Transport Layer: TCP/UDP

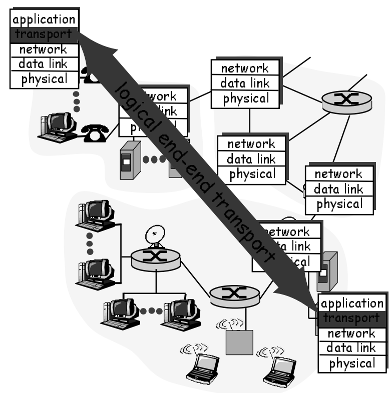
Chapter 24, 16

Transport Layer

- Purpose of transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Connection-less transport: UDP
- Connection-oriented transport: TCP
  - reliable transfer
  - flow control
  - connection management

Transport services and protocols

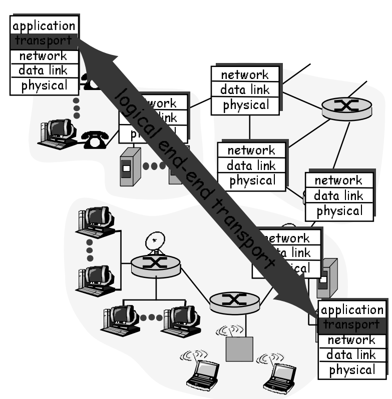
- provide logical communication between application processes running on different hosts
- transport protocols run in end systems via software
- transport vs network layer services:
- network layer: data transfer between end systems
- transport layer: data transfer between processes
  - relies on, enhances, network layer services



Transport-layer protocols

Internet transport services:

- reliable, in-order unicast delivery (TCP)
  - congestion
  - flow control
  - connection setup
- unreliable ("best-effort"), unordered unicast or multicast delivery: UDP
- services not available:
  - real-time
  - bandwidth guarantees
  - reliable multicast

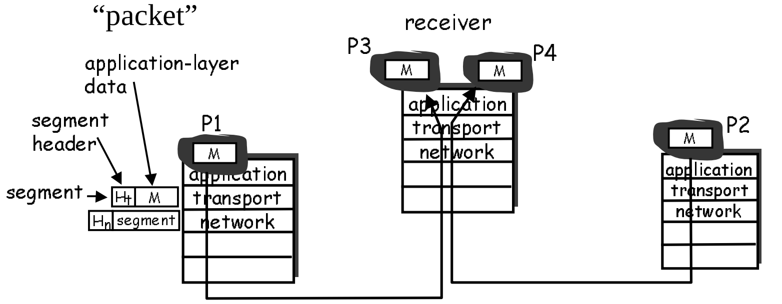


Multiplexing/demultiplexing

Recall: segment - unit of data exchanged between transport layer entities

– aka TPDU: transport protocol data unit or "packet"

Demultiplexing: delivering received segments to correct app layer processes

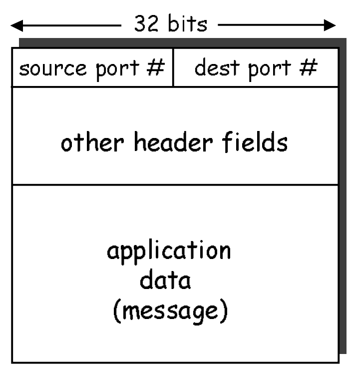


Multiplexing/demultiplexing

Multiplexing:
gathering data from multiple app processes, enveloping data with header (later used for demultiplexing)

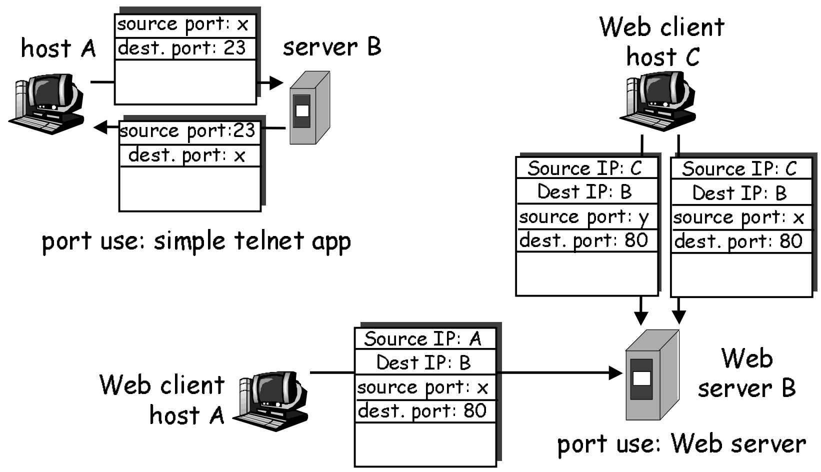
multiplexing/demultiplexing:

- based on sender, receiver port numbers, IP addresses
  - source, dest port #s in each segment
  - recall: well-known port numbers for specific applications



TCP/UDP segment format

Multiplexing/demultiplexing: examples



UDP: User Datagram Protocol [RFC 768]

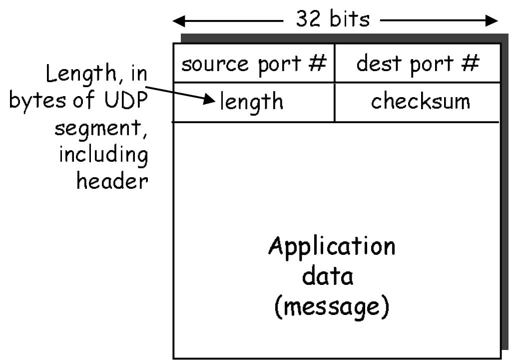
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP: more

- often used for streaming multimedia apps
  - Controversial: no congestion control
- other UDP uses (why?):
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recover!



