Network Programming: Part 2

CS351: Systems Programming
Day 24: Nov. 15, 2022

Instructor:
Nik Sultana

Slides adapted from Bryant and O’Hallaron
Next time: back to **in-person in SB104**

<table>
<thead>
<tr>
<th>Nov 14</th>
<th>Nov 15</th>
<th>Nov 16</th>
<th>Nov 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAB</td>
<td>LEC 24: Network Programming: Part 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Preparation: Read CS.APP 11.5-11.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LEC 25: Concurrent Programming</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Preparation: Read CS.APP 12.1-12.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Third lab assignment

- **Good overall!**
- **Zero grades:** ensure timely completion of lab.
- **Low grades:** work with TA to get feedback.
State of the art: SDN

- “Software-Defined Networking”

- "Production Experience with SDN Systems”
  Dr Richard Alimi (Principal Engineer at Google)
  Thursday 1st December 2022 at 1pm-2pm
  Sign up: https://forms.gle/3By54f6MV1iamoiB7
Host and Service Conversion: getaddrinfo

- getaddrinfo is the modern way to convert string representations of hostnames, host addresses, ports, and service names to socket address structures.
  - Replaces obsolete gethostbyname and getservbyname funcs.

- Advantages:
  - Reentrant (can be safely used by threaded programs).
  - Allows us to write portable protocol-independent code
    - Works with both IPv4 and IPv6

- Disadvantages
  - Somewhat complex
  - Fortunately, a small number of usage patterns suffice in most cases.
Host and Service Conversion: getaddrinfo

Given host and service, getaddrinfo returns result that points to a linked list of addrinfo structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.

**Helper functions:**
- freeaddrinfo frees the entire linked list.
- gai_strerror converts error code to an error message.
Linked List Returned by getaddrinfo

- Clients: walk this list, trying each socket address in turn, until the calls to `socket` and `connect` succeed.
- Servers: walk the list until calls to `socket` and `bind` succeed.
Each addrinfo struct returned by getaddrinfo contains arguments that can be passed directly to socket function.

Also points to a socket address struct that can be passed directly to connect and bind functions.
Host and Service Conversion: `getnameinfo`

- `getnameinfo` is the inverse of `getaddrinfo`, converting a socket address to the corresponding host and service.
  - Replaces obsolete `gethostbyaddr` and `getservbyport` funcs.
  - Reentrant and protocol independent.

```c
int getnameinfo(const SA *sa, socklen_t salen, /* In: socket addr */
    char *host, size_t hostlen, /* Out: host */
    char *serv, size_t servlen, /* Out: service */
    int flags); /* optional flags */
```
#include "csapp.h"

int main(int argc, char **argv)
{
    struct addrinfo *p, *listp, hints;
    char buf[MAXLINE];
    int rc, flags;

    /* Get a list of addrinfo records */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET; /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* Connections only */
    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
        exit(1);
    }
}
Conversion Example (cont)

/* Walk the list and display each IP address */
flags = NI_NUMERICHOST; /* Display address instead of name */
for (p = listp; p; p = p->ai_next) {
    Getnameinfo(p->ai_addr, p->ai_addrlen,  
                buf, MAXLINE, NULL, 0, flags);
    printf("%s\n", buf);
}

/* Clean up */
Freeaddrinfo(listp);
exit(0);
Running hostinfo

fourier> ./hostinfo localhost
127.0.0.1

fourier> ./hostinfo www.cs.iit.edu
216.47.157.249

fourier> ./hostinfo twitter.com
104.244.42.129
104.244.42.1
1. Start server
   - `Server`
   - `getaddrinfo`
   - `socket`
   - `bind`
   - `listen`
   - `accept`
   - `connect`

2. Start client
   - `Client`
   - `getaddrinfo`
   - `socket`
   - `bind`
   - `listen`
   - `accept`
   - `connect`

3. Exchange data
   - `Client`
   - `Server`
   - `rio_readlineb`
   - `rio_writen`
   - `rio_readlineb`
   - `rio_writen`

4. Disconnect client
   - `close`

5. Drop client
   - `close`

Client / Server Session

Open clientfd

Await connection request from next client

EOF

Connection request

Open listenfd
Recall: Socket Address Structures

- **Generic socket address:**
  - For address arguments to `connect`, `bind`, and `accept`
  - Necessary only because C did not have generic (`void *`) pointers when the sockets interface was designed
  - For casting convenience, we adopt the Stevens convention:
    ```c
typedef struct sockaddr SA;

struct sockaddr {
    uint16_t sa_family;  /* Protocol family */
    char sa_data[14];    /* Address data. */
};
```

- `sa_family` is specific to the protocol family.
Recall: Socket Address Structures

- Internet-specific socket address:
  - Must cast (struct sockaddr_in *) to (struct sockaddr *) for functions that take socket address arguments.

