Introduction to Network Programming: Communication Endpoints

- **Network Application Programming Interfaces (Network API)**
  - **Application Programming Interface**:
    - **Application Programming Interface (API)** = is a **collection** of **library functions** through which **application programmers** (= ***you*** can gain **access** to the **services** (provided by the system)

  - **Network APIs**:
    - The **network communication services** are accessed through **2 commonly used *network APIs***:
      - **Berkeley *Sockets* API**
      - **(ATT's) Transport Layer Interface (TLI)** --- has been **superseded** by **XTI**

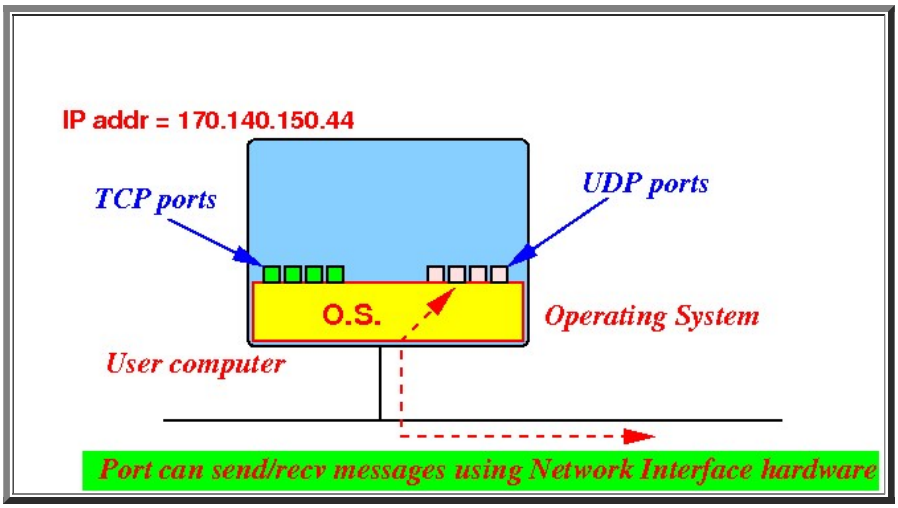
  - **Today:**
    - **Berkeley *sockets*** is now **by far** the **most popular Network API**

    (We will learn **Berkeley sockets**....)

- **Communication end points**
  - **Communication *endpoint***:
    - **Communication endpoint** = a **logical location** where **messages** are **transmitted/received**
  - **2 kinds** of **communication endpoints**
    - ***User program* endpoint** = an **object** used in an **user program** that represents a **communication endpoint**
    - ***(Operating) System* endpoint** = an **object** used in the **Operating System** that represents a **communication endpoint**

- **Communication end-points in the *Operating System***
  - **Ports:**
    - **Port** = a **communication end-point** inside the **Operating System**
      - **Ports** can make use of the **network interface card** of the **computer** to transmit **messages**

    Graphically:



- **Type of ports**

- **Types** of **ports**:

  - The **Operating System** commonly implements **2 different types** of **ports** that provide ***different*** **type** of **communication services**

  - **User *Datagram* Protocol (UDP) ports**:

    - **UDP ports** provides ***unreliable*** **communication service**

  - **Transport Control Protocol (TCP) ports**:

    - **TCP ports** provides ***reliable*** **communication service**

- **Communication end-points in a *computer program***

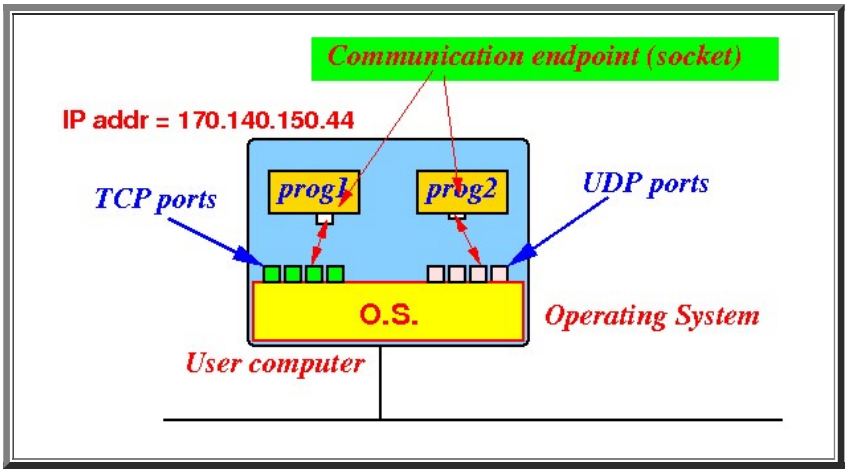
  - **Socket:**

    - ***Socket*** = a **communication endpoint** in a **network program**

      - The **network application program** can ***send*** **and** ***receive*** **messages** through ***sockets***

    - A **socket** must be ***associated*** **(bound)** to some **port** inside the **Operating System** !!!

  - **Network programming model** in **computer programs**



Explanation:

- A **socket** can be ***bound*** to some **port** in the **Operating System**

- The **computer program** will use the **socket** to transmit **message** through the **port** to which the **socket** was **bound**

- **Types of Sockets**

  - ***Different*** **types** of **sockets**:

    - There are ***different*** **types** of **sockets**

    - Each **type** of **socket** provides a specific ***quality of service***

  - There are **two types** of **sockets** in use today

    - **UDP sockets**

      - A **UDP socket** is **bound** to an ***UDP*** **port**

      - **UDP sockets** will therefore provides ***unreliable*** **communication** service

      - The **UDP socket** uses a ***packet*-oriented protocol**: a **unit** of **transmission** is a **packet**

- **TCP sockets**

  - A **TCP socket** is **bound** to an ***TCP* port**

  - **TCP sockets** will therefore provides ***reliable* communication** service

  - **TCP socket** uses a ***byte*-oritented protocol**: the **unit** of **transmission** is a ***byte***

Introduction to Network Programming: socket variables

- **Sockets inside a *computer program*: socket *variable***
  - **Recall:**
    - A **socket** is an **end-point of communication**
    - **Network application program** can **send and receive messages** thorugh a **socket**
  - **Socket variables:** representing a **socket** inside a **program**
    - **Socket *variable*** = a **variable** used in a **computer program** that **represents/identifies** a **socket**
    - **Implementation** in **UNIX**:
      - A **socket variable** is just an **integer variable**
      - The ***value*** of the **socket (integer) variable** is used to **identify** an ***entry*** in the **communication ports (= a data structure)** **inside** the **(UNIX) Operating System**

      (If you had taken **CS450**: a **socket** is like a **UNIX file descriptor**)

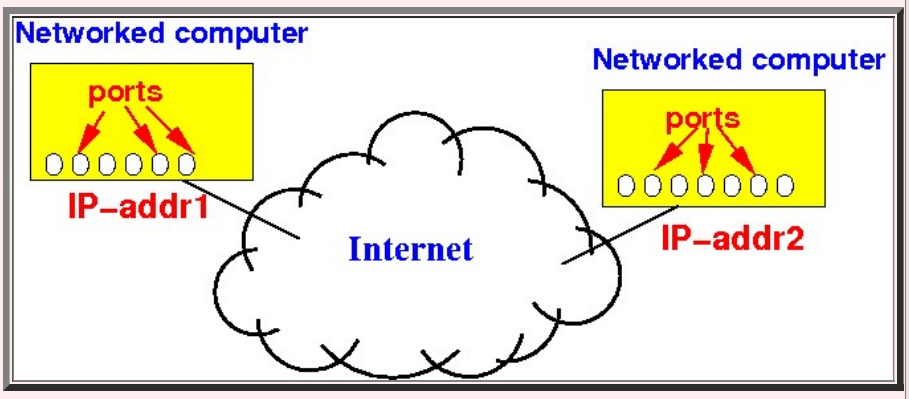
      Don't worry if you had not taken CS450; you can understand this material without it.
- **Operations on a socket**
  - **Operations on a *socket*:**
    - You can perform a **write operation** on a **socket (variable)**
      - **writing** some **data** to a **socket (variable)** will **transmit** the **data** in a **message**
    - You can perform a **read operation** on a **socket (variable)**
      - **reading** a **socket (variable)** will **receive** a **message** (that was **transmitted** by some network program)

      If there is **no message** in the ***receive buffer***, the **read operation** will **block** until a **message** is **received**

Overview network programming

- **Overview network communication**
  - **Infra-structure** to provide **network communication**:

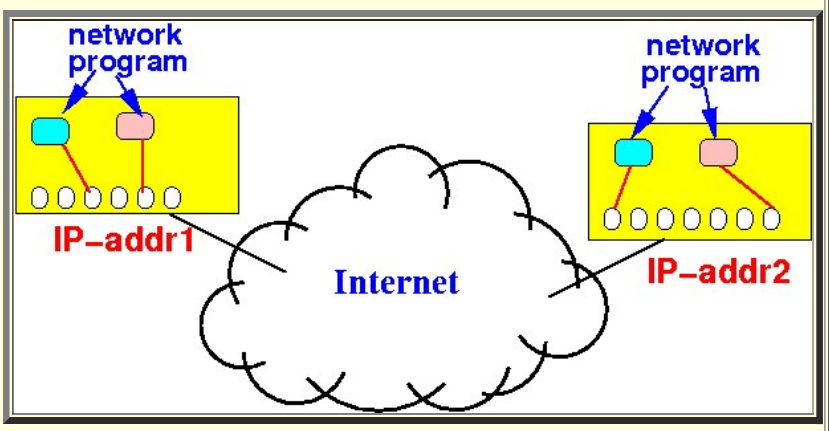
    - A **networked computer** on the **Internet** has $2^{16} = 65536$ **network** ***ports***:

      Each **networked computer** has a ***unique*** **IP address** (like a **telephone number**)

  - **Network programming:**

    - A **networked program** will **associate** a **socket variable** with a **port**:

    - **Networked programs** will **send messages** to **each other** using:

      ```
      (IP-address, port#)
      ```

      as **identification**

Network Programming with UDP sockets

- **C-language *Header files* for Network Programming**
  - **Network programs** written in **C** must include the following **header files** to define the **necessary *data structures*** used by the **Socket API**:

```
#include <stdio.h>           // Standard IO, all C programs need this
#include <stdlib.h>          // Standard Library, all C programs need this

#include <sys/types.h>       // System variable types
#include <sys/socket.h>      // Type definitions related to sockets
#include <netinet/in.h>      // Type definitions related to Internet protocol

#include <arpa/inet.h>       // Network utility functions
#include <netdb.h>           // Name services functions
```

Postscript - comments

- **Advanced network programming: non-blocking sockets**
  - **Default** socket behavior for the **read operation**:

    - The **read operation** on a **socket** by default is:
      - ***blocking***

      The **program** will ***wait*** until a **message** on the **socket** is **received**; then the **read operation** will **return** (with the **data** in the **newly received message**)

  - **Non-blocking** socket:

    - You can **change** a **socket** into ***non-blocking*** using the following **C-function call**:

```
fcntl(socket_variable, F_SETFL, fcntl(recvfd, F_GETFL)|O_NDELAY);
```

  - We will learn to use ***blocking*** **sockets** only....

- **Network programming resources**
  - **Socket programming guides**:

    - A very good introduction to network programming by L. Besaw: click here .
    - A advanced socket API documentation: click here .

Intro to UDP

- **Properties of UDP communication**
  - **Properties** of **UDP communciation**:

    - **Block oriented**:

      - A **UDP message** is **transmitted** as a ***whole***

      - The **receiver** will **receive** a transmitted **UDP message** as a ***whole***

        I.e.:

        - The **receiver** will **receive** what was **sent**

        - The **receiver** will **not** receive a ***portion*** of the **transmitted message**

    - **Unreliable**:

      - A **message** transmitted through **UDP** is **"send" and "done"** with no guarantee whatsoever

        So:

        - You can **transmit** a **UDP message** ***even*** when there is ***no*** **receiver** at the ***receiving*** **end** !!!

        (It can be quite **disturbing** to **novice users**)

Creating a UDP socket

- Creating a socket in general
  - *Defining* a socket variable in general:

```
int s;

s = socket(domain, socket-type, protocol-used);  // Creates a socket
```

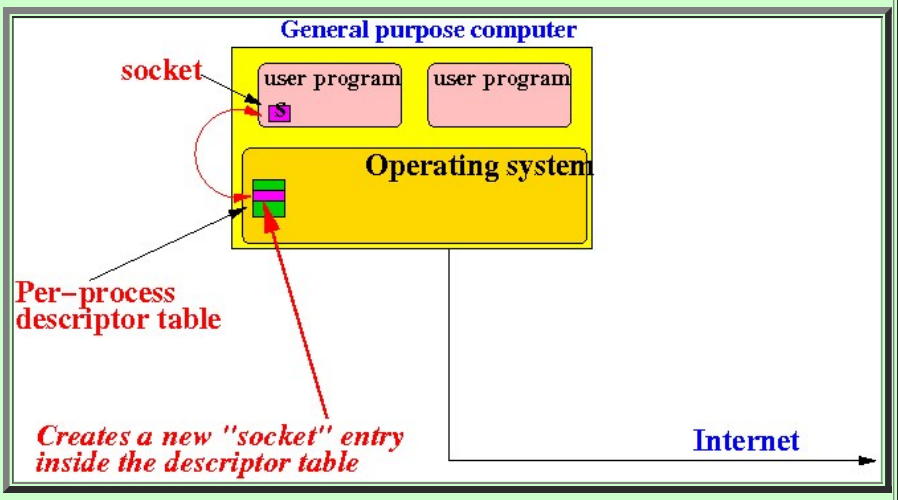
  - Meaning of the parameters:

```
domain parameter:

     PF_INET  = will use the IPv4 protocol (PF = protocol family)
     PF_INET6 = will use the IPv6 protocol


socket-type parameter:

     SOCK_DGRAM  = UDP socket
     SOCK_STREAM = TCP socket


protocol parameter:

     used to select a network protocol that will/can provide the service

     0 = use the default network protocol

          SOCK_DGRAM's  default protocol is:  UDP
          SOCK_STREAM's default protocol is:  TCP
```

  - Result of the `socket` system call:

    - The `socket()` *system call* will:
      - create an entry in the ("per-process") descriptor table inside the Operating System

    Graphically:



    - The descriptor table is used to:
      - record the state of all *opened* system resources by the process (running program)

    Example of system resources:

      - Files
      - Pipes
      - Sockets
      - Etc., etc

- Creating a UDP socket
  - System call to create a UDP socket:

```
int s;
```

```
s = socket(PF_INET, SOCK_DGRAM, 0);  // Creates a IPv4 UDP socket
```

- **Associating a UDP socket with an UDP port (in the kernel)**
  - **Fact:**
    - **After creating** a **UDP socket**, we **still** need to ***associate*** the **socket** with an **UDP port** inside the **Operating System**

  - **But first**:
    - We need to **learn**:
      - **How to** specify a **Internet network address** (that **identifies** a **UDP port**)

Identifying a host UDP endpoint (IP-Addr,Port#)

- **The socket address structure for *Internet* programming in C**
  - **Sockets**:

    - **(Network) socket** = = a **program endpoint** for **sending/receiving data**

    - A **socket** is **associated** with a **network port** that is **identified** by a **socket address**

    - **Socket address** consists of:

      - **IP address** of the **host**
      - The **port #** of the **kernet (UDP) port**

  - **Data structure** in **C** used to **store Socket addresses** of an **(Internet) UDB port**:

```
struct sockaddr_in
{
   sa_family_t     sin_family;   /* Socket's address family
                                    - Must contain the value AF_INET */

   in_port_t       sin_port;     /* Socket's Internet Port #
                                    - a 2 bytes port number */
   struct  in_addr sin_addr;     /* Socket's Internet Address (IP-address)
                                    - a 4 bytes IP address  */
};
```

    Meaning of the **variables**:

    - `sin_family`: (Internet socket family)

      - The `sin_family` ***must*** be set to the **(constant) value** `AF_INET` for the **IP protocol**

    - `sin_port`: (the **data type** of this **variable** is **short** --- Internet socket port#)

      - If `sin_port > 0` then:

        - `sin_port` = the **port#** of some **port** in the **Operating System**

      - If `sin_port = 0` then:

        - the **system** will **pick** an ***unassigned* port** for you

    Very important:

      - `sin_port` must be **stored** in ***network* byte order** (more later) !!!!

