Socket Programming
Lecture 2

Daniel Spangenberger
15-441 Computer Networks, Fall 2007

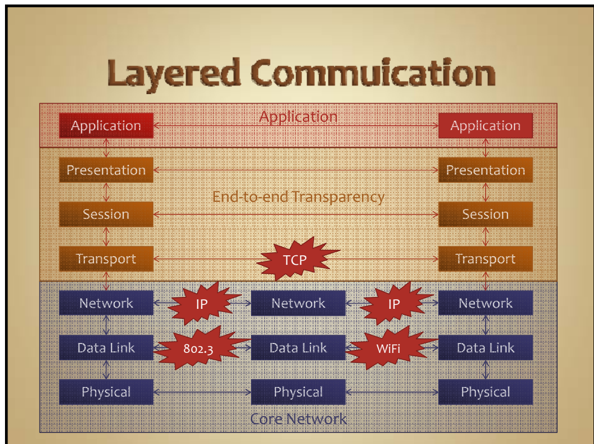
Why Do I Want Networking?

- Goal of Networking: Communication
  - Share data
  - Pass Messages
- Say I want to talk to a friend in Singapore…
  - How can I do this?
  - What applications and services must I use?
  - Where can I access them?
  - How will the data get there?
  - Will it be reliable?

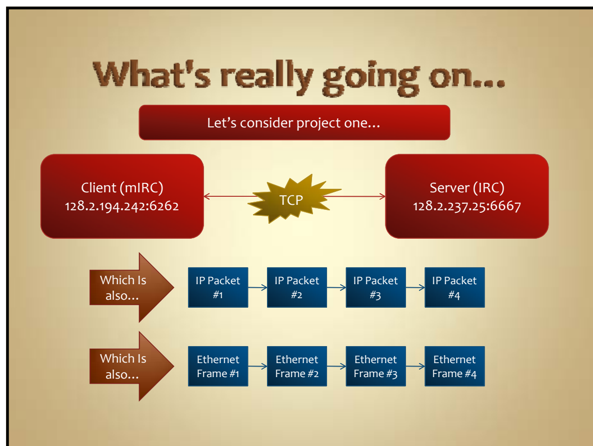
Lecture Today…

- Motivations for Sockets
- What's in a Socket?
- Working with Sockets
- Concurrent Network Applications
- Software Engineering for Project 1

Layered Commuication



What's really going on…



Which is easier?

- An application programmer (writing an IRC server)
  - Doesn't need to send IP packets
  - Doesn't need to send Ethernet frames
  - Doesn't need to worry about reliability
  - Shouldn't have to!
- Sockets do this!
  - TCP streams
  - UDP packetized service (Project 2)
- You'll be doing this! (using sockets)
  - To share data
  - To pass messages

Interlude: Project 1 & 2

- IRC Server (Project 1)
  - Login
  - Reverse host lookup
  - Support for channels
  - Support for private messages
- Message Routing (Project 2)
  - Unicast routing (OSPF)
  - Multicast routing (MOSPF)
  - Server extensions for message forwarding

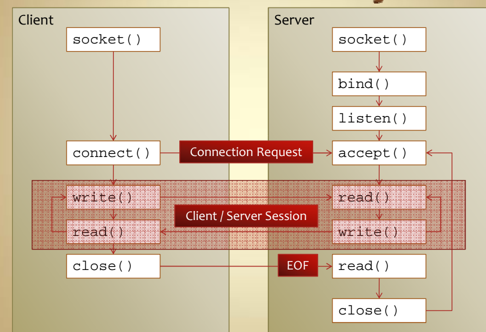
Car Talk: What's in a Socket

- Some information needed…
  - Where is the remote machine?
    - IP Address
    - Hostname (resolved to IP)
  - Which service do I want?
    - Port
- After that…
  - You get a file! A plain old file!
  - As simple as other Unix I/O
  - Don't forget to close it when you're done!

Car Talk: How do I do it?

- Request a socket descriptor
  - Both the client and the server need to
  - Bunch of kernel allocations…
- And the server…
  - Binds to a port
    - "I am offering a service on port x. Hear me roar"
  - Listens to the socket
    - "Hey! Say something!"
  - Accepts the incoming connection
    - "Good, you spoke up!"
- And the client…
  - Connects
    - "I'm interested!"