UDP segment format

UDP checksum

Goal: detect “errors” (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’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
    - Toss data OR
    - Pass to app with warning
  - YES - no error detected.

Connection Oriented Transport Protocol Mechanisms

- Properties of connection-oriented Transport Protocols:
  - Logical connection
  - Establishment
  - Maintenance termination
  - Reliable
  - e.g. TCP

Connection-Oriented Transport via Reliable Network Layer

- Transport Layer Services like TCP are complicated – to start, let's first assume we are working with a reliable network layer service
  - e.g. reliable packet switched network using X.25
  - e.g. frame relay using LAPF control protocol
  - e.g. IEEE 802.3 using connection oriented LLC service
  - NOT IP!   IP is unreliable
- Assume arbitrary length message
- Transport service is end to end protocol between two systems on same network

Issues in a Simple Transprot Protocol

- If we have a reliable network layer, then the transport layer must consider:
  - Addressing
  - Multiplexing
  - Flow Control
  - Connection establishment and termination

Addressing

- Target user specified by:
  - User identification
    - Usually host, port
      - Called a socket in TCP/UDP
    - Port represents a particular transport service (TS), e.g. HTTPD
  - Transport protocol identification
    - Generally only one per host
    - If more than one, then usually one of each type
      - Specify transport protocol (TCP, UDP)
  - Host address
    - An attached network device
    - In an internet, a global internet address (IP Address)
    - A well-known address or lookup via name server

Multiplexing

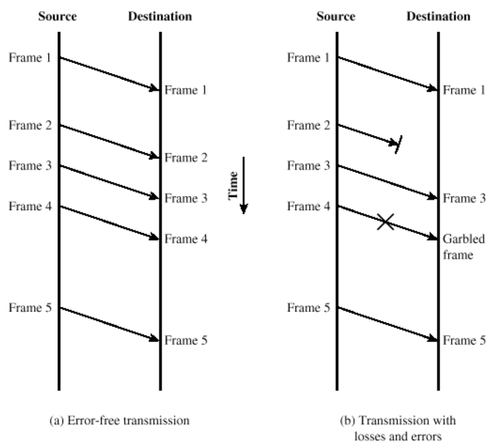
- Multiple users employ same transport protocol
- User identified by port number or service access point (SAP)
- Described previously

Flow Control

- Can be difficult than flow control at the data link layer – data is likely traveling across many networks, not one network. Some potential problems:
  - Longer transmission delay between transport entities compared with actual transmission time
    - Delay in communication of flow control info
  - Variable transmission delay
    - Difficult to use timeouts
- Flow may be controlled because:
  - The receiving user cannot keep up
  - The receiving transport entity cannot keep up
  - If either happens, the results is a buffer that can get full and eventually lose data

Model of Frame Transmission

Diagram for Frame/Packet Transmission



We'll use this model to discuss flow control issues

Coping with Flow Control Requirements (1)

- Do nothing
  - Segments that overflow are discarded
  - Sending transport entity will fail to get ACK and will retransmit
    - Thus further adding to incoming data and could exacerbate the flow control problem
- Refuse further segments from network layer
  - Clumsy
  - Multiplexed connections are controlled on aggregate flow

Coping with Flow Control Requirements (2)

- One protocol: Stop-and-Wait
  - Sender must wait for recipient to send ACK before sending the next packet
    - Not very efficient usage of the network, only one outstanding message can be in transit at a time
  - Works well on reliable network
    - Failure to receive ACK is taken as flow control indication
  - Does not work well on unreliable network
    - Cannot distinguish between lost segment and flow control

Coping with Flow Control

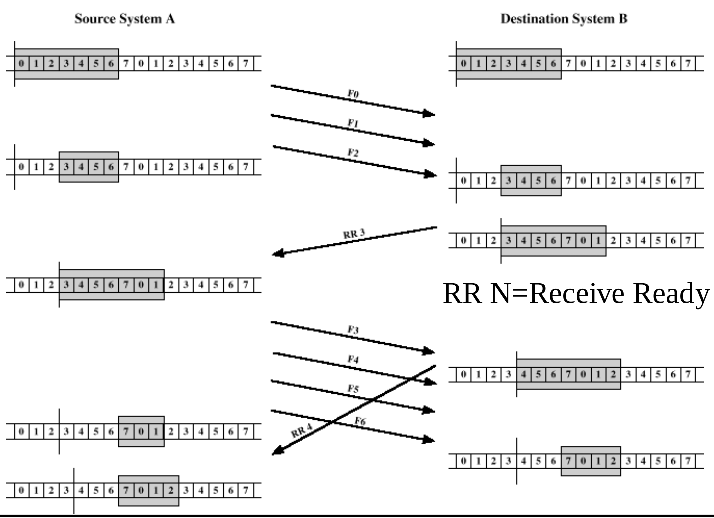
- Credit-Based Scheme
  - Credit = How much data sender can transmit
    - Sliding window idea, sender can send a number of frames up to the window size
    - Receiver sends single ACK that acknowledges all previous frames
    - Window size varies based on credit available
    - Receiver can control credit of the sender
      - In acknowledgement, receiver could change the window size
  - Advantages
    - Better network usage, allows outstanding messages to be in transit than Stop-And-Wait
    - More effective on unreliable network
      - Decouples flow control from ACK
    - May ACK without granting credit and vice versa
  - Each octet has sequence number
  - Each transport segment has a sequence number, acknowledgement number and window size in header

