realistic model of the underlying channel is one in which bits in a packet may be corrupted (**rdt 2.0**). Such bit errors occur in physical network components as packet is transmitted, propagates, or is buffered.
* Before going through, consider how people might communicate reliably over a communication channel.
  + If a message is clearly heard and understood, message taker will say “OK”
  + If message taker hears a garbled message, will say “repeat that please”
  + Note that this message-diction protocol uses both positive acknowledgements (“OK”) and negative acknowledgements (“Please repeat that”)
  + This allows receiver to let sender know what was received correctly, and also what was not received correctly and needs to be resent

## TCP seq. numbers, acknowledgement (ACK)

* In a computer network setting, reliable data transfer protocols based on such retransmission are known as **ARQ (Automatic Repeat reQuest)** protocols.   
  However, 3 more protocol capabilities required to handle bit errors:
  + Error detection: e.g. the Internet checksum field
  + Receiver feedback: only way for sender to know that receiver got packet correctly is for receiver to provide positive (ACK) or negative (NAK) feedback to sender.
  + Retransmission: A packet received in error at receiver will be resent by sender
* Still one problem: the ACK or NAK packets may get corrupted; sender has no way of knowing if receiver correctly received last piece of transmitted data‼!
  + Solution: Add new field to data packet and have the sender *number* its data packets by putting a **sequence number** in this field. That way, receiver knows if this is a retransmission or not!
* sequence numbers:
  + byte stream “number” of first byte in segment’s data
* acknowledgements:
  + seq # of next byte expected from other side
  + cumulative ACK
* Q: how does receiver handles out-of-order segments?
* A: TCP spec doesn’t say, it is up to implementor

## TCP seq. numbers, ACKs: example



## TCP sender events

1. data received from app:

* create segment with seq #
* seq # is byte-stream number of first data byte in segment
* start timer if not already running
  + think of timer as for oldest unACKed segment
  + expiration interval: **TimeOutInterval**

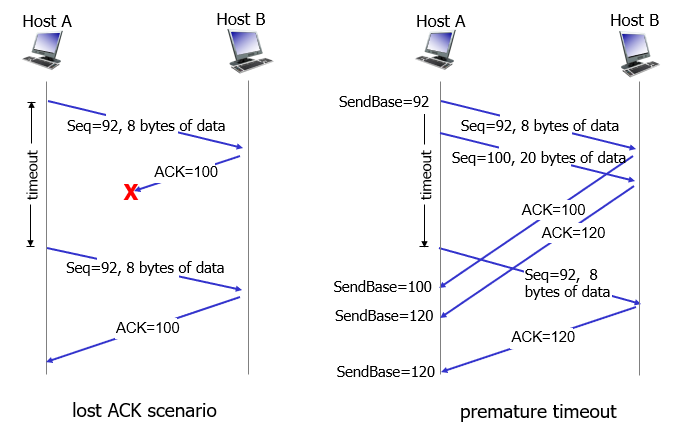
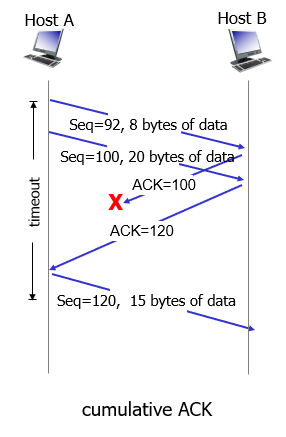
1. timeout:

* retransmit segment that caused timeout
* restart timer

1. acknowledgement received:

* if ACK acknowledges previously unACKed segments
  + update what is known to be ACKed
  + start timer if there are still unACKed segments

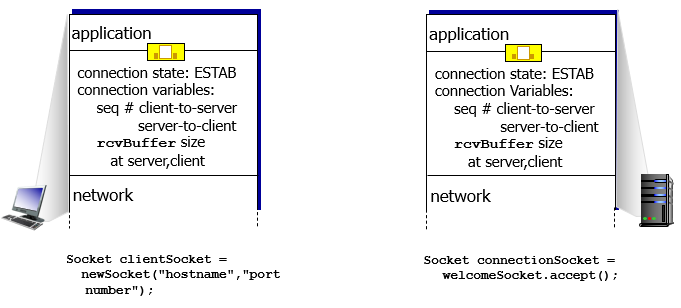
## TCP: retransmission scenarios

## Connection Management

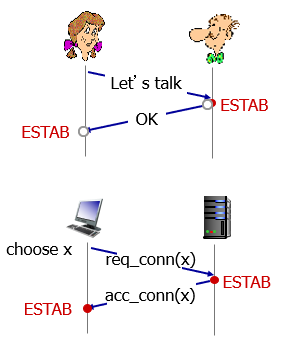
Before exchanging data, sender/receiver “handshake”:

* agree to establish connection (each knowing the other willing to establish connection)
* agree on connection parameters



