Computer Architecture and Operating Systems
Lecture 13: Sockets

Andrei Tatarnikov

atatarnikov@hse.ru
@andrewt0301

Communication Protocol

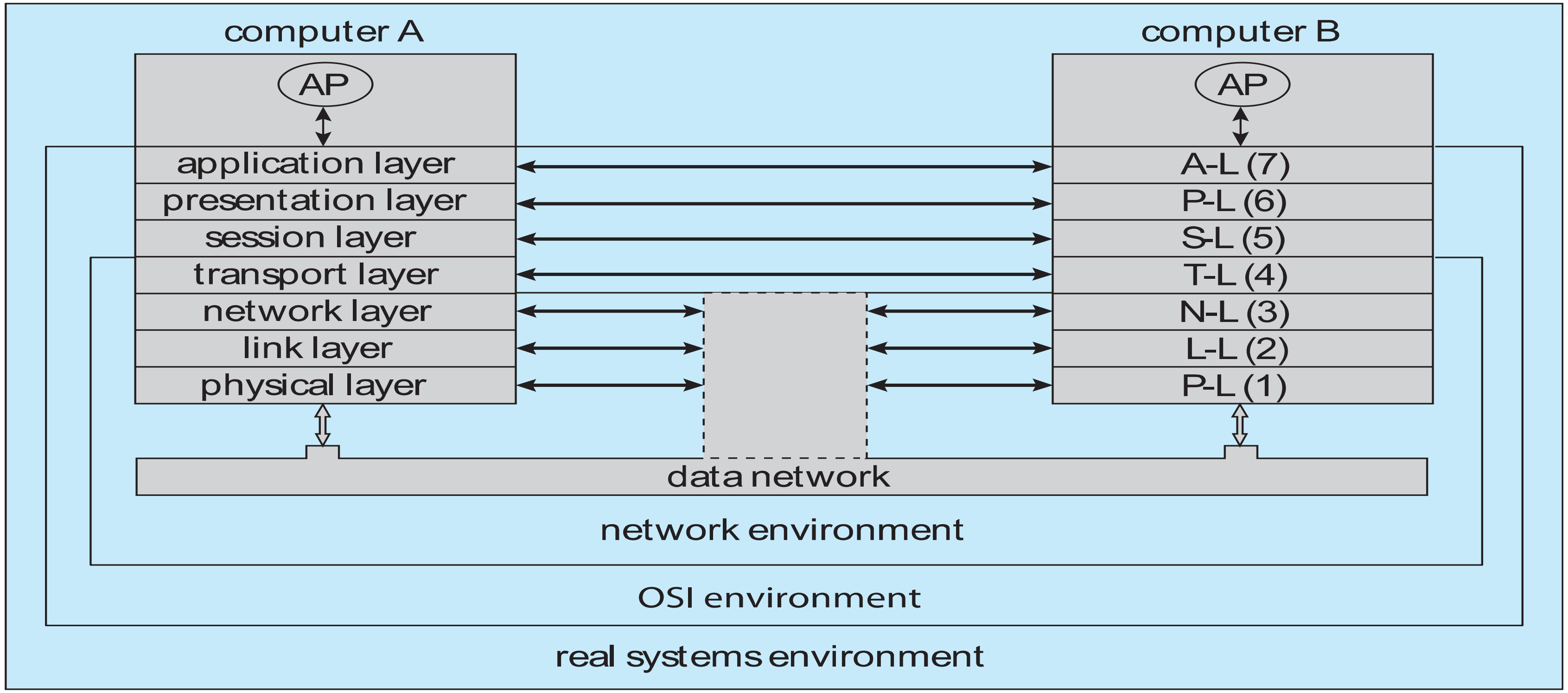
- **Layer 1: Physical layer** – handles the mechanical and electrical details of the physical transmission of a bit stream
- **Layer 2: Data-link layer** – handles the *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer
- **Layer 3: Network layer** – provides connections and routes packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels

Communication Protocol (Cont.)

