Chapter 3: Transport Layer

Applications
… built on ...

**Reliable (or unreliable) transport**
… built on ...

Best-effort global packet delivery
… built on ...

Best-effort local packet delivery
… built on ...

Physical transfer of bits

The source PowerPoint slides are public available, provided by Authors (JFK/KWR). They are revised for CS536@Purdue.
Roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality
Transport services and protocols

- provide **logical communication** between application processes running on different hosts

- transport protocols actions in end systems:
  - sender: breaks application messages into **segments**, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer

- two transport protocols available to Internet applications
  - TCP, UDP
Transport Layer Actions

Sender:
- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP
Transport Layer Actions

Receiver:
- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket
Transport vs. network layer services and protocols

- **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
Two principal Internet transport protocols

- **TCP**: Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup

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

- services not available:
  - delay guarantees
  - bandwidth guarantees
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality
**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:
- when creating socket, must specify **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 receiving host receives **UDP** segment:
- checks destination port # in segment
- directs UDP segment to socket with that port #

IP/UDP datagrams with **same dest. port #**, but different source IP addresses and/or source port numbers will be directed to **same socket** at receiving host
Connectionless demultiplexing: an example

DatagramSocket mySocket2 = new DatagramSocket (9157);

DatagramSocket serverSocket = new DatagramSocket (6428);

DatagramSocket mySocket1 = new DatagramSocket (5775);

source port: 9157
dest port: 6428

source port: 9157
dest port: 6428

source port: 6428
dest port: 9157

source port: 6428
dest port: 5775

source port: 5775
dest port: 6428

Transport Layer: 3-18
Connection-oriented demultiplexing

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

- demux: receiver uses all four values (4-tuple) to direct segment to appropriate socket

- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client
Connection-oriented demultiplexing: example

Three segments, all destined to IP address: B, dest port: 80 are demultiplexed to different sockets
Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality
UDP: User Datagram Protocol

- “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 RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion
UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3

- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer
UDP: Transport Layer Actions

SNMP client

application
transport (UDP)
network (IP)
link
physical

SNMP server

application
transport (UDP)
network (IP)
link
physical
UDP: Transport Layer Actions

UDP sender actions:
- got an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP
UDP: Transport Layer Actions

UDP receiver actions:
- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket
UDP segment header

UDP segment format

- source port #
- dest port #
- length
- checksum
- application data (payload)

length, in bytes of UDP segment, including header

data to/from application layer

32 bits
**Goal:** detect errors (i.e., flipped bits) in transmitted segment

<table>
<thead>
<tr>
<th></th>
<th>1st number</th>
<th>2nd number</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitted:</td>
<td>5</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>Received:</td>
<td>4</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

receiver-computed checksum ≠ sender-computed checksum (as received)
UDP checksum

**Goal:** detect errors (i.e., flipped bits) in transmitted segment

**sender:**
- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- **checksum:** addition (one’s complement sum) of segment content
- checksum value put into UDP checksum field

**receiver:**
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - Not equal - error detected
  - Equal - no error detected. *But maybe errors nonetheless?* More later ....
Internet checksum: an example

example: add two 16-bit integers

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}
\]

wraparound

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

checksum

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

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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/
example: add two 16-bit integers

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\hline
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

wraparound

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}
\]

digits changed (bit flips), but **no** change in checksum!
Summary: UDP

- "no frills" protocol:
  - segments may be lost, delivered out of order
  - best effort service: "send and hope for the best"

- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)

- build additional functionality on top of UDP in application layer (e.g., HTTP/3)
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality
Principles of reliable data transfer

- important @ application, transport, link layers
  - Reliable transport of packets
    - A single sender and a single receiver
  - Packet delivery imperfect
    - With bit errors, dropping packets, out-of-order delivery, duplicate copies, long delay, ....

Logical end-end **reliable transport**

**sender**
- packets in queue/buffer
- errors

**receiver**
- packets received
- X loss

Packet delivery misbehaviors

Transport Layer: 3-36
Principles of reliable data transfer

reliable service *abstraction*
Principles of reliable data transfer

Reliable service abstraction

Sender-side of reliable data transfer protocol

Receiver-side of reliable data transfer protocol

Transport Layer: 3-38
Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)
Principles of reliable data transfer

Sender, receiver do not know the “state” of each other, e.g., was a message received?
- unless communicated via a message

Sender-side of reliable data transfer protocol

Transport Layer: 3-40
Reliable data transfer protocol (rdt): interfaces

- **rdt_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer.
- **udt_send()**: called by rdt to transfer packet over unreliable channel to receiver.
- **rdt_rcv()**: called when packet arrives on receiver side of channel.
- **deliver_data()**: called by rdt to deliver data to upper layer.

**Bi-directional communication over unreliable channel**

**sender-side implementation of rdt reliable data transfer protocol**

**receiver-side implementation of rdt reliable data transfer protocol**
Reliable data transfer: getting started

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

![Finite State Machine Diagram]

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

**event causing state transition**

**actions taken on state transition**

**event**

**actions**
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
“Stop and Wait” Scenario

- Simple setting: one packet at a time (stop and wait)
  - One sender, one receiver
  - Sender has infinite number of packets to transfer to the receiver
  - Sender starts one-packet transmission at a time, and will not proceed with the next new packet transmission until the current packet has been successfully received & acknowledged by the receiver.

