Transport Layer

Class 15
Basics and UDP
Transport Layer

- Transport Layer Services
- Multiplexing/demultiplexing
- Reliable data transfer
- Flow control
- Congestion control
Network layer

- transport segment from sending to receiving host
- on sending side, encapsulates segments into datagrams
- on receiving side, delivers segments to transport layer
- network layer protocols in every host, router
- Router examines header fields in all IP datagrams passing through it
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
Transport services and protocols

- **send side**: breaks app messages into **segments**, passes to network layer
- transport protocols run in **end systems**
- transport vs network layer services:
  - **network layer**: data transfer between **end systems**
  - **transmit side**: reassembles segments into **messages**, passes to app layer
  - **receive side**: reassembles segments into **messages**, passes to app layer
Transport vs. Network Layer

- *network layer*: logical communication between *hosts*

- *transport layer*: logical communication between *processes*
TCP/IP Transport Layer Protocols

- unreliable, unordered unicast or multicast delivery: UDP

- reliable, in-order unicast delivery: TCP

- services not available at the network layer:
  - Performance guarantees

  - Non-unicast delivery models
Multiplexing and De-multiplexing
Multiplexing/Demultiplexing

- Use same communication channel between hosts for several logical communication processes
- How does Mux/DeMux work?
  - Sockets: doors between process & host
Client/server socket interaction: UDP

Server (running on hostid)

create socket, port=x, for incoming request:
serverSocket = DatagramSocket()

read request from serverSocket

write reply to serverSocket specifying client host address, port number

to Client

Client

create socket, clientSocket = DatagramSocket()

Create, address (hostid, port=x, send datagram request using clientSocket

read reply from clientSocket

close clientSocket
Multiplexing/Demultiplexing

• Each end-system has a single protocol “stack”

• Multiplexing is the process of allowing multiple applications to use the network simultaneously

• Demultiplexing is the process of delivering received data to the appropriate application
Layering Terms

- **At the sender, each layer takes data from above**
  - May subdivide into multiple data units at sending layer
  - Adds header information to create new data unit
  - Passes new data unit to layer below

- **The process is reversed at the receiver**
Connectionless Multiplexing

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

Multiplexing at send host:
- gather data from multiple Sockets
- envelop data with header (for later demultiplexing)

Diagram:
- Application-layer data
- Segment header
- Segments
- P1
- P3
- Receiver
Connectionless Demux

• 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 Demultiplexing

Demultiplexing at rcv host:

- deliver received segments to correct socket (app layer processes)
How demultiplexing works

- host receives IP datagrams
  - each datagram
    - has source IP address, destination IP address
    - 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

<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>

TCP/UDP segment format
Multiplexing/demultiplexing: examples

Port use: simple telnet app

Port use: Web server

SMU CSE 5344/7344
DatagramSocket serverSocket = new DatagramSocket(6428);
DatagramSocket myclientSocket = new DatagramSocket();
How demultiplexing works - summary

- Demultiplexing is the process of delivering received segments to the correct application-layer process
  - IP address (in network-layer datagram header) identifies the receiving machine
  - Port number (in transport-layer segment header) identifies the receiving process
Example UDP client

- **Input:** receives packet (TCP received "byte stream")
- **Output:** sends packet (TCP sent "byte stream")

Client process

- Keyboard input stream
- Monitor output

Client UDP socket

- To Transport Layer
- From Transport Layer
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)
Connectionless Transport
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
## Transport service requirements of common apps

<table>
<thead>
<tr>
<th>Application</th>
<th>Data loss</th>
<th>Bandwidth</th>
<th>Time Sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>file transfer</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>e-mail</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>Web documents</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>real-time audio/video</td>
<td>loss-tolerant</td>
<td>audio: 5kbps-1Mbps, video: 10kbps-5Mbps</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td>stored audio/video</td>
<td>loss-tolerant</td>
<td>same as above</td>
<td>yes, few secs</td>
</tr>
<tr>
<td>interactive games</td>
<td>loss-tolerant</td>
<td>few kbps up</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td>instant messaging</td>
<td>no loss</td>
<td>elastic</td>
<td>yes and no</td>
</tr>
</tbody>
</table>
## Internet apps: application, transport protocols

