Lecture 15: Networking, Clients

Principles of Computer Systems

Winter 2020

Stanford University

Computer Science Department

Lecturers: Chris Gregg and

Nick Troccoli

PDF of this presentation

Lecture 15: API Servers, Threads, Processes

- I want to implement an API server that's architecturally in line with the way Google, Twitter, Facebook, and LinkedIn architect their own API servers.
- This example is inspired by a website called Lexical Word Finder.
    - Our implementation assumes we have a standard Unix executable called **`scrabbleword-finder`**. The source code for this executable—completely unaware it'll be used in a larger networked application—can be found right here.
    - **`scrabble-word-finder`** is implemented using only CS106B techniques—standard file I/O and procedural recursion with simple pruning.
    - Here are two abbreviated sample runs:

```
cgregg@myth61:$ ./scrabble-word-finder lexical
ace
// many lines omitted for brevity
lei
lex
lexica
lexical
li
lice
lie
lilac
xi
cgregg@myth61:$
```

```
cgregg@myth61:$ ./scrabble-word-finder network
en
// many lines omitted for brevity
wonk
wont
wore
work
worn
wort
wot
wren
wrote
cgregg@myth61:$
```

Lecture 15: API Servers, Threads, Processes

- I want to implement an API service using HTTP to replicate what **`scrabble-wordfinder`** is capable of.
  - We'll expect the API call to come in the form of a URL, and we'll expect that URL to include the rack of letters.
  - Assuming our API server is running on **`myth54:13133`**, we expect **`http://myth54:13133/lexical`** and **`http://myth54:13133/network`** to generate the following JSON payloads:


```
{
  "time":0.041775,
  "cached": false,
  "possibilities": [
    'ace',
    // several words omitted
    'lei',
    'lex',
    'lexica',
    'lexical',
    'li',
    'lice',
    'lie',
    'lilac',
    'xi'
  ]
}
```

```
{
  "time": 0.223399,
  "cached": false,
  "possibilities": [
    'en',
    // several words omitted
    'wonk',
    'wont',
    'wore',
    'work',
    'worn',
    'wort',
    'wot',
    'wren',
    'wrote'
  ]
}
```

Lecture 15: API Servers, Threads, Processes

- One might think to cannibalize the code within **`scrabble-word-finder.cc`** to build the core of [**`scrabble-word-finder-server.cc`**].
- Reimplementing from scratch is wasteful, time-consuming, and unnecessary. **`scrabble-word-finder`** already outputs the primary content we need for our payload. We're packaging the payload as JSON instead of plain text, but we can still tap **`scrabble-word-finder`** to generate the collection of formable words.
- Can we implement a server that leverages existing functionality? Of course we can!
- We can just leverage our **`subprocess_t`** type and **`subprocess`** function from Assignment 3.

```
struct subprocess_t {
    pid_t pid;
    int supplyfd;
    int ingestfd;
};

subprocess_t subprocess(char *argv[],
        bool supplyChildInput, bool ingestChildOutput) throw (SubprocessException);
```

Lecture 15: API Servers, Threads, Processes

- Here is the core of the **`main`** function implementing our server:

```
int main(int argc, char *argv[]) {
    unsigned short port = extractPort(argv[1]);
    int server = createServerSocket(port);
    cout << "Server listening on port " << port << "." << endl;
    ThreadPool pool(16);
    map<string, vector<string>> cache;
    mutex cacheLock;
    while (true) {
        struct sockaddr_in address;
        // used to surface IP address of client
        socklen_t size = sizeof(address); // also used to surface client IP address
        bzero(&address, size);
        int client = accept(server, (struct sockaddr *) &address, &size);
        char str[INET_ADDRSTRLEN];
        cout << "Received a connection request from "
            << inet_ntop(AF_INET, &address.sin_addr, str, INET_ADDRSTRLEN) << "." << endl;
        pool.schedule([client, &cache, &cacheLock] {
                publishScrabbleWords(client, cache, cacheLock);
                });
    }
    return 0;
}
```

