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
  - 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, 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 in-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
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
3.6 principles of congestion control
3.7 TCP congestion control
**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

### TCP/UDP segment format

<table>
<thead>
<tr>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
</tr>
<tr>
<td>other header fields</td>
</tr>
<tr>
<td>application data (payload)</td>
</tr>
</tbody>
</table>
Connectionless demultiplexing

- created socket has host-local port 

```java
DatagramSocket mySocket1 = new DatagramSocket(12534);
```

- 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

```
DatagramSocket serverSocket = new DatagramSocket(6428);
DatagramSocket mySocket1 = new DatagramSocket(5775);
DatagramSocket mySocket2 = new DatagramSocket(9157);
```

Transport Layer 3-9
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

Connection-oriented demux: example

three segments, all destined to IP address B, dest port: 80 are demultiplexed to different sockets
Connection-oriented demux: example

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

- length, in bytes of UDP segment, including header

**why is there a UDP?**

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

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
\end{array}
\]

wraparound
\[
\begin{array}{cccccccccc}
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}
\]

sum checksum
\[
\begin{array}{cccccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

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)
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

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

Transport Layer 3-24
**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

```
Wait for call from above
rdt_send(data)
packet = make_pkt(data)
udt_send(packet)
```

```
Wait for call from below
rdt_rcv(packet)
extract (packet, data)
deliver_data(data)
```

sender

receiver

**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?
```
**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
  - sender retransmits pkt on receipt of NAK
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - feedback: control msgs (ACK, NAK) from receiver to sender

**rdt2.0: FSM specification**

```
# sender
rdt_send(data)
sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)

# receiver
rdt_rcv(rcvpkt) && isACK(rcvpkt) rdt_send(sndpkt)
udt_send(rcvpkt)

# Wait for call from above
```

```
# Wait for ACK or NAK
```

```
# Wait for call from below
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```

```
# rdt_send(data)
```

```
# rdt_rcv(rcvpkt) && isNAK(rcvpkt)
```

```
udt_send(sndpkt)
```

```
# Wait for call from above
```

```
# Wait for ACK or NAK
```

```
# rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
```

```
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
**rdt2.0: operation with no errors**

- `rdt_send(data)`
  - `snkpkt = make_pkt(data, checksum)`
  - `udt_send(sndpkt)`

- `Wait for call from above`
  - `rdt_rcv(rcvpkt) && isNAK(rcv pkt)`
  - `udt_send(sndpkt)`

- `Wait for ACK or NAK`
  - `rdt_rcv(rcvpkt) && isACK(rcv pkt)`
  - `rdt_send(data)`

**rdt2.0: error scenario**

- `Wait for call from above`
  - `snkpkt = make_pkt(data, checksum)`
  - `udt_send(sndpkt)`

- `rdt_rcv(rcv pkt) && isNAK(rcv pkt)`
  - `udt_send(snd pkt)`

- `Wait for ACK or NAK`
  - `rdt_send(data)`

- `rdt_rcv(rcv pkt) && isACK(rcv pkt)`
  - `udt_send(snd pkt)`

- `notcorrupt(rcv pkt)`
  - `extract(rcv pkt, data)`
  - `deliver_data(data)`
  - `udt_send(ACK)`

- `Wait for NAK`
  - `rdt_rcv(rcv pkt) && isNAK(rcv pkt)`
  - `udt_send(NAK)`

- `Wait for call from below`
**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

**Stop and Wait**: sender sends one packet, then waits for receiver response

---

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

![Diagram of rdt2.1 logic](image-url)
**rdt2.1: receiver, handles garbled ACK/NAKs**

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 “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 #
**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**

```
sender FSM fragment

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

Wait for call 0 from above

Wait for ACK 0

rdt_rcv(rcvpkt) &&
( corrupt(rcvpkt) ||
  isACK(rcvpkt,1) )
udt_send(sndpkt)

```

```
receiver FSM fragment

udt_send(sndpkt)

Wait for 0 from below

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt) &&
isACK(rcvpkt,0)

```

```
extract(rcvpkt.data)
deliver_data(data)
sndpkt = make_pkt(ACK1, checksum)
udt_send(sndpkt)
```

Transport Layer 3-36
**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. #’s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

---

**rdt3.0 sender**