    - `sin_addr`: (the **data type** of this **variable** is a `struct` --- one of the field (`s_addr` contains the IP address)

      - `sin_addr` = the **4 byte IP address**

        Note:

        - There is a **special *wildcard* IP address** named: `INADDR_ANY`

          We will **discuss** this in the ***multi-homed* IP host** section **later**

    Very important:

      - `sin_addr` must ***also*** be **stored** in ***network* byte order** (more later) !!!!

  - **More** on the `sin_addr` field:

    - The `sin_addr` variable is a `struct` **typed variable** used to store an **IP address**:

```
/* Internet address. */
struct in_addr
{
```

```
        uint32_t  s_addr;     /* 4 bytes IP address in network byte order */
    } sin_addr;
```

- **Pre-liminary example**: how to initialize an the IPv4 address structure

```
(1) We first define a socket address variable:

      struct sockaddr_in   in_addr;

(2) Then we initialize the socket address:

      in_addr.sin_family      = AF_INET;     /* Protocol domain is: Internet */
      in_addr.sin_addr.s_addr = ....;        /* IP address in netw byte order */
      in_addr.sin_port        = ....;        /* Port # in netw byte order */
```

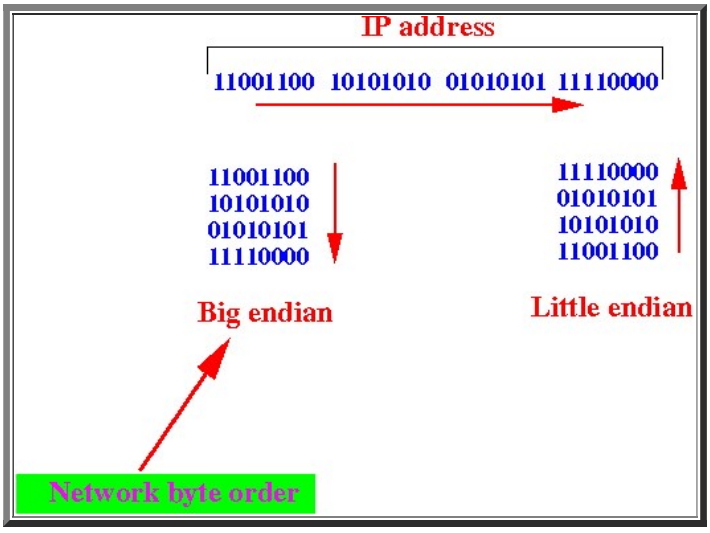
(We will complete the example later after a discussion on byte ordering)

- Host byte ordering
  - Host byte ordering:
    - Host byte ordering = the byte ordering used by the host computer

---

  - Recall from CS255:
    - A host computer can use one of 2 different ways to store serie of bytes in the memory of a computer:
      - big endian and
      - little endian.

Example:



- Network byte ordering
  - Network byte order:
    - Network programming must *always* use the big endian order

---

  - Specifically:
    - These variables in the socket Internet address structure
      - `sin_port` (Socket's Internet Port number)
      - `sin_addr.s_addr` (Socket's Internet Address)

      must use the network byte ordering !!!

- Utility functions to convert from host ordering *to* network ordering

- **Converting** from **host ordering** to **network ordering**:

  - `int htonl(int)` : (host to network long)

    - **input** = **4 bytes** in **host ordering**
    - **output** = **4 bytes** in **network ordering**

    Example usage:

```
struct sockaddr_in    in_addr;

int  IP = 170*256*256*256 + 140*256*256 + 150*256 + 1;   // 170.140.150.1

in_addr.sin_addr.s_addr = htonl(IP);
```

  - `short htons(short)` : (host to network long)

    - **input** = **2 bytes** in **host ordering**
    - **output** = **2 bytes** in **network ordering**

    Example usage:

```
struct sockaddr_in    in_addr;

in_addr.sin_port = htons(8000);
```

- **Example:** setting up an **IPv4 socket address structure** with

  - **IP address** = **170.140.150.1**
  - **Port #** = **8000**

  Program code:

```
/* ================================================
   Define a socket address variable
   ================================================ */
struct sockaddr_in    in_addr;

/* ========================================================================
   Initialize the socket address
   ======================================================================== */

in_addr.sin_family      = AF_INET;      /* Protocol domain is: Internet */


int  IP = 170*256*256*256 + 140*256*256 + 150*256 + 1;
          // IP address is in host byte order !!!

in_addr.sin_addr.s_addr = htonl(IP);    /* IP address in network byte order */
in_addr.sin_port        = htons(8000);  /* Port # in network byte order */
```

- **Utility functions to convert from network ordering *to* host ordering**

  - **Fact:**

    - If we want to **print** the **IP address** or the **port #** stored in an **socket address variable**:

      - We must **first convert** the **network byte order representation** to **host byte order** before **printing** !!!

  - **Converting** from **network ordering** to **host ordering**:

    - `int ntohl(int)` : (network to host long)

      - **input** = **4 bytes** in **network ordering**
      - **output** = **4 bytes** in **host ordering**

      Example usage:

```
struct sockaddr_in    in_addr;

int  IP;

IP = ntohl(in_addr.sin_addr.s_addr);

/* ============================================================
   If you need to use (e.g., print) the value, you need to
   convert it to HOST order first
   ============================================================ */
```

```
printf("%d", IP);   // Prints the integer repr of the IP address

          // Computer program statements always use "host ordering" !!
```

- `short ntohs(short)` : (host to network long)

  - **input** = **2 bytes** in **network ordering**
  - **output** = **2 bytes** in **host ordering**

  Example usage:

```
struct sockaddr_in    in_addr;

short                 portNum;

portNum = ntohs(in_addr.sin_port);
```

Binding to a specific (kernel) UDP port#

- **Binding a UDP socket**
  - **Associating (= Binding)** a ***socket* variable** to a given ***network address***:

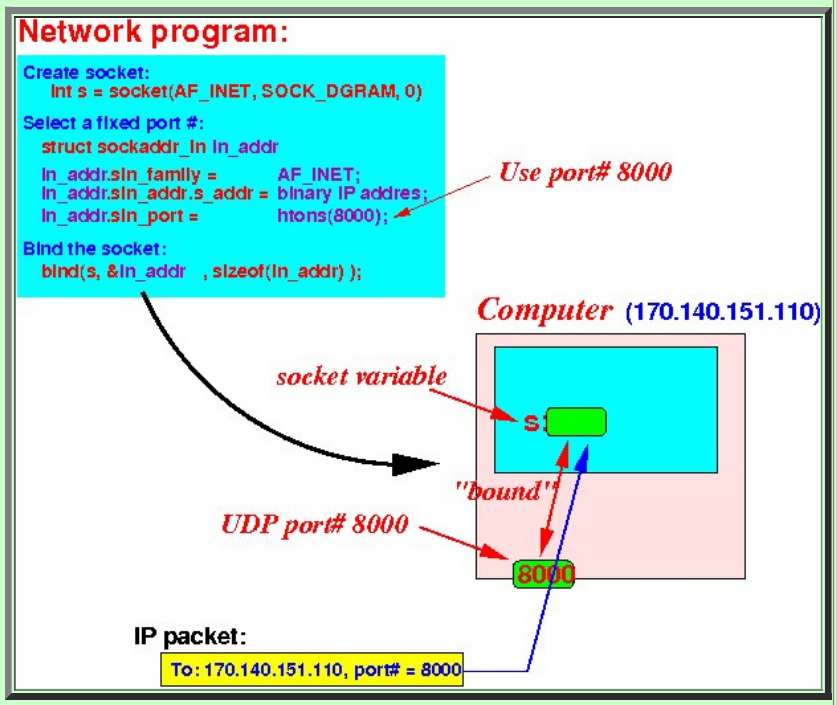
```
int bind(int s, (struct sockaddr *)addr, int length)

  s      = socket variable
  addr   = a socket IP address structure
  length = length of the socket IP address structure

Return value:

  0              = success
  Non-zero value = error code
```

Example:

```
int   IPAddress;              (a 32 bit IP address)
short PortNumber;             (a 16 bit port number)

int   s;
struct sockaddr_in in_addr;

s = socket(PF_INET, SOCK_DGRAM, 0);          // Create UDP socket

/* --------------------------------
   Set up Internet network address
   -------------------------------- */
in_addr.sin_family = AF_INET;                /* IP Protocol domain */
in_addr.sin_addr.s_addr = htonl(IPAddress); /* IP address */
in_addr.sin_port = htons(PortNumber);        /* UDP port # */

/* --------------------------------
   Bind socket to network address
   -------------------------------- */
if ( bind(s, (struct sockaddr *)&in_addr, sizeof(in_addr)) != 0 )
   perror("Bind error: ");
```

What does the **program** achieve:

- The following **diagram** shows you what the **program statements** has achieved:



Explanation:

- The **socket variable** `s` is now ***related*** to the **UDP port #8000** of the **computer**

- ***That means:*** **IP messages** that have **destination address**:

```
IP address = the IP address of this computer
port #     = 8000
```

will be ***delivered*** to the **socket** `s` !!!

- **Example Program:** (Demo above code) **Example**
  - Prog file: click here

Notes:

- Compile it with: **cc -o udp1 udp1.c**
- Run it with: **udp1 IP-address Port#**

Experiment:

- Try run udp1 using an IP-address that is NOT assigned to the host and see what happens
- Use the **"host" command** to find the **IP-address** of the host and run udp1 with a legal IP-address
- Also, try to bind to port# lower than 1000 (system reserved ports)

Example: (on **W301** computer):

```
/* ------------------------------------------------
   Find W301's IP address in dotted decimal notation
   ------------------------------------------------ */
cheung@W301(1007)> ifconfig
eth0      Link encap:Ethernet  HWaddr 3C:D9:2B:76:D7:09
          inet addr:170.140.151.25  Bcast:170.140.151.255  Mask:255.255.254.0
          inet6 addr: fe80::3ed9:2bff:fe76:d709/64 Scope:Link

/* ------------------------------------
   Convert IP address in binary
   ------------------------------------ */
cheung@W301(1008)> bc
170*256*256*256 + 140*256*256 + 151*256 + 25
2861340441


/* ------------------------------------
   Bind call with W301's IP address
   ------------------------------------ */
cheung@W301(1011)> udp1 2861340441 8000
**** Bind was successful....
Socket is bind to:
  addr = 2861340441
  port = 8000


/* ------------------------------------------------
   ERROR test: Bind call with not-your-own IP address
   ------------------------------------------------ */
cheung@W301(1010)> udp1 2861340442 8000
Error: bind to addr = 2861340442,, port = 8000 FAILED
Bind ?: Cannot assign requested address
```

- **Binding to a *used* port #**
  - **Warning:**
    - When a **port #** is **used (= bound)** by some ***other* network program**, then:
      - The `bind( )` call to the ***used* port #** will **fail** (and returns an **error code**)

- **Comment**
  - **Fact:**
    - There is **no need** to **bind** to a ***specific* port#** if:
      - just want to **transmit messages**

    - You ***only*** need to **bind** to a ***specific* port#** if:
      - **Some *other* process** needs to **contact** you

    I.e.:

      - When you are **writing** a **server** !!!

Binding to an available (kernel) UDP port

- **"System Selected" (= arbitrary selected) Port Numbers**
  - **Fact:**
    - ***Most* network applications** can use an ***arbitrary* (available) port #**

  - ***System selected* port number:**
    - ***System selected* port number** = an (***unused***) **port number** that is **picked (selected)** by the **Operating System**
      - A ***system selected* port number** is just an ***arbitrary* (unused) port #**

- **Binding to an arbitrary available (kernel) port**
  - **How to bind** to a ***System Selected* (kernel) port**
    - **Specify 0 (zero)** for the `sin_port` variable in the **network address**

    Example:

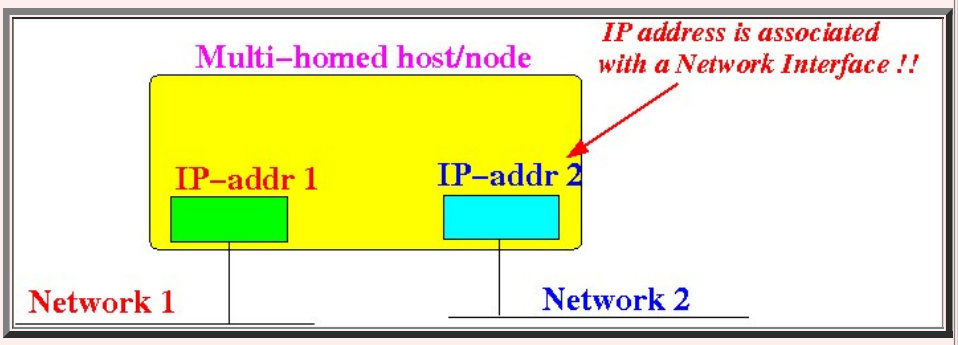
```
in_addr.sin_family = AF_INET;
in_addr.sin_addr.s_addr = htonl(INADDR_ANY);
in_addr.sin_port = htons( 0 );  /* Let system assign port # */
                          ^^^

if ( bind(s, (struct sockaddr *)&in_addr, sizeof(in_addr)) != 0 )
        perror("Bind: ");
```

  - **Example Program:** (Demo above code) ***Example***
    - Prog file: **/home/cs455001/demo/netw-prog-ipv4/udp3.c**

      Use port = 0 in the **arguments**

Binding a UDP socket in a multi-homed host

- **Multi-homed hosts**
  - **Recall** that:
    - A **multi-homed host** has ***multiple*** **IP addresses**



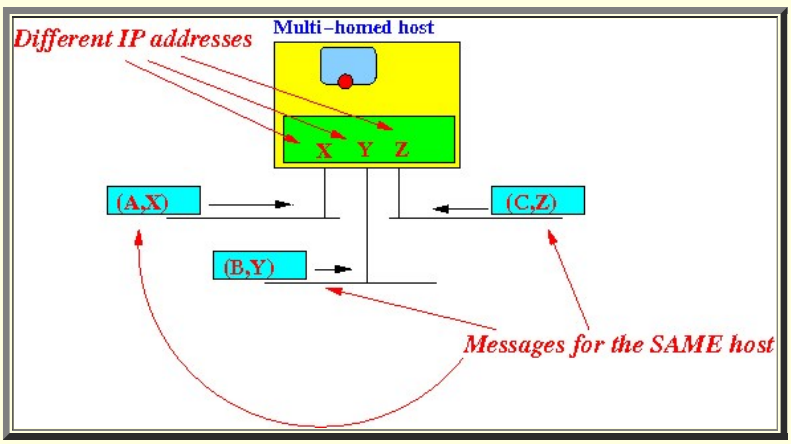
- **The association of a socket to a network address**
  - **Recall** the **structure** of a **network address**:

```
struct sockaddr_in
{
   sa_family_t     sin_family;   /* Address family
                                    - Must contain the value AF_INET */

   in_port_t       sin_port;     /* Internet Port #
                                    - a 2 bytes port number */

   struct  in_addr sin_addr;     /*  Internet Address (IP-address)
                                    - a 4 bytes IP address  */
};
```

Facts:

- An **network address** contains ***one*** **IP address** value
- A **socket** that is ***bound*** to an **IP address** ***X*** will:
  - ***Only*** receive **IP messages** with the **destination IP address** ***X***

  - **Situation:**
    - A **multi-homed node** can **receive messages** with ***different*** **IP addresses**:

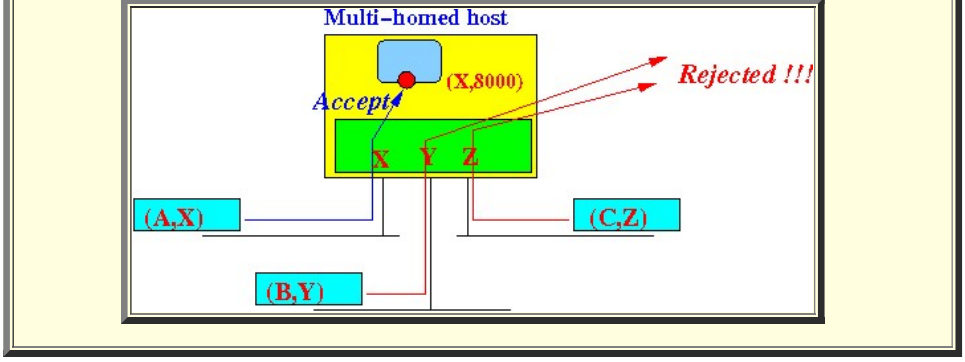


IP packets with **destination IP address**:

- **X** or **Y** or **Z**

will be **delivered** to the **multi-homed host** !!!

