CSE 333 Section 8

Client-Side Networking

Logistics

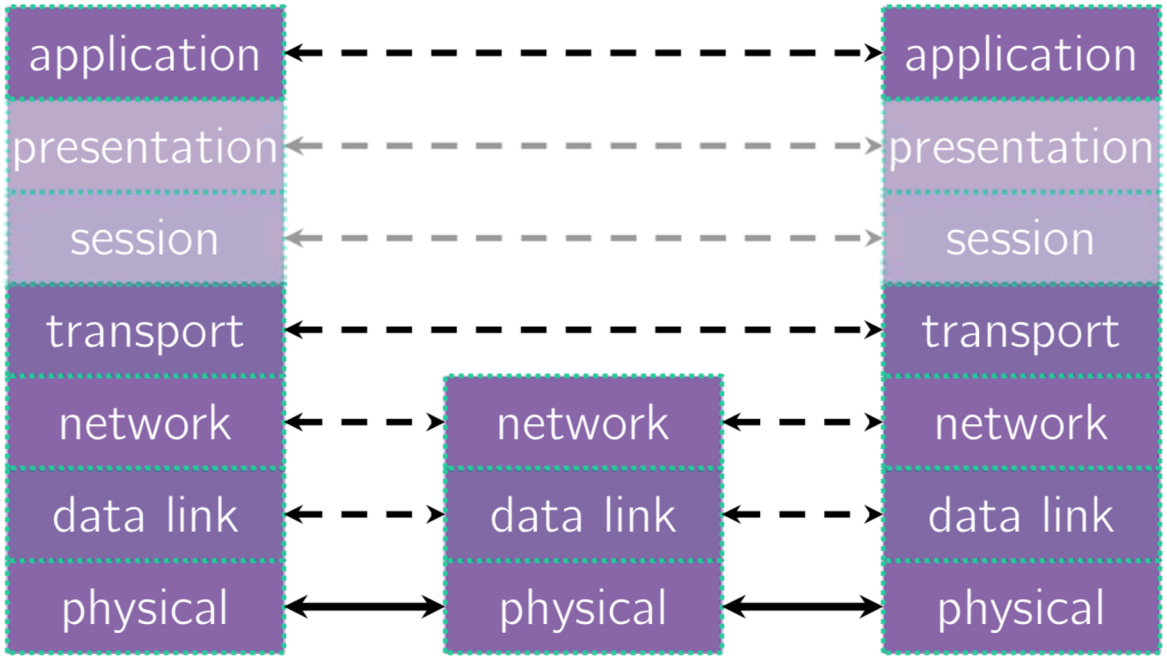
Monday:

Exercise 15 @ 10 am

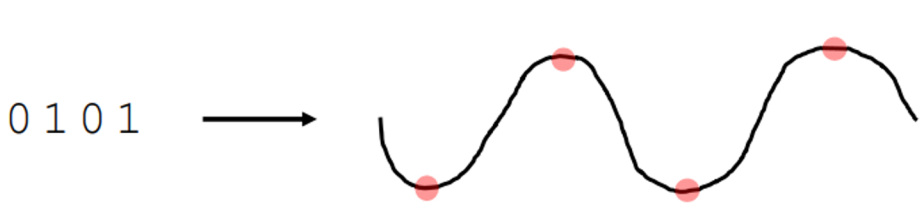
Wednesday:

