Review

- Client-server
- P2p
- Application layer vs. transport layer
Transport Layer: Part I

- Transport Layer Services
  - connection-oriented vs. connectionless
  - multiplexing and demultiplexing
- UDP: Connectionless Unreliable Service
- TCP: Connection-Oriented Reliable Service
  - connection management: set-up and tear down
  - reliable data transfer protocols (Part II)
  - flow and congestion control (Part II)

Readings: Chapter 3, Lecture Notes
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. Application and Network Layer

- **application layer**: application processes and message exchange
- **network layer**: logical communication between hosts
- **transport layer**: logical communication support for app 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
End to End Issues

• Transport services built on top of (potentially) unreliable network service
  - packets can be corrupted or lost
  - Packets can be delayed or arrive “out of order”

• Do we detect and/or recover errors for apps?
  - Error Control & Reliable Data Transfer

• Do we provide “in-order” delivery of packets?
  - Connection Management & Reliable Data Transfer

• Potentially different capacity at destination, and potentially different network capacity
  - Flow and Congestion Control
Internet Transport Protocols

TCP service:
• connection-oriented: setup required between client, server
• reliable transport between sender and receiver
• flow control: sender won’t overwhelm receiver
• congestion control: throttle sender when network overloaded

UDP service:
• unreliable data transfer between sender and receiver
• does not provide: connection setup, reliability, flow control, congestion control

Both provide logical communication between app processes running on different hosts!
Multiplexing/Demultiplexing

Demultiplexing at rcv host: delivering received segments to correct application process

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

= API ("socket")
= process

| application | P3 | transport | host 1 |
| network     |    | link      |        |
| physical    |    |           |        |

| P1 | application | P2 |
| transport | | network |
| link | physical |     |

| P4 | application |
| transport | network |
| link | physical |

host 2

host 3
How Demultiplexing Works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate app process (identified by “socket’)

TCP/UDP segment format

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>other header fields</td>
<td></td>
</tr>
<tr>
<td>application data (message)</td>
<td></td>
</tr>
</tbody>
</table>
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 (cont’d)

- 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!

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....
Checksum: Example (from book)

arrange data segment in sequences of 16-bit words

\[ \begin{align*}
0110011001100000 \\
0101010101010101 \\
1000111100001100
\end{align*} \]

\[ \text{sum:} \quad 01001010110000010 \]

checksum (1’s complement):

\[ \begin{align*}
1011010100111101 \\
1011010100111101
\end{align*} \]

verify by adding:

\[ \begin{align*}
1111111111111111 \\
1111111111111111
\end{align*} \]

binary addition, with overflow wrapped around
TCP: Overview

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

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

- **pipelined:**
  - TCP congestion and flow control
    set window size

- **send & receive buffers**

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

- **connection-oriented:**
  - handshaking (exchange of control msgs) init’s sender, receiver state before data exchange

- **flow controlled:**
  - sender will not overwhelm receiver
TCP Segment Structure

**URG**: urgent data (generally not used)

**ACK**: ACK # valid

**PSH**: push data now (generally not used)

**RST, SYN, FIN**: connection estab (setup, teardown commands)

Internet checksum (as in UDP)

- `source port #`
- `dest port #`
- `sequence number`
- `acknowledgement number`
- `rcvr window size`
- `checksum`
- `ptr urgent data`

**Options** (variable length)

- `application data`
  - (variable length)

# bytes rcvr willing to accept

Counting by bytes of data (not segments!)
TCP Seq. #'s and ACKs

**Seq. #'s:**
- byte stream
- "number" of first byte in segment's data

**ACKs:**
- seq # of next byte expected from other side

---

Host A

User types 'C'

Host B

seq # of next byte expected from other side

host ACKs receipt of echoed 'C'

host ACKs receipt of 'C', echoes back 'C'

red: A-to-B    green: B-to-A

simple telnet scenario
TCP: Error Scenarios

Questions for you:

• How to detect lost packets?
• How to “recover” lost packets?
• Potential consequence of retransmission?
• How to detect duplicate packets?
• “State” maintained at sender & receiver?
TCP: Error Scenarios (cont’d)

Host A

Seq=92, 8 bytes data

ACK=100

lost ACK scenario

time

time

timeout

Host B

X

loss

Seq=92, 8 bytes data

ACK=100

Host A

Seq=92, 8 bytes data

ACK=100

duplicate packets

timeout

Host B

ACK=100

X

Host A

Seq=92, 8 bytes data

ACK=100

A Simple Reliable Data Transfer Protocol

“Stop & Wait” Protocol (aka “Alternate Bit” Protocol)

Sender algorithm:

• **Send Phase**: send data segment (n bytes) w/ seq=x, buffer data segment, set timer

• **Wait Phase**: wait for ack from receiver w/ ack= x+n
  - if received ack w/ ack=x+n, set x:=x+n, and go to sending phase with next data segment
  - if time out, resend data segment w/ seq=x.
  - if received ack w/ ack != x+n, ignore (or resend data segment w/ seq=x)

Receiver algorithm: ??
SRDTP: Finite State Machine

Sender FSM

\[ \text{: state} \quad \text{event} \quad \text{action} \quad : \text{transition} \quad \text{Receiver FSM?} \]

*Upper layer:*
- send data (n bytes)
- make data sgt, seq = x, set timer
- pass data sgt to lower layer

