CSE 333
Section 9

HW4 Intro, Client-side Networking

UNIVERSITY of WASHINGTON

Logistics

- Exercise 15
  - Due **Tomorrow (11/21) @ 10:00am**
- Homework 4
  - Due **12/03 @ 10:00pm**

HW4 and `netcat`

Web Server

1. Establish client connections
   a. Server socket set up
      in hw4/ServerSocket.cc
2. Read client requests
   a. Parse HTTP requests
      in hw4/HttpConnection.cc
3. Respond to requests
   a. Write HTTP responses
      in hw4/HttpServer.cc
4. Fix security vulnerabilities
   a. Escape characters
      in hw4/Utils.cc

Okay to copy and modify lecture/exercise code for HW4, just make sure you know what's going on!

Steps 2, 3, and 4 involve a lot of string manipulation which can be tedious! There might be something to help with that ☺

Using `netcat` with HW4

1. Launch the server

```
./http333d <port> ../projdocs/ unit_test_indices/*
```

2. Connect with telnet

```
nc -C <HostName> <port>
```

3. Write an HTTP request and send it

```
(Note: nc -C is needed on attu/vm/CSE workstations to use \r\n for newlines when talking
to web servers.  The option might be different on other machines (e.g., macs)
```

Writing an HTTP Request

- Example HTTP Request layout can be found in `HttpRequest.h`

- Example HW4 file request:
    - `GET /static/test_tree/books/artofwar.txt HTTP/1.1`

- Example HW4 query request:
    - `GET /query?terms=books+of+war HTTP/1.1`

- To send a request, hit [Enter] ***twice***

- Compare the output of `solution_binaries/http333d` to `./http333d`

Boost Library (HW4)

Boost

Boost is a free C++ library that provides support for various tasks in C++

- **Note:** Boost does NOT follow the Google style guide!!!
- These will be helpful for you in hw4 to parse HTTP Requests!

Boost adds many string algorithms that you may have seen in Java

- Include with `#include <boost/algorithm/string.hpp>`
- Documentation: https://www.boost.org/doc/libs/1_60_0/doc/html/string_algo.html
- **DO NOT** use the regex library, the string library should be enough.
    - i.e., OK to use any boost libraries that do not require changing hw4 Makefile

Helpful Functions

```
void boost::trim(string& input);
```

- Removes all leading and trailing whitespace from the string
- `input` is an input *and* output parameter (non-const reference)

```
void boost::replace_all(string& input,
                        const string& search,
                        const string& format);
```

- Replaces all instances of `search` inside `input` with `format`

Helpful Functions

```
void boost::split(vector<string>& output,
                  const string& input,
                  boost::PredicateT match_on,
                  boost::token_compress_mode_type compress);
```

- Split the string by the characters in `match_on`

```
boost::PredicateT boost::is_any_of(const string& tokens);
```

- Returns predicate that matches on any of the characters in tokens

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

Remember these are POSIX operations called using glibc C functions, though we are using them in our C++ programs

*difficulty is subjective

Sockets (Berkeley Sockets)

- Just a file descriptor for network communication
  - Defines a local endpoint for network communication
  - Built on various operating system calls

- Types of Sockets
  - Stream sockets (TCP)
  - Datagram sockets (UDP)
  - There are other types, which we will not discuss

- Each TCP socket is associated with **a TCP port number (`uint16_t`)** and **an IP address**
  - These are in network order (not host order) in TCP/IP data structures! (https://www.gnu.org/software/libc/manual/html_node/Byte-Order.html)
  - `ai_family` will help you to determine what is stored for your socket!