Exercise 16 @ 10 am

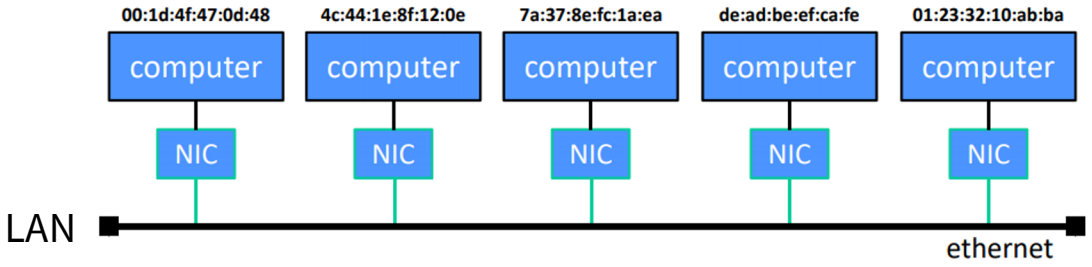
Computer Networks: A 7-ish Layer Cake



Computer Networks: A 7-ish Layer Cake

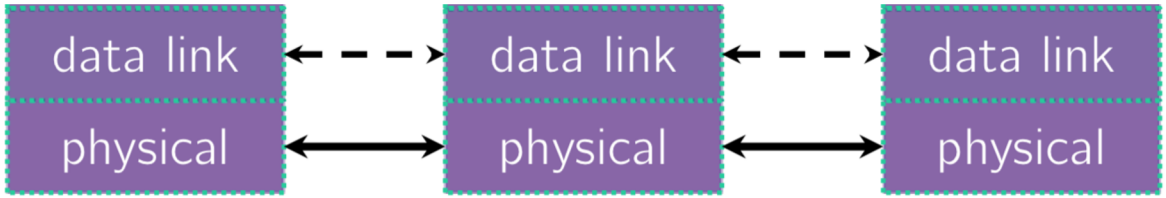


Computer Networks: A 7-ish Layer Cake

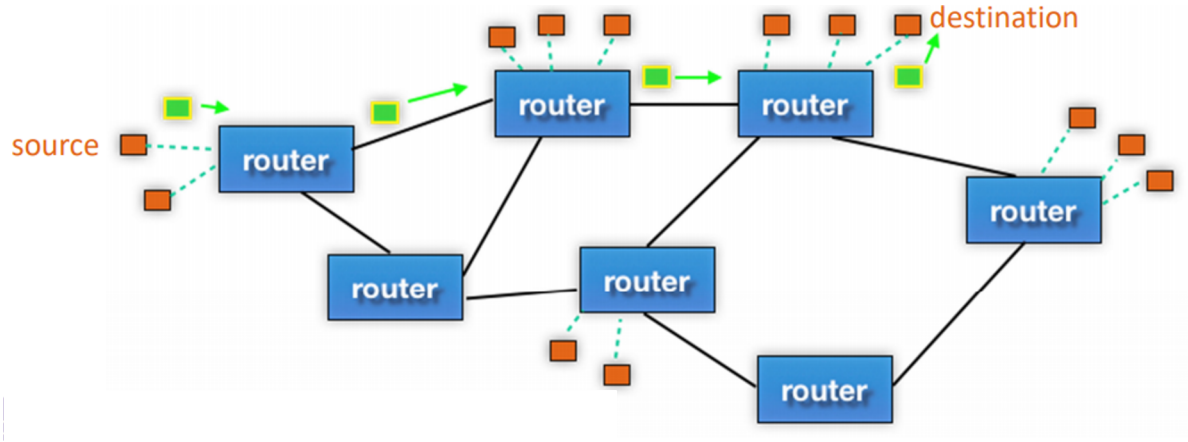


multiple computers on a local network

bit encoding at signal level



Computer Networks: A 7-ish Layer Cake

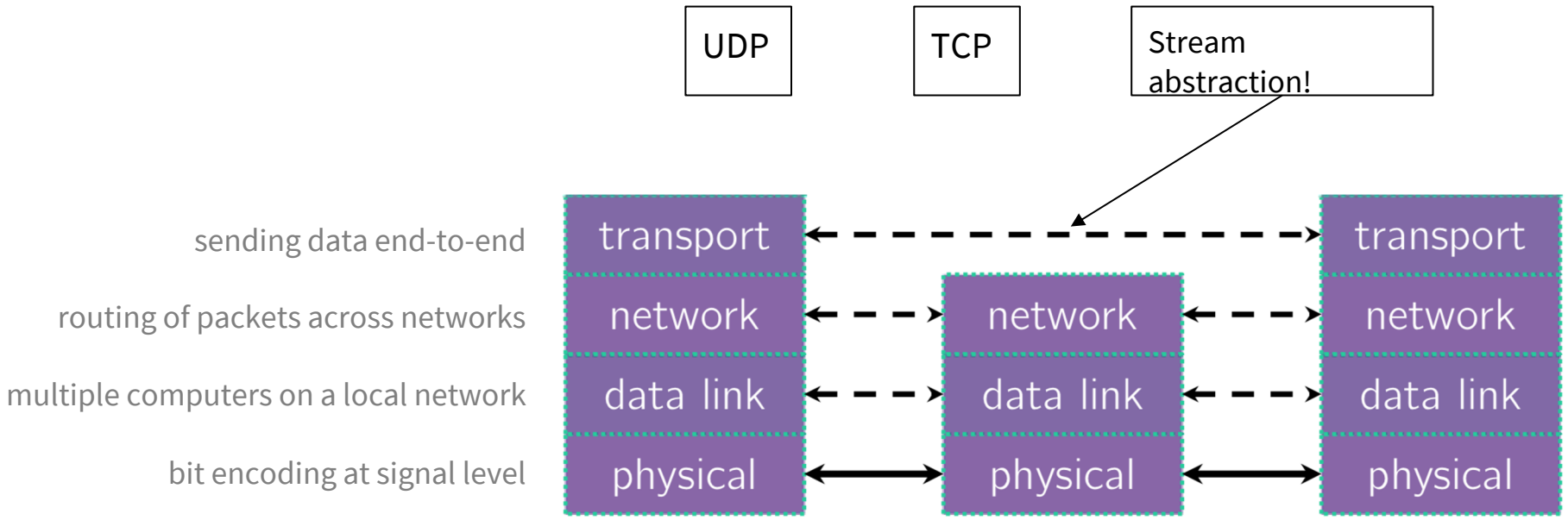


routing of packets across networks

