CS214 Recitation Sec.7

Nov. 28, 2017

Topics

1. OSI
2. IPv4 vs IPv6
3. TCP vs UDP
4. Sockets

Open Systems Interconnection Model

- A technology standard maintained by the International Standards Organization (ISO)
- An abstract model of how *network protocols* and *equipment* should communicate and work together
- Contains *seven* layers in *two* groups: Host layers & Media layers
- Host layers perform *application-specific functions* such as data formatting, encryption, transmission, and connection management
- Media layers provide more primitive *network-specific functions* such as routing, addressing, and flow control

Open Systems Interconnection Model

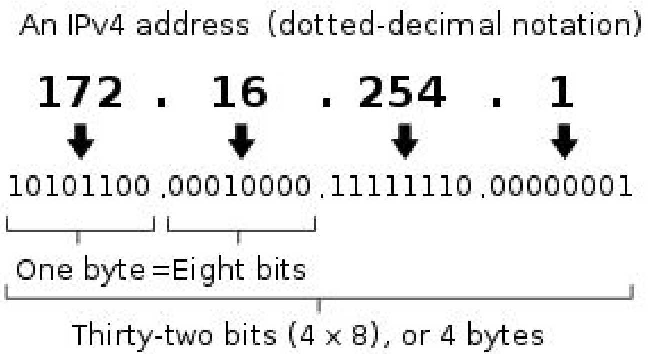
OSI Model				
Layer		Data unit	Function[3]	Examples
Host layers	7. Application	Data	High-level APIs, including resource sharing, remote file access, directory services and virtual terminals	HTTP, FTP, SMTP
	6. Presentation		Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption	ASCII, EBCDIC, JPEG
	5. Session		Managing communication sessions, i.e. continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes	RPC, PAP
	4. Transport	Segments	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing	TCP, UDP
Media layers	3. Network	Packet/Datagram	Structuring and managing a multi-node network, including addressing, routing and traffic control	IPv4, IPv6, IPsec, AppleTalk
	2. Data link	Bit/Frame	Reliable transmission of data frames between two nodes connected by a physical layer	PPP, IEEE 802.2, L2TP
	1. Physical	Bit	Transmission and reception of raw bit streams over a physical medium	DSL, USB

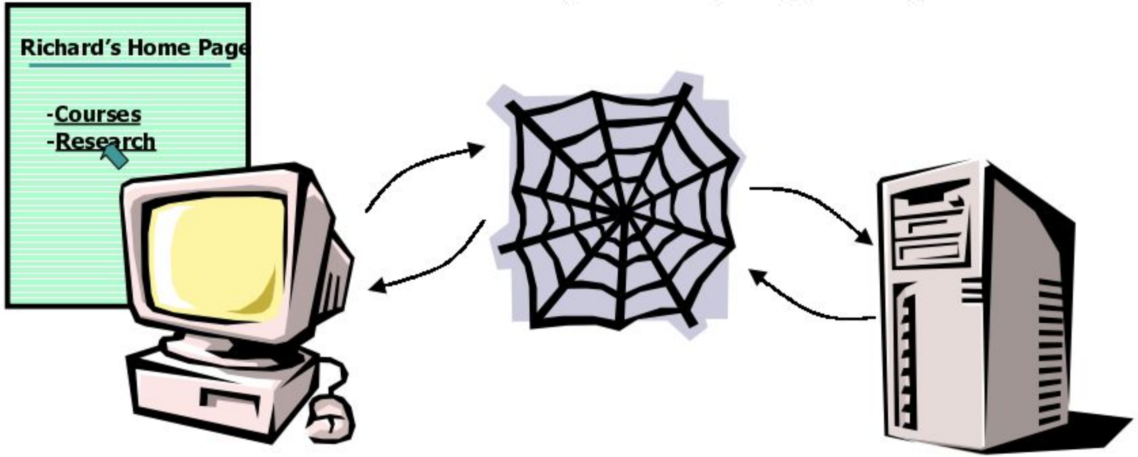
Benefits of OSI

- Simplifies the design of network protocols
- Ensure different types of equipment would all be compatible if built by different manufacturers
- Makes network designs more extensible as new protocols are easier to add to a layered architecture

How do we find the server

- Every computer on the Internet has an Internet address called an **IP** address (Internet Protocol)
- An IP address is four 8-bit numbers separated by dots
- Example: Remote Terminal Server at Rutgers python.cs.rutgers.edu 128.6.13.233





