

The problem when you could not connect to unix sockets server was because of the *tmp* folder permissions.

Let's test it with a changed folder:

```
nc -U /home/mbarsky/tmp/something
```

Internet Sockets

Lecture 08.03

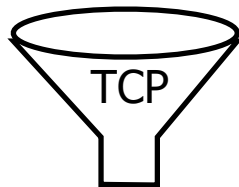
Internet protocol: 2 layers – TCP/IP

- TCP breaks data into packets and give each packet a header:
 - sequence number
 - checksum
- IP – adds envelope with IP addresses

source address		dest. address
bytes	ack	port
data		

Preparing stream of data for transmission

01100111001001
00100010001111
10100010111 ← Data stream



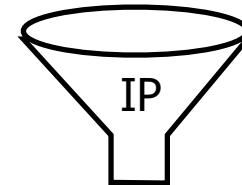
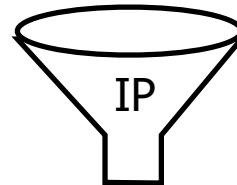
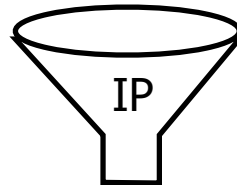
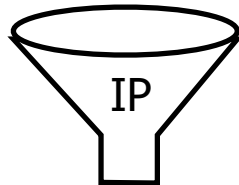
TCP: make
packets

101010001
111010101
100110010
110101111
001011011

101010001
111010101
100110010
110101111
001011011

101010001
111010101
100110010
110101111
001011011

101010001
111010101
100110010
110101111
001011011



put in an
IP envelope
with another
header

To
24.197.0.67

To
24.197.0.67

To
24.197.0.67

To
24.197.0.67

IP addresses

- The IP host address is used to uniquely identify machines connected to the Internet
- It is a 32-bit quantity interpreted as 4 8-bit numbers or octets (IPv4)
- An IP address is usually written in a dotted-decimal notation of the form $N_1.N_2.N_3.N_4$, where each N_i is a decimal number between 0 and 255
- Because of the growth of the Internet and the depletion of available IPv4 addresses, a new version of IP (IPv6), using 128 bits for the IP address, was developed in 1995

Host names are mapped to unique string names

- Host names in terms of numbers are difficult to remember and hence they are termed by ordinary names such as *google.com* or *yahoo.com*
- To connect, we need to find out the dotted IP address corresponding to a given name
- The process of finding out dotted IP address from host name is known as *hostname resolution*
- A hostname resolution is done by special software residing on Domain Name Servers (DNS): they keep the mapping of IP addresses and the corresponding ordinary names

Name, address, route

- An IP address serves two functions:
 - identifies the host, or more specifically its network interface
 - provides the location of the host in the network, and thus the capability of finding a path (route) to that host
- "A *name* indicates what we seek. An *address* indicates where it is. A *route* indicates how to get there."
- The header of each IP packet contains the IP address of the sending host, and that of the destination host
- DNS server resolves the address and finds the route to it on the network

In C we can get the real numeric host address with *getaddrinfo()*

Sample code in *showip.c*

run

`./showip <str_host_name>`

To find IP **address** of your machine:

`ifconfig`

or another system-specific command- `ipconfig` for Windows

To find your **hostname**:

`hostname`

Identifying the process on a host machine with **port**

- If the client knows the 32-bit Internet address of the host machine, it can contact that host
- To identify the **particular server process** running on that host we define a ***port number***
- New port number should be an integer between 1024 and 65535:
 - Port numbers **smaller than 1024** are considered **well-known** (telnet on port 23, http on port 80, ftp on port 21 etc.)
 - You can see port assignments in file `/etc/services`:
more /etc/services
 - In your own application you need to make sure that your port is not assigned to any other service (Any port number more than 5000 is a good choice)

Let's build internet socket echo server

The same

Bind

Listen

Accept

Sample code in:
server_inet.c
client_inet.c

socket()

- This is the same as before, except it says "INET" instead of "UNIX"

```
if ((serv_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {  
    perror("socket");  
    return(1);  
}
```

Defining address and port for INET server socket

```
struct sockaddr\_in serv_addr;  
memset(& serv_addr, '\0', sizeof (serv_addr));  
serv_addr.sin_family = AF_INET;
```

```
serv_addr.sin_addr.s_addr = INADDR_ANY;
```

This means that if there are more than one IP address for this machine, use any of them

```
serv_addr.sin_port = htons(12345);
```

