CS457 – Networking and the Internet
Fall 2016

Topics
• Principles underlying transport-layer services
  – Demultiplexing
  – Detecting corruption
  – Reliable delivery
  – Flow control
• Transport-layer protocols
  – User Datagram Protocol (UDP)
  – Transmission Control Protocol (TCP)

Problem
• How to turn a host-to-host packet delivery service into a process-to-process communication channel
Role of Transport Layer

- **Application layer**
  - Communication between networked applications
  - Protocols: HTTP, FTP, NNTP, and many others

- **Transport layer**
  - Communication between processes (e.g., socket)
  - Relies on network layer and serves the application layer
  - Protocols: TCP and UDP

- **Network layer**
  - Communication between nodes
  - Protocols: IP

End-to-end / Transport Protocols

- Typical limitations of the network on which transport protocol will operate
  - Drop messages
  - Reorder messages
  - Deliver duplicate copies of a given message
  - Limit messages to some finite size
  - Deliver messages after an arbitrarily long delay

- Common properties that a transport protocol can be expected to provide
  - Guarantees message delivery
  - Delivers messages in the same order they were sent
  - Delivers at most one copy of each message
  - Supports arbitrarily large messages
  - Supports synchronization between the sender and the receiver
  - Allows the receiver to apply flow control to the sender
  - Supports multiple application processes on each host

Transport Protocols

- Challenge for Transport Protocols
  - Develop algorithms that turn the less-than-desirable properties of the underlying network into the high level of service required by application programs
Transport Protocols

• Provide **logical communication** between application processes running on different hosts
• Run on end hosts
  – Sender: breaks application messages into **segments**, and passes to network layer
  – Receiver: reassembles segments into messages, passes to application layer
• Multiple transport protocol available to applications
  – Internet: TCP and UDP

Internet Transport Protocols

• Datagram **messaging** service (UDP)
  – No-frills extension of “best-effort” IP
  – Just send the data – **each send is a message**
• Reliable, **streaming**, in-order delivery (TCP)
  – Connection set-up
  – Discarding of corrupted packets
  – Retransmission of lost packets
  – Flow control
  – Congestion control (next lecture)
• Services **not** available from UDP or TCP
  – Delay guarantees
  – Bandwidth guarantees

Multiplexing and Demultiplexing

• Host receives IP datagrams
  – Each datagram has source and destination IP address,
  – Each datagram carries one transport-layer segment
  – Each segment has source and destination port number
• Host uses IP addresses and port numbers to direct the segment to appropriate socket
UDP

Simple Demultiplexer (UDP)

- Extends host-to-host delivery service of the underlying network into a process-to-process communication service
- Adds a level of demultiplexing which allows multiple application processes on each host to share the network

User Datagram Protocol (UDP)

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Send messages to and receive them from a socket
- Lightweight delivery service
  - IP plus port numbers to support (de)multiplexing
  - Optional error checking on the packet contents

<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
<tr>
<td>DATA</td>
<td></td>
</tr>
</tbody>
</table>
Why Would Anyone Use UDP?

- Finer control over what data is sent and when
  - As soon as an application process writes into the socket
  - ... UDP will package the data and send the packet
- Low delay
  - UDP just blasts away without any formal preliminaries
  - ... which avoids introducing delays such as setup
- No connection state
  - No allocation of buffers, parameters, sequence #s, etc.
  - ... making it easier to handle many active clients
- Small packet header overhead
  - UDP header is only eight-bytes long

Popular Applications That Use UDP

- Multimedia streaming
  - Retransmitting lost/corrupted packets is not worthwhile
  - By the time the packet is retransmitted, it’s too late
  - E.g., telephone calls, video conferencing, gaming
- Simple query protocols like Domain Name System
  - Overhead of connection establishment is overkill
  - Easier to have application retransmit if needed

"Address for www.cnn.com?"
"12.3.4.15"
Transmission Control Protocol (TCP)

