Chapter 3: Transport Layer

our goals:
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
3.6 principles of congestion control
3.7 TCP congestion control

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
  - recv 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, 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 who demux to ilth house siblings
- network-layer protocol = postal service

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees
Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP

Multiplexing/demultiplexing

Multiplexing at sender:
- handle data from multiple sockets, add transport header (later used for demultiplexing)

Demultiplexing at receiver:
- use header info to deliver received segments to correct socket

How demultiplexing works

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

Connectionless demultiplexing

- recall: created socket has host-local port #:
  - DatagramSocket mySocket1 = new DatagramSocket(12534);
- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #
- recall: when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #
- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #
- IP datagrams with same dest. port #, but different source IP addresses and/or source port numbers will be directed to same socket at dest

Connectionless demux: example

Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver 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
Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management
3.6 principles of congestion control
3.7 TCP congestion control

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
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header

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. But maybe errors nonetheless? More later...
Internet checksum: example

Example: add two 16-bit integers

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

1. Wraparound: 1 0 1 1 1 0 1 1 1 0 1 1 0 1 1 1 1
2. Sum: 1 0 1 1 1 0 1 1 1 0 1 1 1 0 0 1 1
3. Checksum: 0 1 0 0 0 0 1 0 0 0 0 1 1

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

Chapter 3 outline

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
- Segment structure
- Reliable data transfer
- Flow control
- Connection management
3.6 Principles of congestion control
3.7 TCP congestion control

Principles of reliable data transfer

- Important in application, 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: getting started

- `rdt_send()`: called from above, e.g., by app. Passed data to `deliver_data()` to deliver to receiver upper layer
- `deliver_data()`: called by `rdt` to deliver data to upper layer
- `udt_send()`: called by `rdt` to transfer packet over unreliable channel to receiver
- `rdt_recv()`: called when packet arrives on rcv-side of channel
- `deliver_data()`: called by `rdt` to deliver data to upper layer
- `udt_send()`: called by `rdt` to transfer packet over unreliable channel to receiver
- `rdt_recv()`: called when packet arrives on rcv-side of channel
Transport Layer 3-25

Reliable data transfer: getting started

we'll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

state: when in this “state” next state uniquely determined by next event

event causing state transition

actions taken on state transition

Transport Layer 3-26

rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

sender

receiver

Transport Layer 3-27

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- the question: how to recover from errors:

How do humans recover from “errors” during conversation?

Transport Layer 3-28

rdt2.0: operation with no errors

receiver

sender

Transport Layer 3-29

rdt2.0: FSM specification

Transport Layer 3-30

rdt2.0: operation with no errors
### rdt2.0: error scenario

- `rdt_send(data)`
- `snpkt = make_pkt(data, checksum)`
- `udt_send(snpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`

### rdt2.0 has a fatal flaw!

**what happens if ACK/NAK corrupted?**
- sender doesn’t know what happened at receiver!
- can’t just retransmit: possible duplicate

**handling duplicates:**
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds sequence number to each pkt
- receiver discards (doesn’t deliver up) duplicate pkt

### rdt2.1: sender, handles garbled ACK/NAKs

- `sndpkt = make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(1, data, checksum)`
- `udt_send(sndpkt)`

### rdt2.1: receiver, handles garbled ACK/NAKs

- `rdt_recv(rcvpkt)`
- `notcorrupt(rcvpkt)`
- `isACK(rcvpkt)`

### rdt2.1: discussion

**sender:**
- seq # added to pkt
- two seq #’s (0,1) will suffice. Why?
- must check if received packet is duplicate
- twice as many states
- state indicates whether “expected” pkt should have seq # of 0 or 1

**receiver:**
- must check if received packet is duplicate
- state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can not know if its last ACK/NAK received OK at sender

### rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must explicitly include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: retransmit current pkt
rdt2.2: sender, receiver fragments

```
rtt2.2: sender, receiver fragments

 - wait for call 0 from above
 - sndpkt = make_pkt(0, data, checksum)
 - udt_send(sndpkt)
 - rdt_send(data)
 - udt_send(sndpkt)
 - rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt))
 - rdt_send(data)
 - udt_send(sndpkt)
 - rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)
 - Wait for ACK

sender FSM

fragment
```

rdt3.0: channels with errors and loss

new assumption: underlying channel can also lose packets (data, ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help ...
- but not enough

approach: sender waits "reasonable" amount of time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. # is already handled this
  - requires countdown timer

```
rdt3.0 sender

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_send(data)
Wait for ACK0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 1))
Wait for call 1 from above
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer
rdt_send(data)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)
stop_timer
stop_timer
udt_send(sndpkt)
timeout
udt_send(sndpkt)
timeout
```

```
rdt3.0 in action

sender
sender
receiver
receiver
receiver
rcv pkt0
rcv pkt0
rcv pkt1
rcv pkt1
rcv pkt0
rcv pkt0
rcv pkt1
rcv pkt1
rcv ack0
rcv ack0
rcv ack1
rcv ack1
send pkt0
send pkt0
send pkt1
send pkt1
send pkt0
send pkt0
send pkt1
send pkt1
(a) no loss
(b) packet loss
```

```
rdt3.0 in action

sender
sender
receiver
receiver
receiver
rcv pkt0
rcv pkt0
rcv pkt1
rcv pkt1
rcv pkt0
rcv pkt0
rcv pkt1
rcv pkt1
rcv ack0
rcv ack0
rcv ack1
rcv ack1
send pkt0
send pkt0
send pkt1
send pkt1
send pkt0
send pkt0
send pkt1
send pkt1
(a) no loss
(b) packet loss
```

```
Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:
  \[ D_{max} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs} \]
- \( U_{sender} \): utilization — fraction of time sender busy sending
  \[ U_{sender} = \frac{L/R}{R/T + L/R} = \frac{0.008}{0.008} = 0.00027 \]
- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!
```

Transport Layer 3-37
Transport Layer 3-36
Transport Layer 3-39
Transport Layer 3-38
Transport Layer 3-40
Transport Layer 3-37
Transport Layer 3-41
Transport Layer 3-40
Transport Layer 3-41
**Pipelining: increased utilization**

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 oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window

Selective Repeat:
- sender can have up to N unack'ed packets in pipeline
- receiver only sends cumulative ack
- sender has timer for oldest unacked packet
- when timer expires, retransmit only that unacked packet

Selective Repeat:
- sender can have up to N unack'ed packets in pipeline
- receiver sends individual ack for each packet
- sender maintains timer for each unacked packet
- when timer expires, retransmit only that unacked packet

**Pipelining: overview**

Go-Back-N:
- 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

**Pipelined protocols: overview**

Go-Back-N:
- sender can have up to N unack'ed packets in pipeline
- receiver only sends cumulative ack
- doesn't ack packet if there's a gap
- sender has timer for oldest unack'ed packet
- when timer expires, retransmit all unack'ed packets

Selective Repeat:
- sender can have up to N unack'ed packets in pipeline
- receiver sends individual ack for each packet
- sender maintains timer for each unack'ed packet
- when timer expires, retransmit only that unack'ed packet

**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 oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window

**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
**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
  - limits seq #s of sent, unACKed pkts.

**Selective repeat in action**

Sender:
- Send pkt0, send pkt1, send pkt2
- Send pkt0, send pkt1, send pkt2
- Send pkt0, send pkt1, send pkt2

Receiver:
- Receive pkt0, send ack0
- Receive pkt1, send ack1
- Receive pkt2, send ack2
- Receive pkt3, send ack3
- Receive pkt4, send ack4
- Receive pkt5, send ack5
- Receive pkt6, send ack6
- Receive pkt7, send ack7
- Receive pkt8, send ack8

Q: What happens when ack2 arrives?
**Selective repeat: dilemma**

example:
- seq If's: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq If size and window size to avoid problem in (b)?