Sliding Window Enhancements

- Receiver can acknowledge frames without permitting further transmission (Receive Not Ready)
- Must send a normal acknowledge to resume
- If full duplex two-way communications, we need two windows: one for transmit and one for receive
  - Piggybacking – if sending data and acknowledgement frame, combine together
- More efficient than stop-and-wait since many frames may be in the pipeline

Example Sliding Window

Fixed Window Size



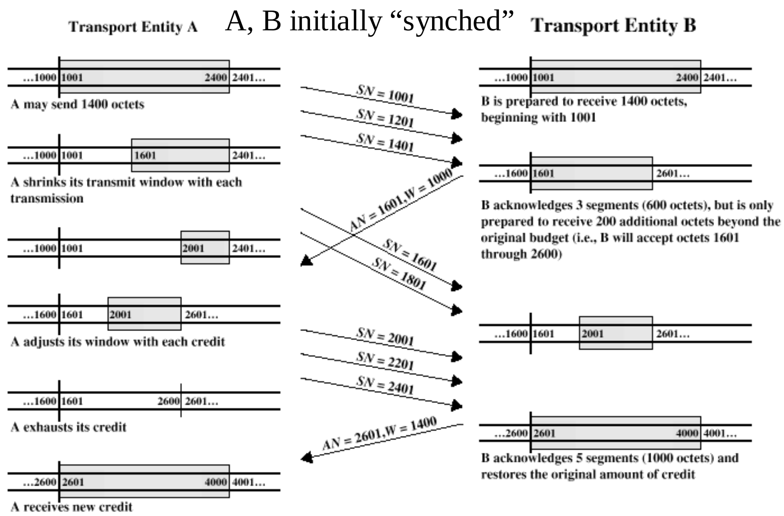
RR N=Receive Ready on N

Use of Header Fields

- For credit-based window size
  - When sending, Sequence Number is that of first octet in segment
  - ACK includes AN=i (Acknowledgement Number), W=j (Window Size)
  - All octets through SN=i-1 acknowledged
    - Next expected octet is i
  - Permission to send additional window of W=j octets
    - i.e. octets through i+j-1

Credit Allocation Example

A, B initially “synched”

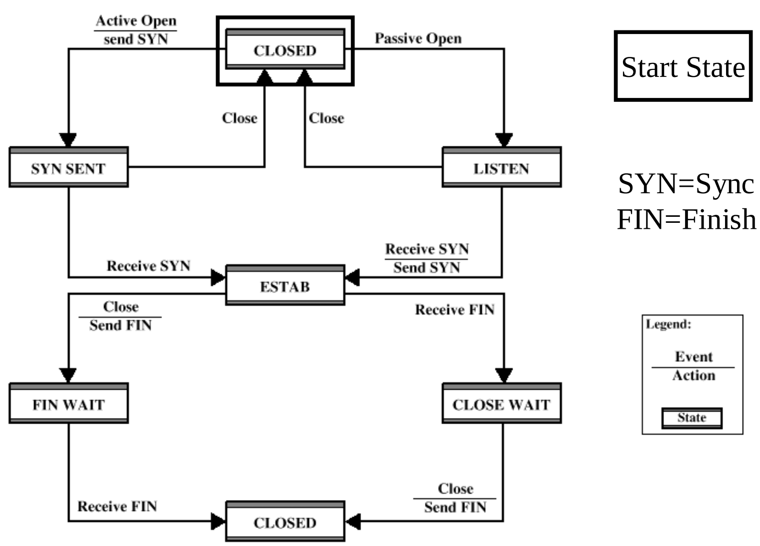


200 octets per segment

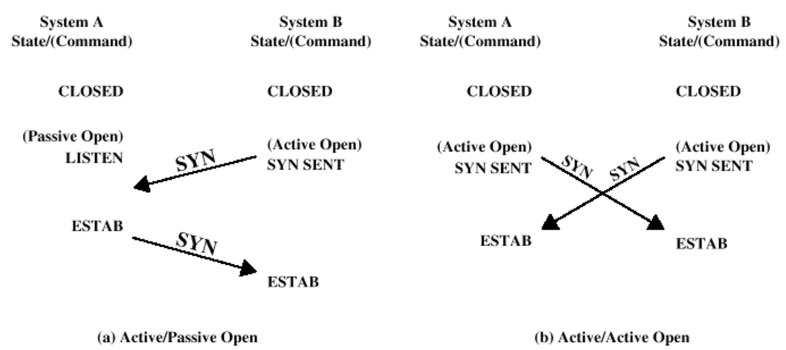
Establishment and Termination

- Even with a reliable network service, both ends need to “set up” the connection:
  - Allow each end to know the other exists and is listening
  - Negotiation of optional parameters
    - Maximum Segment Size
    - Maximum Window Size
  - Triggers allocation of transport entity resources
    - Buffer space allocated
    - Entry in connection tables

Connection State Diagram – Reliable Network Service



Connection Establishment



Setting up the connection

- What if a SYN received while not in the Listen state?
  - Reject with RST (Reset)
  - Queue request until matching open issued
  - Signal TS user to notify of pending request

Termination

- Connection can be terminated by sending FIN
- Graceful termination
  - CLOSE_WAIT state and FIN_WAIT must accept incoming data until FIN received
  - Ensures both sides have received all outstanding data and that both sides agree to connection termination before actual termination