IPv4 vs IPv6

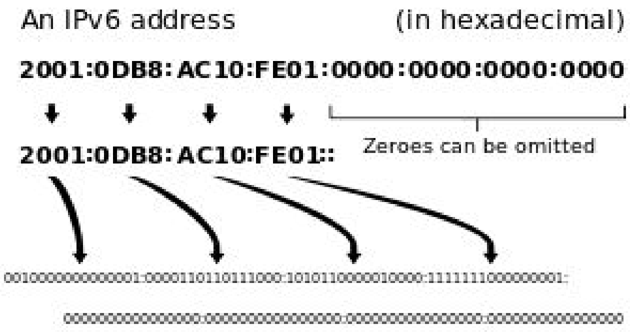
- "**IPv4**" is version 4 of the Internet Protocol how to send packets of information across a network from one machine to another
- Roughly **95%** of all packets on the Internet today are IPv4 packets
- A significant limitation of IPv4 is that source and destination addresses are limited to **32 bits** (8bits * 4)

IPv4 Header Format

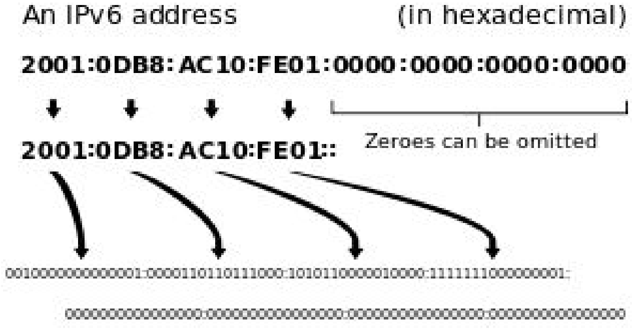
<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>4</td><td>32</td><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>8</td><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>12</td><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>20</td><td>160</td><td colspan="32">Options (if IHL > 5)</td></tr>
</table>