- A **socket** bound to a ***specific*** **IP address** will ***only*** **receive IP packets** destined for the ***specific*** **IP address**:



- **Wild Card IP address**
  - **Wild card IP address**

    - `INADDR_ANY` = the ***wild card*** **IP address** used in **network programming**

    A **wild card IP address** will ***match*** to ***any*** **IP address**

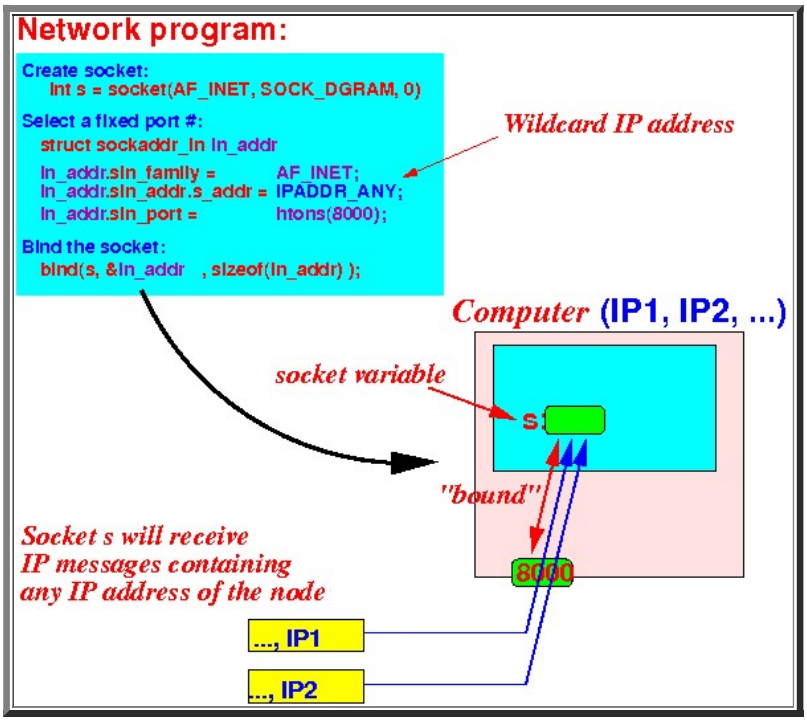
  - **Fact:**

    - A **socket** that is **bound** to the `INADDR_ANY` is **able** to:
      - **Receive IP packets** containing ***any*** **IP address** !!!!

    Example:

```
in_addr.sin_family = AF_INET;                /* Protocol domain */
in_addr.sin_addr.s_addr = htonl(INADDR_ANY); /* Use wild card IP address  */
in_addr.sin_port = htons(PortNumber);        /* UDP port # of socket */

/* --------------------------------
   Bind socket to socket address
   -------------------------------- */
if ( bind(s, (struct sockaddr *)&in_addr, sizeof(in_addr)) != 0 )
   perror("Bind error: ");
```

    Result:



- **Common practice of network programming**
  - **Common practice** in **network programming:**

    - **Network programmers** ***always*** use the `INADDR_ANY` value in an **network address**

      Example:

```
in_addr.sin_family = AF_INET;
in_addr.sin_addr.s_addr = htonl(INADDR_ANY);
                        /* Always use wild card IP address  */
in_addr.sin_port = htons(PortNumber);

/* --------------------------------
   Bind socket to socket address
   -------------------------------- */
if ( bind(s, (struct sockaddr *)&in_addr, sizeof(in_addr)) != 0 )
   perror("Bind error: ");
```

- It is a ***general*** **practice** to use `INADDR_ANY` ***even*** for ***single-homed*** **computers**...

  (Because it's **easier**)

- **Example Program:** (Bind to `INADDR_ANY`) ***Example***
  - Prog file: click here

Notes:

- Compile it with: **gcc -o udp2 udp2.c**
- Run it with: **udp2 Port#**

Example:

```
cheung@compute(1011)> udp2 8000
OK: bind to port = 9000 SUCCESS
**** Bind was successful....
Socket is bind to:
  addr = 0
  port = 8000
```

Client/server programming with UDP sockets

- **Server**
  - **Server:**

    - **Server** = a **program** that provides a **service**
      - A **server** will ***wait*** for **requests**

- **Client**
  - **Client**:

    - **Client** = a **computer program** that send **requests** to a **server (program)**

- **Writing a server program**
  - **Server programming:**

    - A **server application** will be ***contacted*** by a **client (program)** that **requires** the service

      ***Therefore*:**

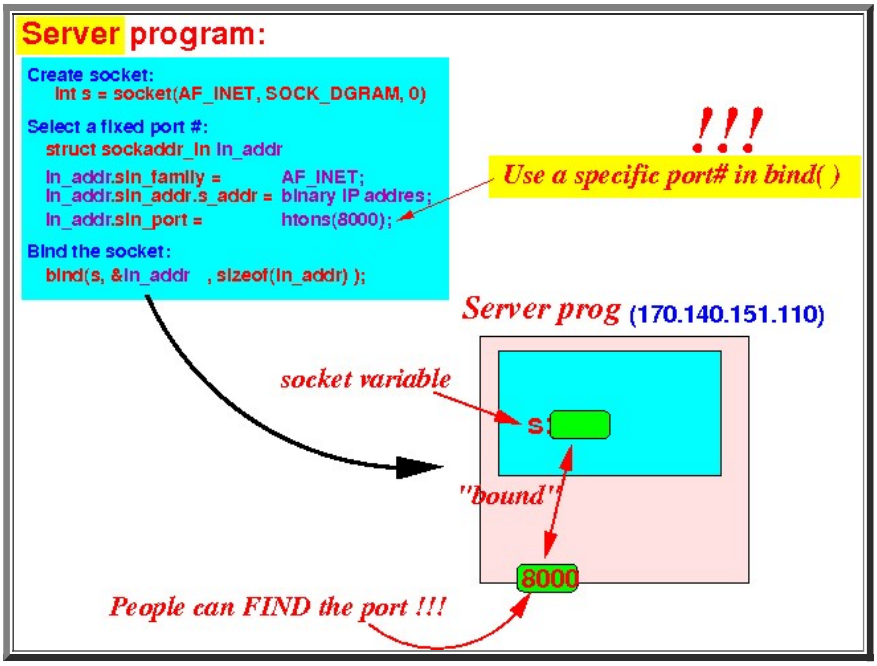
      - **Server application must** have a ***fixed*** **contact point** (= **port#**) !!!

      Analog:

      - A ***pizza*** **take out service** must has a ***fixed*** **telephone number** for **clients** to ***call*** to ***order*** **pizza** !!!

        (If the **phone number** can **change**, this **pizza service** will be ***impossible*** to ***find*** !!!)

  - **Summary**: when **writing** a **server**, **use** a ***fixed*** **(UDP) port#**



- **Writing a client program**
  - **Client programming:**

    - The **client** will ***always*** **initiate** the **communication with the server**

    - **Fact**:

      - **Messages sent** by **client** will contain:

- The **IP address** and the **port#** of the ***socket*** of the **client program** !!!

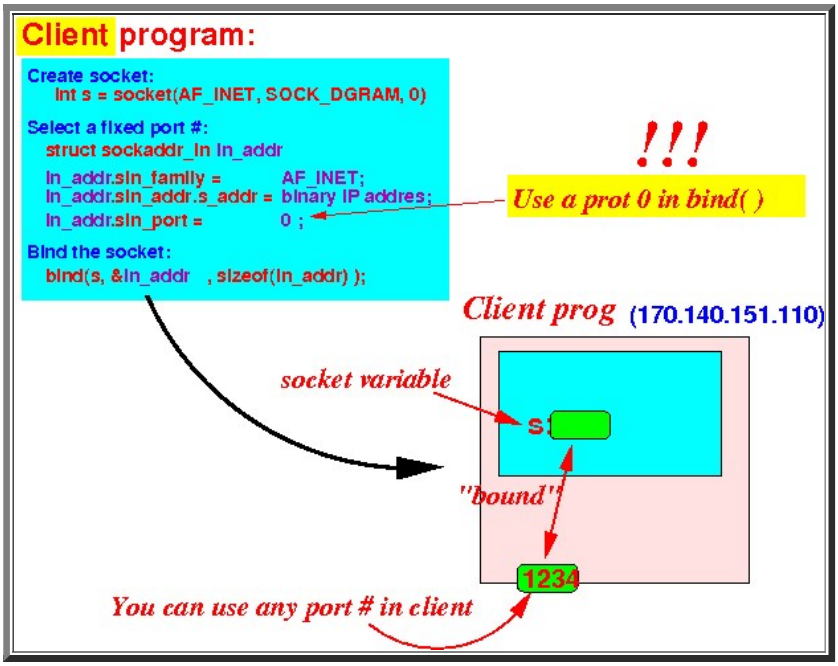
Analogy:

- We ***you*** calls to **order pizza**, the **pizzaria** will have your **phone number** (as in ***caller ID***)

- **Consequently:**
  - The **server** can **find out** the **network address** of the **client** !!!
  - A **client** can **use** a ***arbitrary*** **(available) port number** !!!

- **Conclusion:**
  - When **writing** a **client application**, **bind** the **socket** to a ***system assigned*** **port #**

- **Summary**



Sending and receiving with UDP sockets

- **Sending Packets using a UDP port**
  - **System call** to **send data** with **(bound) UDP socket:**

```
int sendto(int sock,
           void *data,   int len,              // Transmitted data
         i n t  flags,                         // Transmit options
           struct sockaddr *to,  int to_len);  // Destination network address
```

Meaining of the **parameters**:

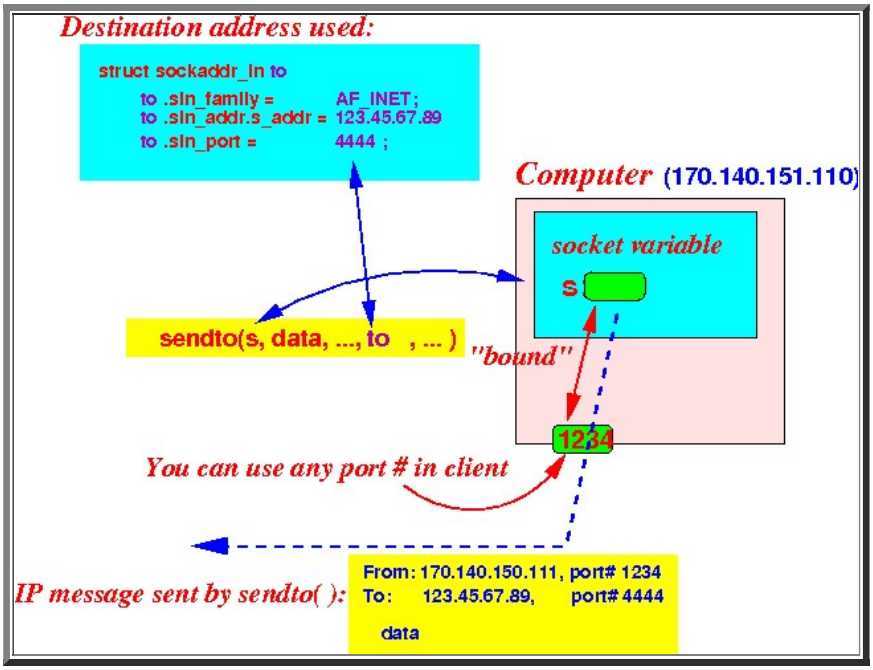
```
sock   = socket variable (should be bounded)

data   = pointer to data buffer location
len    = number bytes in data buffer to be sent

flag   = flags (indicate send options)
         E.g., Set MSG_OOB bit for  "out-of-band" (high priority) data

to     = Internet address (typed) variable
         contains: network address of the destination
to_len = length of the "to" data structure
```

  - **Effect:**

- **Transmits** a **UDP message** containing the **data** to the specify **destination** (**IP address** + **UDP port#**)

- The **UDP mesasge** will **contain** the **network address (IP addr** + **UDP port#)** of the **socket** `s`

Graphically:



  - ***Return*** **value:**

```
Error:     returns -1
Otherwise: returns len (≥ 0)
```

General **error processing**:

- If a ***system call*** (such as `sendto` returns an **error**), you can use:

```
perror( "Notice text" );    // print error
```

to **print** the **error message** to the **terminal**

  - **Example:** send the **string** `"Hello"`

```
char data[] = "Hello";   // String has 6 characters: H e l l o \0 !!!

struct sockaddr_in dest_addr;  // Internet address type
```

```
/* ----------------------------------------
   Set up destination: (IPaddress, Port#)
   ---------------------------------------- */
dest_addr.sin_family      = AF_INET;
dest_addr.sin_addr.s_addr = htonl(DestIPaddr);
dest_addr.sin_port        = htons(DestPosrtNumber);

if ( sendto(s, data, 6,
            0 /* no flags */,
           (struct sockaddr *)&dest_addr, sizeof(dest_addr))
     == -1 )
{
   perrer("sendto()");
}
```

- **Example Program:** (Demo above code) ***Example***
  - Prog file: click here

How to run the program:

- **Right click** on link(s) and **save** in a scratch directory
- To compile: `gcc -o udp4-s1 udp4-s1.c`
- To run: `udp4-s1 Dest-IP-addr Dest-Port#`

Example: `udp4-s1 2827975501 (=170.140.150.1) 9000`

Note:

- use **host NameOfComputer** and **bc** to **compute** the **destination IP address (in binary)**

- **Warning**: you won't see anything....
  - because the **receiver** is **not yet running** :-)

- **Remarks about the `UDP` send program**
  - **Important observation:**

- When you **run** the **above demo program**, you should **notice that**:
  - The **program** can send a **UDP message** even when there is ***no*** **receiver** !!!

This can be done **because**:

- **UDP messages** are ***unreliable***
- **I.e.:** `sendto( )` does ***not*** require any **acknowledgement** from a **receiver** !!!

Analogy:

- `sendto( )` is like **mailing a letter**

- The **sender** does **not** know the **result** of the **transmission**

- **Sending with an unbounded UDP socket**
  - **Hacker's note:**

- It is **possible** to send **UDP messages** using an ***unbounded*** **UDP socket**:

```
int   IPAddress;             (a 32 bit IP address)
short PortNumber;            (a 16 bit port number)

int   s;
struct sockaddr_in in_addr;

s = socket(AF_INET, SOCK_DGRAM, 0);         // Create socket

// s is unbounded


/* --------------------------------
   OMIT the bind step
   -------------------------------- */
in_addr.sin_family = AF_INET;               /* Protocol domain */
in_addr.sin_addr.s_addr = htonl(IPAddress); /* IP address of socket */
```

```
~~in_addr.sin_port = htons(PortNumber);         /* UDP port # of socket */~~

~~/* ---------------------------------
     Bind socket to socket address
     --------------------------------- */~~
~~if ( bind(s, (struct sockaddr *)&in_addr, sizeof(in_addr)) != 0 )~~
~~    perror("Bind error: ");~~

  /* ---------------------------------------
     Set up destination: (IPaddress, Port#)
     --------------------------------------- */
  dest_addr.sin_family      = AF_INET;
  dest_addr.sin_addr.s_addr = htonl(DestIPaddr);
  dest_addr.sin_port        = htons(DestPosrtNumber);

  if ( sendto(s, data, 6,
              0 /* no flags */,
             (struct sockaddr *)&dest_addr, sizeof(dest_addr))
       == -1 )
  {
     perrer("sendto()");
  }
```

(An **unbounded socket** is a **socket variable s** where you has ***not*** called `bind(s, ....)`

- **Effect** of **sending** with an **unbounded socket**:

  - **UDP Messages *sent*** with an ***unbounded*** **UDP socket** is:

    - **"undefined"**

      (= **depends** on the **UDP implementation**)

  - **Possible results**:

    - Some **system** will **bind** the **socket** (to an **available UDP port**) first

    - **Other systems** will send **UDP messages** with:

      - **Empty** (= garbage) **sender's IP address and port number**

      (**I.e.:** ***anonymous*** IP messages !!!)