Unreliable Network Service

- Now let’s look at the more general case if we are building our transport service on top of an unreliable network layer

- An unreliable network service makes the transport layer much more complicated if we want to ensure reliability
- Examples of unreliable network services:
  - Internet using IP,
  - Frame Relay using LAPF
  - IEEE 802.3 using unacknowledged connectionless LLC
- Segments may get lost
- Segments may arrive out of order

Problems

- Ordered Delivery
- Retransmission strategy
- Duplication detection
- Flow control
- Connection establishment
- Connection termination
- Crash recovery

Ordered Delivery

- Segments may arrive out of order
- Number segments sequentially
- TCP numbers each octet sequentially
- Segments are numbered by the first octet number in the segment

- TCP actually numbers segments starting at a random value!
  - Minimizes possibility that a segment still in the network from an earlier, terminated connection between the same hosts is mistaken for a valid segment in a later connection (who would also have to happen to use the same port numbers)

Retransmission Strategy

- Need to re-transmit when
  - Segment damaged in transit
  - Segment fails to arrive
- Receiver must acknowledge successful receipt
- Use cumulative acknowledgement
- Time out waiting for ACK triggers re-transmission

- How long to wait until re-transmitting?
  - Too short: duplicate data
  - Too long: Unnecessary delay delivering data

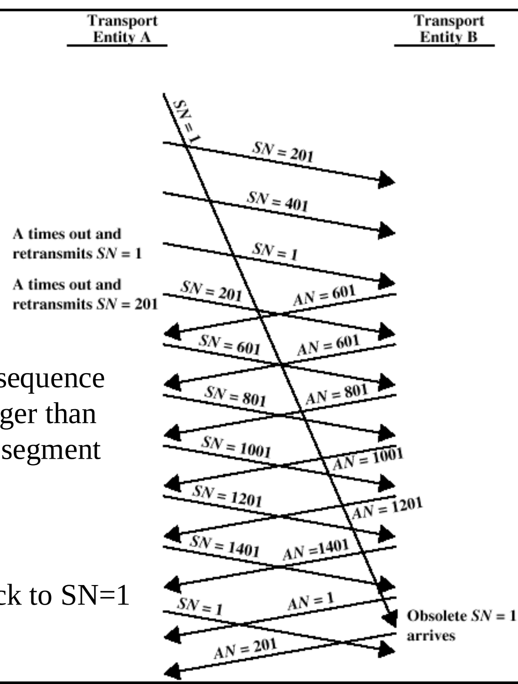
Timer Value

- Fixed timer
  - Based on understanding of network behavior
  - Can not adapt to changing network conditions
  - Too small leads to unnecessary re-transmissions
  - Too large and response to lost segments is slow
  - Should be a bit longer than Round Trip Time (RTT)
- Adaptive scheme
  - E.g. set timer to average of previous ACKs
  - Problems:
    - Sender may not ACK immediately
    - Cannot distinguish between ACK of original segment and re-transmitted segment
    - Conditions may change suddenly

Duplication Detection

- If ACK lost, segment is re-transmitted
- Receiver must recognize duplicates
- Duplicate received prior to closing connection
  - Receiver assumes ACK lost and ACKs duplicate
  - Sender must not get confused with multiple ACKs
  - Sequence number space large enough to not cycle within maximum life of segment

- Also possible to receive a duplicate after closing the connection!



Illustrates need for the sequence number space to be larger than the maximum possible segment lifetime

Note: cycle back to SN=1

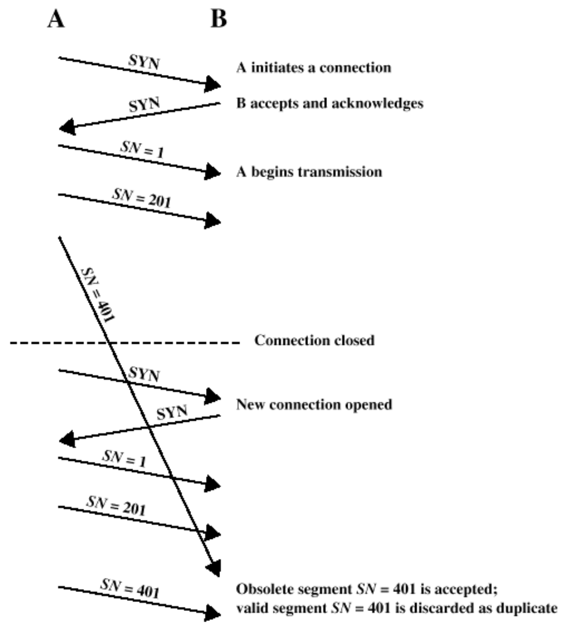
Flow Control

- Can use credit allocation described earlier
- Generally little harm if a single ACK/Credit segment is lost, will resynchronize the next time
- Problem if B sends AN=i, W=0 closing window
- Later, B sends AN=i, W=j to reopen, but this is lost
- Sender thinks window is closed, receiver thinks it is open
- Solution: use window timer
- If timer expires, send something to break the deadlock
  - Could be re-transmission of previous segment

