Internet Protocol Stack

- **Application**: supporting network applications
  - FTP, SMTP, HTTP
- **Transport**: host-host data transfer
  - TCP, UDP
- **Network**: routing of datagrams from source to destination
  - IP, routing protocols
- **Link**: data transfer between neighboring network elements
  - WiFi, Ethernet
- **Physical**: bits “on the wire”
  - Radios, coaxial cable, optical fibers
Transport Layer
Roadmap

- Overview transport layer
- Wireless TCP to address the challenges
  - Unreliable links
  - Node mobility
Transport services and protocols

- provide logical communication between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP
Transport 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 sending letters to 12 kids
- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service
Internet transport-layer protocols

- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup

- services not available:
  - delay guarantees
  - bandwidth guarantees
Multiplexing/demultiplexing

Demultiplexing at rcv host:
delivering received segments to correct socket

Multiplexing at send host:
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

= socket  = process

<table>
<thead>
<tr>
<th>application</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport</td>
<td></td>
</tr>
<tr>
<td>network</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
<tr>
<td>physical</td>
<td></td>
</tr>
</tbody>
</table>

host 1

<table>
<thead>
<tr>
<th>P1</th>
<th>application</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport</td>
<td></td>
<td></td>
</tr>
<tr>
<td>network</td>
<td></td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
<td></td>
</tr>
<tr>
<td>physical</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

host 2

<table>
<thead>
<tr>
<th>P4</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport</td>
<td></td>
</tr>
<tr>
<td>network</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
<tr>
<td>physical</td>
<td></td>
</tr>
</tbody>
</table>

host 3
How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket

TCP/UDP segment format:

- source port #
- dest port #
- other header fields
- application data (message)
Connectionless demultiplexing

- Create sockets with port numbers:
  
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(12534);
  DatagramSocket mySocket2 = new DatagramSocket(12535);
  ```

- UDP socket identified by two-tuple:
  
  `(dest IP address, dest port number)`

- When host receives UDP segment:
  
  - checks destination port number in segment
  - directs UDP segment to socket with that port number

- IP datagrams with different source IP addresses and/or source port numbers directed to same socket
Connectionless demux (cont)

```java
DatagramSocket serverSocket = new DatagramSocket(6428);
```

SP provides “return address”
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number

- Receiving host uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple

- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request
Connection-oriented demux (cont)
Connection-oriented demux: Threaded Web Server
Client side

- Create a socket with the socket() system call
- Connect the socket to the address of the server using the connect() system call
- Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.
Server side

- Create a socket with the socket() system call
- Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the listen() system call
- Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
- Send and receive data
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?
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
  - loss tolerant
  - rate sensitive

- other UDP uses
  - DNS
  - SNMP

- reliable transfer over UDP:
  add reliability at application layer
  - application-specific error recovery!

UDP segment format:

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

Length, in bytes of UDP segment, including header

Application data (message)
UDP checksum

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

Sender:
- treat segment content 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
  - YES - no error detected. But maybe errors nonetheless? More later....
Internet Checksum Example

- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result

- Example: add two 16-bit integers
  - Take one’s complement
  - Add them
  - Take one’s complement
Internet Checksum Example (Cont.)

**Note**
- When adding numbers, a carryout from the most significant bit needs to be added to the result.

**Example: add two 16-bit integers**

```
1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

wraparound
1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1

sum
1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum
0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

Benefits: easy to compute; can do incremental update; endian-independent
Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Reliable Data Transfer

Reliable data transfer over a reliable channel

• over a reliable channel
Reliable Data Transfer

Reliable data transfer over a reliable channel
- over a reliable channel
- over a channel with error
Reliable Data Transfer

Reliable data transfer over a reliable channel
- over a reliable channel
- over a channel with error
  - Checksum + NACK or ACK
- over a channel with error and loss
Reliable Data Transfer

Reliable data transfer over a reliable channel
- over a reliable channel
- over a channel with error
  - Checksum + NACK or ACK
- over a channel with error and loss
  - Checksum + ACK + sequence no. + timeout
Reliable transfer in action

