Chapter 3: 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
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 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

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
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 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

host 1  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 socket

---

Connectionless demultiplexing

- Create sockets with port numbers:
  ```java
  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)

```java
DatagramSocket serverSocket = new DatagramSocket(6428);
```

```
+---------+     +---------+     +---------+
| P2      | --- > | P3      | --- > | P1      |
|         |       |         |       |         |
| SP: 9157|       | SP: 6428|       | SP: 5775|
| DP: 6428|       | SP: 6428|       | DP: 6428|
| client  | --- > | server  | --- > | Client  |
| IP: A   |       | IP: C   |       | IP: B   |
|         |       |         |       |         |
```

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
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 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
  - DNS
  - SNMP
- Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recovery!

UDP segment format

**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 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
  sum      1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
  checksum 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```