CSE 333
Section 8

Client-side Networking

Netcat is listening on port 80

Infosec memes for pentesting teams

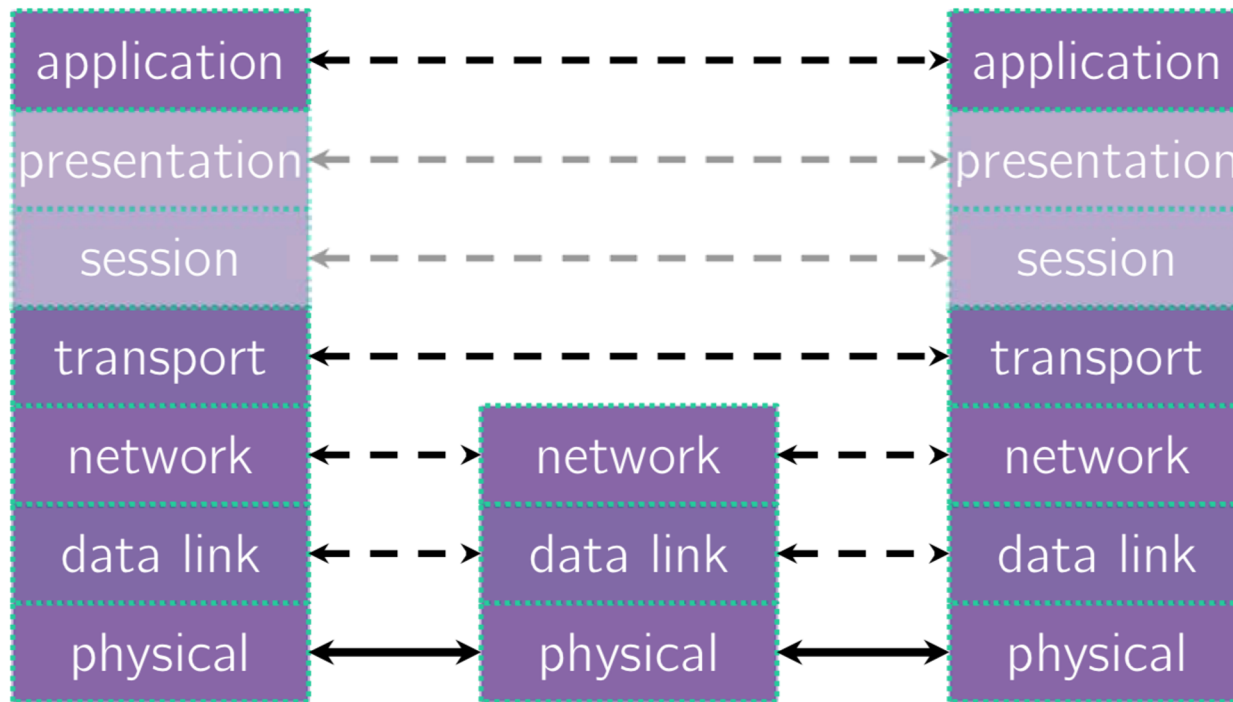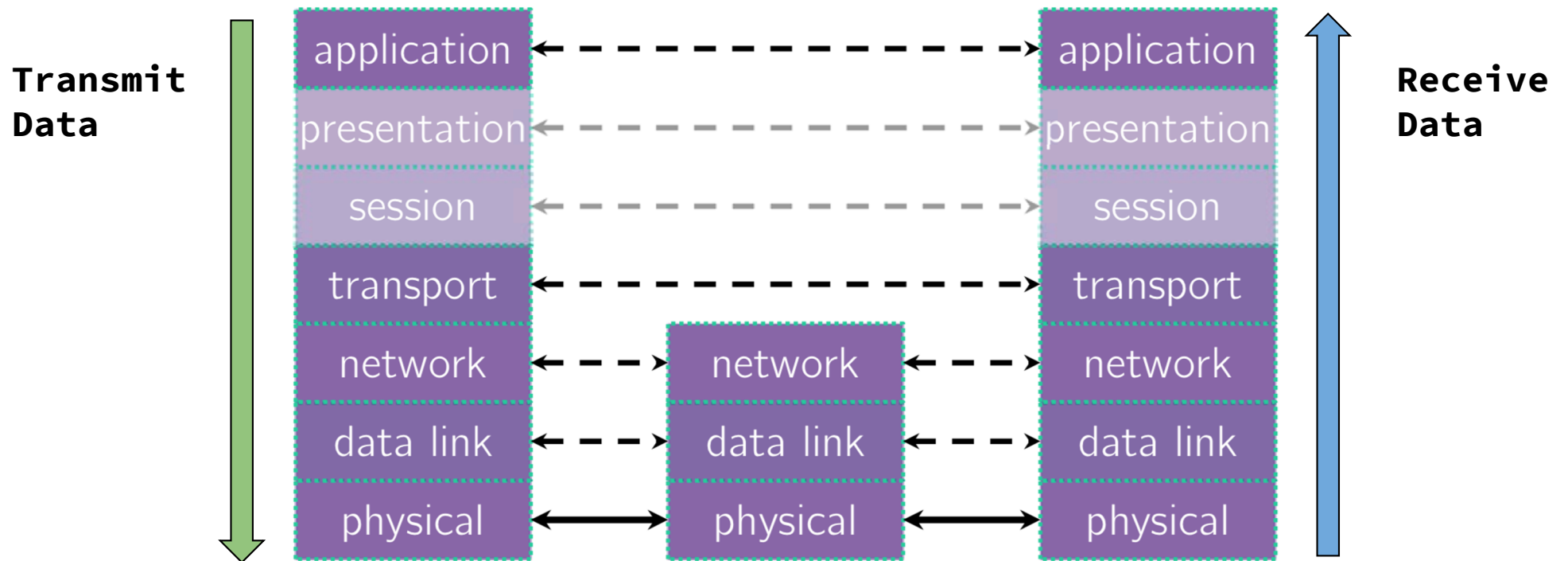W UNIVERSITY of WASHINGTON