Sockets: The lifecycle



Step One: Socket-time

- Both the client and server need to setup the socket
  - `int socket(int domain, int type, int protocol)`
- Domain
  - `AF_INET` (IPv4, also IPv6 available)
- Type
  - `SOCK_STREAM` TCP (your IRC server)
  - `SOCK_DGRAM` UDP (your routing daemon)
- Protocol
  - `0` (trust us, or, read the manpage)

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Step Two: Bind it up

- Server-only (read the man-page!)
  - `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);`
- sockfd
  - A file descriptor to bind with, what socket returned!
- my_addr
  - It's a struct (duh), describing an Internet socket/endpoint

```
struct sockaddr_in {
short           sin_family;     // e.g. AF_INET
unsigned short  sin_port;       // e.g. htons(3490)
struct in_addr  sin_addr;       // see struct in_addr, below
char            sin_zero[8];    // zero this if you want to
};
struct in_addr {
unsigned long   s_addr;  // load with inet_aton()
};
```

Step Two: Bind it up (cont)

- addrlen
  - `sizeof(your sockaddr_in struct)`

```
struct sockaddr_in my_addr;
int sockfd;
unsigned short port = 80;
if (0 > (sockfd = socket(AF_INET, SOCK_STREAM, 0))) {
        printf("Error creating socket\n");
        ...
}
memset(&saddr, '\0', sizeof(saddr)); // zero structure out
my_addr.sin_family = AF_INET; // match the socket() call
my_addr.sin_addr.s_addr = htonl(INADDR_ANY); // bind to any local address
my_addr.sin_port = htons(port); // specify port to listen on
if((bind(sockfd, (struct sockaddr *) &my_addr, sizeof(saddr)) < 0) {
        printf("Error binding\n");
        ...
}
```

What was that cast?

- `bind()` takes a `sockaddr`

```
struct sockaddr {
        short int sa_family;        // "virtual pointer"
        char sa_data[14];           // address info
}
```

- C polymorphism
  - There's a different `sockaddr` for IPv6!
  - And options for more in the future…

htonl() what?

- A little lesson on byte ordering…
  - Network byte ordering is defined to be big-endian
  - x86, x86-64 are little endian
- So how do we convert?
  - htons() / htonl() – Convert host order to network order
  - ntohs() / ntohl() – Convert network order to host order
- And what needs to be converted?
  - Addresses
  - Ports
  - Practically anything that deals with a network syscall
  - Maybe even data (up to the protocol designer)

Step Three: Listen in

- Allows the server to listen for new connections
  - `int listen(int sockfd, int backlog)`
- sockfd
  - A file descriptor to listen on, what socket returned!
- backlog
  - The number of connections to queue

```
listen(sockfd, 10);
```

Step Four: Accept the masses!

- The server must explicitly accept connections
  - `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
- sockfd
  - A file descriptor to listen on, what socket returned!
- addr
  - Pointer to a `sockaddr_in`, cast as `sockaddr*` to store the client's address information in
- addrlen
  - Pointer to an int to store the returned size of `addr`, should be initialized as `sizeof(addr)`

```
int csock = accept(sockfd, (struct sockaddr_in *)
                          &caddr, &clen);
```

Tying the server up now…

```
struct sockaddr_in saddr, caddr;
int sockfd, clen, isock;
unsigned short port = 80;
if (0 > (sockfd=socket(AF_INET, SOCK_STREAM, 0))) {
        printf("Error creating socket\n");
        ...
}
memset(&saddr, '\0', sizeof(saddr)); // zero structure out
saddr.sin_family = AF_INET; // match the socket() call
saddr.sin_addr.s_addr = htonl(INADDR_ANY); // bind to any local address
saddr.sin_port = htons(port); // specify port to listen on
if (0 > (bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr))) {
        printf("Error binding\n");
        ...
}
if (0 > listen(sockfd, 5)) { // listen for incoming connections
        printf("Error listening\n");
        ...
}
clen = sizeof(caddr)
if (0 > (isock = accept(sockfd, (struct sockaddr *) &caddr, &clen))) {
        printf("Error accepting\n");
        ...
}
```

So what about the client?

- Client does not need to `bind`, `listen`, or `accept`
- Client needs only to `socket` and `connect`
  - `int connect(int sockfd, const struct sockaddr *saddr, socklen_t addrlen);`

```
connect(sockfd, (struct sockaddr *) &saddr, sizeof(saddr));
```

And now for the client...