Note the use of *htons* (host-to-network-short) which converts a given port number to a network format

1. Bind

- For AF_INET sockets, there is no filesystem token representing them - only a list of bound ports/IPs kept track of by the kernel
- Looks exactly the same as with Unix domain sockets

struct `sockaddr_in` serv_addr; //all set up above

Here we are casting again to struct `sockaddr` – so it can compile, but the address structure itself is quite different

```
if (bind(serv_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)))  
{  
    perror("bind");  
    return(1);  
}
```

Returns zero on success

2. Listen

```
if (listen(serv_fd, 5)) {  
    perror("listen");  
    return(1);  
}
```

3. **A**ccept

```
struct sockaddr_in client_addr;
```

```
int len = sizeof (client_addr);
```

```
if ((client_fd = accept(serv_fd, (struct sockaddr *)&client_addr,  
                        &len)) < 0) {
```

```
    perror("accept");
```

```
    return(1);
```

```
}
```

```
printf("connection from %s\n", inet_ntoa(r.sin_addr));
```


Important: server ports are sticky

- If you terminate your server program, and then start it again, you may receive the following error message when calling `bind()`:

Can't bind the port: Address already in use

- When you bind a socket to a port, the operating system will prevent anything else from rebinding to it for the next 30 seconds or so, and that includes the program that bound the port in the first place.
- To get around the problem, you need to set an option on the socket before you bind it:

```
int reuse = 1;
```

```
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR,  
                (char *)&reuse, sizeof(int)) == -1)  
    error("Can't set the 'reuse' option on the socket.");
```

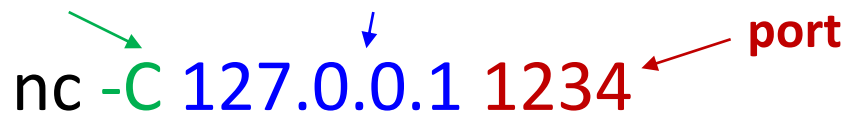
Let's run echo server and connect to it from multiple clients

- You can connect with general netcat client:

Send `\r\n` as
line ending

IP **address**
of a server

nc -C 127.0.0.1 1234



nc -C src-code.simons-rock.edu 1234

Domain **name**
of a server

- You can write your own client - *client_inet.c*:
`gcc client_inet.c -o client && ./client`

Internet Stream sockets (TCP): checklist

Server

- Create a socket: **socket()**
- Create an address variable and fill in the fields: **port** number
Make sure to set option REUSE_ADDRESS
- Bind socket to an address: **bind()**
- Establish a queue for connections and start listening: **listen()**
- Get a connection from the queue: **accept()**

Client

- Create a socket: **socket()**
- Initiate a connection: **connect()**

Multi-client socket server: blocking

Blocking server: `server_select.c`

Blocking

- A *blocking* call does not return to the next line of your program until the event you requested has been completed
- Most of system calls in socket programming are blocking:

Server: `accept()`, `read()`, `write()` are blocking

Client: `connect()`, `read()`, `write()` are blocking

Server:

accept() and read() are blocking

```
while (1) {
```

While waiting for a new client to connect – cannot not do anything else

```
if ((clientfd = accept(fd, (struct sockaddr *)&q, &len)) < 0)  
    error("accept");
```

While waiting for an accepted client to write something – cannot accept new clients

```
if ((len = read(clientfd, buf, MAX_LINE)) < 0)  
    error("read");  
}
```

Avoiding blocking in complex programs

- For simple programs, blocking is convenient
- What about more complex programs?
 - multiple connections
 - simultaneous reads and writes
 - simultaneously doing non-networking processing

Ways to handle multiple clients without blocking

- Forking a child process for each client
- Processing each client in a separate thread (not covered)
- Using `poll()` (not covered)
- ➔ • Using *`select()`*

Select()

- **Problem:** from which socket the server should accept connections or receive messages?
- **Solution:** `select()`
 - specifies a list of descriptors to check for pending I/O operations
 - blocks until one of the descriptors is ready (or timeout)
 - returns which descriptors are ready

Preparing file descriptor sets

- Populate sets of socket file descriptors you are interested in using macros
- Once you have the set, you pass it into the function as one of the following parameters:
 - *readfds* if you want to know when there is something to read from these file descriptors (client contacted server)
 - *writefds* if any of the file descriptors wants to write() data
 - *exceptfds* if you need to know when an exception (error) occurs on any of the sockets
- Any of these parameters can be NULL if you're not interested in those types of events

