### OSI Network Model

<table>
<thead>
<tr>
<th>Layer</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application</strong></td>
<td>HTTP, FTP, DNS</td>
</tr>
<tr>
<td><strong>Transport</strong></td>
<td>TCP, UDP</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>IP, ICMP (ping)</td>
</tr>
<tr>
<td><strong>Link</strong></td>
<td>Ethernet, WiFi</td>
</tr>
<tr>
<td><strong>Physical</strong></td>
<td>wires, signal encoding</td>
</tr>
</tbody>
</table>

(Hard to draw firm lines here)
Transport Layer Protocols

**User Datagram Protocol (UDP)**
- unreliable
- unordered delivery
- connectionless
- best-effort, segments might be lost, delivered out-of-order, duplicated
- reliability handled by app

**Transmission Control Protocol (TCP)**
- reliable
- in-order delivery
- connection setup
- flow control
- congestion control
Sockets Interface

1. Start server
   - socket
   - bind
   - listen
   - accept

Client / Server Session

- socket
- connect
- read
- write
- close

Connection request

Await connection request from next client

EOF
Sockets Interface: `socket`

- Clients and servers use the `socket` function to create a `socket descriptor`

```
int socket(int domain, int type, int protocol)
```

- Example:

```
int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

  Indicates that we are using 32-bit IPV4 addresses

  Indicates that the socket will be the end point of a connection

- Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.
Sockets Interface: `bind`

- A server uses `bind` to ask the kernel to associate the server’s socket address with a socket descriptor:

  ```c
  int bind(int sockfd, SA *addr, socklen_t addrlen);
  ```

- The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`.

- Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`.

- Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`. 
Sockets Interface: `listen`

- By default, kernel assumes that descriptor from socket function is an active socket that will be on the client end of a connection.

- A server calls the `listen` function to tell the kernel that a descriptor will be used by a server rather than a client:

  ```c
  int listen(int sockfd, int backlog);
  ```

- Converts `sockfd` from an active socket to a **listening socket** that can accept connection requests from clients.

- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.
Sockets Interface: `accept`

- Servers wait for connection requests from clients by calling `accept`:

  ```c
  int accept(int listenfd, SA *addr, int *addrlen);
  ```

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client’s socket address in `addr` and size of the socket address in `addrlen`.

- Returns a **connected descriptor** that can be used to communicate with the client via Unix I/O routines.
Connected vs. Listening Descriptors

- **Listening descriptor**
  - End point for client connection requests
  - Created once and exists for lifetime of the server

- **Connected descriptor**
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

- **Why the distinction?**
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request
Sockets Interface

1. Start server
   - socket
   - bind
   - listen
   - accept

2. Start client
   - socket
   - connect

Connection request

Client / Server Session

write
read
write
read
close

Await connection request from next client

EOF
Sockets Interface: `connect`

- A client establishes a connection with a server by calling `connect`:

  ```c
  int connect(int clientfd, SA *addr, socklen_t addrlen);
  ```

- Attempts to establish a connection with server at socket address `addr`
  - If successful, then `clientfd` is now ready for reading and writing.
  - Resulting connection is characterized by socket pair
    
    `(x:y, addr.sin_addr:addr.sin_port)`
  
  - `x` is client address
  - `y` is ephemeral port that uniquely identifies client process on client host

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`. 
1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

2. Client makes connection request by calling and blocking in `connect`

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`
TCP Connections

- TCP is connection-oriented

- A connection is initiated with a three-way handshake

- Recall: server will typically create a new socket to handle the new connection
Exercise 1: TCP Handshake

• Explain why three messages are required to set up a TCP connection
Sockets Interface

1. Start server
   - socket
   - bind
   - listen
   - accept

2. Start client
   - socket
   - connect

3. Exchange data
   - Connection request
   - write
   - read
   - write
   - close
   - read
   - close

Await connection request from next client
Reliable Transport

- Each SYN segment will include a randomly chosen sequence number.
- Sequence number of each segment is incremented by data length.
- Receiver sends ACK segments acknowledging latest sequence number received.
- Sender maintains copy of all sent but unacknowledged segments; resends if ACK does not arrive within timeout.
- Timeout is dynamically adjusted to account for round-trip delay.
# Transport-Layer Segment Formats

**UDP**

<table>
<thead>
<tr>
<th>Source Port #</th>
<th>Dest. Port #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>application message (payload)</td>
</tr>
</tbody>
</table>

**TCP**

<table>
<thead>
<tr>
<th>Source Port #</th>
<th>Dest. Port #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sequence number</td>
</tr>
<tr>
<td></td>
<td>acknowledgement number</td>
</tr>
<tr>
<td>HL U A P R S F</td>
<td>receive window</td>
</tr>
<tr>
<td></td>
<td>checksum</td>
</tr>
<tr>
<td></td>
<td>U data pointer</td>
</tr>
<tr>
<td></td>
<td>options</td>
</tr>
<tr>
<td></td>
<td>application message (payload)</td>
</tr>
</tbody>
</table>
Pipelined Protocols

- Pipelining allows sender to send multiple "in-flight", yet-to-be-acknowledged packets
  - increases throughput
  - needs buffering at sender and receiver

- how big should the window be?

what if a packet in the middle goes missing?
Example: Window Size = 4

- sender can have up to 4 unacknowledged messages
- when ACK for first message is received, it can send another message
TCP Fast Retransmit

- Receiver always acks the last id it successfully received
- Sender detects loss without waiting for timeout, resends missing packet
Exercise 2: TCP Sequence Numbers

Consider the sequence of transmitted messages shown on the right

- What will be the next ACK number sent by the server?
- What will be the next Seq number sent by the client?
TCP Congestion Control

- TCP operates under a principle of additive increase-multiplicative decrease
  - window size++ every RTT if no packets lost
  - window size/2 if a packet is dropped
TCP Fairness

- Goal: if $k$ TCP sessions share the same bottleneck link of bandwidth $R$, each should have average rate of $R/k$.

Loss: decreases throughput proportional to current bandwidth.

Congestion avoidance: increases throughput linearly (evenly).
TCP Slow Start

• Problem: linear increase takes a long time to build up a decent window size, and most transactions are small

• Solution: allow window size to increase exponentially until first loss
Exercise 3: TCP Window Size

• Assume someone changes the code of their TCP client by modifying the congestion avoidance as follows:

  • instead of increasing the window size by 1 each time an ACK is received,

  • they double the window size each time an ACK is received (like in the slow-start phase).

• What would be the pros and cons of this modification?
**Sockets Interface**

1. **Start server**
   - socket
   - bind
   - listen
   - accept

2. **Start client**
   - socket
   - connect

3. **Exchange data**
   - read
   - write
   - read
   - write

4. **Disconnect client**
   - close

5. **Drop client**
   - close

**Client / Server Session**

- Connection request
- EOF
- Await connection request from next client
TCP Connections

- TCP is connection-oriented

- A connection is initiated with a three-way handshake

- Recall: server will typically create a new socket to handle the new connection

- FIN works (mostly) like SYN but to teardown a connection
TCP Summary

- Reliable, in-order message delivery

- Connection-oriented, three-way handshake

- Transmission window for better throughput
  - timeouts based on link parameters (e.g., RTT, variance)

- Congestion control
  - Linear increase, exponential backoff

- Fast adaptation
  - Exponential increase in the initial phase