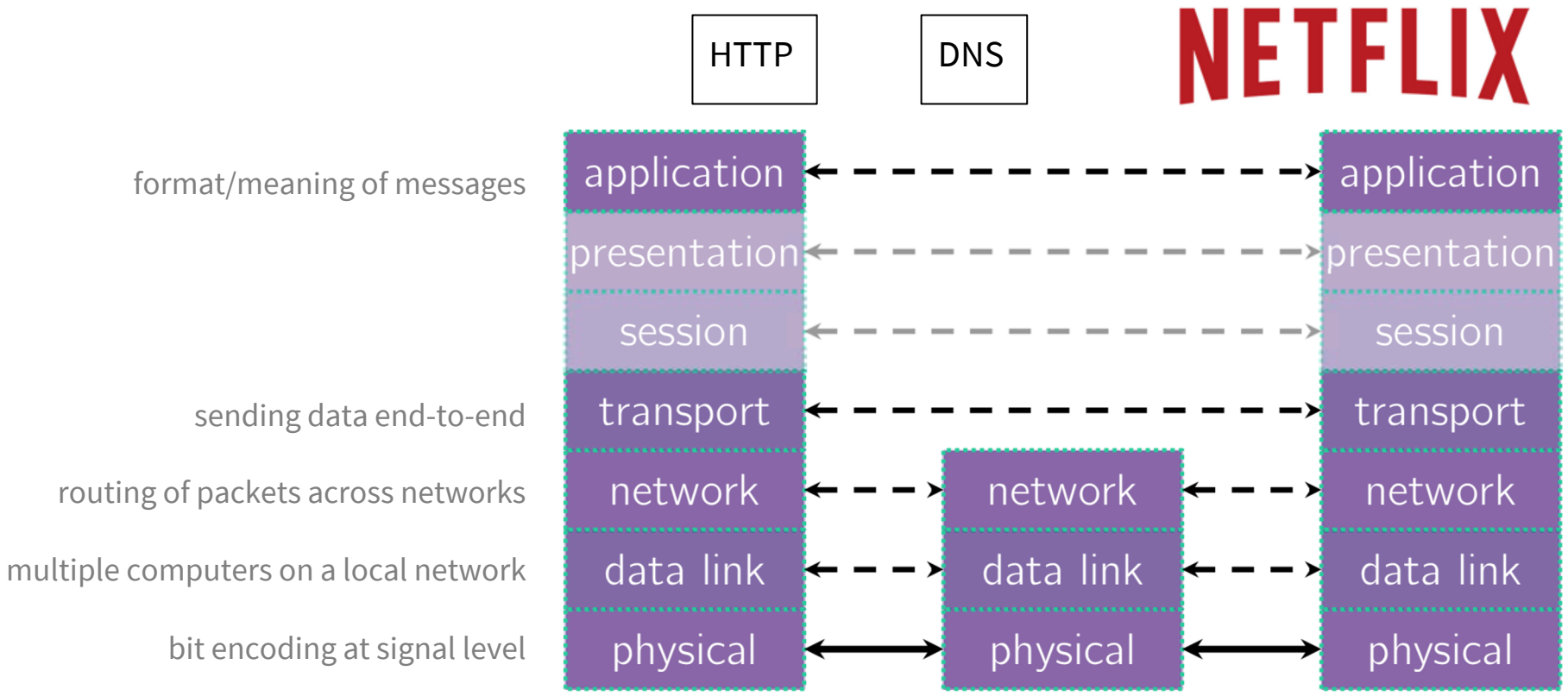
multiple computers on a local network

bit encoding at signal level

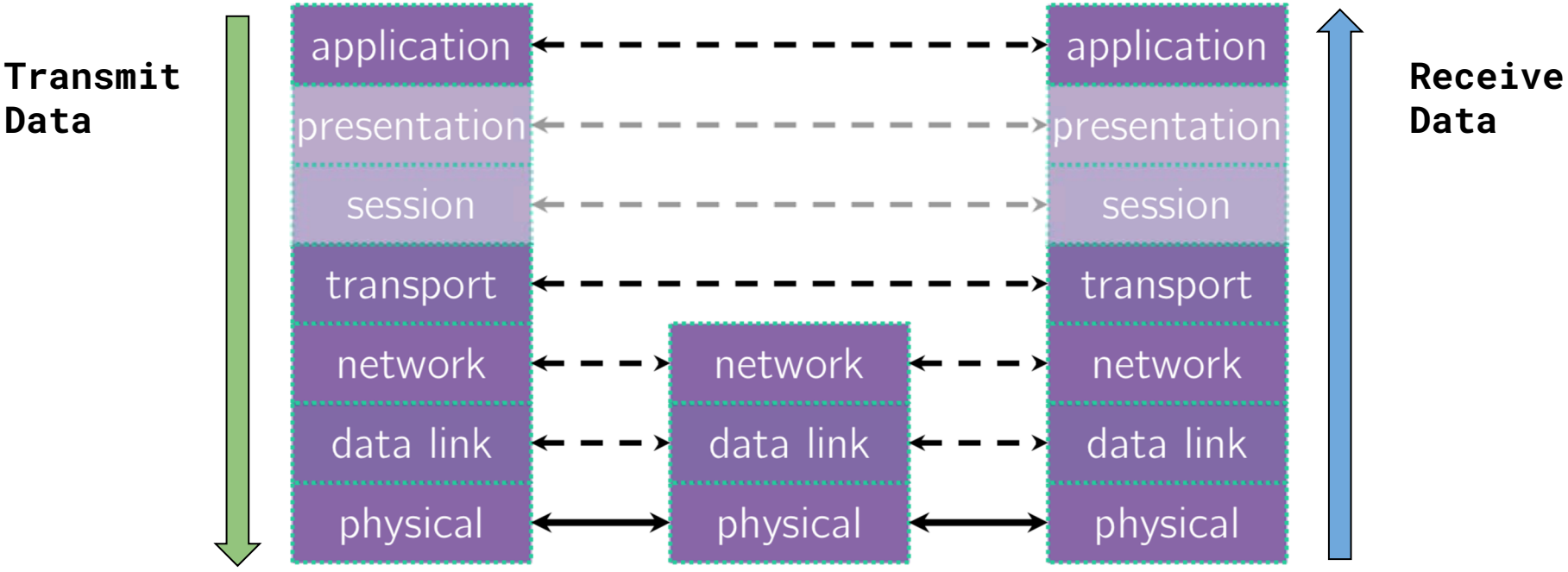
Computer Networks: A 7-ish Layer Cake



Computer Networks: A 7-ish Layer Cake



Data flow


Transmit Data
application
presentation
session
transport
network
data link
physical
network
data link
physical
application
presentation
session
transport
network
data link
physical
Receive Data