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

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`

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

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

Name of host whose IP we want

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

Hints for the lookup server/refine results

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

Step 1: Obtaining your server's socket address

```
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

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`!

Steps 4 and 5: Using your Connection

```
// 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);
```

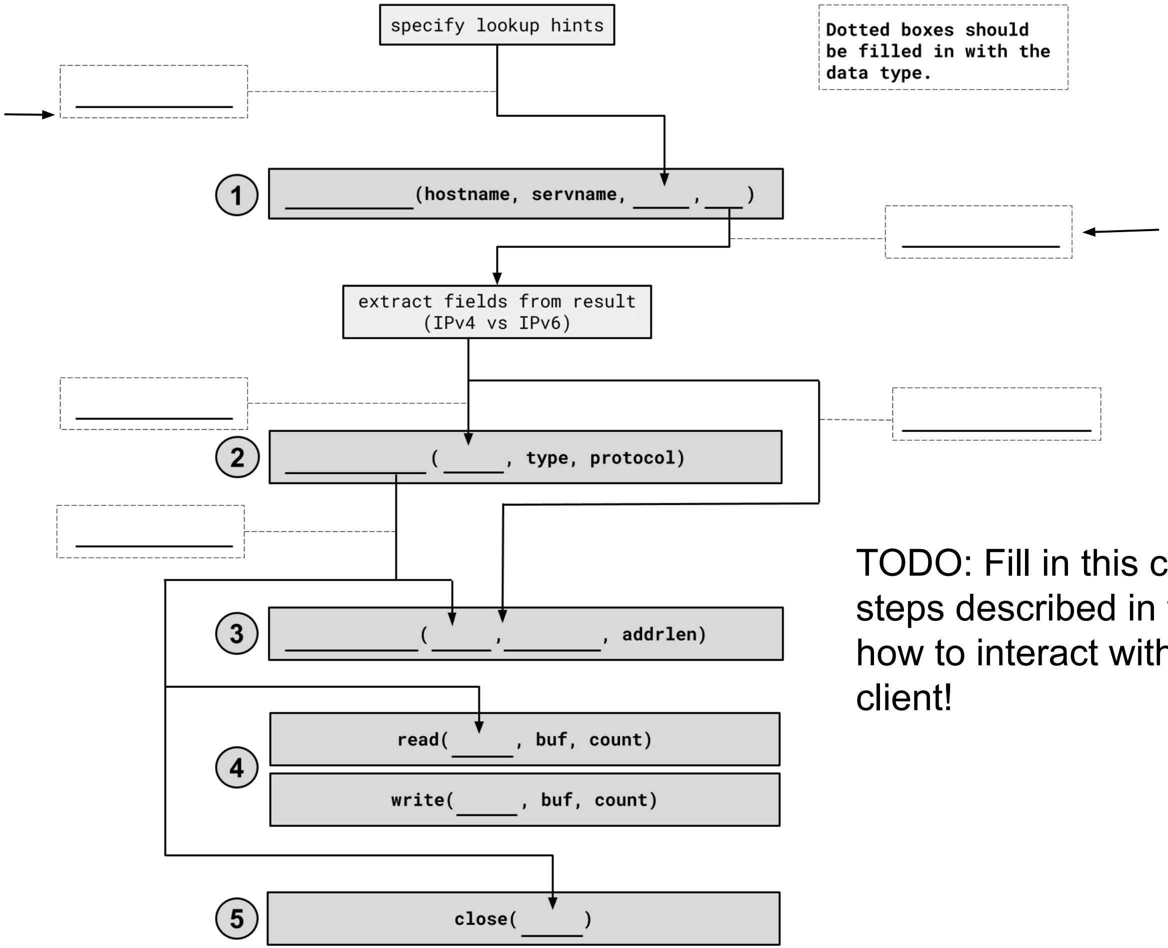
- Same POSIX methods we used for file I/O!
  (so they require the same error checking...)

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 (same as above)
5. Close the socket connection

Exercise 1

Input param



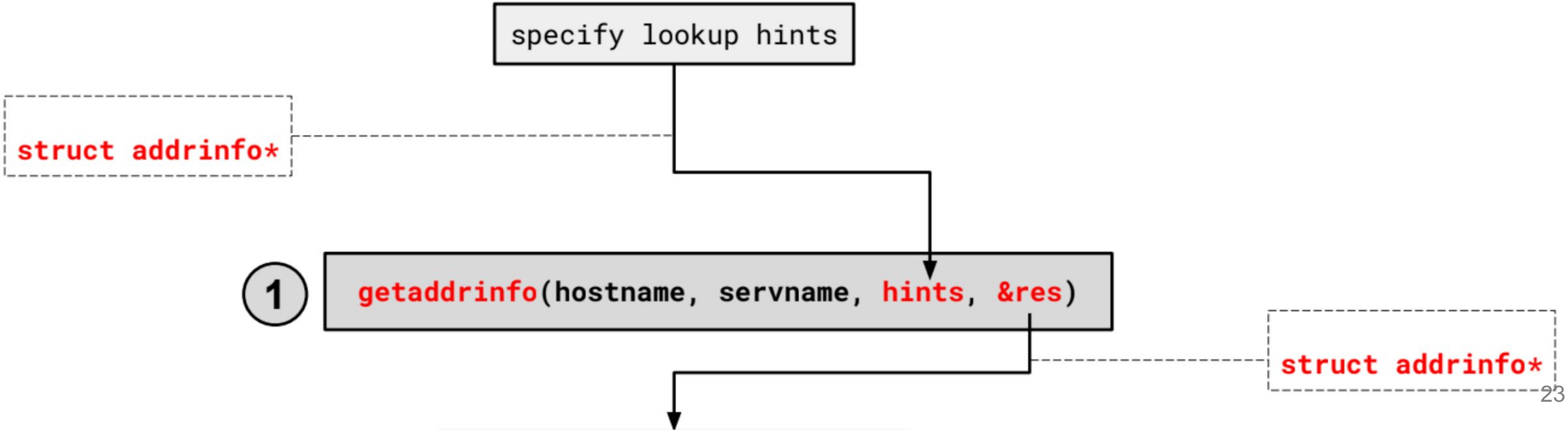
Output param

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

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

