CSE/EE 461 – Lecture 14

Connections

Last Time

• We began on the Transport layer

• Focus
  – How do we send information reliably?

• Topics
  – ARQ and sliding windows

This Time

• More on the Transport Layer

• Focus
  – How do we connect processes?

• Topics
  – Naming processes
  – Connection setup / teardown
  – Flow control
**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 process uniquely as (IP address, protocol, port)
    - OS converts into process-specific channel, like “socket”

**Processes as Endpoints**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Write(), sendto(), send()</td>
<td>Socket file descriptor</td>
</tr>
<tr>
<td></td>
<td>Read(), recvfrom(), recv()</td>
<td>Socket file descriptor</td>
</tr>
<tr>
<td>Operating</td>
<td>OS</td>
<td>Socket file descriptor</td>
</tr>
<tr>
<td></td>
<td>Protocol</td>
<td>Port</td>
</tr>
<tr>
<td></td>
<td>App Stuff</td>
<td>Port</td>
</tr>
</tbody>
</table>

**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

- Application process
- Message Queue
- DeMux on Port #
- 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 pseudoheder
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 Delivery

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, PSH, RST, SYN, FIN

TCP Header Format

- Advertised window is used for flow control
Other TCP Header Fields

- Header length allows for variable length TCP header
  - options for extensions such as timestamps, selective acknowledgements, etc.
- Checksum is analogous to that of UDP
- Urgent pointer/data not used in practice

TCP 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
  - Active opener (client)
  - Passive listener (server)
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
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
Web browser
FIN
CLOSE_WAIT
LAST_ACK
ACK
FIN
CLOSE_WAIT
FIN
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
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 4.3c 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)

- Server
  - `Socket()`
  - `Bind()`
  - `Listen()`
  - `Accept()`
  - `Recv()`
  - `Send()`
  - Block until `connect`
  - Process request:
    - Connection Establishment
    - Data (request)
    - Data (reply)

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

UDP (connectionless)

- Server
  - `Socket()`
  - `Bind()`
  - `Sendto()`
  - Block until `Data from client`
  - Process request:
    - Data (request)
    - `Sendto()`

- Client
  - `Socket()`
  - `Hbind()`
  - `Send()`
  - `Recvfrom()`
  - `Data (reply)`
Socket call

- Means by which an application attached to the network
  - #include <sys/socket.h>
- `int socket(int family, int type, int protocol)`
- **Family**: address family (protocol family)
  - AF_UNIX, AF_INET, AF_NS, AF_IMPLINK
- **Type**: semantics of communication
  - SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- **Protocol**: Usually set to 0 but can be set to specific value.
  - Not all combinations of family and type are valid
- Return value is a handle for new socket

Bind call

- Typically a server call
- Binds a newly created socket to the specified address
  - `int bind(int socket, struct sockaddr *address, int addr_len)`
- **Socket**: newly created socket handle
- **Address**: data structure of address of local system
  - IP address and port number (demux keys)
  - Same operation for both connection-oriented and connectionless servers
  - Can use well known port or unique port

Listen call

- Used by connection-oriented servers to indicate an application is willing to receive connections
- `int listen(int socket, int backlog)`
- **Socket**: handle of newly creates socket
- **Backlog**: number of connection requests that can be queued by the system while waiting for server to execute accept call.
Accept call

- A server call
- After executing `listen`, the accept call carries out a passive open (server prepared to accept connects).
- `int accept(int socket, struct sockaddr *address, int addr_len)`
- It blocks until a remote client carries out a connection request.
- When it does return, it returns with a new socket that corresponds with new connection and the address contains the client's address

Connect call

- A client call
- Client executes an active open of a connection
  - `int connect(int socket, struct sockaddr *address, int addr_len)`
  - How does the OS know where the server is?
- Call does not return until the three-way handshake (TCP) is complete
- Address field contains remote system's address
- Client OS usually selects random, unused port

Input and Output

- After connection has been made, application uses `send/recv` to data
- `int send(int socket, char *message, int msg_len, int flags)`
  - Send specified message using specified socket
- `int recv(int socket, char *buffer, int buf_len, int flags)`
  - Receive message from specified socket into specified buffer
- Or can use `read/write`
- Or can sometimes use `sendto/recvfrom`
- Or can use `sendmsg, recvmsg` for "scatter/gather"
Sample Code

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