Exercise 1

Exercise 1

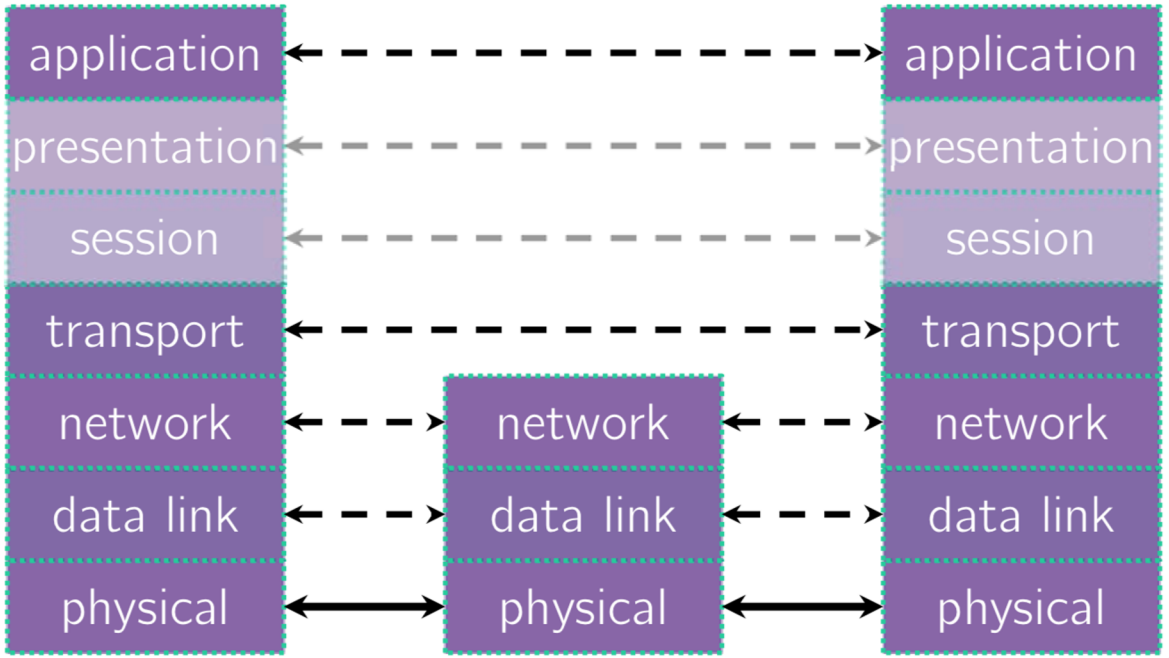
format/meaning of messages

sending data end-to-end

routing of packets across networks

multiple computers on a local network

bit encoding at signal level

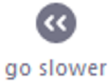


Exercise 1

Vote in Zoom!

= application layer

= transport layer



= network layer

= link layer

- DNS: Translating between IP addresses and host names. (Application Layer)
- IP: Routing packets across the Internet. (Network Layer)
- TCP: Reliable, stream-based networking on top of IP. (Transport Layer)
- UDP: Unreliable, packet-based networking on top of IP. (Transport Layer)
- HTTP: Sending websites and data over the Internet. (Application Layer)

TCP versus UDP

Transmission Control Protocol(TCP)

- Connection oriented Service
- Reliable and Ordered
- Flow control

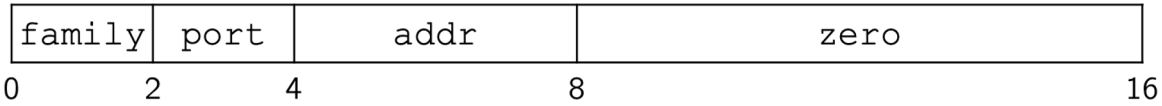
User Datagram Protocol(UDP)

- Connectionless service
- Unreliable packet delivery
- Faster
- No feedback

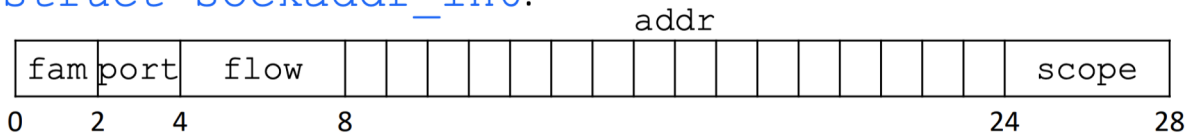
Sockets

- Just a file descriptor for network communication
- Types of Sockets
  - Stream sockets (TCP)
  - Datagram sockets (UDP)
- Each socket is associated with **a port number** and **an IP address**
  - Both port and address are stored in network byte order (big endian)

`struct sockaddr_in`:

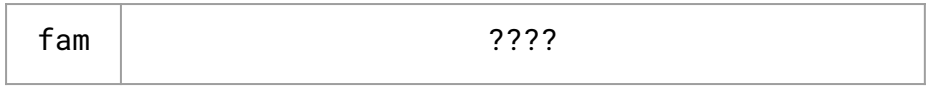


`struct sockaddr_in6`:



Sockets

struct sockaddr (pointer to this struct is used as parameter type in system calls)



....

struct sockaddr_in (IPv4)

16

struct sockaddr_in6 (IPv6)