Logistics

- Exercise 15:
  - Out later today
  - Due **Monday** (11/21) @ 10:00am

- Exercise 16:
  - Out after lecture tomorrow
  - Due **Wednesday** (11/23) @ 10:00am

- Homework 3:
  - Due **Tonight** (1/17) @ 11:00pm

Computer Networking Review

Computer Networks: A 7-ish Layer Cake

Data Flow

Transmit Data

Receive Data

application	application
presentation	presentation
session	session
transport	transport
network	network	network
data link	data link	data link
physical	physical	physical

Computer Networks: A 7-ish Layer Cake

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

application		application
presentation		presentation
session		session
transport		transport
network	network	network
data link	data link	data link
physical	physical	physical

12

Exercise 1

Terminology Review

- DNS: Translating between IP addresses and host names. (Application Layer)

- IP: Routing packets across the Internet. (Network Layer)

- TCP: Reliable transport protocol on top of IP. (Transport Layer)

- UDP: Unreliable transport protocol on top of IP. (Transport 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
- High speed, no feedback

TCP guarantees reliability for things like messaging or data transfers. UDP has less overhead since it doesn't make those guarantees, but is often fine for streaming applications (e.g., YouTube or Netflix) or other applications that manage packets on their own or do not want occasional pauses for packet retransmission or recovery.

Client-Side Networking

Client-Side Networking in 5 Easy* Steps!

1. Figure out what IP address and port to talk to
2. Build a socket from the client
3. Connect to the server using the client socket and server socket
4. Read and/or write using the socket
5. Close the socket connection

*difficulty is subjective

Step 1: Figuring out the port and IP

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

- Use "hints" to specify constraints (`struct addrinfo*`)

- Get back a linked list of `struct addrinfo` results

Name of host whose IP we want

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

We will set this to `nullptr`
to get the default; otherwise
you can specify service/port

Output parameter; `*res` **is**
set to the first result in LL

Hints for the lookup server/refine results

18

Step 1: Obtaining your server's socket address

```c
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 have linked list of records
}
```

- ai_addr points to a `struct sockaddr` describing a socket address, can be IPv4 or IPv6

Understanding `struct sockaddr*`

- It's just a pointer. To use it, we're going to have to dereference it and cast it to the right type (Very strange C "inheritance")
  - It is the endpoint your connection refers to

- Convert to a `struct sockaddr_storage`
  - Read the `sa_family` to determine whether it is IPv4 or IPv6
  - IPv4: AF_INET (macro) → cast to `struct sockaddr_in`
  - IPv6: AF_INET6 (macro) → cast to `struct sockaddr_in6`

Understanding Socket Addresses

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

fam	????