Receiving with UDP sockets

- **Receiving Packets using UDP port**
  - **System call** to **receive data** with **UDP socket**:

```
int recevfrom(int sock,
              void *buf, int len,                     // Receive buffer
          i  n  t   flags,
              struct sockaddr *from, int *from_len);  // Sender's network addr.
```

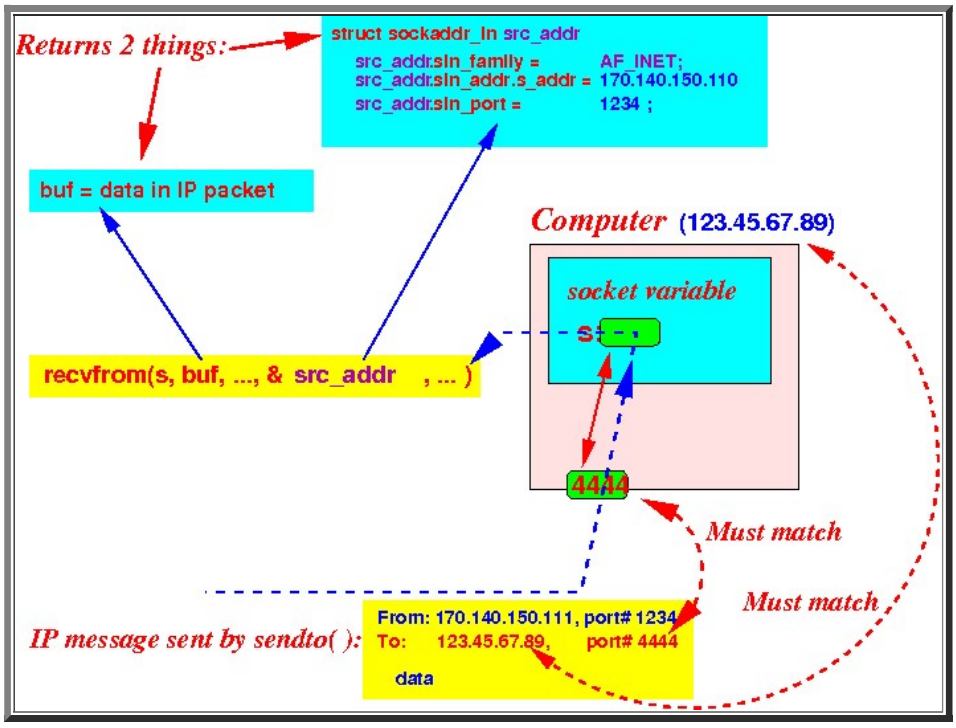
```
Parameter:

     sock     = socket variable

     buf      = pointer to user data buffer location (stores recv'd message)
     len      = buffer size of "buf"

     flag       = various flags (options)
                  E.g., Set MSG_OOB bit for  "out-of-band" (high priority) data

     from     = IP address + Port number of the sender !!!
     from_len = length of the "from" structure
```

  - **Effect:**
    - *If* there is *no* **UDP message** in the **receive buffer**:
      - The `recvfrom( )` call will **block**.....

    - *Otherwise*, `recvfrom( )` will:
      - **retrieves** the **next UDP message** from **(kernel's) receive buffer associated** with the **socket** `sock`

    Specifically:
      - The **entire UDP message** will be **stored** into the `buf` variable
      - The **length** of the **UDP message** will be **returned** in the `len` variable
      - The **sender's network address** is **returned** in the `from` variable

Graphically:



  - **Very important fact:**
    - The **network address (= IP addr + port#)** of the **sender** is **return** in:
      - The **variable** `from`

Therefore:

- The **receiver** can **send back** a **reply** using:

```
sendto( s, reply, len, 0, from, from_len);
```

- ***Return* value:** (of `recvfrom( )`)

```
Error: returns -1

Otherwise: returns the number bytes stored in buf
```

Example:

```
/* ======================================
   Assume we have done these already:

        create socket  s
 bound   s   to  a  port
   ====================================== */

char line[1000];                        /* User buffer to receive data */


struct sockaddr_in   src_addr;          /* Used to store sender's address */
int  length;                            /* Length of the sockaddr_in data type */


/* ------------------------------------------------
   Set up the length to tell recvfrom how much space
   it can write into the "src_addr" variable
   ------------------------------------------------ */
length = sizeof(src_addr);             /* Length of the src_addr variable */

/* ------------------------------------------------------------
   Receive the next message into the variable "msg"
   The sender of the message will be recorded in "src_addr"
   (You can use "src_addr" to reply to the sender !)
   ------------------------------------------------------------ */
recvfrom( s, line, 1000, 0 /* flags */,
         (struct sockaddr *)&src_addr, &length);
```

- **Example Program:** (Demo above code) ***Example***
  - Prog file: click here

How to run the program:

- **Right click** on link(s) and **save** in a scratch directory
- To compile: `cc -o udp4-r1 udp4-r1.c`
- To run: `udp4-r1 My-IP-address my-Port#`
  - use **My-IP-address = 0** for **wildcard IP address**
  - You need to use the **same port#** in the **send program above** to send message to this program

---

- **Sending a *reply* back to the sender**
  - You can **send** a **reply** back to the **sender** using the `src_addr` variable from the `recv_from( )` call:

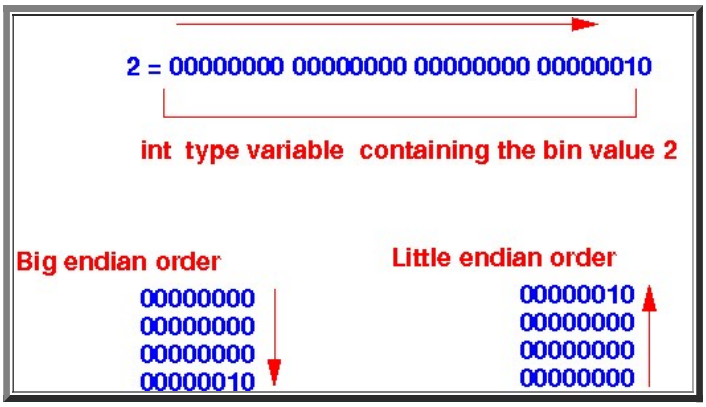
```
/* ------------------------------------------------------------
   Receive the next message into the variable "msg"
   The sender of the message will be recorded in "src_addr"
   (You can use "src_addr" to reply to the sender !)
   ------------------------------------------------------------ */
recvfrom( s, line, 1000, 0 /* flags */,
         (struct sockaddr *)&src_addr, &length);

....

/* ------------------------------------------------
   Program can send reply to sender using "src_addr"
   ------------------------------------------------ */
sendto( s ,  buf,  length_of_buf,   flags,
        (struct sockaddr *)&src_addr, &length);
```

Sending and receiving binary numbers

- **Storing integer (or any other type of) binary numbers**
  - **Computers store binary numbers** in **one** of **2 possible order**:

    - **Big endian** ordering
    - **Little endian** ordering

    Graphically:

  - ***Host* byte ordering:**

    - **Host byte ordering** = the **storage** ***ordering*** of **binary numbers** used in a **computer** (= ***host***)

- **Transmiting integer (or any other type of) binary numbers**
  - **Fact:**

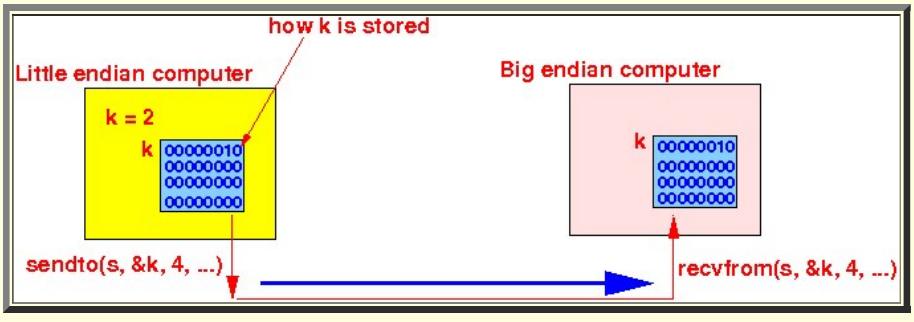
    - The **function call**:

      ```
      sendto( s, data, len, flag, toAddr, toAddrLen )
      ```

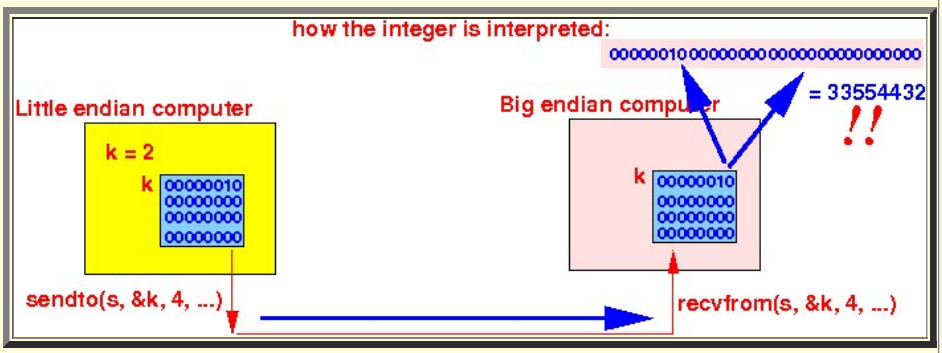
      will **transmit** the `len` **bytes** (stored) in the variable `data` in the **order** stored in **(host) memory**

- **What happens if we transmit binary data between a big endian computer and a little endian computer**
  - **Consider** the following **data transmission** of an **integer binary number** from **little endian computer** to a **big endian computer**:

    - A **little endian computer** send the **integer value 2** to a **big endian computer**

    - The **big endian computer** will ***interpret*** the **integer binary number** as **follows**:

- **Example Program:** (Demo above code) **Example**
  - Prog file:
    - /home/cs455001/demo/netw-prog-ipv4/udp-send-int.c
    - /home/cs455001/demo/netw-prog-ipv4/udp-recv-int.c

How to run the program:

- To compile:
  - `gcc -o udp-send-int udp-send-int.c` on **W301** (or a lab machine)
  - `gcc -o udp-recv-int udp-recv-int.c -lnsl -lsocket` on **compute**
- To run:
  - On **compute**: `udp-recv-int 8000`
  - On **W301**: `udp-send-int compute 8000`

- ***How to* transmit binary data**
  - **Technique** to **transmit binary data**:
    - **Sender:**
      1. **First**, **convert** the **binary data** to ***network* byte ordering**
      2. **Then**, send the **converted data**

      Example:

```
int k = 1;     // k is stored in host byte order

printf("Sending: int value = %d\n", k);

int  x = htonl(k);      // Convert to network byte ordering

sendto(s,  &x, sizeof(int), 0 /* flags */,
          (struct sockaddr *)&dest_addr, sizeof(dest_addr));
```

    - **Receiver:**
      - **Receiver** the **binary data** (we know it is in ***network* byte order**
      - **Convert** the **data** to ***host* byte order** using:

```
int   ntohl( int ):  convert 4 byte data from network byte ordering
                     to host byte ordering

short ntohs( short): convert 2 byte data from network byte ordering
                     to host byte ordering
```

      Example:

```
int x;

src_addr_len = sizeof(src_addr);
len = recvfrom(s, &x, sizeof(int), 0 /* flags */,
               (struct sockaddr *) &src_addr, &src_addr_len);

k = ntohl( x );       // Convert int from network byte ordering
                      // to host byte ordering

printf("k = %d", k);
```

- **Example Program:** (Demo above code) **Example**
  - Prog file:

- Edit the same demo programs:
  - /home/cs455001/demo/netw-prog-ipv4/udp-send-int.c
  - /home/cs455001/demo/netw-prog-ipv4/udp-recv-int.c

  Uncomment the `htonl( )` and `ntohl( )` calls in the code

How to run the program:

- To compile:
  - `gcc -o udp-send-int udp-send-int.c` on W301 (or a lab machine)
  - `gcc -o udp-recv-int udp-recv-int.c -lnsl -lsocket` on compute
- To run:
  - On compute: `udp-recv-int 8000`
  - On W301: `udp-send-int compute 8000`

- **Transmitting structured data**
  - **Fact:**
    - Structured data can contain:
      - String/text data (stored in the *same* order in both big endian computer and little endian computer)
      - integer/float binary data (stored in *different* order in big endian computer and little endian computer)

  - **How to transmit *structured* data**:
    - Transmitting the *structured* data:
      1. Use `htonl( )` (or `htons( )`) to convert every integer/float variable to network byte order

      ---

      2. Send the converted `struct` using `sendto( )`

  - **How to receive *structured* data:**
    - Transmitting the *structured* data:
      1. Use `recvfrom( )` to receive the transmitted `struct` data

      ---

      2. Use `ntohl( )` (or `ntohs( )`) to convert every integer/float variable to host byte order

    ---

  - **Example Program:** (Demo above code) ***Example***
    - Prog file:
      - /home/cs455001/demo/netw-prog-ipv4/udp-send-struct.c
      - /home/cs455001/demo/netw-prog-ipv4/udp-recv-struct.c

    How to run the program:

    - To compile:
      - `gcc -o udp-send-struct udp-send-struct.c` on W301 (or a lab machine)
      - `gcc -o udp-recv-struct udp-recv-struct.c -lnsl -lsocket` on compute
    - To run:
      - On compute: `udp-recv-struct 8000`
      - On W301: `udp-send-struct compute 8000`

Intro to "Support Functions" for Network Programming

- **Network Support Functions**
  - **Fact:**
    - **Computer** on the **Internet** are **identified** by their **IP addresses**
      - When we **wish to *visit* google.com**, we **need** to use its **IP address** !!!

      (**Analogy:** dial the **telephone number**)
    - **IP-addresses** are ***binary* numbers**
      - It is **tedious** to **remember** a **binary number** !!!
  - **Functions** to support **network programming**:
    - **Lookup function** to **find** the corresponding **IP address** of a ***symbolic* host name**

      Example:
      - You are given the **symbolic host name**, e.g.: **www.google.com**

        Find the **IP address** for **www.google.com**

      Analogy: look up in a **telephone directory**
    - ***Reverse* look up** function:
      - You are given an **IP address**

        Find the **symbolic host name** for that **IP address**
    - **Dotted decimal address** to **binary IP address** translation:
      - You are given a **dotted decimal IP address**, e.g.: `"170.140.150.1"`

        Find the corresponding **binary IP address**

- **The network service *library***
  - The **network service library** for **C**:
    - **network service library (nsl)** = a **function library** in **C programming language** that contains the **IP address conversion** functions
  - **Compiling** programs that uses the **network service library functions**:
    - When your **network program** uses some **network service function(s)**:
      - You need to ***link*** the **network service library (nsl)** when you **compile** your **program**
  - **How to link** the **network service library**:

```
cc  -o  my-netw-prog    my-netw-prog.c   -lnsl
```

Converting "dotted-decimal-format" to (binary) IP address

- **"Dotted Decimal Notations" ===> *binary* IP addresses conversion**
  - **Converting** a **"Dotted Decimal Notation" string** (such as **"170.140.151.110"**) to a **binary IP address** :

    - `int inet_addr("xxx.xxx.xxx.xxx")`
      - **Input:**
        - A ***string*** with a **dotted decimal IP address**

          E.g.: `inet_addr("170.140.151.110")`
      - **Output:**
        - The corresponding **binary IP address** in ***network*** **byte order**
        - Returns **−1** if the **input string** does **not contain** a **dotted decimal IP address**

      Example:

      ```
      struct  sockaddr_in     dest_addr;

      dest_addr.sin_addr.s_addr = inet_addr("170.140.151.110");
      ```

  - **Important note:**

    - **Do *not*** use `htonl( )` on the **output** of `inet_addr( )`

      Because:
      - the **output** of `inet_addr( )` is ***already*** in ***network*** **byte ordering** !!!

