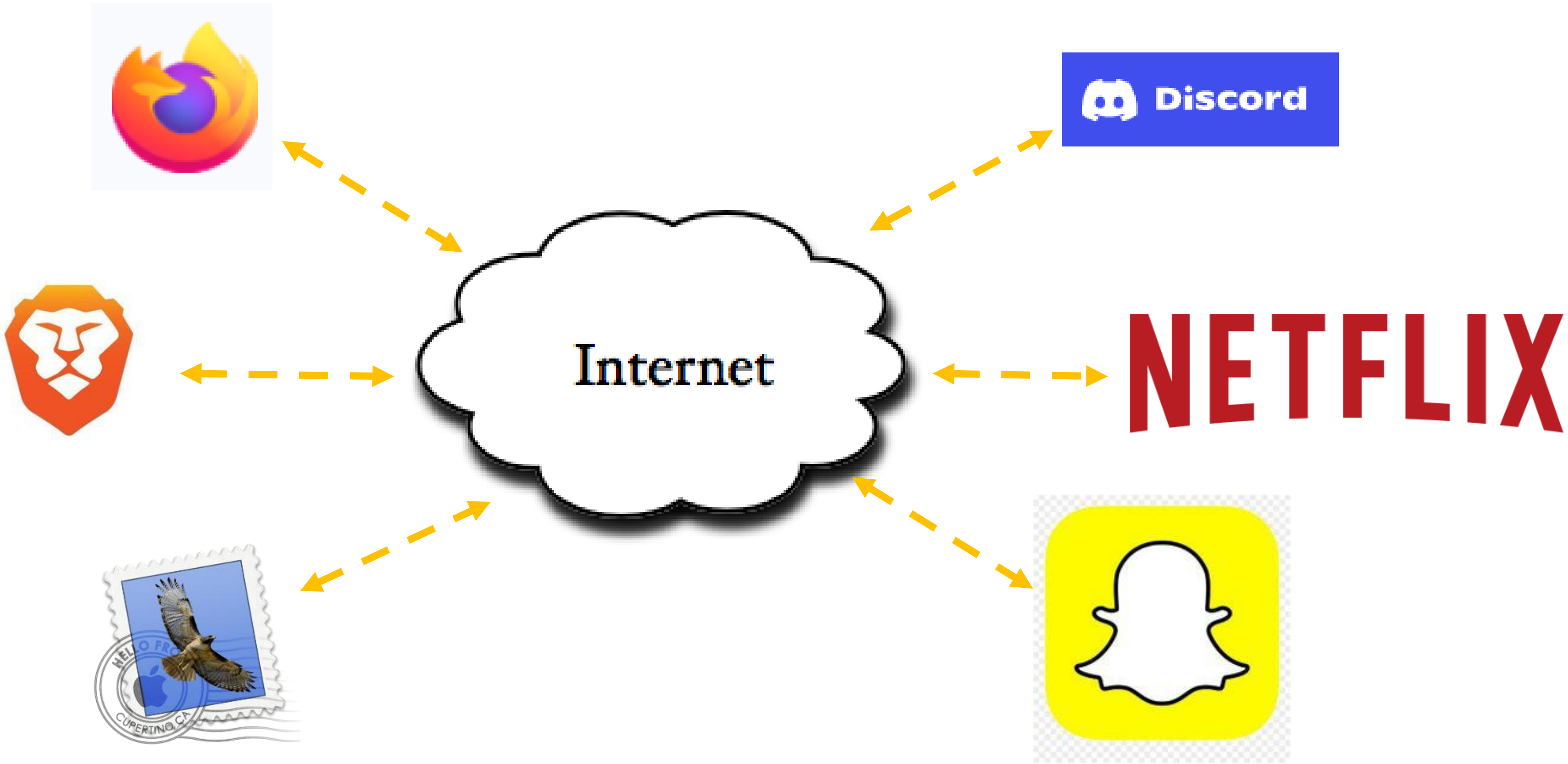
Networks

Sridhar Alagar

How can two processes communicate?

- Both processes in the same machine
  - Use pipes, or other IPC mechanisms such a shared memory, message passing
- Both the processes are in different machines (that are far apart)
  - networks

Networking



clients

servers

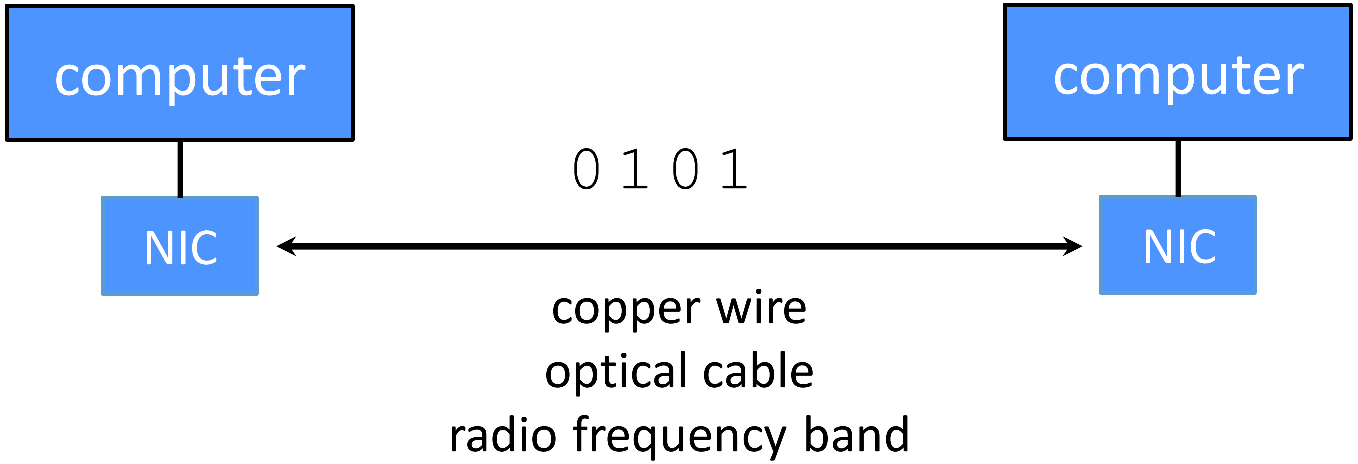
Layers

- Computer has several layers
  - Physical, OS, libraries, application

- Similarly, network has layers
  - Each layer provide some abstraction

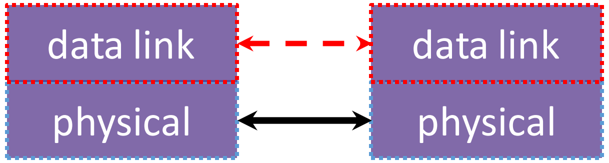
The Physical Layer

- Send/receive bits over a medium (wire/air)
  - 0 – off
  - 1 – on
- Bandwidth/Latency matters