Lecture 15: API Servers, Threads, Processes

- The second and third arguments to **`accept`** are used to surface the IP address of the client.
- Ignore the details around how I use **`address`**, **`size`**, and the **`inet_ntop`** function until next week, when we'll talk more about them. Right now, it's a neat-to-see!
- Each request is handled by a dedicated worker thread within a **`ThreadPool`** of size 16.
- The thread routine called **`publishScrabbleWords`** will rely on our **`subprocess`** function to marshal plain text output of scrabble-word-finder into JSON and publish that JSON as the payload of the HTTP response.
- The next slide includes the full implementation of **`publishScrabbleWords`** and some of its helper functions.
- Most of the complexity comes around the fact that I've *elected* to maintain a cache of previously processed letter racks.

Lecture 15: API Servers, Threads, Processes

- Here is **`publishScrabbleWords`**:

```
static void publishScrabbleWords(int client, map<string, vector<string>>& cache,
                                 mutex& cacheLock) {
    sockbuf sb(client);
    iosockstream ss(&sb);
    string letters = getLetters(ss);
    sort(letters.begin(), letters.end());
    skipHeaders(ss);
    struct timeval start;
    gettimeofday(&start, NULL); // start the clock
    cacheLock.lock();
    auto found = cache.find(letters);
    cacheLock.unlock(); // release lock immediately, iterator won't be invalidated by competing find calls
    bool cached = found != cache.end();
    vector<string> formableWords;
    if (cached) {
        formableWords = found->second;
    } else {
        const char *command[] = {"./scrabble-word-finder", letters.c_str(), NULL};
        subprocess_t sp = subprocess(const_cast<char **>(command), false, true);
        pullFormableWords(formableWords, sp.ingestfd);
        waitpid(sp.pid, NULL, 0);
        lock_guard<mutex> lg(cacheLock);
        cache[letters] = formableWords;
    }
    struct timeval end, duration;
    gettimeofday(&end, NULL); // stop the clock, server-computation of formableWords is complete
    timersub(&end, &start, &duration);
    double time = duration.tv_sec + duration.tv_usec/1000000.0;
    ostringstream payload;
    constructPayload(formableWords, cached, time, payload);
    sendResponse(ss, payload.str());
}
```

Lecture 15: API Servers, Threads, Processes

- Here's the **`pullFormableWords`** and **`sendResponse`** helper functions.

```
static void pullFormableWords(vector<string>& formableWords, int ingestfd) {
    stdio_filebuf<char> inbuf(ingestfd, ios::in);
    istream is(&inbuf);
    while (true) {
        string word;
        getline(is, word);
        if (is.fail()) break;
        formableWords.push_back(word);
    }
}

static void sendResponse(iosockstream& ss, const string& payload) {
    ss << "HTTP/1.1 200 OK\r\n";
    ss << "Content-Type: text/javascript; charset=UTF-8\r\n";
    ss << "Content-Length: " << payload.size() << "\r\n";
    ss << "\r\n";
    ss << payload << flush;
}
```

Lecture 15: API Servers, Threads, Processes

- Finally, here are the **`getLetters`** and the **`constructPayload`** helper functions. I omit the implementation of **`skipHeaders`**—you saw it with **`web-get`**—and **`constructJSONArray`**, which you're welcome to view right here.

```
static string getLetters(iosockstream& ss) {
    string method, path, protocol;
    ss >> method >> path >> protocol;
    string rest;
    getline(ss, rest);
    size_t pos = path.rfind("/");
    return pos == string::npos ? path : path.substr(pos + 1);
}
static void constructPayload(const vector<string>& formableWords, bool cached,
                             double time, ostringstream& payload) {
    payload << "{" << endl;
    payload << "  \"time\":"  << time << "," << endl;
    payload << "  \"cached\": " << boolalpha << cached << "," << endl;
    payload << "  \"possibilities\": " << constructJSONArray(formableWords, 2) << endl;
    payload << "}" << endl;
}
```

- Our **`scrabble-word-finder-server`** provided a single API call that resembles the types of API calls afforded by Google, Twitter, or Facebook to access search, tweet, or friend-graph data.