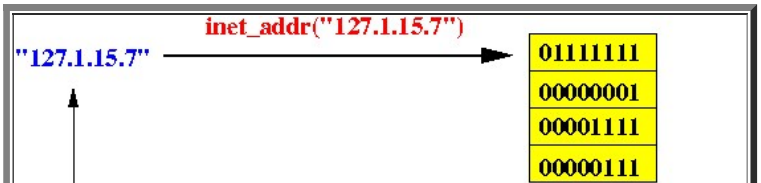
- **(Binary) IP address ====> "dotted-decimal" conversion**
  - **Converting** a **binary IP address *back*** to a **"Dotted Decimal Notation" string** (such as **"170.140.151.110"**):

    - `char * inet_ntoa(int)`
      - **Input:**
        - A ***binary*** **IP address** in ***network*** **byte ordering**
      - **Output:**
        - A **string** containing ***dotted decimal*** **representation** of the **input IP address**

    Example:

    ```
    Suppose:

        struct  sockaddr_in  dest_addr;     contains an IP address in network byte order


    char IPaddr[100];

    strcpy(IPaddr, inet_ntoa(dest_addr.sin_addr.s_addr) );
    ```

- **Summary:** `inet_addr` **and** `inet_ntoa`
  - The **following figure summarizes** the **operation** of the `inet_addr()` and `inet_ntoa()` functions:



Output is in NETWORK byte order !!!

inet_ntoa(int)

- **Network programming using *dotted decimal* IP address**
  - The **UDP sender** using ***dotted decimal* IP address** to specify the **receiver**:

```
int main(int argc, char **argv)
{
   struct sockaddr_in dest_addr;        /* Destination socket address */

   ...

    /* ================================================================
       argv[1] is the string containg the dotted decimal IP address
    ================================================================ */
    if ( ( dest_addr.sin_addr.s_addr = inet_addr( argv[1] ) ) == -1 )
    {
       printf("Dotted IP-address must be of the form a.b.c.d\n");
     exit(1);
    }

    ...

}
```

  - **Example Program:** (Demo above code) ***Example***

    - **Sender:** `/home/cs455001/demo/netw-prog-ipv4/udp4-s2.c`
    - **Receiver:** `/home/cs455001/demo/netw-prog-ipv4/udp4-r2.c`

    - The **new** UDP sender using **dotted decimal IP address** Prog file: click here
    - The **(unmodified)** UDP receiver Prog file: click here

    How to run the program:

    - To compile the **sender**: `gcc -o udp4-s2 udp4-s2.c -lnsl`
    - To compile the **receiver**: `gcc -o udp4-r2 udp4-r2.c`

    - To run the **receiver**: `udp4-r2 port#`
    - To run the **sender**: `udp4-s2 xx.xx.xx.xx port#`

Converting "symbolic names" to (binary) IP address

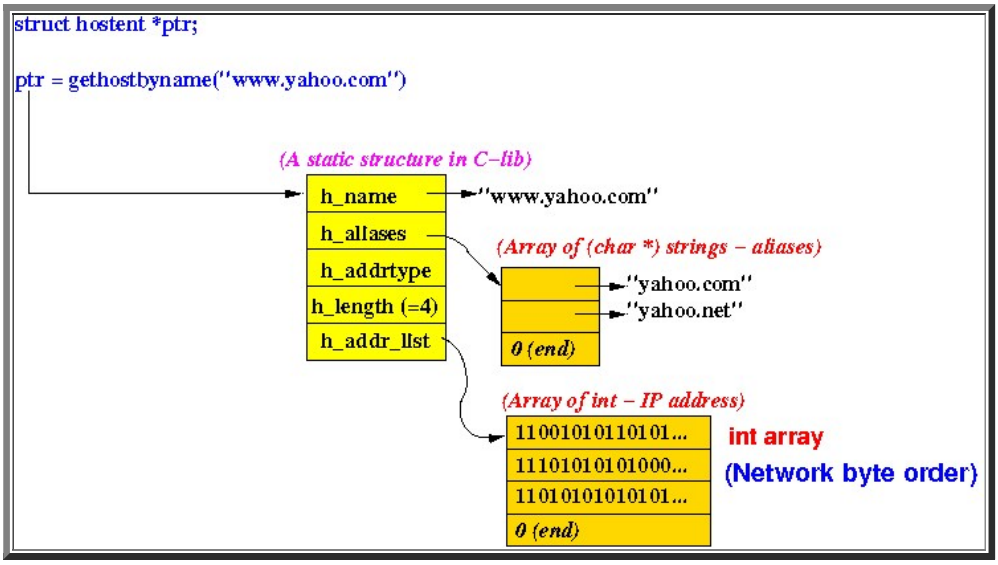
- **gethostbyname(): Find *host information* using a *Symbolic* Host name**
  - **Converting** a ***symbolic* host name** (such as **"www.facebook.com"**) to a **binary IP address**:

    - `struct hostent*  gethostbyname("hostname")`
      - **Input:** a ***string*** with a ***symbolic* host name** (e.g.: **"www.facebook.com"**)
      - **Output:** a **pointer** to a ***host entry*** typed **data structure**
        - Returns `NULL` if error (e.g., **hostname** is **not found**).

- **The "host entry" data structure**
  - The **host entry** `struct hostent` data structure:

```
struct hostent
{
   char *  h_name;       /* The canonical (official) name of host */
                         // a string variable

   char *  h_aliases[ ];  /* The list of "aliases" for the hostname */
                   /  /   h_aliases is an array of string
                           // h_aliases[i] == NULL marks the end of the array

   int     h_addrtype;     /* host address type (should be = AF_INET (2)) */
   int     h_length;       /* length of address (= 4 bytes IP addresses) */

   char *  h_addr_list[ ]; /* list of addresses */
                           // h_addr_list is an array of int (binary IP addresses)
                 / /    i  n   network byte order
                         / /  h_addr_list[i] == 0 marks the end of the array
};
```

  - **Graphical representation** of the **"host entry"** data structure (with ***sample* content** to make things clearer):



- **How to use/access the data in a host entry (too much info)**

  Example: **how to** use the `gethostbyname()` call

```
/* ------------------------------
   Variables
   ------------------------------ */
char hname[100];
 struct hostent *hp;  /* host pointer */

int  addr;
int  i;

printf("Enter hostname: ");
scanf("%s", hname);

hp = gethostbyname(hname);

if (hp == NULL)
```

```
{
   printf("host information for %s not found\n", hname);
   exit(1);
}

/* ------------------------------------
   Print the information
   ------------------------------------ */

/* --------------
   Canonical name
   -------------- */
printf("official hostname h_name = `%s'\n", hp->h_name);

/* ----------------------------
   List of aliases
   ---------------------------- */
printf("Host has this alias list:\n");

char *p[];           /* help variable to access h_aliases[ ] */

p = hp->h_aliases;   // p[i] is a string (char *)

for ( i = 0 ; p[i] != NULL; i++ )
{
    printf("\tAlias %d: %s\n", i+1, p[i] );
}


printf("Host's address type = %d (should be AF_INET = %d)\n",
             hp->h_addrtype, AF_INET);

/* -------------------------------------
   Length of an IP address
   ------------------------------------- */
printf("Host's address length = %d\n", hp->h_length);

/* -------------------------------------
   List of IP addresses
   ------------------------------------- */
printf("List of Internet addresses for host:\n");

int  *a[];                    /* Help variable to access h_addr_list[ ] */

a = (int **) hp->h_addr_list;  // a[i] is an IP address

for (i = 0; a[i] != 0; i++)
{
   printf("\tIP address %d: ", i+1);
   paddr( (void*) a[i] );     // Print IP address as dotted decimal
   printf("\n");
}
```

- **Example *output*:**

```
Enter hostname: www.yahoo.com


official hostname h_name = `ds-any-fp3-real.wa1.b.yahoo.com'

Host has this alias list:
      Alias 1: www.yahoo.com
     Alias 2: fd-fp3.wg1.b.yahoo.com
     Alias 3: ds-fp3.wg1.b.yahoo.com
     Alias 4: ds-any-fp3-lfb.wa1.b.yahoo.com

Host's address type = 2 (should be AF_INET = 2)

Host's address length = 4

List of Internet addresses for host:
     IP address 1: 98.138.253.109
     IP address 2: 206.190.36.45
     IP address 3: 98.138.252.30
     IP address 4: 206.190.36.105
```

- **Example Program:** (Demo above code) ***Example***
  - Prog file: click here

Notes:

- Compile with: `cc -o gethostbyname gethostbyname.c -lnsl`

- **Network programming using *symbolic* host names**
  - **Fact:**
    - We just **need** a (**any**) **IP address** of a **IP host** !!!

- **Variable** in the **host entry** structure that contain the **IP addresses** of the **host**:

```
h_addr_list[ ]      // It's an array (multiple IP addresses)
```

We just need **1**, so the **safest choice** is:

```
h_addr_list[0]   // The other entries may be invalid !!!
```

- The **UDP sender** using a ***symbolic*** **host name**:

```
int main(int argc, char **argv)
{
   struct sockaddr_in dest_addr;           /* Destination socket address */
   struct hostent *hp;

   ...

   /* ============================================================
      argv[1] is the string containing the symbolic host name
   ============================================================ */

   hp = gethostbyname(argv[1]);        // E.g. argv[1] = "wwww.google.com"

   if (  h == NULL )
   {
      printf("Error... \n");
      exit(1);
   }

   /* ============================================================
      Use:
    hp->haddr_list[0]      // This one is for sure valid :)

      Note: hp->haddr_list[0] is in network byte ordering !!!
      ============================================================ */

   memcpy( (char *) &(dest_addr.sin_addr.s_addr),
           hp->h_addr_list[0],        // Copy first entry and
           hp->h_length);             // perserve byte ordering

    ...

}
```

Note:

- `memcpy(dest, src, len)` copies `len` bytes from `src` address to `dest` address while ***preserving*** **the byte ordering**
  - **Do *not*** use `htonl( )`

  because the IP address in `h_addr_list[0]` is ***already*** **in network byte ordering** !!!)

- **Example Program:** (Demo above code) ***Example***

  - The **new** UDP sender using ***symbolic*** **host names** Prog file: click here
  - The **(unmodified)** UDP receiver Prog file: click here

How to run the program:

- **Right click** on link(s) and **save** in a scratch directory

- To compile the **sender**: `gcc -o udp4-s3 udp4-s3.c -lnsl`
- To compile the **receiver**: `gcc -o udp4-r2 udp4-r2.c`

- To run the **receiver**: `udp4-r2 port#`
- To run the **sender**: `udp4-s3 lab??.mathcs.emory.edu port#`

Find host information using an IP address

- **gethostbyaddr(): Find *host information* from an IP address**
  - This **functionality** is ***rarely*** **used**:

    - Find **host information** (e.g., the ***symbolic*** **host name**) using a ***binary*** **IP address**

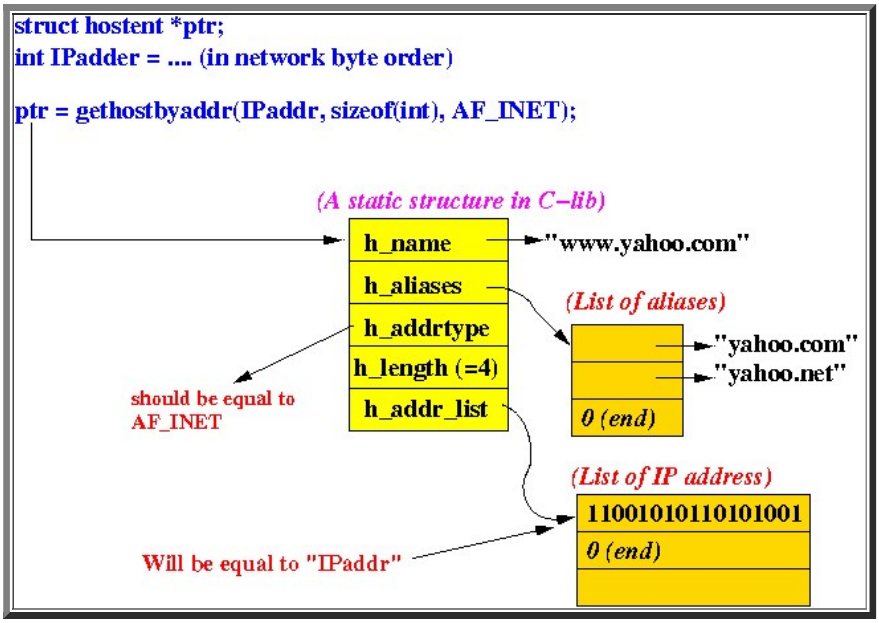
    In case you **do** need it, the **function** is:

    ```
    gethostbyaddr( ... )
    ```

  - Find **host information** using its ***binary*** **IP address:**

    - `struct hostent*  gethostbyaddr( char *addr, int len, int type)`
      - **Input:**
        - **addr** = **starting address** of the **4 byte** IP address in **network byte order**
        - **len** = always set to **4** for **IPv4 address**
        - **type** = always set to **AF_INET** for **IPv4 address**
      - **Output:**
        - a **pointer** to a ***host entry*** **data structure**
        - Returns **NULL** if **error**.

  - `gethostbyaddr( )` will ***also*** return a **"host entry" data structure** (just like `gethostbyname( )`):

    (It is the ***same*** **data structure** returned by **gethostbyname()**

    Therefore, I will **skip** on how to use this data structure....

    You can read the **program** below yourself....)

  - **Example Program:** (Demo above code) *Example*
    - Prog file: click here

    Notes:

    - Compile with: **cc -o gethostbyaddr gethostbyaddr.c -lnsl**

Programming with Timeout

- **Recall: the Stop-and-wait protocol**
  - The **stop-and-wait** protocol (click here click here) **transmits** a **message** and must **wait** for **acknowledgement**:

```
stop-and-wait-send( input-data )
{
   seqno ++;     // use next sequence number

   msg.seqno = seqno;       // label message with the seqno
   msg.data  = input-data   // put inpute data inside msg

   send msg;                // send message to receive (use sendto( ) !)

   ack_recv = false;

   while ( ! ack_recv )
   {
       wait for acknowledgement;

   i f   (  timeout while waiting for ACK )
       {
    send msg (again);      // retransmit message
       }
    e l s e
       {
     if  ( ACK.seqno = seqno )
         ack_recv = true;
       }
   }
}
```

- **Functions that support "Timeout programming"**
  - **Time out:**
    - **Network programs** often use **timeout** to **recover** from **transmission errors**
  - **System calls (= functions)** that can be used in **timeout programming**:
    - `select( )` (***originally*** defined in the **BSD (Berkeley Software Distribution)** UNIX)
    - `poll( )` (***originally*** from **UNIX System V**)

    Most UNIX systems nowadays provide **both** system calls.