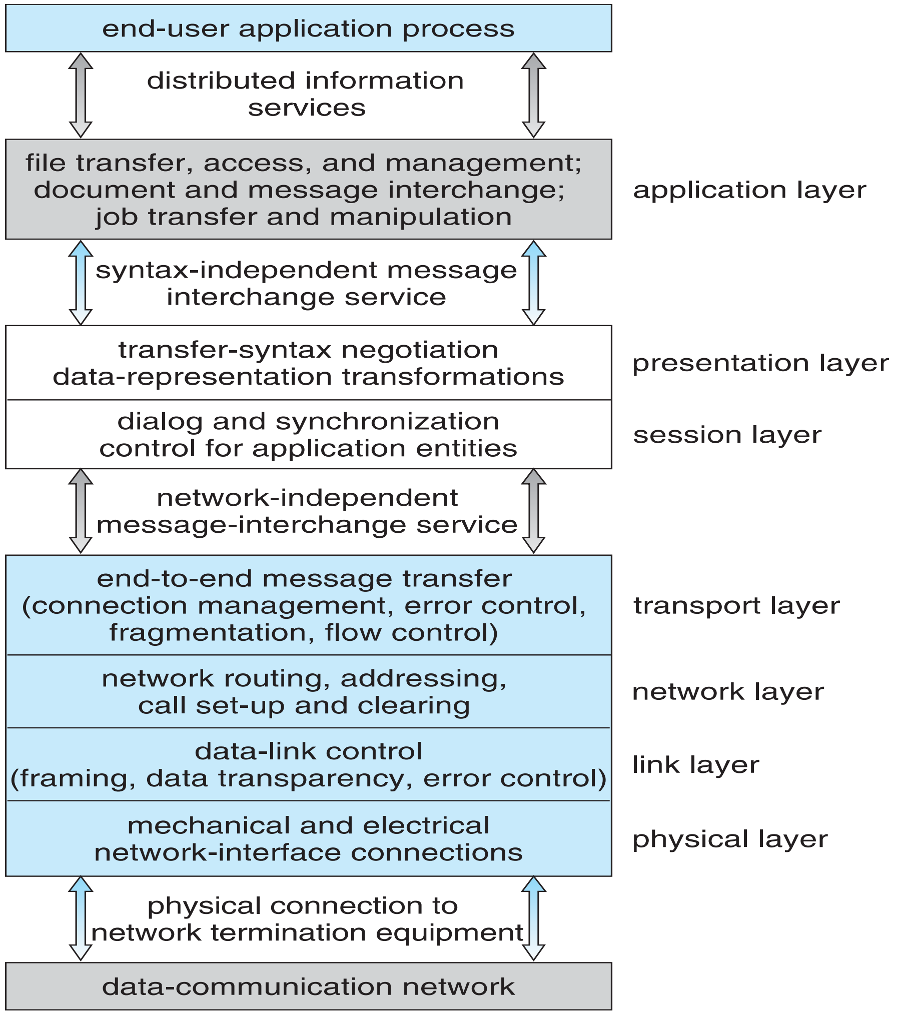
- **Layer 4: Transport layer** – responsible for low-level network access and for message transfer between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses
- **Layer 5: Session layer** – implements sessions, or process-to-process communications protocols
- **Layer 6: Presentation layer** – resolves the differences in formats among the various sites in the network, including character conversions, and half duplex/full duplex (echoing)
- **Layer 7: Application layer** – interacts directly with the users, deals with file transfer, remote-login protocols and electronic mail, as well as schemas for distributed databases

