

15-213 Recitation 12: Networking

4 April 2016

Ralf Brown and the 15-213 staff

Agenda

- Reminders
- Debugging and Optimizing Malloc
- Networking: Sockets
- Demo
- Appendix

Reminders

- Malloclab is due on **Friday!**

Debugging Malloc – Heap Checker

- Have you written `mm_checkheap()` yet?
 - If not, *what are you waiting for?*
 - Make the checking detailed enough that it passes if and only if the heap is truly well-formed (*what should it check?*)
- Call the heap checker before and after major operations
- Define macros to conveniently enable/disable heap checks

```
#ifdef DEBUG
#define CHECKHEAP(verbose) \
    printf("%s line %d\n", __func__, __LINE__); \
    mm_checkheap(verbose);
#else
#define CHECKHEAP(verbose)
#endif /* DEBUG */
```

Optimizing Malloc – Parameter Tuning

- You may want to tweak certain parameters for better performance
 - number of size classes
 - amount by which the heap is extended
- Much better to write modular and encapsulated code
 - Should only require changing a few lines of code to change parameters
 - Well-designed use of macros can accomplish this:

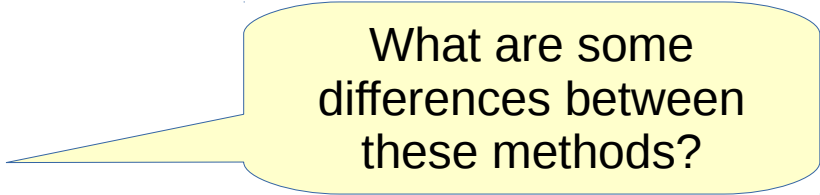
```
#define NUM_SIZE_CLASSES 5
freelist *buckets;
int mm_init(void) {
    buckets = mem_sbrk(NUM_SIZE_CLASSES*sizeof(freelist))
    if (buckets == NULL) return -1;
    for (int i=0; i<NUM_SIZE_CLASSES; i++) {
        buckets[i] = NULL;
    }
}
```

Inter-Process Communication

- What are the different ways for processes to communicate with each other that we've discussed in this course?

Inter-Process Communication

- Signals
- Files
- Pipes
- Remote Procedure Calls



What are some differences between these methods?

Inter-Process Communication

- Signals

- single-bit communication

- Files

- no real limit on amount of information transmitted
 - both processes must be able to see the file
 - synchronization issues

- Pipes

- synchronization by OS – blocks sender if receiver doesn't read fast enough

- Remote Procedure Calls

- acts like a regular function call, just slower (OS hides all the details)
 - limits on what can be passed and returned

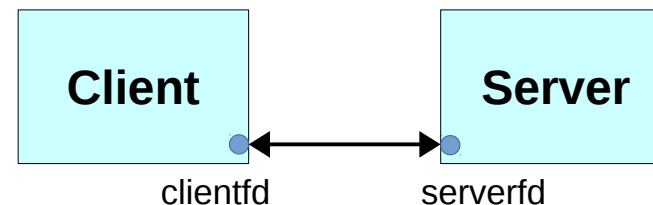
Sockets

- What is a socket?

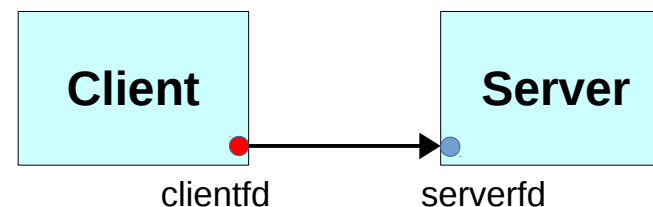
Sockets

- What is a socket?
 - A file descriptor that lets an application read/write from/to the network
 - (remember that all Unix I/O devices are modeled as files)
- Clients and servers communicate with each other using socket descriptors