```
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 sockaddr for address
    char* ai_canonname; // canonical name
    struct addrinfo* ai_next; // can form a linked list
};
```

*Note that we get a linked list of results

1. getaddrinfo() - Interpreting Results

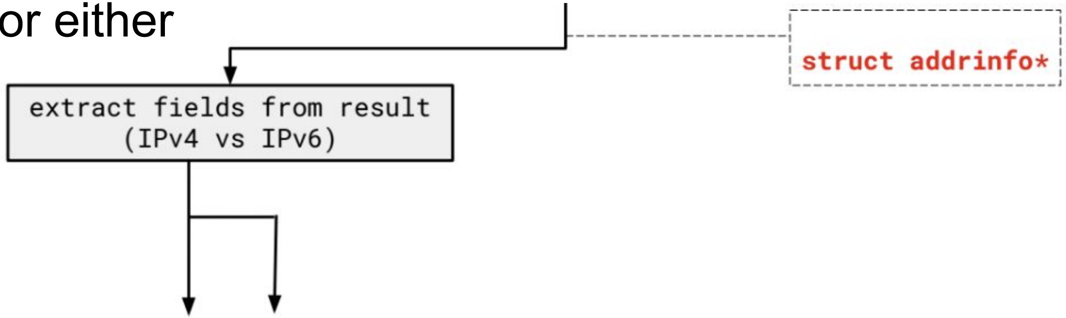
```
struct addrinfo {
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC
    struct sockaddr* ai_addr; // pointer to socket addr