```
rdt_send(data)
start_timer

wait for call 0 from above

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 1)
stop_timer

udt_send(sndpkt)

wait for ACK0

rdt_rcv(rcvpkt) && ( corrupt(rcvpkt) || isACK(rcvpkt, 1) )
Lambda

timeout
udt_send(sndpkt)
start_timer

wait for call 0 from above

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)
stop_timer

udt_send(sndpkt)
start_timer

wait for ACK1

rdt_rcv(rcvpkt) && ( corrupt(rcvpkt) || isACK(rcvpkt, 0) )
Lambda

timeout
udt_send(sndpkt)
start_timer

wait for call 1 from above

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

---

Transport Layer 3-37

Transport Layer 3-38
rdt3.0 in action

sender

send pkt0
rcv ack0
send pkt1
rcv ack1
send pkt0

receiver

pkt0
ack0
pkt1
ack1
pkt0
ack0

(a) no loss

sender

send pkt0
rcv ack0
send pkt1
rcv ack1
send pkt0

receiver

pkt0
ack0
pkt1
ack1
pkt0
ack0

(b) packet loss

sender

send pkt0
rcv ack0
send pkt1
rcv ack1
send pkt0

receiver

pkt0
ack0
pkt1
ack1
pkt0
ack0

(c) ACK loss

sender

send pkt0
rcv ack0
send pkt1
rcv ack1
send pkt0
rcv ack1
send duplicate

receiver

pkt0
ack0
pkt1
ack1
pkt0
ack0

(d) premature timeout/ delayed ACK
**Performance of rdt3.0**

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:
  
  \[ D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs} \]

  - **\( U_{\text{sender}} \): utilization** – fraction of time sender busy sending
    
    \[ U_{\text{sender}} = \frac{L}{R} \frac{L}{RTT + \frac{L}{R}} = \frac{0.008}{30.008} = 0.00027 \]

  - if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

---

**rdt3.0: stop-and-wait operation**

\[ U_{\text{sender}} = \frac{L}{R} \frac{L}{RTT + \frac{L}{R}} = \frac{0.008}{30.008} = 0.00027 \]
**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**

\[ U_{\text{sender}} = \frac{3L/R}{RTT + L/R} = \frac{0.024}{30.008} = 0.00081 \]

3-packet pipelining increases utilization by a factor of 3!
Pipelined protocols: overview

**Go-back-N:**
- sender can have up to $N$ unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn’t ack packet if there’s a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit all unacked packets

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

---

Transport Layer 3-45

Transport Layer 3-46
GBN: sender extended FSM

\[
\text{wait} \rightarrow \text{udt\_send(sndpkt}\[\text{base}\]) \rightarrow \text{udt\_send(sndpkt}\[\text{base+1}\]) \rightarrow \ldots \rightarrow \text{udt\_send(sndpkt}\[\text{nextseqnum-1}\]) \rightarrow \text{timeout} \rightarrow \text{start\_timer} \rightarrow \text{udt\_send(sndpkt}\[\text{base}\]) \rightarrow \text{udt\_send(sndpkt}\[\text{base+1}\]) \rightarrow \ldots \rightarrow \text{udt\_send(sndpkt}\[\text{nextseqnum-1}\]) \rightarrow \text{start\_timer} \\
\text{base} = \text{getacknum(rcvpkt)}+1 \rightarrow \text{stop\_timer} \rightarrow \text{start\_timer} \\
\text{else} \\
\text{refuse\_data(data)}
\]

GBN: receiver extended FSM

\[
\text{default} \rightarrow \text{udt\_send(sndpkt)} \rightarrow \text{rdt\_rcv(rcvpkt)} \&\& \text{notcorrupt(rcvpkt)} \&\& \text{hasseqnum(rcvpkt,expectedseqnum)} \\
\text{extract(rcvpkt,data)} \rightarrow \text{deliver\_data(data)} \rightarrow \text{sndpkt = make\_pkt(expectedseqnum,ACK,chksum)} \rightarrow \text{udt\_send(sndpkt)} \rightarrow \text{expectedseqnum++}
\]

