CS110 Lecture 16: Network System Calls

Principles of Computer Systems

Winter 2020

Stanford University

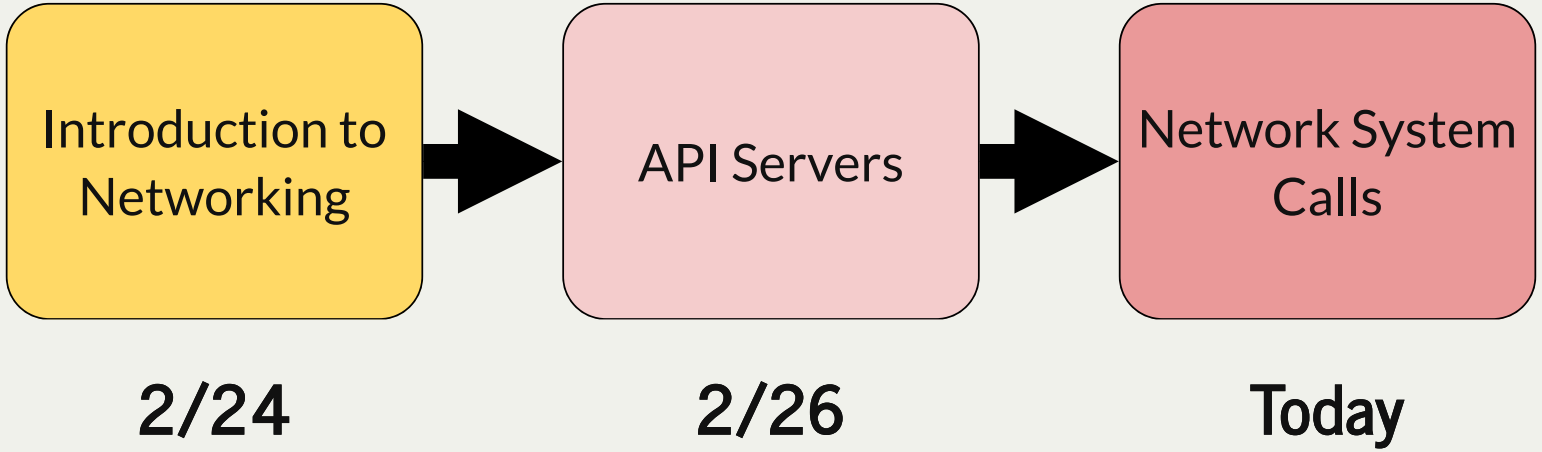
Computer Science Department

Instructors: Chris Gregg and Nick Troccoli

PDF of this presentation

CS110 Topic 4: How can we write programs that communicate over a network with other programs?

Learning About Networking



Today's Learning Goals

- Learn how we can create a client socket descriptor to connect to server
- Learn how we can create a server socket descriptor to listen for client connections

Plan For Today

- **Recap:** Networking So Far
- createClientSocket
- **Break:** Announcements
- createServerSocket
- assign7 Overview

Plan For Today

- **Recap:** Networking So Far
- createClientSocket
- **Break:** Announcements
- createServerSocket
- assign7 Overview

Networking So Far

- Networking allows a program to communicate with a program on another machine.
- Networking code relies on the same *descriptor* framework we used for files
  - Create a descriptor (a number) that represents that resource
  - Read or write with that descriptor
  - Close the descriptor when you're done

Networking So Far

- An ***IP Address*** identifies a machine on a network
- A ***port number*** identifies a specific networked program running on a machine
  - This allows multiple networked programs to run on the same machine
  - Analogy: San Fran = IP address, waterfront piers = ports
- A **socket** is the endpoint of a single connection over a port. It is represented as a descriptor we can read from/write to.
- *"Port" is to "socket descriptor" as "filename" is to "file descriptor"*

Networking So Far

- We can use socket descriptors the same as file descriptors (read, write)
- However, **iosockstream** lets us wrap a socket descriptor in a *stream* (so that we can read/write like we do with **cout** instead of via **read/write**.)

```
static void writeToSocket(int socketDescriptor) {
  sockbuf sb(socketDescriptor);
  iosockstream ss(&sb);
  ss << [CONTENT HERE] << endl;
} // sockbuf destructor closes client
```

Plan For Today

- **Recap:** Networking So Far
- **createClientSocket**
- **Break:** Announcements
- createServerSocket
- assign7 Overview

Clients And Servers

"client" and "server" are the two main program "roles" in networking.