    ...
};
```

- 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*`:

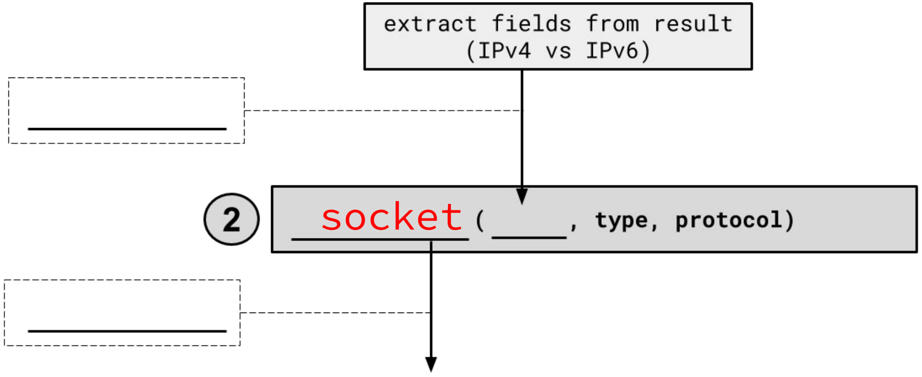
- The field `sa_family` describes if it is IPv4 or IPv6
- Cast to `struct sockaddr_in*` (v4) or `struct sockaddr_in6*` (v6) to access/modify specific fields (i.e. ports)
- Store results in a `struct sockaddr_storage` to have a space big enough for either



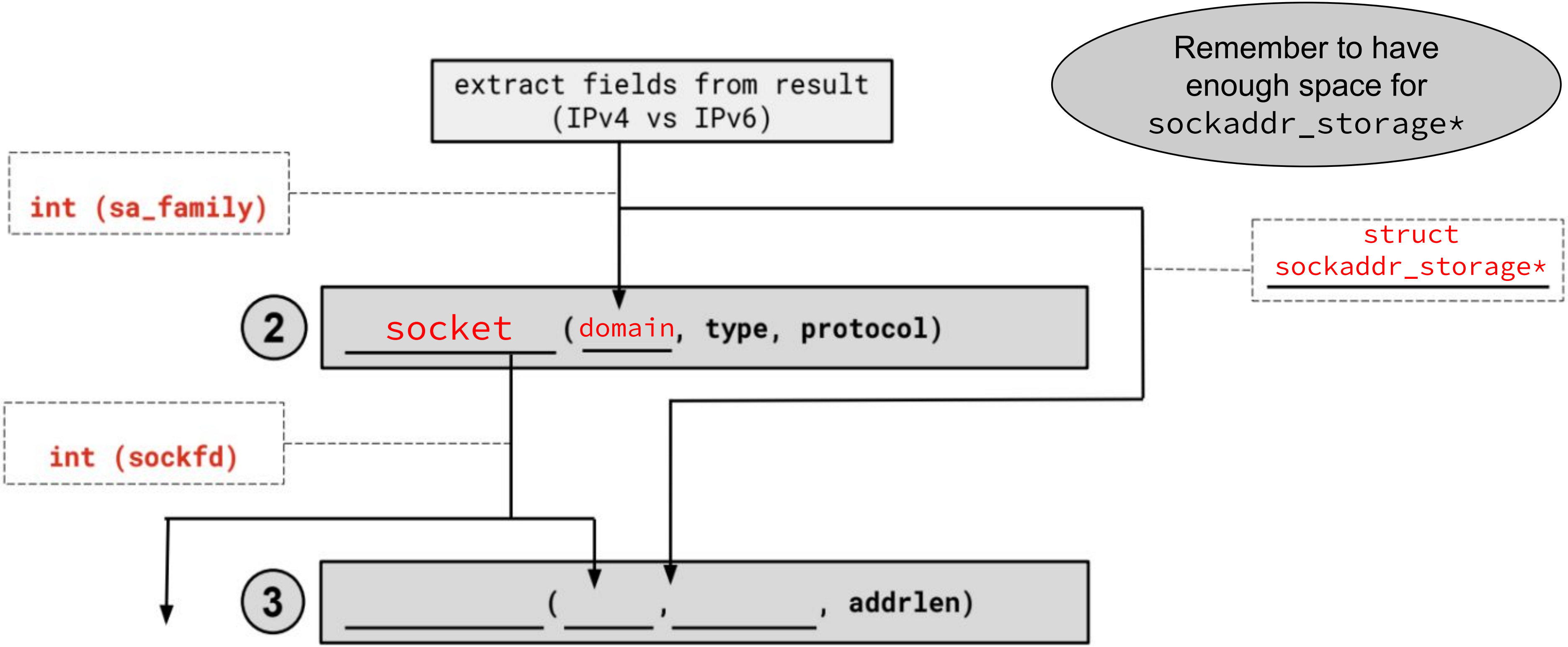
2. Build client side socket