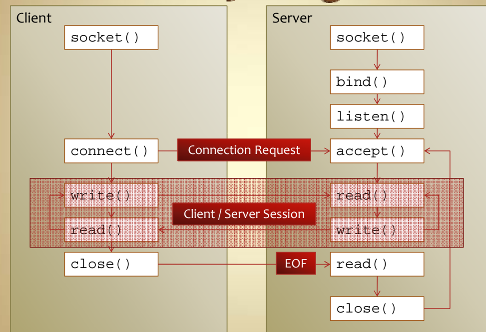
```
struct sockaddr_in saddr;
struct hostent *h;
int sockfd, connfd;
unsigned short port = 80;
if (0 > (sockfd=socket(AF_INET, SOCK_STREAM, 0))) {
        printf("Error creating socket\n");
        ...
}
// looking up the hostname
if (NULL == (h=gethostbyname("www.slashdot.org"))) {
        printf("Unknown host\n");
        ...
}
memset(&saddr, '\0', sizeof(saddr)); // zero structure out
saddr.sin_family = AF_INET; // match the socket() call
memcpy((char *) &saddr.sin_addr.s_addr,
                h->h_addr_list[0],
                h->h_length); // copy the address
saddr.sin_port = htons(port); // specify port to connect to
if (!connect(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) {
        printf("Cannot connect\n");
        ...
}
```

A Connection, at last!

- But what now? Send data of course!
  - `write()`
  - `read()`
- Both are used by the client and the server
- To write and read
  - `ssize_t read(int fd, void* buf, size_t len);`
  - `ssize_t write(int ffd, const void* buf, size_t len);`

```
read(sockfd, buffer, sizeof(buffer));
write(sockfd, "what's up?\n", strlen("what's up?\n"));
```

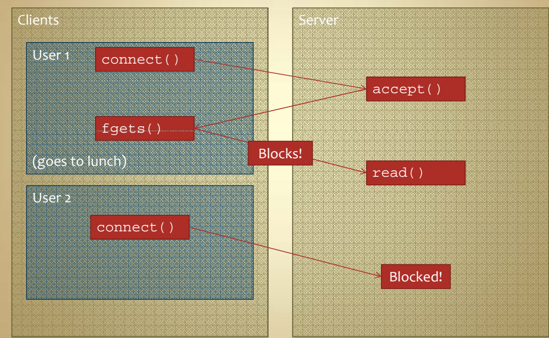
The lifecycle again...



And when were done...

- You must `close()` it!
  - Just like a file (since it appears as one!)
- What next? Loop around...
  - Accept new connections
  - Process requests
  - Close them
  - Rinse, repeat
- What's missing here?

A scenario...



Concurrency

"[…]a property of systems in which several computational processes are executing at the same time[…]" (thanks Wikipedia!)

How do we add concurrency?

- Threads
  - Natural concurrency (new thread per connection)
  - Easier to understand (you know it already)
  - Complexity is increased (possible race conditions)
- Use non-blocking I/O
  - Uses `select()`
  - Explicit control flow (no race conditions!)
  - Explicit control flow more complicated though

There are good arguments for each but you must use `select()`!

Adding Concurrency: Step One

- Start with allowing address re-use

```
int sock, opts;
sock = socket(…);
// getting the current options
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opts, sizeof(opts));
```

- Then we set the socket to non-blocking

```
// getting current options
if (0 > (opts = fcntl(sock, F_GETFL)))
        printf("Error…\n");
// modifying and applying
opts = (opts | O_NONBLOCK);
if (fcntl(sock, F_SETFL, opts))
        printf("Error…\n");
bind(…);
```

Adding Concurrency: Step Two

- Monitor sockets with `select()`
  - `int select(int maxfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *timeout);`
- So what's an `fd_set`?
  - Bit vector with `FD_SETSIZE` bits
- `maxfd` – Max file descriptor + 1
- `readfs` – Bit vector of read descriptors to monitor
- `writefds` – Bit vector of write descriptors to monitor
- `exceptfds` – Read the manpage, set to `NULL`
- `timeout` – How long to wait with no activity before returning, `NULL` for eternity

How does the code change?

```
struct sockaddr_in saddr, caddr;
int sockfd, clen, isock;
unsigned short port = 80;
if (0 > (sockfd=socket(AF_INET, SOCK_STREAM, 0)))
        printf("Error creating socket\n");
memset(&saddr, '\0', sizeof(saddr)); // zero structure out
saddr.sin_family = AF_INET; // match the socket() call
saddr.sin_addr.s_addr = htonl(INADDR_ANY); // bind to any local address
saddr.sin_port = htons(port); // specify port to listen on
if (0 > (bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)))
        printf("Error binding\n");
if (listen(sockfd, 5) < 0) { // listen for incoming connections
        printf("Error listening\n");
clen = sizeof(caddr)
// Setup your read_set with FD_ZERO and the server socket descriptor
while (1) {
        pool.ready_set = &pool.read_set;
        pool.nready = select(pool.maxfd+1, &pool.ready_set,
                        &pool.write_set, NULL, NULL);
        if (FD_ISSET(sockfd, &pool.ready_set)) {
                if (0 > (isock = accept(sockfd, (struct sockaddr *)
                        &caddr, &clen)))
                        printf("Error accepting\n");
                add_client(isock, &caddr, &pool);
        }
        check_clients(&pool);
}
// close it up down here
```

What was pool?