![Diagram showing the "Stop and Wait" scenario](image-url)
“Stop and Wait” Scenario

- We progressively consider more complex cases
  - Bit errors
  - Packet loss
  - Duplicate copies of the same packet
  - Long delay (thus also out of order)
  - ....

- Designs: rdt2.0 (initial) → rdt3.0 (stop & wait)
rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- *the* question: how to recover from errors?

How do humans recover from “errors” during conversation?
How to detect bit errors in packet?
- Internet checksum algorithm

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 packet upon receiving NAK

**new mechanisms in rdt2.0 (beyond rdt1.0):**
- Error detection at receiver
- Feedback from receiver: control messages (ACK, NAK) from receiver to sender
- Retransmission at the sender upon NAK feedback
rdt2.0: FSM specifications

sender

\texttt{rdt\_send(data)}
\texttt{snkpkt = make\_pkt(data, checksum)}
\texttt{udt\_send(sndpkt)}
Wait for call from above
Wait for ACK or NAK
\texttt{udt\_send(NAK)}
\texttt{rdt\_rcv(rcvpkt) && corrupt(rcvpkt)}}

Wait for call from below
extract(rcvpkt, data)
\texttt{deliver\_data(data)}
\texttt{udt\_send(ACK)}
\texttt{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)}}

\texttt{snkpkt = make\_pkt(data, checksum)}
\texttt{udt\_send(sndpkt)}
\texttt{rdt\_rcv(rcvpkt) && isACK(rcvpkt)}}

\texttt{\Lambda}
**rdt2.0: FSM specification**

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

Wait for call from above

Wait for ACK or NAK

rdt_send(NAK)

Wait for call from below

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

rdt_rcv(rcvpkt) && isNAK(rcvpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
```

**Note:** “state” of receiver (did the receiver get my message correctly?) isn’t known to sender unless somehow communicated from receiver to sender
- that’s why we need a protocol!
rdt2.0: operation with no errors

sender