- Connection oriented
  - Explicit set-up and tear-down of TCP session
- Stream-of-bytes service
  - Sends and receives a stream of bytes, not messages
  - Similar to file I/O
- Reliable, in-order delivery
  - Checksums to detect corrupted data
  - Acknowledgments & retransmissions for reliable delivery
  - Sequence numbers to detect losses and reorder data
- Flow control
  - Prevent overflow of the receiver’s buffer space
- Congestion control
  - Adapt to network congestion for the greater good

TCP Segments

TCP “Stream of Bytes” Service
...Emulated Using TCP “Segments”

TCP Segment

- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes on an Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header is typically 20 bytes long
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream

Sequence Numbers
Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - Why not a de facto ISN of 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … and there is a chance an old packet is still in flight
  - … and might be associated with the new connection
- Security issue
  - An adversary can guess ISNs and hijack a connection
- So, TCP requires changing the ISN over time
  - Set from a 32-bit clock that ticks every 4 microseconds
  - … which only wraps around once every 4.55 hours!
- But, this means the hosts need to exchange ISNs

TCP Three-Way Handshake

Establishing a TCP Connection

- Three-way handshake to establish connection
  - Host A sends a SYN (open) to the host B
  - Host B returns a SYN acknowledgment (SYN ACK)
  - Host A sends an ACK to acknowledge the SYN ACK
  - Each host tells its ISN to the other host.
TCP Header

<table>
<thead>
<tr>
<th>Flags: SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Destination port</td>
<td>Sequence number</td>
<td>Acknowledgment</td>
<td>Flags</td>
<td>Advertised window</td>
</tr>
<tr>
<td>Flags</td>
<td>Checksum</td>
<td>Urgent pointer</td>
<td>Options (variable)</td>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

Step 1: A’s Initial SYN Packet

A tells B it wants to open a connection...

Step 2: B’s SYN-ACK Packet

B tells A it accepts, and is ready to hear the next byte...

... upon receiving this packet, A can start sending data
Step 3: A’s ACK of the SYN-ACK

<table>
<thead>
<tr>
<th>Flags</th>
<th>A’s port</th>
<th>B’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSH</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>URG</td>
<td>checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>ACK</td>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

A tells B it’s okay to start sending

... upon receiving this packet, B can start sending data

What if the SYN Packet Gets Lost?

• Suppose the SYN packet gets lost
  – Packet is lost inside the network, or
  – Server rejects the packet (e.g., listen queue is full)
• Eventually, no SYN-ACK arrives
  – Sender sets a timer and waits for the SYN-ACK
  – … and retransmits the SYN if needed
• How should the TCP sender set the timer?
  – Sender has no idea how far away the receiver is
  – Hard to guess a reasonable length of time to wait
  – Some TCPs use a very conservative default of 3 or 6 seconds

SYN Loss and Web Downloads

• User clicks on a hypertext link
  – Browser creates a socket and does a “connect”
  – The “connect” triggers the OS to transmit a SYN
• If the SYN is lost…
  – The 3-6 seconds of delay may be very long
  – The user may get impatient
  – … and click the hyperlink again, or click “reload”
• User triggers an “abort” of the “connect”
  – Browser creates a new socket and does a “connect”
  – Essentially, forces a faster send of a new SYN packet!
  – Sometimes very effective, and the page comes fast
TCP Retransmissions

Automatic Repeat reQuest (ARQ)

• Automatic Repeat Request
  – Receiver sends acknowledgment (ACK) when it receives packet
  – Sender waits for ACK and timeouts if it does not arrive within some time period
• Simplest ARQ protocol
  – Stop and wait
  – Send a packet, stop and wait until ACK arrives

Reasons for Retransmission

- Packet lost
- ACK lost
- Early timeout
- DUPPLICATE PACKET
- DUPPLICATE PACKETS
How Long Should Sender Wait?