OSI Network Model

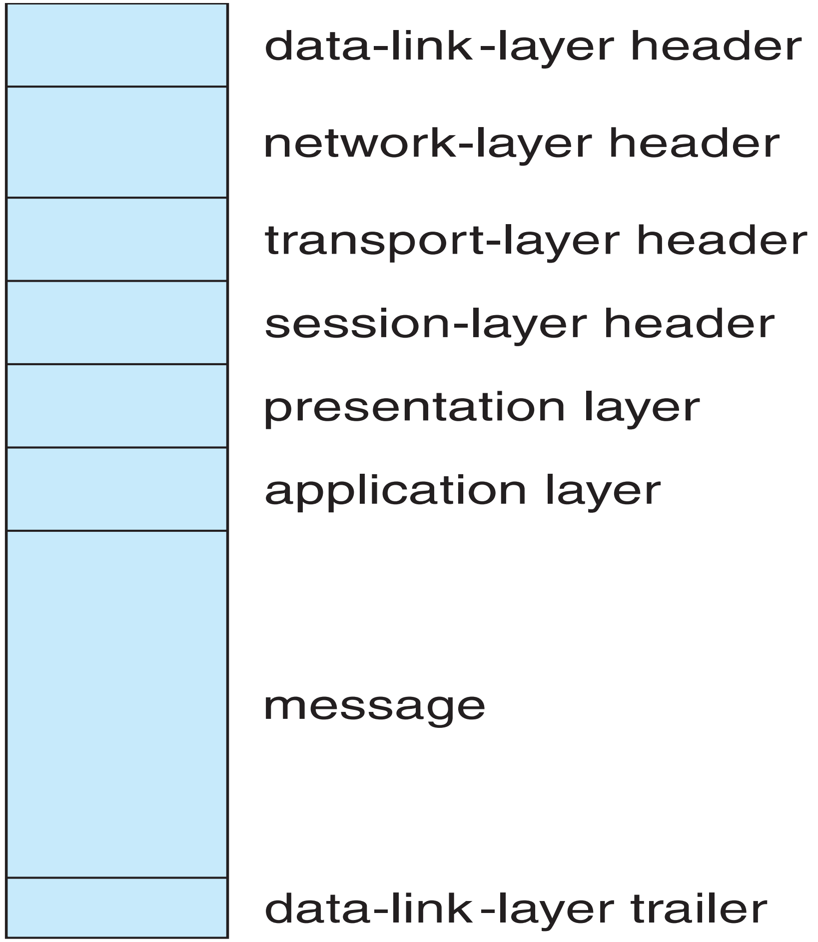
Logical communication between two computers, with the three lowest-level layers implemented in hardware



OSI Protocol Stack



OSI Network Message



OSI Model

- The OSI model formalizes some of the earlier work done in network protocols but was developed in the late 1970s and is currently not in widespread use
- The most widely adopted protocol stack is the TCP/IP model, which has been adopted by virtually all Internet sites
- The TCP/IP protocol stack has fewer layers than the OSI model. Theoretically, because it combines several functions in each layer, it is more difficult to implement but more efficient than OSI networking
- The relationship between the OSI and TCP/IP models is shown in the next slide

The OSI and TCP/IP Protocol Stacks

OSI	TCP/IP
application	HTTP, DNS, Telnet SMTP, FTP
presentation	not defined
session	not defined
transport	TCP-UDP
network	IP
data link	not defined
physical	not defined

TCP/IP Example

- Every host has a name and an associated **IP address** (host-id)
  - Hierarchical and segmented
- Sending system checks routing tables and locates a router to send packet
- Router uses segmented network part of host-id to determine where to transfer packet
  - This may repeat among multiple routers
- Destination system receives the packet
  - Packet may be complete message, or it may need to be reassembled into larger message spanning multiple packets

TCP/IP Example (Cont.)

