Our goals:
- understand principles behind transport layer services
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
- 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 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

Processes can be different applications (HTTP, DNS, etc) running on the same host and they are multiplexed together.

Multiplexing/demultiplexing
- Multiplexing at send host: gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)
- Demultiplexing at rcv host: delivering received segments to correct socket
Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - reliable data service
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best effort" IP
  - services not available:
    - delay guarantees
    - bandwidth guarantees

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 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 1 0 0 1 1 0 0 1 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 1 0 0 1 0 0 1 0 0 0 1 0 0 1
wraparound: 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1
sum: 1 0 1 1 1 0 1 1 1 1 1 1 1 1 0 0
checksum: 0 1 0 0 0 1 0 0 0 1 1 0 0 1 1 1
```

UDP: more

- often used for streaming multimedia apps
- loss tolerant
- rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

Principles of Reliable data transfer

- important in app., transport, link layers

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

Reliable data transfer: getting started

We'll:

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

Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- the question: how to recover from errors:
  - acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
  - negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

Rdt2.0: FSM specification

- event causing state transition
  - next state uniquely determined by next event

Rdt2.0: operation with no errors

- sender sends one packet, then waits for receiver response

Transport Layer

Transport Layer

Transport Layer
**rdt2.0: error scenario**

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

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
  udt_send(ACK)
```

```
rdt_rcv(rcvpkt) && isACK(rcvpkt)
  udt_send(rcvpkt)
  rdt_send(data)
```

```
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
  udt_send(NAK)
```

```
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
  Wait for ACK or 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

**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