....

struct sockaddr_in (IPv4)

fam	port	addr	zero

16

struct sockaddr_in6 (IPv6)

fam	port	flow	addr	scope

28

struct sockaddr_storage

fam	

Big enough to hold either

21

Sockets (Berkeley Sockets)

- Defines a local endpoint communication between server and client
  - Similar to a file descriptor for network communication
  - Built on various operating system calls

- Each socket is associated with **a port number (`uint16_t`)** and **an IP address**
  - Both port and address are stored in network byte order (big endian)
  - `ai_family` will help you to determine what is stored for your socket!

Steps 2 and 3: Building a Connection

2. Create a client socket to manage (returns an integer file descriptor, just like POSIX open)

```
// returns file descriptor on success, -1 on failure (errno set)
int socket(int domain,          // AF_INET, AF_INET6, etc.
           int type,            // SOCK_STREAM, SOCK_DGRAM, etc.
           int protocol);       // just put 0 (network abstraction)
```

3. Use that created client socket to connect to the server socket

```
// Connects to the server
// returns 0 on success, -1 on failure (errno set)
int connect(int sockfd,                   // socket file descriptor
            struct sockaddr *serv_addr,   // socket addr of server
            socklen_t addrlen);           // size of serv_addr
```

Usually from getaddrinfo!

23

Steps 4 and 5: Using your Connection

```c
// returns amount read, 0 for EOF, -1 on failure (errno set)
ssize_t read(int fd, void *buf, size_t count);

// returns amount written, -1 on failure (errno set)
ssize_t write(int fd, void *buf, size_t count);

// returns 0 for success, -1 on failure (errno set)
int close(int fd);
```
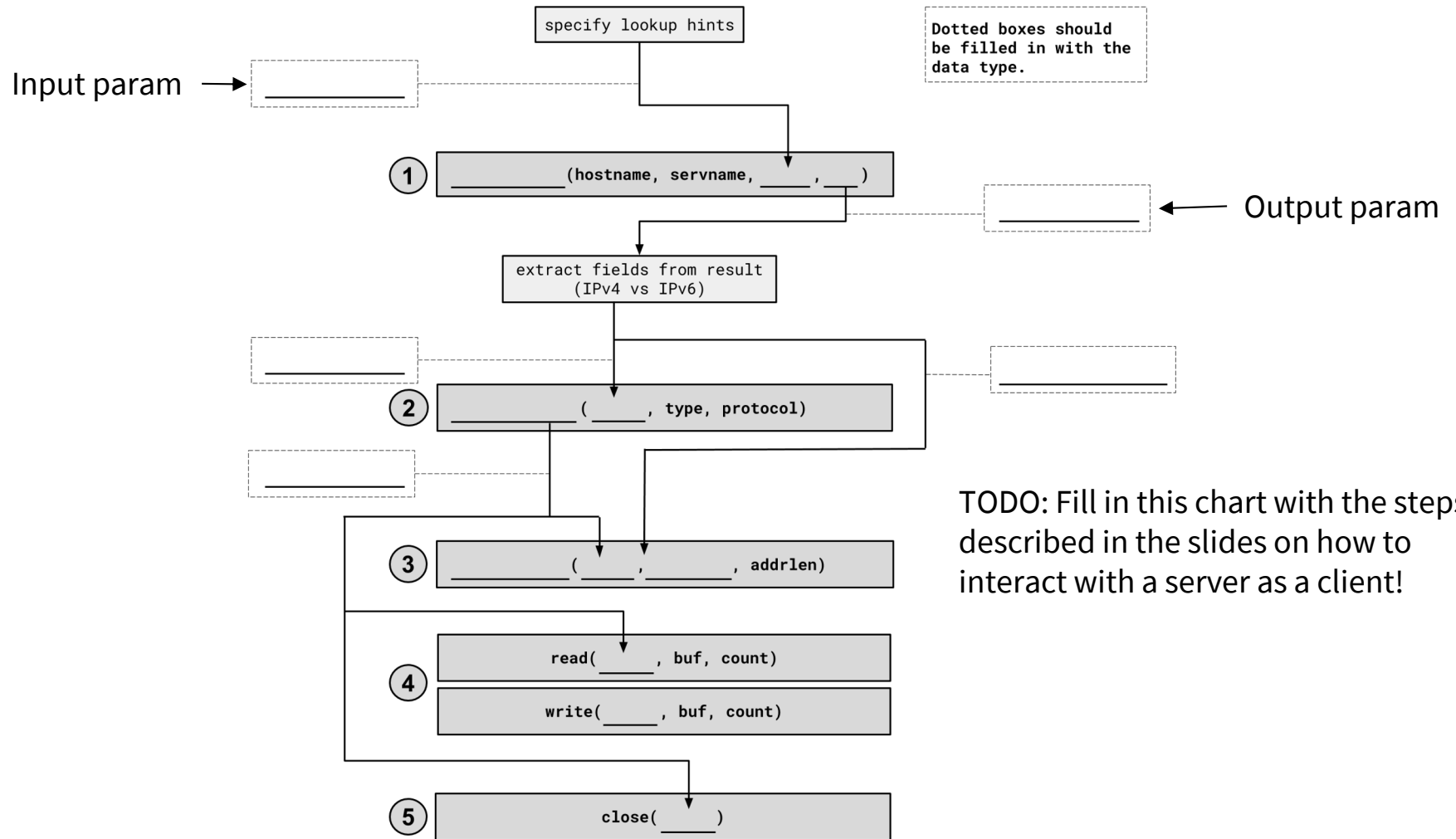
- Same POSIX methods we used for file I/O!
     (so they require the same error checking...)