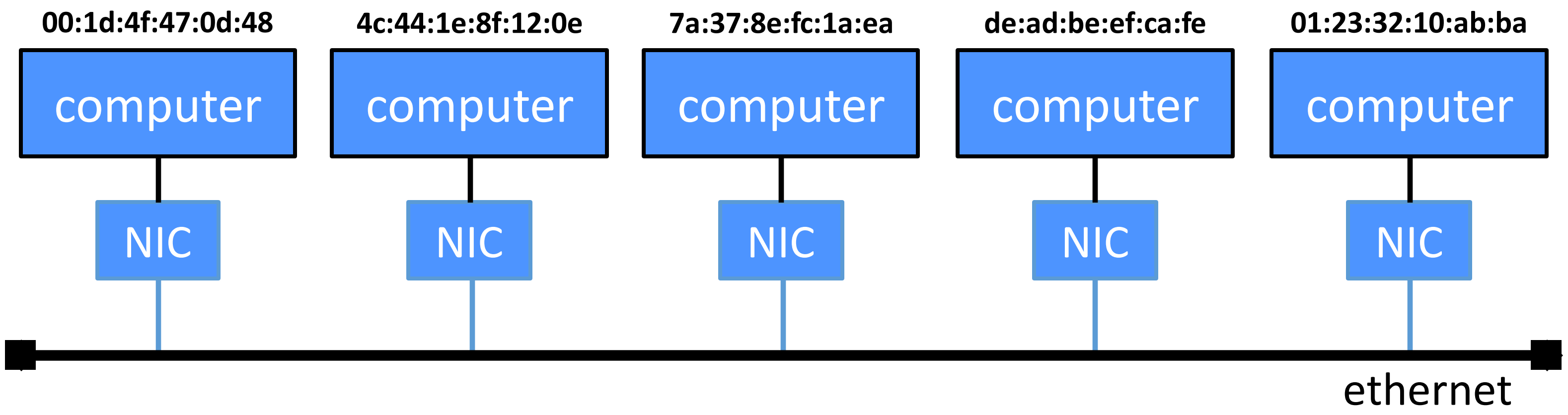
The Data Link Layer

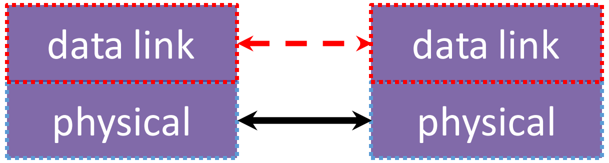
- Link layer is responsible for sending packets between two computers that are directly connected
- Specifies how
  - bits are "packetized"
  - network interface controllers (NICs) are addressed



The Data Link Layer

- Multiple computers on a LAN contend for the network medium
  - Media access control (MAC) specifies how computers cooperate



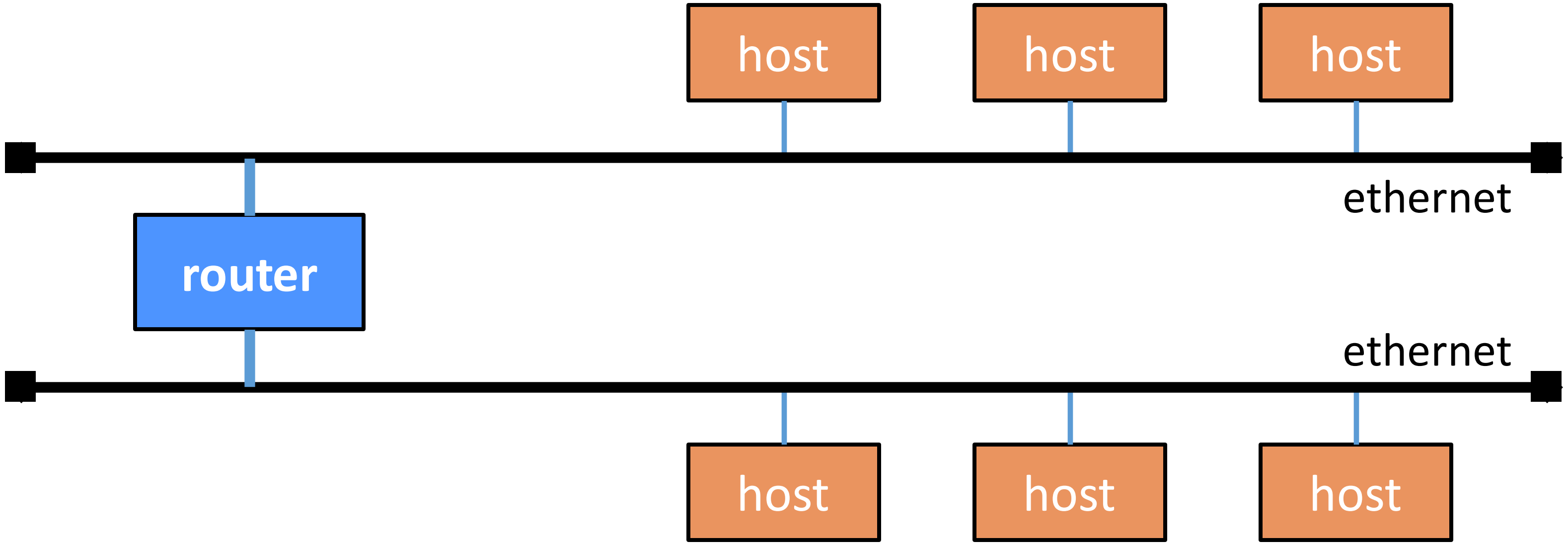


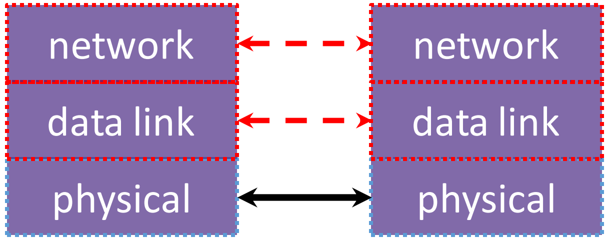
destination address	source address	data

ethernet header | ethernet payload

The Network Layer

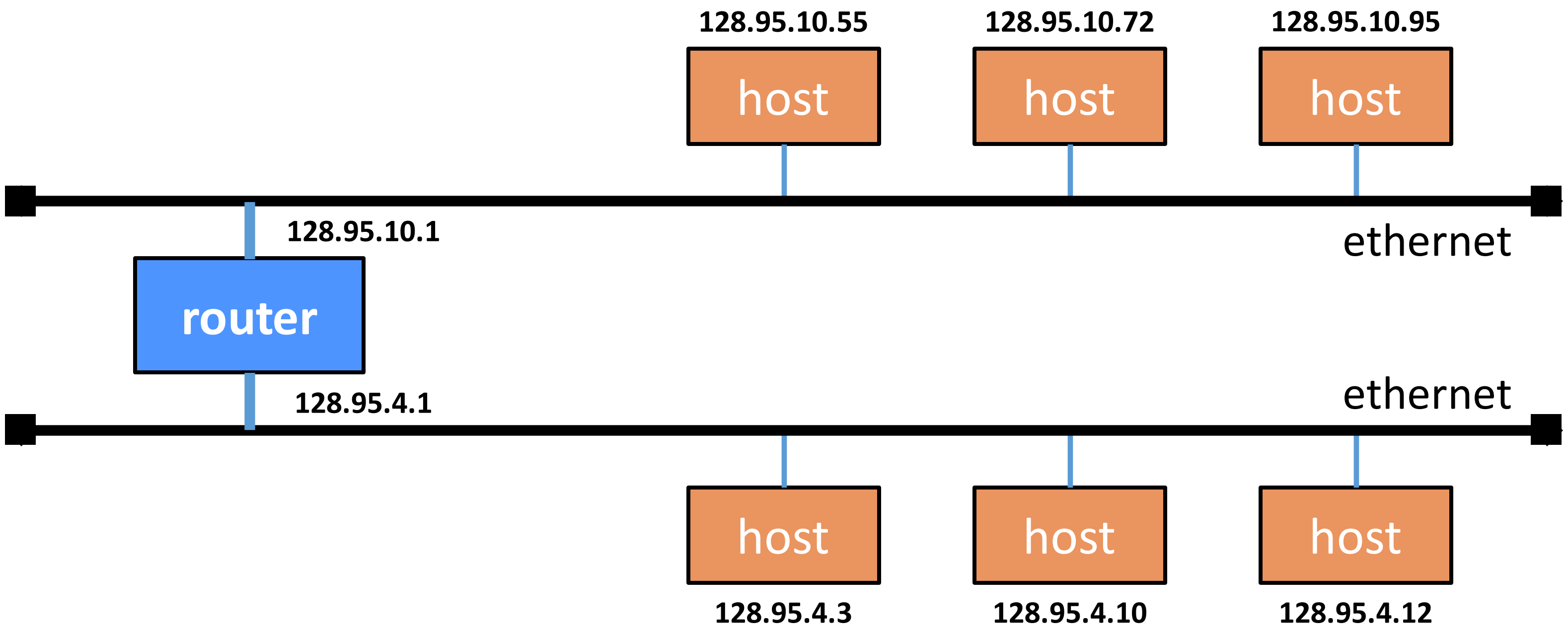
- Trying to transmit packets across different networks. This is where the "internet" is
- Individual networks are connected by routers that span networks

