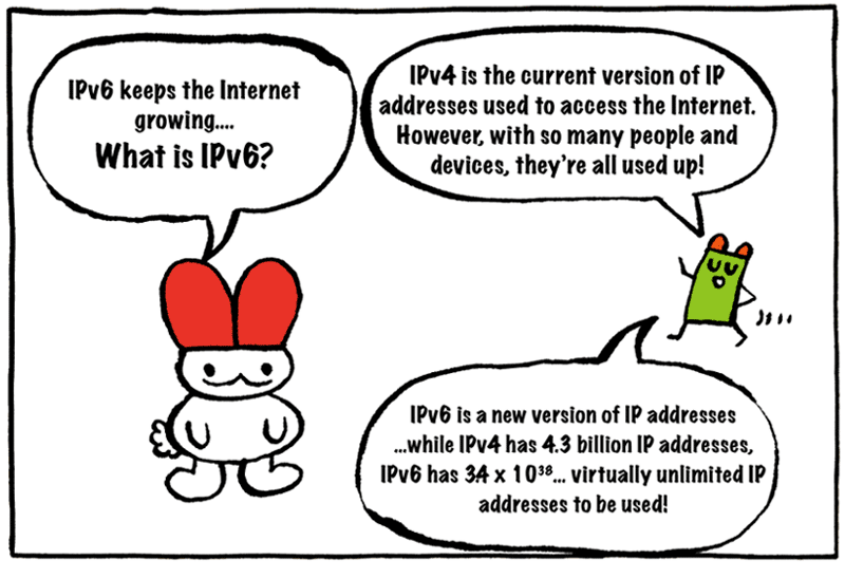
CSCI 4061: Sockets and Network Programming

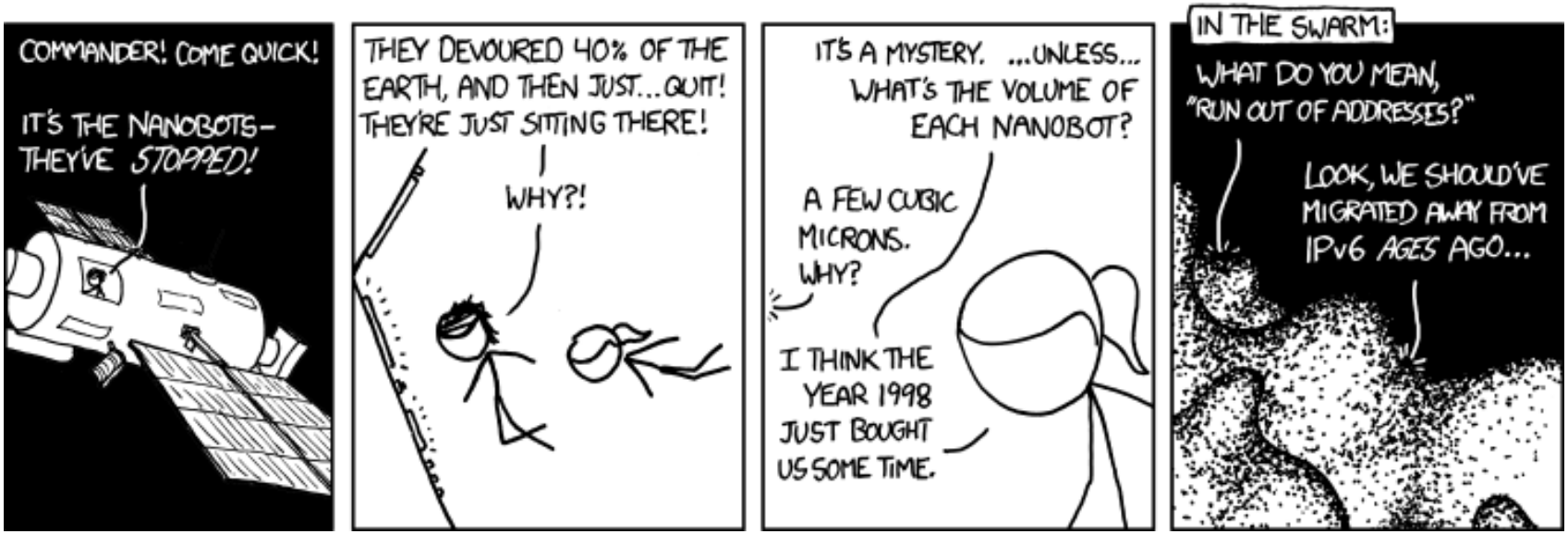
Chris Kauffman

Last Updated:
Tue Dec 5 13:30:56 CST 2017

Networks are Aging



Source: www.ipv6now.hk



Source: XKCD #865

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

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)

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

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 until data is received over network from other party

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 so that
- Server on other side must be listening

If all goes well...

