Chapter 3
Transport Layer

A note on the use of these ppt slides:
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides including the ones and slide contents to suit your needs. They obviously represent a lot of work on our part. In return for use, we only ask the following:
❑ If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
❑ If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.
Thanks and enjoy! JFK/KWR

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
❑ 3.6 Principles of congestion control
❑ 3.7 TCP congestion control

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 sending letters to 12 kids
processes = kids
app messages = letters in envelopes
hosts = houses
transport protocol = Ann and Bill
network-layer protocol = postal service

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

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
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

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)

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 (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket

Connectionless demultiplexing

- Create sockets with port numbers:
  - DatagramSocket mySocket1 = new DatagramSocket(99111);
  - DatagramSocket mySocket2 = new DatagramSocket(99222);
- UDP socket identified by two-tuple:
  - (dest IP address, dest port number)
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux (cont)

DatagramSocket serverSocket = new DatagramSocket(6428);

Connection-oriented demux

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

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 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
  - 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

  \[
  \begin{array}{c}
  \text{1 1 1 0 0 1 1 0 0 1 1 0 1 1 0 1 1 0} \\
  \text{1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0} \\
  \text{1 0 1 1 1 0 1 1 1 0 1 1 1 0 0} \\
  \text{0 1 0 0 1 1 0 0 0 0 1 1 1 0 0 1 1 1} \\
  \end{array}
  \]

  - wraparound
  - sum
  - checksum

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 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: 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

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
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:
  - acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
  - negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK, NAK) rcvr→sender

Rdt2.0: FSM specification

Rdt2.0: operation with no errors

Rdt2.0: error scenario

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 adds sequence number to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn’t deliver up) duplicate pkt
- stop and wait
  Sender sends one packet, then waits for receiver response
rdt2.1: receiver, handles garbled ACK/NAKs

```
seq # added to pkt
two seq. #s (0,1) will suffice. Why?
must check if received
ACK/NAK corrupted
twice as many states
state must "remember"
whether "current" pkt
has 0 or 1 seq.
```

Sender:
- seq # added to pkt
- two seq. #s (0,1) will suffice. Why?
- must check if received
  ACK/NAK corrupted
twice as many states
- state must "remember"
  whether "current" pkt
  has 0 or 1 seq.

Receiver:
- must check if received packet is duplicate
- state indicates whether 0 or 1 is expected pkt
- 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

```
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

rdt_send(data)
```

```
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
```

```
stop_timer
```

```
start_timer
```

```
wait for ACK1
```

```
wait for call 0 from above
```

Transport Layer 3-13

Transport Layer 3-14

---

rdt3.0 sender

```
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
```

```
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
```

```
stop_timer
```

```
start_timer
```

```
wait for ACK
```

```
wait for call 1 from above
```

Transport Layer 3-35

Transport Layer 3-36

---

rdt3.0: channels with errors and loss

New assumption:
underlying channel can also lose packages (data
or 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
  (not last):
  - retransmission will be
    duplicate, but use of seq.
    #s already handles this
  - receiver must specify seq
    # of pkt being ACKed
- requires countdown timer

Transport Layer 3-35
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

\[
T_{\text{transmit}} = \frac{L}{R} \left( \frac{8 \text{KB/pkt}}{10^{10} \text{b/sec}} \right) = 8 \text{ microsec}
\]

\[
U_{\text{sender}} = \frac{L}{R} = \frac{1\text{KB}}{30\text{ms}} = \frac{33\text{kB/sec}}{1\text{Gbps}} = 0.0027
\]

- \( U_{\text{sender}} \): utilization - fraction of time sender busy sending
- 1KB pkt every 30 msec \( \rightarrow \) 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!

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

\[
U_{\text{sender}} = \frac{3L}{R} = \frac{3 \times 33\text{kB}}{30\text{ms}} = 0.0008
\]
Go-Back-N

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

\[ \begin{align*}
\text{send}_{\text{base}} & \quad \text{nextseqnum} \\
\text{already ack'ed} & \quad \text{usable, not yet sent} \\
\text{sent, not yet ack'ed} & \quad \text{not usable}
\end{align*} \]