```
rdt_send(data)
```

```
fnkpkt = make_pkt(data, checksum)
```

```
udt_send(sndpkt)
```

```
Wait for call from above
```

```
Wait for ACK or NAK
```

```
rdt_read(rcvpkt) && isACK(rcvpkt)
```

```
extract(rcvpkt, data)
```

```
deliver_data(data)
```

```
udt_send(ACK)
```

```
udt_send(NAK)
```

```
Wait for call from below
```

```
rdt_read(rcvpkt) && isNACK(rcvpkt)
```

```
udt_send(sndpkt)
```

```
Wait for ACK or NAK
```

```
rdt_read(rcvpkt) && corrupt(rcvpkt)
```

```
Wait for call from above
```

```
rvt_rcv(rcvpkt) && notcorrupt(rcvpkt)
```

```
eject(rcvpkt, data)
```

```
deliver_data(data)
```

```
udt_send(ACK)
```

```
udt_send(NAK)
```

Transport Layer: 3-50
**rdt2.0: corrupted packet scenario**

**Sender**
- `rdt_send(data)`
- `snkpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`

**Receiver**
- `wait for call from below`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`

---

Transport Layer: 3-51
rdt2.0 in action

(a) no error

(sender) send pkt0
(rcv) rcv pkt0
(rcv) send ack
(rcv) rcv ack
(sender) send pkt1
(rcv) rcv pkt1
(rcv) send ack
(rcv) rcv ack
(sender) send pkt2
(rcv) rcv pkt2
(rcv) send ack
(rcv) rcv ack

(b) packet with bit errors

(sender) send pkt0
(rcv) rcv pkt0
(rcv) send ack
(rcv) rcv ack
(sender) send pkt1
(rcv) rcv pkt1
(rcv) send ack
(rcv) rcv ack
(sender) send pkt2
(rcv) rcv pkt2
(rcv) send ack
(rcv) rcv ack

(rcv garbled pkt1, drop pkt1)

(sender) resend pkt1
(rcv) rcv nack
(sender) send NACK
(rcv) nack
(sender) resend pkt1
(rcv) rcv pkt1
(rcv) send ack
(sender) send pkt2
(rcv) rcv pkt2
(rcv) 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

---

stop and wait
sender sends one packet, then waits for receiver response
rdt2.0’s flaw: garbled ACK/NACK

(a) Corrupted ack

Sender cannot tell whether the corrupted message is ACK or NACK!
Receiver cannot tell whether the received message is a new packet or a retransmitted packet!

(b) Corrupted NACK

Simply retransmitting upon corrupted ACK/NACK is not sufficient!
**rdt2.1: need seq #!**

(a) Corrupted ack

- **sender**
  - send pkt0
  - rcv ack
  - send pkt1
  - rcv garbled
  - resend pkt1
  - rcv ack
  - send(pkt2)

- **receiver**
  - rcv pkt0
  - send ack
  - rcv pkt1
  - send ack
  - rcv pkt2
  - send ack

(b) Corrupted NACK

- **sender**
  - send pkt0
  - rcv pkt0
  - send ack
  - rcv dup pkt1
  - drop dup pkt1
  - send ack
  - rcv garbled
  - resend pkt1

- **receiver**
  - rcv garbled
  - drop pkt1
  - send NACK
  - rcv pkt1
  - send ack
  - rcv pkt2
  - send ack

Transport Layer: 3-55
rdt2.1: sender, handles garbled ACK/NAKs

[Diagram showing the flow of data and acknowledgments, including the handling of corrupted packets and NAKs.]
rdt2.1: receiver, handles garbled ACK/NAKs

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
 extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && not corrupt(rcvpkt) &&
has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```
## Summary: reliable data transfer

<table>
<thead>
<tr>
<th>Version</th>
<th>Channel</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdt1.0</td>
<td>Reliable channel</td>
<td>nothing</td>
</tr>
<tr>
<td>rdt2.0</td>
<td>bit errors</td>
<td>(1) error detection via checksum</td>
</tr>
<tr>
<td></td>
<td>(no loss)</td>
<td>(2) receiver feedback (ACK/NAK)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3) retransmission upon NAK</td>
</tr>
<tr>
<td>rdt2.1</td>
<td>Same as 2.0</td>
<td>handling fatal flaw with rdt 2.0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(4) need seq #. for each packet</td>
</tr>
</tbody>
</table>

Transport Layer: 3-60
rdt2.1: 1-bit seq # is enough!

(a) no error

Sender
send pkt0
rcv ack
send pkt1
rcv ack1
send pkt0 *(new pkt!)*

receiver
rcv pkt0
send ack
pkt0
ack
pkt1
ack
pkt0
ack

(b) packet with bit errors

sender
send pkt0
rcv ack
send pkt1
rcv NACK
resend pkt1
rcv ack
send pkt0 *(new pkt!)*

receiver
send pkt0
rcv pkt0
send ack
pkt0
ack
pkt1
ack
pkt1
ack
pkt0
ack

rcv garbled pkt1
drop pkt1
send NACK
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*

As we will see, TCP uses this approach to be NAK-free.
rdt2.2: NAK-free

(a) Corrupted ack

(b) dup ack for garbled 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
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)`
- `udt_send(sndpkt)`

**Receiver FSM Fragment**
- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))`  
- `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: 3-66
## Summary: reliable data transfer

<table>
<thead>
<tr>
<th>Version</th>
<th>Channel</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdt1.0</td>
<td>Reliable channel</td>
<td>nothing</td>
</tr>
<tr>
<td>rdt2.0</td>
<td>bit errors (no loss)</td>
<td>(1) error detection via checksum, (2) receiver feedback (ACK/NAK), (3) retransmission upon NAK</td>
</tr>
<tr>
<td>rdt2.1</td>
<td>Same as 2.0 (fatal flaw)</td>
<td>(4) seq# (1 bit, 0/1) for each pkt</td>
</tr>
<tr>
<td>rdt2.2</td>
<td>Same as 2.0</td>
<td>A variant to rdt2.1 (no NAK), Duplicate ACK = NAK</td>
</tr>
</tbody>
</table>
rdt3.0: channels with errors and loss

**New channel assumption:** underlying channel can also lose packets (data, ACKs)
- checksum, sequence #s, ACKs, retransmissions will be of help ... but not quite enough

**Q:** How do humans handle lost sender-to-receiver words in conversation?
rdt3.0: channels with errors and loss