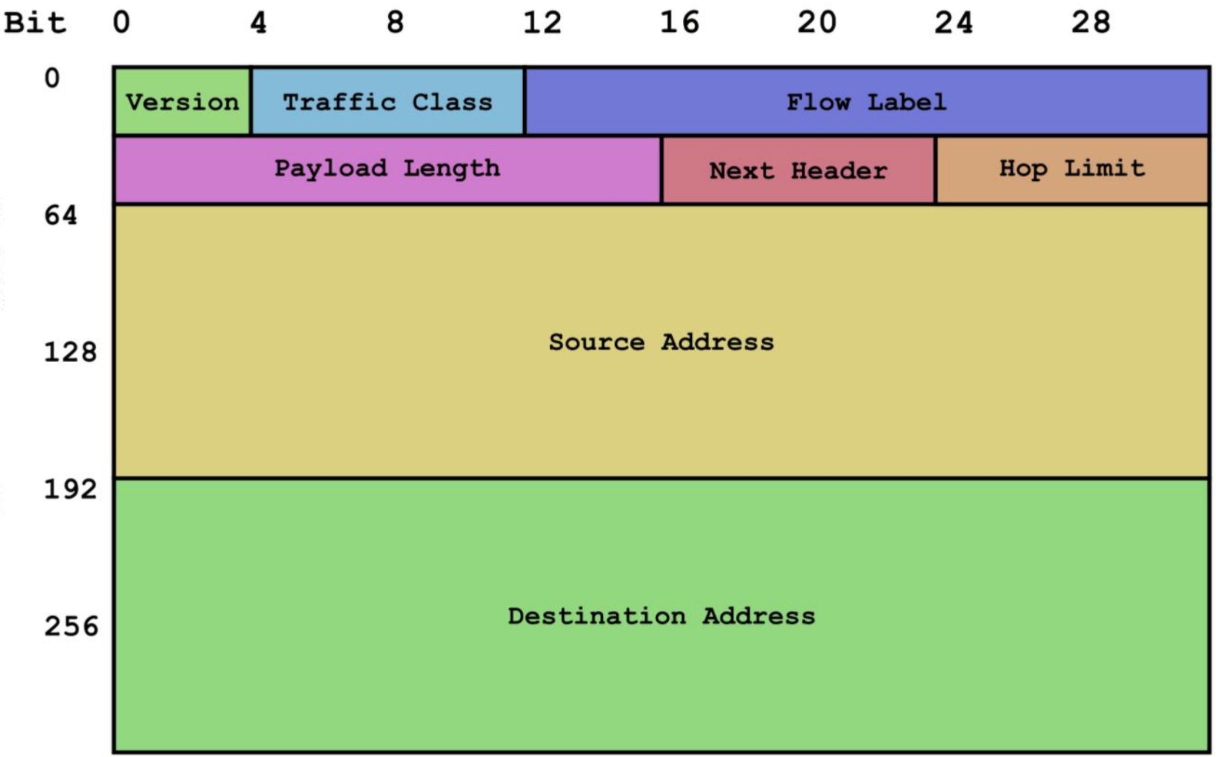
IPv4 vs IPv6

- Each IPv4 packet includes a very small header - typically **20 bytes** (more precisely, "octets"), that includes *a source and destination address*
- Conceptually the source and destination addresses can be split into two: a ***network number*** (the upper bits) and the lower bits represent a ***particular host number*** on that network.
- A newer packet protocol "**IPv6**" solves many of the limitations of IPv4 (e.g. makes routing tables simpler and **128 bit** addresses) however less than 5% of web traffic is IPv6 based. Example:



IPv6 packet header





TCP vs UDP

- UDP is a **connectionless protocol** that is built on top of IPv4 and IPv6
- UDP is very simple to use: 1. Decide the destination address and port and 2. send your data packet
- UDP makes **no guarantee** about whether the packets will arrive
- A typical use case for UDP is when **receiving up to date data is more important** than receiving all of the data

TCP vs UDP

- TCP is a **connection-based protocol** that is built on top of IPv4 and IPv6 (and therefore can be described as "**TCP/IP**" or "TCP over IP")
- TCP creates a ***pipe*** between two machines and under most conditions, bytes sent from one machine will eventually arrive at the other end without duplication or data loss
- TCP will **automatically manage** resending packets, ignoring duplicate packets, re-arranging out-of-order packets and changing the rate at which packets are sent
- To create a pipe between two machines, TCP use a three-way handshake mechanism which is known as ***SYN, SYN-ACK, and ACK***.

TCP Handshake

Host A **sends** a TCP **SYN**chronize packet to Host B

Host B receives A's **SYN**

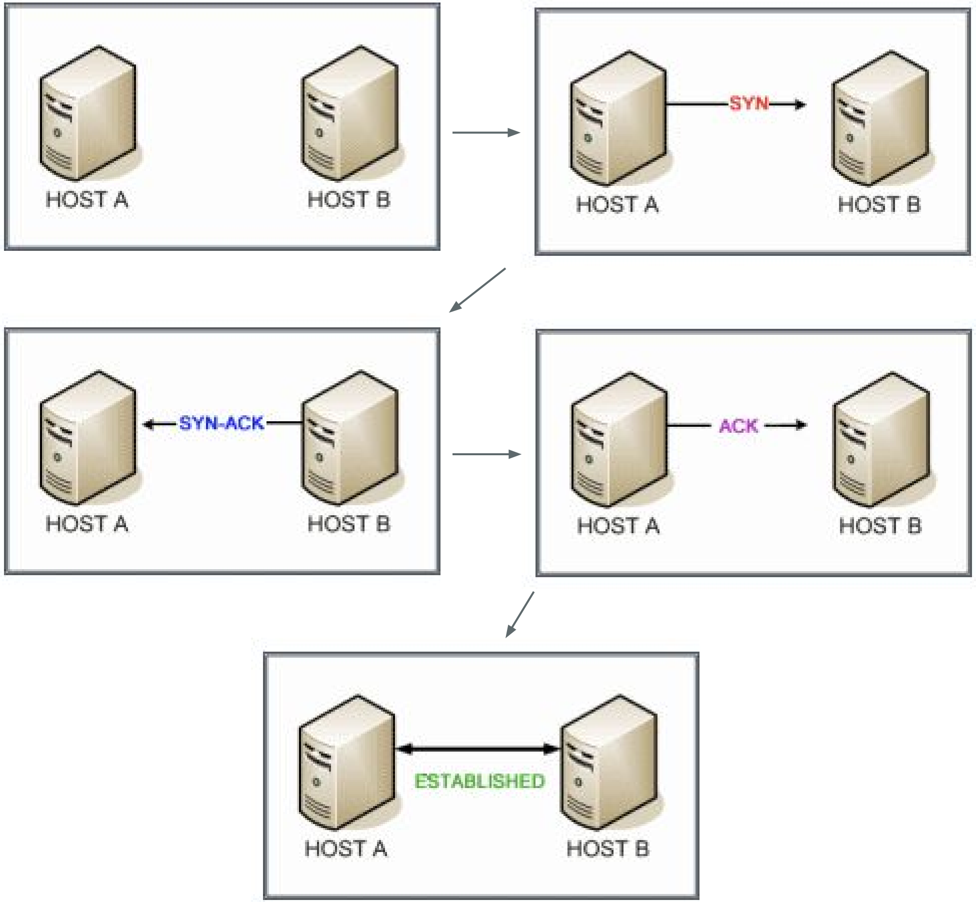
Host B **sends** a **SYN**chronize-**ACK**nowledgement