28

struct sockaddr_storage

15

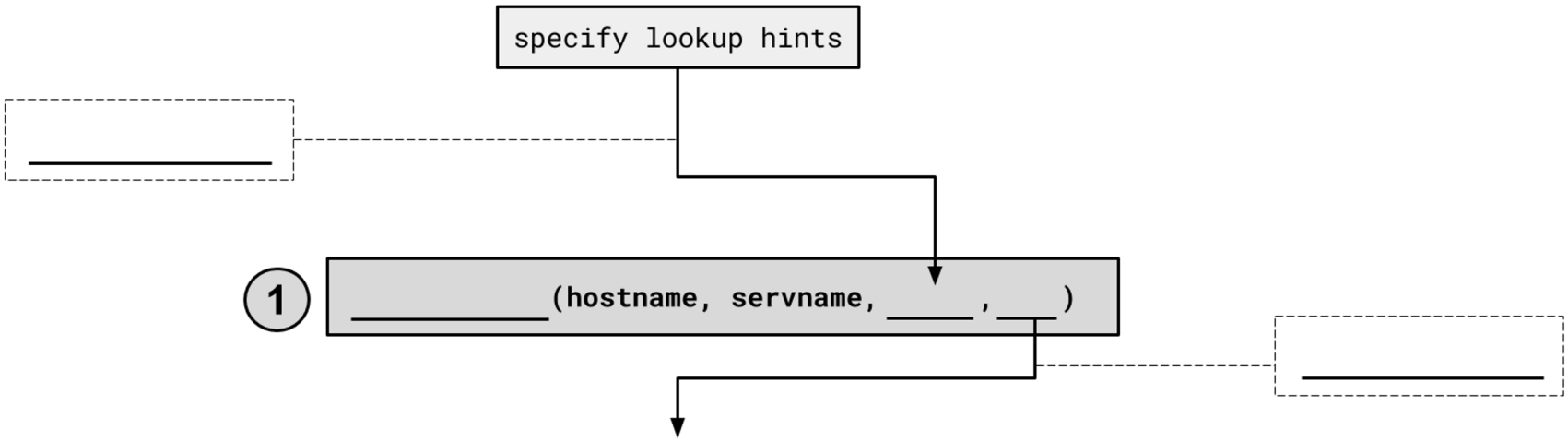
Big enough to hold either

Byte Ordering and Endianness

- **N**etwork Byte Order (Big Endian)
  - The most significant byte is stored in the highest address
- **H**ost byte order
  - Might be big or little endian, depending on the hardware
- To convert between orderings, we can use
  - `uint16_t htons (uint16_t hostlong);`
  - `uint16_t ntohs (uint16_t hostlong);`
  - `uint32_t htonl (uint32_t hostlong);`
  - `uint32_t ntohl (uint32_t hostlong);`

Exercise 2

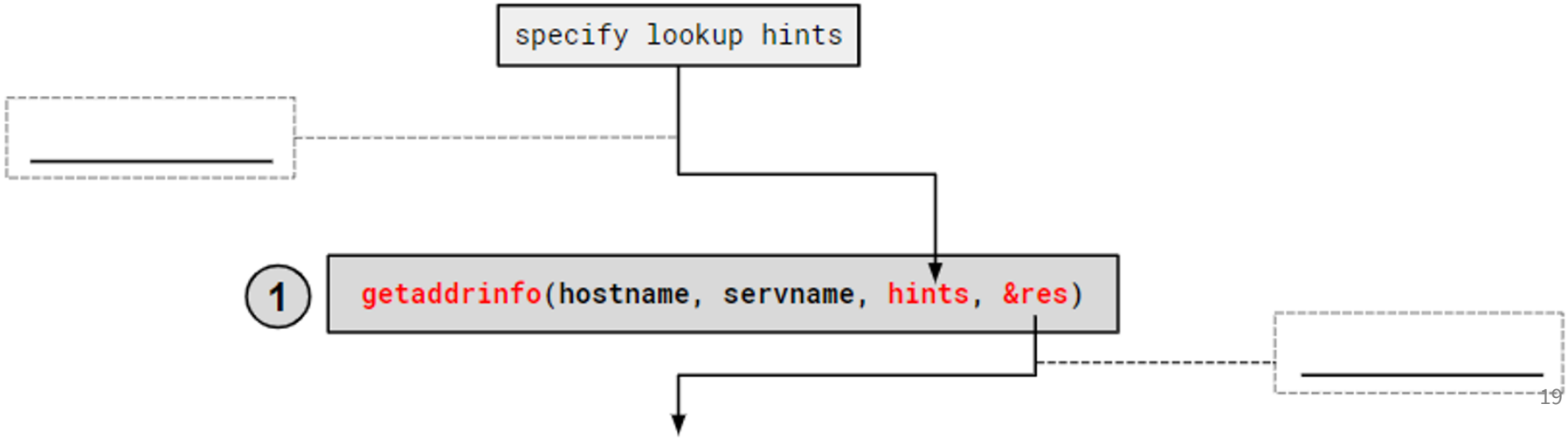
1.



1. getaddrinfo()

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

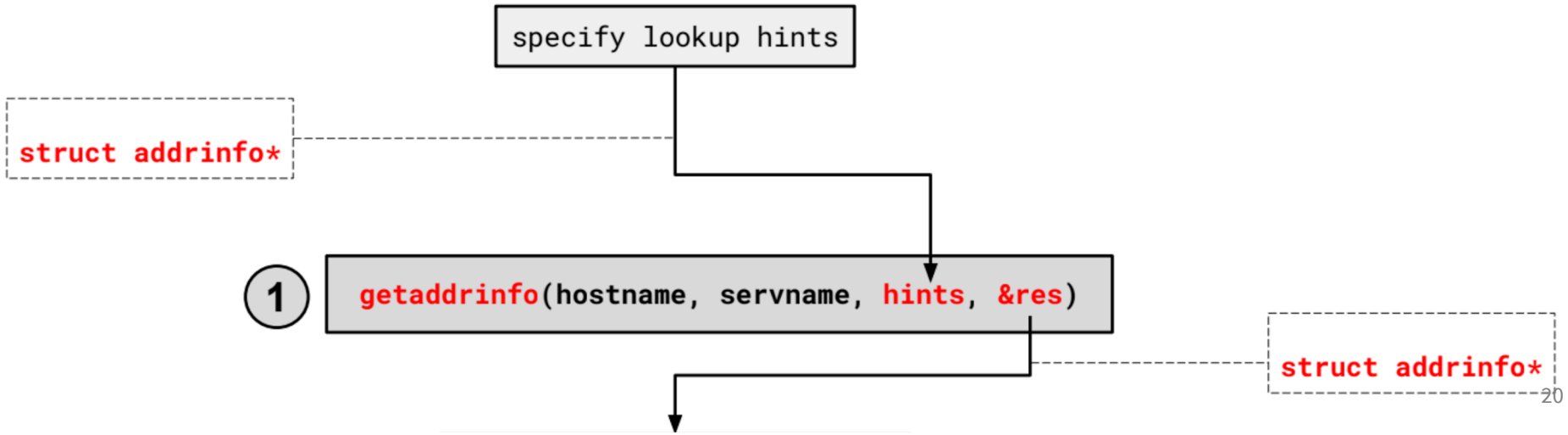
- Performs a **DNS Lookup** for a hostname