- A **server** continually listens for incoming requests and sends responses back
- A **client** sends a request to a server and does something with the response
- Examples:
  - YouTube app (client) sends requests to the YouTube servers for what content (e.g. videos) to display
  - Web browser (client) sends requests to the server at the URL you enter for what content (e.g. webpage, images) to display

Clients

```
int main(int argc, char *argv[]) {
    int clientSocket = createClientSocket("myserver.com", 12345);
    sockbuf sb(clientSocket);
    iosockstream ss(&sb);
    string responseData;
    getline(ss, responseData);
    cout << responseData << endl;
    return 0;
}
```

1. We create a client socket to connect to a given server
2. This returns a descriptor we can use to read/write
3. We use an iosockstream to simplify using this file descriptor
4. Finally, we print out the line of data sent from the server

But what is **createClientSocket** really doing?

createClientSocket

```
int createClientSocket(const string& host, unsigned short port);
```

1. Check that the specified server and port are valid
2. Create a new socket descriptor
3. Associate this socket descriptor with a connection to that server
4. Return the socket descriptor

createClientSocket

```
int createClientSocket(const string& host, unsigned short port);
```

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**
3. Associate this socket descriptor with a connection to that server - **connect()**
4. Return the socket descriptor

createClientSocket

```
int createClientSocket(const string& host, unsigned short port);
```

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**
3. Associate this socket descriptor with a connection to that server - **connect()**
4. Return the socket descriptor

createClientSocket

- We check the validity of the host by attempting to look up their IP address
- **gethostbyname()** gets host info for the given name (e.g. "www.facebook.com")
- **gethostbyaddr()** gets host info for the given IPv4 address (e.g. "31.13.75.17")
  - First argument is the base address of a character array with ASCII values of 171, 64, 64, and 137 in **network byte order.** For IPv4, the second argument is usually **`sizeof(struct in_addr)`** and the third the **`AF_INET`** constant.

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

- Both are technically deprecated in favor of **getAddrInfo**, but still prevalent and good to know
- Returns a statically allocated **`struct hostent`** with host's info (or NULL if error)

Idea: let's use **gethostbyname()** to look up this host and see if it's valid (non-NULL).

gethostbyname()

```
// represents an IP Address
struct in_addr {
    unsigned int s_addr // stored in network byte order (big endian)
};

// represents a host's info
struct hostent {
    // official name of host
    char *h_name;

    // NULL-terminated list of aliases
    char **h_aliases;

    // host address type (typically AF_INET for IPv4)
    int h_addrtype;

    // address length (typically 4, or sizeof(struct in_addr) for IPv4)
    int h_length;

    // NULL-terminated list of IP addresses
    // This is really a struct in_addr ** when hostent contains IPv4 addresses
    char **h_addr_list;
};
```

Note: `h_addr_list` is typed to be a `char *` array, but for IPv4 records it's really `struct in_addr **`, so we cast it to that in our code.

Why the confusion?

- **h_addr_list** needs to represent an array of pointers to IP addresses.
- **struct hostent** must be generic and work with e.g. both IPv4 and IPv6 hosts.
- Thus, **h_addr_list** could be an array of **in_addr *s** (IPv4) or an array of **in6_addr *s** (IPv6).
- No void * back then, so char ** it is.

createClientSocket

1. Check that the specified server and port are valid - **gethostbyname()**

```
int createClientSocket(const string& host, unsigned short port) {
    struct hostent *he = gethostbyname(host.c_str());
    if (he == NULL) return -1;
    ...
```

createClientSocket

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**

```
int socket(int domain, int type, int protocol);

int createClientSocket(const string& host, unsigned short port) {
    ...
    int s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) return -1;
    ...
```

createClientSocket

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**
3. Associate this socket descriptor with a connection to that server - **connect()**

```
int connect(int clientfd, const struct sockaddr *addr, socklen_t addrlen);
```