Host A receives B's **SYN-ACK**

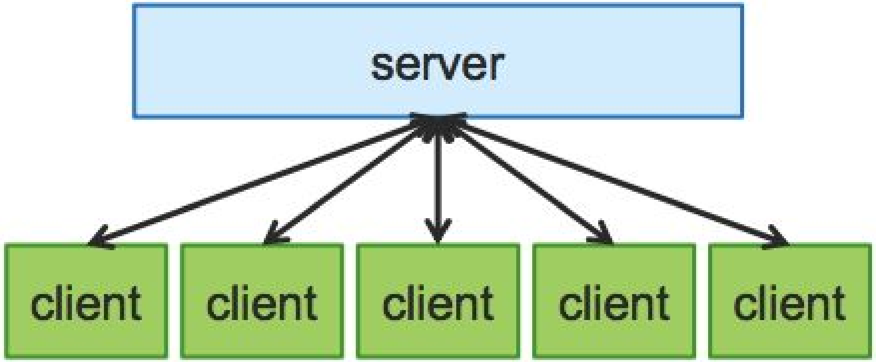
Host A **sends** **ACK**nowledge

Host B receives **ACK**.

TCP socket connection is ESTABLISHED.



Client-Server Model



All communications across a network happen over a **network socket**

- Client:
  - Initiates contact
  - Waits for server's response
- Server:
  - Well-known name
  - Waits for contact
  - Processes requests, sends replies

Socket

- One form of communication between processes, but it is used between processes on different machines
- Bi-directional
- Connection made via a **socket address**
- **IP address** is the destination of computer
- **Port number** is the destination of process

- A ***socket address*** is:
  - IP Address
  - Port Number

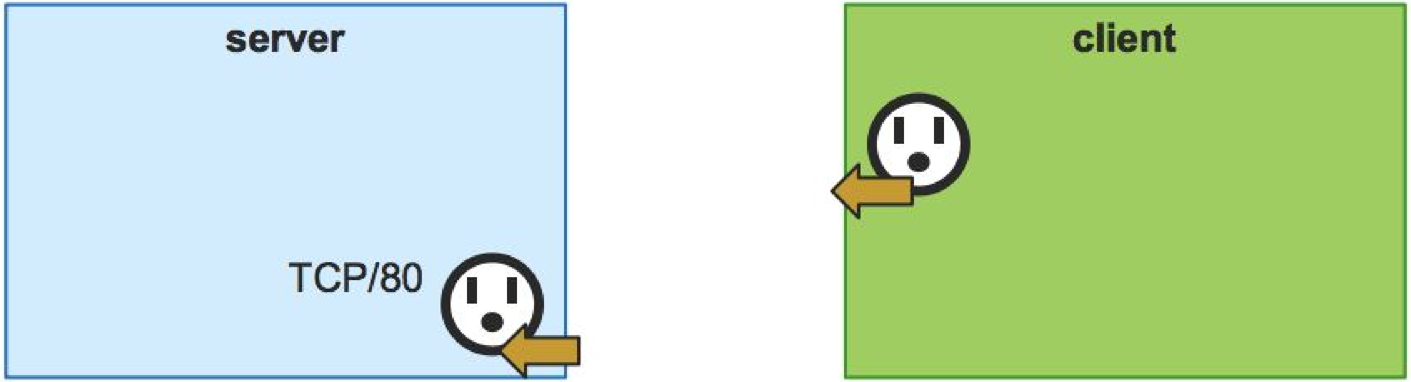
- A socket must also bind to a specific transport-layer protocol.
  - TCP
  - UDP