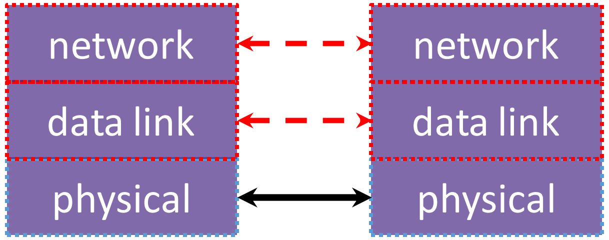




The Network Layer (IP)

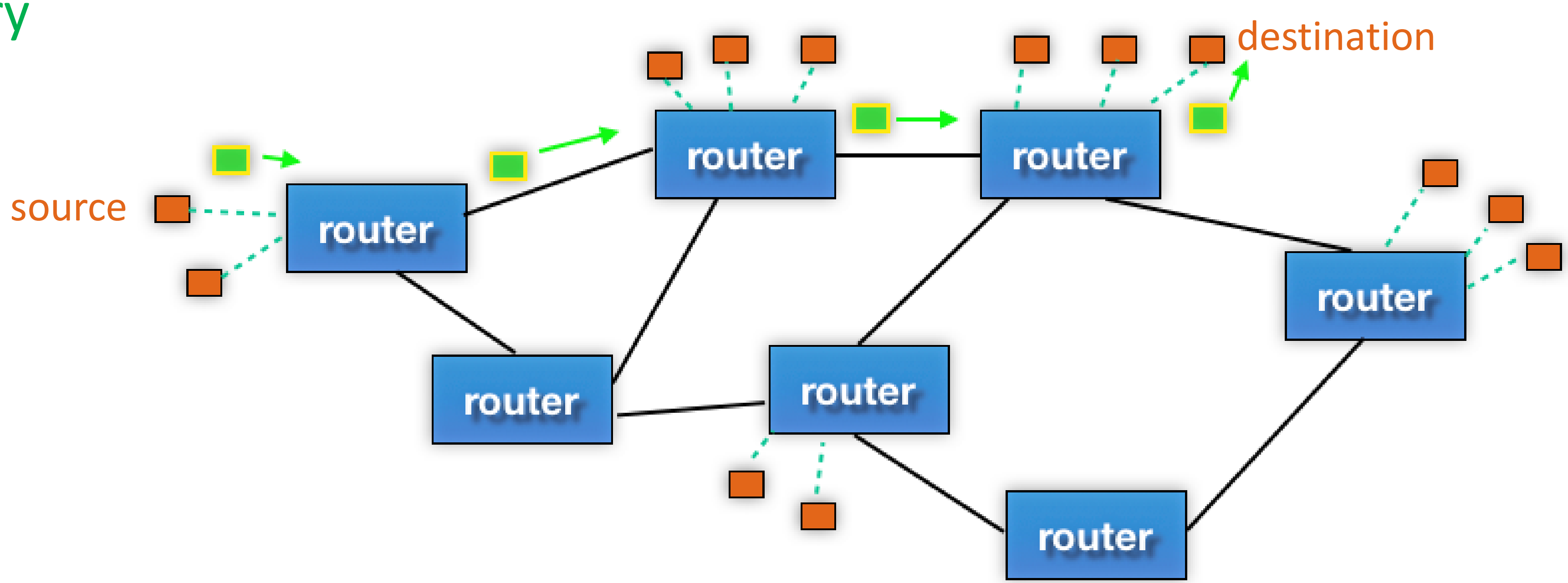
- Internet Protocol (IP) routes packets across multiple networks
  - Individual networks are connected by routers that span networks
  - Every computer has a unique IP address

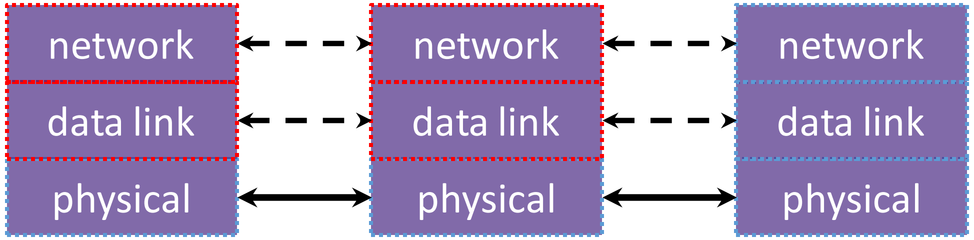




The Network Layer (IP)