<table>
<thead>
<tr>
<th>Application</th>
<th>Application layer protocol</th>
<th>Underlying transport protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-mail</td>
<td>SMTP [RFC 2821]</td>
<td>TCP</td>
</tr>
<tr>
<td>remote terminal access</td>
<td>Telnet [RFC 854]</td>
<td>TCP</td>
</tr>
<tr>
<td>Web</td>
<td>HTTP [RFC 2616]</td>
<td>TCP</td>
</tr>
<tr>
<td>file transfer</td>
<td>FTP [RFC 959]</td>
<td>TCP</td>
</tr>
<tr>
<td>streaming multimedia</td>
<td>proprietary (e.g. RealNetworks)</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>proprietary (e.g., Dialpad)</td>
<td>typically UDP</td>
</tr>
</tbody>
</table>
UDP Segment Structure

- 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)

Checksum includes Header, Data, & Pseudoheader

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
- Example:
  
  Sum of segment = 1010101110011011
  Checksum      = 010101001100100

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest. port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

application data (message payload)

UDP segment format
UDP checksum

**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment

**Receiver:**
- compute checksum of received segment
- check if computed checksum equals checksum field value

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest. port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

application data
(message payload)

UDP segment format
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

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]
Network Considerations of UDP
UA-1 has transaction with UA-A, UA-2 with UA-B, etc.

Stateless proxies P1 and P2 use UDP

App1 and App2 talk TCP over the same physical link

P1<->P2 monopolizes the link, getting 90% of bandwidth

If the link becomes congested, P1 and P2 don't know and don't react. UAs react only within a given transaction.

App1<->App2 starves.
Issue with UDP

- Lacks Congestion Control

- UDP’s lack of complete reliability is detrimental

- Cyclic UDP
Cyclic UDP (CUDP)

- Cyclic UDP

- CUDP enables some reliable transmission by using an automatic repeat request (ARQ) based prioritized sliding window mechanism
Methods for Reliability

• ARQ & FEC
• What are the fundamental differences between ARQ schemes and FEC
  - Feedback vs. no feedback
  - Recovery latency
  - Effective bandwidth
• Packet recovery vs. Loss amelioration
Types of ARQ

- Hard ARQ = TCP

- Soft ARQ
Dimensions of Soft-ARQ

• Feedback
  - What form, how often
• What to resend
  - Don’t always have to resend the original
• When to give up
  - Difference between soft and hard ARQ
• Interaction with rate control
  - Accounting for retransmission bandwidth
Can we get it there in time?

• The ultimate question
  - If we can’t, don’t bother trying
  - If we can, is it worth the bandwidth?

• How much time is there?
  - What does it depend on?
Source vs Receiver Driven

• Either the source or the receiver can be in charge of driving the ARQ scheme

• If source based:
  - Feedback must include measure of recv. buffer
  - Source keeps estimate of one-way delay

• If receiver based:
  - Receiver maintains estimate of RTT
  - Sends ARQ requests only if it expects recovery is possible
What to Resend

• Simple system: resend missing packet
• Why might that not be what we want to do?
  - Tradeoff quality for bandwidth
When to Resend

- **IMMEDIATELY**

- **Implications?**
  - Feedback processing has to be done at a very high priority
When to Give Up

• Usually: after one retransmit

• Not always the case, though
  - Large buffers in non-interactive applications
  - Media specific oddities
Interaction with Rate Control

- Accounting for retransmission bandwidth
- Don’t account for it
  - Rate control only applied to original data stream
  - OK if loss caused by transient congestion, but thrashes if congestion is persistent
- Account strictly
  - Utility tradeoff between new stuff at full rate and value of retransmitted data
Cyclic-UDP

- Application constructs a relatively large list of packets to send in priority order
- Cyclic-UDP sends packets as fast as possible with ACK-based feedback
- If loss detected
  - retransmit earlier packets before sending new packets
- Sending rate adaptively estimated
  - Not AIMD, but rate control is orthogonal issue
Cyclic-UDP Illustrated

Fill

Send
Cyclic-UDP Illustrated

Fill  Send

Feedback

Data
Cyclic-UDP Illustrated
Cyclic-UDP Illustrated

![Diagram showing Fill and Send processes with Feedback and Data flows]
Cyclic-UDP Illustrated

Fill  Send

Feedback  Data
Cyclic-UDP Illustrated

Fill

Send

18
17
16
15
14
13
12
11
10

Data

Feedback
Cyclic-UDP Features

• Advantages

• Disadvantages
End of Class 15