Socket

- **Port number** is a 16-bit unsigned interger (0 - 65535)
- A *unique resource* shared across the entire system (eg. port 80 can only be utilized by one process)
- Ports below 1024 are *reserved* by operating system
- *Public HTTP servers* always listen for new connections on **port 80**

Initializing a socket

- To listen for an incoming connection, and listen on a specific protocol/port (**Server Socket**)
- To connect to a “server socket” - remote computer (**Client**)



Get socket address with `getaddrinfo`

- The function `getaddrinfo` can convert a human readable domain name (e.g. www.cs.rutgers.edu) into an IPv4 and IPv6 address
- A linked-list of `addrinfo` structs will be returned after calling `getaddrinfo`

```
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;
};
```

Get socket address with `getaddrinfo`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(const char *restrict node,
                const char *restrict service,
                const struct addrinfo *restrict hints,
                struct addrinfo **restrict res);
```

- Parameters
  - `node`: host name or IP address to connect to
  - `service`: a port number ("80") or the name of a service (found /etc/services: "http")
  - `hints`: a filled out struct addrinfo

Using `getaddrinfo`

output:

```
~/2017F/CS 214/recitation_11_28 » ./addinfo
128.6.68.137
128.6.68.137
```

convert *www.cs.rutgers.edu* into an IPv4 address with `getaddrinfo`

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

struct addrinfo hints, *infoptr; // So no need to use memset global variables

int main() {

  hints.ai_family = AF_INET; // AF_INET means IPv4 only addresses

  int result = getaddrinfo("www.cs.rutgers.edu", NULL, &hints, &infoptr);
  if (result) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
    exit(1);
  }

  struct addrinfo *p;
  char host[256];

  for(p = infoptr; p != NULL; p = p->ai_next) {

    getnameinfo(p->ai_addr, p->ai_addrlen, host, sizeof(host), NULL, 0, NI_NUMERICHOST);
    puts(host);
  }

  freeaddrinfo(infoptr);
  return 0;
}
```

Using getaddrinfo

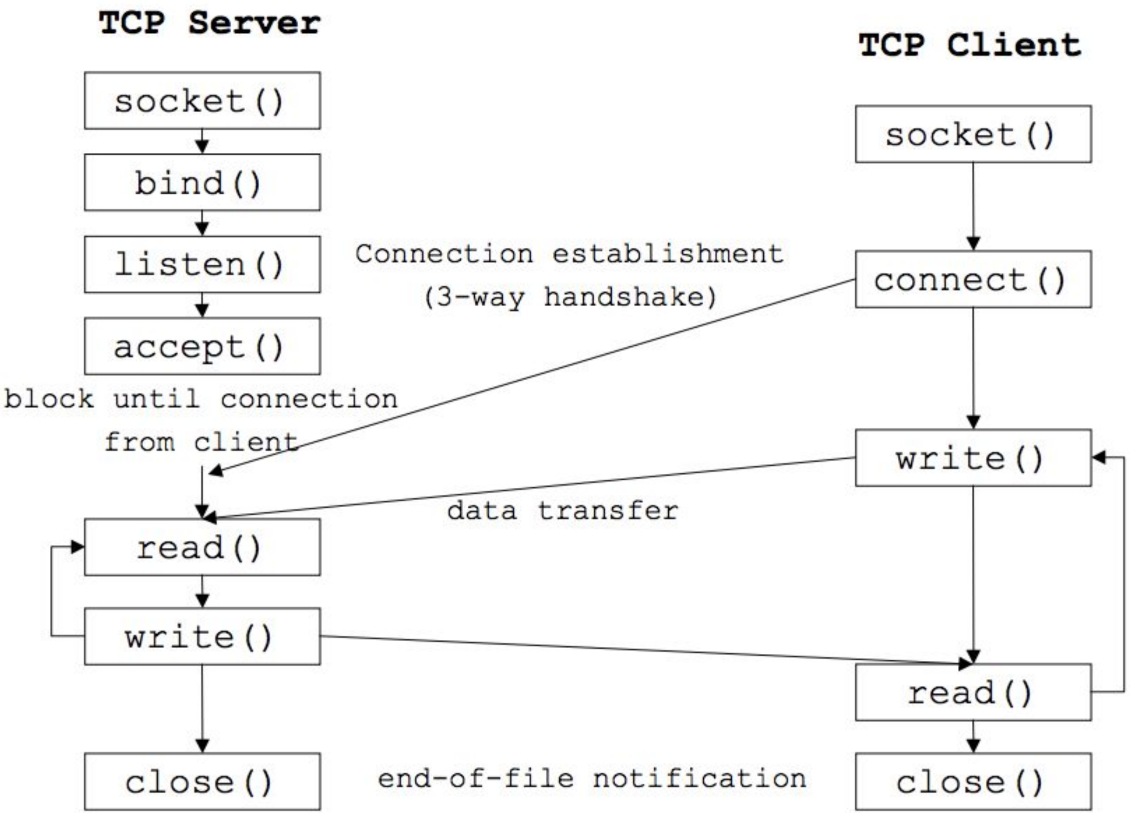
convert *www.cs.rutgers.edu* into an IPv6 address with `getaddrinfo`