- Protocol to route packets
  - Best effort delivery

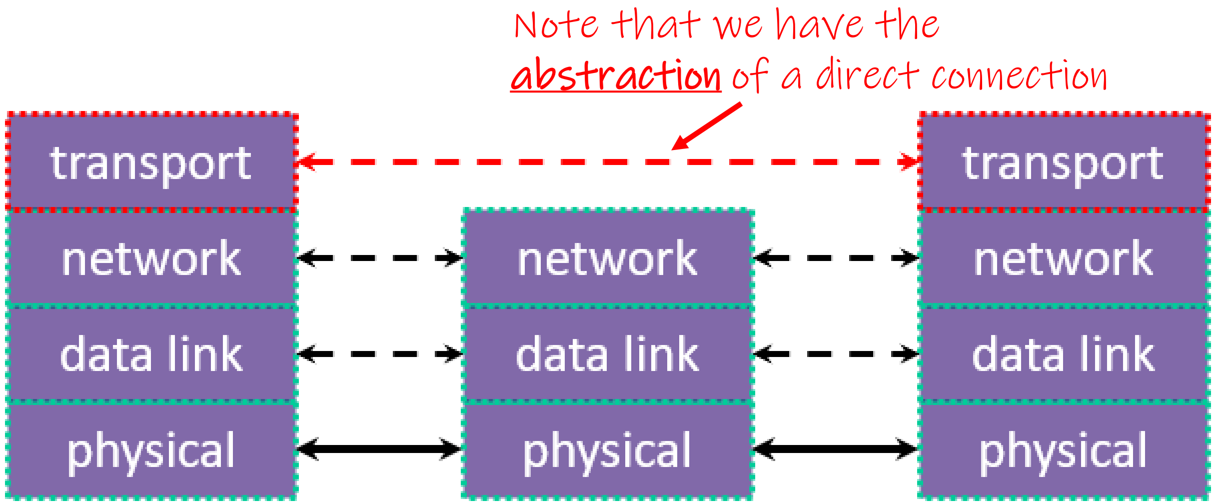




- Latency depends on
  - Distance
  - No of routers in path

The Transport Layer

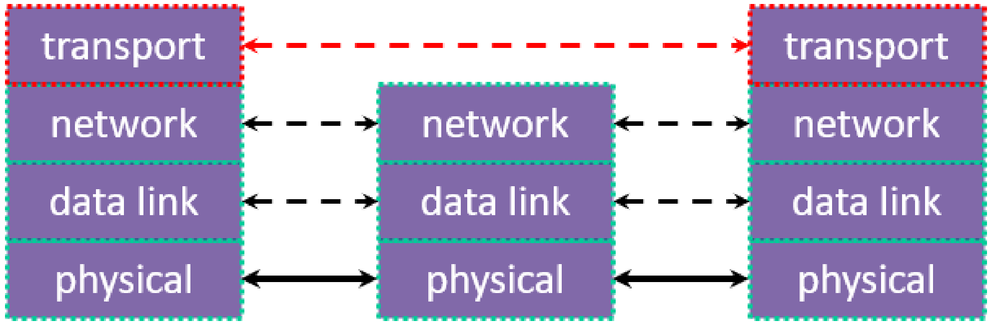
- Provides an end-to-end (direct) connection
  - Hides the complexities of the network layer and below
- Provides different protocols to interface between source and destination
  - e.g., Transmission Control Protocol (TCP), User Datagram Protocol (UDP)
  - These protocols still work with packets, but manages their order, reliability, multiple applications using the network



The Transport Layer - TCP

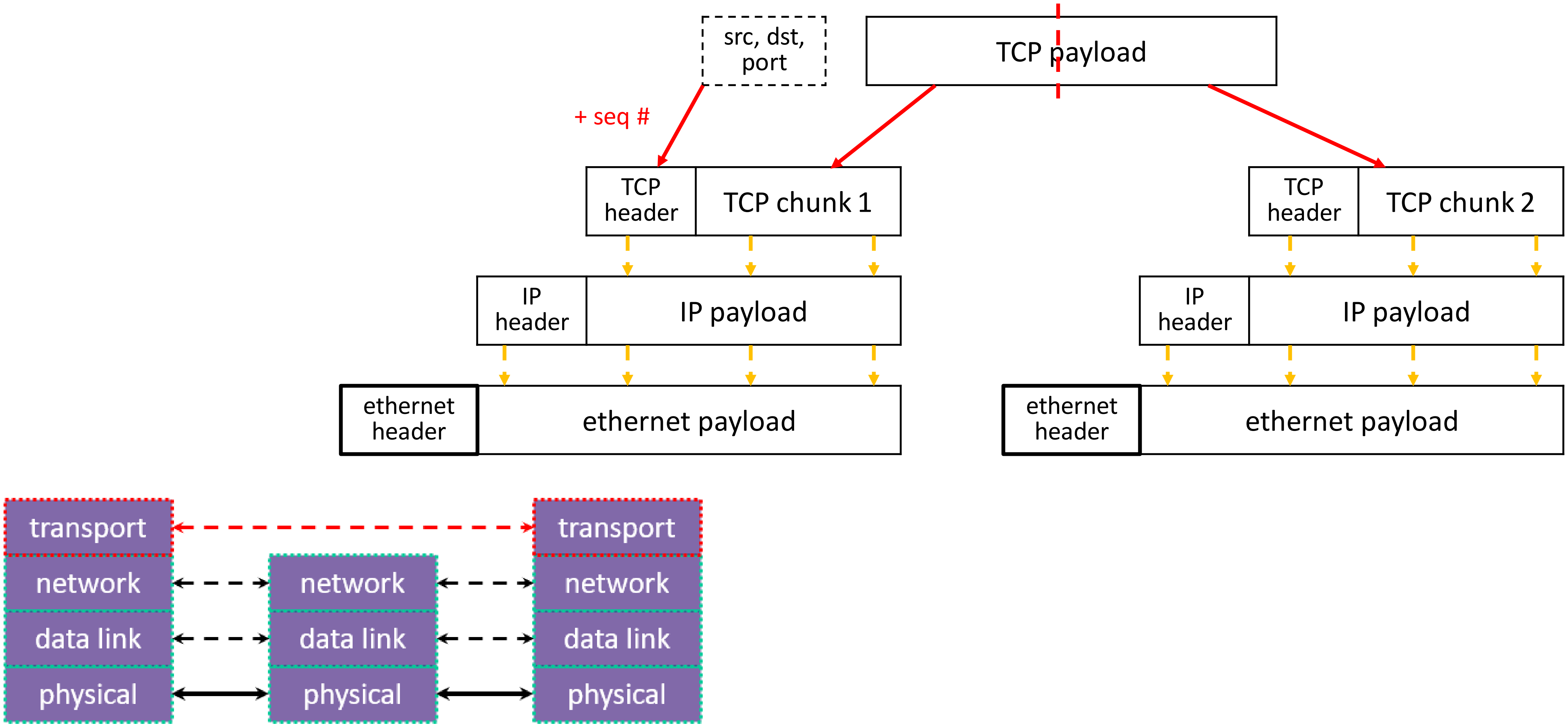
Transmission Control Protocol (TCP)

- Provides applications with reliable, ordered byte streams
  - Sends stream data as multiple IP packets (differentiated by sequence numbers) and retransmits them as necessary
  - When receiving, puts packets back in order and detects missing packets
- A single host (IP address) can have up to 2^{16} = 65,535 “ports”
  - Kind of like an apartment number at a postal address (your applications are the residents who get mail sent to an apt. #)



The Transport Layer - TCP

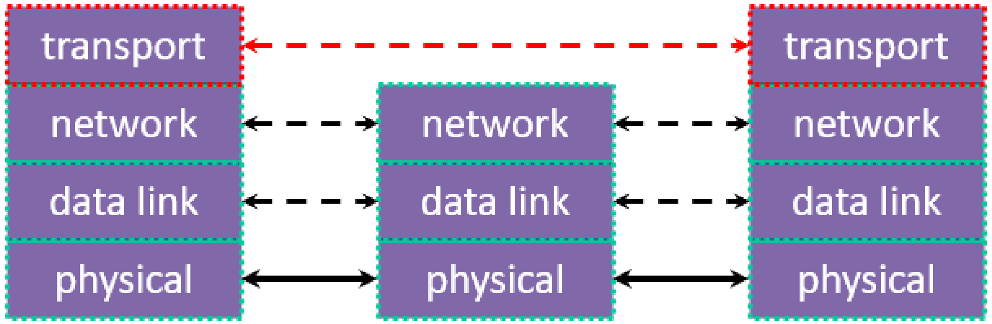
- Packet encapsulation



The Transport Layer - UDP

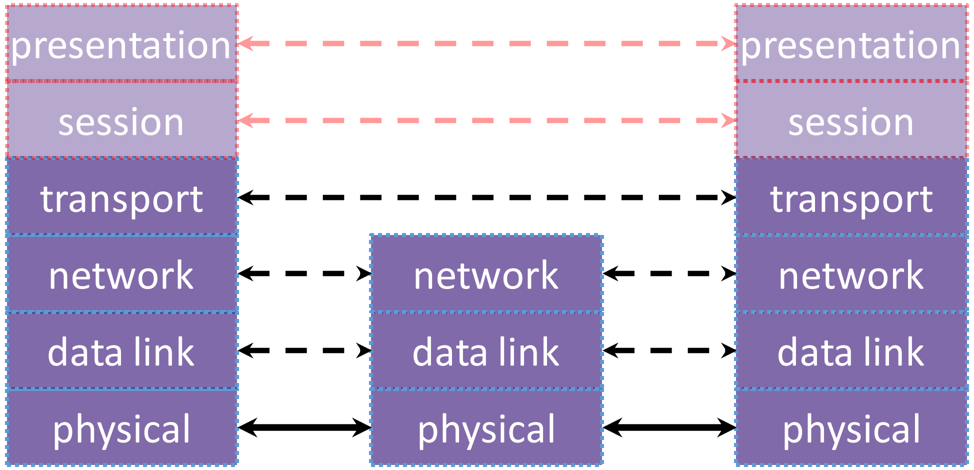
User Datagram Protocol (UDP)

