CSCI 4061: Sockets and Network Programming

Chris Kauffman

Last Updated:
Mon Apr 26 03:57:32 PM CDT 2021

Logistics

Reading

Stevens/Rago Ch 16

Goals

- Finish up Threads
- Sockets Basics
- Servers and Clients

Project 2

- Tests tomorrow
- Questions?

Date	Event
Mon 4/26	Threads/Sockets
Wed 4/28	Sockets
Mon 5/03	Lecture: Review Lab: Review P2 Due
Mon 5/10	Final Exam 10am - 10pm CST 4-6pm Questions

Questions on anything?

Reminder: Course Evals

```
CSCI 4061    : Intro to Operating Systems
Lecture 001 : Kauffman
```

- Official UMN Evals are done online this semester
- Available here: https://srt.umn.edu/blue
- Due Mon 5/03/2021, last day of summer semester

Overview

- Computer Networks are their own topic/course, we won't go into great detail
- Communication programs usually require (1) understanding of OS concepts and (2) **Network Protocols**: how to talk and what to say when working across a connection
- We will demonstrate a few facilities that combine these 2 concepts
- All of you are aware that Computers are NOT isolated anymore: constantly talking to each other across a variety of connection types
- Up against several technical challenges when discussing Network Programming
  1. Concepts in network programming are advancing rapidly
  2. Examples that worked in the recent past may not work now
  3. Previous techniques/protocols are quickly supplanted by new (hopefully better) ones

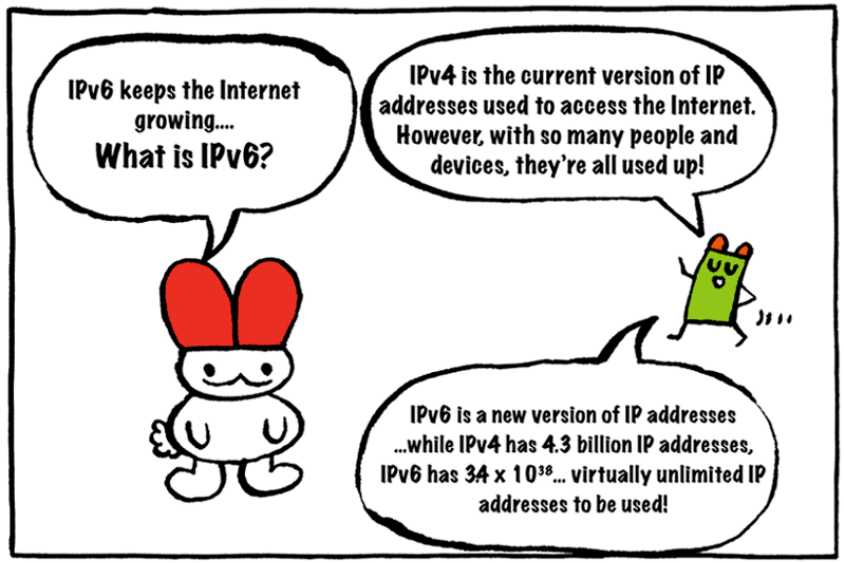
Goals

- Give a few examples of the Unix interface to network programming via **sockets** and ports to set up simple server-client
- Relate abstraction to previous I/O experience
- Touch on a few network-specific details, underlying details
- Leave the full she-bang to CSCI 4211 (Intro Networking)