Lecture 15: Network System Calls, Library Functions

- Hostname Resolution: IPv4
- Linux C includes directives to convert host names (e.g. "**`www.facebook.com`**") to IPv4 address (e.g. "**`31.13.75.17`**") and vice versa. Functions called **`gethostbyname`** and **`gethostbyaddr`**, while technically deprecated, are still so prevalent that you should know how to use them.
- In fact, your B&O textbook only mentions these deprecated functions:

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

- Each function populates a statically allocated **`struct hostent`** describing some host machine on the Internet.
  - **`gethostbyname`** assumes its argument is a host name (e.g. "**`www.google.com`**").
  - **`gethostbyaddr`** assumes the first argument is a binary representation of an IP address (e.g. not the string "**`171.64.64.137`**", but the base address of a character array with ASCII values of 171, 64, 64, and 137 laid down side by side in **`network byte order`**. For IPv4, the second argument is usually 4 (or rather, **`sizeof(struct in_addr)`**) and the third is typically the **`AF_INET`** constant.

Lecture 15: Network System Calls, Library Functions

- Hostname Resolution: IPv4
  - The **`struct hostent`** record packages all of the information about a particular host:

```
struct in_addr {
    unsigned int s_addr // four bytes, stored in network byte order (big endian)
};
struct hostent {
    char *h_name;
    // official name of host
    char **h_aliases;
    // NULL-terminated list of aliases
    int h_addrtype;
    // host address type (typically AF_INET for IPv4)
    int h_length;
    // address length (typically 4, or sizeof(struct in_addr) for IPv4)
    char **h_addr_list; // NULL-terminated list of IP addresses
}; // h_addr_list is really a struct in_addr ** when hostent contains IPv4 addresses
```

- The **`struct in_addr`** is a one-field record modeling an IPv4 address.
  - The **`s_addr`** field packs each figure of a dotted quad (e.g. 171.64.64.136) into one of its four bytes. Each of these four numbers numbers can range from 0 up through 255.
- The **`struct hostent`** is used for all IP addresses, not just IPv4 addresses. For non-IPv4 addresses, **`h_addrtype`**, **`h_length`**, and **`h_addr_list`** carry different types of data than they do for IPv4

Lecture 15: Network System Calls, Library Functions

Users prefer the host naming scheme behind "**`www.facebook.com`**", but network communication ultimately works with IP addresses like "31.13.75.17".

- Not surprisingly, **`gethostbyname`** and **`gethostbyaddr`** are used to manage translations between the two.
- Here's the core of larger program (full program here) that continuously polls the users for hostnames and responds by publishing the set of one or more IP addresses each hostname is bound to:

```
static void publishIPAddressInfo(const string& host) {
    struct hostent *he = gethostbyname(host.c_str());
    if (he == NULL) { // NULL return value means resolution attempt failed
        cout << host << " could not be resolved to an address. Did you mistype it?" << endl;
        return;
    }

    cout << "Official name is \"" << he->h_name << "\"" << endl;
    cout << "IP Addresses: " << endl;
    struct in_addr **addressList = (struct in_addr **) he->h_addr_list;
    while (*addressList != NULL) {
        char str[INET_ADDRSTRLEN];
        cout << "+ " << inet_ntop(AF_INET, *addressList, str, INET_ADDRSTRLEN) << endl;
        addressList++;
    }
}
```

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv4

- **`h_addr_list`** is typed to be a **`char *`** array, implying it's an array of C strings, perhaps dotted quad IP addresses. However, that's not correct. For IPv4 records, **`h_addr_list`** is an array of **`struct in_addr *s`**.
- The **`inet_ntop`** function places a traditional C string presentation of an IP address into the provided character buffer, and returns the the base address of that buffer.
- The while loop crawls over the **`h_addr_list`** array until it lands on a **`NULL`**.

```
static void publishIPAddressInfo(const string& host) {
    struct hostent *he = gethostbyname(host.c_str());
    if (he == NULL) { // NULL return value means resolution attempt failed
        cout << host << " could not be resolved to an address. Did you mistype it?" << endl;
        return;
    }

    cout << "Official name is \"" << he->h_name << "\"" << endl;
    cout << "IP Addresses: " << endl;
    struct in_addr **addressList = (struct in_addr **) he->h_addr_list;
    while (*addressList != NULL) {
        char str[INET_ADDRSTRLEN];
        cout << "+ " << inet_ntop(AF_INET, *addressList, str, INET_ADDRSTRLEN) << endl;
        addressList++;
    }
}
```