- Provides applications with unreliable packet delivery
- UDP is a thin, simple layer on top of IP
  - Datagrams still are fragmented into multiple IP packets



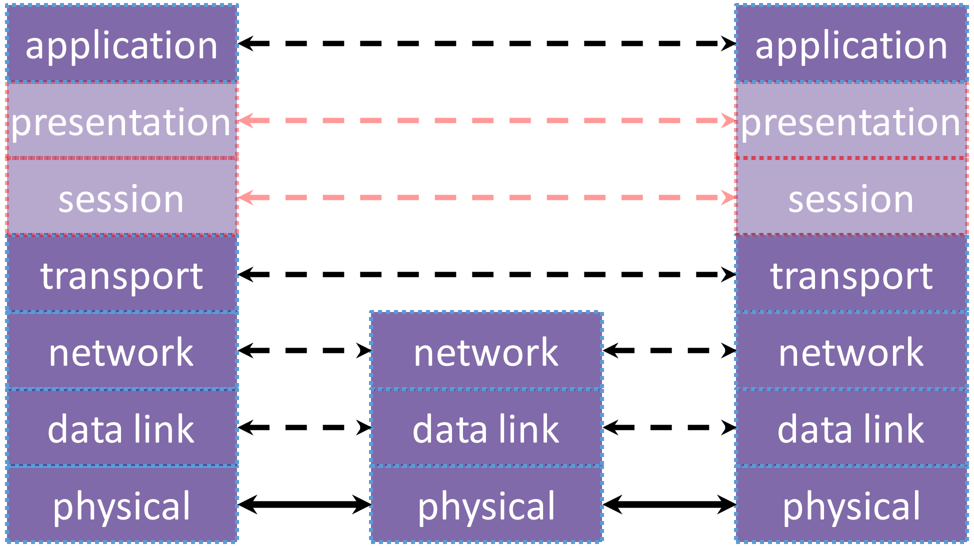
The (missing) Layers 5 and 6

- Layer 5 – session layer
  - Supposedly handles establishing and terminating application sessions
- Layer 6 – presentation layer
  - Datagrams still are fragmented into multiple IP packets



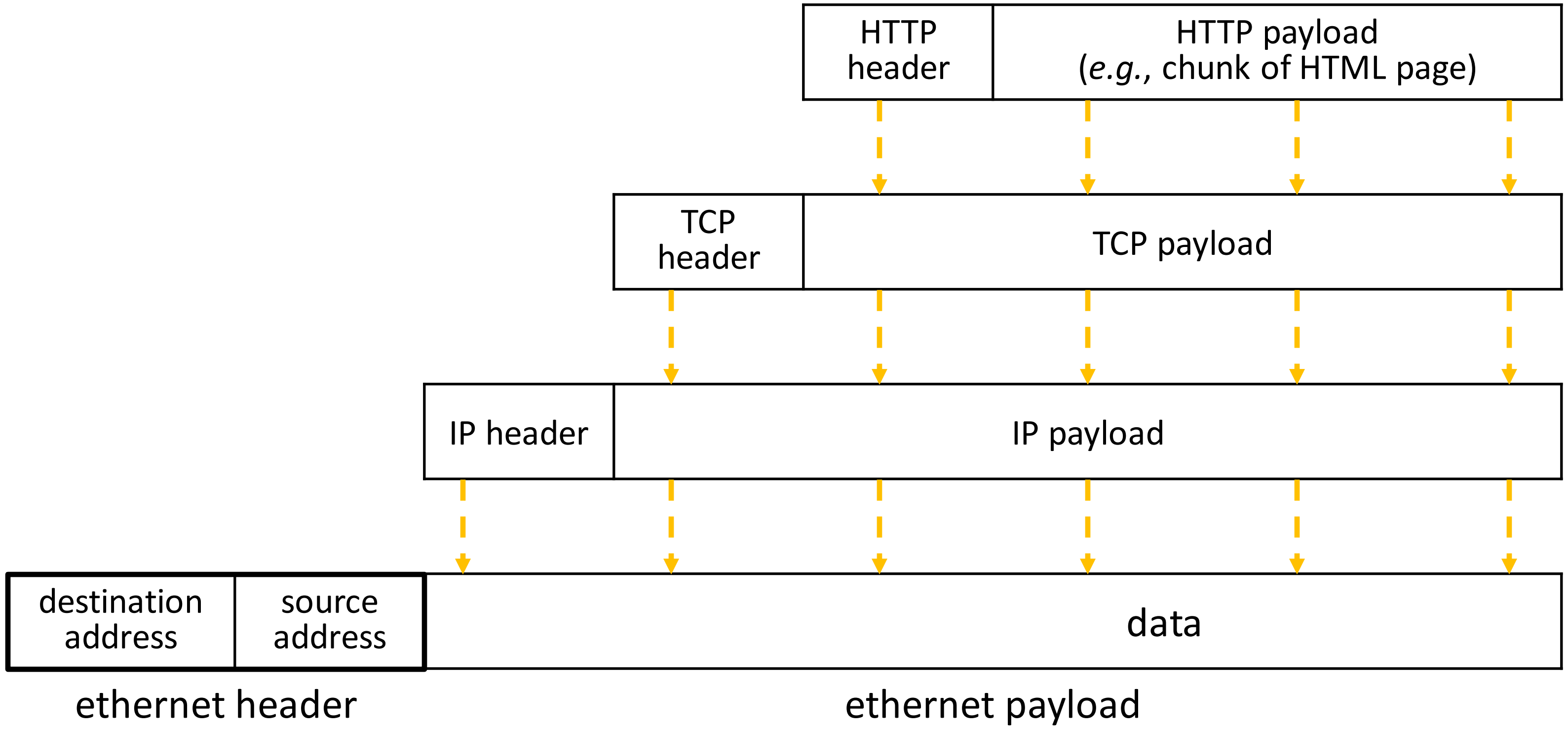
The Application Layer

- Application protocols
  - The format and meaning of messages between application entities
  - e.g., HTTP is an application-level protocol that dictates how web browsers and web servers communicate
    - HTTP is implemented on top of TCP streams