- Within a network, how does a packet move from sender (host or router) to receiver?
  - Every Ethernet/WiFi device has a **medium access control** (MAC) address
  - Two devices on same LAN communicate via MAC address
  - If a system needs to send data to another system, it needs to discover the IP to MAC address mapping
    - Uses **address resolution protocol** (ARP)
  - A broadcast uses a special network address to signal that all hosts should receive and process the packet
    - Not forwarded by routers to different networks

Ethernet Packet

bytes		
7	preamble—start of packet	each byte pattern 10101010
1	start of frame delimiter	pattern 10101011
2 or 6	destination address	Ethernet address or broadcast
2 or 6	source address	Ethernet address
2	length of data section	length in bytes
0–1500	data	message data
0–46	pad (optional)	message must be > 63 bytes long
4	frame checksum	for error detection

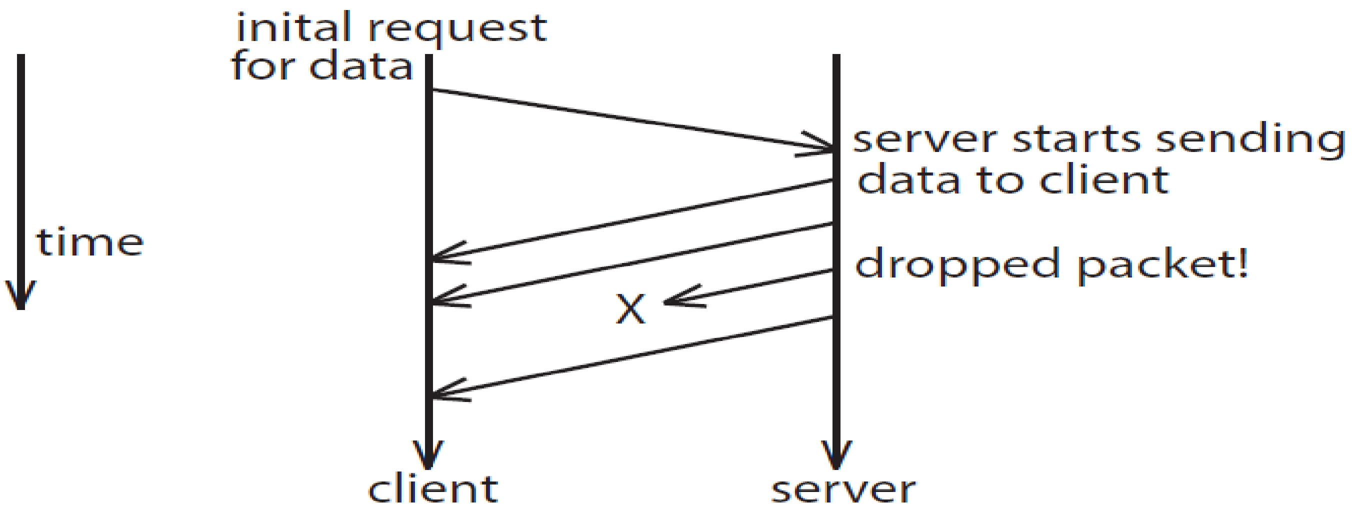
Transport Protocols UDP and TCP

- Once a host with a specific IP address receives a packet, it must somehow pass it to the correct waiting process
- Transport protocols TCP and UDP identify receiving and sending processes through the use of a port number
  - Allows host with single IP address to have multiple server/client processes sending/receiving packets
  - ***Well-known*** port numbers are used for many services
    - FTP – port 21
    - ssh – port 22
    - SMTP – port 25
    - HTTP – port 80
- Transport protocol can be simple or can add reliability to network packet stream

User Datagram Protocol

- UDP is ***unreliable*** – bare-bones extension to IP with addition of port number
  - Since there are no guarantees of delivery in the lower network (IP) layer, packets may become lost
  - Packets may also be received out-out-order
- UDP is also ***connectionless*** – no connection setup at the beginning of the transmission to set up state
  - Also no connection tear-down at the end of transmission
- UDP packets are also called **datagrams**