- Sender sets a timeout to wait for an ACK
  - Too short: wasted retransmissions
  - Too long: excessive delays when packet lost
- TCP sets timeout as a function of the RTT
  - Expect ACK to arrive after an RTT
  - … plus a fudge factor to account for queuing
- But, how does the sender know the RTT?
  - Can estimate the RTT by watching the ACKs
  - Smooth estimate: keep a running average of the RTT
    - \( \text{EstimatedRTT} = a \times \text{EstimatedRTT} + (1 - a) \times \text{SampleRTT} \)
  - Compute timeout: \( \text{TimeOut} = 2 \times \text{EstimatedRTT} \)

Example RTT Estimation

A Flaw in This Approach

- An ACK doesn’t really acknowledge a transmission
  - Rather, it acknowledges receipt of the data
- Consider a retransmission of a lost packet
  - If you assume the ACK goes with the 1st transmission
  - … the SampleRTT comes out way too large
- Consider a duplicate packet
  - If you assume the ACK goes with the 2nd transmission
  - … the Sample RTT comes out way too small
Retransmission Ambiguity

Simple solution in the Karn/Partridge algorithm:
Only collect samples for segments sent one single time

Yet Another Limitation…

• Doesn’t consider variance in the RTT
  – If variance is small, the EstimatedRTT is pretty accurate
  – … but, if variance is large, the estimate isn’t all that good
• Key observation:
  – Using 2\*RTT for timeout doesn’t work
  – At high loads round trip variance is high
• Solution:
  – If D denotes mean variance
  – Timeout = RTT + 4D
• Jacobson/Karels algorithm
  – See Section 5.2 of the Peterson/Davie book for details

TCP Sliding Window
Motivation for Sliding Window

- Stop-and-wait is inefficient
  - Only one TCP segment is “in flight” at a time
  - Especially bad when delay-bandwidth product is high
- Numerical example
  - 1.5 Mbps link with a 45 msec round-trip time (RTT)
  - Delay-bandwidth product is 67.5 Kbits (or 8 KBytes)
  - But, sender can send at most one packet per RTT
  - Assuming a segment size of 8 Kbits/segment
  - That’s just one-eighth of the 1.5 Mbps link capacity!

Sliding Window

- Allow a larger amount of data “in flight”
  - Allow sender to get ahead of the receiver
  - … though not too far ahead

Receiver Buffering

- Window size
  - Amount that can be sent without acknowledgment
  - Receiver needs to be able to store this amount of data
- Receiver advertises the window to the sender
  - Tells the sender the amount of free space left
  - … and the sender agrees not to exceed this amount
TCP Header for Receiver Buffering

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>HdrLen</td>
<td>0</td>
<td>Flags</td>
<td>Advertised window</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fast Retransmission

Timeout is Inefficient

- Timeout-based retransmission
  - Sender transmits a packet and waits until timer expires
  - ... and then retransmits from the lost packet onward
Fast Retransmission

• Better solution possible under sliding window
  – Although packet n might have been lost
  – … packets n+1, n+2, and so on might get through

• Idea: have the receiver send ACK packets
  – ACK says that receiver is still awaiting nth packet
    • And repeated ACKs suggest later packets have arrived
  – Sender can view the “duplicate ACKs” as an early hint
    • … that the nth packet must have been lost
    • … and perform the retransmission early

• Fast retransmission
  – Sender retransmits data after the triple duplicate ACK

Effectiveness of Fast Retransmit

• When does Fast Retransmit work best?
  – Long data transfers
    • High likelihood of many packets in flight
  – Large window size
    • High likelihood of many packets in flight
  – Low burstiness in packet losses
    • Higher likelihood that later packets arrive successfully

• Implications for Web traffic
  – Most Web transfers are short (e.g., 10 packets)
    • Short HTML files or small images
  – So, often there aren’t many packets in flight
  – … making fast retransmit less likely to “kick in”
  – Forcing users to like “reload” more often…

