Transport layer

Our goals:

- Understand principles behind transport layer services:
  - Multiplexing/demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control

- Learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control
Transport layer: Outline

- 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
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 sending letters to 12 kids

- Processes = kids
- App messages = letters in envelopes
- Hosts = houses
- Transport protocol = Ann and Bill
- Network-layer protocol = postal service
Internet transport-layer protocols

- 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
Transport layer: Outline

- 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
Multiplexing/demultiplexing

Demultiplexing at rcv host:
delivering received segments to correct socket

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

= socket  = process

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

host 1

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

host 2

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

host 3
Connectionless demultiplexing

- Create sockets with port numbers:
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(99111);
  DatagramSocket mySocket2 = new DatagramSocket(99222);
  ```

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

- When host receives UDP segment:
  - Checks destination port number in segment
  - Directs UDP segment to socket with that port number

- IP datagrams with different source IP addresses and/or source port numbers directed to same socket
DatagramSocket serverSocket = new DatagramSocket(6428);

SP provides “return address”
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number

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

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

- Web servers have different sockets for each connecting client:
  - Non-persistent HTTP will have different socket for each request
Connection-oriented demux (cont.)

Client IP: A

Server IP: C

Client IP: B
Connection-oriented demux: Threaded Web Server
Transport layer: Outline

- 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
UDP: User Datagram Protocol [RFC 768]

- "No frills," "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be:
  - Lost
  - Delivered out of order to app
- Connectionless:
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

Why is there a UDP?
- No connection establishment (which can add delay)
- Simple: no connection state at sender, receiver
- Small segment header
- No congestion control: UDP can blast away as fast as desired
UDP: More

- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses (why?):
  - DNS
  - SNMP
- Reliable transfer over UDP:
  - add reliability at application layer
    - Application-specific error recover!

**UDP segment format**

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

Length, in bytes of UDP segment, including header

Application data (message)

UDP segment format
UDP checksum

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

Sender:
- Treat segment contents as sequence of 16-bit integers
- Checksum: addition (1’s complement sum) of segment contents
- Sender puts checksum value into UDP checksum field

Receiver:
- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. But *maybe errors nonetheless?* More later ....
Internet Checksum Example

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

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

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

Wraparound sum:

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

Checksum:

```
0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```
Transport layer: Outline

- 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
Principles of reliable data transfer

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

Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Principles of reliable data transfer

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

Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Reliable data transfer: Getting started

**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

**deliver_data()**: called by rdt to deliver data to upper

**rdt_rcv()**: called when packet arrives on rcv-side of channel
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 transition 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

---

sender

receiver
Rdt2.0: Channel with bit errors

- Underlying channel may flip bits in packet
  - Recall: UDP 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
  - Human scenarios using ACKs, NAKs?

New mechanisms in rdt2.0 (beyond rdt1.0):
- Error detection
- Receiver feedback: control msgs (ACK, NAK) rcvr->sender
Rdt2.0: Operation with no errors

1. `rdt_send(data)`
2. `snkpkt = make_pkt(data, checksum)`
3. `udt_send(sndpkt)`
4. `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
5. `udt_send(sndpkt)`
6. `rdt_send(sndpkt)`
7. `Wait for call from above`
Rdt2.0: Error scenario

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

Wait for call from above

- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`

Wait for ACK or NAK

- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `udt_send(sndpkt)`

Wait for ACK or NAK

- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`

Wait for call from below

- `rdt_rcv(rcvpkt) && corrupt(rcvpkt)`
- `udt_send(NAK)`
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

What to do?
- Sender ACKs/NAKs receiver’s ACK/NAK? What if sender ACK/NAK lost?
- Retransmit, but this might cause retransmission of correctly received pkt!

Handling duplicates:
- Sender retransmits current pkt if ACK/NAK garbled
- 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 with garbled ACK/NAKs

```
rdt_send(data)

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

wait for call 0 from above
rdt_send(data)

wait for ACK or NAK 0
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
( corrupt(rcvpkt) ||
isNAK(rcvpkt) )

udt_send(sndpkt)

wait for call 1 from above
wait for ACK or NAK 1
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ

rdt_send(data)

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

wait for call 1 from above
wait for ACK or NAK 1
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ
```
Rdt2.1: Receiver with garbled ACK/NAKs