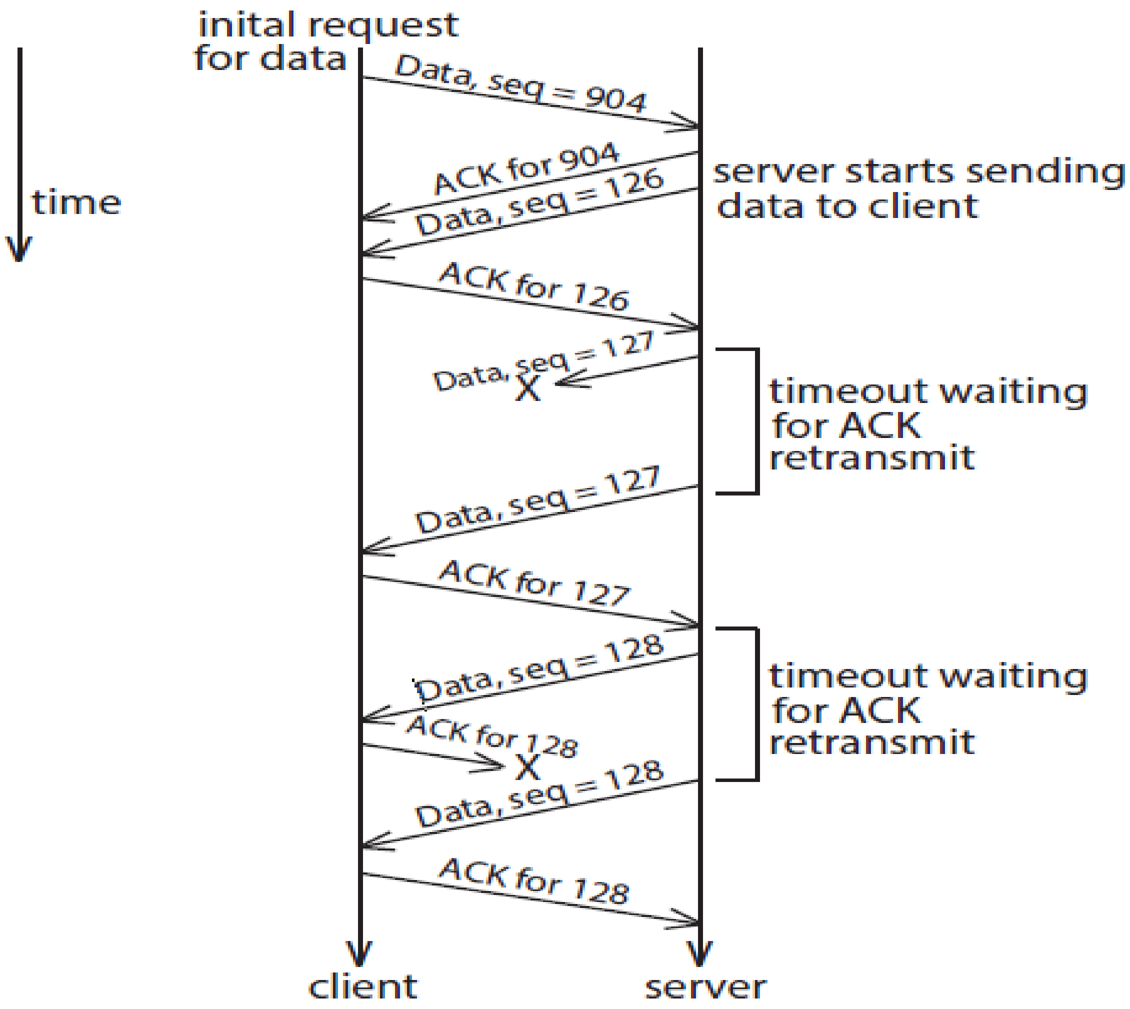
UDP Dropped Packet Example



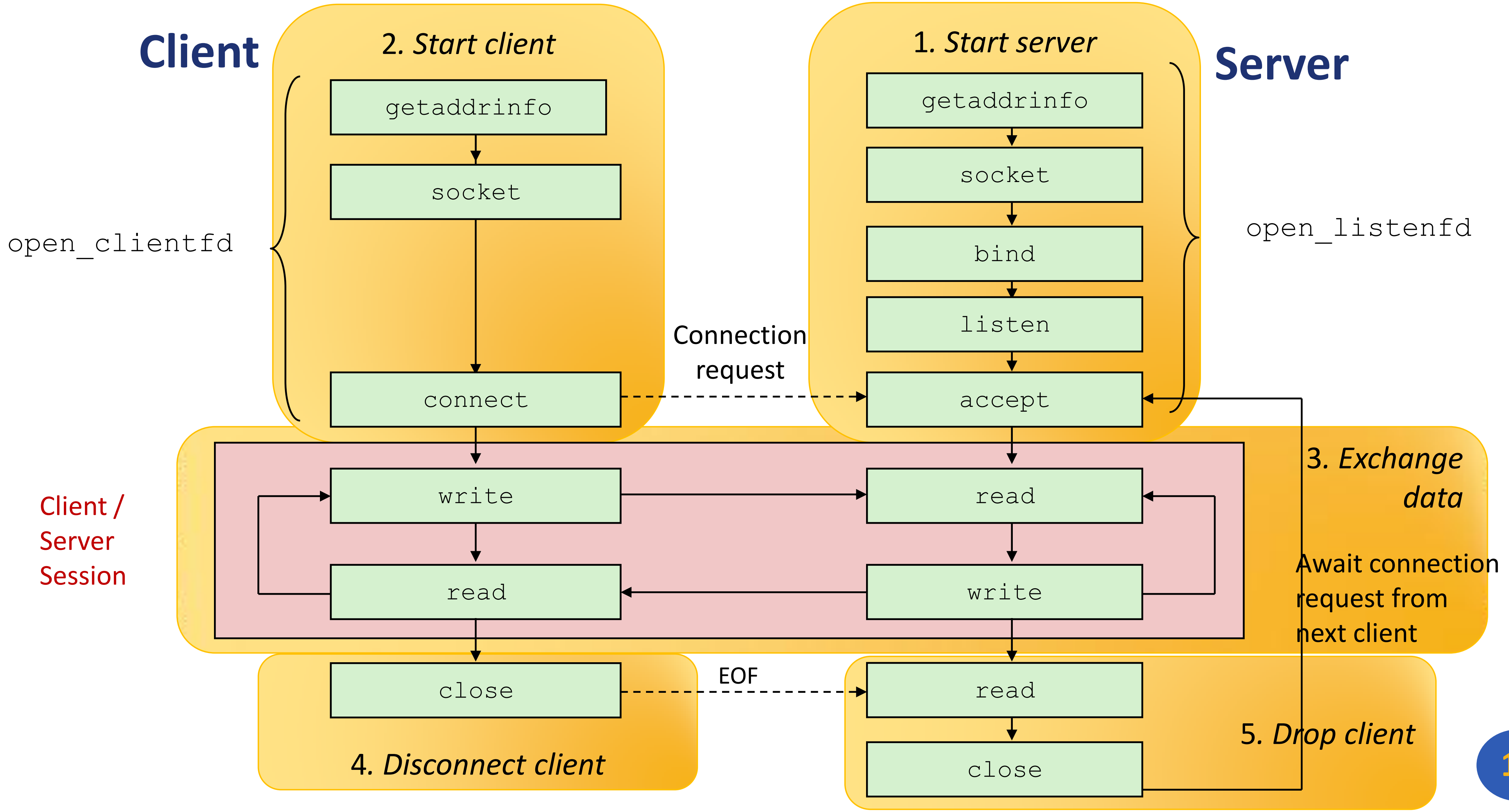
Transmission Control Protocol

- TCP is both ***reliable*** and ***connection-oriented***
- In addition to port number, TCP provides abstraction to allow in-order, uninterrupted ***byte-stream*** across an unreliable network
  - Whenever host sends packet, the receiver must send an **acknowledgement packet** (ACK). If ACK not received before a timer expires, sender will resend.
  - **Sequence numbers** in TCP header allow receiver to put packets in order and notice missing packets
  - Connections are initiated with series of control packets called a ***three-way handshake***
    - Connections also closed with series of control packets