Lecture 15: Network System Calls, Library Functions

- The three data structures presented below are in place to model the IP address/port pairs:

```
struct sockaddr { // generic socket
    unsigned short sa_family; // protocol family for socket
    char sa_data[14];
    // address data (and defines full size to be 16 bytes)
};
```

```
struct sockaddr_in { // IPv4 socket address record
    unsigned short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

```
struct sockaddr_in6 { // IPv6 socket address record
    unsigned short sin6_family;
    unsigned short sin6_port;
    unsigned int sin6_flowinfo;;
    struct in6_addr sin6_addr;
    unsigned int sin6_scope_id;
};
```

The **`sockaddr_in`** is used to model IPv4 address/port pairs.

- The **`sin_family`** field should always be initialized to be **`AF_INET`**, which is a constant used to be clear that IPv4 addresses are being used. If it feels redundant that a record dedicated to IPv4 needs to store a constant saying everything is IPv4, then stay tuned.
- The **`sin_port`** field stores a port number in network byte (i.e. big endian) order.
- The **`sin_addr`** field stores an IPv4 address as a packed, big endian **`int`**, as you saw with **`gethostbyname`**and the **`struct hostent`**.
- The **`sin_zero`** field is generally ignored (though it's often set to store all zeroes). It exists primarily to pad the record up to 16 bytes.

Lecture 15: Network System Calls, Library Functions

- The three data structures presented below are in place to model the IP address/port pairs:

```
struct sockaddr { // generic socket
    unsigned short sa_family; // protocol family for socket
    char sa_data[14];
    // address data (and defines full size to be 16 bytes)
};
```

```
struct sockaddr_in { // IPv4 socket address record
    unsigned short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

```
struct sockaddr_in6 { // IPv6 socket address record
    unsigned short sin6_family;
    unsigned short sin6_port;
    unsigned int sin6_flowinfo;;
    struct in6_addr sin6_addr;
    unsigned int sin6_scope_id;
};
```

The **`sockaddr_in6`** is used to model IPv6 address/port pairs.

- The **`sin6_family`** field should always be set to **`AF_INET6`**. As with the **`sin_family`** field, **`sin6_family`** field occupies the first two bytes of surrounding record.
- The **`sin6_port`** field holds a two-byte, network-byte-ordered port number, just like sin_port does.
- A **`struct in6_addr`** is also wedged in there to manage a 128-bit IPv6 address.
- **`sin6_flowinfo`** and **`sin6_scope_id`** are beyond the scope of what we need, so we'll ignore them.

Lecture 15: Network System Calls, Library Functions

- The three data structures presented below are in place to model the IP address/port pairs:

```
struct sockaddr { // generic socket
    unsigned short sa_family; // protocol family for socket
    char sa_data[14];
    // address data (and defines full size to be 16 bytes)
};
```

```
struct sockaddr_in { // IPv4 socket address record
    unsigned short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

```
struct sockaddr_in6 { // IPv6 socket address record
    unsigned short sin6_family;
    unsigned short sin6_port;
    unsigned int sin6_flowinfo;;
    struct in6_addr sin6_addr;
    unsigned int sin6_scope_id;
};
```

The **`struct sockaddr`** is the best C can do to emulate an abstract base class.

- You rarely if ever declare variables of type **`struct sockaddr`**, but many system calls will accept parameters of type **`struct sockaddr *`**.
- Rather than define a set of network system calls for IPv4 addresses and a second set of system calls for IPv6 addresses, Linux defines one set for both.
- If a system call accepts a parameter of type struct sockaddr *, it really accepts the address of either a **`struct sockaddr_in`** or a **`struct sockaddr_in6`**. The system call relies on the value within the first two bytes—the **`sa_family`** field—to determine what the true record type is.

createClientSocket

1. Check that the specified server and port are valid - **gethostbyname()**
2. Create a new socket descriptor - **socket()**
3. Associate this socket descriptor with a connection to that server - **connect()**