Helpful References

1. Figure out what IP address and port to talk to
   - dnsresolve.cc
2. Build a socket from the client
   - connect.cc
3. Connect to the server using the client socket and server socket
   - sendreceive.cc
4. Read and/or write using the socket
   - sendreceive.cc
5. Close the socket connection

*Note: sendreceive.cc uses connect.cc's function LookUpName() which hides implementation of getaddrinfo(). They are under the same directory and can be found here. The following links are implementations of each step and can be helpful to read side-by-side with the slides. If there are any confusions about a function feel free to make a post on edStem or look at the man page

Exercise 2

specify lookup hints

Dotted boxes should be filled in with the data type.

Input param ⟶ [_____]

① [_____(hostname, servname, _____, ___)]

[_____] ⟵ Output param

extract fields from result
(IPv4 vs IPv6)

[_____]

② [_____ (_____, type, protocol)]

[_____]

[_____]

③ [_____ (_____, _____, addrlen)]

TODO: Fill in this chart with the steps described in the slides on how to interact with a server as a client!

④ read(_____, buf, count)

write(_____, buf, count)

⑤ close(_____)

28

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
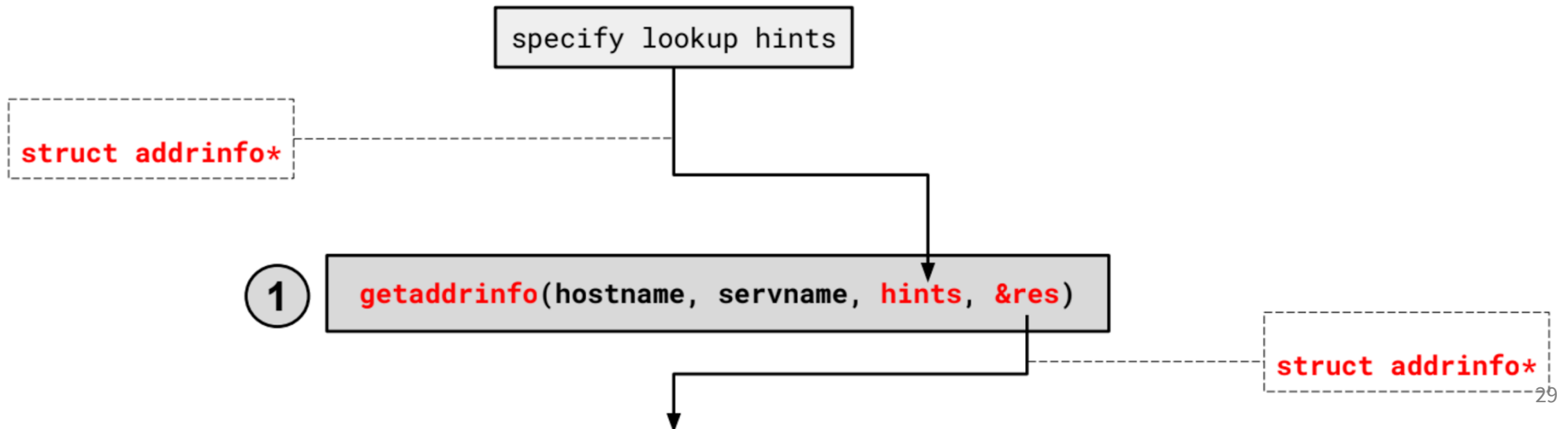
1. getaddrinfo() - Interpreting Results

```c
struct addrinfo {
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC
    struct sockaddr* ai_addr; // pointer to socket addr
    ...
};
```

