Connections

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

• Topics
  – Naming processes
  – TCP / UDP
  – Connection setup / teardown
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 (ARQ with a sliding window, connections)
  - Plus flow and 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>
</tbody>
</table>
Relation to layers

Transport layer sends segments in packets (in frames)
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
  – Ports below 1024 reserved for “well-known” services, look in /etc/services

• Clients use OS-assigned temporary (ephemeral) ports
  – Above 1024, recycled by OS when client finished
Some well-known TCP ports

- Popular servers run on well-known ports

<table>
<thead>
<tr>
<th>Port</th>
<th>Protocol</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>20, 21</td>
<td>FTP</td>
<td>File transfer</td>
</tr>
<tr>
<td>22</td>
<td>SSH</td>
<td>Remote login, replacement for Telnet</td>
</tr>
<tr>
<td>25</td>
<td>SMTP</td>
<td>Email</td>
</tr>
<tr>
<td>80</td>
<td>HTTP</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>110</td>
<td>POP-3</td>
<td>Remote email access</td>
</tr>
<tr>
<td>143</td>
<td>IMAP</td>
<td>Remote email access</td>
</tr>
<tr>
<td>443</td>
<td>HTTPS</td>
<td>Secure Web (HTTP over SSL/TLS)</td>
</tr>
<tr>
<td>538</td>
<td>RTSP</td>
<td>Media player control</td>
</tr>
<tr>
<td>631</td>
<td>IPP</td>
<td>Printer sharing</td>
</tr>
</tbody>
</table>
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

**Protocol stuff**

Local Network  
Router  
Internet  
Router  
Local Network
## 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

1. Socket()
2. Bind()
3. Listen()
4. Accept()
   - Block until connect
   - Process request
5. Recv()
6. Send()

Client

1. Socket()
2. Connect()
3. Send()
4. Recv()

Connection Establishment
Data (request)
Data (reply)
UDP (connectionless)

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

Block until Data from client

Process request

Sendto()

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

Data (request)

Sendto()

Data (reply)

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 checksum**

Checksum covers UDP segment and IP pseudoheader

- Fields that change in the network are zeroed out
- Provides an end-to-end delivery check
UDP Delivery

Packets arrive
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

```
+-----------------+-----------------+-----------------+
| Source port     | Destination port|
| UDP length      | UDP checksum    |
+-----------------+-----------------+
```

32 Bits
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
The TCP Service Model

Applications using TCP see only the byte stream [right] and not the segments [left] sent as separate IP packets.

Four segments, each with 512 bytes of data and carried in an IP packet.

2048 bytes of data delivered to application in a single READ call.
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 Diagram]

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>TCP header length</td>
<td>Window size</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (0 or more 32-bit words)</td>
<td>Data (optional)</td>
</tr>
</tbody>
</table>
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

![Diagram showing the three-way handshake process with active (client) and passive (server) participants. The process involves initial SYN (SEQ = x) from the client, followed by SYN (SEQ = y, ACK = x + 1) from the server, and finally (SEQ = x + 1, ACK = y + 1) confirmation back from the client.]
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
Connection Establishment (4)

Three-way handshake protects against odd cases:

a) Duplicate CR. Spurious ACK does not connect

b) Duplicate CR and DATA. Same plus DATA will be rejected (wrong ACK).
TCP State Transitions

- Wow!
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></td>
</tr>
<tr>
<td>Acknowledgment = x + 1</td>
<td></td>
</tr>
<tr>
<td>ACK, Acknowledgment = y + 1</td>
<td></td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>+data</td>
<td></td>
</tr>
</tbody>
</table>
TCP Connections – simultaneous connect

TCP sets up connections with the three-way handshake

Normal case

Simultaneous connect
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
Connection Release (1)

Key problem is to ensure reliability while releasing

Asymmetric release (when one side breaks connection) is abrupt and may lose data
Symmetric release (both sides agree to release) can’t be handled solely by the transport layer

- Two-army problem shows pitfall of agreement
Normal release sequence, initiated by transport user on Host 1

- DR=Disconnect Request
- Both DRs are ACKed by the other side
TCP Connection Teardown

Web server

FIN_WAIT_1

FIN

FIN_WAIT_2

ACK

TIME_WAIT

FIN

CLOSE_WAIT

CLOSED

Web browser

LAST_ACK

ACK

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