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv4

- A sample run of our hostname resolver is presented on the right.
- In general, you see that most of the hostnames we recognize are in fact the officially recorded hostnames.
- **`www.yale.edu`** is the exception. It looks like Yale relies on a content delivery network called Cloudflare, and **`www.yale.edu`** is catalogued as an alias.
- Google's IP address is different by geographical location, which is why it exposes only one IP address.
- Billions of people use okcupid every second, though, which is why it necessarily expose five.

```
myth61$ ./resolve-hostname
Welcome to the IP address resolver!
Enter a host name: www.google.com
Official name is "www.google.com"
IP Addresses:
+ 216.58.192.4
Enter a host name: www.coinbase.com
Official name is "www.coinbase.com"
IP Addresses:
+ 104.16.9.251
+ 104.16.8.251
Enter a host name: www.yale.edu
Official name is "www.yale.edu.cdn.cloudflare.net"
IP Addresses:
+ 104.16.140.133
+ 104.16.141.133
Enter a host name: www.okcupid.com
Official name is "www.okcupid.com"
IP Addresses:
+ 198.41.209.132
+ 198.41.209.133
+ 198.41.208.132
+ 198.41.209.131
+ 198.41.208.133
Enter a host name: www.wikipedia.org
Official name is "www.wikipedia.org"
IP Addresses:
+ 198.35.26.96
Enter a host name:
All done!
myth61$
```

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv6

- Because IPv4 addresses are 32 bits, there are 2^32, or roughly 4 billion different IP addresses. That may sound like a lot, but it was recognized decades ago that we'd soon run out of IPv4 addresses.
- In contrast, there are 340,282,366,920,938,463,463,374,607,431,768,211,456 IPv6 addresses. That's because IPv6 addresses are 128 bits.
- Here are a few IPv6 addresses:
  - Google's 2607:f8b0:4005:80a::2004
  - MIT's 2600:1406:1a:396::255e and 2600:1406:1a:38d::255e
  - Berkeley's 2600:1f14:436:7801:15f8:d879:9a03:eec0 and 2600:1f14:436:7800:4598:b474:29c4:6bc0
  - The White House's 2600:1406:1a:39e::fc4 and 2600:1406:1a:39b::fc4

A more generic version of **`gethostbyname`**—inventively named **`gethostbyname2`**—can be used to extract IPv6 address information about a hostname.

```
struct hostent *gethostbyname2(const char *name, int af);
```

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv6

- There are only two valid address types that can be passed as the second argument to **`gethostbyname2`**: **`AF_INET`** and **`AF_INET6`**.
  - A call to **`gethostbyname2(host, AF_INET)`** is equivalent to a call to **`gethostbyname(host)`**
  - A call to **`gethostbyname2(host, AF_INET6)`** still returns a **`struct hostent *`**, but the struct hostent is populated with different values and types:
    - the **`h_addrtype`** field is set to **`AF_INET6`**,
    - the **`h_length`** field houses a 16 (or rather, sizeof(struct in6_addr)), and
    - the **`h_addr_list`** field is really an array of **`struct in6_addr`** pointers, where each **`struct in6_addr`** looks like this:

```
struct in6_addr {
    u_int8_t s6_addr[16]; // 16 bytes (128 bits), stored in network byte order
};
```