？

Send phase

Wait phase

receive Ack w/ ack = x+n

\[ x := x+n, \text{stop timer} \]

info ("state") maintained at sender:
- phase it is in (send, or wait), ack expected, data sgt sent (seq ≠), timer

receive Ack w/ ack ≠ x+n

no op, or resend data sgt

time out

resend data sgt
TCP Connection Set Up

TCP sender, receiver establish “connection” before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info

- **client**: end host that initiates connection

- **server**: end host contacted by client

**Three way handshake:**

**Step 1:** client sends TCP SYN control segment to server
- specifies initial seq #

**Step 2:** server receives SYN, replies with SYN+ACK control segment
- ACKs received SYN
- specifies server → receiver initial seq. #

**Step 3:** client receives SYN+ACK, replies with ACK segment (which may contain 1st data segment)
Question:

a. What kind of “state” client and server need to maintain?

b. What initial sequence # should client (and server) use?
3-Way Handshake: Finite State Machine

Client FSM?

Server FSM?

**Upper layer:** initiate connection

sent SYN w/ initial seq = x

info ("state") maintained at client?
Connection Setup Error Scenarios

• Lost (control) packets
  - What happen if SYN lost? client vs. server actions
  - What happen if SYN+ACK lost? client vs. server actions
  - What happen if ACK lost? client vs. server actions

• Duplicate (control) packets
  - What does server do if duplicate SYN received?
  - What does client do if duplicate SYN+ACK received?
  - What does server do if duplicate ACK received?
Connection Setup Error Scenarios (cont’d)

• Importance of (unique) initial seq. no.?
  - When receiving SYN, how does server know it’s a new connection request?
  - When receiving SYN+ACK, how does client know it’s a legitimate, i.e., a response to its SYN request?

• Dealing with old duplicate (aka “ghost”) packets from old connections (or from malicious users)
  - If not careful: “TCP Hijacking”

• How to choose unique initial seq. no.?
  - randomly choose a number (and add to last syn# used)

• Other security concern:
  - “SYN Flood” -- denial-of-service attack
TCP: Closing Connection

Remember TCP duplex connection!

Client wants to close connection:

**Step 1:** client end system sends TCP FIN control segment to server

**Step 2:** server receives FIN, replies with ACK. half closed

**Step 3:** client receives ACK.

half closed, wait for server to close

Server finishes sending data, also ready to close:

**Step 4:** server sends FIN.
TCP: Closing Connection (cont’d)

**Step 5:** client receives FIN, replies with ACK. connection fully closed

**Step 6:** server, receives ACK. connection fully closed

Well Done!

Problem Solved?
Two-Army Problem
TCP: Closing Connection (revised)

Two Army Problem!

**Step 5:** client receives FIN, replies with ACK.
- Enters "timed wait" - will respond with ACK to received FINs

**Step 6:** server, receives ACK. Connection fully closed

**Step 7:** client, timer expires, connection fully closed
TCP Connection Management FSM

TCP client lifecycle

- CLOSED
  - client application initiates a TCP connection
  - send SYN

- TIME_WAIT
  - wait 30 seconds
  - receive FIN, send ACK

- SYN_SENT
  - receive SYN & ACK, send ACK

- ESTABLISHED
  - client application initiates close connection
  - send FIN

- FIN_WAIT_1
  - receive ACK, send nothing

- FIN_WAIT_2
  - receive FIN, send ACK

TCP client lifecycle
TCP Connection Management FSM

TCP server lifecycle

- **CLOSED**: Server application creates a listen socket.
- **LISTEN**: Receive SYN, send SYN & ACK.
- **SYN_RCVD**: Receive ACK, send nothing.
- **ESTABLISHED**: Receive FIN, send ACK.
- **CLOSE_WAIT**: Send FIN.
- **LAST_ACK**: Receive ACK, send nothing.

TCP server lifecycle.
Socket: Conceptual View

socket()

sendto()
bind()
recvfrom()

USER APPL.

SOCKET LAYER

buffered data yet to be sent
buffered data yet to be read
socket parameters

Socket
Descriptor

Port
Number

TRANSPORT LAYER

Operating Systems

User Space
BSD Socket Programming (connectionless)

**Server**
- connectionless protocol
  - `socket()`
  - `bind()`
  - `recvfrom()`

**Client**
- `socket()`
- `bind()`
- `sendto()`
- `recvfrom()`

`recvfrom()` blocks until data received from a client.
BSD Socket Programming Flows (connection-oriented)

Server
- socket()
- bind()
- listen()
- accept()

blocks until connection from client

connection establishment

Client
- socket()
- bind()
- connect()
- write()

- read()

process request

write()

- data (request)

- data (reply)

- read()
Transport Layer: Part I Summary

• Transport Layer Services
  - Issues to address
  - Multiplexing and Demultiplexing
  - Connectionless vs. Connection-Oriented

• UDP: Unreliable, Connectionless

• TCP: Reliable, Connection-Oriented
  - Packet (“Segment”) Format: Sequence #, ACK, flags, ...
  - A “Simple Reliable Data Transfer Protocol”
  - Connection Management: 3-way handshake, closing connection

• Preview of Part II:
  - more efficient reliable data transfer protocols
  - round-trip time estimation and flow/congestion control