output:

```
~/2017F/CS_214/recitation_11_28 » ./addinfo
getaddrinfo: nodename nor servname provided, or not known
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

struct addrinfo hints, *infoptr; // So no need to use memset global variables

int main() {

  hints.ai_family = AF_INET6; // Only want IPv6 (use AF_INET for IPv4)
  hints.ai_socktype = SOCK_STREAM; // Only want stream-based connection

  int result = getaddrinfo("www.cs.rutgers.edu", NULL, &hints, &infoptr);
  if (result) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
    exit(1);
  }

  struct addrinfo *p;
  char host[256];

  for(p = infoptr; p != NULL; p = p->ai_next) {

    getnameinfo(p->ai_addr, p->ai_addrlen, host, sizeof(host), NULL, 0, NI_NUMERICHOST);
    puts(host);
  }

  freeaddrinfo(infoptr);
  return 0;
}
```

TCP Server-Client Model



Creating a “Server Socket”

- **socket()**: Creates a new socket for a specific protocol (eg: TCP)
- **bind()**: Binds the socket to a specific port (eg: 80)
- **listen()**: Moves the socket into a state of listening for incoming connections.
- **accept()**: Accepts an incoming connection.

Creating a “Client Socket”

- **socket()**: Creates a new socket for a specific protocol (eg: TCP)
- **connect()**: Makes a network connection to a specified IP address and port.

socket()

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

- Create a socket.
  - Returns file descriptor or -1. Also sets `errno` on failure.
  - `family`: address family (namespace)
    - `AF_INET` for IPv4
    - other possibilities: `AF_INET6` (IPv6), `AF_UNIX` or `AF_LOCAL` (Unix socket), `AF_ROUTE` (routing)
  - `type`: style of communication
    - `SOCK_STREAM` for TCP (with `AF_INET`)
    - `SOCK_DGRAM` for UDP (with `AF_INET`)
  - `protocol`: protocol within family
    - typically 0

bind()

```
int bind (int sockfd, struct sockaddr*
    myaddr, int addrlen);
```

- Bind a socket to a local IP address and port number
  - Returns 0 on success, -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `myaddr`: includes IP address and port number
    - IP address: set by kernel if value passed is `INADDR_ANY`, else set by caller
    - port number: set by kernel if value passed is 0, else set by caller
  - `addrlen`: length of address structure
    - `= sizeof (struct sockaddr_in)`

listen()

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

- Put socket into passive state (wait for connections rather than initiate a connection)
  - Returns 0 on success, -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `backlog`: bound on length of unaccepted connection queue (connection backlog); kernel will cap, thus better to set high
  - Example:

```
if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
}
```

Establishing a Connection

- Include file `<sys/socket.h>`

```
int connect (int sockfd, struct
    sockaddr* servaddr, int addrlen);
```

  - Connect to another socket.

```
int accept (int sockfd, struct sockaddr*
    cliaddr, int* addrlen);
```

  - Accept a new connection. Returns file descriptor or -1.

connect()

```
int connect (int sockfd, struct
    sockaddr* servaddr, int addrlen);
```

- Connect to another socket.
  - Returns 0 on success, -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `servaddr`: IP address and port number of server
  - `addrlen`: length of address structure
    - `= sizeof (struct sockaddr_in)`
- Can use with UDP to restrict incoming datagrams and to obtain asynchronous errors

accept()

```
int accept (int sockfd, struct sockaddr* cliaddr,
    int* addrlen);
```