TCP Data Transfer Scenario



Sockets Interface

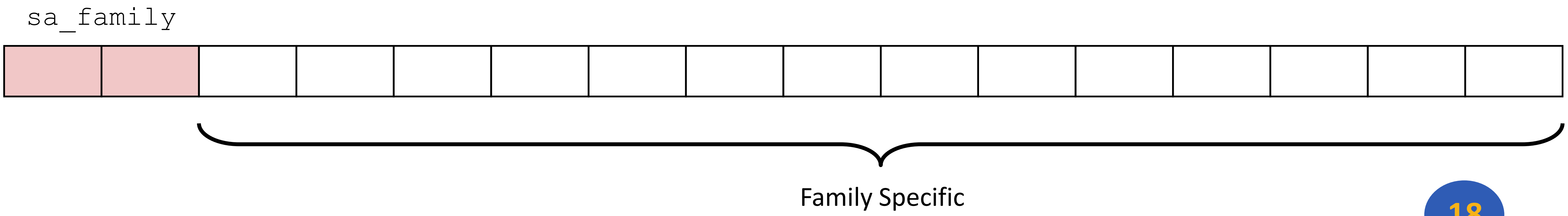


Recall: Socket Address Structures

- Generic socket address:
  - For address arguments to **connect**, **bind**, and **accept**
  - Necessary only because C did not have generic (**void ***) pointers when the sockets interface was designed
  - For casting convenience, we adopt the Stevens convention:

    typedef struct sockaddr SA;

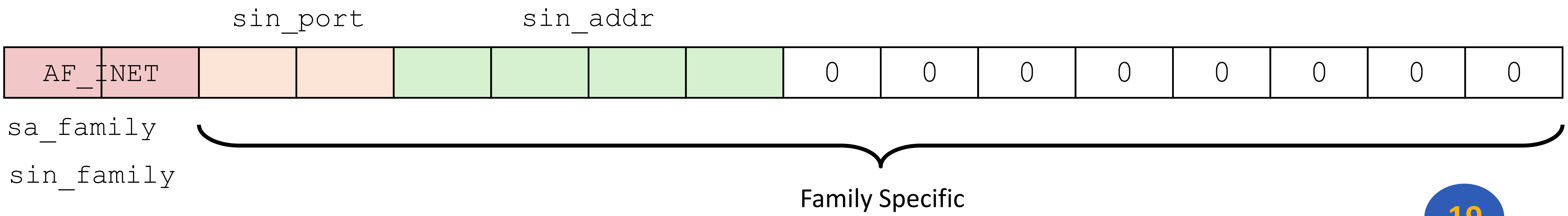
```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data.   */
};
```



Socket Address Structures

- Internet-specific socket address:
  - Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

```
struct sockaddr_in  {
  uint16_t        sin_family;  /* Protocol family (always AF_INET) */
  uint16_t        sin_port;    /* Port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```



Sockets Interface: **`socket`**

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

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

- Example:

Prot

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

ent.

Indicates that we are using 32-bit IPV4 addresses

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

Sockets Interface: `bind`

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

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