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

GBN: sender extended FSM

\[ \begin{align*}
\text{wait} & \quad \text{start}_{\text{timer}} \\
\text{udt}_{\text{send}} & (\text{sndpkt}[\text{base}]) \\
& \ldots \\
\text{udt}_{\text{send}} & (\text{sndpkt}[\text{nextseqnum}-1]) \\
\text{timeout} & \quad \text{rdt}_{\text{send}} (\text{data}) \\
& \quad \text{if} \ (\text{nextseqnum} < \text{base}+N) \\
& \quad \quad \text{sndpkt}[\text{nextseqnum}] = \text{make}_\text{pkt}(\text{nextseqnum}, \text{data}, \text{chksum}) \\
& \quad \quad \text{udt}_{\text{send}} (\text{sndpkt}[\text{nextseqnum}]) \\
& \quad \quad \text{if} \ (\text{base} = \text{nextseqnum}) \\
& \quad \quad \quad \text{start}_{\text{timer}} \\
& \quad \quad \quad \text{nextseqnum}++ \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{refuse}_{\text{data}} (\text{data}) \\
& \quad \text{base} = \text{getacknum}(\text{rcvpkt})+1 \\
& \quad \text{If} \ (\text{base} = \text{nextseqnum}) \\
& \quad \quad \text{stop}_{\text{timer}} \\
& \quad \quad \text{else} \\
& \quad \quad \text{start}_{\text{timer}} \\
\text{default} & \quad \text{rdt}_{\text{rcv}} (\text{rcvpkt}) \land \text{notcorrupt}(\text{rcvpkt}) \\
& \quad \quad \text{if} \ (\text{rcvpkt}, \text{hasseqnum}(\text{expectedseqnum})) \\
& \quad \quad \quad \text{extract}(\text{rcvpkt}, \text{data}) \\
& \quad \quad \quad \text{deliver}_{\text{data}} (\text{data}) \\
& \quad \quad \quad \text{sndpkt} = \text{make}_\text{pkt}(\text{expectedseqnum}, \text{ACK}, \text{chksum}) \\
& \quad \quad \quad \text{udt}_{\text{send}} (\text{sndpkt}) \\
& \quad \quad \quad \text{expectedseqnum}++ \\
\end{align*} \]

GBN: receiver extended FSM

\[ \begin{align*}
\text{wait} & \quad \text{udt}_{\text{send}} (\text{sndpkt}) \\
\text{default} & \quad \text{rdt}_{\text{rcv}} (\text{rcvpkt}) \land \text{notcorrupt}(\text{rcvpkt}) \land \text{hasseqnum}(\text{rcvpkt}, \text{expectedseqnum}) \\
& \quad \quad \text{extract}(\text{rcvpkt}, \text{data}) \\
& \quad \quad \text{deliver}_{\text{data}} (\text{data}) \\
& \quad \quad \text{sndpkt} = \text{make}_\text{pkt}(\text{expectedseqnum}, \text{ACK}, \text{chksum}) \\
& \quad \quad \text{udt}_{\text{send}} (\text{sndpkt}) \\
& \quad \quad \text{expectedseqnum}++ \\
\end{align*} \]

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

\[ \begin{align*}
\text{(a) sender view of sequence numbers} & \quad \text{(b) receiver view of sequence numbers}
\end{align*} \]

GBN in action
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 smallest unACKed pkt, advance window base to next unACKed seq #

Receiver:
- Pkt n in [rcvbase,rcvbase-N-1):
  - Send ACK(n)
- In-order: Deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
- Out-of-order: Buffer
- Otherwise:
  - Ignore

Selective repeat in action

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

- 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

TCP segment structure

RFCs: 793, 1122, 1323, 2018, 2581

- Full duplex data:
  - Bi-directional data flow in same connection
  - MSS: Maximum segment size
- Connection-oriented:
  - Handshaking (exchange of control msgs) initiates sender, receiver states before data exchange
- Flow controlled:
  - Sender will not overwhelm receiver

Internet checksum (as in UDP)

Options (variable length)

32 bits

Source port #
Dest port #
Sequence number
Acknowledgement number
TCP payload
TCP Checksum
TCP Options
TCP Window
TCP Urgent Pointer
TCP Flags
TCP Reserved
TCP Data

TCP source port
TCP destination port
TCP sequence number
TCP acknowledgement number
TCP data
TCP options
TCP header length
TCP reserved bits
TCP urgent pointer
TCP flags
TCP window
TCP checksum
TCP options
TCP data

# bytes receiver willing to accept

Counting by bytes of data (not segments)
TCP seq. #’s and ACKs

- **Seq. #’s:**
  - Byte stream "number" of first byte in segment’s data
  - Seq. # of next byte expected from other side
  - Cumulative ACK

- **Q:** how receiver handles out-of-order segments
  - Host A: TCP spec doesn’t say, up to implementor

- **User types C**
  - User ACKs
  - Host B: receipt of echoed C
  - User ACKs receipt of echoed C, echoes back C

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
  - SampleRTT will vary, want estimated RTT "smoother"
  - Average several recent measurements, not just current SampleRTT

TCP Round Trip Time and Timeout

- **Exponential weighted moving average**
  - Influence of past sample decreases exponentially fast
  - Typical value: $\alpha = 0.125$

Example RTT estimation:

- **EstimatedRTT** = $(1 - \alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}$
  - Exponential weighted moving average
  - Influence of past sample decreases exponentially fast
  - Typical value: $\alpha = 0.125$

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:
    - DevRTT = $(1 - \beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT-EstimatedRTT}|$
      - Typically, $\beta = 0.25$
  - Then set timeout interval:
    - TimeoutInterval = EstimatedRTT + 4 * 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
- 3.6 Principles of congestion control
- 3.7 TCP congestion control
**TCP reliable data transfer**

- **TCP creates rdt service on top of IP's unreliable service**
- **Pipelined segments**
- **Cumulative acks**
- **TCP uses single retransmission timer**

**Retransmissions are triggered by:**
- timeout events
- duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

**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
  - Ack rcvd:
    - If acknowledges previously unacked segments
    - update what is known to be acked
    - start timer if there are outstanding segments

**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
  }
}
```

**TCP: retransmission scenarios**

- **Host A**
  - Seq=100, 20 bytes data
  - ACK=100
  - loss
  - timeout

- **Host B**
  - Seq=92, 8 bytes data
  - Seq=92 timeout
  - ACK=120
  - loss

- **Cumulative ACK scenario**

**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
- Arrival of in-order segment with expected seq #: One other segment has ACK pending
- Arrival of out-of-order segment higher-than-expect seq #: Gap detected
- Arrival of segment that partially or completely fills gap

**TCP Receiver action**

- Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
- Immediately send single cumulative ACK, ACKing both in-order segments
- Immediately send duplicate ACK, indicating seq. # of next expected byte
- Immediate send ACK, provided that segment starts at lower end of gap

**TCP retransmission scenarios (more)**

- **Host A**
  - Seq=92, 8 bytes data
  - ACK=100
  - loss

- **Host B**
  - Seq=120
  - last ACK scenario
  - premature timeout

**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
- Arrival of in-order segment with expected seq #: One other segment has ACK pending
- Arrival of out-of-order segment higher-than-expect seq #: Gap detected
- Arrival of segment that partially or completely fills gap

**TCP Receiver action**

- Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
- Immediately send single cumulative ACK, ACKing both in-order segments
- Immediately send duplicate ACK, indicating seq. # of next expected byte
- Immediate send ACK, provided that segment starts at lower end of gap
**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.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - **fast retransmit:** resend segment before timer expires
- **Time-out period** often relatively long:
  - long delay before resending lost packet
  - Sender usually sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.

**Fast retransmit algorithm:**

- **event:** ACK received, with ACK field value of $y$
- if ($y > \text{SendBase}$) {
  - **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 $y$
      }
    }
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - **fast retransmit:** resend segment before timer expires

---

**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**

- receive side of TCP connection has a receive buffer:
  - **flow control** — sender won't overflow receiver's buffer by transmitting too much, too fast
  - speed-matching service: matching the send rate to the receiving app's drain rate
- app process may be slow at reading from buffer

**TCP Flow control: how it works**

- **Rcvr** advertises spare room in buffer by including value of RcvrWindow in segments
- **Sender** limits unACKed data to RcvrWindow
  - guarantees receive buffer doesn't overflow

- **Rcvr** advertises spare room in buffer by including value of RcvrWindow in segments
- **Sender** limits unACKed data to RcvrWindow
  - guarantees receive buffer doesn't overflow

---

**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 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
  - Socket clientSocket = new Socket("hostname","port number");
- server: contacted by client
  - 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

Closing a connection:
- client closes socket: 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.
- 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.

TCP Connection Management (cont.)

TCP Connection Management (cont.)

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 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
- limited shared output link buffers

Causes/costs of congestion: scenario 3
- four senders
- multihop paths
- timeout/retransmit

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
**Case study: ATM ABR congestion control**

**ABR:** available bit rate:
- "elastic service"
- if sender's path "underloaded": 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")
  - NE bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact

**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**

- end-end control (no network assistance)