The Application Layer

- Packet encapsulation



The Application Layer

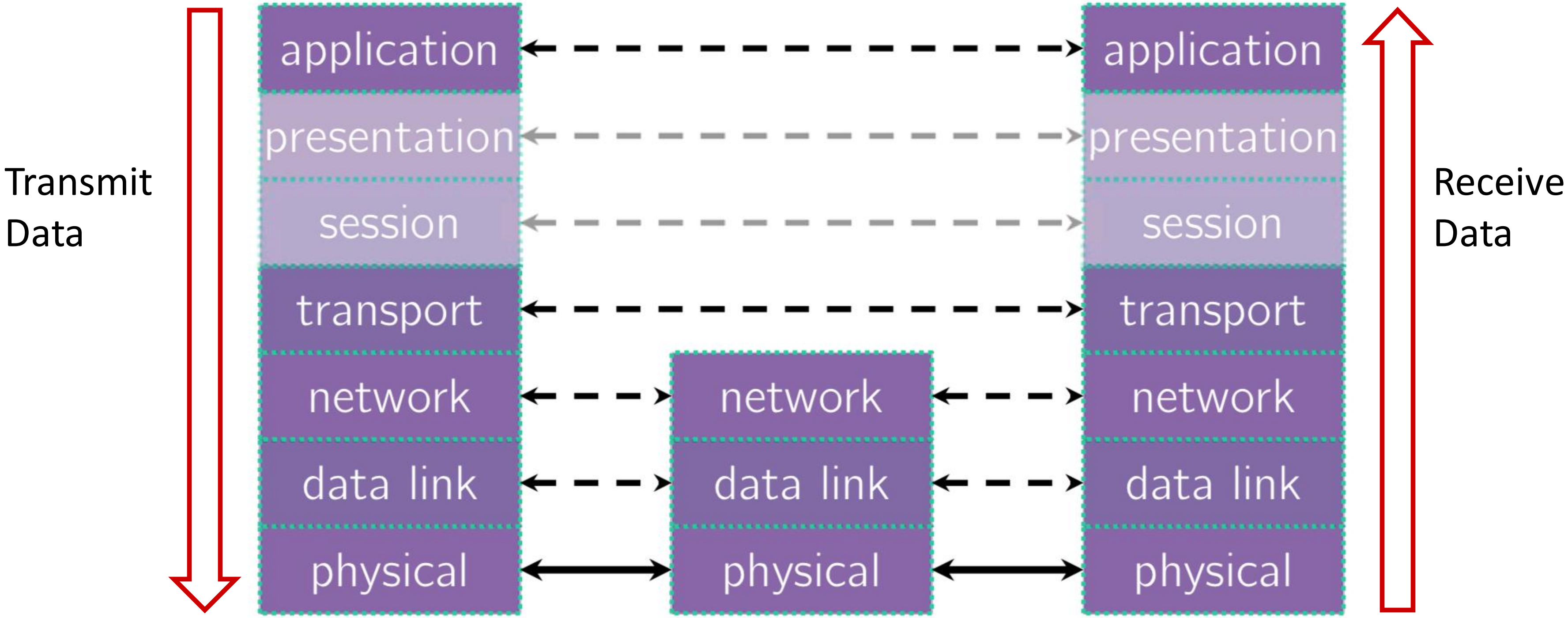
- Packet encapsulation

ethernet header	IP header	TCP header	HTTP header	HTTP payload (*e.g.*, chunk of HTML page)

The Application Layer

- Popular application-level protocols:
- DNS: translates a domain name (e.g., www.utdallas.edu) into one or more IP addresses (e.g., 10.176.92.9)
  - Domain Name System
  - A hierarchy of DNS servers cooperate to do this
- HTTP: web protocols
  - Hypertext Transfer Protocol
- SMTP, IMAP, POP: mail delivery and access protocols
  - Secure Mail Transfer Protocol, Internet Message Access Protocol, Post Office Protocol
- SSH: secure remote login protocol
  - Secure Shell
- bittorrent: peer-to-peer, swarming file sharing protocol

Actual Data Flow



Netcat demo

- netcat (`nc`) is “a computer networking utility for reading from and writing to network connections using TCP or UDP”
  - https://en.wikipedia.org/wiki/Netcat

  - Listen on port: `nc -l <port>`
  - Connect: `nc <hostname> <port>`

Files and File Descriptors

- Remember `open()`, `read()`, `write()`, and `close()`?

- POSIX system calls for interacting with files
  - `open()` returns a file descriptor
    - An integer that represents an open file
    - This file descriptor is then passed to `read()`, `write()`, and `close()`

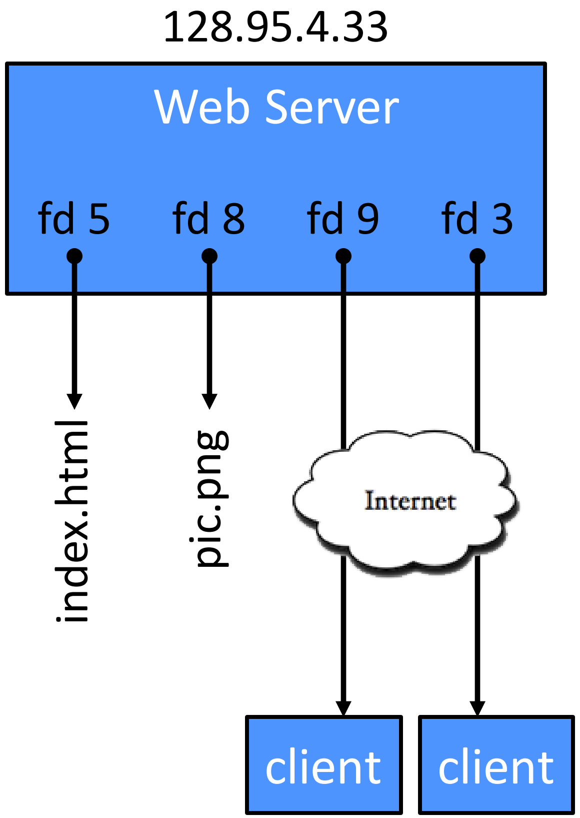
- Inside the OS, the file descriptor is used to index into a table that keeps track of any OS-level state associated with the file, such as the file position (offset)

- Remember any other system call that returns file descriptor?

Network and Sockets

- UNIX likes to make *all* I/O look like file I/O
- You use **`read`**`()` and **`write`**`()` to communicate with remote computers over the network!
- A file descriptor used for network communications is called a socket
- Just like with files:
  - Your program can have multiple network channels open at once
  - You need to pass a file descriptor to **`read`**`()` and **`write`**`()` to let the OS know which network channel to use

File Descriptor Table



OS's File Descriptor Table for the Process

File Descriptor	Type	Connection
0	pipe	stdin (console)
1	pipe	stdout (console)
2	pipe	stderr (console)
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

Types of Sockets

- Stream sockets
  - For connection-oriented, point-to-point, reliable byte streams
  - Using TCP, SCTP, or other stream transports

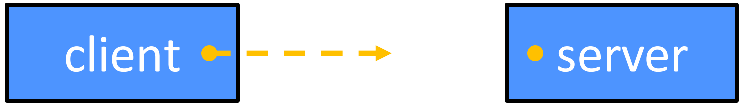
- Datagram sockets
  - For connection-less, one-to-many, unreliable packets
  - Using UDP or other packet transports

- Raw sockets
  - For layer-3 communication (raw IP packet manipulation)

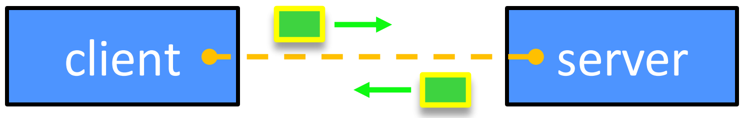
Stream Sockets

- Typically used for client-server communications
  - Client: An application that establishes a connection to a server
  - Server: An application that receives connections from clients
  - Can also be used for other forms of communication like peer-to-peer

1) Establish connection:



2) Communicate:

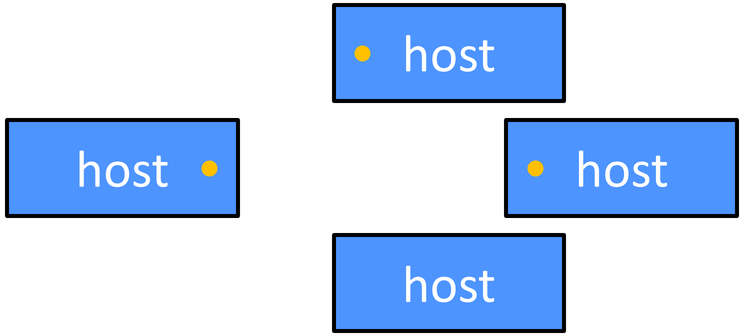


3) Close connection:

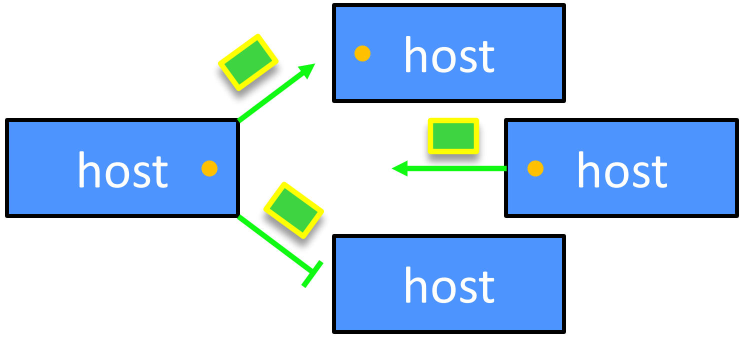
Datagram Sockets

- Often used as a building block
  - No flow control, ordering, or reliability, so used less frequently
  - *e.g.* streaming media applications or DNS lookups

1) Create Sockets:



2) Communicate:



Sockets API

- It is the standard API for network programming
- Available on most OS
- Written in C

Socket API: Client TCP Connection

- We'll start by looking at the API from the point of view of a client connecting to a server over TCP

- There are five steps:
  1) Figure out the IP address and port to which to connect
  2) Create a socket
  3) Connect the socket to the remote server
  4) **`read`**`()` and **`write`**`()` data using the socket
  5) Close the socket

Step 1: Figure Out IP Address and Port

- Several parts:
  - Network addresses
  - Data structures for address info
  - DNS (Domain Name System) – finding IP addresses

IPv4 Addresses

- An IPv4 address is a **4-byte** tuple
  - For humans, written in “dotted-decimal notation”
  - *e.g.* 128.95.4.1 (`80:5f:04:01` in hex)

- IPv4 address exhaustion
  - There are $2^{32} \approx 4.3$ billion IPv4 addresses
  - There are ≈ 7.87 billion people in the world (May 2021)

https://en.wikipedia.org/wiki/IPv4_address_exhaustion

IPv6 Addresses

- An IPv6 address is a **16-byte** tuple
  - Typically written in “hextets” (groups of 4 hex digits)
    - Can omit leading zeros in hextets
    - Double-colon replaces consecutive sections of zeros
  - *e.g.* `2d01:0db8:f188:0000:0000:0000:0000:1f33`
    - Shorthand: `2d01:db8:f188::1f33`
  - Transition is still ongoing

IPv4 Address Structures

```c
#include <arpa/inet.h>
```

```c
// IPv4 4-byte address
struct in_addr {
  uint32_t s_addr;                   // Address in network byte order
};

// An IPv4-specific address structure
struct sockaddr_in {
  sa_family_t    sin_family;    // Address family: AF_INET
  in_port_t      sin_port;      // Port in network byte order
  struct in_addr sin_addr;      // IPv4 address
  unsigned char  sin_zero[8];   // Pad out to 16 bytes
};
```

`struct sockaddr_in`:

family	port	addr	zero

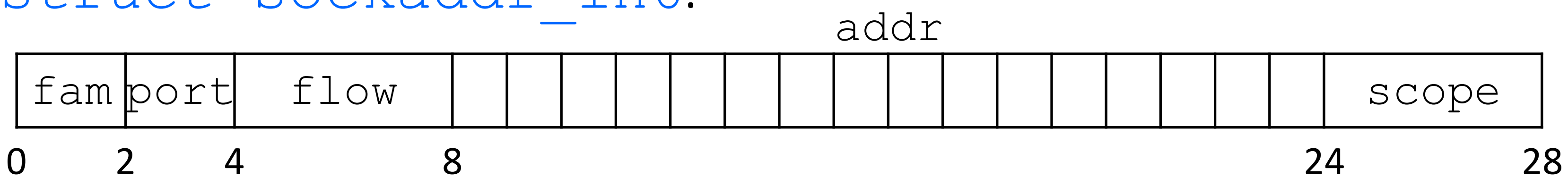
0 2 4 8 16

IPv6 Address Structures

```
// IPv6 16-byte address
struct in6_addr {
  uint8_t s6_addr[16];        // Address in network byte order
};

// An IPv6-specific address structure
struct sockaddr_in6 {
  sa_family_t     sin6_family;    // Address family: AF_INET6
  in_port_t       sin6_port;      // Port number
  uint32_t        sin6_flowinfo;  // IPv6 flow information
  struct in6_addr sin6_addr;      // IPv6 address
  uint32_t        sin6_scope_id;  // Scope ID
};
```

`struct sockaddr_in6`:

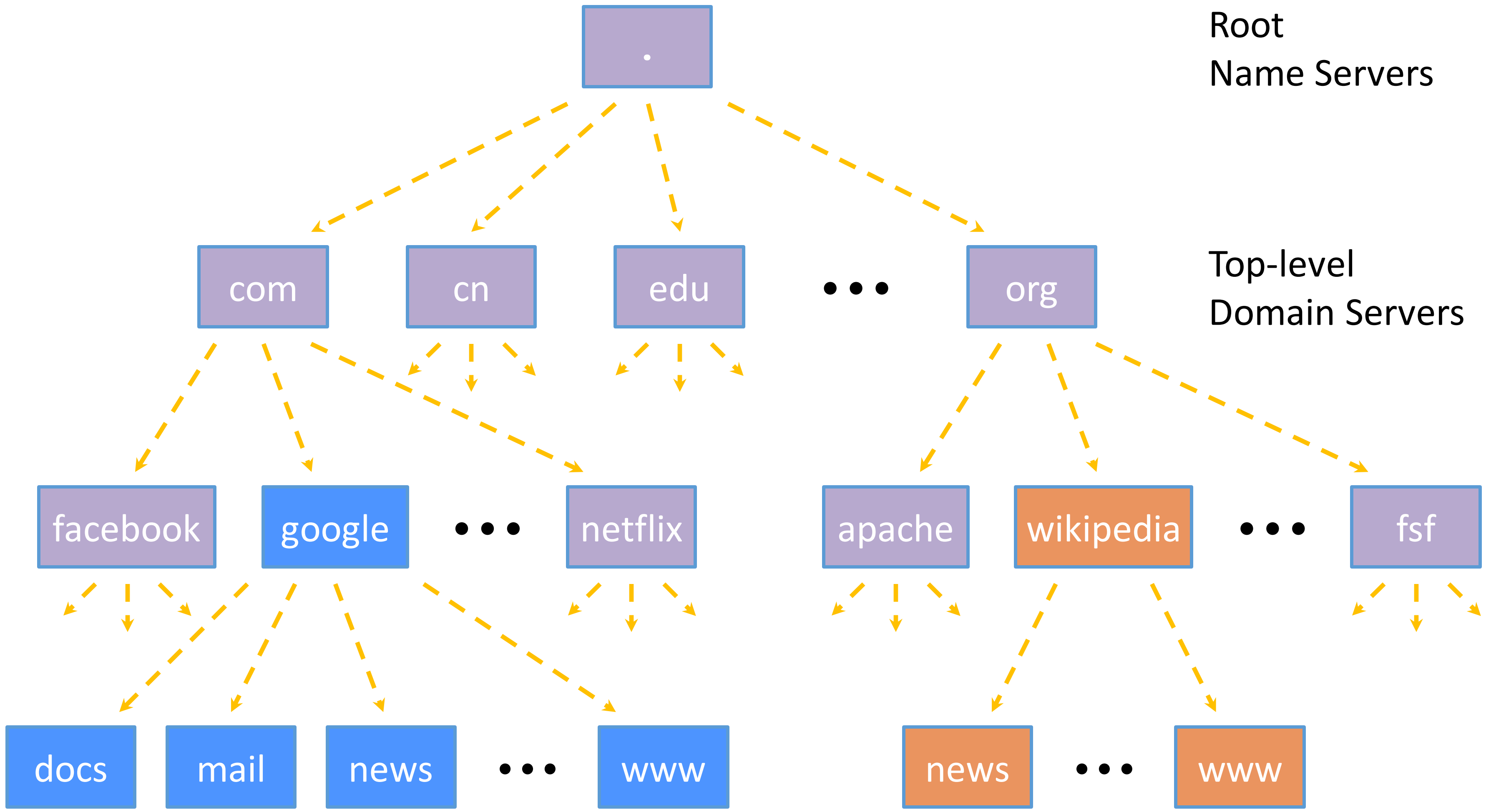


Domain Name System

- People tend to use DNS names, not IP addresses
  - The Sockets API lets you convert between the two
  - It's a complicated process, though:
    - A given DNS name can have many IP addresses
    - Many different IP addresses can map to the same DNS name
      - An IP address will reverse map into at most one DNS name
    - A DNS lookup may require interacting with many DNS servers

- You can use the Linux program "`dig`" to explore DNS
  - `dig @server name type (+short)`
    - `server`: specific name server to query
    - `type`: `A` (IPv4), `AAAA` (IPv6), `ANY` (includes all types)

DNS Hierarchy



Resolving DNS Names

- Use sys call **`getaddrinfo`**`()` `to get ip addresses for a name`

```
int getaddrinfo(const char* hostname,
                const char* service,
                const struct addrinfo* hints,
                struct addrinfo** res);
```

- Basic idea:
  - Tell **`getaddrinfo`**`()` which host and port you want resolved
    - String representation for host: DNS name or IP address
  - Set up a "`hints`" structure with constraints you want respected
  - **`getaddrinfo`**`()` gives you a list of addresses packed into an "`addrinfo`" structure/linked list
    - Returns 0 on success; returns *negative number* on failure
  - Free the `struct addrinfo` later using **`freeaddrinfo`**`()`

DNS Lookup Procedure

```
struct addrinfo {
  int      ai_flags;           // additional flags
  int      ai_family;          // AF_INET, AF_INET6, AF_UNSPEC
  int      ai_socktype;        // SOCK_STREAM, SOCK_DGRAM, 0
  int      ai_protocol;        // IPPROTO_TCP, IPPROTO_UDP, 0
  size_t   ai_addrlen;         // length of socket addr in bytes
  struct sockaddr* ai_addr;    // pointer to socket addr
  char*    ai_canonname;       // canonical name
  struct addrinfo* ai_next;    // can form a linked list
};
```

1) Create a `struct addrinfo hints`
2) Zero out `hints` for "defaults"
3) Set specific fields of `hints` as desired
4) Call **`getaddrinfo`**`()` using `&hints`
5) Resulting linked list `res` will have all fields appropriately set

- See dnsresolve.c

Socket API: Client TCP connection

- There are five steps:
  1) Figure out the IP address and port to connect to
  2) Create a socket
  3) Connect the socket to the remote server

     `read``()` and **`write`**`()` data using the socket
  5) Close the socket

Step 2: Creating a Socket

- Use the `socket()` system call

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

  - Creates an endpoint for communication
  - Creating a socket doesn’t bind it to a local address or port yet
  - Returns file descriptor or -1 on error

Step 3: Connect to a Server

- The `connect()` system call establishes a connection to a remote host

```
int connect(int sockfd, const struct sockaddr* addr,
            socklen_t addrlen);
```

  - `sockfd`: Socket file description from Step 2
  - `addr` and `addrlen`: Usually from one of the address structures returned by `getaddrinfo` in Step 1 (DNS lookup)
  - Returns `0` on success and `-1` on error
- `connect()` may take some time to return
  - It is a *blocking* call by default
  - The network stack within the OS will communicate with the remote host to establish a TCP connection to it
    - This involves ~2 *round trips* across the network

Step 4: read()

- If there is data that has already been received by the network stack, then read will return immediately with it
  - `read()` might return with *less* data than you asked for
- If there is no data waiting for you, by default `read()` will *block* until something arrives
  - Can `read()` return 0?

Step 4: write()

- `write()` queues your data in a send buffer in the OS and then returns
  - The OS transmits the data over the network in the background
  - When `write()` returns, the receiver probably has not yet received the data!

- If there is no more space left in the send buffer, by default `write()` will *block*

Step 5: close()

- Straightforward. Same function as with file I/O.
- Shuts down the socket and frees resources and file descriptors associated with it on both ends of the connection

Summary: Client TCP Connection

- There are five steps:
  1) Figure out the IP address and port to which to connect
  2) Create a socket
  3) Connect the socket to the remote server
  4) **`read`**`()` and **`write`**`()` data using the socket
  5) Close the socket

Socket API: Server TCP Connection

- Pretty similar to clients, but with additional steps:
  1) Figure out the IP address and port on which to listen
  2) Create a socket
  3) `bind()` the socket to the address(es) and port
  4) Tell the socket to `listen()` for incoming clients
  5) `accept()` a client connection

     `read()` and `write()` to that connection
  7) `close()` the client socket

Step 1: Figure out IP address(es) & Port

- **`getaddrinfo`**`()` invocation may or may not be needed (but we'll use it)
  - Do you know your IP address(es) already?
    - Static vs. dynamic IP address allocation
    - Even if the machine has a static IP address, don't wire it into the code – either look it up dynamically or use a configuration file
  - Can request listen on all local IP addresses by passing `NULL` as `hostname` and setting `AI_PASSIVE` in `hints.ai_flags`

Step 2: Create a Socket

- **`socket`**`()` call is same as before

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

  - Can directly use constants or fields from result of **`getaddrinfo`**`()`
  - Recall that this just returns a file descriptor – IP address and port are not associated with socket yet

Step 3: Bind the socket

- ```
  int bind(int sockfd, const struct sockaddr* addr,
               socklen_t addrlen);
  ```

  - Looks nearly identical to `connect()`!
  - Returns `0` on success, `-1` on error
  ```

Step 4: Listen for Incoming Clients

- `int listen(int sockfd, int backlog);`
  - Tells the OS that the socket is a listening socket that clients can connect to
  - `backlog`: maximum length of connection queue
    - Gets truncated, if necessary, to defined constant `SOMAXCONN`
    - The OS will refuse new connections once queue is full until server `accept()`s them (removing them from the queue)
  - Returns `0` on success, `-1` on error
  - Clients can start connecting to the socket as soon as `listen()` returns
    - Server can't use a connection until you `accept()` it

Step 5: Accept a Client Connection

- ```
  int accept(int sockfd, struct sockaddr* addr,
             socklen_t* addrlen);
  ```
  - Returns an active, ready-to-use socket file descriptor connected to a client (or `-1` on error)
    - `sockfd` must have been created, bound, *and* listening
    - Pulls a queued connection or waits for an incoming one
  - `addr` and `addrlen` are output parameters
    - `*addrlen` should initially be set to `sizeof(*addr)`, gets overwritten with the size of the client address
    - Address information of client is written into `*addr`
      - Use **`inet_ntop`**`()` to get the client's printable IP address
      - Use **`getnameinfo`**`()` to do a *reverse DNS lookup* on the client
  ```

Example

- See server.c
  - Takes in a port number from the command line
  - Opens a server socket, prints info, then listens for connections
    - *Can connect to it using netcat (`nc`)*
  - Accepts connections as they come
  - Echoes any data the client sends to it on `stdout` and also sends it back to the client