- These records are dynamically allocated; you should pass the head of the linked list to `freeaddrinfo()`

- The field `ai_family` describes if it is IPv4 or IPv6

- `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
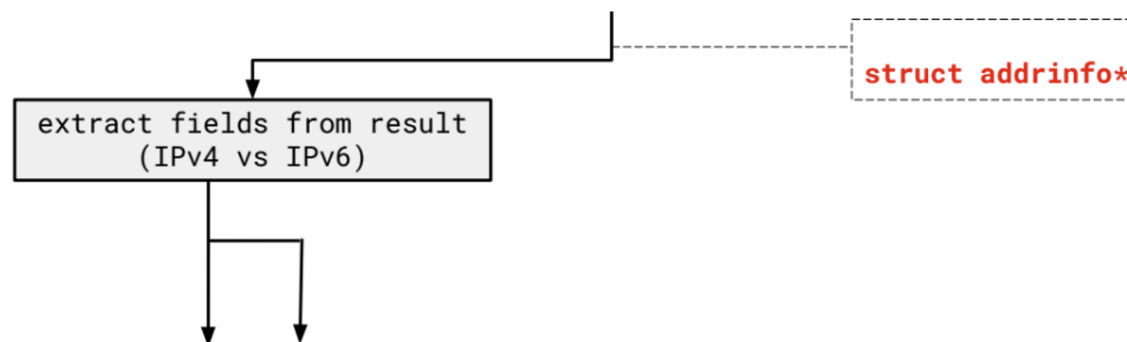
2. Build client side socket
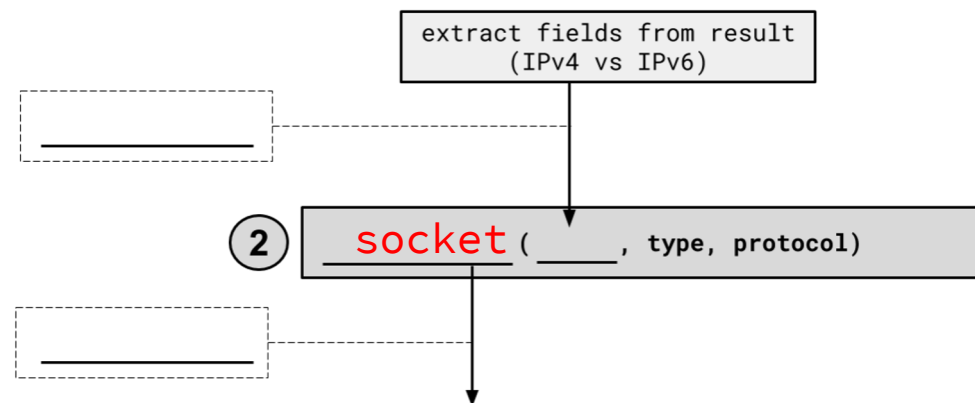
```
int socket(int domain,      // AF_INET, AF_INET6
           int type,        // SOCK_STREAM (for TCP)
           int protocol);   // 0 for the default
```

- This gives us an unbound socket that's not connected to anywhere in particular
- Returns a socket file descriptor (we can use it everywhere we can use any other file descriptor as well as in socket specific system calls)