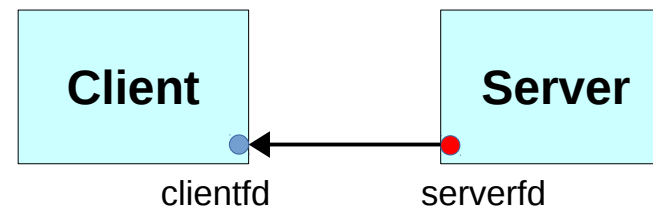
Sockets are bidirectional



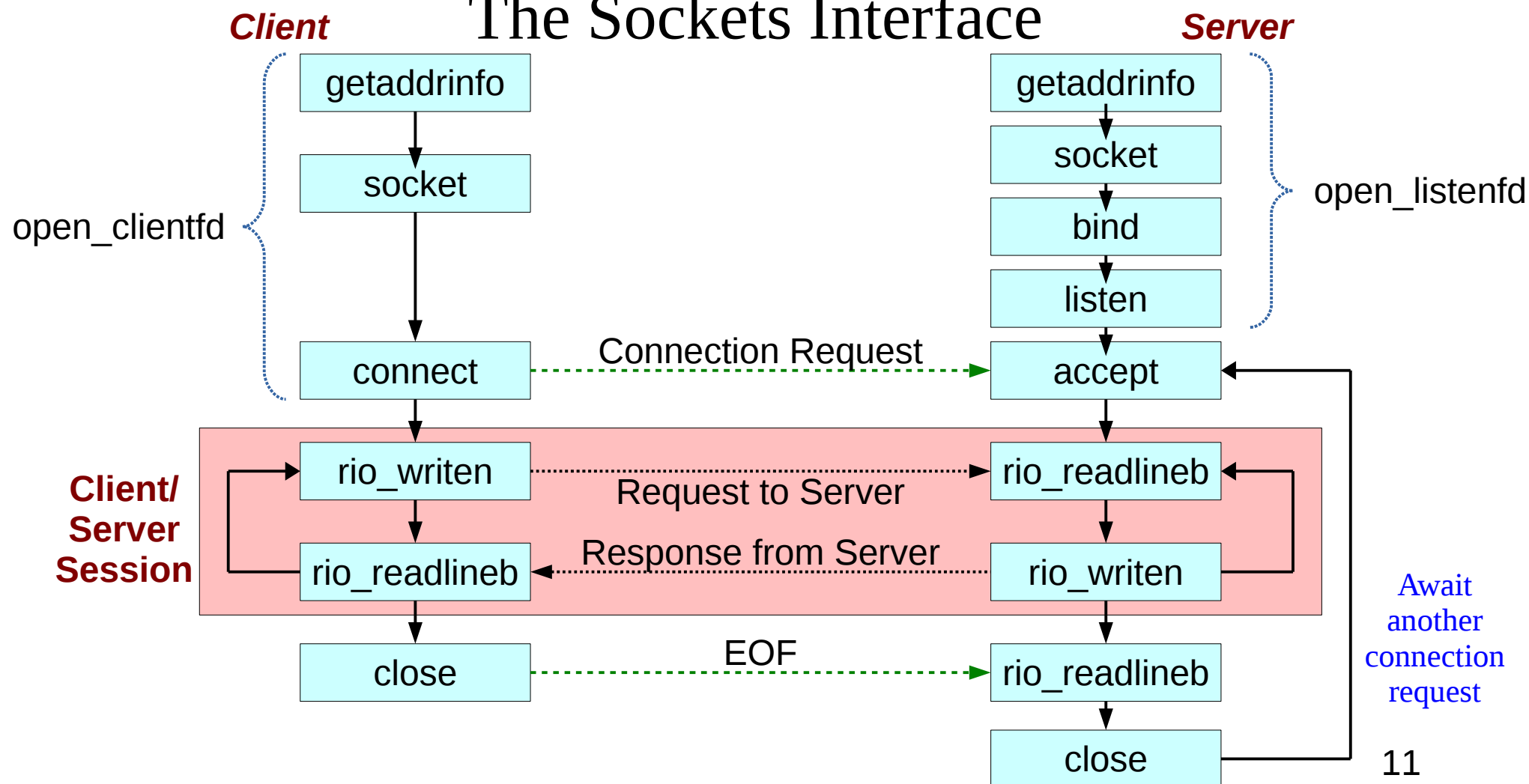
Data written to clientfd can be read from serverfd



