The Transport Layer De/Multiplexing, Reliability

CS 352, Lecture 6
http://www.cs.rutgers.edu/~sn624/352-S19

Srinivas Narayana
(slides heavily adapted from text authors’ material)
This lecture: Transport
Transport services and protocols

• Provide **logical communication** between app processes running on different hosts

• Transport protocols run @ hosts
  • send side: breaks app messages into **segments**, passes to network layer
  • recv 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
  • relies on and enhances, network layer services

• **Transport layer**: logical communication between processes
  • processes = kids
  • app messages = letters in envelopes
  • hosts = houses
  • transport protocol = Alice and Bob who de/mux to in-house siblings
  • network-layer protocol = postal service

**Household analogy:**
12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Alice and Bob who de/mux to in-house siblings
- 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
Layering: in terms of packets
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’s uses

- Often used for streaming multimedia apps
  - loss tolerant
  - Delay sensitive
- Other UDP uses: need “lightweight”
  - DNS
  - SNMP
- If you want reliable transfer over UDP, you must add reliability at application layer
  - Can implement 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>

Application data (message)

Length, in bytes of UDP segment, including header

32 bits
How demultiplexing works

- Host receives IP datagrams
  - Datagram contains a transport-level segment
  - each segment has source IP address, destination IP address
  - each segment has source, destination port number

- 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
Connectionless demultiplexing

- Create sockets with host-local port numbers to receive data

  // Example: Java UDP socket
  DatagramSocket socket1 = new DatagramSocket(12534);

- When creating data to send into UDP socket, you must specify
  (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
UDP client + server (Python API)

UDPsender():
    try:
        ssd=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    except socket.error as err:
        exit()

    # Define the port on which you want to send to the receiver
    RPort = 50007
    hisip=socket.gethostbyname("ilab.cs.rutgers.edu")
    receiver_binding=(hisip, RPort)
    MESSAGE="hello world"
    msg=MESSAGE.encode('utf-8')
    ssd.sendto(msg, receiver_binding)
    # no "connection" to other side needed before sending data!

    # Close the sender socket
    ssd.close()
    exit()

UDPreceiver():
    try:
        rsd=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    except socket.error as err:
        exit()

    # Define the port on which you want to receive from the server
    Rport = 50007
    myip = socket.gethostbyname(socket.gethostname())
    # connect to the server on local machine
    server_binding=(myip, Rport)
    rsd.bind(server_binding)
    data, addr = rsd.recvfrom(1024)
    # no need to "accept" a connection from other side before receiving data!
    print(data.decode("utf-8"))

    # Close the receiver socket
    rsd.close()
    exit()
UDP Checksum

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

Solution principle: compute a function over data, store it along with data.

**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.
UDP 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

\[
\begin{array}{c}
1110011001100110 \\
1101010101010101 \\
\hline
110111011101110111 \\
\hline
\text{wraparound}
\end{array}
\]

\[
\begin{array}{c}
10111011101111100 \\
01000100010000111 \\
\hline
\text{checksum}
\end{array}
\]
Internet Checksum Example

- Complement of the sum is stored in the checksum field
- At the receiver, all the byte fields are added along with the checksum
- Sum + checksum must be all 1s
  - All 1s, No error else discard packet
- UDP checksum is optional in IPv4
- UPD checksum is mandatory in IPv6
UDP summary

• A thin shim around best-effort IP
• Provides basic multiplexing/demultiplexing for applications
• Basic error detection (bit flips) using checksums
Reliable data transfer
Reliable Data Transfer

• Problem: Reliability
  • Applications want an abstraction of a reliable link even though packets can be corrupted or get lost.

• Where can packets be corrupted or lost?
  • In the network
  • At the receiver

• Solution: keep track of packets reaching other side
Reliability support

• Sender needs to know if a packet was corrupted or lost
• How?
  • Acknowledgements (ACKs)
  • Positive ACKs and negative ACKs (NAKs)
• Sender needs to retransmit on receiving a negative ACK
• But what if packets are lost?
  • Timeouts
  • Remember, ACKs can also get lost!
Reliable delivery algorithms for transport

• Consider a series of increasingly complex (and realistic) networks

• “Stop and wait” protocols
  • An ideal network without bit errors or packet loss
  • Channels with bit errors
  • Channels with packet losses

• Pipelined data transfer (“sliding window protocols”)
  • Go Back N
  • Selective Repeat
Transport in an ideal network

Assumptions:
- Error free transmission link,
- Infinite buffer at the receiver

No acknowledgement of frames necessary
Since the data link is error-free and the receiver can buffer as many frames as it likes, no frame will ever be lost
Stop-and-wait: normal operation

Packet Length = L; Bandwidth = R; RTT = 2*Prop Delay

- First packet bit transmitted, $t = 0$
- Last packet bit transmitted, $t = L / R$
- First packet bit arrives
- Last packet bit arrives, send ACK
- ACK arrives, send next packet, $t = RTT + L / R$
Stop-and-wait: packet corrupted

Packet Length = L; Bandwidth = R; RTT = 2 * Prop Delay

sender

first packet bit transmitted, $t = 0$

last packet bit transmitted, $t = L / R$

receiver

RTT

first packet bit arrives

last packet bit arrives, send ACK

ACK arrives, send next packet, $t = RTT + L / R$

NACK
Stop-and-wait: packet lost

sender

Timeout

receiver

- first packet bit arrives
- last packet bit arrives, send ACK

Stop and wait: packet lost
Stop-and-wait: ACK lost!
Stop-and-wait: ACKs may be delayed!
Stop-and-wait: Detecting duplicates

Timeout too short
Duplicate Transmission
Performance of stop and wait

- example: 1 Gbps link, 1.5 ms end to end prop. delay, 1 KB packet:

\[ T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{9} \text{ b/sec}} = 8 \text{ microsec} \]

\[ U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{0.008}{3.008} = 0.0027 \]

- \( U_{\text{sender}} \): utilization – fraction of time sender busy sending
- 1KB pkt every 3 msec -> 330kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!
Bandwidth-delay product

• Continuously send data until first ACK
• How much? BW*RTT
• Known as Bandwidth delay product
• Number of packets $N = \frac{BW\times RTT}{\text{Packet size}}$
Pipelined protocols

**Pipelining:** sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts
Pipelining Example: increased utilization

\[ U_{\text{sender}} = \frac{3 \times L / R}{RTT + L / R} = \frac{0.024}{3.008} = 0.008 \]

Increase utilization by a factor of 3!
Reliable transmission & Flow Control

• What to do when there is a packet loss?
  • On the link (in the network)
  • At the receiver (buffer overflow)

• Need to recoup losses

• What happens if the packet is lost in the network?
  • A random event, retransmit

• What happens if the sender tries to transmit faster than the receiver can accept?
  • Data will be lost unless flow control is implemented
Flow control in an ideal network  (cont’d)