```c
struct sockaddr_in {
    uint16_t sin_family; /* Protocol family (always AF_INET) */
    uint16_t sin_port; /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```

<table>
<thead>
<tr>
<th>sa_family</th>
<th>sin_port</th>
<th>sin_addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF_INET</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
</tbody>
</table>
Sockets Interface: `socket`

- Clients and servers use the `socket` function to create a socket descriptor:

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

- Example:

  ```c
  int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
  ```

  Indicates that we are using 32-bit IPV4 addresses
  Indicates that the socket will be the end point of a connection

Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.
Sockets Interface: `bind`

- A server uses `bind` to ask the kernel to associate the server’s socket address with a socket descriptor:

  ```c
  int bind(int sockfd, SA *addr, socklen_t addrlen);
  ```

- The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`.

- Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`.

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`. 
Client / Server Session

**Client**

- `getaddrinfo`
- `socket`
- `connect`
- `rio_readlineb`
- `rio_writen`
- `close`

**Server**

- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `rio_readlineb`
- `rio_writen`
- `close`

Connections:
- `open_clientfd`
- `open_listenfd`
- `Connection request` from client to server
- `Await connection request from next client`

Diagram notes:
- `bind` and `listen` operations are part of the server setup.
- `accept` is the server operation to accept incoming connections.
- `connect` is the client operation to connect to the server.
- `rio_readlineb` and `rio_writen` are used for reading and writing data.
Sockets Interface: `listen`

- By default, kernel assumes that descriptor from socket function is an *active socket* that will be on the client end of a connection.
- A server calls the `listen` function to tell the kernel that a descriptor will be used by a server rather than a client:

  ```c
  int listen(int sockfd, int backlog);
  ```

- Converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients.

- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.
Client / Server Session

**Client**
- getaddrinfo
- socket
- connect
- rio_readlineb
- rio_writen
- close
- EOF

**Server**
- getaddrinfo
- socket
- bind
- listen
- accept
- rio_readlineb
- rio_writen
- close
- EOF

Sockets Interface

open_clientfd → Client
- getaddrinfo
- socket
- connect
- rio_readlineb
- rio_writen
- close
- EOF

open_listenfd → Server
- getaddrinfo
- socket
- bind
- listen
- accept
- rio_readlineb
- rio_writen
- close
- EOF

Connection request from next client

Await connection request from next client
Sockets Interface: `accept`

- Servers wait for connection requests from clients by calling `accept`:

  ```c
  int accept(int listenfd, SA *addr, int *addrlen);
  ```

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client’s socket address in `addr` and size of the socket address in `addrlen`.

- Returns a *connected descriptor* that can be used to communicate with the client via Unix I/O routines.
**Client / Server Session**

**Client**
- `getaddrinfo`
- `socket`
- `connect`
- `rio_readlineb`
- `rio_writen`
- `close`

**Server**
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `rio_readlineb`
- `rio_writen`
- `close`

**Sockets Interface**

- `open_clientfd`
- `open_listenfd`

- **Connection request**: `connect` to `accept`

- **Await connection request from next client**: `rio_readlineb` to `rio_writen`
Sockets Interface: connect

- A client establishes a connection with a server by calling connect:

```c
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

- Attempts to establish a connection with server at socket address `addr`
  - If successful, then `clientfd` is now ready for reading and writing.
  - Resulting connection is characterized by socket pair
    
    
    \[(x:y, addr.sin_addr:addr.sin_port)\]
    
    - `x` is client address
    - `y` is ephemeral port that uniquely identifies client process on client host

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`. 
1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`.

2. Client makes connection request by calling and blocking in `connect`.

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`.
Connected vs. Listening Descriptors

- **Listening descriptor**
  - End point for client connection requests
  - Created once and exists for lifetime of the server

- **Connected descriptor**
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

- **Why the distinction?**
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request
Sockets Interface

**Client**

1. `getaddrinfo`
2. `socket`
3. `connect` (Connection request)
4. `rio_readlineb`
5. `rio_writen`
6. `close`

**Server**

1. `getaddrinfo`
2. `socket`
3. `bind`
4. `listen`
5. `accept`
6. `rio_readlineb`
7. `rio_writen`
8. `close` (EOF)

- `open_clientfd`
- `open_listenfd` (Await connection request from next client)

---

**Client / Server Session**
Sockets Interface

Client

- `getaddrinfo`
- `socket`
- `connect`
- `rio_readlineb`
- `rio_writen`
- `close`

Server

- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `rio_readlineb`
- `rio_writen`
- `close`

open_clientfd

Await connection request from next client
Establish a connection with a server