```
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`:

```
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.

Sockets Interface: `connect`

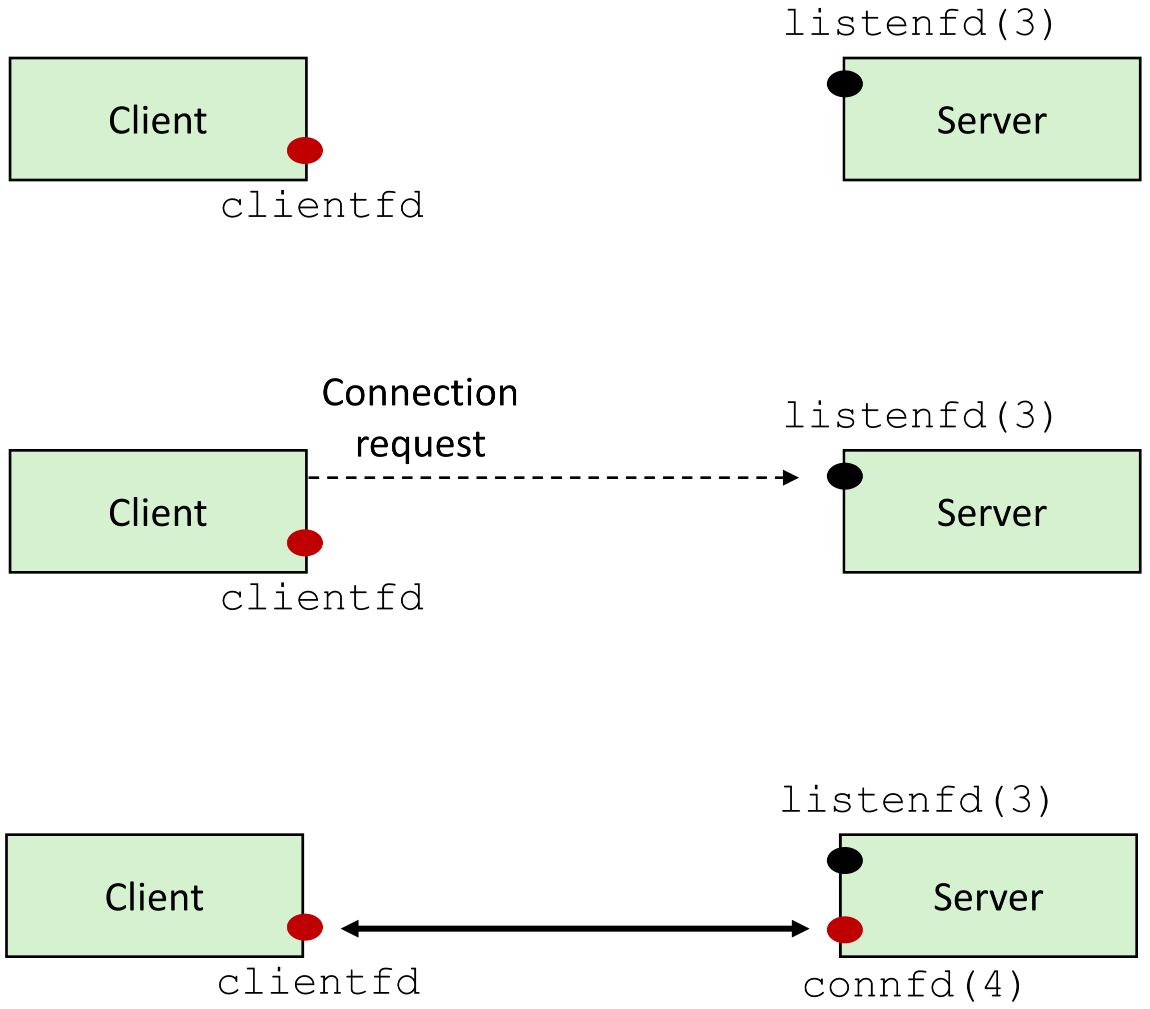
- A client establishes a connection with a server by calling connect:
- Attempts to establish a connection with server at socket address `addr`
  - If successful, then `clientfd` is now ready for reading and writing.
  - Resulting connection is characterized by socket pair

    `(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`.

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

`accept` Illustrated



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`

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

Testing Servers Using `telnet`

- The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections
  - Our simple echo server
  - Web servers
  - Mail servers

- Usage:
  - **`linux>`** ***`telnet <host> <portnumber>`***
  - Creates a connection with a server running on ***`<host>`*** and listening on port ***`<portnumber>`***

Any Questions?

```
            .text
__start:    addi t1, zero, 0x18
            addi t2, zero, 0x21
cycle:      beq t1, t2, done
            slt t0, t1, t2
            bne t0, zero, if_less
            nop
            sub t1, t1, t2
            j cycle
            nop
if_less:    sub t2, t2, t1
            j cycle
done:       add t3, t1, zero
```