Tearing Down the Connection
### Tearing Down the Connection

Closing the connection
- Finish (FIN) to close and receive remaining bytes
  - And other host sends a FIN ACK to acknowledge
- Reset (RST) to close and not receive remaining bytes

### Sending/Receiving the FIN Packet

- Sending a FIN: close()
  - Process is done sending data via the socket
  - Process invokes “close()” to close the socket
  - Once TCP has sent all of the outstanding bytes…
  - … then TCP sends a FIN
- Receiving a FIN: EOF
  - Process is reading data from the socket
  - Eventually, the attempt to read returns an EOF

### DoS Attacks On TCP

- Denial of service (DoS): an action that prevents or impairs the authorized use of networks, systems, or applications by exhausting resources such as central processing units (CPU), memory, bandwidth, and disk space
  - TCP SYN Spoofing Attacks
  - TCP SYN Flooding Attacks
TCP SYN Spoofing

- A common DoS attack
- Attacks ability of a server to respond to future connection requests
- Overflowing tables used to manage them
  - Hence an attack on system resource

TCP Connection Management

**Three way handshake:**

**Step 1:** client host sends TCP SYN segment to server
- Specifies initial seq #
- Client address / port #, no data

**Step 2:** server host receives SYN, replies with SYNACK segment
- Server allocates buffers
- Specifies server initial seq #

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data

SYN Spoofing Attack
SYN Spoofing Attack

• Attacker often uses either
  – Random source addresses
  – Or that of an overloaded spoofed host
  – To block return of (most) reset packets
• Has much lower traffic volume
  – Attacker can be on a much lower capacity link

TCP SYN Flooding Attacks

• 90% of DoS attacks use TCP SYN floods
• Streaming spoofed TCP SYNs
• Takes advantage of three way handshake
• Server start “half-open” connections
• These build up… until queue is full and all additional requests are blocked

Recall TCP Handshake

![TCP Handshake Diagram]
SYN Flooding

C

SYNCl

SYNc2

SYNc3

SYNc4

SYNc5

S

Listening

Store data

SYN Flood Attack

• Attacker causes TCP buffer to be exhausted with half-open connections
• No reply from target needed, so source may be spoofed.
• Claimed source must not be an active host.

SYN Flood Attack

• Attacker causes TCP buffer to be exhausted with half-open connections
• No reply from target needed, so source may be spoofed.
• Claimed source must not be an active host.
SYN Flooding

- For an effective attack, it is important that the spoofed IP addresses be unresponsive to the SYN-ACK segments that the victim will generate
  - If addresses of normal connected hosts are used, then those hosts will send the victim a TCP reset segment that will immediately free the corresponding resources

TCP Window Games

- Modified TCP Handshake
  - **client**
    - sends SYN packet and ACK number to server
    - waits for SYN-ACK from server to establish ACK number
  - **server**
    - responds with SYN-ACK packet to initial connection request
    - sends SYN packet to client with matching sequence number
  - **client**
    - sends ACK to server with matching sequence number, but no data
    - Receiving ACK, client sends data

Spoofed packets will not see 4096 window and will send data without waiting.

SYN-Cookies

- Modified TCP Handshake
  - **client**
    - sends SYN packet and ACK number to server
    - waits for SYN-ACK from server with matching ACK number
  - **server**
    - responds with SYN-ACK packet with initial SYN-cookie sequence number
    - Sequence number is cryptographically generated value based on client address, port, and time.
    - No TCP buffers are allocated
  - **client**
    - sends ACK to server with matching sequence number
  - **server**
    - If ACK is to an unopened socket, server validates returned sequence number as SYN-cookie, and responds with:
      - TCP buffer allocated
      - SYN-Cookie
      - No buffer allocated
      - TCP buffer allocated

Spoofed packets will not contain TCP buffers.
Conclusions

• Transport protocols
  – Multiplexing and demultiplexing
  – Sequence numbers
  – Window-based flow control
  – Timer-based retransmission
  – Checksum-based error detection