

ECE 435 – Network Engineering

Lecture 3

Vince Weaver

`https://web.eece.maine.edu/~vweaver`

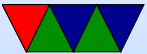
`vincent.weaver@maine.edu`

Barrows 123, Mon 10:00am

28 January 2026

Announcements

- Homework 1 was posted
- Schedule off a bit due to snow day



TIME_WAIT

- If you quit and immediately try to restart server might get error saying socket busy.
- Spec says you should wait a minute for all packets to clear out. You can wait, or can force with

```
int on=1;    /* we want to turn the feature on */
setsockopt(s, SOL_SOCKET, SO_REUSEADDR,
           (char *)&on, sizeof(on));
```



Client Code

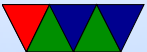
- You'll find it's much more straightforward than the server code
- Mostly because you don't have to handle the possibility of multiple simultaneous connections
- Very similar to telnet code of yore



socket() again

```
/* Open a socket file descriptor */
/* AF_INET means an IP network socket, not a local (AF_UNIX) one */
/* There are other types you can open too */
/* SOCK_STREAM means reliable two-way byte stream (TCP) */
/* last argument is protocol subset. We leave at zero */

socket_fd = socket(AF_INET, SOCK_STREAM, 0);
if (socket_fd < 0) {
    fprintf(stderr, "Error socket: %s\n",
            strerror(errno));
}
```



Lookup Address by Name

- We need address of the server we want to connect to
- Note for this example using “localhost”
- This is a special case, 127.0.0.1 (loopback, local machine) on IPv4.

```
#define DEFAULT_HOSTNAME "localhost"
struct hostent *server;

/* Look up the server info based on its name */
server=gethostbyname(DEFAULT_HOSTNAME);
if (server==NULL) {
    fprintf(stderr, "ERROR!  No such host!\n");
    exit(0);
}
```



gethostbyname() Notes

- Note that `gethostbyname()` is deprecated
- TODO: should update this to use `getaddrinfo()` instead
- You can see from above part of the problem. Where is the memory for holding the server data?
- It lives in the C library, a single copy allocated once, and it's not thread safe



Set Destination Address / Port

```
struct sockaddr_in server_addr;

/* clear out the server_addr structure and set some fields */
/* Set it to connect to the address and port of our server */
memset(&server_addr,0,sizeof(server_addr));

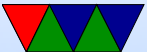
/* Copy in the address from the previous name lookup (already network byte order) */
memcpy(server->h_addr,&server_addr.sin_addr.s_addr,
        server->h_length);

/* port should be in "network byte order" (big-endian) */
/* htons = host to network [byte order] short */
server_addr.sin_port=htons(port);
server_addr.sin_family=AF_INET;
```



connect system call

```
/* Call the connect system call to actually connect to server */  
if (connect(socket_fd, (struct sockaddr *) &server_addr,  
    sizeof(server_addr)) < 0) {  
    fprintf(stderr, "Error connecting! %s\n",  
        strerror(errno));  
}
```



Get Input from Keyboard to Send

An aside on C strings. Remember they need to be NUL (0) terminated. This code below zeros the whole buffer and then reads a maximum of size-1 bytes which ensures that happens.

```
/* Prompt for a message */  
printf("Please enter a message to send: ");  
memset(buffer,0,BUFFER_SIZE);  
  
/* Read message */  
fgets(buffer,BUFFER_SIZE-1,stdin);
```



Send message to server

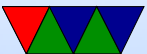
- Note, to get the length of a C string use `strlen()` not `sizeof()`

```
/* Write to socket using the "write" system call */
n = write(socket_fd,buffer,strlen(buffer));
if (n<0) {
    fprintf(stderr,"Error writing socket! %s\n",
            strerror(errno));
}
```



Do we need to NUL Terminate when sending?