Intro to the `select( )` function

- **Header files for `select`**
  - In order to use the `select( )` function in **C**, you must include this **header file**:

```
#include <sys/select.h>
```

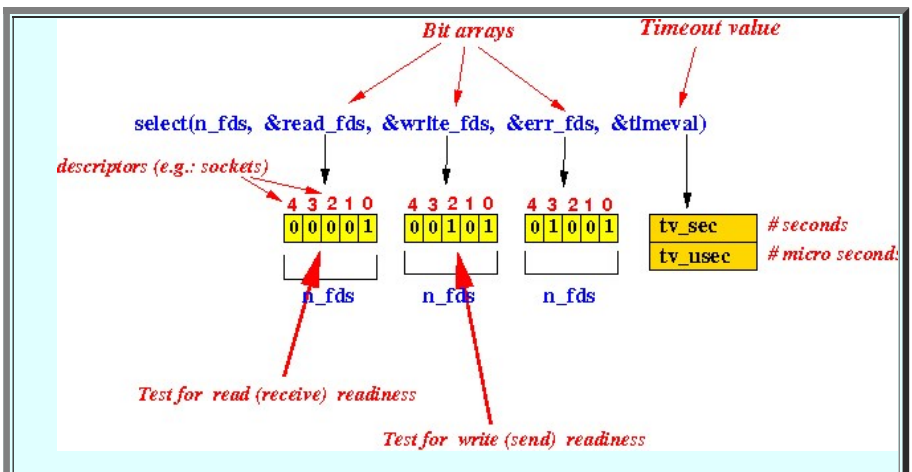
- **Usage of the `select( )` function**
  - **Usage** of the `select( )` function:
    - `select( )` allow a **program** to **check/test** and/or **wait until**:
      - one or more **file descriptors/sockets** is **"ready"** for **reading (receiving)** or **writing (sending)**
  - **"Ready"** for **reading/writing**:
    - **Recall** that:
      - A **read/recvfrom** operation will **block** if:
        - There is ***no*** **data** in the **receive buffer**
    - A **socket** is ***ready*** for some **operation** (**read/write**) if:
      - The **operation** (**read/write**) will ***not*** **block**

- **The `select` system call**
  - **Syntax:**

```
#include <sys/select.h>
#include <sys/time.h>               /* Header files */

int select( int             nfds,
           fd_set *         readfds,    // a bit array
           fd_set *         writefds,   // a bit array
           fd_set *         errorfds,   // a bit array
           struct timeval * timeout   );
```

```
The data types used:

fd_set = a bit array type (an int array interpreted as bit array)

struct timeval
{
   unsigned  tv_sec;    /* # seconds in the timeout value */
unsigned  tv_usec;   /* # microseconds  */
}
```

Meaning of the **parameters**:



```
nfds      = number of bits used in readfds, writefds and errorfds

readfds   = descriptor bit array to test for read readiness
            (NULL means empty set)

            The  i^th bit in the bit array corresponds to the file descriptor i

writefds  = descriptor bit array to test for write readiness
            (NULL means empty set)

errorfds  = descriptor bit array to test for error conditions
            (NULL means empty set)

timeout   = maximum wait duration   (time out value)
            (NULL means wait indefinitely)
```

- **Effect** of the `select( )` function:

  - The `select( )` **function** examines the **file (or *sockets*) descriptor sets** in the **variables**:

    - **readfds** for **read readiness** (= has data)
    - **writefds** for **write readiness** (= has buffer space)
    - **errorfds** for **error**

    for a **maximum duration** given by the `timeout` **variable**

  - **If *some*** of these **fds/sockets** are **ready *before*** the `timeout` expires:

    - The `select( )` **call** will **return *immediately***

    Return values:

    1. The **function's return value** is **equal to**:

       - the **number of file descriptors/sockets** that are **ready**

       ---

    2. The **bit array** variables **readfds**, **writefds** and **errorfds** will contains:

       - the **file descriptors** that are **ready** for the **corresponding operation**

         (I.e., the **bit position** of the **file descriptors** in the **bit array** = **1**)

       ---

    3. The **timeout** variable will be:

       - **reduced** by the **amount of time** spent inside the `select( )` **call**

       (I.e.: the **timeout** variable will **contain** the ***remaining*** **waiting time** before the **timeout expires**)

  - **Otherwise** (**nothing** is **ready** after **waiting** for `timeout` amount of time):

    - `select( )` will return (the **function value**) **0**

Simplified use of the `select( )` function

- A *simplified* usage of `select( )`
  - **In general:**

    - The `fd_set` **variable** used in `select( )` is:
      - a **bit array** of **1024 bits**

    Manipulating (set/reset) the **bits** in such a **large bit array** requires the use of **macros**

  - **However:**

    - you ***can*** use **smaller** of **bit array** variables in `select( )` !!!!

    Example:

    - An `int` variable is a **bit array** of **32 bits**

    I will ***first*** show you **how** to use `select( )` with an `int` **(32 bit)** bit array.

- **Testing read-readiness of a socket**
  - **Recall:**

    - An `int` is an **array** of **32 bits**

  - We can **clear all bits** in an `int` using:

```
int s = 0;  // Clear all bits
```

  - We can **set** the **bit position** `i` using:

```
s = s | (1 << i);
```

  - **How to** test the **read (receive) readiness** of a **socket** using `select( )`:

```
int   fdset;          // bit array of 32 bits

int   s;              // socket
                      // Assume s is initialized and bound to some port...

/* ==========================================
   Set up timeout value
   ========================================== */
struct timeval  timeout;

timeout.tv_sec = 0;
timeout.tv_usec = 50;  // 50 micro sec time out

/* ==============================================
   Prepare file descriptors for select( )
   ============================================== */
fdset = 0;                    // Clear bit array
fdset = fdset | (1 << s);     // s = an initialized socket variable


/* ================================================
   Check for message arrival with in timeout
   ================================================ */
if ( select( 32, &fdset, NULL, NULL, &timeout) != 0 )
{
    // s is read for reading
}
else
{
    // timeout occured
}
```

- **Example program on using** `select( )`
  - **Example program** to **demonstrate** the **effect** of `select( )`:

```
// I need to create a scoket and bind it first....

/* ---
   Create a socket
   --- */
s = socket(PF_INET, SOCK_DGRAM, 0);

/* ---
   Set up socket end-point info for binding
   --- */
in_addr.sin_family = AF_INET;                   /* Protocol family */
in_addr.sin_addr.s_addr = htonl(INADDR_ANY);    /* Wildcard IP address */
in_addr.sin_port = htons(atoi(argv[1]));        /* Use any UDP port */

/* ---
   Bind socket s
   --- */
bind(s, (struct sockaddr *)&in_addr, sizeof(in_addr))

/* ================================
   How to use select
   ================================ */
int    recv_fds;                  /* I can use a simple int */
struct timeval timeout;           /* Time value for time out */

/* ==============================
   Select the bit for socket s
   ============================== */
recv_fds = 0;                        // Clear all bits in recv_fds
recv_fds = recv_fds | ( 1 << s );  // Set bit for socket s

/* ================================
   Set timeout value to 5 sec
   ================================ */
timeout.tv_sec  = 5;         // Timeout = 5 sec + 0 micro sec
timeout.tv_usec = 0;         // number of micro seconds

int result ;

printf("Now send a UDP message to port %s before %d sec !!!....\n",
        argv[1], timeout.tv_sec );

result =  select(32, (fd_set *)&recv_fds, NULL, NULL, &timeout);
printf("select( ) returns: %d\n", result);
```

- **Example Program:** (Demo above code) ***Example***

  - The `select( )` test Prog file: click here
  - Use this **sender** to send a **message** to the `select` program: click here

How to run the program:

- **Right click** on link(s) and **save** in a scratch directory
- To compile the `select1` test prog: `gcc -o select1 select1.c`
- To compile the `udp4-s3` test prog: `gcc -o udp4-s3 udp4-s3.c -lnsl`
- To run: `select1 8000` on machine **X**
- To send a msg to `select` prog: `udp4-s3 X 8000`

Selecting (clearing/setting) the (socket) descriptors in general

- **The *data type* of the *descriptor sets* `recv_fds`**
  - **What is** a **Descriptor set**:

    - The variable `recv_fds` in the `select( )` call:

      ```
      select ( 32, &recv_fds, NULL, NULL, &timeout );
      ```

      is the ***descriptor set*** (that is being ***checked*** for **readiness**)

  - The **data type** of the `recv_fds` variable is:

    ```
    fd_set   recv_fds ;
    ```

    The `fd_set` **variable** is:

    - a **bit array** of **size** `FD_SETSIZE` **(=1024)** number of bits
      - **Each bit** in the **bit array** variable represents ***one* entry** in the **descriptor table** of the **process (= running program)**

      (Use: **"man select"** to find out more)

  - In my **previous example**, I used a ***short* bit array**:

    ```
    int  recv_fds ;          // int is a bit array of 32 bits
    ```

    This will **work** as long as you use **≤ 32 descriptors** (i.e., use less than around **30 sockets** !!!)

- **Using `select( )` with a `fd_set` variable**
  - **How** the `select( )` function with a `fd_set` variable:

    ```
    fd_set  recv_fds;   // bit array variable

    int s = socket(PF_INET, SOCK_DGRAM, 0);     // Socket that we want to
                                                // test for receive readiness


    (1) clear all bits in the bit array variable  recv_fds
    (2) set the bit position s in the bit array variable  recv_fds
    (3) call:

           select ( FD_SETSIZE, &recv_fds, NULL, NULL, timeout )
    ```

  - **Compare** to my ***simplified*** usage (with an `int` variable):

    ```
    int  recv_fds;   // Shorter bit array variable

    int s = socket(PF_INET, SOCK_DGRAM, 0);     // Socket that we want to
                                                // test for receive readiness

    (1) clear all bits in recv_fds

           recv_fds = 0

    (2) set the bit position s in recv_fds

          recv_fds = recv_fds | ( 1 << s )

    (3) Call:

            select ( 32, &recv_fds, NULL, NULL, timeout );

     (I need to use 32 to tell select( ) that it only need to
      scan the first 32 entries in the descriptor table)
    ```

- **Clearing, Setting and Testing *bits* in a bit array**
  - **Bit array** manipulation **macros** (defined inside `select.h`):

    - **void FD_ZERO(fd_set *fdset)**
      - **Clear all bits** in the **"fdset"** bit array variable

Example:

```
fd_set  recv_fds;

FD_ZERO ( & recv_fds );
```

```
Result:  recv_fds = 0000....000
```

- **void FD_SET(int s, fd_set *fdset)**

  - **Set bit "s"** in the **"fdset"** bit array variable to **1** (i.e., select **"s"** for checking)

Example:

```
fd_set  recv_fs;

FD_ZERO ( & recv_fs );
FD_SET  ( 2, & recv_fs );
```

```
Result:  recv_fds = 0000....0100  // Bit position 2 is set
```

- **void FD_CLR(int fd, fd_set *fdset)**

  - **Clear (reset) bit at position "fd"** in the **"fdset"** bit array

- **int FD_ISSET(int fd, fd_set *fdset)**

  - **Test the bit at position "fd"** in the **"fdset"** bit array

Returns:

- **true (non-zero)** if **bit** is **set**
- **false (= 0)** if **bit** is ***not*** **set**

- **The *general* usage of `select( )` to check receive readiness of a socket**

  - Here is the ***same* demo** program **re-written** using a **`fd_set` typed** variable:

    I have **highlighted** the **changes**

```
// I need to create a scoket and bind it first....

/* ---
   Create a socket
   --- */
s = socket(PF_INET, SOCK_DGRAM, 0);

/* ---
   Set up socket end-point info for binding
   --- */
in_addr.sin_family = AF_INET;                   /* Protocol family */
in_addr.sin_addr.s_addr = htonl(INADDR_ANY);    /* Wildcard IP address */
in_addr.sin_port = htons(atoi(argv[1]));        /* Use any UDP port */

/* ---
   Bind socket s
   --- */
bind(s, (struct sockaddr *)&in_addr, sizeof(in_addr))

/* ================================
   How to use select
   ================================ */
fd_set   recv_fds;                /* I'm using a fd_set var now */
struct timeval timeout;           /* Time value for time out */

/* ==============================
   Select the bit for socket s
   ============================== */
FD_ZERO( &recv_fds );     // Clear all bits in recv_fds
FD_SET( s, &recv_fds);    // Set bit for socket s

/* ================================
   Set timeout value to 5 sec
   ================================ */
timeout.tv_sec  = 5;         // Timeout = 5 sec + 0 micro sec
timeout.tv_usec = 0;         // number of micro seconds

int result ;
```

```
printf("Now send a UDP message to port %s before %d sec !!!....\n",
        argv[1], timeout.tv_sec );

result =  select(FD_SETSIZE, &recv_fds, NULL, NULL, &timeout);
printf("select( ) returns: %d\n", result);
```

- **Example Program:** (Demo above code) ***Example***

  - The ***modified*** `select( )` test Prog file: click here
  - Use this **sender** to send a **message** to the `select` program: click here

How to run the program:

- **Right click** on link(s) and **save** in a scratch directory
- To compile the `select2` test prog: `gcc -o select2 select2.c`
- To compile the `udp4-s3` test prog: `gcc -o udp4-s3 udp4-s3.c -lnsl`
- To run: `select2 8000` on machine **X**
- To send a msg to `select` prog: `udp4-s3 X 8000`

Timeout Programming using the `poll( )` system call

- **Timeout programming using the `poll()` system call**
  - The `poll( )` system call:
    - The `poll( )` performs a **similar task** to `select( )`:
      - `poll( )` waits for **one** of a **set of sockets** to become **ready** to perform **read (receive) / write (send)** operation

  - **Differences:**
    - ***How to*** **specify** the **socket** to test for **readiness**
    - ***How to*** to **specify** the **time out value**

- **Header file for using `poll( )`**
  - **Header file** for `poll( )`:

```
#include  <poll.h>
```

- **The `poll( )` function**
  - The `poll( )` function **syntax**:
    - **Syntax:**

```
#include <poll.h>                /* Header file */

int poll( struct pollfd fds[],  nfds_t nfds,  int n_msec );
```

Meaning of the **parameters**:

```
fds     = an array of "struct pollfd"
          This array contains the file descriptors (sockets) that
     you want to check.

          (The pollfd structure will be explained later)

n_fds   = the length of the "struct pollfd" array
    (= number of descriptors to poll).

n_msec  = the timeout value in milli-seconds

    Special timeout values:

      n_msec =  0 ==>  poll() returns immediately with the poll results
      n_msec = -1 ==>  poll() blocks until a requested event occurs
```

  - **Effect:**
    - The **poll() function** examines the **array** of **file/socket descriptors fds[ ]** for **n_msec** milli-seconds.
    - If ***some*** **descriptor (e.g., a socket)** becomes ***ready*** before the **before** the **time out** value of **n_msec** time:
      - `poll( )` will return **immediately**, and:
        - The **return value** is equal to the **number of descriptors (sockets)** that are ***ready***
        - The `fds[ ]` array will contain **information** on **which** of the **descriptors/sockets** are **ready**
    - If **after waiting** for **n_msec**, there are **not any descriptor (= socket) ready**:
      - The `poll()` **function** will **return**
      - The **return value** will be equal to **0 (ZERO)**

- **The `pollfd` (poll file descriptor) data structure**
  - The `struct pollfd` data structure is as follows:

```
struct  pollfd
{
   int    fd;       // A descriptor (e.g., socket)

   short  events;   // This is a bit array
                    // Each bit represents a certain operation
          //  Set  that  bit  to  check for readiness
         // E.g, one or the bit represent receive

   short  revents;  // ready events
                    // This is also a bit array
                    // Each bit represents whether some operation is ready
}
```

Meaning of the **parameters**:

```
fd       = the (one) file descriptor/socket to be checked

events   = input bit array of events that will be checked

revents  = output bit array of events that are ready
```

The **types** of **events** are **defined** by the **following bit masks**:

```
POLLIN  (= 00001) = ready for input (receive)
POLLOUT (= 00010) = ready for output (send)


Other less used functions:

POLLRDBAND (= 00100) = high priority read data ready
POLLWRBAND (= 01000) = high priority output (send) ready
POLLERR    (= 10000) = error detected
```

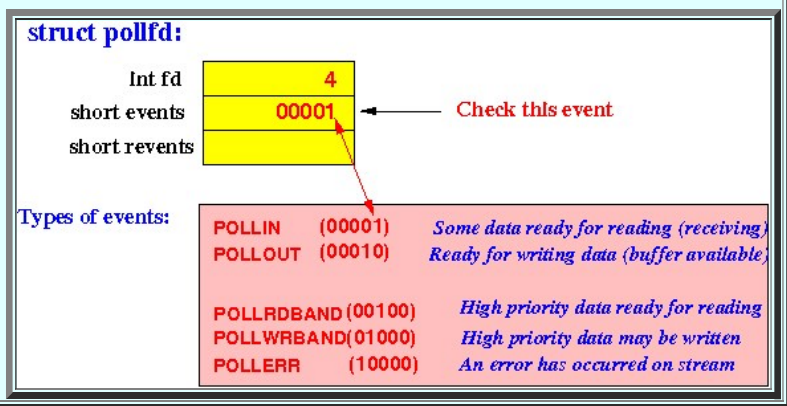
  - **Examples** on **how** the **data field** is used:

    - Suppose you want to **check** if:
      - the **socket `s`** has **data** available for **reading (= receiving)**

      then you **set** the **values** as follows:

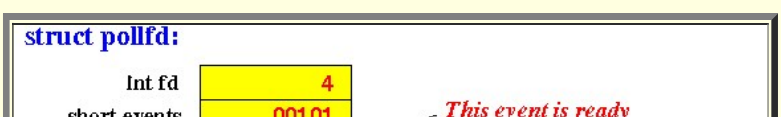
```
fds[i].fd    = s;
fds[i].event = POLLIN;
```