```
int createClientSocket(const string& host, unsigned short port) {
    ...
    struct sockaddr_in address;
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_port = htons(port);

    // h_addr is #define for h_addr_list[0]
    address.sin_addr = *((struct in_addr *)he->h_addr);
    if (connect(s, (struct sockaddr *) &address, sizeof(address)) == 0) return s;
    ...
```

Lecture 15: Network System Calls, Library Functions

Here is the full implementation of **`createClientSocket`**:

```
int createClientSocket(const string& host, unsigned short port) {
    struct hostent *he = gethostbyname(host.c_str());
    if (he == NULL) return -1;
    int s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) return -1;
    struct sockaddr_in address;
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_port = htons(port);

    // h_addr is #define for h_addr_list[0]
    address.sin_addr = *((struct in_addr *)he->h_addr);
    if (connect(s, (struct sockaddr *) &address, sizeof(address)) == 0) return s;

    close(s);
    return -1;
}
```

Lecture 15: Network System Calls, Library Functions

Here are a few more details:

- **`address`** is declared to be of type **`struct sockaddr_in`**, since that's the data type specifically set up to model IPv4 addresses. Had we been dealing with IPv6 addresses, we'd have declared a **`struct sockaddr_in6`** instead.
  - It's important to embed **`AF_INET`** within the **`sin_family`** field, since those two bytes are examined by system calls to determine the type of socket address structure.
  - The **`sin_port`** field is, not surprisingly, designed to hold the port of interest. **`htons`**—that's an abbreviation for **`h`**ost-**`to`**-**`n`**etwork-**`s`**hort—is there to ensure the port is stored in network byte order (which is big endian order). On big endian machines, **`htons`** is implemented to return the provided short without modification. On little endian machines (like the **`myth`**s), **`htons`** returns a figure constructed by exchanging the two bytes of the incoming **`short`**. In addition to **`htons`**, Linux also provided **`htonl`** for four-byte **`long`**s, and it also provides **`ntohs`** and **`ntohl`** to restore host byte order from network byte ordered figures.
- The call to **`connect`** associates the descriptor **`s`** with the host/IP address pair modeled by the supplied **`struct sockaddr_in *`**. The second argument is downcast to a **`struct sockaddr *`**, since **`connect`** needs accept a pointer to **any** type within the entire **`struct sockaddr`** family, not just **`struct sockaddr_in`**s.

Plan For Today

- **Recap:** Networking So Far
- createClientSocket
- **Break: Announcements**
- createServerSocket
- assign7 Overview

Announcements

- Assignment 7 due this Fri. 11:59PM

Plan For Today

- **Recap:** Networking So Far
- createClientSocket
- **Break:** Announcements
- **createServerSocket**
- assign7 Overview

createServerSocket

```
int createServerSocket(unsigned short port, int backlog = kDefaultBacklog);
```

1. Create a new socket descriptor - **socket()**
2. Bind this socket to a given port and IP address- **bind()**
3. Make the socket descriptor *passive* to listen for incoming requests - **listen()**
4. Return socket descriptor

Lecture 15: Network System Calls, Library Functions

Here is the full implementation of **`createServerSocket`**:

```
int createServerSocket(unsigned short port, int backlog) {
    int s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) return -1;
    struct sockaddr_in address;
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = htonl(INADDR_ANY);
    address.sin_port = htons(port);
    if (bind(s, (struct sockaddr *)&address, sizeof(address)) == 0 &&
            listen(s, backlog) == 0) return s;