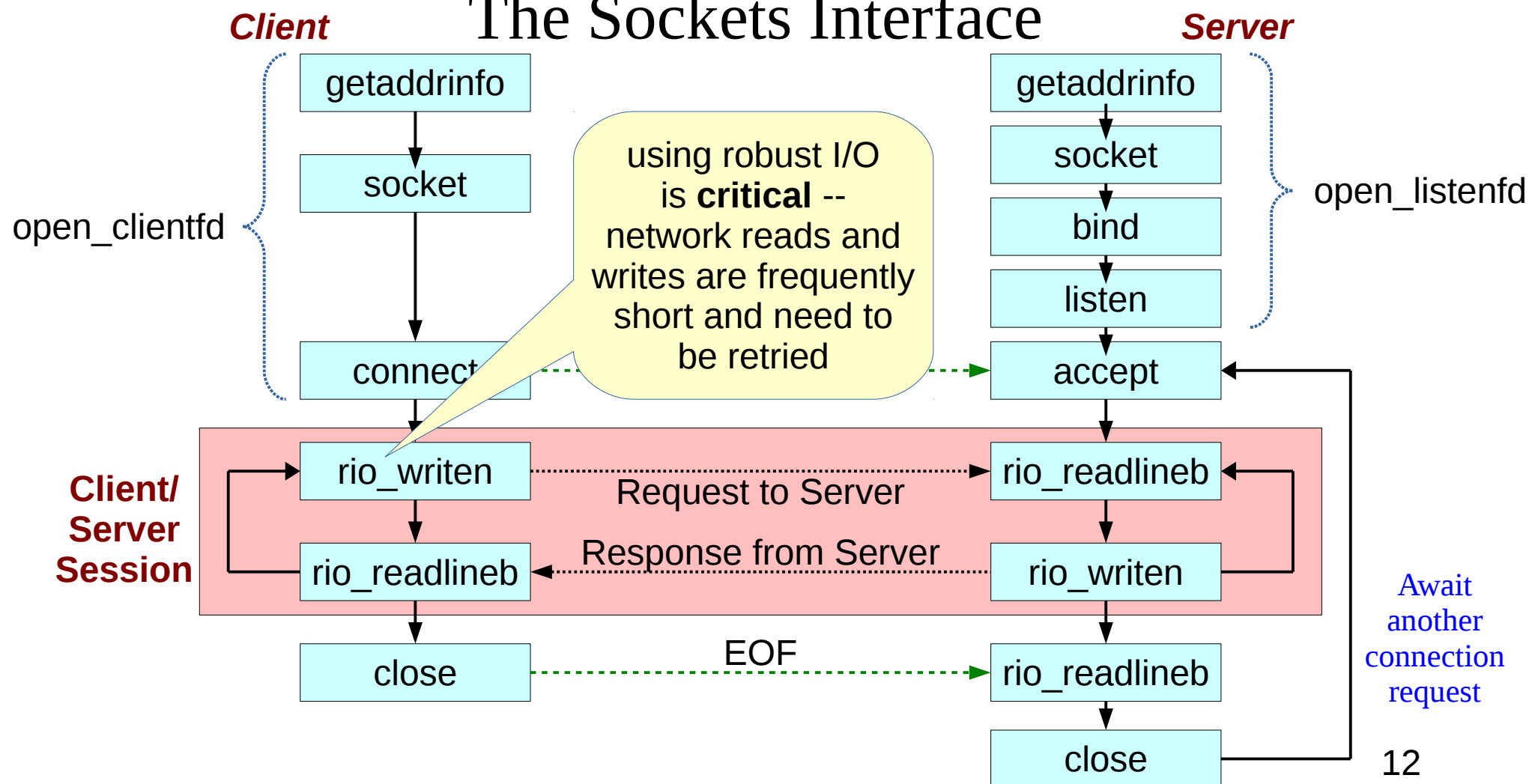
Data written to serverfd can be read from clientfd



The Sockets Interface



The Sockets Interface



If You Get Stuck

- ***Please read the writeup!***
- CS:APP Chapter 9
- View lecture notes and course FAQ at <http://www.cs.cmu.edu/~213>
- Office hours Sunday through Thursday 5:00-9:00pm in WeH 5207
- Post a **private** question on Piazza

DEMO: CLIENT/SERVER PROGRAMMING

APPENDIX

Debugging Malloc – Driver Error Messages

- mdriver can identify a number of errors
- the error message is straightforward
 - “garbled byte” means (part of) the payload has been overwritten
 - pointer arithmetic error
 - allocating an already-allocated block
 - “out of memory” results when you lose blocks or use memory very inefficiently
- Unfortunately, you'll typically get a segmentation fault...
 - use “gdb mdriver” to find the faulting line
 - remember that a segfault in line 150 could be caused by a bug in line 70!
 - use `mm_checkheap()` liberally to narrow down the bug's location

TO SAVE YOUR SANITY,
WRITE A GOOD
HEAP CHECKER!