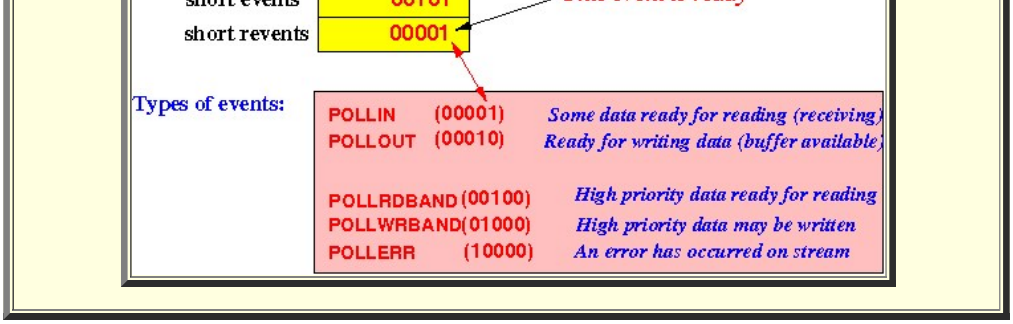
      Result:



- **Checking for ready events in `poll( )`**
  - **Checking** for **readiness**:
    - The **`revents` variable** is an **output field** that contains the **events** that are *ready*

  - **Example:**
    - **If** the **file/socket #4** has some **normal priority data ready (= available)**, then the **`revents`** variable will be **set** to:





- **Setting the descriptor sets in the `pollfd` structure with sockets**
  - **Example 1**:

    - We have **1 socket**:

      ```
      int s = socket( PF_INET, SOCK_DGRAM, 0 );
      ```

      in our **program**

    - We want to **check** the **socket `s`** for ***receive*** **readiness**

    Solution:

    Define an pollfd arary of 1 element:

    ```
    struct pollfd   fds[1];      // Array of 1 element

    (The array element has index 0: fds[0])
    ```

    Initialize the poll structure fd[0] to test s for read readiness:

    ```
    fds[0].fd = s;                              // set descriptor fd to s

    fds[0].events = 0                           // Clear the bit array
    fds[0].events = fds[0].events | POLLIN ;    // Set the POLLIN bit
    ```

    Call poll( ) with timeout:

    ```
    timeout = 100;  // 100 msec = 0.1 sec

    if ( poll ( fds, 1, timeout ) == 0 )
    {
       // Time out
    }
    else
    {
       // Ready
    }
    ```

  - **Example 2**:

    - We have **2 sockets:**

      ```
      int s1 = socket( PF_INET, SOCK_DGRAM, 0 );

      int s2 = socket( PF_INET, SOCK_DGRAM, 0 );
      ```

      in our **program**

    - We want to **check** when ***either*** **socket `s1` or `s2`** has **some message** that can be **received**

    Solution:

    Define a pollfd arary of 2 elements (because we have 2 sockets):

    ```
    struct pollfd   fds[2];      // Array of 2 element

    (The array elements are: fds[0] and fds[1])
    ```

Initialize the poll structure fd[0] to test s1 for read readiness:

```
fds[0].fd = s1;                              // set descriptor fd to s1

fds[0].events = 0                            // Clear the bit array
fds[0].events = fds[0].events | POLLIN ;     // Set the POLLIN bit
```

Initialize the poll structure fd[1] to test s2 for read readiness:

```
fds[1].fd = s2;                              // set descriptor fd to s2

fds[1].events = 0                            // Clear the bit array
fds[1].events = fds[0].events | POLLIN ;     // Set the POLLIN bit
```

Call poll( ):

```
timeout = 100;  // 100 msec = 0.1 sec

if ( poll ( fds, 2, timeout ) == 0 )
{
   // Time out
}
else
{
    for ( i = 0; i < nfds; i++ )
    {
     i f   (  (fds[i].revents | POLLIN) == POLLIN )
        {
         / /  fds[i].fd is ready
        }
    }
}
```

- **Example: Timeout programming with `poll()` system call**
  - Here is the **`select` example** (discussed here: **click here**) **re-written** using **`poll()`**:

```
// I need to create a scoket and bind it first....

/* ---
   Create a socket
   --- */
s = socket(PF_INET, SOCK_DGRAM, 0);

/* ---
   Set up socket end-point info for binding
   --- */
in_addr.sin_family = AF_INET;                   /* Protocol family */
in_addr.sin_addr.s_addr = htonl(INADDR_ANY);    /* Wildcard IP address */
in_addr.sin_port = htons(atoi(argv[1]));        /* Use any UDP port */

/* ---
   Bind socket s
   --- */
bind(s, (struct sockaddr *)&in_addr, sizeof(in_addr))

/* ================================================
   How to use poll( )
   ================================================ */
struct pollfd  fds[1];                    // 1 socket
int    timeout = 5000;                    // 5000 msec = 5 sec

/* ======================================
   Set up fds[0] to monitor socket s
   ====================================== */
fds[0].fd = s;                            // Test socket s
fds[0].events = 0;                        // Clear events
fds[0].events = fds[0].events | POLLIN; ; // Set INPUT bit

int result ;

printf("Now use udp4-s3 and send a UDP message to port %s before %d sec\n",
        argv[1], timeout/1000 );

/* ================================
   Here we go....
   ================================ */
result =  poll( fds, 1, timeout );
printf("poll( ) returns: %d\n", result);
```

  - **Example Program:** (Demo above code) *Example*

- The `poll( )` test Prog file: click here
- Use this **sender** to send a **message** to the `select` program: click here

How to run the program:

- **Right click** on link(s) and **save** in a scratch directory

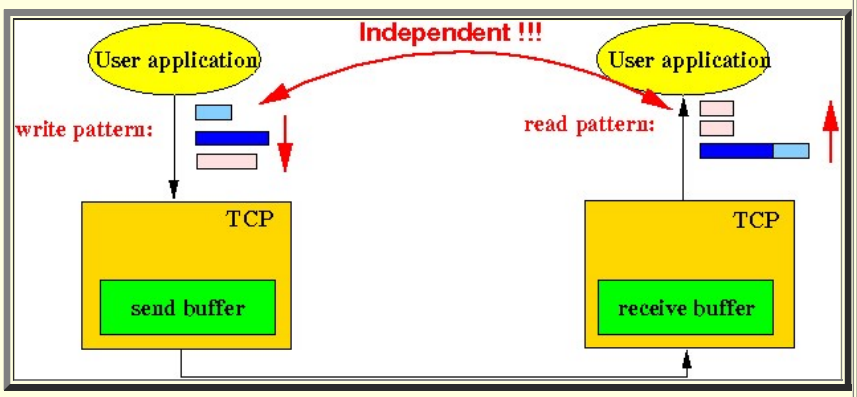
- To compile the `poll1` test prog: `gcc -o poll1 poll1.c`
- To compile the `udp4-s3` test prog: `gcc -o udp4-s3 udp4-s3.c -lnsl`

- To run: `poll1 8000` on machine **X**
- To send a msg to `select` prog: `udp4-s3 X 8000`

Intro to TCP communication

- **Properties of TCP communication**
  - **Properties** of **TCP communication**:
    - **Reliable**:
      - **Data** sent using a **TCP socket** is **received** ***once*** and **in the transmission order**
    - ***Stream* oriented**
      - **Data transmitted** using **TCP** can be **received** in ***different*** **chunk sizes** that ***transmitted*** !!!

        Graphically:

        See: click here
  - **Note** for **CS450 students**:
    - A **TCP socket** behaves **exactly the same** as a **UNIX** ***pipe***/

      Except that:
      - You use ***one* (= same) descriptor** for **reading/writing** !!!

        I.e.:
        - A **TCP socket** is ***bi-directional***

- **TCP and Network programs**
  - **Comment:**
    - **TCP sockets** are the ***most commonly* used communication end point** in **network programs**

      because:
      - **Retransmissions** (to provide ***reliablity***) is **handled** by the **TCP protocol** itself.

        (So the **programmer** can **focus** on **programming** the **application** itself....)
    - **Virtually *every* network program** is written using **TCP sockets**

    (**UDP sockets** are only used by **network research projects** that experiment with **new network transmission algorithms**)

TCP sockets and TCP connections

- **Creating a TCP socket**
  - **Create** a **TCP socket**:

```
int s;          // Descriptor index.

s = socket(AF_INET, SOCK_STREAM, 0);

                // Creates a TCP socket in the per-process descriptor table
  // Returns the index in the per-process descriptor table
```

- **TCP sockets and TCP connections**
  - **TCP socket:**
    - **TCP socket** = a **end point** of **communication** ***using*** the **TCP protocol**
      - **Unlike UDP**, the **TCP protocol** does **not *only*** use the **TCP socket** to **identify** a **programming communication endpoint**

  - **TCP *connection*:**
    - **TCP connection** = a **communication link** between **2 network programs** that uses **TCP sockets**
      - **Network programs** communicate with one another using **TCP *connections***
    - A **TCP connection** is **uniquely identified** by:

```
(srcIP, srcPort#, destIP, destPort#)
```

- **Multiple TCP connections using the same (IP-address, Port#)**
  - **Fact:**
    - **Because:**
      - A **TCP connection** (= **communication pathway between 2 network programs**)) is **identified** by:

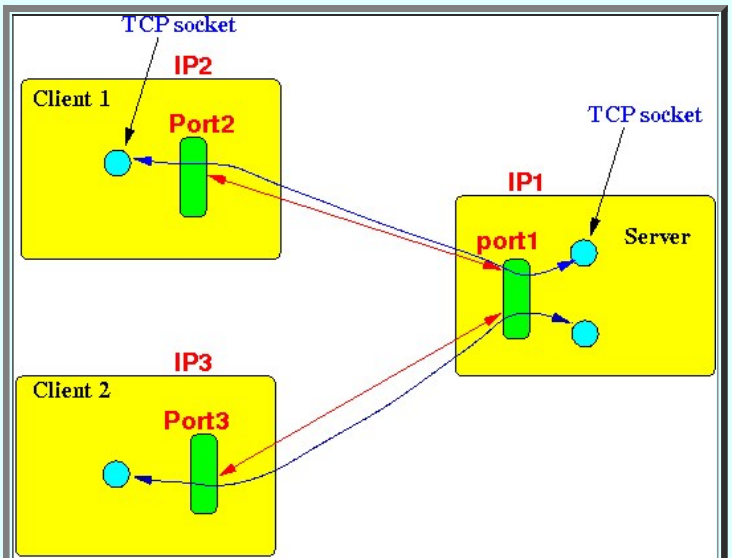
```
(srcIP, srcPort#, destIP, destPort#)
```

        it is **possible** to **form** the following ***different*** **TCP connections**:

```
TCP connection 1:    ( IP1, port1, IP2, port2 )

TCP connection 2:    ( IP1, port1, IP3, port3 )
```

        using the ***same*** `(IP1, port1)`

    Graphically:



This is the **situation** where a **web-server** running on the **host IP1** is **servicing 2 web clients**:

- One **client** running on **host IP2**
- One **client** running on **host IP3**

In other words:

- ***One*** **TCP port** in the **Operating System kernel** can **support** ***multiple*** **TCP connections** for the ***same*** **IP address**

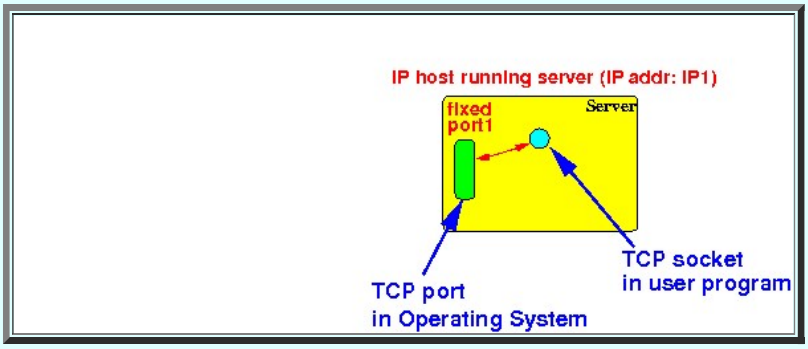
Programming with TCP

- **TCP programming model**
  - **Programming** using **TCP** will ***always*** use the ***client/server* paradigm**:

    - The **server** will **start *first***:
      - The **server** will ***wait*** for **requests** from some **client**
      - When a **request** is **received**, the **server** will **send back** a **reply**
    - The **client** will **start *later***:
      - The **client** sends a **request** to the **server**
      - **Then** the **client waits** for a **reply**

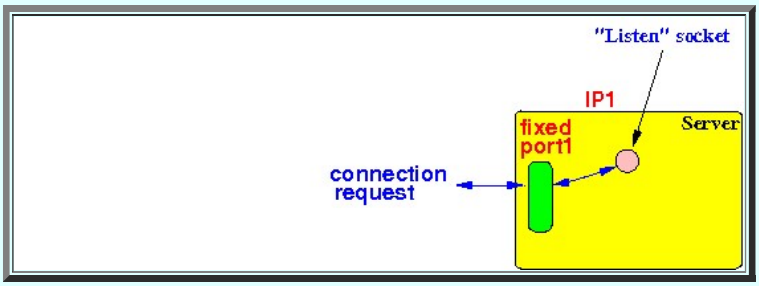
- **How to use TCP in client/server programming**
  - **Steps** in writing a **client/server network program** using **TCP**:

    1. The **server program** first creates a **TCP socket** and **binds** it to a ***fixed* TCP port#**:

       (We have **learned this step** before in **UDP programming**)

    2. Then the **server program change** the **socket** to ***listen* socket** that is used to **receive connection requests**:

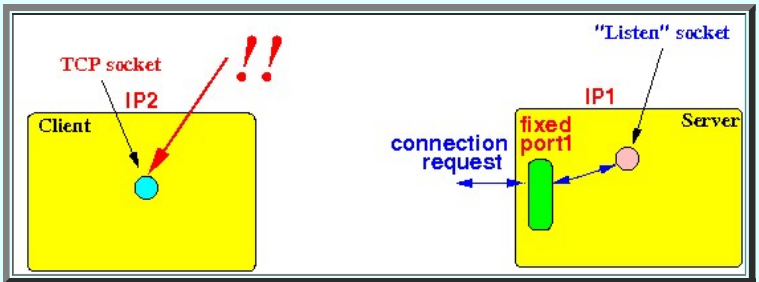
       

       The ***listen* socket** is a **special type** of **socekt** that **execute** the **"3-way handshake" algorithm** to **establish** a **reliable connection**

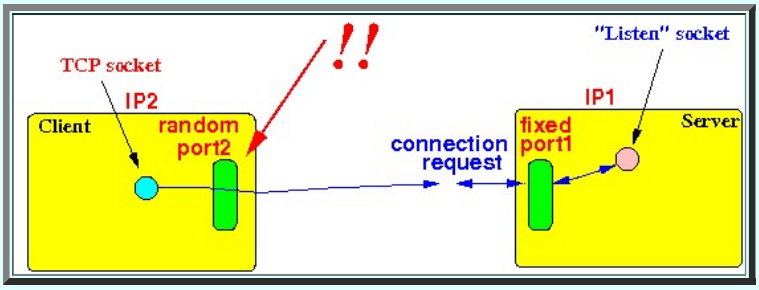
       - The **server** will **now wait** for **"client connection requests"**

         (This is done using the `accept( ... )` **system call** - discussed **later**)

    3. A **client program** first **create** a **TCP socket**:

Bind the **socket** to an ***arbitrary*** **TCP port #** and **use** the **socket** to make a **connection request** to the **server program** (at the ***listen*** **socket**):

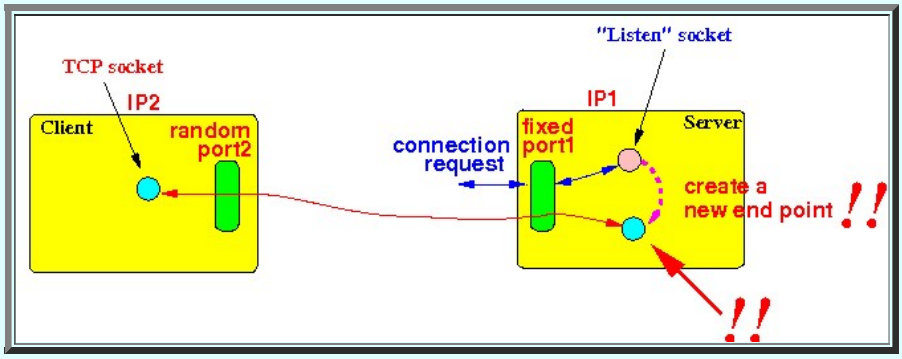


The **client program** use the `connect( )` **system call** to do this

(The **TCP connection process** will establish a **pair** of **send/ACK sequence numbers** for the **sliding window algorithm**)