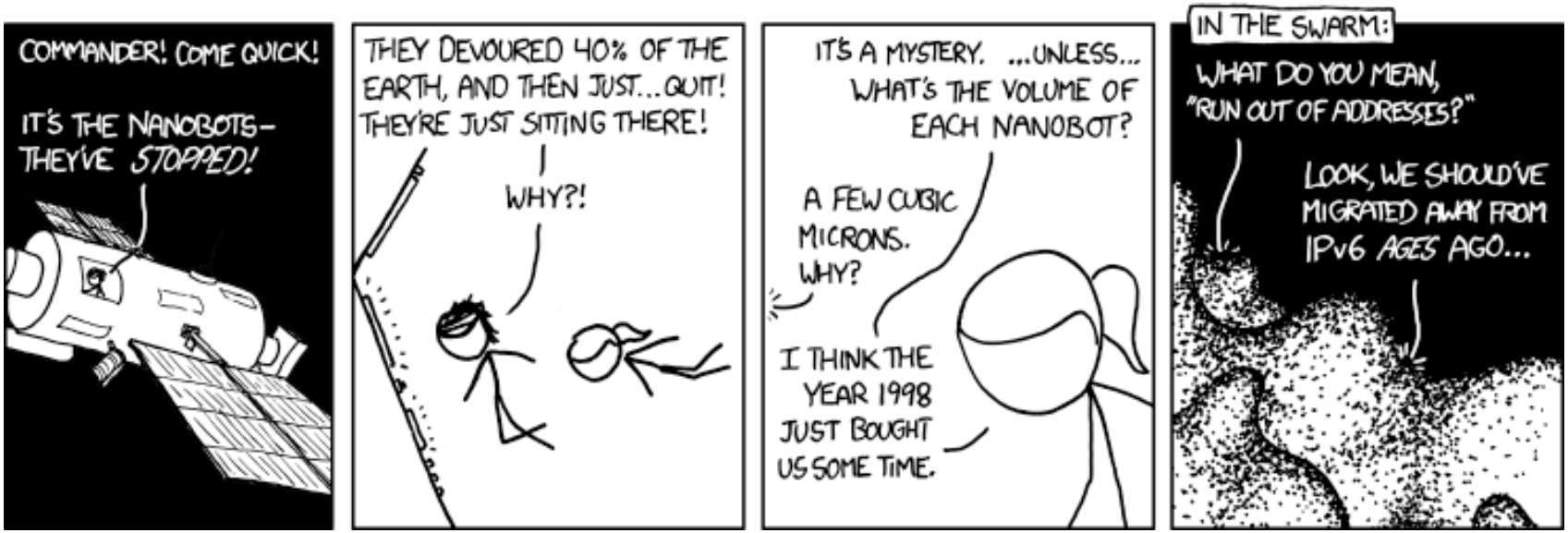
Aging Networks Makes Network Programming a Mess

- Due to Internet technology advancing, network programming has changed so there are MANY historical relics
- Network is a physical connection but many **protocols** for communication exist over the same network to fulfill different needs
- There are a LOT of network functions, some of them are **deprecated** or **obsolete**: don't handle newest protocols/electronics
  - `gethostbyname()` simple, only works with IPv4
  - `getaddrinfo()` complex, works with IPv6

Networks are Aging



Source: www.ipv6now.hk



Source: XKCD #865

Immediate Limitations

- Most networked computing resources use **Firewalls** to block most communications
- Firewall prevents internal programs from connecting to outside programs through unauthorized ports
- Makes programming examples a little tough but can do local examples using address `127.0.0.1` which is IPv4 for "home"
- Would need to run your own machine to open up ports to the whole web

Historically true, but these days "There's no place like `::1`" is more accurate.

Sockets

- An abstraction like files, a number referring to OS internal data structures
- Allow for communication with the outside world
- Sockets represent end-to-end connection: two parties involved
- Sockets are two-way: can read or write from them (like files)
  - Writes send data over the network to other party
  - Reads block a process until data is received over network from other party
- Sockets give a two-way "stream" of data like FIFOs: can't `lseek()` for either reads or writes

Addresses

To communicate over the network, must use functions to translate addresses from plain text like "google.com" to binary IP addresses.

```
char *hostname = "127.0.0.1"; // or "google.com"
struct addrinfo *servinfo;
int ret = getaddrinfo(hostname, PORT, NULL, &servinfo);
if(ret != 0){
  printf("getaddrinfo failed: %s\n",
         gai_strerror(ret));
  exit(1);
}
```

Note that the address `127.0.0.1` is IPv4 for "this computer" and will be used a lot in examples

addrinfo struct

```
struct addrinfo {
  int              ai_flags;
  int              ai_family;
  int              ai_socktype;
  int              ai_protocol;
  socklen_t        ai_addrlen;
  struct sockaddr *ai_addr;
  char            *ai_canonname;
  struct addrinfo *ai_next;
};
```

- Notice the last field - **what kind of data structure is** `addrinfo`?
- `getaddrinfo(hostname, PORT, NULL, &servinfo);` may return multiple addresses which can all be tried to get the connection

Socket Creation / Connection

```
struct addrinfo *servinfo; // filled by getaddrinfo()
int sockfd = socket(servinfo->ai_family,
                    servinfo->ai_socktype,
                    servinfo->ai_protocol);
```