## Agreeing to establish a connection

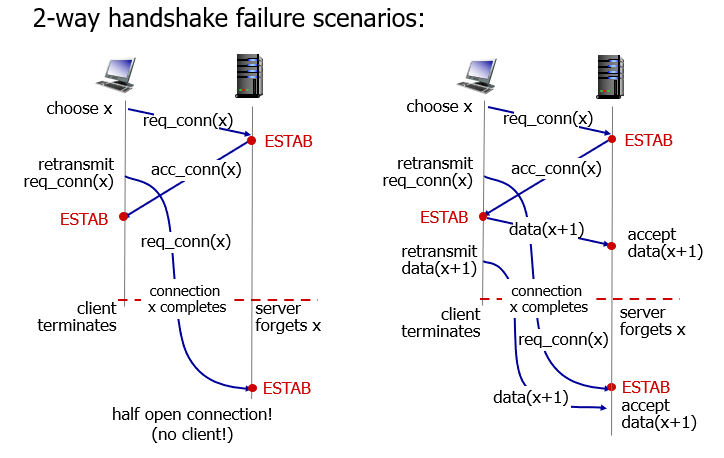
* 2-way handshake looks like this:



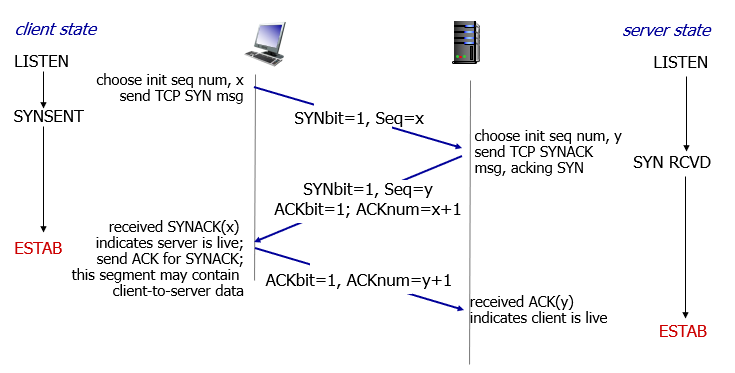
* Q: Will 2-way handshake always work in network?
* A: nope, there are multiple issues here…
  + Variable delays
  + Retransmitted messages [e.g. req\_conn(x)] due to message loss
  + Message reordering
  + Can’t “see” other side

## Agreeing to establish a connection: scenarios

* 2-way handshake failure scenarios:

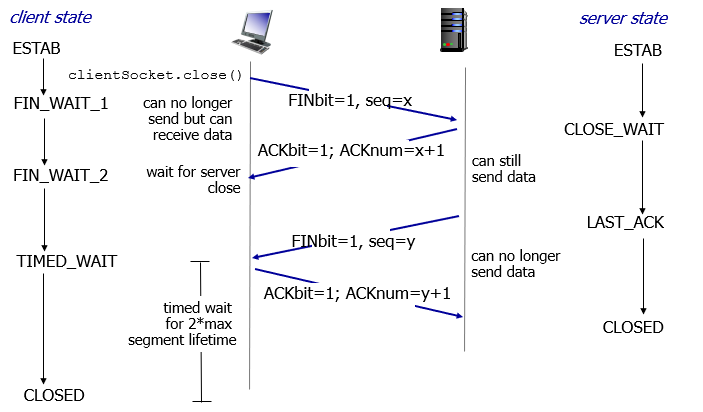


## TCP 3-way Handshake



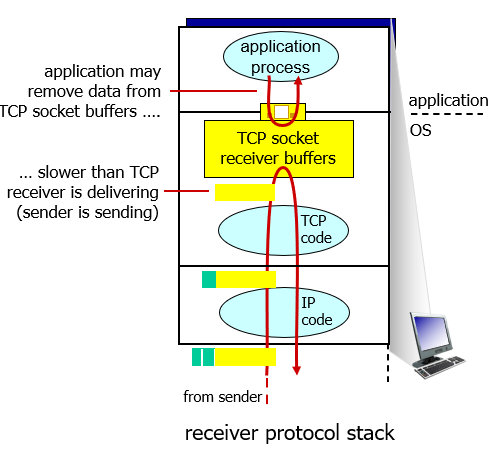
## TCP: closing a connection

* client, server each close their side of connection
  + send TCP segment with FIN bit = 1
* respond to received FIN with ACK
  + on receiving FIN, ACK can be combined with own FIN
* simultaneous FIN exchanges can be handled

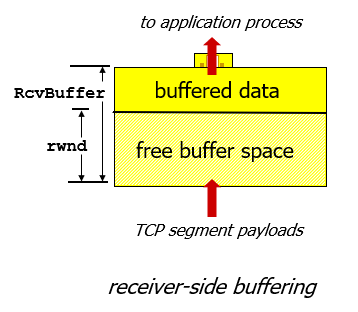


## TCP flow control

* receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast

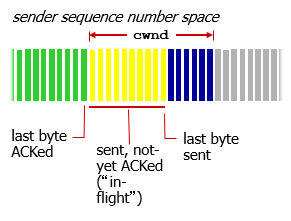


* receiver “advertises” free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  + **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  + many operating systems auto adjust **RcvBuffer**
* sender limits amount of unACKed (“in-flight”) data to receiver’s **rwnd** value
* guarantees receive buffer will not overflow



## TCP congestion control

* informally: “too many *sources* sending too much data too fast for *network* to handle”
* different from flow control!
* manifestations:
  + lost packets (buffer overflow at routers)
  + long delays (queueing in router buffers)
* a top-10 problem!
* sender limits transmission:



* **cwnd** is dynamic, function of perceived network congestion

## TCP congestion control: in action