- A struct something like this:

```
typedef struct s_pool {
        int maxfd;              // largest descriptor in sets
        fd_set read_set;        // all active read descriptors
        fd_set write_set;       // all active write descriptors
        fd_set ready_set;       // descriptors ready for reading
        int nready;             // return of select()

        int clientfd[FD_SETSIZE];       // max index in client array

        // might want to write this
        read_buf client_read_buf[FD_SETSIZE];

        // what else might be helpful for project 1?
} pool;
```

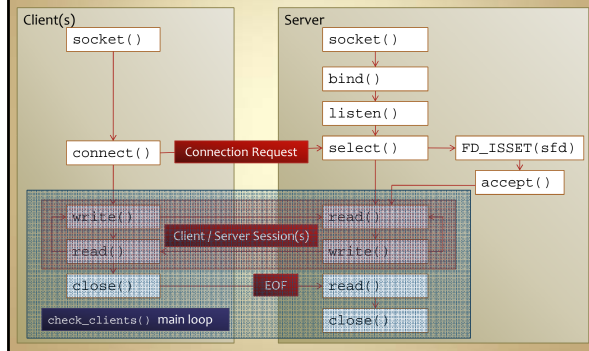
So what about bit vectors?

- `void FD_ZERO(fd_set *fdset);`
  - Clears all the bits
- `void FD_SET(int fd, fd_set *fdset);`
  - Sets the bit for fd
- `void FD_CLR(int fd, fd_set *fdset);`
  - Clears the bit for fd
- `int FD_ISSET(int fd, fd_set *fdset);`
  - Checks whether fd's bit is set

What about checking clients?

- The code only tests for new incoming connections
  - But we have many more to test!
- Store all your client file descriptors
  - In `pool` is a good idea!
- Several scenarios
  - Clients are sending us data
  - We may have pending data to write in a buffer
- Keep the `while(1)` thin
  - Delegate specifics to functions that access the appropriate data
  - Keep it orthogonal!
- Note the design presented here is not the best
  - Think up your own!

Back to that lifecycle...



Some subtleties...

- IRC commands are terminated by a newline
  - But you might not get one at the end of a `read()`!
- Buffers are your friend
  - `read()` returns exactly what is available—that might not be what you want!
  - If you don't have an entire line, buffer it and wait for more (but don't block!)
- Do not use the "Robust I/O" package from 213
  - It's not robust
  - Don't use anything from `csapp.h`

So What Now?

- So what do I do now?
  - Read the handout (Tuesday!)
  - Come to recitation (Wednesday)
  - Meet with your partner
- This may be a progression goals to achieve...
  - Construct a simple echo server for a single client
  - Construct a simple client to talk to that server
  - Modify your server to work with multiple clients
  - Modify your echo server to be a chat server
  - IRC?

Software Engineering
Tools for Project 1

- Version Control – Subversion
- Automated Build "Management" – Makefiles
- Automated Test "Management" – Makefiles
- Unit Tests – CUnit
- Integration Tests – Our custom test harness (or yours!)
- Debugging
  - Logging Macros
  - GDB
  - Valgrind (we'll run it for you if you don't!)
- Linked list and hash table library
  - You **don't** have to write your own!

Software Engineering
Skills for Project 1

- Team Programming
  - Code review (every check-in!)
  - Pair Programming (sometimes two sets of eyes are better!)
- Debugging
  - It will take days to figure some out!
- Writing Test Cases
  - Know how to find the corner cases!
- Design
  - Strive for orthogonal design, refactor to achieve it!

Suggestions

- Start early
  - Most groups that did not do well on Project 1 last fall started the IRC server just days before its checkpoint…
  - Find a partner, before the handout is released
- Meet early and often
  - Set goals with each other
    - "I will have basic network code for an echo server by tomorrow"
  - Pair programming is useful
    - In the beginning it is difficult to start on something with no direction… plan it out together
- Work ahead of the checkpoints
  - Checkpoint scripts likely released a day prior to checkpoint…
- Read the manpages, please

Good Luck!
Questions?

Email project partners to Albert:

albert@cmu.edu

http://www.cs.cmu.edu/~srini/15-441/F07
academic.cs.15-441