```
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()

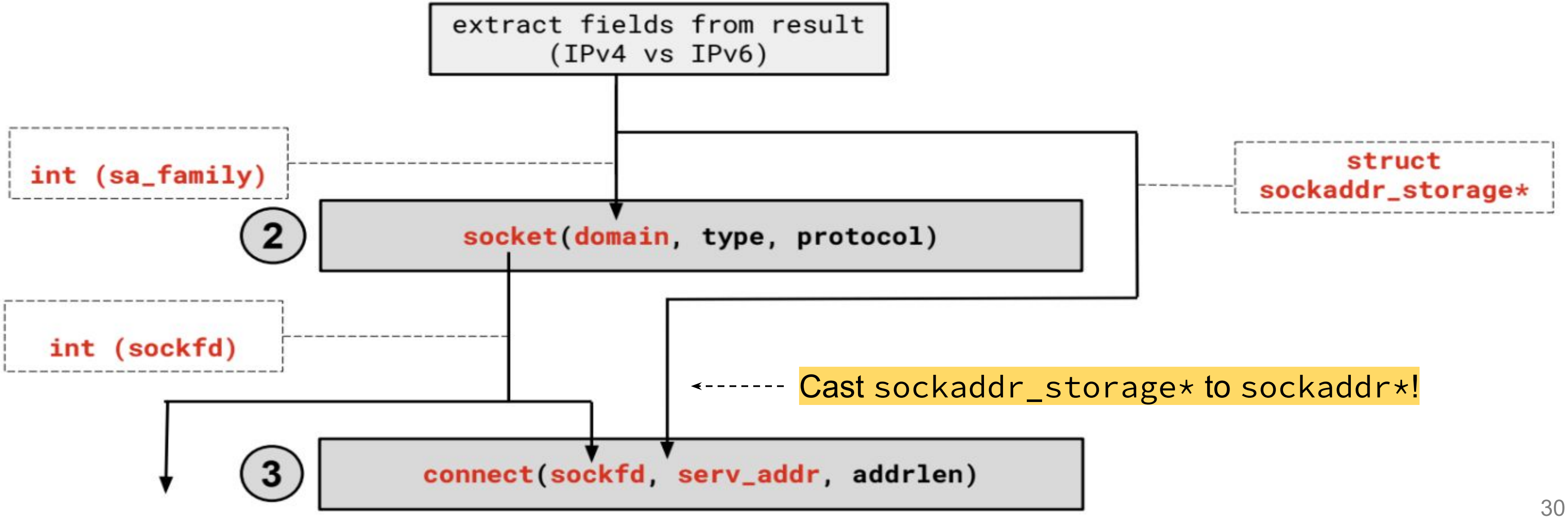
```
int connect(int socket,                         // socket fd
            const struct sockaddr *addr, // address to connect to
            socklen_t addr_len);         // length of *addr
```

- 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()

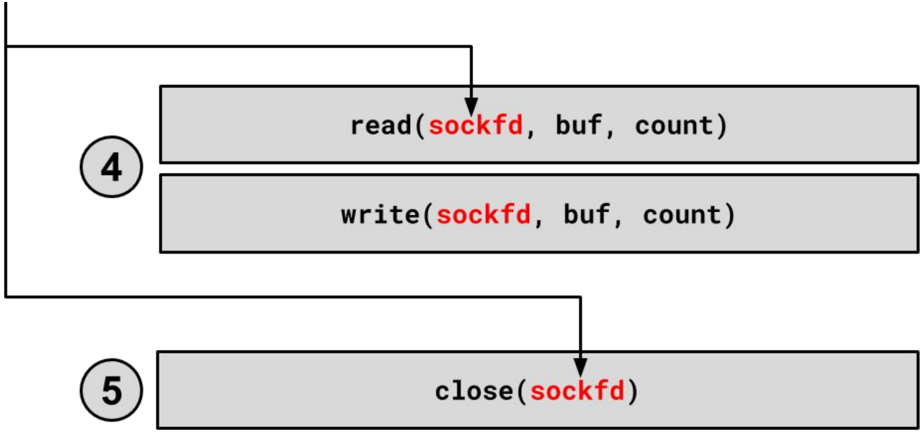
```
int connect(int socket,                       // from 1
            const struct sockaddr *addr,   // from 2
            socklen_t addr_len); // size of serv_addr
```