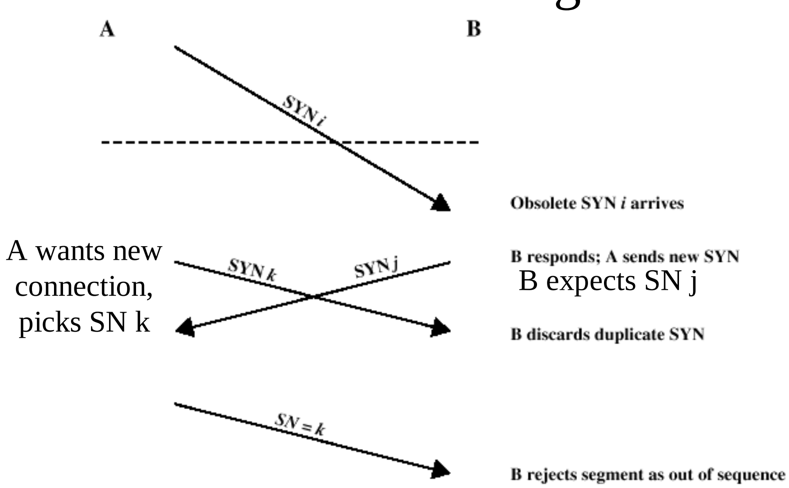
Connection Establishment

- Two way handshake
  - A send SYN, B replies with SYN
  - Lost SYN handled by re-transmission
    - Can lead to duplicate SYNs
  - Ignore duplicate SYNs once connected
- Lost or delayed data segments can cause connection problems
  - Segment from old connections

Two Way Handshake: Obsolete Data Segment



Two Way Handshake: Obsolete SYN Segment

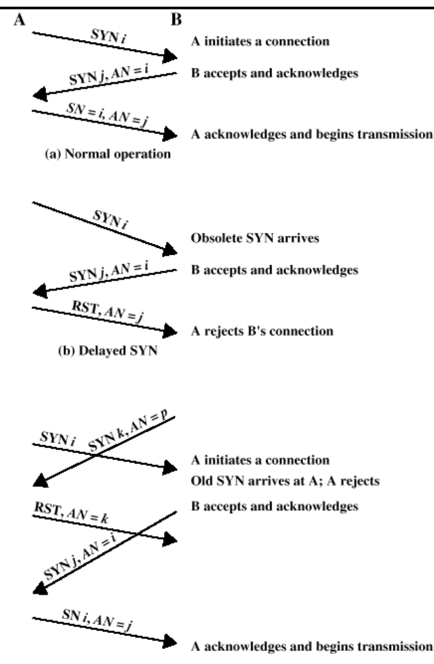


Connection Establishment – Three Way Handshake

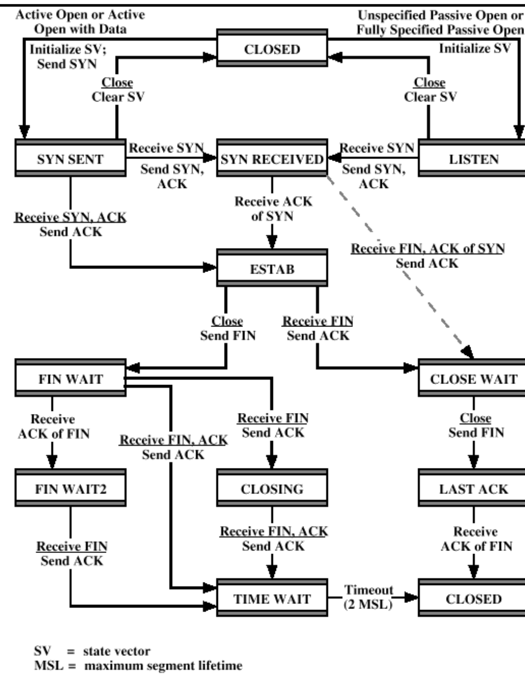
- Solution: Explicitly acknowledge each other's SYN and sequence number
  - Use SYN i
  - Need ACK to include i

- Called the Three Way Handshake

Three Way Handshake: Examples



Three Way Handshake: State Diagram



Connection Termination

- Same problems we had with connection establishment can also occur with connection termination
  - Lost or obsolete FIN segment
  - Can lose last data segment if FIN arrives before last data segment
- Solution: associate sequence number with FIN
- Receiver waits for all segments before FIN sequence number
- Must explicitly ACK FIN

Graceful Close

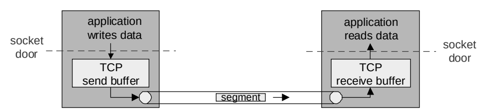
- Send FIN i and receive AN i
- Receive FIN j and send AN j
- Wait twice maximum expected segment lifetime

Crash Recovery

- If the transport service crashes and restarts, after restart all state info is lost
- Connection is half open
  - Side that did not crash still thinks it is connected
- Close connection using persistence timer
  - Wait for ACK for (time out) * (number of retries)
  - When expired, close connection and inform user
- Send RST i in response to any i segment arriving
- User must decide whether to reconnect
  - Problems with lost or duplicate data

TCP:Overview RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no “message boundaries”
- pipelined:
  - TCP congestion and flow control set window size
- send & receive buffers