1. getaddrinfo()

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

- Performs a **DNS Lookup** for a hostname
- Use “hints” to specify constraints (`struct addrinfo *`)
- Get back a linked list of `struct addrinfo` results



Network Addresses

- For IPv4, an IP address is a 4-byte tuple
  - e.g., 128.95.4.1 (80:5f:04:01 in hex)

- For IPv6, an IP address is a 16-byte tuple
  - e.g., 2d01:0db8:f188:0000:0000:0000:0000:1f33
  - 2d01:0db8:f188::1f33 in shorthand

DNS – Domain Name System/Service

- A hierarchical distributed naming system any resource connected to the Internet or a private network.
- Resolves queries for names into IP addresses.
- The sockets API lets you convert between the two.
  - Aside: getnameinfo() is the inverse of getaddrinfo()
- Is on the application layer on the Internet protocol suite.
- POSIX form of resolving DNS names is **getaddrinfo()**
  - dig +trace attu.cs.washington.edu shown later

1. getaddrinfo() - Interpreting Results

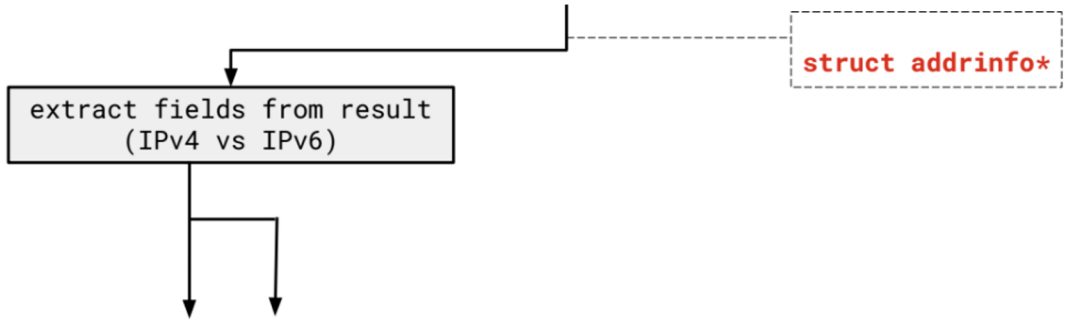
```
struct addrinfo {
    int ai_flags; // additional flags
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype; // SOCK_STREAM, SOCK_DGRAM, 0
    int ai_protocol; // IPPROTO_TCP, IPPROTO_UDP, 0
    size_t ai_addrlen; // length of socket addr in bytes
    struct sockaddr* ai_addr;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

- `ai_addr` points to a `struct sockaddr` describing the socket address

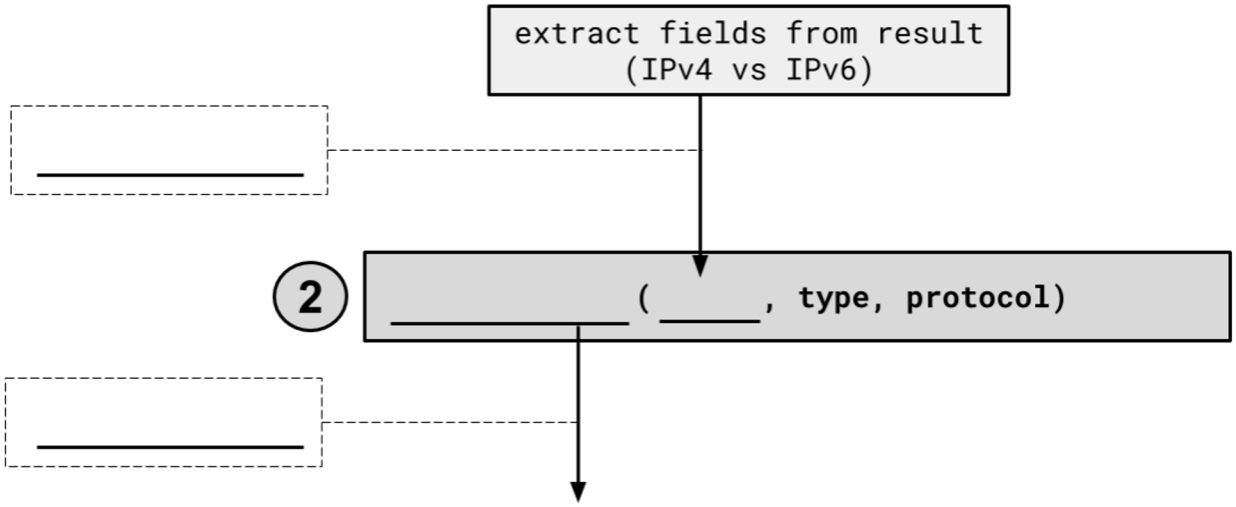
1. getaddrinfo() - Interpreting Results

With a `struct sockaddr*`:

- The field `sa_family` describes if it is IPv4 or IPv6
- Cast to `struct sockaddr_in* (v4)` or `struct sockaddr_in6* (v6)` to access/modify specific fields
- Store results in a `struct sockaddr_storage` to have a space big enough for either



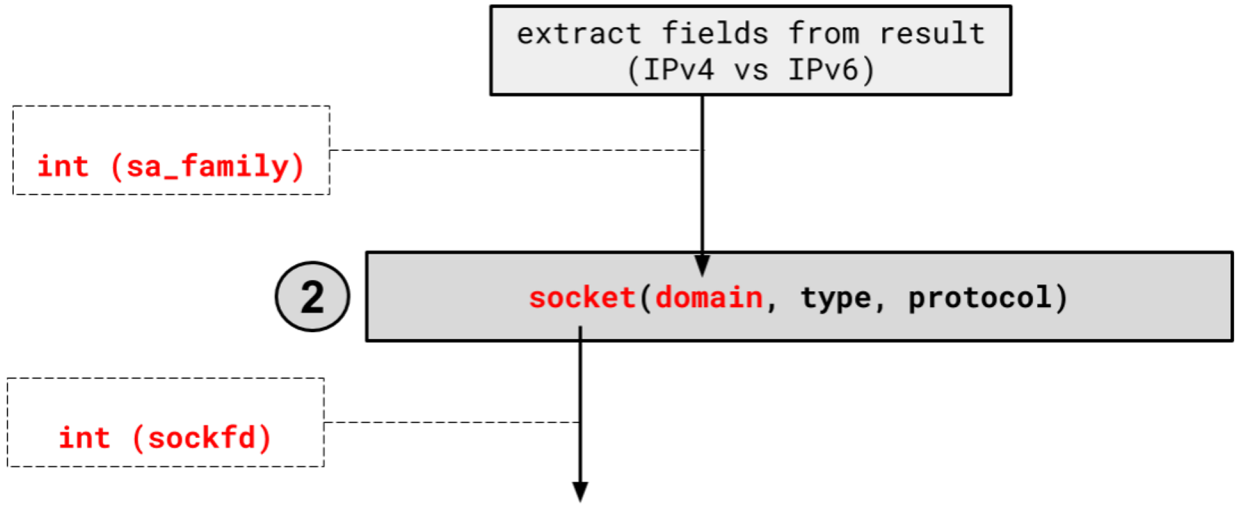
2.



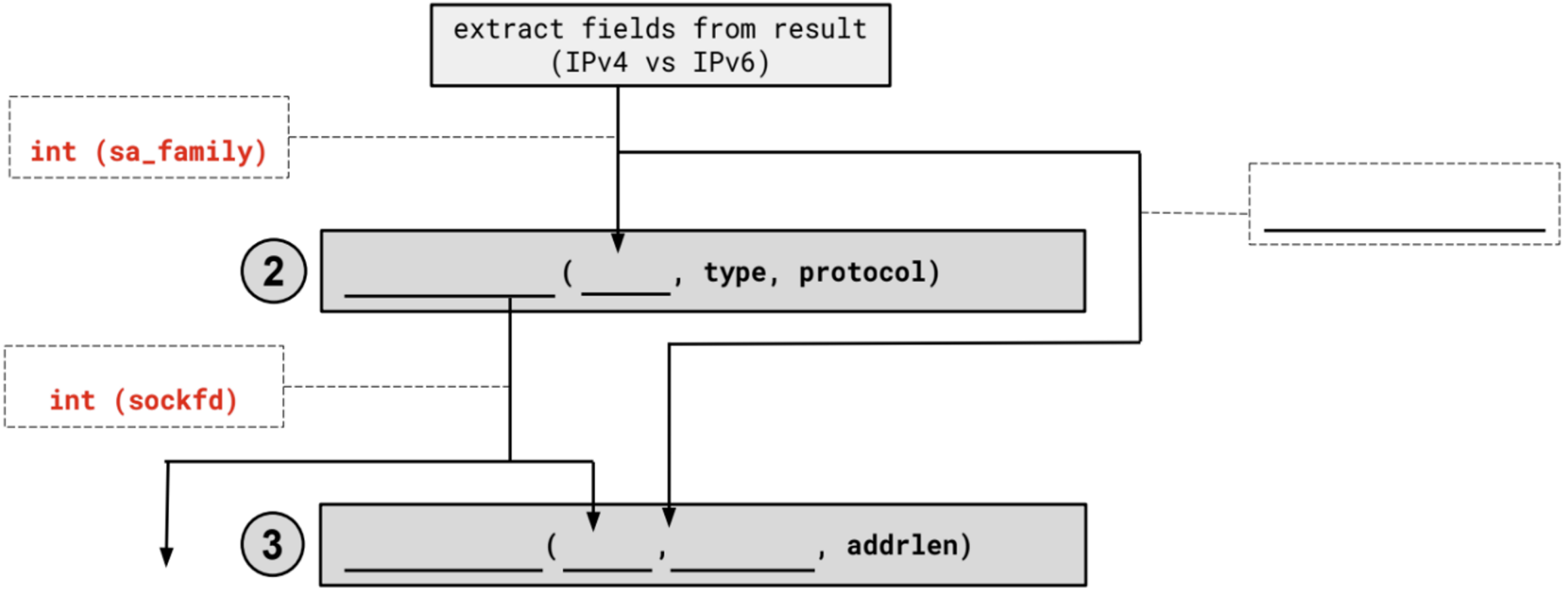
2. socket()