```c
int open_clientfd(char *hostname, char *port) {
    int clientfd;
    struct addrinfo hints, *listp, *p;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;  /* Open a connection */
    hints.ai_flags = AI_NUMERICSERV;  /* ...using numeric port arg. */
    hints.ai_flags |= AI_ADDRCONFIG;  /* Recommended for connections */
    Getaddrinfo(hostname, port, &hints, &listp);
}
```
/ * Walk the list for one that we can successfully connect to */
for (p = listp; p; p = p->ai_next) {
  /* Create a socket descriptor */
  if ((clientfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
    continue; /* Socket failed, try the next */

  /* Connect to the server */
  if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
    break; /* Success */
  Close(clientfd); /* Connect failed, try another */
}

/* Clean up */
Freeaddrinfo(listp);
if (!p) /* All connects failed */
  return -1;
else /* The last connect succeeded */
  return clientfd;
}
Sockets Helper: open_listenfd

- Create a listening descriptor that can be used to accept connection requests from clients.

```c
int open_listenfd(char *port)
{
    struct addrinfo hints, *listp, *p;
    int listenfd, optval=1;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;               /* Accept connect. */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG;    /* ...on any IP addr */
    hints.ai_flags |= AI_NUMERICSERV;              /* ...using port no. */
    Getaddrinfo(NULL, port, &hints, &listp);
}
```
/* Walk the list for one that we can bind to */
for (p = listp; p; p = p->ai_next) {
    /* Create a socket descriptor */
    if ((listenfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
        continue; /* Socket failed, try the next */

    /* Eliminates "Address already in use" error from bind */
    Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval, sizeof(int));

    /* Bind the descriptor to the address */
    if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
        break; /* Success */
    Close(listenfd); /* Bind failed, try the next */
}
Sockets Helper: open_listenfd (cont)

```c
/* Clean up */
Freeaddrinfo(listp);
if (!p) /* No address worked */
    return -1;

/* Make it a listening socket ready to accept conn. requests */
if (listen(listenfd, LISTENQ) < 0) {
    Close(listenfd);
    return -1;
}
return listenfd;
```

Key point: open_clientfd and open_listenfd are both independent of any particular version of IP.
#include "csapp.h"

int main(int argc, char **argv)
{
    int clientfd;
    char *host, *port, buf[MAXLINE];
    rio_t rio;

    host = argv[1];
    port = argv[2];

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
Iterative Echo Server: Main Routine

```c
#include "csapp.h"

void echo(int connfd);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; /* Enough room for any addr */
    char client_hostname[MAXLINE], client_port[MAXLINE];

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage); /* Important! */
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *) &clientaddr, clientlen,
                    client_hostname, MAXLINE, client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

echoserveri.c
Echo Server: echo function

- The server uses RIO to read and echo text lines until EOF (end-of-file) condition is encountered.
  - EOF condition caused by client calling \texttt{close(clientfd)}

```c
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
        Rio_writen(connfd, buf, n);
    }
}
```

The server uses RIO to read and echo text lines until EOF (end-of-file) condition is encountered. EOF condition caused by client calling \texttt{close(clientfd)}.
Testing Servers Using `telnet`

- The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections
  - Our simple echo server
  - Web servers
  - Mail servers

**Usage:**

- `linux> telnet <host> <portnumber>`
- Creates a connection with a server running on `<host>` and listening on port `<portnumber>"
Testing the Echo Server With `telnet`

testmachine > ./echoserveri 10315
Connected to (testmachine.cs.iit.edu, 58700)
server received 18 bytes
server received 8 bytes

fourier > telnet testmachine.cs.iit.edu 10315
Trying 216.47.155.6...
Connected to testmachine.cs.iit.edu.
Escape character is '^[]'.
Can you hear me?
Can you hear me?
Hellow?
Hellow?
^[] 
telnet> quit
Connection closed.
fourier>
Web Server Basics

- Clients and servers communicate using the HyperText Transfer Protocol (HTTP)
  - Client and server establish TCP connection
  - Client requests content
  - Server responds with requested content
  - Client and server close connection (eventually)

- Current version is HTTP/1.1
  - RFC 2616, June, 1999.

http://www.w3.org/Protocols/rfc2616/rfc2616.html
Web Content

- Web servers return content to clients
  - content: a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type

- Example MIME types
  - text/html: HTML document
  - text/plain: Unformatted text
  - image/gif: Binary image encoded in GIF format
  - image/png: Binary image encoded in PNG format
  - image/jpeg: Binary image encoded in JPEG format

You can find the complete list of MIME types at:
http://www.iana.org/assignments/media-types/media-types.xhtml
Static and Dynamic Content

- The content returned in HTTP responses can be either **static** or **dynamic**
  - **Static content**: content stored in files and retrieved in response to an HTTP request
    - Examples: HTML files, images, audio clips
    - Request identifies which content file
  - **Dynamic content**: content produced on-the-fly in response to an HTTP request
    - Example: content produced by a program executed by the server on behalf of the client
    - Request identifies file containing executable code