(One which doesn't print anything unless it finds an error.)

Heap Invariants (Non-Exhaustive)

- Block level
 - header and footer match
 - payload area is aligned
- List level
 - next/prev pointers in consecutive free blocks are consistent
 - no allocated blocks in free list, all free blocks are in the free list
 - no contiguous free blocks unless you defer coalescing
 - no cycles in free list unless you use a circular list
 - each segregated list contains only blocks in the appropriate size class
- Heap level
 - all blocks between heap boundaries
- Add your own invariants (e.g. address order)

Other Malloc Problems to Watch Out For

- Uninitialized pointers and/or memory
- Make sure `mm_init()` actually initializes everything
 - it gets called by the driver between iterations of every trace
 - incomplete initialization can result in passing every trace file individually, but losing points on the complete driver test
- Make sure your heap checker is stable and correct! A faulty heap checker could complain even if your malloc code is correct.
 - Tracking down a bug in code that is actually correct is no fun – voice of experience speaking here....

Useful Tools

- Valgrind

- fix anything it reports as illegal access, uninitialized value, etc.

- GDB

- watchpoints

- `watch {location}` stop program when it writes to {location}
 - `rwatch {location}` stop program when it reads from {location}
 - `awatch {location}` stop program on **any** access to {location}

Beyond Debugging: Version Control

- “I had 60 util points just five minutes ago!”
- Save your allocator code after each major bit of progress
 - basic: copy files around using “cp”
 - alternative: keep different versions in separate .c files and use symbolic links
 - “ln -s mm-version-x.c mm.c” to start using a particular version
- Or use git/svn/cvs/...
 - Make sure your repository is **private** if you use remote repos (e.g. github or SourceForge)

Optimizing Malloc – Profiling

- When you hit a bottleneck, figure out which *part* of your code is limiting your performance
 - usually one particular spot is using up a large part of the run time
- A profiler is good for this kind of job
- “gprof” is available on most Linux machines, including the sharks
 - Add “-pg” to the compilation flag in your Makefile
 - Run the driver as normal. The profiled executable generates a file called gmon.out.
 - Run “gprof ./mdriver” to see the result in human-readable form
 - **Don't forget to remove the “-pg” from the Makefile when done** – profiled code runs noticeably slower

Sockets API: The `sockaddr` Struct

- Describes the network address to use
- Different variants for different networking protocols (the Internet is not the only network!)
 - you'll need to cast from the variant in use to `sockaddr*` when calling socket functions
- For Internet connections, we'll use `struct sockaddr_in`
- Key fields
 - `sin_family`: the network protocol, `AF_INET` for Internet
 - `sin_addr.s_addr`: the IP address, or `INADDR_ANY` for “don't care”
 - `sin_port`: the IP port number (e.g. 22 for SSH, 80 for HTTP)

Sockets API: Converting Host Name to Address

- `#include <sys/socket.h>`
- `#include <netdb.h>`
- `int getaddrinfo(const char *host, const char *service, const struct addrinfo *hints, struct addrinfo **result);`
 - return a list of all matching possibilities for connecting to the given host and service
- `void freeaddrinfo(struct addrinfo *result);`
 - free the list of `addrinfo` structures returned by `getaddrinfo`
- `getaddrinfo` also simplifies generating the necessary `sockaddr` structure when setting up as a server or connecting to a server

Sockets API: Creating a Socket

- `#include <sys/socket.h>`
- `int socket(int domain, int type, int protocol);`
 - creates a file descriptor for network communications
 - used by both clients and servers
 - domain: `PF_INET` – Internet, `PF_UNIX` – files and pipes, etc.
 - type: `SOCK_STREAM` – reliable connection, `SOCK_DGRAM` – connectionless, etc.
 - protocol: `IPPROTO_TCP` – TCP
- for Proxylab, you'll use
 - `int sock_fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);`