**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 packet being ACKed
- use countdown timer to interrupt after “reasonable” amount of time
**rdt3.0 sender**

- rdt_send(data)
  - sndpkt = make_pkt(0, data, checksum)
  - udt_send(sndpkt)
  - start_timer
- rdt_rcv(rcvpkt)
  - stop_timer
- Wait for call 0 from above

- rdt_send(data)
  - sndpkt = make_pkt(1, data, checksum)
  - udt_send(sndpkt)
  - start_timer
- rdt_rcv(rcvpkt)
  - && notcorrupt(rcvpkt)
  - && isACK(rcvpkt,0)
  - && isACK(rcvpkt,1)
  - stop_timer
- Wait for ACK0

- Wait for call 1 from above

- rdt_send(data)
  - sndpkt = make_pkt(0, data, checksum)
  - udt_send(sndpkt)
  - start_timer
- rdt_rcv(rcvpkt)
  - && notcorrupt(rcvpkt)
  - && isACK(rcvpkt,0)
  - && isACK(rcvpkt,1)
  - stop_timer
- Wait for ACK1
rdt3.0 sender

- \texttt{sendpkt} = \texttt{make_pkt(0, data, checksum)}
- \texttt{udt\_send(sndpkt)}
- \texttt{start\_timer}

\textbf{Wait for call 0 from above}

- \texttt{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)}
- \texttt{stop\_timer}
- \texttt{udt\_send(sndpkt)}

\textbf{Wait for ACK0}

- \texttt{rdt\_rcv(rcvpkt)} \&\& ( corrupt(rcvpkt) || isACK(rcvpkt,1) )
- \texttt{start\_timer}

\textbf{Wait for ACK1}

- \texttt{rdt\_rcv(rcvpkt) && isACK(rcvpkt,1)}
- \texttt{stop\_timer}
- \texttt{udt\_send(sndpkt)}

\textbf{Wait for call 1 from above}

- \texttt{rdt\_rcv(rcvpkt) \&\& notcorrupt(rcvpkt) \&\& isACK(rcvpkt,0)}
- \texttt{stop\_timer}
- \texttt{udt\_send(sndpkt)}

- \texttt{rdt\_send(data)}
- \texttt{sndpkt = make_pkt(1, data, checksum)}
- \texttt{udt\_send(sndpkt)}
- \texttt{start\_timer}
Example: rdt3.0 in action

(a) no loss

(b) packet loss
rdt3.0 in action

**sender**
- send pkt0
- rcv ack0
- send pkt1
- rcv ack1
- send pkt0

**receiver**
- rcv pkt0
- send ack0
- rcv pkt1
- send ack1
- rcv pkt0
- send ack0

(c) ACK loss

**sender**
- send pkt0
- rcv ack0
- send pkt1
- rcv ack1
- send pkt0
- rcv ack1
- send pkt1

**receiver**
- rcv pkt0
- send ack0
- rcv pkt1
- send ack1
- rcv pkt0
- send ack0

(d) premature timeout/ delayed ACK
# Summary: reliable data transfer

<table>
<thead>
<tr>
<th>Version</th>
<th>Channel</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdt1.0</td>
<td>Reliable channel</td>
<td>nothing</td>
</tr>
</tbody>
</table>
| rdt2.0  | bit errors (no loss)| (1) error detection via checksum   
|         |                     | (2) receiver feedback (ACK/NAK)                                           |
|         |                     | (3) retransmission upon NAK                                               |
| rdt2.1  | Same as 2.0         | (4) seq# (1 bit) for each pkt                                             |
| rdt2.2  | Same as 2.0         | A variant to rdt2.1 (no NAK)                                              |
| Rdt3.0  | Bit errors + loss   | (5) retransmission upon timeout                                           |
|         |                     | No NAK, only ACK                                                         |

Rdt3.0

Transport Layer: 3-76
Performance of rdt3.0 (stop-and-wait)

- $U_{\text{sender}}$: *utilization* – fraction of time sender busy sending

- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  
  - time to transmit packet into channel:

  $$D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$
rdt3.0: stop-and-wait operation

first packet bit transmitted, $t = 0$

ACK arrives, send next packet, $t = \text{RTT} + \frac{L}{R}$

first packet bit arrives

last packet bit arrives, send ACK

Transport Layer: 3-78
rdt3.0: stop-and-wait operation

\[ U_{\text{sender}} = \frac{L / R}{RTT + L / R} \]

= \frac{0.008}{30.008}

= 0.00027

- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)
Mechanisms for reliable data transfer

- **Error detection**
  - via algorithms such as Internet checksum (in UDP), CRC (later in Chapter 6)

- **Receiver feedback via (ACK + sequence #)**
  - Duplicate ACK = negative acknowledgment

- **Timer & sequence # for each transmitted packet**
  - Number of seq. #: $\geq 2$ for stop & wait protocol
  - Timeout not too small, not too big ($\approx RTT$)

- **Retransmission upon timeout or duplicate ACK (i.e., negative ACK)**
rdt3.0: pipelined protocols operation

**pipelining**: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver
Pipelining: increased utilization

3-packet pipelining increases utilization by a factor of 3!

\[
U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081
\]
Go-Back-N: sender

- **sender**: “window” of up to \(N\), consecutive transmitted but unACKed pkts
  - \(k\)-bit seq # in pkt header

  ![Diagram showing window size, send_base, nextseqnum, already ack'ed, sent, not yet ack'ed, usable, not yet sent, not usable]

- **cumulative ACK**: \(\text{ACK}(n)\): ACKs all packets up to, including seq # \(n\)
  - on receiving \(\text{ACK}(n)\): move window forward to begin at \(n+1\)

- timer for oldest in-flight packet

- **timeout\((n)\)**: retransmit packet \(n\) and all higher seq # packets in window
Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest in-order seq #
  - may generate duplicate ACKs
  - need only remember rcv_base

- on receipt of out-of-order packet:
  - can discard (don’t buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:

- received and ACKed
- Out-of-order: received but not ACKed
- Not received
Go-Back-N in action: No loss

**sender window (N=4)**