(a) operation with no loss

(b) lost packet
Reliable transfer in action

(c) lost ACK

(d) premature timeout
Any problem with the above protocol?
Performance

- It works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:
Performance

- It works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

\[ T_{\text{transmit}} = \frac{L}{R} \text{ (packet length in bits)} = \frac{8\text{kb/pkt}}{10^9 \text{ b/sec}} = 8 \text{ microsec} \]

- \( U_{\text{sender}} \) utilization - fraction of time sender busy sending

\[ U_{\text{sender}} = \frac{L}{RTT + L/R} = \frac{.008}{30.008} = 0.00027 \]

- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!
Stop-and-wait operation

\[ U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027 \]
How to fix the problem?
Pipelined protocols

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts
- range of sequence numbers must be increased
- buffering at sender and/or receiver

Two generic forms of pipelined protocols: go-Back-N, selective repeat
Pipelining: increased utilization

\[ U_{\text{sender}} = \frac{3 \times L / R}{\text{RTT} + L / R} = \frac{.024}{30.008} = 0.0008 \]
Go-Back-N

Sender:
- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed

- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- timeout(n): retransmit pkt n and all higher seq # pkts in window
GBN in action

sender
send pkt0
send pkt1
send pkt2
send pkt3 (wait)
rcv ACK0
send pkt4
rcv ACK1
send pkt5
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
receiver
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3
What is the drawback of GBN?
How to eliminate that?
Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #‘s
  - again limits seq #’s of sent, unACKed pkts
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
Selective repeat

**sender**

- data from above:
  - if next available seq # in window, send pkt

**timeout(n):**
- resend pkt n, restart timer

**ACK(n) in [sendbase, sendbase+N]:**
- mark pkt n as received
- if n is smallest unACKed pkt, advance window base to next unACKed seq #

**receiver**

- **pkt n in** [rcvbase, rcvbase+N-1]
  - send ACK(n)
  - out-of-order: buffer
  - in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

- **pkt n in** [rcvbase-N,rcvbase-1]
  - ACK(n)

- otherwise:
  - ignore
Selective repeat in action

pkt0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 sent
0 1 2 3 4 5 6 7 8 9

pkt2 sent
0 1 2 3 4 5 6 7 8 9

pkt3 sent, window full
0 1 2 3 4 5 6 7 8 9

ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9

ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9

pkt2 TIMEOUT, pkt2 resent
0 1 2 3 4 5 6 7 8 9

ACK3 rcvd, nothing sent
0 1 2 3 4 5 6 7 8 9

pkt0 rcvd, delivered, ACK0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 rcvd, delivered, ACK1 sent
0 1 2 3 4 5 6 7 8 9

pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9

pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9

pkt5 rcvd, buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, pkt2,pkt3,pkt4,pkt5 delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9
**Reliable Data Transfer Mechanisms**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checksum</td>
<td>Detect bit errors</td>
</tr>
<tr>
<td>Timer</td>
<td>Detect packet loss at sender</td>
</tr>
<tr>
<td>Sequence number</td>
<td>Detect packet loss and duplicates at receiver</td>
</tr>
<tr>
<td>ACK</td>
<td>Inform sender that pkt has been received</td>
</tr>
<tr>
<td>NACK</td>
<td>Inform sender that pkt has not been received correctly</td>
</tr>
<tr>
<td>Window, pipelining</td>
<td>Increase throughput, and adapt to receiver buffer size and network congestion</td>
</tr>
</tbody>
</table>
TCP: Overview

- **point-to-point:**
  - one sender, one receiver

- **reliable, in-order byte steam:**
  - 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 segment structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>head len</td>
<td>Head length</td>
</tr>
<tr>
<td>Urg</td>
<td>Urgent flag bit</td>
</tr>
<tr>
<td>Num</td>
<td>Number flag bit</td>
</tr>
<tr>
<td>PSH</td>
<td>Push flag bit</td>
</tr>
<tr>
<td>RST</td>
<td>RST flag bit</td>
</tr>
<tr>
<td>SYN</td>
<td>SYN flag bit</td>
</tr>
<tr>
<td>FIN</td>
<td>FIN flag bit</td>
</tr>
<tr>
<td>Urg data pnter</td>
<td>Urg data pointer</td>
</tr>
<tr>
<td>Receive window</td>
<td>Receive window size</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Options</td>
<td>Variable length options</td>
</tr>
<tr>
<td>application data</td>
<td>Application data (variable length)</td>
</tr>
</tbody>
</table>

- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection estab (setup, teardown commands)
- **Internet checksum**: (as in UDP)

- **Counting by bytes of data (not segments!)**
- **# bytes rcvr willing to accept**
TCP Connection Management

Recall: TCP sender, receiver establish “connection” before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. RcvWindow)

- client: connection initiator
  ```java
  Socket clientSocket = new Socket("hostname","port number");
  ```

- server: contacted by client
  ```java
  Socket connectionSocket = welcomeSocket.accept();
  ```

Three way handshake:

Step 1: client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data

Step 2: server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data
TCP Connection Management (cont.)

**Closing a connection:**

client closes socket:

```java
clientSocket.close();
```

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

**Step 2:** server receives FIN, replies with ACK. Closes connection, sends FIN.
TCP Connection Management (cont.)

**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.

**Note:** with small modification, can handle simultaneous FINs.
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 receiver handles out-of-order segments
- A: TCP spec doesn’t say, - up to implementor

---

**Simple telnet scenario**

1. **Host A** sends a segment with `Seq=42, ACK=79, data = 'C'`.
2. Host B sends an ACK `Seq=79, ACK=43` for the received segment, and echoes back `C`.
3. Host A sends another segment with `Seq=43, ACK=80`.
4. Host B sends an ACK `Seq=43, ACK=80` for the received segment.
TCP Timeout

Q: how to set TCP timeout value?
TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?
TCP Round Trip Time and Timeout

**Q:** how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- 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
  - Why?
- `SampleRTT` will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current `SampleRTT`
TCP Round Trip Time and Timeout

\[ \text{EstimatedRTT} = (1- \alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT} \]

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \( \alpha = 0.125 \)
Example RTT estimation:

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

SampleRTT

Estimated RTT
TCP Round Trip Time and Timeout

Setting the timeout

- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

\[
\text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
\]

(typically, \( \beta = 0.25 \))

Then set timeout interval:

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
TCP
sender
(simplified)

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
    switch(event)

    event: data received from application above
        create TCP segment with sequence number NextSeqNum
        if (timer currently not running)
            start timer
        pass segment to IP
    NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
        retransmit not-yet-acknowledged segment with
        smallest sequence number
        start timer

    event: ACK received, with ACK field value of y
        if (y > SendBase) {
            SendBase = y
            if (there are currently not-yet-acknowledged segments)
                start timer
        }

} /* end of loop forever */

Comment:
• SendBase-1: last cumulatively ack'ed byte
Example:
• SendBase-1 = 71;
y= 73, so the rcvr wants 73+;
y > SendBase, so that new data is acked
TCP: retransmission scenarios

Host A

SendBase = 100

Host B

Seq=92, 8 bytes data

ACK=100

X

timeout

loss

Seq=92, 8 bytes data

Host A

Sendbase = 100

Host B

Seq=92, 8 bytes data

Seq=100, 20 bytes data

ACK=100

ACK=120

SendBase = 120

SendBase = 120

Seq=92 timeout

timeout

premature timeout

host A

host B

lost ACK scenario

time

SendBase = 100
TCP retransmission scenarios (more)

Cumulative ACK scenario

Host A

SendBase = 120

timeout

time

Cumulative ACK scenario

Host B

X

loss

Seq=92, 8 bytes data

ACK=120

Seq=100, 20 bytes data

ACK=100
Sending ACKs is an overhead. How to reduce the overhead?
# TCP ACK generation [RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq. #. Gap detected</td>
<td>Immediately send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediately send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
Retransmission upon timeout incurs significant delay. Can we retransmit sooner?
Fast Retransmit

- Time-out period often relatively long:
  - Long delay before resending lost packet

- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs for that segment

- If sender receives 3 ACKs for same data, it assumes that segment after ACKed data was lost:
  - Fast retransmit: resend segment before timer expires
  - Why 3 dup acks?
Fast retransmit algorithm:

