# CSE 333 – SECTION 8

Client-Side Network Programming

# Overview

- Domain Name Service (DNS) Review
- Client side network programming steps and calls
- dig and ncat tools

# Network programming for the client side

- Recall the five steps, here's the corresponding calls:
  1. getaddrinfo() to figure out IP address and port to talk to
  2. socket() for creating a socket
  3. connect() to connect to the server
  4. read() and write() to transfer data through the socket
  5. close() to close the socket

# Network programming for the client side

- Recall the five steps, here's the corresponding calls:
  1. **getaddrinfo() to figure out IP address and port to talk to**
  2. socket() for creating a socket
  3. connect() to connect to the server
  4. read() and write() to transfer data through the socket
  5. close() to close the socket

# Network Addresses

- For IPv4, an IP address is a 4-byte tuple
- - e.g., 128.95.4.1 (80:5f:04:01 in hex)
- For IPv6, an IP address is a 16-byte tuple
- - e.g., 2d01:0db8:f188:0000:0000:0000:0000:1f33
- ‣ 2d01:0db8:f188::1f33 in shorthand

# DNS – Domain Name System/Service

- A hierarchical distributed naming system any resource connected to the Internet or a private network.
- Resolves queries for names into IP addresses.
- The sockets API lets you convert between the two.
  - Aside: getnameinfo() is the inverse of getaddrinfo()
- Is on the application layer on the Internet protocol suite.

# Dig demo

dig +trace attu.cs.washington.edu

# Resolving DNS names

- The POSIX way is to use **getaddrinfo( ).**
- Set up a “hints” structure with constraints, e.g. IPv6, IPv4, or either.
- Tell getaddrinfo( ) which host and port you want resolved.
- Host - a string representation: DNS name or IP address
- getaddrinfo() gives you a list of results in an “addrinfo” struct.

# IPv4 address structures

```
// Port numbers and addresses are in *network order*.

// A mostly-protocol-independent address structure.
struct sockaddr {
    short int   sa_family;   // Address family; AF_INET, AF_INET6
    char        sa_data[14]; // 14 bytes of protocol address
};

// An IPv4 specific address structure.
struct sockaddr_in {
    short int          sin_family;   // Address family, AF_INET == IPv4
    unsigned short int sin_port;     // Port number
    struct in_addr     sin_addr;     // Internet address
    unsigned char      sin_zero[8];  // Same size as struct sockaddr
};

struct in_addr {
    uint32_t s_addr; // IPv4 address
};
```

# IPv6 address structures

```
// A structure big enough to hold either IPv4 or IPv6 structures.
struct sockaddr_storage {
    sa_family_t ss_family;    // address family
    // a bunch of padding; safe to ignore it.
    char       __ss_pad1[_SS_PAD1SIZE];
    int64_t    __ss_align;
    char       __ss_pad2[_SS_PAD2SIZE];
};
// An IPv6 specific address structure.
struct sockaddr_in6 {
    u_int16_t        sin6_family;     // address family, AF_INET6
    u_int16_t        sin6_port;       // Port number
    u_int32_t        sin6_flowinfo;   // IPv6 flow information
    struct in6_addr  sin6_addr;       // IPv6 address
    u_int32_t        sin6_scope_id;   // Scope ID
};
struct in6_addr {
    unsigned char s6_addr[16];        // IPv6 address
};
```

# getaddrinfo() and structures

```
int getaddrinfo(const char *hostname,          // hostname to look up
                const char *servname,          // service name
                const struct addrinfo *hints,  // desired output type
                struct addrinfo **res);        // result structure

// Hints and results take the same form. Hints are optional.
struct addrinfo {
    int             ai_flags;      // Indicate options to the function
    int             ai_family;     // AF_INET, AF_INET6, or AF_UNSPEC
    int             ai_socktype;   // Socket type, (use SOCK_STREAM)
    int             ai_protocol;   // Protocol type
    size_t          ai_addrlen;    // INET_ADDRSTRLEN, INET6_ADDRSTRLEN
    struct sockaddr *ai_addr;      // Address (input to inet_ntop)
    char            *ai_canonname; // canonical name for the host
    struct addrinfo *ai_next;      // Next element (It's a linked list)
};

// Converts an address from network format to presentation format
const char *inet_ntop(int af,                          // family (see above)
                      const void * restrict src,      // in_addr or in6_addr
                      char * restrict dest,           // return buffer
                      socklen_t size);                // length of buffer
```

# Generating these structures

```
#include <stdlib.h>
#include <arpa/inet.h>

int main(int argc, char **argv) {
  struct sockaddr_in sa;    // IPv4
  struct sockaddr_in6 sa6; // IPv6

  // IPv4 string to sockaddr_in.
  inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr));

  // IPv6 string to sockaddr_in6.
  inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));
  return EXIT_SUCCESS;
}
```

# Generating these structures

```
#include <stdlib.h>
#include <arpa/inet.h>

int main(int argc, char **argv) {
  struct sockaddr_in6 sa6;        // IPv6
  char astring[INET6_ADDRSTRLEN]; // IPv6

  // IPv6 string to sockaddr_in6.
  inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));

  // sockaddr_in6 to IPv6 string.
  inet_ntop(AF_INET6, &(sa6.sin6_addr), astring, INET6_ADDRSTRLEN);
  printf("%s\n", astring);
  return EXIT_SUCCESS;
}
```

# DNS Resolution Demo

dnsresolve.cc

# Network programming for the client side

- Recall the five steps, here's the corresponding calls:
  1. getaddrinfo() to figure out IP address and port to talk to
  2. **socket() for creating a socket**
  3. connect() to connect to the server
  4. read() and write() to transfer data through the socket
  5. close() to close the socket