**ACK-only:** always send ACK for correctly-received pkt with highest *in-order* seq #
- may generate duplicate ACKs
- need only remember *expectedseqnum*

**out-of-order pkt:**
- discard (don’t buffer): *no receiver buffering!*
- re-ACK pkt with highest in-order seq #
GBN in action

sender window (N=4)  sender  receiver
0 1 2 3 4 5 6 7 8  send pkt0  receive pkt0, send ack0
0 1 2 3 4 5 6 7 8  send pkt1  receive pkt1, send ack1
0 1 2 3 4 5 6 7 8  send pkt2  receive pkt2
0 1 2 3 4 5 6 7 8  send pkt3  receive pkt3, discard, (re)send ack1

(loss)

rcv ack0, send pkt4  receive pkt4, discard, (re)send ack1
rcv ack1, send pkt5  receive pkt5, discard, (re)send ack1

ignore duplicate ACK

pkt 2 timeout

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

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: sender, receiver windows

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)
  - 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

sender window \((N=4)\)

<table>
<thead>
<tr>
<th>seq</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pkt</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>wait</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transmission</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pkt</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**sender**
- send pkt0
- send pkt1
- send pkt2
- send pkt3
- wait
- rcv ack0, send pkt4
- rcv ack1, send pkt5
- record ack3 arrived
- pkt 2 timeout
- send pkt2
- record ack4 arrived
- record ack4 arrived
- Q: what happens when ack2 arrives?

**receiver**
- receive pkt0, send ack0
- receive pkt1, send ack1
- receive pkt3, buffer, send ack3
- receive pkt4, buffer, send ack4
- receive pkt5, buffer, send ack5
- rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2

Selective repeat: dilemma

example:
- seq #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 # size and window size to avoid problem in (b)?
## Chapter 3 outline

<table>
<thead>
<tr>
<th>3.1 transport-layer services</th>
<th>3.5 connection-oriented transport: TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 multiplexing and demultiplexing</td>
<td>segment structure</td>
</tr>
<tr>
<td>3.3 connectionless transport: UDP</td>
<td>reliable data transfer</td>
</tr>
<tr>
<td>3.4 principles of reliable data transfer</td>
<td>flow control</td>
</tr>
<tr>
<td></td>
<td>connection management</td>
</tr>
</tbody>
</table>

### TCP: Overview

**RFCs: 793, 1122, 1323, 2018, 2581**

- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order byte steam:**
  - 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) inits 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><strong>source port #</strong></td>
<td>Source port number</td>
</tr>
<tr>
<td><strong>dest port #</strong></td>
<td>Destination port number</td>
</tr>
<tr>
<td><strong>sequence number</strong></td>
<td>Sequence number of first byte in segment's data</td>
</tr>
<tr>
<td><strong>acknowledgement number</strong></td>
<td>Acknowledgement number of next byte expected from other side</td>
</tr>
<tr>
<td><strong>sequence number</strong></td>
<td>Sequence number of first byte in segment's data</td>
</tr>
<tr>
<td><strong>option</strong></td>
<td>Options block (variable length)</td>
</tr>
<tr>
<td><strong>data</strong></td>
<td>Application data (variable length)</td>
</tr>
<tr>
<td><strong>window size</strong></td>
<td>Receiver window size</td>
</tr>
<tr>
<td><strong>sent ACKed</strong></td>
<td>Sent segment acknowledged</td>
</tr>
<tr>
<td><strong>sent, not ACKed</strong></td>
<td>Segment not yet acknowledged</td>
</tr>
<tr>
<td><strong>window size</strong></td>
<td>Window size</td>
</tr>
<tr>
<td><strong>not usable</strong></td>
<td>Segment not usable</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)

**TCP seq. numbers, ACKs**

**sequence numbers:**
- Byte stream “number” of first byte in segment’s data

**acknowledgements:**
- Seq # of next byte expected from other side
- Cumulative ACK

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

**Host A**
- User types 'C'
- Seq=42, ACK=79, data = 'C'
- host ACKs receipt of echoed 'C'

**Host B**
- host ACKs receipt of 'C', echoes back 'C'
- Seq=79, ACK=43, data = 'C'
- Seq=43, ACK=80

**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
TCP round trip time, timeout

EstimatedRTT = (1 - α) * EstimatedRTT + α * SampleRTT

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125

TCP round trip time, timeout

- 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, β = 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

