CSE 461: Transport Layer Connections

Naming Processes/Services

- Process here is an abstract term for your Web browser (HTTP), Email servers (SMTP), hostname translation (DNS), RealAudio player (RTSP), etc.

- How do we identify for remote communication?
  - Process id or memory address are OS-specific and transient

- So TCP and UDP use Ports
  - 16-bit integers representing mailboxes that processes “rent”
    - typically from OS
  - Identify endpoint uniquely as (IP address, protocol, port)
    - OS converts into process-specific channel, like “socket”
Processes as Endpoints

Picking Port Numbers

- We still have the problem of allocating port numbers
  - What port should a Web server use on host X?
  - To what port should you send to contact that Web server?

- Servers typically bind to "well-known" port numbers
  - e.g., HTTP 80, SMTP 25, DNS 53, ... look in /etc/services
  - Ports below 1024 reserved for "well-known" services

- Clients use OS-assigned temporary (ephemeral) ports
  - Above 1024, recycled by OS when client finished
User Datagram Protocol (UDP)

- Provides message delivery between processes
  - Source port filled in by OS as message is sent
  - Destination port identifies UDP delivery queue at endpoint

![UDP Delivery Diagram](image)

UDP Delivery
**UDP Checksum**

- UDP includes optional protection against errors
  - Checksum intended as an end-to-end check on delivery
  - So it covers data, UDP header

```
<table>
<thead>
<tr>
<th>0</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>DstPort</td>
<td>Length</td>
</tr>
<tr>
<td>Checksum</td>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>
```

**Transmission Control Protocol (TCP)**

- Reliable bi-directional bytestream between processes
  - Message boundaries are not preserved

- Connections
  - Conversation between endpoints with beginning and end

- Flow control
  - Prevents sender from over-running receiver buffers

- Congestion control
  - Prevents sender from over-running network buffers
TCP Header Format

- Ports plus IP addresses identify a connection

TCP Header Format

- Sequence, Ack numbers used for the sliding window
TCP Header Format

- Flags may be URG, ACK, PUSH, RST, SYN, FIN

```
<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>10</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SrcPort</td>
<td>DstPort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SequenceNum</td>
<td>Acknowledgment</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>0</td>
<td>Flags</td>
<td>AdvertisedWindow</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td></td>
<td>UrgPtr</td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

TCP Header Format

- Advertised window is used for flow control

```
<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>10</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SrcPort</td>
<td>DstPort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SequenceNum</td>
<td>Acknowledgment</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>0</td>
<td>Flags</td>
<td>AdvertisedWindow</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td></td>
<td>UrgPtr</td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
TCP Connection Establishment

- Both connecting and closing are (slightly) more complicated than you might expect.
- That they can work is reasonably straightforward.
- Harder is what to do when things go wrong:
  - TCP SYN+ACK attack.
- Close looks a bit complicated because both sides have to close to be done:
  - Conceptually, there are two one-way connections.
  - Don’t want to hang around forever if other end crashes.

Three-Way Handshake

- Opens both directions for transfer.

Active opener (client) Passive listener (server)

SYN, SequenceNum = x
SYN + ACK, SequenceNum = y
ACK, Acknowledgment = x + 1
Acknowledgment = y + 1
+data

\[ \text{SYN, SequenceNum} = x \]
\[ \text{SYN + ACK, SequenceNum} = y \]
\[ \text{ACK, Acknowledgment} = x + 1 \]
\[ \text{Acknowledgment} = y + 1 \]
Some Comments

- We could abbreviate this setup, but it was chosen to be robust, especially against delayed duplicates
  - Three-way handshake from Tomlinson 1975

- Choice of changing initial sequence numbers (ISNs) minimizes the chance of hosts that crash getting confused by a previous incarnation of a connection

- But with random ISN it actually proves that two hosts can communicate
  - Weak form of authentication

TCP State Transitions
Again, with States

<table>
<thead>
<tr>
<th>Active participant (client)</th>
<th>Passive participant (server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN_SENT</td>
<td>LISTEN</td>
</tr>
<tr>
<td>SYN, SequenceNum = x</td>
<td>SYN_RCVD</td>
</tr>
<tr>
<td>SYN + ACK, SequenceNum = y,</td>
<td>ACK, Acknowledgment = x + 1</td>
</tr>
<tr>
<td>Acknowledgment = y + 1</td>
<td></td>
</tr>
</tbody>
</table>

Connection Teardown

- Orderly release by sender and receiver when done
  - Delivers all pending data and “hangs up”

- Cleans up state in sender and receiver

- TCP provides a “symmetric” close
  - both sides shutdown independently
The TIME_WAIT State

- We wait 2MSL (two times the maximum segment lifetime of 60 seconds) before completing the close

- Why?
  - ACK might have been lost and so FIN will be resent
  - Could interfere with a subsequent connection
Berkeley Sockets interface

- Networking protocols implemented in OS
  - OS must expose a programming API to applications
  - most OSs use the “socket” interface
  - originally provided by BSD 4.1c in ~1982.

- Principle abstraction is a “socket”
  - a point at which an application attaches to the network
  - defines operations for creating connections, attaching to network, sending and receiving data, closing connections

TCP (connection-oriented)
**UDP (connectionless)**

**Server**
- Socket()
- Bind()
- Recvfrom()

**Client**
- Socket()
- Bind()
- Sendto()

- Block until Data from client
- Process request

- Data (request) → Data (reply)
- Sendto() → Recvfrom()

**Key Concepts**

- We use ports to name processes in TCP/UDP
  - "Well-known" ports are used for popular services
- Connection setup and teardown complicated by the effects of the network on messages
  - TCP uses a three-way handshake to set up a connection
  - TCP uses a symmetric disconnect