- Allocates OS internal data structures for 2-way communication
- **Does not** connect socket for communication yet

```
int ret = connect(sockfd,
                  servinfo->ai_addr,
                  servinfo->ai_addrlen);
```

- Connects socket to given address to allow data send/receive
- Server on other side must be listening

If all goes well...

```
printf("Sending 'hello' to server\n");
char *msg = "hello";
write(sockfd, msg, strlen(msg));

char buf[MAXDATASIZE];
int nread = read(sockfd, buf, MAXDATASIZE-1);
buf[nread] = '\0';
printf("client: received '%s'\n",buf);
```

How dull: it's just another fd to `read()` / `write()`

Alternatively

```
int nwrite = send(sockfd, msg, strlen(msg), 0);
int nread  = recv(sockfd, buf, MAXDATASIZE-1, 0);
```

allows additional sending / receiving options over the socket.

Experiment with `simple_client.c`

- Requires `simple_server.c` to be running (discussed later)
- Client connects to server on local computer and receives a `hello world`

read() / recv() and write() / send()

- Socket file descriptors can be treated just as others so that standard I/O calls like `read() / write() / select()` work for them
- Thus Network communication via sockets has an **identical interface** to other files
- Alternatively can use `recv()` to get data from a socket fd
  Allows options like

  `MSG_PEEK` `Peeks at an incoming message. The data is treated as unread and the next recv() or similar function shall still return this data.`
- Alternatively use `send()` to put data into a socket fd
  Sample options

  `MSG_DONTWAIT` `Enables nonblocking operation`

Exercise: Servers and Sockets

- Have discussed the client side of sockets:
  - get address
  - make socket
  - connect socket and address
  - read() / write()
- Server side has a few more tricks to it
- Multiple clients must connect using the same address, e.g.

  `www.google.com PORT 80`
- **What kind of problems** might this present?
- How might one solve this with a system design?

Answer: Servers and Sockets

- Servers use one socket to **listen** for connections
- All incoming clients initially establish a connection through that socket with a known port #
- When a client connects, a **second server socket** is created which is specific to the client
- Communication between server and client continues on the second separate socket
- **Sound like anything familiar?**

Server Setup

```
// INITIAL SETUP

// fd of socket on which the server will listen
int listen_fd = socket(serv_addr->ai_family,
                       serv_addr->ai_socktype,
                       serv_addr->ai_protocol);

// bind the socket to the server address given
// allows listening for connections later on
ret = bind(listen_fd,
           serv_addr->ai_addr,
           serv_addr->ai_addrlen);
```