Sockets API: Associating with a Network Port

- `int bind(int socket, const struct sockaddr *addr, socklen_t addr_len);`
- associate the socket with an IP address and port number
- used by servers

```
struct addrinfo hints, *info;
int status;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
if ((status=getaddrinfo(NULL, "http", &hints, &info)) != 0)
    addrinfo_error(status);
if (bind(sock_fd, info->ai_addr, info->ai_addrlen) < 0)
    unix_error("bind");
freeaddrinfo(info);
```

Sockets API: Associating with a Network Port

- `int bind(int socket, const struct sockaddr *addr, socklen_t addr_len);`

- associate the socket with an IP address

- used by servers

```
struct addrinfo hints, *info;  
int status;
```

```
memset(&hints, 0, sizeof(hints));  
hints.ai_family = AF_INET;  
hints.ai_socktype = SOCK_STREAM;  
hints.ai_flags = AI_PASSIVE;
```

```
if ((status=getaddrinfo(NULL, "http", &hints, &info)) != 0)  
    addrinfo_error(status);
```

```
if (bind(sock_fd, info->ai_addr, info->ai_addrlen) < 0)  
    unix_error("bind");
```

```
freeaddrinfo(info);
```

because getaddrinfo can return multiple addrinfo records, we need to iterate through them (by following info->ai_next) until one succeeds or we find a NULL pointer

put the bind() in a while loop

Sockets API: Accepting Connections from Clients

- `int listen(int socket, int backlog);`
 - wait for incoming connections, with a limit on waiting connections
 - new connections will be refused once this limit is hit
- `int accept(int socket, struct sockaddr *addr, socklen_t *addr_len);`
 - create a new socket for communicating with the client that just requested a connection; returns -1 on error
 - fills in `addr` with the client's address on success

```
struct sockaddr_in client_addr;  
  
client_fd = accept(listener_fd, &client_addr,  
                  sizeof(client_addr)) ;
```

Sockets API: Connecting to a Server

- `int connect(int socket, struct sockaddr *addr, socklen_t addr_len);`
- used by clients
- use `getaddrinfo()` to build the `sockaddr` for you:

```
struct addrinfo hints, *info;
int status;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
if ((status=getaddrinfo(hostname, "http", &hints, &info)) != 0)
    addrinfo_error(status);
if (connect(sock_fd, info->ai_addr, info->ai_addrlen) < 0)
    unix_error("connect");
freeaddrinfo(info);
```

Sockets API: Connecting to a Server

- `int connect(int socket, struct sockaddr *addr, socklen_t addr_len):`

- used by clients
- use `getaddrinfo()` to build the `sockaddr`

```
struct addrinfo hints, *info;  
int status;
```

```
memset(&hints, 0, sizeof(hints));  
hints.ai_family = AF_INET;  
hints.ai_socktype = SOCK_STREAM;
```

```
if ((status=getaddrinfo(hostname, "http", &hints, &info)) != 0)  
    addrinfo_error(status);
```

```
if (connect(sock_fd, info->ai_addr, info->ai_addrlen) < 0)  
    unix_error("connect");
```

```
freeaddrinfo(info);
```

because `getaddrinfo` can return multiple `addrinfo` records, we need to iterate through them (by following `info->ai_next`) until one succeeds or we find a NULL pointer

put the `connect()` in a while loop

Sockets API: Reading and Writing

- `ssize_t read(int fd, void *buf, size_t nbytes);`
 - same as for file I/O: read up to `nbytes` bytes into the buffer at `buf`
 - used by both clients and servers
- `ssize_t write(int fd, void *buf, size_t nbytes);`
 - same as for file I/O: write `nbytes` bytes from the buffer at `buf`
 - used by both clients and servers

Sockets API: Closing the Socket

- `int close(int fd);`
 - same as for file I/O: close the stream and free any resources allocated to it
 - used by both clients and servers

CS:APP Sockets Utility Functions

- `int open_clientfd(char *host, char *port);`
 - open a connection to the given port on the named host
- `int open_listenfd(char *port);`
 - create a socket, bind it to the given port, and set it to listening
- Both functions return -2 on `getaddrinfo` error, -1 on other errors (with `errno` set)
- Wrapped versions `Open_clientfd` and `Open_listenfd` are also available.