2. Build client side socket
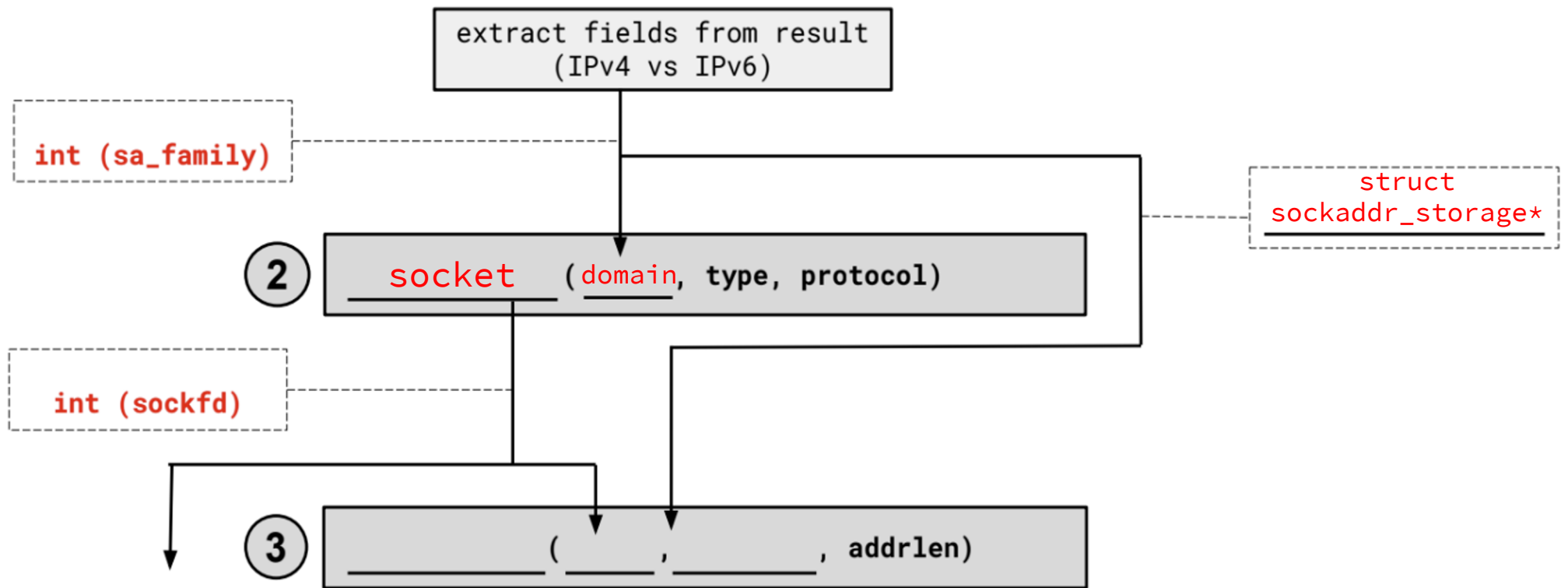
3. connect()

```
int connect(int socket,                    // socket fd
            const struct sockaddr *addr,   // address to connect to
            socklen_t addr_len);           // length of *addr
```
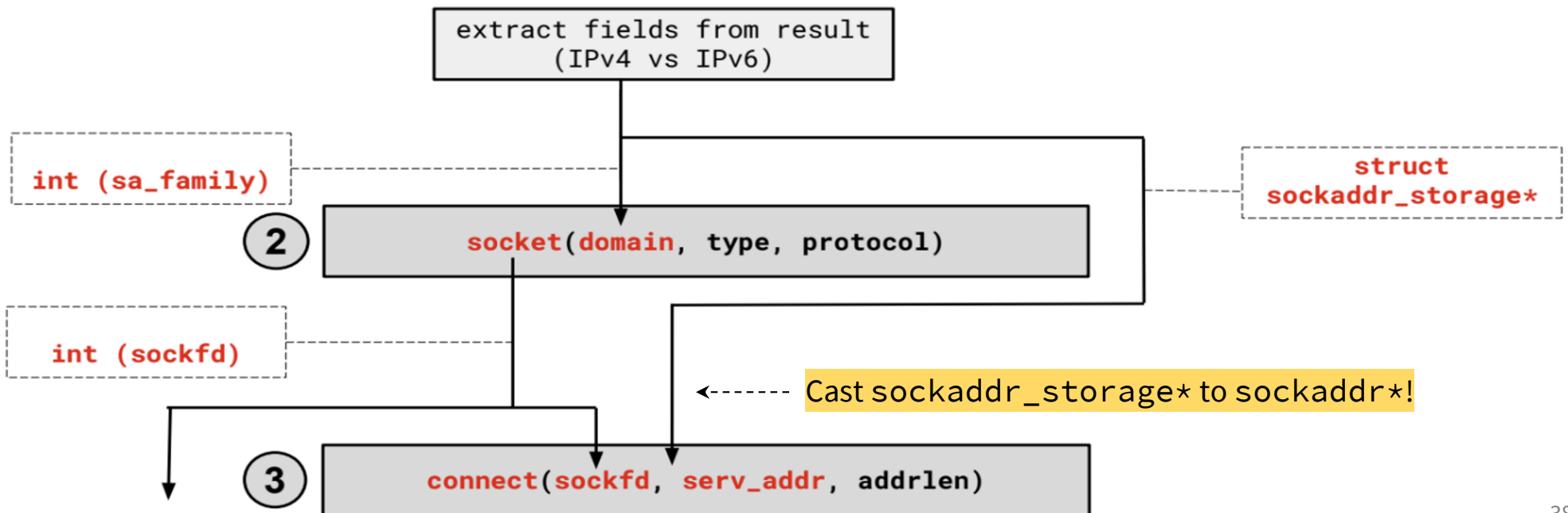
- This takes our unbound socket and connects it to the host at `addr`
- Returns 0 on success, -1 on error with `errno` set appropriately
- After this call completes, we can actually use our socket for communication!

