Chapter Goals

- Understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Instantiation and implementation in the Internet

Chapter Overview

- Transport layer services
- Multiplexing/demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - reliable 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
- Transport vs. network layer services:
  - network layer: data transfer between end systems
  - transport layer: data transfer between processes
  - Relies on, enhances, network layer services

Transport-layer protocols

Internet transport services:
- Reliable, in-order unicast delivery (TCP)
  - congestion
  - flow control
  - connection setup
- Unreliable (“best-effort”), unordered unicast or multicast delivery: UDP
- Services not available:
  - real-time
  - bandwidth guarantees
  - reliable multicast
Multiplexing / demultiplexing

Recall: segment - unit of data exchanged between transport layer entities
- aka TPDU: transport protocol data unit

Multiplexing: gathering data from multiple app processes, enveloping data with header (later used for demultiplexing)

Multiplexing/demultiplexing:
- based on sender, receiver port numbers, IP addresses
  - source, dest port #s in each segment
- recall: well-known port numbers for specific applications

Demultiplexing: delivering received segments to correct app layer processes

Multiplexing/demultiplexing: examples

TCP/UDP segment format

UDP: User Datagram Protocol [RFC 768]

Why is there a UDP?
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

UDP: User Datagram Protocol

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 recovery!
**UDP: more**

<table>
<thead>
<tr>
<th>Length, in bytes of UDP segment, including header</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
</tr>
</tbody>
</table>

**UDP segment format**

- **Application data (message)**

**UDP segment format**

**UDP feature details**

- **Port number:** Used for (de)multiplexing.
  - Client ports are ephemeral (short-lived).
  - Server ports are “well known”.
- **UDP checksum:**
  - Pseudo-header (to help double-check source/destination address validity)
  - UDP checksum optional, but RFC 1122/23 (host reqts) requires it to be enabled
- **Application message is simply encapsulated and sent to IP => can result in fragmentation.**

**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

**UDP checksum**

**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 ….*

**Some UDP Effects**

- When UDP datagram fragments at the host, each fragment may generate an ARP request: **ARP flooding**
- Datagram **truncation** possible at destination if dest app not prepared to handle that datagram size! (note: TCP)
- Does not have this problem
- UDP sources **ignore** source quench messages: no response to packet losses.

**UDP Servers**

- Most UDP servers are “iterative” => a single server process receives and handles incoming requests on a “well-known” port.
- Can filter client requests based on incoming IP/port addresses or wild card filters
- Port numbers may be reused, but packet is delivered to at most one end-point.
- Queues to hold requests if server busy
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

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
  - 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
- new mechanisms in Rdt2.0 (beyond Rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

Temporal Redundancy Model

- Packets
  - Sequence Numbers
  - CRC or Checksum
- Status Reports
  - ACKs
  - NAKs, SACKs
  - Bitmaps
- Retransmissions
  - Packets
  - FEC information

Reliability Models

- Reliability => requires redundancy to recover from uncertain loss or other failure modes.
- Two types of redundancy:
  - Spatial redundancy: independent backup copies
    - Forward error correction (FEC) codes
    - Problem: requires huge overhead, since the FEC is also part of the packet(s) it cannot recover from erasure of all packets
  - Temporal redundancy: retransmit if packets lost/error
    - Lazy: trades off response time for reliability
    - Design of status reports and retransmission optimization important

Rdt2.0: FSM Specification

sender FSM

receiver FSM

Rdt2.0: In Action (no errors)
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 adds sequence number to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- 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
- 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. #
Rdt2.1: Discussion

Receiver:
• **must check** if received packet is duplicate
  – state indicates whether 0 or 1 is expected pkt seq #
• **Note:** receiver **cannot** know if its last ACK/NAK received OK at sender

Rdt2.2: a NAK-free Protocol

• Same functionality as rdt2.1, using ACKs only. **Why bother?**
• 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**

Rdt3.0: Channels with Errors and Loss

**New assumption:** underlying channel can also lose packets (data or ACKs)
– What's the difference, anyway?
  – 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
  – yuck: drawbacks?

Rdt3.0 Sender

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 in Action**

- **Performance of Rdt3.0**
  - rdt3.0 works, but **performance stinks!**
  - example: 1 Gbps link, 15 ms e2e prop. delay, 1KB packet:
    
    ```
    T_{transmit} \times 10^9 \text{ b/sec} = 8 \text{ microsec}
    
    Utilization = U = \frac{\text{fraction of time sender busy sending}}{\text{total time}} = \frac{8 \text{ microsec}}{30.016 \text{ msec}} = 0.00015
    
    - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
    - **Problem:** 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
  - Also called “sliding window” protocols

**Go-Back-N**

- Sender:
  - k-bit seq # in pkt header
  - “window” of up to N, consecutive unack’ed pkts allowed

**Pipelined protocols**

- Two generic forms of pipelined protocols:
  1. **go-Back-N**
  2. **selective repeat**
Go-Back-N
- ACK\(n\): ACKs all pkts up to, including seq # \(n\) - “cumulative ACK” – may deceive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- \(\text{timeout}(n)\): retransmit pkt \(n\) and all higher seq # pkts in window

GBN: Sender Extended FSM

GBN: Receiver Extended FSM
- Receiver simple:
  - ACK-only: always send ACK for correctly-received pkt with highest \(\text{in-order}\) seq #
    - may generate duplicate ACKs
    - need only remember expectedseqnum

GBN: Receiver Extended FSM
- \(\text{out-of-order pkt}\):
  - discard (don’t buffer) \(\rightarrow\) \(\text{no receiver buffering!}\)
  - ACK pkt with highest in-order seq #

Selective Repeat
- receiver \(\text{individually}\) 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

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

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?