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv6

- Here is the **`IPv6`** version of the **`publishIPAddressInfo`** we wrote earlier (we call it **`publishIPv6AddressInfo`**).

```
static void publishIPv6AddressInfo(const string& host) {
    struct hostent *he = gethostbyname2(host.c_str(), AF_INET6);
    if (he == NULL) { // NULL return value means resolution attempt failed
        cout << host << " could not be resolved to an address. Did you mistype it?" << endl;
        return;
    }

    cout << "Official name is \"" << he->h_name << "\"" << endl;
    cout << "IPv6 Addresses: " << endl;
    struct in6_addr **addressList = (struct in6_addr **) he->h_addr_list;
    while (*addressList != NULL) {
        char str[INET6_ADDRSTRLEN];
        cout << "+ " << inet_ntop(AF_INET6, *addressList, str, INET6_ADDRSTRLEN) << endl;
        addressList++;
    }
}
```

- Notice the call to **`gethostbyname2`**, and notice the explicit use of **`AF_INET6`**, **`struct in6_addr`**, and **`INET6_ADDRSTRLEN`**.
- Full program is right here.

Lecture 15: Network System Calls, Library Functions

Hostname Resolution: IPv6

- A sample run of our IPv6 hostname resolver is presented below.
  - Note that many hosts aren't IPv6-compliant yet, so they don't admit IPv6 addresses.

```
myth61$ ./resolve-hostname6
Welcome to the IPv6 address resolver!
Enter a host name: www.facebook.com
Official name is "star-mini.c10r.facebook.com"
IPv6 Addresses:
+ 2a03:2880:f131:83:face:b00c:0:25de
Enter a host name: www.microsoft.com
Official name is "e13678.dspb.akamaiedge.net"
IPv6 Addresses:
+ 2600:1406:1a:386::356e
+ 2600:1406:1a:397::356e
Enter a host name: www.google.com
Official name is "www.google.com"
IPv6 Addresses:
+ 2607:f8b0:4005:801::2004
Enter a host name: www.berkeley.edu
Official name is "www-production-1113102805.us-west-2.elb.amazonaws.com"
IPv6 Addresses:
+ 2600:1f14:436:7800:4598:b474:29c4:6bc0
+ 2600:1f14:436:7801:15f8:d879:9a03:eec0
Enter a host name: www.harvard.edu
www.harvard.edu could not be resolved to an address. Did you mistype it?
Enter a host name:
All done!
myth61$
```

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 `sockaddr_in` 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.

Lecture 15: Network System Calls, Library Functions

At this point, we know most of the directives needed to implement and understand how to implement **`createClientSocket`** and **`createServerSocket`**.

- **`createClientSocket`** is the easier of the two, so we'll implement that one first. (For simplicity, we'll confine ourselves to an IPv4 world.)
- Fundamentally, createClientSocket needs to:
-
  - Confirm the host of interest is really on the net by checking to see if it has an IP address. **`gethostbyname`** does this for us.
  - Allocate a new descriptor and configure it to be a socket descriptor. We'll rely on the **`socket`** system call to do this.
  - Construct an instance of a **`struct sockaddr_in`** that packages the host and port number we're interested in connecting to.
  - Associate the freshly allocated socket descriptor with the host/port pair. We'll rely on an aptly named system call called **`connect`** to do this.
  - Return the fully configured client socket.
- The full implementation of createClientSocket is on the next slide.

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 details about my implementation of `createClientSocket` worth calling out:

- We call `gethostbyname` first before we call `socket`, because we want to confirm the host has a registered IP address—which means it's reachable—before we allocate any system resources.
- Recall that `gethostbyname` is intrinsically IPv4. If we wanted to involve IPv6 addresses instead, we would need to use `gethostbyname2`.
- The call to `socket` finds, claims, and returns an unused descriptor. `AF_INET` configures it to be compatible with an IPv4 address, and `SOCK_STREAM` configures it to provide reliable data transport, which basically means the socket will reorder data packets and requests missing or garbled data packets to be resent so as to give the impression that data that is received in the order it's sent.
  - The first argument could have been `AF_INET6` had we decided to use IPv6 addresses instead. (Other arguments are possible, but they're less common.)
  - The second argument could have been `SOCK_DGRAM` had we preferred to collect data packets in the order they just happen to arrive and manage missing and garbled data packets ourselves. (Other arguments are possible, though they're less common.)

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.

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.