```
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
```

**sender**

- send pkt0
- send pkt1
- send pkt2
- send pkt3 (wait)

**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

**Timeouts**

- pkt0 timeout
- pkt1 timeout
- pkt2 timeout
- pkt3 timeout
- pkt4 timeout

**Transport Layer:** 3-86
Go-Back-N in action: Loss

**sender window (N=4)**

<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

**sender**

- send pkt0
- send pkt1
- send pkt2
- send pkt3 (wait)
- send pkt2
- send pkt3
- send pkt4
- send pkt5

**receiver**

- receive pkt0, send ack0
- receive pkt1, send ack1
- receive pkt3, discard, (re)send ack1
- receive pkt4, discard, (re)send ack1
- receive pkt5, discard, (re)send ack1
- rcv pkt2, deliver, send ack2
- rcv pkt3, deliver, send ack3
- rcv pkt4, deliver, send ack4
- rcv pkt5, deliver, send ack5

**pkt 2 timeout**

- ignore duplicate ACK

Transport Layer: 3-87
Selective repeat

- Receiver *individually* acknowledges all correctly received packets
  - Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender times-out/retransmits individually for unACKed packets
  - Sender maintains timer for each unACKed pkt
- Sender window
  - $N$ consecutive seq #s
  - Limits seq #s of sent, unACKed packets
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers
Selective repeat: sender and receiver

**sender**

Data from above:
- if next available seq # in window, send packet

Timeout($n$):
- resend packet $n$, restart timer

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

**receiver**

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

Packet $n$ in [rcvbase-N, rcvbase-1]
- ACK($n$)

Otherwise:
- ignore
Selective Repeat in action

sender window (N=4)

<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

sender

send pkt0
send pkt1
send pkt2
send pkt3 (wait)

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

Q: what happens when ack2 arrives?
Selective repeat: a dilemma!

Example:
- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

(a) No problem

(b) Oops!

Transport Layer: 3-92
Selective repeat: a dilemma!

example:
- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?
Selective repeat: dilemma (N+1)

Example:
- Window size = 3
- Seq #':s: 0, 1, 2, 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)?

2N
### Summary: reliable data transfer

<table>
<thead>
<tr>
<th>Version</th>
<th>Channel</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdt1.0</td>
<td>No error/loss</td>
<td>nothing</td>
</tr>
<tr>
<td>rdt2.0</td>
<td>bit errors</td>
<td>(1) error detection via checksum</td>
</tr>
<tr>
<td></td>
<td>(no loss)</td>
<td>(2) receiver feedback (ACK/NAK)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3) retransmission upon NAK</td>
</tr>
<tr>
<td>rdt2.1</td>
<td>Same as 2.0</td>
<td>(4) seq# (1 bit) for each pkt</td>
</tr>
<tr>
<td>rdt2.2</td>
<td>Same as 2.0</td>
<td>(no NAK): Unexpected ACK = NAK</td>
</tr>
<tr>
<td>Rdt3.0</td>
<td>errors + loss</td>
<td>(5) Retransmission upon timeout; ACK-only</td>
</tr>
</tbody>
</table>

#### Performance issue: low utilization

- **Goback-N**: Same as 3.0
  - N sliding window (pipeline)
  - Discard out-of-order pkts (recovery)
- **Selective Repeat**: Same as 3.0
  - N sliding window, selective recovery
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control
TCP: overview  RFCs: 793, 1122, 2018, 5681, 7323

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

- reliable, in-order byte steam:
  - no “message boundaries"

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size

- cumulative ACKs

- pipelining:
  - TCP congestion and flow control set window size

- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange

- flow controlled:
  - sender will not overwhelm receiver
TCP segment structure

- **source port #**
- **dest port #**
- **sequence number**
- **acknowledgement number**
- **receive window**
- **segment seq #:** counting bytes of data into bytestream (not segments!)
- **flow control:** # bytes receiver willing to accept
- **checkum**
- **Internet checksum**
- **segment seq #:** counting bytes of data into bytestream (not segments!)
- **options (variable length)**
- **UCF, E:** congestion notification
- **TCP options**
- **RST, SYN, FIN:** connection management
- **data sent by application into TCP socket**

**TCP segment structure**

- **32 bits**
- **sequence number**
- **segment seq #:** counting bytes of data into bytestream (not segments!)
- **flow control:** # bytes receiver willing to accept
- **application data**
- **data sent by application into TCP socket**

**ACK:** seq # of next expected byte; A bit: this is an ACK

**length (of TCP header)**

**Internet checksum**

**C, E:** congestion notification

**TCP options**

**RST, SYN, FIN:** connection management
**TCP sequence 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 sequence numbers, ACKs

simple telnet scenario

User types ‘C’
host ACKs receipt of echoed ‘C’

Seq=42, ACK=79, data = ‘C’
Seq=79, ACK=43, data = ‘C’
Seq=43, ACK=80

host ACKs receipt of ‘C’, echoes back ‘C’
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 (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 1/8

![Graph showing RTT values over time](RTT: gaia.cs.umass.edu to fantasia.eurecom.fr)
TCP round trip time, timeout

- Timeout interval: $\text{EstimatedRTT}$ plus “safety margin”