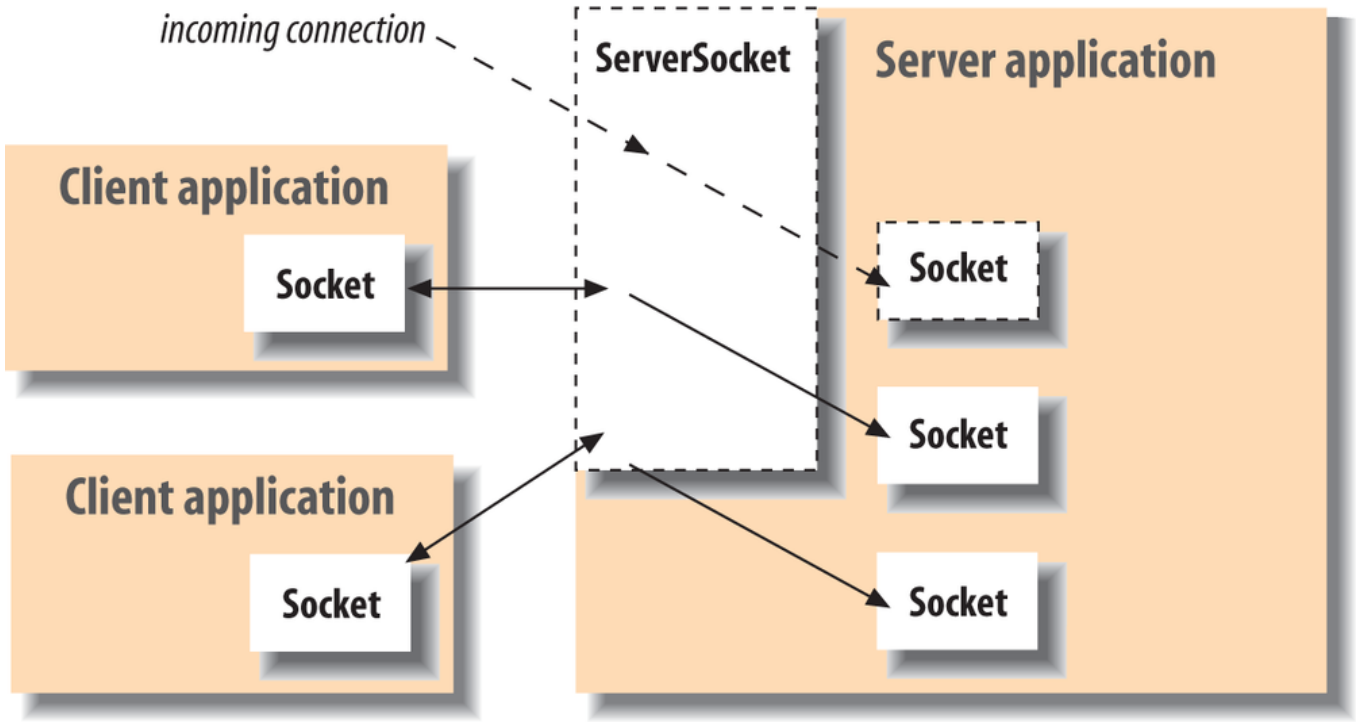
Server Main Loop

```
// MAIN LOOP
listen(listen_fd, BACKLOG);

while(1){
  // block until a client tries to connect
  // accept a connection from the open port from a
  // client produces a new file descriptor for
  // socket created to communicate with the client
  // and fills in client address info
  int client_fd = accept(listen_fd,
                         client_addr,
                         &client_addr_size);

  read(client_fd, ...);
  write(client_fd, ...);
}
```

Sockets On server Side



Source: Learning Java, 4th Edition by Patrick Niemeyer, Daniel Leuck

- Each call to `accept()` creates another socket associated specifically with a **peer**
- Typically done on by server in client/server architecture
- Single server Port stays open and accepts new connections

Socket Identification

Based on: SO: How does the socket API accept() function work?

Sockets are uniquely identified by a quartet of information:

```
| Local Address : Port | Peer Address : Port |
```

Example

- Server at 192.168.1.1 Port 80, accepting connections
- Client 1 10.0.0.1 Port 1234, connects to server
- Client 2 15.3.7.9 Port 5678, connects to server

```
SERVER KERNEL SOCKET TABLE
| Local (Server)   | Peer (Clients)  |
|------------------+-----------------|
| 192.168.1.1 : 80 | 10.0.0.1 : 1234 |
| 192.168.1.1 : 80 | 15.3.7.9 : 5678 |

CLIENT 1 KERNAL SOCKET TABLE
| Local (Client1) | Peer (Server)    |
|-----------------+------------------|
| 10.0.0.1 : 1234 | 192.168.1.1 : 80 |

CLIENT 2 KERNEL SOCKET TABLE
| Local (Client2) | Peer (Server)    |
|-----------------+------------------|
| 15.3.7.9 : 5678 | 192.168.1.1 : 80 |
```

Handy Network Commands

Kernel tracks all sockets/connections, can report on command line

```
## I have an ssh connection to apollo, show find evidence of this

## ss: show open ports with by quartet with stats
> ss -tuna
Netid State   Recv-Q Send-Q   Local Address:Port  Peer Address:Port

tcp   ESTAB   0      0          10.0.0.187:44354  128.101.38.191:22
...                                                           ^^

## getent: lookup addresses by name/number
> getent hosts 128.101.38.191
128.101.38.191  csel-apollo.cselabs.umn.edu

> getent hosts apollo.cselabs.umn.edu
128.101.38.191  csel-apollo.cselabs.umn.edu apollo.cselabs.umn.edu

## lsof: list open files, -i for internet files
> lsof -i
COMMAND    PID     USER   FD   TYPE  DEVICE SIZE/OFF NODE NAME
chromium 22900 kauffman   99u  IPv4 9369301      0t0  UDP *:mdns
chromium 22900 kauffman  191u  IPv6 9462076      0t0  TCP phaedrus:46126->
...                                                     ord37s07-in-x0a.1e100.net:https (
ssh      30563 kauffman    3u  IPv4 9420568      0t0  TCP phaedrus:44354->
                                                        csel-apollo.cselabs.umn.edu:ssh (
simple_s 43254 kauffman    3u  IPv6 793630       0t0  TCP *:12344 (LISTEN)
...
```