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

Wait, it's just `read()`?
Alternatively:

```
int numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0);
```

allows additional 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()` /`poll()` work for them
- Alternative 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.`
- Alternative 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);
```

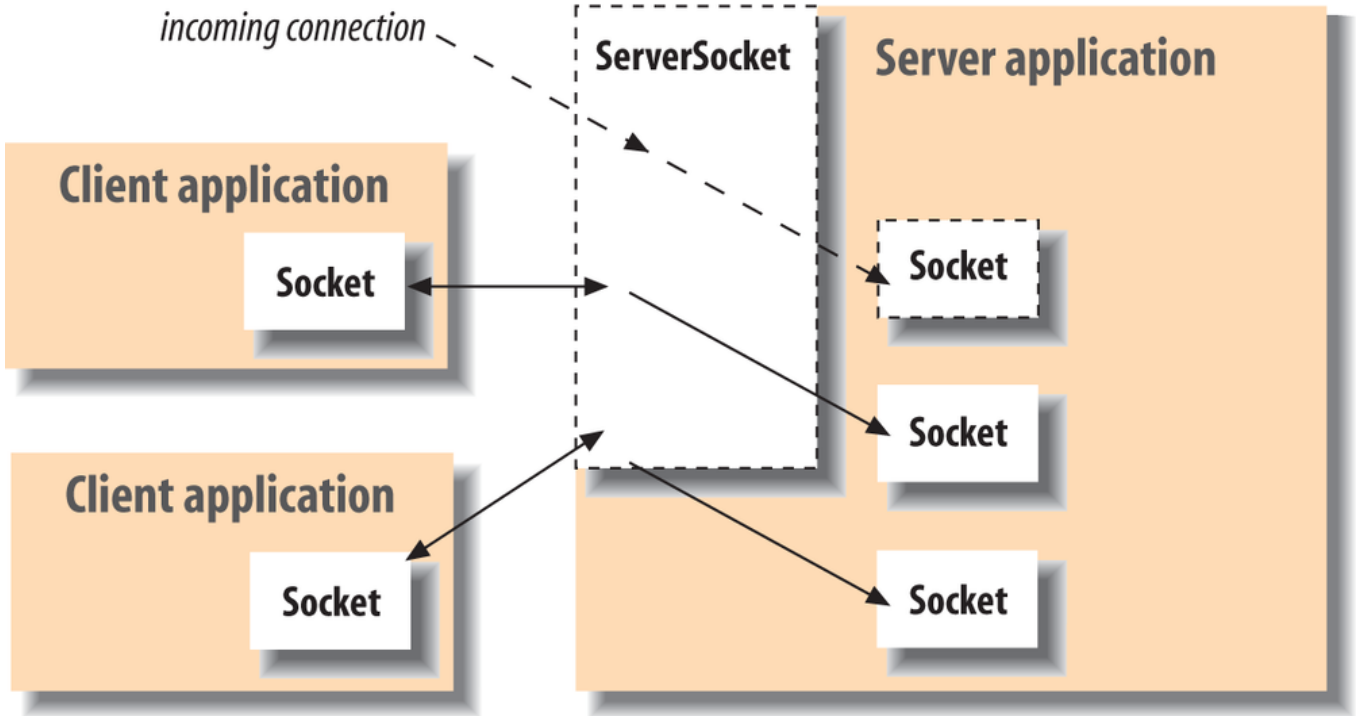
Server Main Loop

```
// 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:

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

- Server at 192.168.1.1 Port 80
- Client 1 10.0.0.1
- Client 2 10.0.0.2

Client 1 at 10.0.0.1 opens a connection on local port 1234 and connects to the server. Now the server has one socket identified as follows:

```
| Peer (Client)    | Local (Server)    |
|------------------+-------------------|
| 10.0.0.1 : 1234 | 192.168.1.1 : 80 |
```

Now Client 2 at 10.0.0.2 opens a connection on local port 5678 and connects to the server. Now the server has two sockets identified as follows :

```
| Peer (Client)    | Local (Server)    |
|------------------+-------------------|
| 10.0.0.1 : 1234 | 192.168.1.1 : 80 |
| 10.0.0.2 : 5678 | 192.168.1.1 : 80 |
```

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

Answer: 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);

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);
```

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 like `/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 so only one socket is needed
- A good summary: `https://troydhanson.github.io/network/Unix_domain_sockets.html`