  - large variation in $\text{EstimatedRTT}$: want a larger safety margin

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

- $\text{DevRTT}$: EWMA of $\text{SampleRTT}$ deviation from $\text{EstimatedRTT}$:

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

  (typically, $\beta = 1/4$)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/
TCP Sender (simplified)

**event: data received from application**
- 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

**event: timeout**
- retransmit segment that caused timeout
- restart timer

**event: ACK received**
- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments
TCP Receiver: ACK generation [RFC 5681]

**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 expected seq.
- Gap detected

**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
- Send duplicate ACK, indicating seq. # of next expected byte
- Immediate send ACK, provided that segment starts at lower end of gap
TCP: retransmission scenarios

Host A — Host B
Seq=92, 8 bytes of data
ACK=100

lost ACK scenario

SendBase=100
SendBase=120
SendBase=120

prefix timeout

Host A — Host B
Seq=92, 8 bytes of data
ACK=100

send cumulative
ACK for 120

Transport Layer: 3-106
TCP: retransmission scenarios

Host A

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

Host B

ACK=120
Seq=120, 15 bytes of data

cumulative ACK covers for earlier lost ACK
TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #
- likely that unACKed segment lost, so don’t wait for timeout

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer

- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management

- Principles of congestion control
- TCP congestion control
TCP flow control

flow control
receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast.
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 autoadjust `RcvBuffer`
- sender limits amount of unacknowledged (“in-flight”) data to receiver’s `rwnd` value
- guarantees receive buffer will not overflow
TCP connection management

before exchanging data, sender/receiver “handshake”:
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)

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
TCP 3-way handshake

**Client state**

clientSocket = socket(AF_INET, SOCK_STREAM)

LISTEN

clientSocket.connect((serverName,serverPort))

SYNSENT

choose init seq num, \( x \) send TCP SYN msg

ESTAB

received SYNACK(\( x \)) indicates server is live; send ACK for SYNACK; this segment may contain client-to-server data

**Server state**

serverSocket = socket(AF_INET,SOCK_STREAM)

serverSocket.bind(('',serverPort))

serverSocket.listen(1)

connectionSocket, addr = serverSocket.accept()

LISTEN

SYN RCVD

choose init seq num, \( y \) send TCP SYNACK msg, acking SYN

SYNbit=1, Seq=\( x \)

SYNbit=1, Seq=\( y \)

ACKbit=1; ACKnum=\( x+1 \)

ACKbit=1, ACKnum=\( y+1 \)

received ACK(\( y \)) indicates client is live

ESTAB

clientSocket.connect(((serverName,serverPort))

serverSocket = socket(AF_INET,SOCK_STREAM)

serverSocket.bind(('',serverPort))

serverSocket.listen(1)

connectionSocket, addr = serverSocket.accept()
How to set SYNC, ACK bit?

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence number</td>
<td></td>
</tr>
<tr>
<td>acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>receive window</td>
<td></td>
</tr>
<tr>
<td>checksum</td>
<td></td>
</tr>
<tr>
<td>Urg data pointer</td>
<td></td>
</tr>
</tbody>
</table>

options (variable length)

application data (variable length)

ACK: ACK # valid
RST, SYN, FIN: connection estab (setup, teardown commands)
Closing a TCP 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
Closing TCP connection (i.e., two 1-way subconnections)

**client state**
- ESTAB
- FIN_WAIT_1
  - clientSocket.close()
  - can no longer send but can receive data
- FIN_WAIT_2
  - wait for server close
- TIMED_WAIT
  - timed wait for 2*max segment lifetime
- CLOSED
  - Makes the client wait for a duration long enough for an ACK to be lost and a FIN to arrive. If a FIN arrives, restart the timer 2*max-segment-lifetime.
  - Drop any delayed segments during timer=2*max-segment-time (2min default)

**server state**
- ESTAB
- CLOSE_WAIT
- LAST_ACK
- CLOSED
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality
Principles of congestion control

Congestion:

- informally: “too many sources sending too much data too fast for network to handle”
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!
Approaches towards congestion control

End-end congestion control:

- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP
Approaches towards congestion control

Network-assisted congestion control:

- routers provide direct feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECBIT protocols
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality
TCP Congestion Control

- **Idea**
  - Assumes best-effort network
  - Each source determines network capacity for itself
  - Implicit feedback via ACKs or timeout events
    - Feedback control system in practice
  - ACKs pace transmission (self-clocking)

- **Challenge**
  - Determining *initial* available capacity
  - Adjusting to changes in capacity in a *timely* manner
TCP Congestion Control

Assumptions for congestion control
- TCP pipelined reliable data transfer (SR in the common cases)