**Chapter 3 outline**

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management
3.6 principles of congestion control
3.7 TCP congestion control

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

**TCP seq. numbers, ACKs**

sequence numbers:
- byte stream "number" of first byte in segment's data
acknowledgements:
- seq If 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

**TCP segment structure**

source port # | dest port # | sequence number | acknowledgement number | options (variable length) | application data (variable length)
--- | --- | --- | --- | --- | ---

32 bits

- 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)

**TCP seq. numbers, ACKs**

User types 'C'

Host A

Seq=42, ACK=79, data = 'C'

Host B

Seq=79, ACK=43, data = 'C'

Q: what relationship between seq If size and window size to avoid problem in (b)?
**TCP round trip time, 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
- SampleRTT will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT

**Exp. weighted moving average**:

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

**timeout interval**:
- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT → larger safety margin
  - estimate SampleRTT deviation from EstimatedRTT:
    \[
    \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
    \]
    (typically, \( \beta = 0.25 \))

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]

**Chapter 3 outline**

- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer
- 3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

**TCP reliable data transfer**

- 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
  - if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

**TCP sender events:**

- data rcvd 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

- timeout:
  - retransmit segment that caused timeout
  - restart timer
  - if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments
TCP sender (simplified)

- Data received from application above.
- Create segment, seq. #: NextSeqNum.
- Pass segment to IP (i.e., “send”).
- NextSeqNum = NextSeqNum + length(data).
- If timer currently not running, start timer.
- If ACK received, with ACK field value y:
  - If y > SendBase:
    - SendBase = y (SendBase–1: last cumulatively ACKed byte).
    - If there are currently not-yet-acked segments, start timer.
    - Else stop timer.
- If (timer currently not running):
  - Start timer.
- Data received from application above.
- Retransmit not-yet-acked segment with smallest seq. #.
- If timeout:
  - ACK received, with ACK field value y
  - If y > SendBase:
    - SendBase = y

TCP: retransmission scenarios

Lost ACK scenario:

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq=92, 8 bytes of data</td>
<td>ACK=100</td>
</tr>
<tr>
<td>Seq=100, 20 bytes of data</td>
<td>ACK=120</td>
</tr>
<tr>
<td>Seq=120, 15 bytes of data</td>
<td></td>
</tr>
<tr>
<td>Timeout</td>
<td></td>
</tr>
<tr>
<td>Cumulative ACK</td>
<td></td>
</tr>
</tbody>
</table>

Preliminary timeout:

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq=92, 8 bytes of data</td>
<td>ACK=100</td>
</tr>
<tr>
<td>Seq=120, 20 bytes of data</td>
<td>ACK=120</td>
</tr>
<tr>
<td>Seq=120, 20 bytes of data</td>
<td></td>
</tr>
<tr>
<td>Timeout</td>
<td></td>
</tr>
</tbody>
</table>

TCP ACK generation [RFC 1122, RFC 2581]

**Event at Receiver**
- Arrival of in-order segment with expected seq. #. All data up to expected seq. # already ACKed.
- Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK.
- Arrival of in-order segment with expected seq. #. One other segment has ACK pending.
- Immediately send single cumulative ACK, indicating next expected byte.
- Arrival of out-of-order segment higher-than-expect seq. #. Gap detected.
- Immediately send duplicate ACK, indicating seq. # of next expected byte.
- Arrival of segment that partially or completely fills gap.
- Immediate send ACK, provided that segment starts at lower end of gap.

TCP 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.

**TCP fast retransmit**

If sender receives 3 ACKs for same data (“triple duplicate ACKs”), resends unacknowledged segment with smallest seq. #.
- Likely that unacknowledged segment lost, so don’t wait for timeout.

