Project 1

15-441/641: Computer Networks

Many Thanks to past TA’s

Rajul Bhatnagar
Priya Ranjan Jha
What do you have to build?

<table>
<thead>
<tr>
<th>Checkpoint 1 (Echo Server)</th>
<th>Checkpoint 2 (HTTP 1.1)</th>
<th>Final Submission</th>
</tr>
</thead>
<tbody>
<tr>
<td>• I/O Multiplexing using select</td>
<td>• Request Parser</td>
<td>• HTTPS</td>
</tr>
<tr>
<td></td>
<td>• GET, HEAD, POST</td>
<td>• CGI</td>
</tr>
</tbody>
</table>
IP Addresses

- 32 bit—4,294,967,296 possible addresses
Port Numbers

• 16 bit—65,536 possible ports \([0,65535]\)

• \([0,1023]\) are well-known ports (reserved)
  • 80 — Hypertext Transport Protocol (HTTP)
  • 22 — Secure Shell (SSH)
  • 25 — Simple Mail Transfer Protocol (SMTP)

• \([1024,49151]\) are registered ports (IANA)
  • 2967 — Symantec AntiVirus
  • 3074 — XBOX Live

• \([49152,65535]\) are ephemeral ports (temp)
What's “in” a socket?

I want a program on computer A to talk to a program on computer B

Source <ip,port> + Destination <ip,port>
IP? Port!? 

Source `<ip, port>` + Destination `<ip, port>`

- **Identifies the Machine**
- **Identifies a Socket**
  (multiple apps want to network!)
How to: Server

- Create a socket via `socket()`
- Bind to an endpoint via `bind()`
- Listen for connections via `listen()`
- Accept connections via `accept()`
- Read from socket via `recv()`
- Write to socket via `send()`
#include <sys/socket.h>

int socket(int socket_family, int socket_type, int protocol);

Generally set `socket_family` to `PF_INET` → IPv4

`socket_type` to `SOCK_STREAM` → TCP

`protocol` to `0` → Yes, TCP
socket() Code Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

int main(int argc, char* argv[]) {
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    return EXIT_SUCCESS;
}
```
bind()

#include <sys/socket.h>

int bind(int sock, const struct sockaddr* addr, socklen_t addrlen);

After creating a socket as described, then create a sockaddr struct.
bind() Code Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[]) {
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1025);
    addr.sin_addr.s_addr = INADDR_ANY;
    int err = bind(sock, (struct sockaddr *) &addr, sizeof(addr));
    return EXIT_SUCCESS;
}
```
listen()

#include <sys/socket.h>

int listen(int sock, int backlog);

After bind()ing, you can listen() for connections.
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[])
{
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1025);
    addr.sin_addr.s_addr = INADDR_ANY;
    int err = bind(sock, (struct sockaddr *)
                   &addr, sizeof(addr));
    int err2 = listen(sock, 5);
    return EXIT_SUCCESS;
}
backlog

- Queue length for incoming sockets
- Fully established already
accept()

#include <sys/socket.h>

int accept(int sockfd, struct sockaddr * addr, struct socklen_t * len);