- **Bottom line:** *Web content is associated with a file that is managed by the server*
URLs and how clients and servers use them

- Unique name for a file: URL (Universal Resource Locator)
- Example URL: `http://www.iit.edu:80/index.html`
- Clients use **prefix** (`http://www.iit.edu:80`) to infer:
  - What kind (protocol) of server to contact (HTTP)
  - Where the server is (`www.iit.edu`)
  - What port it is listening on (80)
- Servers use **suffix** (`/index.html`) to:
  - Determine if request is for static or dynamic content.
    - No hard and fast rules for this
    - One convention: executables reside in `cgi-bin` directory
  - Find file on file system
    - Initial “/” in suffix denotes home directory for requested content.
    - Minimal suffix is “/”, which server expands to configured default filename (usually, `index.html`)
HTTP Requests

- HTTP request is a request line, followed by zero or more request headers

- Request line: `<method> <uri> <version>`
  - `<method>` is one of GET, POST, OPTIONS, HEAD, PUT, DELETE, or TRACE
  - `<uri>` is typically URL for proxies, URL suffix for servers
    - A URL is a type of URI (Uniform Resource Identifier)
    - See http://www.ietf.org/rfc/rfc2396.txt
  - `<version>` is HTTP version of request (HTTP/1.0 or HTTP/1.1)

- Request headers: `<header name>: <header data>`
  - Provide additional information to the server
HTTP Responses

- HTTP response is a *response line* followed by zero or more *response headers*, possibly followed by *content*, with blank line ("\r\n") separating headers from content.

- **Response line:**
  
  `<version> <status code> <status msg>`
  
  - `<version>` is HTTP version of the response
  - `<status code>` is numeric status
  - `<status msg>` is corresponding English text
    - 200 OK Request was handled without error
    - 301 Moved Provide alternate URL
    - 404 Not found Server couldn’t find the file

- **Response headers:** `<header name>`: `<header data>`
  
  - Provide additional information about response
  - `Content-Type`: MIME type of content in response body
  - `Content-Length`: Length of content in response body
Example HTTP Transaction

$ { echo "GET /index.html HTTP/1.1"; echo "Host: www.iit.edu"; echo; sleep 1; } | nc www.iit.edu 80

HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Wed, 02 Nov 2022 06:03:56 GMT
Content-Type: text/html; charset=iso-8859-1
Content-Length: 238
X-Content-Type-Options: nosniff
Location: https://www.iit.edu/index.html
Cache-Control: max-age=1209600
Expires: Wed, 16 Nov 2022 06:03:56 GMT
X-Request-ID: v-22a5e508-5a74-11ed-b257-7334d81ceddf
Age: 671231
Via: varnish
X-Cache: HIT
X-Cache-Hits: 4
Connection: keep-alive

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="https://www.iit.edu/index.html">here</a>.</p>
</body></html>
Example HTTP Transaction, Take 2

```
$ telnet acme.com 80
Trying 23.93.76.124...
Connected to acme.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: acme.com

HTTP/1.1 200 OK
Server: thttpd/2.30 ??May2019
Content-Type: text/html; charset=UTF-8
Date: Thu, 10 Nov 2022 00:26:38 GMT
Accept-Ranges: bytes
Connection: close
Content-Length: 7956

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html lang="en">

...

Connection closed by foreign host.
```

- HTTP standard requires that each text line end with `\r\n`
- Blank line (`\r\n`) terminates request and response headers
Tiny Web Server

- **Tiny Web server described in the textbook**
  - Tiny is a sequential Web server
  - Serves static and dynamic content to real browsers
    - text files, HTML files, GIF, PNG, and JPEG images
  - 239 lines of commented C code
  - Not as complete or robust as a real Web server
    - You can break it with poorly-formed HTTP requests (e.g., terminate lines with “\n” instead of “\r\n”)

Illinois Tech CS351 Fall 2022
Tiny Operation

- Accept connection from client
- Read request from client (via connected socket)
- Split into `<method> <uri> <version>`
  - If method not GET, then return error
- If URI contains "cgi-bin" then serve dynamic content
  - (Would do wrong thing if had file "abcgi-bingo.html")
  - Fork process to execute program
- Otherwise serve static content
  - Copy file to output
Per-lecture feedback

- Better sooner rather than later!
- I can help with issues sooner.
- There is a per-lecture feedback form.
- The form is anonymous. (It checks that you’re at Illinois Tech to filter abuse, but I don’t see who submitted any of the forms.)
- https://forms.gle/qoeEbBuTYXo5FiU1A
- I’ll remind about this at each lecture.