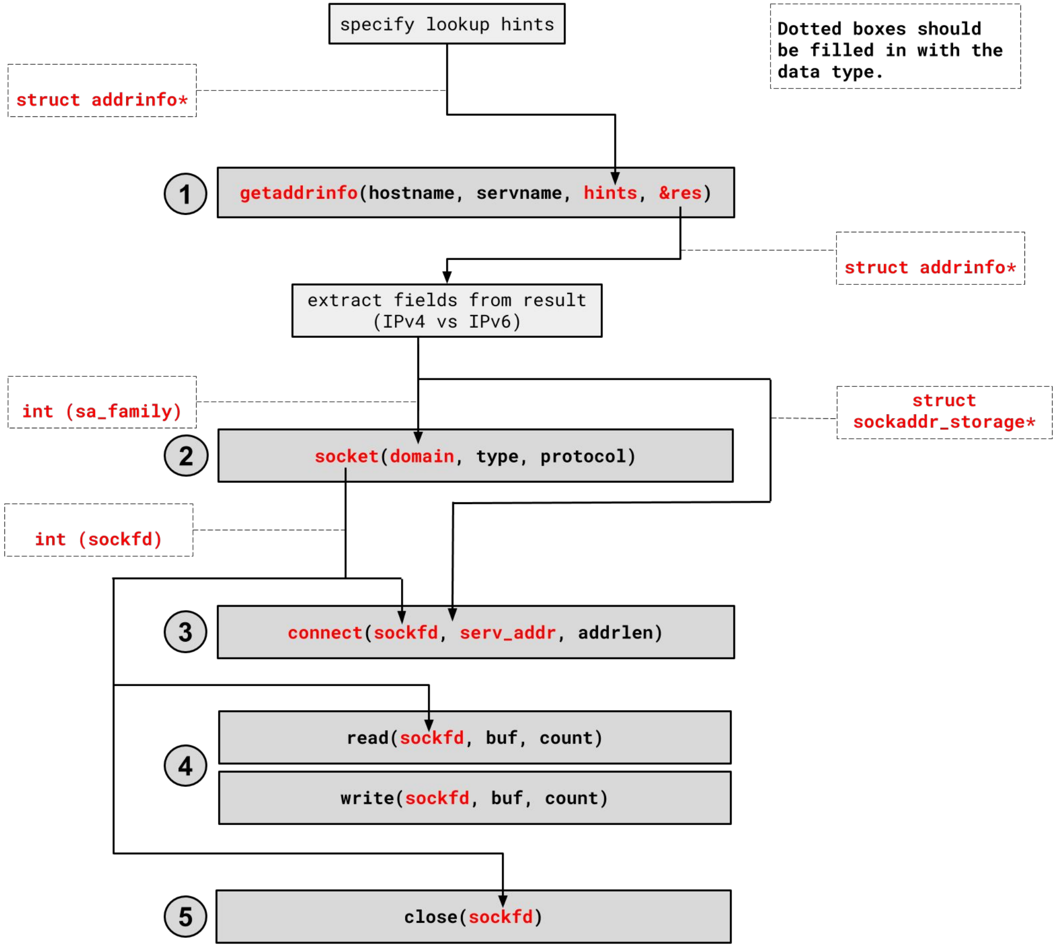
- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure



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)