After `listen()`ing, you can `accept()` connections. Pass in the socket from before, and pointers to data structures defined by you. These represent connected client state for future use.
accept() Code Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[]) {
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr, caddr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1025);
    addr.sin_addr.s_addr = INADDR_ANY;
    int err = bind(sock, (struct sockaddr *) &addr, sizeof(addr));
    int err2 = listen(sock, 5);
    socklen_t len = sizeof(caddr);
    int client = accept(sock, (struct sockaddr *) &caddr, &len);
    return EXIT_SUCCESS;
}
```
recv()

#include <sys/socket.h>

int recv(int sockfd, void * buf, size_t len, int flags);

After accept()ing, you can recv() data. For now set flags to 0
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[]) {
    char buf[256];
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr, caddr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1025);
    addr.sin_addr.s_addr = INADDR_ANY;
    int err = bind(sock, (struct sockaddr *)&addr, sizeof(addr));
    int err2 = listen(sock, 5);
    socklen_t len = sizeof(caddr);
    int client = accept(sock, (struct sockaddr *)&caddr, &len);
    ssize_t read = recv(client, buf, 256, 0);
    return EXIT_SUCCESS;
}
#include <sys/socket.h>

int send(int sockfd, void * buf, size_t len, int flags);

After `accept()`ing, you can `send()` data.
For now set flags to 0
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[]) {
    char buf[256];
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr, caddr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1025);
    addr.sin_addr.s_addr = INADDR_ANY;
    int err = bind(sock, (struct sockaddr *) &addr, sizeof(addr));
    int err2 = listen(sock, 5);
    socklen_t len = sizeof(caddr);
    int client = accept(sock, (struct sockaddr *) &caddr, &len);
    ssize_t sent = send(client, buf, 256, 0);
    return EXIT_SUCCESS;
}
How to: Server

- Create a socket via `socket()`
- Bind to an endpoint via `bind()`
- Listen for connections via `listen()`
- Accept connections via `accept()`
- Read from socket via `recv()`
- Write to socket via `send()`

Source `<ip,port>` + Destination `<ip,port>`
How to: Client

- Create a socket via `socket()`
- Connect to an endpoint via `connect()`
- Read from socket via `recv()`
- Write to socket via `send()`
connect()

#include <sys/socket.h>

int connect(int socket, const struct sockaddr *serv_addr, socklen_t protocol);

Use socket as before, get serv_addr from getaddrinfo().
Free with freeaddrinfo().
connect() Code Example

#include <netdb.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>

int main(int argc, char* argv[])
{
    struct addrinfo addr, *caddr;
    memset(&addr, 0, sizeof(addr));
    addr.ai_family = AF_INET;
    addr.ai_socktype = SOCK_STREAM;
    getaddrinfo("www.google.com", "80", &addr, &caddr);
    int sock = socket(caddr->ai_family, caddr->ai_socktype, caddr->ai_protocol);
    connect(sock, caddr->ai_addr, caddr->ai_addrlen);
    return EXIT_SUCCESS;
}
How to: Client

- Create a socket via `socket()`
- Connect to an endpoint via `connect()`
- Read from socket via `recv()`
- Write to socket via `send()`

Source `<ip,port>` + Destination `<ip,port>`
Socket Programming Gotchas

- **Endianness Matters: Network Byte Order**
  - htons() - host to network short
  - ntohss() - network to host short

- **Cleanup state—avoid memory leaks**
  - freeaddrinfo()
  - Check correctness with **valgrind**

- **Error Handling**
  - Tedious, but worth it (and required!)

- **Timeouts**
  - Implement for robust networking behavior
Socket Programming Gotchas

- **Never expect to** `recv()` **what you** `send()`
  - Assume partial receipt of data possible
  - Use buffers intelligently to mitigate this
  - Send byte counts first, read until finished
- **Prepare your code for random failures**
  - We introduce random faults when grading
  - Test too—`ctrl+c` server and client randomly
- **Cleanup Allocated Memory**
  - `close()` sockets, etc.
Concurrency

• Threads
  • Server gives each client its own thread
  • Not in this class!

• `select()`
  • Watch a set of sockets (in main thread)
  • Use `select()` to find sockets ready for I/O
  • Server-side only—clients are agnostic
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);  // removes fd from set
int FD_ISSET(int fd, fd_set *set);  // is fd in set?
void FD_SET(int fd, fd_set *set);   // adds fd to set
void FD_ZERO(fd_set *set);    // clears set

Manipulate set of descriptors with FD_*, then select().
Select() Usage - Example

//Other inits
fd_set readfds;
// pretend we've connected both to a server at this point
//s1 = socket(...);
//s2 = socket(...);
//connect(s1, ...);
//connect(s2, ...);
// clear the set ahead of time FD_ZERO(&readfds);
// add our descriptors to the set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);
// since we got s2 second, it's the "greater", so we use that for
// the n param in select() n = s2 + 1;
// wait until either socket has data ready to be recv'd
rv = select(n, &readfds, NULL, NULL, &tv);
if (rv == -1) {
    perror("select"); // error occurred in select()
} else {
    // one or both of the descriptors have data
    if (FD_ISSET(s1, &readfds)) {
        recv(s1, buf1, sizeof buf1, 0);
    }
    if (FD_ISSET(s2, &readfds)) {
        recv(s2, buf2, sizeof buf2, 0);
    }
}
Error Checking

• Read documentation first
• Sometimes you need to:

```c
#include <errno.h>
...
switch (errno) {
  ...
}
```
More on Project1
Reading data

- Check return value of `recv()`
  - Error – handle the error and clear up state
  - If peer shutdown connection, clear up state
- Maintain state
  - Maintain a read buffer
  - Keep track of number of bytes to read
  - May need multiple reads to get all data
  - Only one read per socket when `select()` returns
Writing date

• Check return value of `send()`
  • Error – handle the error and clear up state
  • If peer shutdown connection, clear up state

• Maintain state
  • Maintain a write buffer
  • Keep track of number of bytes to write
  • May need multiple writes to send all data
Remember

- Code quality
- Robustness
  - Handle errors
  - Buffer overflows
  - Connection reset by peer
Common mistakes

• Make sure your executable is named correctly
• tar your complete repo and submit. Autolab expects to find a .git folder with your submission
• Don't forget to update the tag when you make changes to your code. We run a checkout with the tag name and not your last commit
Checkpoint 1 docs

- **Makefile** – make sure nothing is hard coded specific to your user; should build a file which runs the echo server (must be named lisod)
- **All of your source code** – all .c and .h files
- **readme.txt** – file containing a brief description of your current implementation of server
- **tests.txt** – file containing a brief description of your testing methods for server
- **vulnerabilities.txt** – identify at least one vulnerability in your current implementation
Questions???
System Call Documentation:

**POSIX** – Portable Operating System Interface for Unix

*IEEE 1003.1-2008*, The Open Group

“POSIX.1-2008 defines a standard operating system interface and environment, including a command interpreter (or “shell”), and common utility programs to support applications portability at the source code level.”

http://pubs.opengroup.org/onlinepubs/9699919799/

Also, more correct, *your system's man pages!*
Another excellent resource:
Beej's Guide to Network Programming
http://beej.us/guide/bgnet/