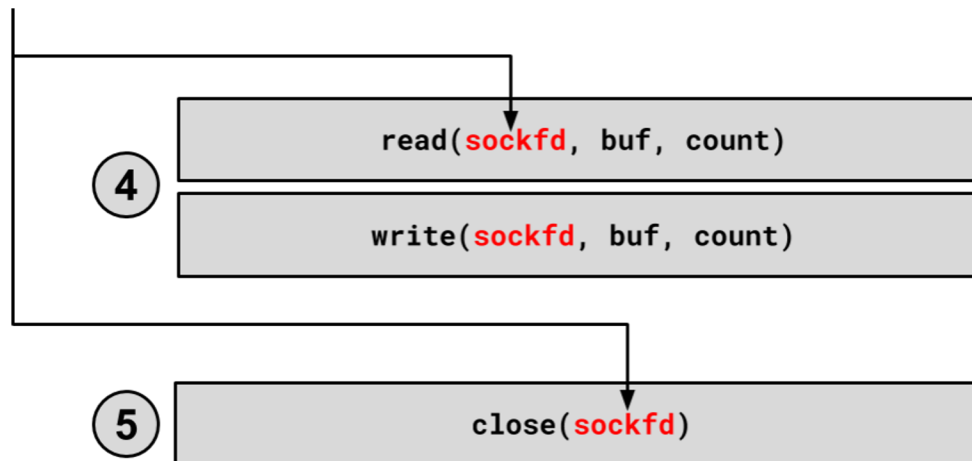# 3. connect()

```
int connect(int socket,                      // from 1
            const struct sockaddr *addr,  // from 2
            socklen_t addr_len); // size of serv_addr
```

- Connects an available socket to a specified address

- Returns 0 on success, -1 on failure

```
                    ┌─────────────────────────────┐
                    │ extract fields from result  │
                    │      (IPv4 vs IPv6)          │
                    └─────────────────────────────┘

┌──────────────────┐                                          ┌──────────────────────┐
│ int (sa_family)  │                                          │      struct          │
└──────────────────┘                                          │ sockaddr_storage*    │
                    ┌─────────────────────────────┐           └──────────────────────┘
               (2)  │ socket(domain, type, protocol)│
                    └─────────────────────────────┘

┌──────────────────┐
│ int (sockfd)     │
└──────────────────┘
                                           <------- Cast sockaddr_storage* to sockaddr*!

               (3)  ┌─────────────────────────────┐
                    │ connect(sockfd, serv_addr, addrlen)│
                    └─────────────────────────────┘
```

38

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

specify lookup hints

Dotted boxes should
be filled in with the
data type.