- Block waiting for a new connection
  - Returns file descriptor or -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `cliaddr`: IP address and port number of client (returned from call)
  - `addrlen`: length of address structure = pointer to `int` set to `sizeof (struct sockaddr_in)`

- `addrlen` is a **value-result** argument
  - the caller passes the size of the address structure, the kernel returns the size of the client's address (the number of bytes written)

Sending and Receiving Data

```
int send(int sockfd, const void * buf,
    size_t nbytes, int flags);
```

- Write data to a stream (TCP) or “connected” datagram (UDP) socket.
  - Returns number of bytes written or -1.

```
int recv(int sockfd, void *buf, size_t
    nbytes, int flags);
```

- Read data from a stream (TCP) or “connected” datagram (UDP) socket.
  - Returns number of bytes read or -1.

send()

```
int send(int sockfd, const void * buf, size_t
    nbytes, int flags);
```

- Send data un a stream (TCP) or "connected" datagram (UDP) socket
  - Returns number of bytes written or -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `buf`: data buffer
  - `nbytes`: number of bytes to try to write
  - `flags`: control flags
    - MSG_PEEK: get data from the beginning of the receive queue without removing that data from the queue

recv()

```
int recv(int sockfd, void *buf, size_t nbytes,
    int flags);
```

- Read data from a stream (TCP) or “connected” datagram (UDP) socket
  - Returns number of bytes read or -1, sets `errno` on failure
  - Returns 0 if socket closed
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `buf`: data buffer
  - `nbytes`: number of bytes to try to read
  - `flags`: see man page for details; typically use 0

Building a simple TCP Client

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int s;
    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

    struct addrinfo hints, *result;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET; /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* TCP */

    s = getaddrinfo("www.cs.rutgers.edu", "80", &hints, &result);
    if (s != 0) {
            fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
            exit(1);
    }

    if(connect(sock_fd, result->ai_addr, result->ai_addrlen) == -1){
                perror("connect");
                exit(2);
        }

    char *buffer = "GET / HTTP/1.0\r\n\r\n";
    printf("SENDING: %s", buffer);
    printf("===\n");
```

```c
    write(sock_fd, buffer, strlen(buffer));


    char resp[1000];
    int len = read(sock_fd, resp, 999);
    resp[len] = '\0';
    printf("%s\n", resp);

    return 0;
}
```

Building a simple TCP Client

```
~/2017F/CS 214/recitation_11_28 » ./tcp_client
SENDING: GET / HTTP/1.0


===
HTTP/1.1 301 Moved Permanently
Date: Wed, 29 Nov 2017 17:13:04 GMT
Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.1e-fips mod_auth_gssapi/1.3.1 mod_auth
_kerb/5.4 mod_fcgid/2.3.9 mod_nss/2.4.6 NSS/3.19.1 Basic ECC PHP/5.4.16 SVN/1.7.
14 mod_wsgi/3.4 Python/2.7.5
Location: http://www.cs.rutgers.edu/
Content-Length: 234
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!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="http://www.cs.rutgers.edu/">here</a>.</p>
</body></html>
```

Building a simple TCP Server

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
    int s;
    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

    struct addrinfo hints, *result;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    s = getaddrinfo(NULL, "1234", &hints, &result);
    if (s != 0) {
            fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
            exit(1);
    }

    if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
        perror("bind()");
        exit(1);
    }
```

```c
    if (listen(sock_fd, 10) != 0) {
        perror("listen()");
        exit(1);
    }

    struct sockaddr_in *result_addr = (struct sockaddr_in *) result->ai_addr;
    printf("Listening on file descriptor %d, port %d\n", sock_fd, ntohs(result_addr->sin_port));

    printf("Waiting for connection...\n");
    int client_fd = accept(sock_fd, NULL, NULL);
    printf("Connection made: client_fd=%d\n", client_fd);

    char buffer[1000];
    int len = read(client_fd, buffer, sizeof(buffer) - 1);
    buffer[len] = '\0';

    printf("Read %d chars\n", len);
    printf("===\n");
    printf("%s\n", buffer);

    return 0;
}
```

Building a simple TCP Server

```
~/2017F/CS 214/recitation_11_28 » ./tcp_server
Listening on file descriptor 3, port 1234
Waiting for connection...
```