# socket() – Create the socket

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain,  // e.g. PF_NET, PF_NET6
           int type,    // e.g. SOCK_STREAM, SOCK_DGRAM
           int protocol); // Usually 0
```

Note that socket() just creates a socket, it isn’t bound yet to a local address.

# Demo

socket.cc

# Network programming for the client side

- Recall the five steps, here's the corresponding calls:
  1. getaddrinfo() to figure out IP address and port to talk to
  2. socket() for creating a socket
  3. **connect() to connect to the server**
  4. read() and write() to transfer data through the socket
  5. close() to close the socket

# connect() – Establish the connection

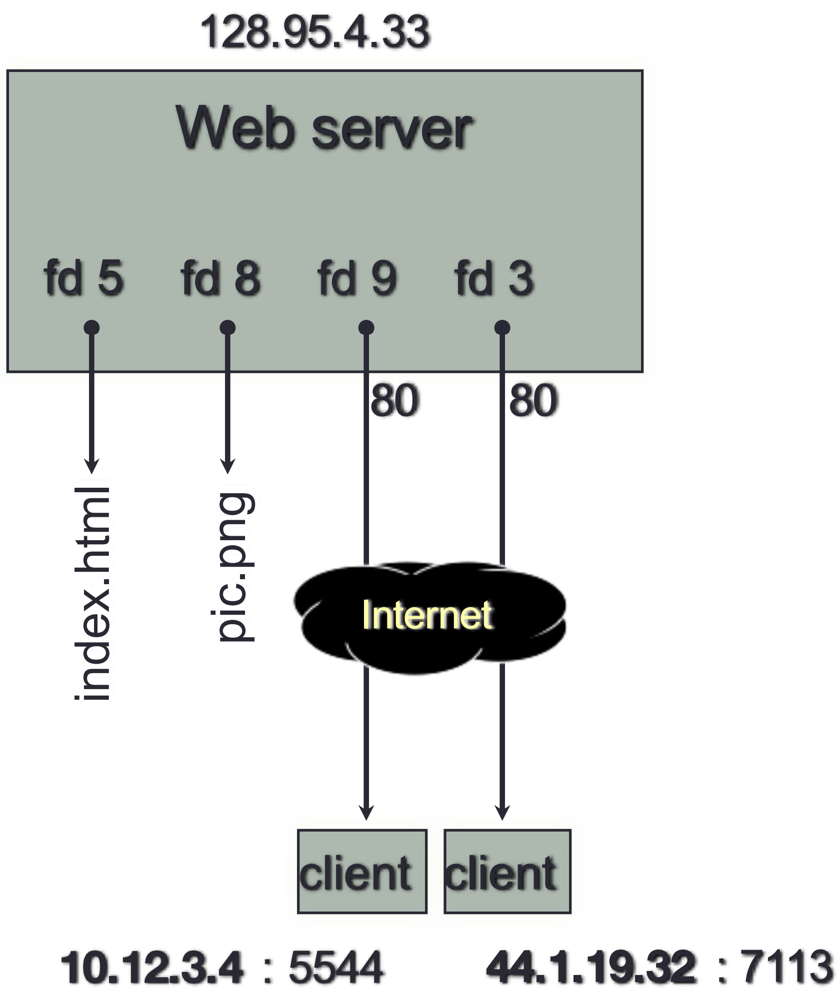
```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd,    // socket fd from step 2
            struct sockaddr *serv_addr,  // server info
                                         // from step 1
            int addrlen);   // size of serv_addr struct
```

# Demo (Along with ncat demo)

connect.cc

(nc –lv 5454 to create listener)

# Pictorially




## OS's descriptor table

| file descriptor | type | connected to? |
|---|---|---|
| 0 | pipe | stdin (console) |
| 1 | pipe | stdout (console) |
| 2 | pipe | stderr (console) |
| 3 | TCP socket | local: 128.95.4.33:80<br>remote: 44.1.19.32:7113 |
| 5 | file | index.html |
| 8 | file | pic.png |
| 9 | TCP socket | local: 128.95.4.33:80<br>remote: 10.12.3.4:5544 |

# Network programming for the client side

- Recall the five steps, here's the corresponding calls:
  1. getaddrinfo() to figure out IP address and port to talk to
  2. socket() for creating a socket
  3. connect() to connect to the server
  4. **read() and write() to transfer data through the socket**
  5. close() to close the socket

# read() and write()

- By default, both are blocking calls
- read() will wait for some data to arrive, then immediately read whatever data has been received by the network stack
    - Might return less data read than asked for
    - Blocks while data isn't received
- conversely, write() enqueues your data to OS' send buffer, then returns while OS does the rest in the background
    - When write returns the receiver probably hasn't received the data yet
    - When the send buffer fills up, write() will also block

# Demo (Along with more ncat)

sendreceive.cc

(nc –l 5454 to create listener)

# Network programming for the client side

- Recall the five steps, here's the corresponding calls:
    1. getaddrinfo() to figure out IP address and port to talk to
    2. socket() for creating a socket
    3. connect() to connect to the server
    4. read() and write() to transfer data through the socket
    5. **close() to close the socket**

# close() – Close the connection

```
#include <unistd.h>

int close(int sockfd);
```

Remember to close the socket when you're done!

# Section Exercise

- The TA has set up a game server for you to communicate with (gameserver.py)
- Using the sample client code from lecture and what you know about I/O calls in C, your job is to implement a C client called gameclient.cc such that you can communicate with the game server much like you can with the netcat tool