CSEP 561 – Connections

David Wetherall
djw@cs.washington.edu
Connections

• Focus
  – How do we (reliably) connect processes?
  – This is the transport layer

• Topics
  – Naming processes
  – Connection setup / teardown
  – Sliding window / Flow control
The Transport Layer

• Builds on the services of the Network layer
  – “TCP/IP”

• Communication between processes running on hosts
  – Naming/Addressing

• Stronger guarantees of message delivery make sense
  – Many applications want reliable connection and data transfer
  – This is the first layer that is talking “end-to-end”
Internet Transport Protocols

- **UDP**
  - Datagram abstraction between processes
  - With error detection

- **TCP**
  - Bytestream (bitpipe) abstraction between processes
  - With reliability
  - Plus congestion control (later!)
## Comparison of TCP/UDP/IP properties

<table>
<thead>
<tr>
<th>TCP</th>
<th>UDP</th>
<th>IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection-oriented</td>
<td>Datagram oriented</td>
<td>Datagram oriented</td>
</tr>
<tr>
<td>Reliable byte-stream</td>
<td>Lost packets</td>
<td>Lost packets</td>
</tr>
<tr>
<td>• In-order delivery</td>
<td>Reordered packets</td>
<td>Reordered packets</td>
</tr>
<tr>
<td>• Single delivery</td>
<td>Duplicate packets</td>
<td>Duplicate packets</td>
</tr>
<tr>
<td>• Arbitrarily length</td>
<td>Limited size packets</td>
<td>Limited size packets</td>
</tr>
<tr>
<td>Synchronization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flow control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Congestion control</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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”
  - Identify process uniquely as (IP address, protocol, port)
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
Berkeley Sockets

- 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
Overall pieces

app stuff write(), sendto(), send()  read(), recvfrom(), recv()

OS stuff Socket file descriptor  Socket file descriptor

Protocol stuff
Berkeley Sockets API

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET</td>
<td>Create a new communication end point</td>
</tr>
<tr>
<td>BIND</td>
<td>Associate a local address with a socket</td>
</tr>
<tr>
<td>LISTEN</td>
<td>Announce willingness to accept connections; give queue size</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Passively establish an incoming connection</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>SEND</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>Receive some data from the connection</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>
TCP (connection-oriented)

**Server**
- `Socket()`
- `Bind()`
- `Listen()`
- `Accept()`

**Client**
- `Socket()`
- `Connect()`
- `Send()`
- `Recv()`

*Connection Establishment*
- Block until connect
- Process request
- Data (request)
- Data (reply)
UDP (connectionless)

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

Block until Data from client
Process request

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

Data (request)

Sendto()

Data (reply)

`Sendto()`

`Recvfrom()`
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

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP length</td>
<td>UDP checksum</td>
</tr>
</tbody>
</table>
UDP Delivery

Packets arrive → DeMux → Message Queues → Application process

Ports → DeMux

Kernel boundary
UDP Checksum

- UDP includes optional protection against errors
  - Checksum intended as an end-to-end check on delivery
  - So it covers data, UDP header, and IP pseudoheader
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 (later)
  - Prevents sender from over-running receiver buffers

- Congestion control (later)
  - Prevents sender from over-running network buffers
TCP Delivery

Application process

TCP
Send buffer

Write bytes

Transmit segments
Segment Segment · · · Segment

TCP
Receive buffer

Application process

Read bytes
TCP Header Format

- Ports plus IP addresses identify a connection
TCP Header Format

- Sequence, Ack numbers used for the sliding window
  - Congestion control works by controlling the window size
TCP Header Format

- Flags bits may be SYN / FIN / RST / ACK, URG, and ECE / CWR
TCP Header Format

- Advertised window is used for flow control
Connection Establishment

• Both sender and receiver must be ready before we start to transfer the data
  – Sender and receiver need to agree on a set of parameters
  – e.g., the Maximum Segment Size (MSS)
• This is signaling
  – It sets up state at the endpoints
  – Compare to “dialing” in the telephone network

• In TCP a Three-Way Handshake is used
Three-Way Handshake

- Opens both directions for transfer
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

• With random ISN it proves two hosts can communicate
  – Weak form of authentication
TCP State Transitions

- Wow!
Again, with States

Active participant (client)

SYN_SENT

SYN, SequenceNum = x

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

ACK, Acknowledgment = y + 1

ESTABLISHED

Passive participant (server)

LISTEN

SYN_RCVD

ESTABLISHED

+data
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
TCP Connection Teardown

Web server

FIN_WAIT_1
FIN

FIN_WAIT_2
ACK

TIME_WAIT
FIN

... ACK

CLOSED

Web browser

CLOSE_WAIT

LAST_ACK

CLOSED

CLOSED
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
Stop-and-Wait

- Only one outstanding packet at a time
- Also called alternating bit protocol
Sliding Windows

- Stop-and-wait provides reliable transfer but has lousy performance if wire time $\ll$ prop. delay
  - How bad? You do the math
- Want to utilize all available bandwidth
  - Need to keep more data “in flight”
  - How much? Remember the bandwidth-delay product?
- Leads to Sliding Window Protocol
Sliding Window Protocol

- There is some maximum number of un-ACK’ed frames the sender is allowed to have in flight
  - We call this “the window size”
  - Example: window size = 2

Once the window is full, each ACK’ed frame allows the sender to send one more frame
Sliding Window: Sender

- Assign sequence number to each frame (SeqNum)
- Maintain three state variables:
  - send window size (SWS)
  - last acknowledgment received (LAR)
  - last frame sent (LFS)
- Maintain invariant: $\text{LFS} - \text{LAR} \leq \text{SWS}$

- Advance LAR when ACK arrives
- Buffer up to SWS frames
Sliding Window: Receiver

- Maintain three state variables
  - receive window size (RWS)
  - largest frame acceptable (LFA)
  - last frame received (LFR)
- Maintain invariant: \( LFA - LFR \leq RWS \)

- Frame \( \text{SeqNum} \) arrives:
  - if \( LFR < \text{SeqNum} \leq LFA \) ⇒ accept + send ACK
  - if \( \text{SeqNum} \leq LFR \) or \( \text{SeqNum} > LFA \) ⇒ discard

- Send cumulative ACKs – send ACK for largest frame such that all frames less than this have been received
Flow Control

- Sender must transmit data no faster than it can be consumed by the receiver
  - Receiver might be a slow machine
  - App might consume data slowly

- Implement by adjusting the size of the sliding window used at the sender based on receiver feedback about available buffer space
  - Receiver tells sender the highest sequence number it can use
Flow Control Example

Diagram showing the flow control process with states and events:
- Application does a 2K write
- Sender is blocked
- Sender may send up to 2K
- Receiver buffer states: Empty, Full, Application reads 2K
- Sequence and window control

Diagram illustrates the interaction between sender and receiver with specific sequence and window control actions.