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

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
**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

---

**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)

```
<table>
<thead>
<tr>
<th>Host 1</th>
<th>Host 2</th>
<th>Host 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>application</td>
<td>transport</td>
<td>network</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

= socket

= process
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

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>

Connectionless demultiplexing

- Create sockets with port numbers:
  - 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(123);

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 application 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)

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**

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

<table>
<thead>
<tr>
<th>checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application data (message)</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...