**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

Connectionless demultiplexing

- Create sockets with port
tables:
  - DatagramSocket mySocket1 = new
    DatagramSocket(99111);
  - DatagramSocket mySocket2 = new
    DatagramSocket(99222);
- UDP socket identified by two-
tuple:
  - (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
Connectionless demux (cont)

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)

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>

**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 1 0 0 1 1 0 1 0 1
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

wraparound: 011110111011110111

```
sum: 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 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)
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

<table>
<thead>
<tr>
<th>State 1</th>
<th>State 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>event causing state transition</td>
<td>event causing state transition</td>
</tr>
<tr>
<td>actions taken on state transition</td>
<td>actions taken on state transition</td>
</tr>
<tr>
<td>event actions</td>
<td>event actions</td>
</tr>
</tbody>
</table>

state: when in this "state" next state uniquely determined by next event
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 read data from underlying channel

```
packet = make_pkt(data)
udt_send(packet)
```

```
rdt_send(data)
```

```
extract (packet, data)
deliver_data(data)
```

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. Wait for call from above
5. rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
6. rdt_rcv(rcvpkt) && isACK(rcvpkt)
7. udt_send(sndpkt)
8. rdt_rcv(rcvpkt) && isNAK(rcvpkt)
9. udt_send(NAK)
10. rdt_rcv(rcvpkt) && corrupt(rcvpkt)
11. Wait for ACK or NAK
12. Wait for call from below
13. rdt_send(data)

rdt2.0: error scenario

1. rdt_send(data)
2. snkpkt = make_pkt(data, checksum)
3. udt_send(sndpkt)
4. Wait for call from above
5. rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
6. rdt_rcv(rcvpkt) && isACK(rcvpkt)
7. extract(rcvpkt, data)
8. deliver_data(data)
9. udt_send(ACK)
10. Wait for call from below
11. rdt_send(data)
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, handles garbled ACK/NAKs
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 “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**

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

Wait for ACK 0

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

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

```
Lambda
```

```
udt_send(sndpkt)
```

Wait for call 0 from above

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

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

Wait for 0 from below

```
rdr_rcv(rcvpkt) && notcorrupt(rcvpkt)  
&& has_seq1(rcvpkt)
```
**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

```plaintext
rdt3.0 sender
```

`rdt3.0 sender`

```plaintext
rdt3.0 sender
```

```plaintext
rdt3.0 sender
```
rdt3.0 in action

(a) operation with no loss

(b) lost packet

(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:

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

- Utilization - fraction of time sender busy sending

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

- 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
- Network protocol limits use of physical resources!
**Pipelined protocols**

**Pipelining:** sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts
- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

- Two generic forms of pipelined protocols: **go-Back-N, selective repeat**
Pipelining: increased utilization

\[ U_{\text{sender}} = \frac{3 \times L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.0008 \]

- first packet bit transmitted, \( t = 0 \)
- last bit transmitted, \( t = L / R \)
- first packet bit arrives
- last bit transmitted, \( t = L / R \)
- ACK arrives, send next packet, \( t = RTT + L / R \)
- last bit of 2nd packet arrives, send ACK
- last bit of 3rd packet arrives, send ACK

Increase utilization by a factor of 3!

Go-Back-N

Sender:
- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed
- Timer for each in-flight pkt
- \( \text{Timeout}(n) \): retransmit pkt n and all higher seq # pkts in window
- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may deceive duplicate ACKs (see receiver)

\[ \text{send_base} \quad \text{nextseqnum} \]

- usable, not yet sent
- sent, not yet ack'ed
- already ack'ed
- not usable
- window size \( N \)
GBN: sender extended FSM

```
GBN: sender extended FSM
```

```
rdt_send(data)
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
        start_timer
        nextseqnum++
} else
    refuse_data(data)

Wait
```
Selective Repeat

- Receiver *individually* acknowledges all correctly received pkts
  - Buffers pkts, as needed, for eventual in-order delivery to upper layer
- Sender only resends pkts for which ACK not received
  - Sender timer for each unACKed pkt
- Sender window
  - N consecutive seq #’s
  - Again limits seq #’s of sent, unACKed pkts
Selective repeat: sender, receiver windows

(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
  - 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

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?