- Works with TCP flow control
- \textbf{All losses of TCP segments} are due to Internet \textit{congestion}
  - Ignore the transmission errors (since link quality is good in general)

Mechanism: Window-based congestion control
- Adjust the window size for SR to change the TCP sending rate

Changes in congestion window size (\textit{cwnd})
- Slow increases to absorb new bandwidth
- Quick decreases to eliminate congestion
TCP Congestion Control

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

  - \( \text{cwnd} \) is dynamic, function of perceived network congestion

**How does sender perceive congestion?**

- loss event = timeout or 3 duplicate acks
- TCP sender reduces rate \((\text{cwnd})\) after loss event

**three mechanisms:**

- AIMD: how to grow cwnd
- slow start: startup
- conservative after loss (timeout, duplicate ACKs) events
AIMD Rule: additive increase, multiplicative decrease

- **Approach:** increase 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 $cwnd$ by 50% after loss

Saw tooth behavior: probing for bandwidth
Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally
TCP Congestion Control (RFC 5681)

How to implement TCP Congestion Control?

Multiple algorithms work together:
- slow start: **how to jump-start**
- congestion avoidance: **additive increase**
- fast retransmit/fast recovery: recover from single packet loss: **multiplicative decrease**
- retransmission upon timeout: **conservative loss/failure handling**
TCP Slow Start

- When connection begins, \( cwnd \leq 2 \text{ MSS} \), typically, set \( cwnd = 1\text{MSS} \)
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps

- available bandwidth may be \( \gg \) MSS/RTT
  - desirable to quickly ramp up to respectable rate

- When connection begins, increase rate exponentially fast until \( cwnd \) reaches a threshold value: slow-start-threshold \( ssthresh \)
  - \( cwnd < ssthresh \)
TCP Slow Start (more)

- When connection begins, increase rate exponentially when $cwnd < ssthresh$
  - Goal: double $cwnd$ every RTT by setting
  - Action: $cwnd += 1$ MSS for every ACK received

- **Summary**: initial rate is slow but ramps up exponentially fast
Congestion Avoidance

- Goal: increase cwnd by 1 MSS per RTT until congestion (loss) is detected
  
  - Conditions: when cwnd > ssthresh and no loss occurs

  - Actions: $\text{cwnd} += \left(\frac{\text{MSS}}{\text{cwnd}}\right)\times\text{MSS (bytes)}$ upon every incoming non-duplicate ACK
## TCP Congestion Control

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>condition</th>
<th>Design</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow Start</td>
<td>cwnd $\leq$ ssthresh;</td>
<td>cwnd doubles per RTT</td>
<td>cwnd+=1MSS per ACK</td>
</tr>
<tr>
<td>Congestion Avoidance</td>
<td>cwnd &gt; ssthresh</td>
<td>cwnd++ per RTT (additive increase)</td>
<td>cwnd+=1/cwnd * MSS per ACK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When loss occurs

Detecting losses and reacting to them:

• through duplicate ACKs
  • fast retransmit / fast recovery
    • Goal: multiplicative decrease cwnd upon loss

• through retransmission timeout
  • Goal: reset everything
Fast Retransmit/Fast Recovery

- **fast retransmit**: to detect and repair loss, based on incoming duplicate ACKs
  - **use** 3 duplicate ACKs to infer packet loss
  - set ssthresh = max(cwnd/2, 2MSS)
  - cwnd = ssthresh + 3MSS
  - retransmit the lost packet

- **fast recovery**: governs the transmission of new data until a non-duplicate ACK arrives
  - **increase** cwnd by 1 MSS upon every duplicate ACK

---

**Philosophy:**
- 3 dup ACKs to infer losses and differentiate from transient out-of-order delivery
- What about only 1 or 2 dup ACKs?
  - Do nothing; this allows for transient out-of-order delivery

- receiving each duplicate ACK indicates one more packet left the network and arrived at the receiver

Transport Layer: 3-165
Algorithm for fast rexmit/fast recovery

- Initially, fastretx = false;
- If upon 3rd duplicate ACK
  - ssthresh = max (cwnd/2, 2*MSS)
  - cwnd = ssthresh + 3*MSS
    - why add 3 packets here?
  - retransmit the lost TCP packet
  - Set fastretx = true;
- If fastretx == true; upon each *additional* duplicate ACK
  - cwnd += 1 MSS
  - transmit a new packet if allowed
    - by the updated cwnd and rwnd
- If fastretx == true; upon a new (i.e., non-duplicate) ACK
  - cwnd = ssthresh
  - Fastretx = false; // After fast retransmit/fast recovery, cwnd decreases by half
Retransmission Timeout

when retransmission timer expires

- $\text{ssthresh} = \max \left( \frac{\text{cwnd}}{2}, \ 2^{*}\text{MSS} \right)$
  - cwnd should be flight size to be more accurate
  - see RFC 2581

- $\text{cwnd} = 1 \text{ MSS}$

- retransmit the lost TCP packet

- why resetting?
  - heavy loss detected
TCP Congestion Window Trace

- **threshold**
- **congestion window**
- **timeouts**
- **fast retransmission**
- **additive increase**
- **slow start period**
### TCP Congestion Control Summary

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>condition</th>
<th>Design</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow Start</td>
<td>cwnd &lt;= ssthresh;</td>
<td>cwnd doubles per RTT</td>
<td>cwnd+=1MSS per ACK</td>
</tr>
<tr>
<td>Congestion Avoidance</td>
<td>cwnd &gt; ssthresh</td>
<td>cwnd++ per RTT (additive increase)</td>