**event:** ACK received, with ACK field value of y
  
  if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments)
      start timer
  }
  
  else {
    increment count of dup ACKs received for y
    if (count of dup ACKs received for y = 3) {
      resend segment with sequence number y
    }
  }

A duplicate ACK for already ACKed segment  
Fast retransmit
TCP Flow Control

- receive side of TCP connection has a receive buffer:
  - RcvWindow
  - spare room
  - RcvBuffer
  - data from IP
  - TCP data in buffer
  - application process

- sender won’t overflow receiver’s buffer by transmitting too much, too fast

- speed-matching service: matching the sending rate to the receiving app’s drain rate

- Rcvr advertises spare room by including value of RcvWindow in segments

- Sender limits unACKed data to RcvWindow
  - guarantees receive buffer doesn’t overflow

- app process may be slow at reading from buffer
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

Host A

Host B

λ_{in} : original data

unlimited shared output link buffers

λ_{out}

C/2

λ_{out}

C/2

λ_{in}

delay

C/2

λ_{in}

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

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda_{in}^' > \lambda_{out}^'$
- retransmission of delayed (not lost) packet makes $\lambda_{in}$ larger (than perfect case) for same $\lambda_{out}$

"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

Q: what happens as $\lambda_{in}$ and $\lambda'_{in}$ increase?
Causes/costs of congestion: scenario 3

another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!"
How to avoid network congestion?
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 (XCP)
TCP congestion control:

- **goal**: TCP sender should transmit as fast as possible, but without congesting network
  - **Q**: how to find rate just below congestion level
- decentralized: each TCP sender sets its own rate, based on *implicit* feedback:
  - **ACK**: segment received (a good thing!), network not congested, so increase sending rate
  - **lost segment**: assume loss due to congested network, so decrease sending rate
TCP congestion control: bandwidth probing

- “probing for bandwidth”: increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate
  - continue to increase on ACK, decrease on loss (since available bandwidth is changing, depending on other connections in network)
- Q: how fast to increase/decrease?
  - details to follow

TCP’s “sawtooth” behavior
TCP Congestion Control: details

sender limits rate by limiting number of unACKed bytes “in pipeline”:

\[ \text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd} \]

- cwnd: differs from rwnd (how, why?)
- sender limited by \( \min(cwnd, rwnd) \)

- roughly,

\[ \text{rate} = \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec} \]

- cwnd is dynamic, function of perceived network congestion
TCP Congestion Control: more details

**segment loss event:** reducing cwnd

- timeout: no response from receiver
  - cut cwnd to 1

- 3 duplicate ACKs: at least some segments getting through (recall fast retransmit)
  - cut cwnd in half, less aggressively than on timeout

**ACK received: increase cwnd**

- Slow start phase:
  - increase exponentially fast (despite name) at connection start, or following timeout

- congestion avoidance:
  - increase linearly
TCP Slow Start

- when connection begins, $cwnd = 1$ MSS
  - example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be $>> \frac{MSS}{RTT}$
  - desirable to quickly ramp up to respectable rate
- increase rate exponentially until first loss event or when threshold reached
  - double $cwnd$ every RTT
  - done by incrementing $cwnd$ by 1 for every ACK received
TCP: congestion avoidance

- when cwnd > ssthresh
  grow cwnd linearly
  - increase cwnd by 1 MSS per RTT
  - approach possible congestion slower than in slowstart
  - implementation: cwnd = cwnd + MSS/cwnd for each ACK received

AIMD

- ACKs: increase cwnd by 1 MSS per RTT: additive increase
- loss: cut cwnd in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease
Popular “flavors” of TCP

Transport round

cwnd window size (in segments)

TCP Tahoe

TCP Reno

ssthresh
TCP Congestion Control

- When CongWin is below Threshold, sender is in slow-start phase, window grows exponentially.

- When CongWin is above Threshold, sender is in congestion-avoidance phase, window grows linearly.

- When a triple duplicate ACK occurs, Threshold set to CongWin/2 and CongWin set to Threshold.

- When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.
Recap

- What is flow control?
- What is congestion control?
- Why do we need congestion control?
- What are the two classes of congestion control?
- How does TCP perform congestion control?