- sender limits transmission: \( \text{LastByteSent - LastByteAcknowledged} \leq \text{CongWin} \)
- Roughly, \( \text{rate} = \frac{\text{CongWin}}{\text{RTT}} \) bytes/sec
  - CongWin is dynamic, function of perceived network congestion

**TCP AIMD**

- multiplicative decrease: cut CongWin in half after loss event
- additive increase: increase CongWin by 1 MSS every RTT in the absence of loss events: probing

**TCP Slow Start**

- When connection begins, \( \text{CongWin} = 1 \text{ MSS} \)
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
  - available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  ❍ double CongWin every RTT
  ❍ done by incrementing CongWin for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast

Refinement

- After 3 dup ACKs:
  ❍ CongWin is cut in half
  ❍ window then grows linearly
- But after timeout event:
  ❍ CongWin instead set to 1 MSS;
  ❍ window then grows exponentially
  ❍ to a threshold, then grows linearly

Summary: TCP Congestion Control

- When CongWin is below Threshold, sender 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.

TCP throughput

- What’s the average throughput of TCP as a function of window size and RTT?
  ❍ Ignore slow start
  ❍ Let W be the window size when loss occurs.
  ❍ When window is W, throughput is W/RTT
  ❍ Just after loss, window drops to W/2, throughput to W/2RTT.
  ❍ Average throughput: .75 W/RTT
TCP Futures

- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:
  \[ \frac{1.22 \cdot \text{MSS}}{\text{RTT} \cdot \sqrt{L}} \]
- \[ L = 2 \cdot 10^{-10} \] Wow
- New versions of TCP for high-speed needed!

TCP Fairness

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

Fairness and UDP

- Multimedia apps often do not use TCP
- do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate \( R \), supporting 9 connections;
  - new app asks for 1 TCP, gets rate \( R/9 \)
  - new app asks for 11 TCPs, gets \( R/2 \)

Delay modeling

Q: How long does it take to receive an object from a Web server after sending a request?

Ignoring congestion, delay is influenced by:
- TCP connection establishment
- data transmission delay
- slow start

Notation, assumptions:
- Assume one link between client and server of rate \( R \)
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

Window size:
- First assume: fixed congestion window, \( W \) segments
- Then dynamic window, modeling slow start

Fixed congestion window (1)

First case:
- \( WS/R \cdot \text{RTT} + S/R \cdot \text{ACK} \times \text{first segment in window returns before window's worth of data sent} \)
- \[ \text{delay} = 2\text{RTT} + O/R \]
**Fixed congestion window (2)**

Second case:
- \( WS/R + RTT + S/R \) wait for \( ACK \) after sending window’s worth of data sent
- delay = \( 2RTT + O/R \)
  \( + (K-1)[S/R + RTT - WS/R] \)

**TCP Delay Modeling: Slow Start (1)**

Now suppose window grows according to slow start.

Will show that the delay for one object is:

\[ \text{Latency} = 2RTT + \frac{O}{R} \left( \frac{RTT}{S} \right) - \frac{2}{P} \left( \frac{S}{R} \right) - (2^P - 1) \frac{S}{R} \]

where \( P \) is the number of times TCP idles at server:

\[ P = \min(K, K-1) \]

- where \( Q \) is the number of times the server idles if the object were of infinite size.
- and \( K \) is the number of windows that cover the object.

**TCP Delay Modeling: Slow Start (2)**

Delay components:
- \( 2RTT \) for connection estab and request
- \( O/R \) to transmit object
- time server idles due to slow start
- Server idles: \( P = \min(K-1, Q) \) times

Example:
- \( O/S = 15 \) segments
- \( K = 4 \) windows
- \( Q = 2 \)
- \( P = \min(K-1, Q) = 2 \)
- Server idles \( P = 2 \) times

**TCP Delay Modeling (3)**

\[ \frac{S}{R} \cdot RTT = \text{time from when server starts to send segment until server receives acknowledgement} \]

\[ 2^P \cdot \frac{S}{R} \cdot RTT = \text{time to transmit the first window} \]

\[ \frac{O}{R} + 2RTT + \sum_{i=1}^{P} \text{idlesTime}_i \]

\[ = \frac{O}{R} + 2RTT + \sum_{i=1}^{P} \text{idlesTime}_i + \frac{S}{R} \cdot RTT - 2^P \cdot \frac{S}{R} \]

\[ = \frac{O}{R} + 2RTT + \left( P \cdot RTT - \frac{S}{R} \cdot (2^P - 1) \right) \]

Calculation of \( Q, \) number of idles for infinite-size object, is similar (see HW).

**HTTP Modeling**

- Assume Web page consists of:
  - 1 base HTML page (of size \( O \) bits)
  - \( M \) images (each of size \( O \) bits)
- Non-persistent HTTP:
  - 1 \( RTT \) to request and receive base HTML file
  - \( RTT \) to request and receive \( M \) images
- Response time = \( (M+1)O/R + (M+2)RTT + \text{sum of idle times} \)
- Persistent HTTP:
  - 2 \( RTT \) to request and receive base HTML file
- Response time = \( (M+1)O/R + 3RTT + \text{sum of idle times} \)
- Non-persistent HTTP with \( X \) parallel connections
  - Suppose \( M/X \) integer.
  - 1 \( RTT \) to request and receive \( M \) images
  - Response time = \( (M+1)O/R + (M/X + 1)2RTT + \text{sum of idle times} \)
Chapter 3: Summary

- Principles behind transport layer services:
  - Multiplexing, demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
  - Instantiation and implementation in the Internet

- Next:
  - Leaving the network “edge” (application, transport layers)

Transport Layer 3-111