    close(s);
    return -1;
}
```

Lecture 15: Network System Calls, Library Functions

Here are a few details about my implementation of **`createServerSocket`** worth calling out:

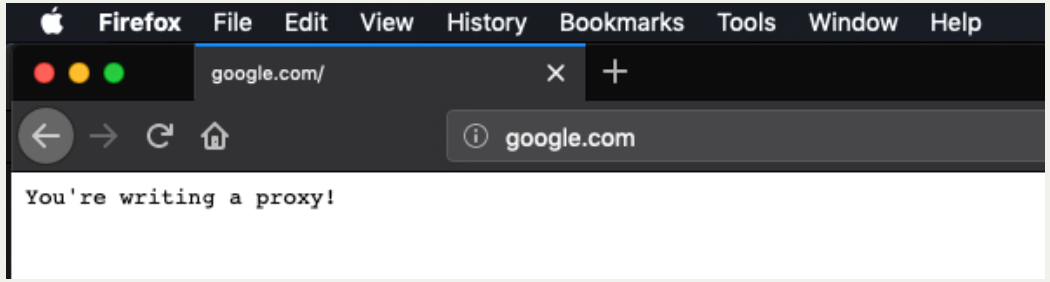
- The call to **`socket`** is precisely the same here as it was in **`createClientSocket`**. It allocates a descriptor and configures it to be a socket descriptor within the **`AF_INET`** namespace.
- The address of type **`struct sockaddr_in`** here is configured in much the same way it was in **`createClientSocket`**, except that the **`sin_addr.s_addr`** field should be set to a local IP address, not a remote one. The constant **`INADDR_ANY`** is used to state that address should represent all local addresses.
- The **`bind`** call simply assigns the set of local IP addresses represented by **`address`** to the provided socket **`s`**. Because we embedded **`INADDR_ANY`** within **`address`**, **`bind`** associates the supplied socket with all local IP addresses. That means once **`createServerSocket`** has done its job, clients can connect to any of the machine's IP addresses via the specified port.
- The **`listen`** call is what converts the socket to be one that's willing to accept connections via **`accept`**. The second argument is a queue size limit, which states how many pending connection requests can accumulate and wait their turn to be **`accept`**ed. If the number of outstanding requests is at the limit, additional requests are simply refused.

Plan For Today

- **Recap:** Networking So Far
- createClientSocket
- **Break:** Announcements
- createServerSocket
- **assign7 Overview**

- A web *proxy server* is a server that acts as a go-between from your browser to sites on the Internet. Proxies can serve many purposes:
  - Block access to certain websites
  - Block access to certain documents (big documents, .zip files, etc.)
  - Act as an anonymizer to strip data from headers about what the real IP address of the client is, or by stripping out cookies or other identifying information. The Tor network, using *onion routing* performs this role (among other roles, such as protecting data with strong encryption)
  - Intercept image requests, serving only upside-down versions of images.
  - Intercept all traffic and redirect to kittenwar.com.
  - Cache requests for static data (e.g., images) so it can later serve local copies rather than re-request from the web.
  - Redirect to a paywall (e.g., what happens at airports)

- Once you have started the proxy (starter code), you should be able to go to any web site, and see just "You're writing a proxy!" in place of the webpage:



- Not much going on!
- After you have set up the proxy, you can leave it (if you browse with another browser), as long as you always ssh into the same myth machine each time you work on the assignment. If you change myth machines, you will need to update the proxy settings (always check this first if you run into issues)
- You should also *frequently* clear the browser's cache, as it can locally cache elements, too, which means that you might load a page without ever going to your proxy.

- Version 1: Sequential Proxy
  - You will eventually add a **`ThreadPool`** to your program, but first, write a sequential version.
  - You will be changing the starter code to be a true proxy, that intercepts the requests and passes them on to the intended server. There are three HTTP methods you need to support:
    - GET: request a web page from the server
    - HEAD: exactly like GET, but only requests the headers
    - POST: send data to the website
  - The request line will look like this:

```
GET http://www.cornell.edu/research/ HTTP/1.1
```

  - For this example, your program forwards the request to www.cornell.edu, with the first line of the request as follows:

```
GET /research/ HTTP/1.1
```

  - You already have a fully implemented **`HTTPRequest`** class, although you will have to update the **`operator<<`** function at a later stage.

Lecture Recap

- **Recap:** Networking So Far
- createClientSocket
- **Break:** Announcements
- createServerSocket
- assign7 Overview

Next Time: Overview of MapReduce