```
int socket(int domain,     // AF_INET, AF_INET6
           int type,       // SOCK_STREAM (TCP)
           int protocol); // 0
```

- Creates a “raw” socket, ready to be bound
- Returns file descriptor (`sockfd`) on success, `-1` on failure



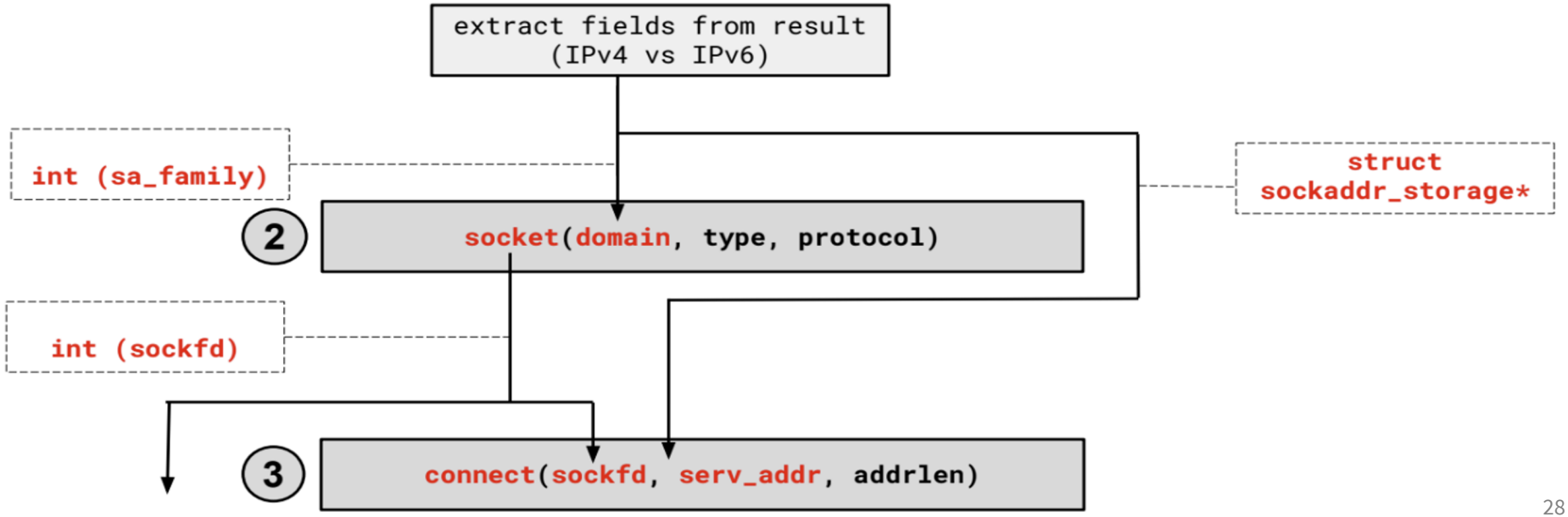
3.


extract fields from result
(IPv4 vs IPv6)
int (sa_family)
2
_____(_____, type, protocol)
int (sockfd)
3
_____(_____, _____, addrlen)

3. connect()

```
int connect  (int sockfd,                // from 2
              const struct sockaddr *serv_addr, // from 1
              socklen_t addrlen);        // size of serv_addr
```

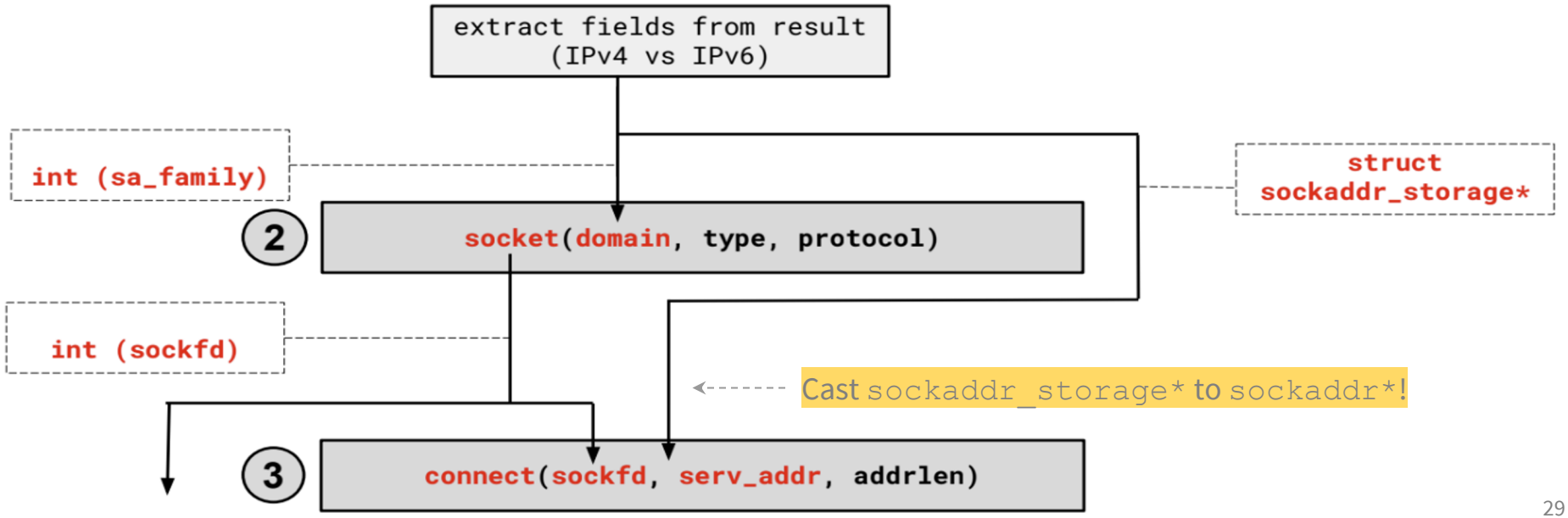
- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure



3. connect()

```
int connect  (int sockfd,                // from 2
              const struct sockaddr *serv_addr, // from 1
              socklen_t addrlen);        // size of serv_addr
```

- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure



4. read/write and 5. close

- Thanks to the file descriptor abstraction, use as normal!
- `read` from and `write` to a buffer, the OS will take care of sending/receiving data across the network
- Make sure to `close` the fd afterward