Example: preparing fd sets

<code>FD_SET(int fd, fd_set *set);</code>	Add fd to the set.
<code>FD_CLR(int fd, fd_set *set);</code>	Remove fd from the set.
<code>FD_ISSET(int fd, fd_set *set);</code>	Return true if fd is in the set.
<code>FD_ZERO(fd_set *set);</code>	Clear all entries from the set.

```
fd_set readfds;
```

```
// pretend we've have socket fds for two clients at this point: s1 and s2
```

```
FD_ZERO(&readfds);
```

```
FD_SET(s1, &readfds);
```

```
FD_SET(s2, &readfds)
```

Select: parameters

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

- The first parameter, *n* is the highest-numbered file descriptor to check - plus one
- The last parameter *timeout* tells *select()* how long to check these sets for before moving on
- Select returns after the timeout, or when any of the file descriptors generated an event, whichever is first

Example: select parameters

- Suppose $s2 > s1$, so we use it for n :

$n = s2 + 1;$

- Wait until either socket has data ready to be read (with timeout 10.5 secs)

`struct timeval tv;`

`tv.tv_sec = 10; //seconds`

`tv.tv_usec = 500000; //microseconds (1,000,000 microseconds in a second)`

`int rv = select(n, &readfds, NULL, NULL, &tv);`

Select return value

- Returns the number of ready descriptors in the set on success, 0 if the timeout was reached, or -1 on error
- After `select()` returns, the values in the sets will be changed to show which are ready for reading or writing, and which have exceptions

Return value: example

```
rv = select(n, &readfds, NULL, NULL, &tv);

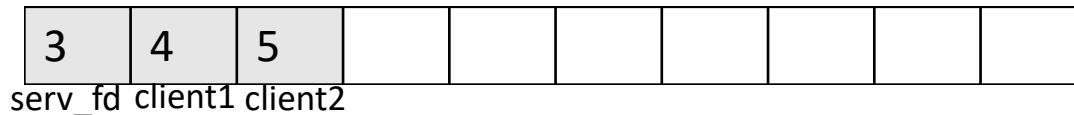
if (rv == -1) {
    perror("select"); // error occurred in select()
} else if (rv == 0) {
    printf("Timeout occurred! No data after 10.5 seconds.\n");
} else {
    // one or both of the descriptors have data
    if (FD_ISSET(s1, &readfds))
        read(s1, buf1, sizeof buf1);
    if (FD_ISSET(s2, &readfds))
        read(s2, buf2, sizeof buf2);
}
```

Select diagram

readfds - empty



Set fds of interest



Max fd: 5

Call to **select**(&

3	4	5							
---	---	---	--	--	--	--	--	--	--

)
waits for activity on any of readfds

Select returned 2: activity on 2 fds



New client sent
connect request

Client 2 sent
data to read

Full code for echo server with select()

```
while (1) {
    fd_set fdlist;
    maxfd = serv_fd;
    FD_ZERO(&fdlist);
    FD_SET(serv_fd, &fdlist);
    for (p = client_list; p; p = p->next) {
        FD_SET(p->fd, &fdlist);
        if (p->fd > maxfd)
            maxfd = p->fd;
    }

    if (select(maxfd + 1, &fdlist, NULL, NULL, &tv) < 0) {
        error("select");
    } else {
        for (p = client_list; p; p = p->next)
            if (FD_ISSET(p->fd, &fdlist))
                break;

        if (p)
            handle(p->fd);
        if (FD_ISSET(serv_fd, &fdlist))
            newconnection(serv_fd);
    }
}
```

```
typedef struct client {
    int fd;
    struct client *next;
    struct client *previous;
}Client;
```

Doubly-linked list to store client fds

Code: *server_nonblocking.c*

```
nc -C 127.0.0.1 8888
```

Commands allowed:

`list_users`

`post` to_username message

`quit`

Multi-client chat server

Preparation for Assignment 4

Code in folder *chat_server*

General program flow

- Setup server socket
- In a loop: call select
- If serv_fd fired:
 - newconnection
 - add client to list of clients
 - ask him for a name
 - Set client status to NAME (not allowed to do commands before identified himself)
- If any of client_fds fired:
 - handle

Handling client commands

- Read bytes sent from the client
- If `nbytes read > 0`:
 - parse command and execute
- Else:
 - there was activity on this fd but no bytes were sent – the only possibility is client disconnected
 - removeclient