- TCP creates rdt service on top of IP’s unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

let’s initially consider simplified TCP sender:
- ignore duplicate acks
- ignore flow control, congestion control

Transport Layer 3-64
**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 ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

---

**TCP sender (simplified)**

---

Transport Layer 3-65

Transport Layer 3-66

---
**TCP: retransmission scenarios**

**lost ACK scenario**

Host A

- Seq=92, 8 bytes of data
- ACK=100

Host B

- Seq=92, 8 bytes of data
- ACK=100

 Timeout

**premature timeout**

Host A

- Seq=92, 8 bytes of data
- ACK=100
- Seq=100, 20 bytes of data

Host B

- Seq=92, 8 bytes of data
- ACK=100
- Seq=92, 8 bytes of data
- ACK=120
- Seq=100, 20 bytes of data
- ACK=120

**cumulative ACK**

Host A

- Seq=92, 8 bytes of data
- Seq=100, 20 bytes of data
- ACK=100
- ACK=120

Host B

- Seq=92, 8 bytes of data
- ACK=100
- Seq=120, 15 bytes of data

### 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-expected 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>Immediate send ACK, provided that segment starts at lower end of gap.</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>TCP fast retransmit</th>
</tr>
</thead>
<tbody>
<tr>
<td>If sender receives 3 ACKs for same data (&quot;triple duplicate ACKs&quot;), resend unacked segment with smallest seq #.</td>
</tr>
<tr>
<td>Likely that unacked segment lost, so don’t wait for timeout.</td>
</tr>
</tbody>
</table>
**TCP fast retransmit**

**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

Application process

TCP socket receiver buffers

TCP code

IP code

OS

Transport layer

TCP flow control

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

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 autoadjust RcvBuffer

Sender limits amount of unacked ("in-flight") data to receiver’s rwnd value

Guarantees receive buffer will not 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

Connection Management

before exchanging data, sender/receiver “handshake”:
  ▶ agree to establish connection (each knowing the other willing to establish connection)
  ▶ agree on connection parameters

Socket clientSocket = newSocket("hostname","port number");

Socket connectionSocket = welcomeSocket.accept();
**Agreeing to establish a connection**

2-way handshake:

Q: 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:

1. Choose `x` → `req_conn(x)` → `ESTAB` → `acc_conn(x)` → `ESTAB` → Client terminates → Server forgets `x` → Half open connection! (no client!)
2. Choose `x` → `req_conn(x)` → `ESTAB` → `req_conn(x)` → `ESTAB` → Retransmit `data(x+1)` → Server accepts `data(x+1)` → Server forgets `x`
TCP 3-way handshake

**Client state**
- **LISTEN**
  - Choose init seq num, x
  - Send TCP SYN msg
- **SYNSENT**
  - Received SYN(x) indicates server is live;
    - Send ACK for SYNACK;
    - This segment may contain client-to-server data
- **ESTABLISHED**
  - SYNRCVD

**Server state**
- **LISTEN**
  - Choose init seq num, y
  - Send TCP SYNACK msg, acking SYN
- **SYNRCVD**
  - SYNbit=1, Seq=x
  - ACKbit=1; ACKnum=x+1
  - Received SYN(x)
- **ESTABLISHED**
  - SYNRCVD

TCP 3-way handshake: FSM

1. **CLOSED**
   - Socket connectionSocket = newSocket("hostname", "port number");
   - SYN(seq=x)

2. **LISTEN**
   - SYN(rcvd)
   - SYNsent
   - SYNACK(seq=y, ACKnum=x+1)
   - Create new socket for communication back to client
   - ACK(ACKnum=y+1)