Exercise: Pause Server

- Server listens for 4 client connections
- Does not respond to any client until 4 have connected
- When 4 connected, issues `Server shutting down` message to all
- Closes connections and shuts down

Frame the server code for this using the system calls

```
| getaddrinfo() | look up address          |
| socket()      | create a socket          |
| bind()        | bind socket to address   |
| listen()      | listen for connections   |
| accept()      | accept connections       |
```

Include control and data structures required

Answers: Pause Server

See `pause_server.c`

```
getaddrinfo(NULL, PORT, &hints, &serv_addr);
int listen_fd = socket(serv_addr->ai_family, serv_addr->ai_socktype,
                       serv_addr->ai_protocol);

bind(listen_fd, serv_addr->ai_addr, serv_addr->ai_addrlen);

listen(listen_fd, BACKLOG);

#define MAX_CLIENTS 4
int client_fds[MAX_CLIENTS];

for(int i=0; i<MAX_CLIENTS; i++){
  client_fds[i]= accept(listen_fd, client_addr, &client_addr_size);
}

for(int i=0; i<MAX_CLIENTS; i++){
  int client_fd = client_fds[i];
  char *msg = "Server shutting down.";
  write(client_fd, msg, strlen(msg));
  close(client_fd);
}
close(listen_fd);
```

Service vs Port

- Recall that port is part of a client/server setup

```
#define PORT "80"
getaddrinfo(hostname, PORT, NULL, &serv_addr);
```

- Not a string by accident: may substitute a **service**

```
#define SERVICE "http"
getaddrinfo(hostname, SERVICE, NULL, &serv_addr);
```

- Known Service/Port association is stored in `/etc/services/`

```
> cat /etc/services
...                             # Transport Protocol (low level)
ftp                 21/tcp
ssh                 22/tcp      # Transfer Control Protocol
ssh                 22/udp      # User Datagram Protocol
ssh                 22/sctp     # Stream Control Transmission Protocol
telnet              23/tcp
...
http                80/tcp
http                80/udp
www                 80/tcp
www                 80/udp
...
doom               666/tcp
...
git               9418/tcp
```

Unix Domain Sockets

> *Remember FIFOs? Remember how they can only send data in one direction, just like a Pipes? Wouldn't it be grand if you could send data in both directions like you can with a socket?*
>
> - *Beej, from Beej's Guide to Unix IPC*

- Can create a socket which is local to a Unix host
- Like FIFO, has a location on the file system such as `/tmp/blather/serv1.sock`
- Server establishes socket location, clients must know about it
- Allows `listen()` / `accept()` to spin up new sockets per client
- **Is bi-directional / full duplex** : a single socket is good for two-way communication (FIFOs are one-directional)

Unix Domain Sockets Demo

- Same call sequence for client/server except no `getaddrinfo()`
- Instead use same local file name to find the local Unix socket

unix_client.c

```
 1  int client_sockfd =
 2    socket(AF_UNIX, SOCK_STREAM, 0);
 3
 4  char *sockfile = "the.sock";
 5  struct sockaddr_un addr = {
 6    .sun_family = AF_UNIX,
 7    .sun_path   = "",
 8  };
 9  strcpy(addr.sun_path, sockfile);
10
11  // local, no getaddrinfo() req'd
12
13  connect(client_sockfd,
14          (struct sockaddr*)&addr,
15          sizeof(addr));
```

unix_server_single.c

```
 1  int connect_sockfd =
 2    socket(AF_UNIX, SOCK_STREAM, 0);
 3
 4  char *sockfile = "the.sock";
 5  struct sockaddr_un addr = {
 6    .sun_family = AF_UNIX,
 7    .sun_path   = "",
 8  };
 9  strcpy(addr.sun_path, sockfile);
10
11  // local, no getaddrinfo() req'd
12
13  bind(connect_sockfd,
14       (struct sockaddr*)&addr,
15       sizeof(addr));
16
17  listen(connect_sockfd, BACKLOG);
18
19  accept(connect_sockfd, NULL, NULL);
```