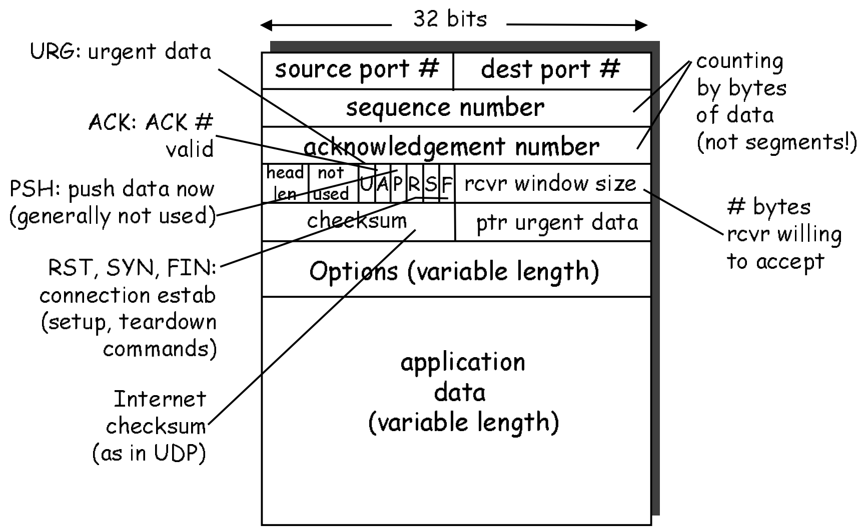


- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init’s sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP Properties

- stream orientation. stream of OCTETS (bytes) passed between send/ recv
- byte stream is full duplex
  - think of it as two independent streams joined with a **piggybacking** mechanism
  - piggybacking - one data stream has control info for the other data stream (going the other way)
- unstructured stream
  - TCP doesn’t show packet boundaries to applications
  - But you can still structure your message if you want
  - Recall usage with sockets:
    - One write() call to send data
    - May require multiple read() calls

TCP segment structure



TCP Fields

- Source, Destination Port: 16 bits each
- Sequence Number: 32 bits
  - Sequence # of first data octet in the segment, initialized randomly as described earlier
- ACK Number: 32 bits
  - Piggybacked ACK, contains sequence number of the next data octet the receiver expects
- Header Len: 4 bits
  - Number of 32 bit words in the header
- Not Used: 6 bits for future use

TCP Fields

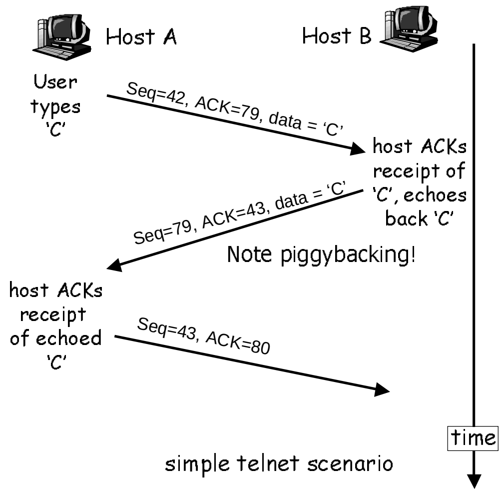
- Flags – 6 bits
  - URG – Urgent Pointer field significant
  - ACK – Ack field significant
  - PSH – Push (flush or "push" buffer now, send data to app)
  - RST – Reset connection
  - SYN – Synchronize sequence numbers
  - FIN – No more data
- Window – 16 bits
  - Flow control credit allocation
- Checksum – 16 bits
  - One's complement sum as in UDP
- Urgent Pointer – 16 bits
  - Last octet in a seq of "urgent" data.   Sometimes not interpreted. Urgent data should be processed now, even before any data sitting in the buffer (e.g. send control-c to terminate)
- Options – Variable
  - Support for timestamping, negotiating MSS

TCP seq. #'s and ACKs

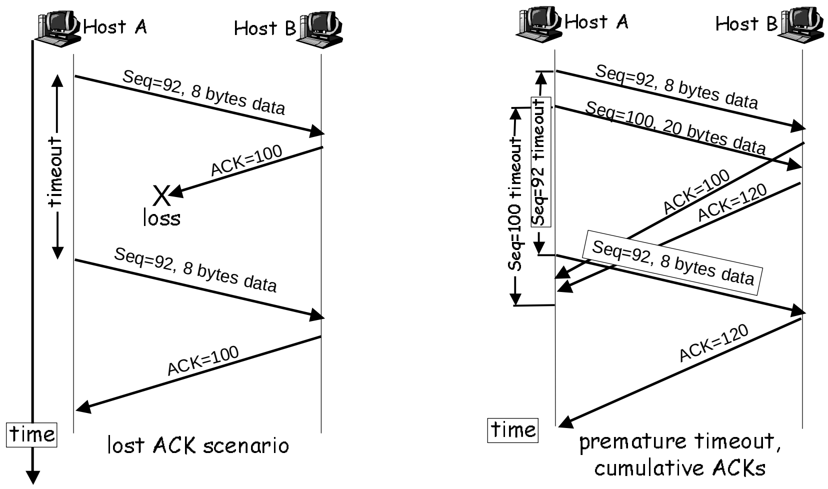
Seq. #'s:
  - byte stream "number" of first byte in segment's data

ACKs:
  - seq # of next byte expected from other side
  - cumulative ACK

Q: How does the receiver handles out-of-order segments?
  - A: TCP spec doesn't say, - up to implementer



TCP: retransmission scenarios



lost ACK scenario

premature timeout, cumulative ACKs

Sender must be smart enough to ignore duplicate ACK

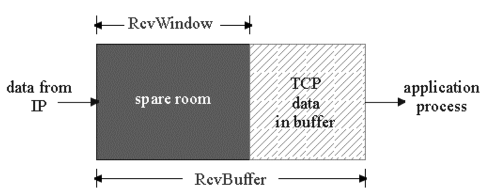
TCP Flow Control

flow control

sender won't overrun receiver's buffers by transmitting too much, too fast

`RcvBuffer` = size or TCP Receive Buffer

`RcvWindow` = amount of spare room in Buffer



receiver buffering

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- **`RcvWindow` field** in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received **`RcvWindow`**