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
```
Rdt2.1: Discussion

**Sender:**
- Seq # added to pkt
- Two seq. #’s (0,1) will suffice. Why?
- Must check if received ACK/NAK corrupted
- Twice as many states
  - State must “remember” whether “current” pkt has 0 or 1 seq. #

**Receiver:**
- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected pkt seq #
- Note: receiver can *not* know if its last ACK/NAK received OK at sender
Rdt2.2: A NAK-free protocol

- Same functionality as rdt2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt received OK
  - Receiver must *explicitly* include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: *retransmit current pkt*
Rdt2.2: Sender, receiver fragments

sender FSM fragment

\[
\text{rdt\_send(data)}
\]
\[
sndpkt = \text{make\_pkt}(0, \text{data}, \text{checksum})
\]
\[
\text{udt\_send(sndpkt)}
\]
\[
\text{Wait for call 0 from above}
\]
\[
\text{Wait for ACK 0}
\]

receiver FSM fragment

\[
\text{rdt\_rcv(rcvpkt) \&\& (corrupt(rcvpkt) || has\_seq1(rcvpkt))}
\]
\[
\text{udt\_send(sndpkt)}
\]
\[
\text{Wait for 0 from below}
\]
\[
\text{rdt\_rcv(rcvpkt) \&\& notcorrupt(rcvpkt)}
\]
\[
\text{\&\& has\_seq1(rcvpkt)}
\]
\[
\text{extract(rcvpkt, data)}
\]
\[
\text{deliver\_data(data)}
\]
\[
\text{sndpkt = make\_pkt(ACK1, checksum)}
\]
\[
\text{udt\_send(sndpkt)}
\]
\[
\Lambda
\]
Rdt3.0: Channels with errors and loss

New assumption: Underlying channel can also lose packets (data or ACKs)
- Checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: How to deal with loss?
- Sender waits until certain data or ACK lost, then retransmits

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 use of seq. #’s already handles this
  - Receiver must specify seq # of pkt being ACKed
- Requires countdown timer
Rdt3.0 sender

```
# Note: The diagram represents the Rdt3.0 sender protocol.
# The code snippets correspond to the states and transitions in the diagram.

# Rdt3.0 sender

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

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

Wait for call 0 from above

Wait for ACK0
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

Wait for call 1 from above
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer

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

timeout
udt_send(sndpkt)
start_timer

timeout
udt_send(sndpkt)
start_timer
```

Lambda
Rdt3.0 in action

(a) operation with no loss

(b) lost packet
Rdt3.0 in action (cont.)

(c) lost ACK

(d) premature timeout
Performance of rdt3.0

- Rdt3.0 works, but performance stinks
- Example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

\[ T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8 \frac{kb}{pkt}}{10^9 \frac{b}{s}} = 8 \mu s \]

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

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

- 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
- Network protocol limits use of physical resources
Rdt3.0: Stop-and-wait operation

First packet bit transmitted, \( t = 0 \)

Last packet bit transmitted, \( t = \frac{L}{R} \)

First packet bit arrives

Last packet bit arrives, send ACK

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

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

First packet bit transmitted, \( t = 0 \)
Last bit transmitted, \( t = \frac{L}{R} \)

RTT

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

Increase utilization by a factor of 3!

\[
U_{\text{sender}} = \frac{3 \cdot L/R}{RTT + L/R} = \frac{0.024 \text{ms}}{30.008 \text{ms}} = 0.00079
\]
Go-Back-N

Sender:
- “Window” of up to N, consecutive unack’ed pkts allowed
- Why limit to N? → later (flow control, congestion control)
- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - May deceive duplicate ACKs (see receiver)
- Timer for each in-flight pkt
- Timeout(n): Retransmit pkt n and all higher seq # pkts in window
- K-bit seq # in pkt header ( ⇒ N < 2^K)
GBN: Sender extended FSM

\[ \text{rdt}_\text{send} \text{(data)} \]

\[
\text{if } (\text{nextseqnum} < \text{base}+N) \{ \\
\text{sndpkt}[\text{nextseqnum}] = \text{make_pkt}(\text{nextseqnum}, \text{data}, \text{checksum}) \\
\text{udt}_\text{send}(\text{sndpkt}[\text{nextseqnum}]) \\
\text{if (base} = \text{nextseqnum)} \\
\text{start_timer} \\
\text{nextseqnum}++ \\
\} \\
\text{else} \\
\text{refuse}_\text{data}(\text{data})
\]

\[ \text{base} = \text{getacknum}(\text{rcvpkt})+1 \]

\[ \text{If (base} = \text{nextseqnum)} \\
\text{stop}_\text{timer} \\
\text{else} \\
\text{start}_\text{timer} \]
GBN: Receiver extended FSM

ACK-only: always send ACK for correctly-received pkt with highest \textit{in-order} seq #
- May generate duplicate ACKs
- Need only remember $\text{expectedseqnum}$

- Out-of-order pkt:
  - Discard (don’t buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #
GBN in action

sender

send pkt0
send pkt1
send pkt2
send pkt3 (wait)
recv ACK0
send pkt4
recv ACK1
send pkt5

pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5

receiver

recv pkt0
send ACK0
recv pkt1
send ACK1
recv pkt3, discard
send ACK1

recv pkt4, discard
send ACK1
recv pkt5, discard
send ACK1
recv pkt2, deliver
send ACK2
recv pkt3, deliver
send ACK3
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
- No need to retransmit correctly received, but out-of-order packets
- Sender window
  - N consecutive seq #'s
  - Again limits seq #s of sent, unACKed pkts
Selective repeat: Sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
Selective repeat

**Sender**

Data from above:
- If next available seq # in window, send pkt

Timeout(n):
- Resend pkt n, restart timer

ACK(n) in [sendbase, sendbase+N]:
- Mark pkt n as received
- Cancel timer for n
- 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

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

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

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

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

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

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

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

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

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

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

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

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

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

pkt2 rcvd, pkt2, pkt3, pkt4, pkt5 delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9
Selective repeat: dilemma

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

- Receiver sees no difference in two scenarios!
- Incorrectly passes duplicate data as new in (a)

Q: What relationship between seq # size and window size?