- Do we need to NUL terminate? This depends if our “protocol” needs it
- The server should NUL terminate anything it gets though if treating it as a string or bad things can happen
- You can't always trust that the input you get from the network is properly written (or not intentionally malicious)



wait for response with read()

This code again zeros the whole buffer and then reads a maximum of size-1 bytes which ensures NUL termination. You can also explicitly do a `buffer[n]=0;` after the error check to be safer

```
/* Clear buffer and read the response from the server */
memset(buffer,0,BUFFER_SIZE);
n = read(socket_fd,buffer,BUFFER_SIZE-1);
if (n<0) {
    fprintf(stderr,"Error reading socket! %s\n",
            strerror(errno));
}
```

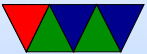


```
/* Print the response we got */  
printf("Received back from server: %s\n\n",buffer);
```



close again

```
/* All finished, close the socket/file descriptor */  
close(socket_fd);
```



Client Summary

- `socket()` – sets up socket
- set up address/port
- `connect()` – connects to server
- `send()/write()` – send data
- `recv()/read()` – get data
- `close()` – close socket



HW#1 Notes

- Make the server loop forever until a string comes in.
- How do you loop forever?
- How do you compare with a string? Can you use `==`
Be careful with `strcmp()`
Unlike most C functions returns 0 if match
You might even want to use `strncmp()`
- Comment your code!
- Try to fix all compiler warnings!



HW #1 notes – Debugging

- `strace` can be useful when tracking down issues and showing what syscalls are doing
- `netstat` on a Linux machine can show what network connections are active, including ports and addresses
- `ss` (socket status) is the more modern tool people use instead of `netstat`



HW #1 notes – Socket Programming

- Finding the “struct sockaddr” can be difficult. even if you find it under /usr/include it’s tricky as it’s a struct that is multiplexed via casting (to handle all possible socket types). Horrible thing about C.



HW #1 notes – Read/Write Issues

- A lot of this comes down to C, and it treats streams of bytes and strings as mostly interchangeable, even though there are a lot of pitfalls with that
- Some people have issues where they are writing 256 bytes (`write()` will write as many bytes as you said, even if they are trailing zeros), but only reading 255. This means the next read is going to get the last 0 rather than the following write.
- When reading, `read(fd,buffer,size);` What



happens if you read 10 bytes but other side only has 4? Only read 4 (result).

- What happens read 10 bytes and other side has 12? You read 10 (result) but to get the rest you need to read again, otherwise it's there the next time you read. You can do a while loop.
- Also note that when you write, you should specify how many bytes you are writing, don't just write the full `BUFFER_SIZE` as you'll send all the extraneous data in the buffer past the part you actually want
- Also be careful as by default read/write are blocking so



if there is no data available it will wait forever until some
is



HW #1 notes – Buffer Management

- Why not just `malloc()` each buffer to the exact size as needed?
- The famous reply: “Now you have two problems”
- C manual memory management is almost as much of a security problem as NUL-strings are
- It’s also inefficient. But on modern GHz machines with GB of RAM maybe that doesn’t matter.



HW #1 notes – Port Numbers

- Incoming port from the client isn't going to be same as the listening port on the server.
- The OS will pick a random, high value (usually in the 40000+ range) for outgoing connections
- Also if you see impossible or unlikely port numbers, be sure you are remembering to use `htons()` and `htonl()` to swap back from network byte order



Traditional Internet Servers

- Often Client/Server
- Server “daemon”
- Listens on port
 - IANA (Internet Assigned Name Authority) “well-known” ports 0-1023
 - Registered (reserved) ports: 1024-49151
 - Dynamic/Private 49152-65535
- Start at boot time? On demand? Old days inetd, these days systemd



Server Types

- Concurrent – handle multiple connections at time (forks or threads)
 - Concurrent Connectionless – when need fast turnaround, low latency DNS, NFS
 - Concurrent Connection – widely used. WWW.
- Iterative – handles one connection at a time, rest wait on queue
 - Iterative Connectionless – common+trivial, short lived
 - Iterative Connection – high latency



Protocols

- What type of protocol should talk?
- Fixed-length binary?
- Free-form ASCII text?
- 7-bit ASCII vs Unicode?
- Encrypted or compressed?
(security issues from compression?)



Internet, Pre-Web

- These days if someone says internet, often mean WWW
- What was the world like before the web?
 - Mostly Text based (simple, some networks not handle binary)
 - Low-speed, high-latency connections
- Linux machines in 90s came pre-installed with all kinds of servers running
 - e-mail, ftp, usenet, telnet
 - chargen / time / motd / fortune



○ finger / talk / write



What changed?

- Commercial companies allowed on
- Eternal September
- Endless security issues