3. **ESTABLISHED**
   - SYNRCVD
   - SYNACK(seq=y, ACKnum=x+1)
   - ACK(ACKnum=y+1)
TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled
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
- output link capacity: \( R \)
- no retransmission

\[
\begin{align*}
\text{original data: } & \lambda_{\text{in}} \\
\text{throughput: } & \lambda_{\text{out}} \\
\text{maximum per-connection throughput: } & R/2 \\
\text{large delays as arrival rate, } & \lambda_{\text{in}} \text{ approaches capacity}
\end{align*}
\]

**Causes/costs of congestion: scenario 2**

- one router, finite buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: \( \lambda_{\text{in}} = \lambda_{\text{out}} \)
  - transport-layer input includes retransmissions: \( \lambda'_{\text{in}} \geq \lambda_{\text{in}} \)

\[
\begin{align*}
\text{original data: } & \lambda_{\text{in}} \\
\text{original data, plus retransmitted data: } & \lambda'_{\text{in}} \\
\text{throughput: } & \lambda_{\text{out}}
\end{align*}
\]
Causes/costs of congestion: scenario 2

**Idealization: perfect knowledge**
- sender sends only when router buffers available

![Diagram showing idealization]

R/2

λ<sub>out</sub>

λ<sub>in</sub>

R/2

Causes/costs of congestion: scenario 2

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

![Diagram showing idealization]

R/2

λ<sub>out</sub>

λ<sub>in</sub>

R/2

Transport Layer 3-87

Transport Layer 3-88
Causes/costs of congestion: scenario 2

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

![Diagram showing original and retransmitted data](transport_layer_3-89)

**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

![Diagram showing original and retransmitted data](transport_layer_3-90)
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 \( \lambda_{in} \) and \( \lambda_{in}' \) increase?

A: as red \( \lambda_{in}' \) increases, all arriving blue pkts at upper queue are dropped, blue throughput \( \rightarrow 0 \)
Causes/costs of congestion: scenario 3

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 “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")
  - **NI bit:** no increase in rate (mild congestion)
  - **CI bit:** congestion indication
- RM cells returned to sender by receiver, with bits intact

**Casе 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

$$\text{rate} \approx \frac{cwnd}{\text{RTT}} \text{ bytes/sec}$$

- sender limits transmission:
  $$\text{LastByteSent} - \text{LastByteAcked} \leq cwnd$$

- $cwnd$ 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

- summary: initial rate is slow but ramps up exponentially fast
TCP: detecting, reacting to loss

- loss indicated by timeout:
  - \texttt{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
  - \texttt{cwnd} is cut in half window then grows linearly
- TCP Tahoe always sets \texttt{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 \texttt{cwnd} gets to 1/2 of its value before timeout.

Implementation:
- variable \texttt{ssthresh}
- on loss event, \texttt{ssthresh} is set to 1/2 of \texttt{cwnd} just before loss event
Summary: TCP Congestion Control

slow start

cwnd = ssthresh = cwnd/2

duplicate ACK
dupACKcount++
cwnd > ssthresh
congestion avoidance

cwnd = cwnd + MSS
transmit new segment(s), as allowed
dupACKcount = 0

duplicate ACK
dupACKcount++
cwnd = ssthresh + 3
retransmit missing segment

dupACKcount == 3
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
transmit new segment(s), as allowed
dupACKcount = 0

duplicate ACK
cwnd = cwnd + MSS
transmit new segment(s), as allowed

dupACKcount = 0
retransmit missing segment

dupACKcount == 3
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
transmit new segment(s), as allowed
dupACKcount = 0

duplicate ACK
cwnd = cwnd + MSS
transmit new segment(s), as allowed

dupACKcount = 0
retransmit missing segment

dupACKcount == 3
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
transmit new segment(s), as allowed
dupACKcount = 0

duplicate ACK
cwnd = cwnd + MSS
transmit new segment(s), as allowed

dupACKcount = 0
retransmit missing segment

dupACKcount == 3
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
transmit new segment(s), as allowed
dupACKcount = 0

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 (# in-flight bytes) is 3/4 W
  - avg. throughput is 3/4W per RTT

avg TCP throughput = \( \frac{3}{4} \) W/RTT bytes/sec
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]:
  $$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

⇒ to achieve 10 Gbps throughput, need a loss rate of $L$
  
  $= 2 \cdot 10^{-10} – a \text{ very small loss rate!}$
- 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”