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

**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

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

Transport Layer 3-5
Transport Layer 3-6
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)

Transport Layer 3-7

Transport Layer 3-8
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
- host uses IP addresses & port numbers to direct segment to appropriate socket

TCP/UDP segment format

<table>
<thead>
<tr>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
</tr>
<tr>
<td>other header fields</td>
</tr>
<tr>
<td>application data (message)</td>
</tr>
</tbody>
</table>

Connectionless demultiplexing

- Create sockets with port numbers:
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(12534);
  DatagramSocket mySocket2 = new DatagramSocket(12535);
  ```
- UDP socket identified by two-tuple:
  ```java
  (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)

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!

<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

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
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)

(a) provided service

(b) service implementation
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

- rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
- deliver_data(): called by rdt to deliver data to upper
- udt_send(): called by rdt, to transfer packet over unreliable channel to receiver
- rdt_rcv(): called when packet arrives on rcv-side of channel
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 read data from underlying channel

```
Wait for call from above
rdt_send(data)
packet = make_pkt(data)
udt_send(packet)
```

```
Wait for call from below
rdt_rcv(packet)
extract (packet, data)
deliver_data(data)
```

sender          receiver

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**

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

 Wait for call from above

rdt_rcv(rcvpkt) &&
 isACK(rcvpkt)
udt_send(sndpkt)

 Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
 isNAK(rcvpkt)
udt_send(sndpkt)

λ
```

**sender**

```
rcv(rdcv)
udt_send(rcvpkt)

 Wait for call from below

rdt_rcv(rcvpkt) &&
 notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```

**receiver**

**rdt2.0: operation with no errors**

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

 Wait for call from above

rdt_rcv(rcvpkt) &&
 isACK(rcvpkt)
udt_send(sndpkt)

 Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
 isNAK(rcvpkt)
udt_send(sndpkt)

λ
```

**sender**

```
rcv(rdcv)
udt_send(rcvpkt)

 Wait for call from below

rdt_rcv(rcvpkt) &&
 notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```

**receiver**
rdt2.0: error scenario

- rdt_send(data)
  - snkpkt = make_pkt(data, checksum)
  - udt_send(sndpkt)
- Wait for call from above
- rdt.rcv(rcvpkt) && isNAK(rcvpkt)
  - udt_send(sndpkt)
- rdt.rcv(rcvpkt) && isACK(rcvpkt)
- Wait for ACK or NAK
  - udt_send(sndpkt)
  - Wait for call from below
- rdt.rcv(rcvpkt) && isNAK(rcvpkt)
  - udt_send(sndpkt)
  - Wait for call from below
- rdt.rcv(rcvpkt) && notcorrupt(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - udt_send(ACK)
- rdt.rcv(rcvpkt) && notcorrupt(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - udt_send(ACK)
- rdt.rcv(rcvpkt) && notcorrupt(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - udt_send(ACK)