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
  - note: RTT will vary
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- **`SampleRTT:`** measured time from segment transmission until ACK receipt
  - ignore retransmissions, cumulatively ACKed segments
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current **`SampleRTT`**

TCP Round Trip Time and Timeout

```
EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT
```

- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.1

Setting the timeout

- **`EstimatedRTT`** plus "safety margin"
- large variation in **`EstimatedRTT ->`** larger safety margin

```
Timeout = EstimatedRTT + 4*Deviation
```

TCP Connection Management

Three way handshake:

Step 1: client end system sends TCP SYN control segment to server
- specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment
- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #

TCP Connection Management (cont.)

Closing a connection:

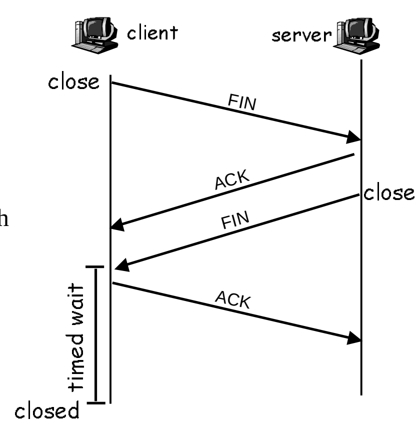
Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.

Step 3: client receives FIN, replies with ACK.

- Enters “timed wait” - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.



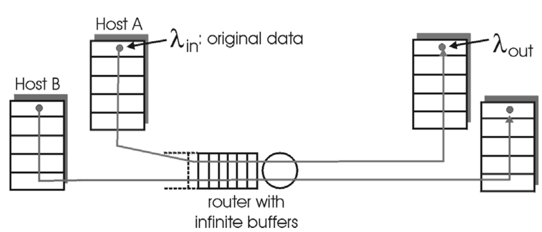
Principles of Congestion Control

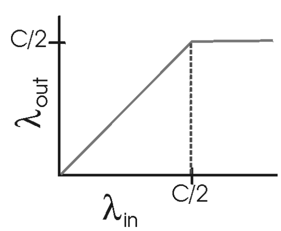
Congestion:

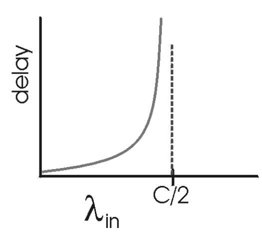
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- A top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission



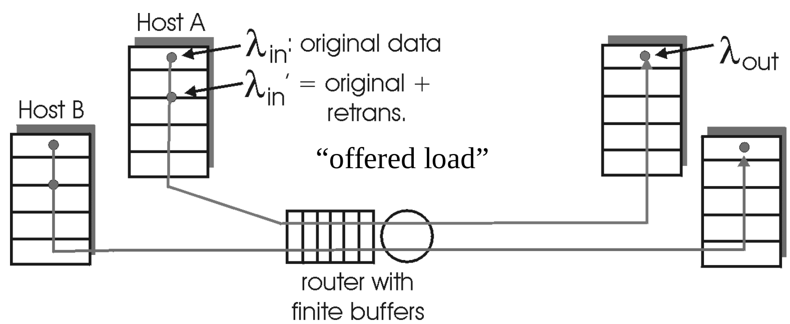




- large delays when congested
- maximum achievable throughput

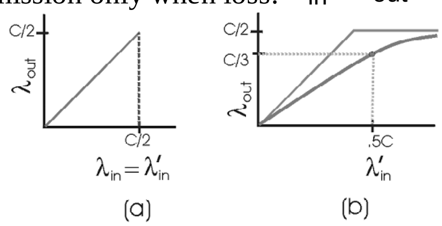
Causes/costs of congestion: scenario 2

- one router, **finite** buffers
- sender retransmission of lost packet



Causes/costs of congestion: scenario 2

- if: $\lambda_{in} = \lambda'_{in}$ (goodput)
- retransmission only when loss: $\lambda'_{in} > \lambda_{out}$



- Even worse: retransmission of delayed (not lost) packet makes larger λ'_{in} than the previous case for the same λ_{out}

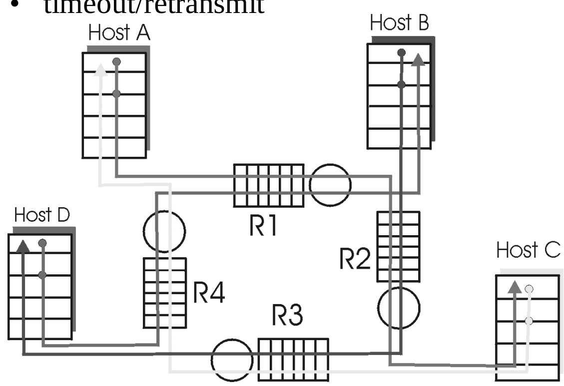
"costs" of congestion:

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

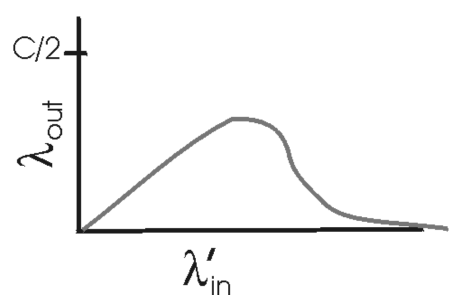
Causes/costs of congestion: scenario 3

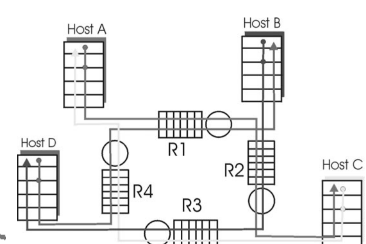
- four senders
- multihop paths
- timeout/retransmit

<u>Q:</u> what happens as λ_{in} and λ'_{in} increase ?