TCP fast retransmit after sender receipt of triple duplicate ACK.
**Chapter 3 outline**

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
   - segment structure
   - reliable data transfer
   - flow control
   - connection management
3.6 principles of congestion control
3.7 TCP congestion control

**TCP flow control**

- 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 autodetect RcvBuffer
- sender limits amount of unacked (“in-flight”) data to receiver’s rwnd value
- guarantees receive buffer will not overflow

**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:
- will 2-way handshake always work in network?
- 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

2-way handshake failure scenarios:

TCP 3-way handshake

Client state

Server state

TCP 3-way handshake: FSM

TCP: closing a connection

Chapter 3 outline
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
- output link capacity: \( R \)
- no retransmission

idealization: perfect knowledge
- sender sends only when router buffers available

Idealization: known loss
- packets can be lost, dropped at router due to full buffers
- sender only resends if packet known to be lost

Causes/costs of congestion: scenario 2
- one router, finite buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: \( \lambda_{in} = \lambda_{out} \)
  - transport-layer input includes retransmissions: \( \lambda_{in} > \lambda_{out} \)

Host A
- in: original data
  - out: original data
  - 'in: original data, plus retransmitted data'

Host B
- in: original data
  - out: original data

when sending at \( R/2 \), some packets are retransmissions but asymptotic goodput is still \( R/2 \)
Causes/costs of congestion: scenario 2

Realistic: duplicates
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending two copies, both of which are delivered

“costs” of congestion:
- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt
  • decreasing goodput

Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as $\lambda_m$ and $\lambda_n$ increase?
- as red $\lambda_m$ increases, all arriving blue pkts at upper queue are dropped, blue throughput $\rightarrow 0$

another “cost” of congestion:
- when packet dropped, any “upstream transmission capacity used for that packet was wasted!”

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 for sender to send at

Case study: ATM ABR congestion control

ABR: available bit rate:
- “elastic service”
  • if sender’s path “under-loaded”:
    - sender should use available bandwidth
  • if sender’s path congested:
    - sender throttled to minimum guaranteed rate
RM (resource management) cells:
- sent by sender, interspersed with data cells
- bits in RM cell set by switches (“network-assisted”)
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact
Case study: ATM ABR congestion control

- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - senders’ send rate thus max supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
3.6 principles of congestion control
3.7 TCP congestion control

TCP congestion control: additive increase multiplicative decrease

- approach: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - additive increase: increase cwnd by 1 MSS every RTT until loss detected
  - multiplicative decrease: cut cwnd in half after loss

TCP Congestion Control: details

- TCP sending rate:
  - roughly: send cwnd bytes, wait RTT for ACKs, then send more bytes
- sender limits transmission:
  - rate = \( \frac{cwnd}{RTT} \) bytes/sec
- cwnd is dynamic, function of perceived network congestion

TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially cwnd = 1 MSS
  - double cwnd every RTT
  - done by incrementing cwnd for every ACK received

TCP: detecting, reacting to loss

- loss indicated by timeout:
  - cwnd set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - cwnd is cut in half window then grows linearly
- TCP Tahoe always sets cwnd to 1 (timeout or 3 duplicate acks)
TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?
A: when cwnd gets to 1/2 of its value before timeout.

Implementation:
- variable ssthresh
- on loss event, ssthresh is set to 1/2 of cwnd just before loss event

TCP throughput

- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume always data to send
  - W: window size (measured in bytes) where loss occurs
  - avg. window size (If in-flight bytes) is 1/2 W
  - avg. throughput is 3/4W per RTT

TCP Futures: TCP over “long, fat pipes”

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

TCP throughput = \( \frac{1.22 \cdot MSS \cdot RTT}{L} \)

- to achieve 10 Gbps throughput, need a loss rate of L = 2 \times 10^{-10} — a very small loss rate!
- new versions of TCP for high-speed

TCP Fairness

fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K

Why is TCP fair?

two competing sessions:
- additive increase gives slope of 1, as throughput increases
- multiplicative decrease decreases throughput proportionally
Fairness (more)

Fairness and UDP
- multimedia apps often do not use TCP
- do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections
- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
  - instantiation, implementation in the Internet
  - UDP
  - TCP

Next:
- leaving the network “edge” (application, transport layers)
- into the network “core”