struct addrinfo*

1 **getaddrinfo**(hostname, servname, **hints**, **&res**)

struct addrinfo*

extract fields from result
(IPv4 vs IPv6)

int (sa_family)

**struct
sockaddr_storage***

2 **socket**(**domain**, type, protocol)

int (sockfd)

3 **connect**(**sockfd**, **serv_addr**, addrlen)

4 read(**sockfd**, buf, count)

write(**sockfd**, buf, count)

5 close(**sockfd**)

40
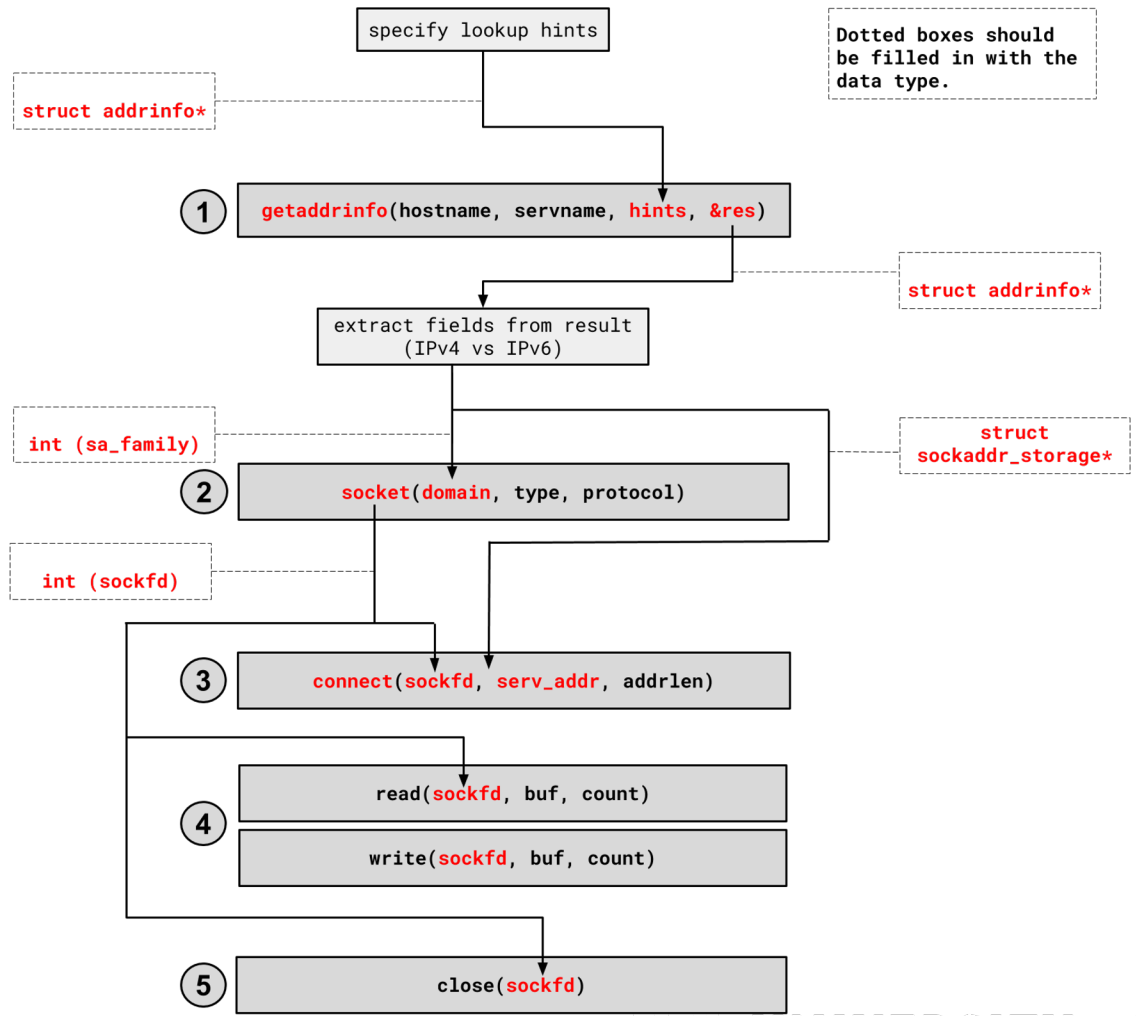
netcat

Using Netcat for the first time