Causes/costs of congestion: scenario 3





Another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!
- Throughput goes to 0 as the heavy traffic approaches infinity
- In everyone's best interest to "back off" on transmission

Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:

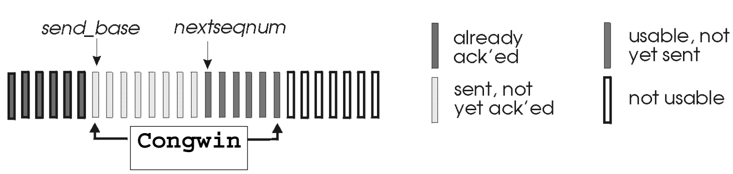
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

TCP Congestion Control

- end-end control (no network assistance)
- transmission rate limited by congestion window size, **`Congwin`**, over segments:



- w segments, each with MSS bytes sent in one RTT:

$$\boxed{\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}}$$

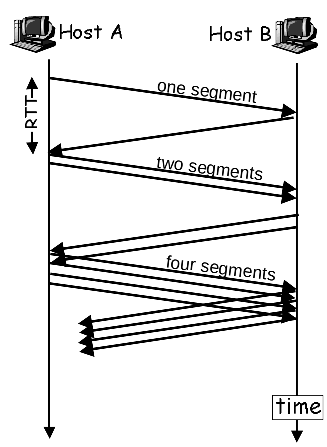
TCP congestion control:

- "probing" for usable bandwidth:
  - **ideally:** transmit as fast as possible (**`Congwin`** as large as possible) without loss
  - **Reality:**
  - *increase* **`Congwin`** until loss (congestion)
  - loss: *decrease* **`Congwin`**, then begin probing (increasing) again
- two "phases"
  - slow start
  - congestion avoidance
- important variables:
  - **`Congwin`**
  - **`threshold:`** defines threshold between two slow start phase, congestion control phase

TCP Slowstart

Slowstart algorithm

```
initialize: Congwin = 1
for (each segment ACKed)
      Congwin++
until (loss event OR
       CongWin > threshold)
```



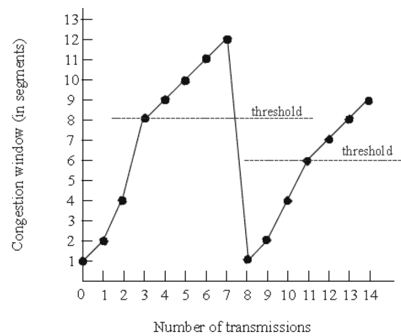
- exponential increase (per RTT) in window size (not so slow!)
- loss event: timeout (Tahoe TCP) and/or or three duplicate ACKs (Reno TCP)

- (What causes duplicate ACKs?)

TCP Congestion Avoidance

Congestion avoidance

```
/* slowstart is over       */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
      Congwin++
  }
threshold = Congwin/2
Congwin = 1
perform slowstart[1]
```



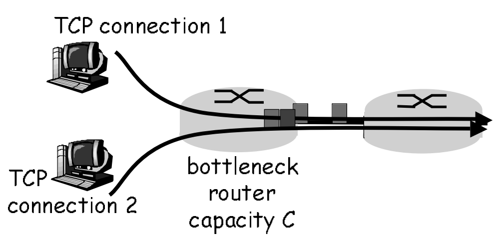
AIMD

TCP congestion avoidance:

- AIMD: *additive increase, multiplicative decrease*
  - increase window by 1 per RTT
  - decrease window by factor of 2 on loss event

TCP Fairness

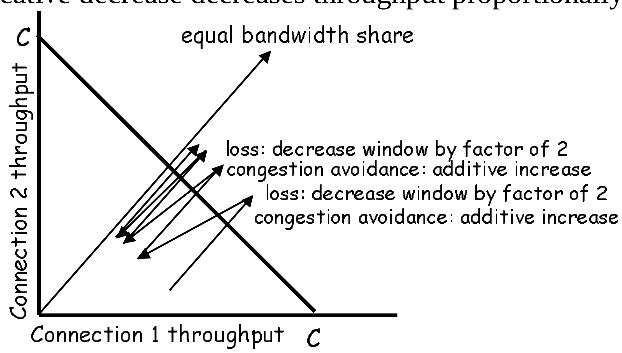
Fairness goal: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughput increases
- multiplicative decrease decreases throughput proportionally



Eventually the two connections fluctuate along equal bandwidth line

TCP vs. UDP

- When to use TCP
  - Need reliable network service
  - Want flow, congestion control
- When to use UDP
  - Don't want overhead of TCP
  - Don't want congestion control! I.e. we don't want to be "fair"
    - Multimedia apps
    - Don't want data rate throttled, but ironically this can lead to unfair transmission rate and could actually bring all traffic to a halt
    - Could also be unfair using TCP by opening multiple parallel connections (often done with web data)