<td>cwnd+=1/cwnd * MSS per ACK</td>
</tr>
<tr>
<td>fast retransmit</td>
<td>3 duplicate ACK</td>
<td>reduce the cwnd by half (multicative</td>
<td>ssthresh = max(cwnd/2,2) cwnd = ssthresh + 3 MSS; retx the lost packet</td>
</tr>
<tr>
<td>fast recovery</td>
<td>receiving a new ACK</td>
<td>finish the 1/2 reduction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>after fast retx</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>upon a dup ACK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after fast retx,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>before fast recovery</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTO timeout</td>
<td>time out</td>
<td>Reset everything</td>
<td>ssthresh = max(cwnd/2,2) cwnd = 1; retx the lost packet</td>
</tr>
</tbody>
</table>

Transport Layer: 3-169
Putting Things Together in TCP

How Selective repeat, congestion control, flow control work together:

- use **selective repeat** to do reliable data transfer for a window of packets $\textit{win}$ at any time

- update $\textit{win} = \text{min} (\textit{cwnd}, \textit{rwnd})$
  - cwnd is updated by TCP congestion control
  - rwnd is updated by TCP flow control

- Example: cwnd = 20; rwnd = 10
  - Then $\textit{win}=10$
Illustrative Example
Example Setting

- Use all following TCP congestion control algorithms:
  - Slow start
  - Congestion avoidance (CA)
  - Fast retransmit/fast recovery
  - Retransmission timeout (say, RTO=500ms)

- When cwnd=ssthresh, use slow start algorithm (instead of CA)

- Assume rwnd is always large enough, then the send window size min(rwnd,cwnd) = cwnd

- Assume 1 acknowledgement per packet (i.e., no delayed ACK is used), and we use TCP cumulative ACK (i.e., ACK # = (largest sequence # received in order at the receiver + 1))

- Assume each packet size is 1 unit (1B) for simple calculation

- TCP sender has infinite packets to send, 1, 2, 3, 4, 5,....

- Assume packet #5 is lost once

- Assume that the receiver will buffer out of order packets (like selective repeat)

We will how TCP congestion control algorithms work together
CC algorithm
slow start
ssh = 4

slow start (upon ack2)
ssh = 4

slow start (upon ack3)
ssh = 4

slow start (upon ack4)
ssh = 4

slow start (upon ack5)
ssh = 4

slow start (upon ack6)
ssh = 4

SSH after algo runs

cwnd = 1
1

cwnd = 1 + 1 = 2
2

Do nothing upon ack5 (1st dup)

Do nothing upon ack5 (2nd dup)

Fast retransmit (upon 3 dup ack5)
ssh = max(2, 5/2) = 2.5 \approx 2

Fast recovery w/ additional dup ACK (upon 4th dup)
ssh = 2, cwnd = 5 + 1 = 6

send pkt 10

Pkt 10

pkt 1

ack2

pkt 2

ack3

pkt 3

ack4

pkt 4

ack5 (1st dup)

pkt 5

ack5 (2nd dup)

pkt 6

ack5 (3rd dup)

pkt 7

ack5 (4th dup)

pkt 8

ack5 (5th dup)

pkt 9

ack10

ack11

Pkt 5

Transport Layer: 3-173
CC algorithm

Fast recovery with additional dup ACK (upon 4th dup)

ssh = 2, cwnd = 5 + 1 = 6; send pkt 10

Fast recovery with a new ACK (upon ack10)

ssh = 2
cwnd = ssh = 2
Fast retransmission/fast recovery is over

Slow start also upon ack10

ssh = 2
cwnd = 2 + 1 = 3
Send new packet 12

Congestion avoidance upon ack11

ssh = 2
cwnd = 3 + 1/3 ≈ 3
Send packet 13

Congestion avoidance upon ack 12

ssh = 2
cwnd = 3 + 2/3 = 3; send pkt 14

Congestion avoidance upon ack 13

ssh = 2
cwnd = 3 + 3/3 = 4
Send packets 15, 16

Fast recovery with additional dup ACK (upon 4th dup)

ssh = 2, cwnd = 5 + 1 = 6; send pkt 10

Transport Layer: 3-174
Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality
Evolving transport-layer functionality

- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long, fat pipes (large data transfers)</td>
<td>Many packets “in flight”; loss shuts down pipeline</td>
</tr>
<tr>
<td>Wireless networks</td>
<td>Loss due to noisy wireless links, mobility; TCP treat this as congestion loss</td>
</tr>
<tr>
<td>Long-delay links</td>
<td>Extremely long RTTs</td>
</tr>
<tr>
<td>Data center networks</td>
<td>Latency sensitive</td>
</tr>
<tr>
<td>Background traffic flows</td>
<td>Low priority, “background” TCP flows</td>
</tr>
</tbody>
</table>

- moving transport–layer functions to application layer, on top of UDP
  - HTTP/3: QUIC
QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)
QUIC: Quick UDP Internet Connections

adopts approaches we’ve studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control**: “Readers familiar with TCP’s loss detection and congestion control will find algorithms here that parallel well-known TCP ones.” [from QUIC specification]

- **connection establishment**: reliability, congestion control, authentication, encryption, state established in one RTT

- multiple application-level “streams” multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control
QUIC: Connection establishment

TCP handshake (transport layer) + TLS (security) = 2 serial handshakes

TCP (reliability, congestion control state) + TLS (authentication, crypto state)

QUIC handshake: reliability, congestion control, authentication, crypto state

QUIC: 1 handshake
QUIC: streams: parallelism, no HOL blocking

(a) HTTP 1.1
Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

- instantiation, implementation in the Internet
  - UDP
  - TCP

Up next:

- leaving the network “edge” (application, transport layers)
- into the network “core”
- two network-layer chapters:
  - data plane
  - control plane