---

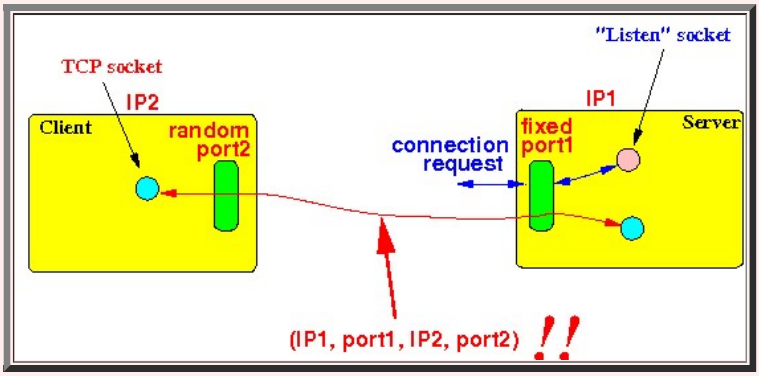
4. When the **processing** of the **connection request** is **complete**, the **TCP module** at the **server program** will **create** a ***new*** **TCP socket** for the **TCP connection**:



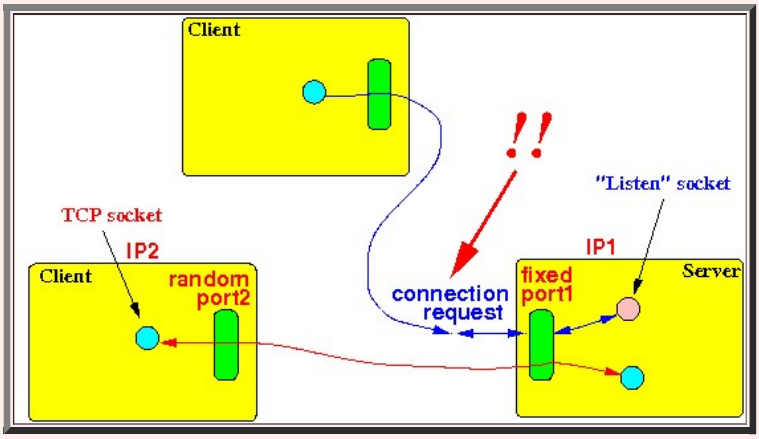
---

5. **Result:**

   1. The **client program** and the **server program** can **communicate** with each other through the **TCP connection** `(IP1, port1, IP2, port2)`:



   2. ***And***, the **server** can ***still*** accept ***other*** **connection requests** using the ***listen*** **TCP socket**:



   The ***listen*** **socket** remains **free** to **receive** ***new*** **connection requests** from ***other*** **clients** !!!

   - This is **how** a **web server** or **SSH server**, and so on... operates

Programming the *server* using TCP

- **Programming a TCP server application**
  - **Recipe (steps)** for **coding** a **TCP server** application:

    - **Create** a **TCP socket**

      ```
      int s = socket( AT_INET, SOCK_STREAM, 0 );
      ```

      This **socket** is ***only*** **used** for:

      - **Receiving TCP connection requests** from **clients** !!!

    - **Bind** the **socket** `s` to a ***fixed*** **TCP port#**:

      ```
      struct sockaddr_in in_addr;

      in_addr.sin_family      = AF_INET;
      in_addr.sin_addr.s_addr = htonl(INADDR_ANY);
      in_addr.sin_port        = htons(SrvPortNumber);  // a fixed port#

      bind(s, (struct sockaddr *)&in_addr, sizeof(in_addr))
      ```

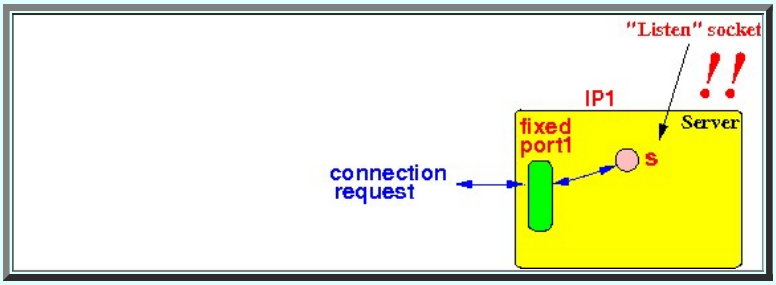
      Result:

      - **TCP messages** can be **received** on this **Internet network address**

    - **Enable** the **socket** `s` to **receive** client's **connection requests**:

      ```
      listen( s,  #pending-connect-requests );
      ```

      Result:

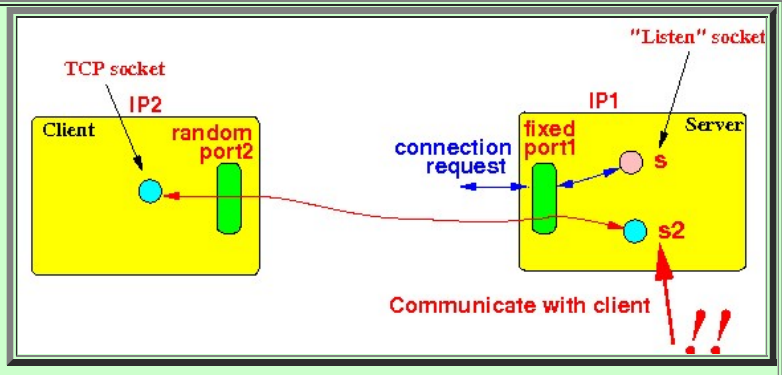
      

      More details on the `listen( )` function in ***later***

    - **Establish TCP connections** with **clients**:

      ```
      int s2;       // Another socket variable

      s2 = accept( s, &in_addr, &length );
      ```

      Result:

      - The **socket** `s2` is used to **communicate** with a **client**

      - The **socket** `s` will ***still*** be used to **establish TCP connections** with **(other) clients**

- **The `listen` system call**
  - **Syntax** of the `listen( )` function:

```
int listen(int s, int backlogLength)

s             = a socket descriptor
backlogLength = max. number of pending requests allowed

listen returns

   0 for success,
  -1 for error
```

  - **Effect** of the `listen(s, ... )` **call**:

    - `listen(s, ... )` **enables** the **socket** `s` to **accept** incoming **connection requests** sent by **TCP *client* applications**

      Note:

      - A **TCP socket** `s` does ***not*** **accept** client's **connection requests**
      - You must **enable** the **connection processing** with the `listen(s, ...)` call

---

- **The `accept` system call**
  - **Syntax** of the `accept( )` function:

```
int accept ( int s,  struct sockaddr *cli_addr,  socklen_t *addrlen)

Input:
       s           = a listen socket

Output:
       cli_addr    = network address (will contain (IPaddr, Port#) of client
       addrlen     = length of addr

Return value:
       a new TCP socket descriptor

       Note: use the returned TCP socket to

                   send messages to client
        receive  messages  from  client
```

  - **Effect** of `accept( )` function:

    - The `accept( )` function will **block** until a **TCP connection** is ***successfully*** **established**:
      - an **incoming** ***connection*** **request** is **received**
      - a ***consistent*** **TCP state** has been **negotiated** (**send/ACK numbers**)

    - The **TCP connection establishment** will **perform** the **following** in the **OS kernel**:
      - **Create** a ***new*** **descriptor entry** in the **per-process descriptor table** for a **TCP socket containing**:
        - The **state (send/ACK numbers)** of the **TCP connection**
        - The **send/receive buffer spaces** for the ***new*** **TCP connection**

    - The `accept( )` function will **return** the **(socket) descriptor**
      - The **server** must use the **return descriptor** to **send/receive data** to/from the **client**

        (Because this **socket** contains the **TCP state** for ***reliable*** **communication**)

- **Structure of a TCP server application program**

- **Structure** of a ***server*** **TCP application** written in **C program code:**

```
int s_listen;                   // The listen socket
int s_data;               / /   T h e  data socket

struct sockaddr_in in_addr;     // Help structures to form netw address
struct sockaddr_in client;
int    length;

/* ---------------------------------------
   Create the TCP listen socket
   --------------------------------------- */
s_listen = socket( AF_INET, SOCK_STREAM, 0 );     // Create a TCP socket

/* ------------------------------------------------------
   Bind the socket to a network address
   This makes the socket identifiable by (IPaddr, Port#)
   ------------------------------------------------------ */
in_addr.sin_family = AF_INET;
in_addr.sin_addr.s_addr = htonl(INADDR_ANY);
in_addr.sin_port = htons(SrvPortNumber);

if ( bind(s_listen, (struct sockaddr *)&in_addr,     //  Bind socket to (IP-addr, Port#)
                    sizeof(in_addr)) == 0 )
{
   perror("Bind: ");
   exit(1);
}

/* ------------------------------------------------------
   Enable socket s_listen to accept connect requests
   ------------------------------------------------------ */
listen(s_listen, 5);          // Change socket to a "listen" socket

/* ------------------------------------------------------
   Accept TCP connection calls
   ------------------------------------------------------ */
while ( 1 )
{
   length = sizeof(struct sockaddr_in);

   /* ======================================================
      Server program waits for client's connection requests
   ====================================================== */
   s_data = accept( s_listen, &client, &length);     // s_data = data socket

        // The accept( ) function will a
      / /  data socket of the TCP connection
  // establish with the client program


   /* ==========================================
      Now you can send and receive data
 u s i n g   t h e  s_data socket
   ========================================== */

      read ( s_data, ....)     to receive client's requests
      write( s_data, ....)     to send data to client
      close( s_data)           to disconnect
}
```

Programming the client using TCP

- **Structure of a TCP client application**
  - **Recipe (steps)** for **coding** a **TCP *client* application** ("client-side" programming):

    - **Create** a **TCP socket**
      - You **do *not*** need to **bind** the **socket**:
        - The **client** can use an ***arbitrary*** **TCP port#** to **communicate** with the **server**

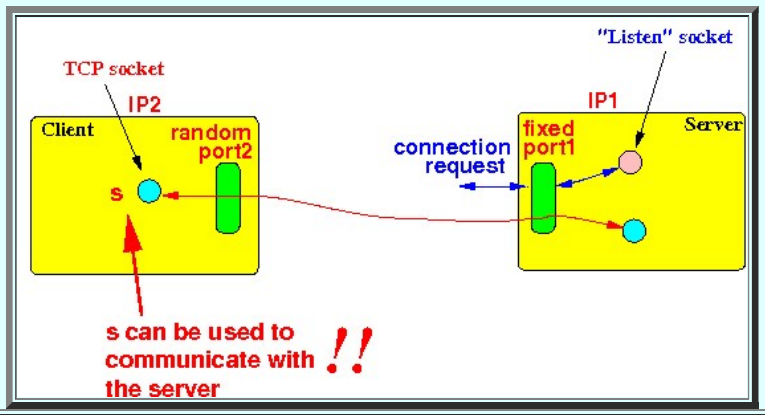
    - **Setup** a **network address** (using a `struct sockaddr_in` variable) containing:
      - The **(IP address, TCP port#)** of the **server**

    - **Establish** the **TCP connection** to the **server** using:

      ```
      connect ( s, server-Netw-addr, sizeof(struct sockaddr_in) );
      ```

      `connect( )` will:
      - **Allocate** the **send/receive buffer space** for the **descriptor** `s` in the **OS kernel**
      - **Send** the **TCP connection request**
      - **Complete** the **TCP connection** protocol (**3-way handshake** !!!)

      Result of `connect( )`:

- **The `connect( )` function**
  - **Syntax** of the **`connect()` system call:**

    ```
    int connect(int s, struct sockaddr *server, length)

    Input:
       s      = an unbounded socket !!!
       server = a "struct sockaddr_in" structure containing
                the IP-address and the port# of the server
       length = must be equal to "sizieof(struct sockaddr_in)"

    Return value:
            0  = success
           -1  = failed
    ```

  - **Effect:**
    - The `connect()` system call will establish a ***reliable*** **connection** between the **client** and the **server application** given at the **Internet address** in `server`
      - The `server` structure must contains the ***listen*** **port** of the **server program**
    - The `connect( )` function will **reserve** an ***unused*** **TCP port** in the **Operating System** and ***bind*** this **TCP port** to the **descriptor entry** `s`

- The **state information** (such as ***send*** **sequence number** and ***receive*** **sequence number** used in the **reliable transfer protocol**) of the **TCP connection** is stored in the **socket descriptor table entry** (and accessed through the **socket variable** `s`)

- ***Result:***

  - After the `connect()` **call** is **completed**, the **client application** can use the **socket variable** `s` to:

    - **send** messages (= requests) to the **server's data TCP port**
    - **receive** messages (= replies) from the **server's data TCP port**

- **Structure of a TCP client application program**

  - **Structure** of a **TCP client** application:

```
int s;                              // Socket variable
struct sockaddr_in server;          // Network addr for server

/* ---------------------------
   Create TCP socket
   --------------------------- */
s = socket( AF_INET, SOCK_STREAM, 0 );   / Create a TCP socket

/* ----------------------------------
   Setup the server's Network Address
   ---------------------------------- */
server.sin_family      = AF_INET;
server.sin_addr.s_addr = htonl(SrvIP-address);
server.sin_port        = htons(SrvPortNumber);

/* ------------------------------------------------------------
   Make a connection request to the server
   (The server must be in an accept() call to be successful)
   ------------------------------------------------------------ */
if ( connect(s, server, sizeof(struct sockaddr_in) ) == 0 )
{
   perror("connect: ");
   exit(1);
}

// After this, the client can use the TCP socket s to communicate
// (to the data TCP port) with server
//
// Use:

      read(s, ....)      to get server's replies
      write(s, ....)     to send data to server
      close(s)           to disconnect
```

Sending/receiving data with TCP

- **Sending messages on TCP sockets**
  - **Sending (= writing)** with a **TCP socket `s`:**

```
int write(int s, void *buf, int nbytes)

Input parameters:
    s       = the TCP socket
    buf     = buffer address containing the data to be sent
    nbytes  = number of bytes in buffer "buf" to send

Return value:
     -1   = error
    >=0   = the actual number of bytes sent
```

Effect:

- `write(s, buf, nbytes)` will *try* to **send** `nbytes` in `buf` through the **socket** `s`

- **Usually**:

  - `write(s, buf, nbytes)` will **send** all of the `nbytes`

  However:

  - *If* the **send buffer space** is **(near)** ***full***:

    - `write(s, buf, nbytes)` **can** send ***less*** **than** `nbytes`

      (Then the **return value** will be **less** than **nbytes**)

- **A commonly used TCPsend( ) function**
  - The following **function** is **guaranteed** to **send** all `nBytes`:

```
int TCPsend( int s, char *buf, int nBytes )
{

   int k, n;

   k = 0;                   // No bytes sent yet....

   /* ------------------------------------------------------------
    Call write( ) as long as we did not send nBytes....
   ------------------------------------------------------------ */
   while ( k < nBytes )
   {
      n = write( s, &buf[k], nBytes-k ); // Send "remaining" nBytes-k bytes

      /* ======================================
   Check if there was an error....
   ====================================== */
      if ( n == -1 )
         return -1;                                  // Return error....

      k = k + n;             // We sent n more bytes, add to k
   }

   return nBytes;            // Success...
}
```

- **Receiving messages on TCP sockets**
  - **Receiving (= reading)** data with a **TCP socket `s`:**

```
int read(int s, void *buf, int nbytes)

Input:
    s       = TCP socket
    buf     = address of the buffer to received read data
    nbytes  = # bytes that we wish to receive

Return value:
     -1   = error
    >=0   = the actual number of bytes transfered
```

Effect:

- `read(s, buf, nbytes)` will **block** until ***some*** **data** is **available**

- **When** **data** is **available**, `read(s, buf, nbytes)` will ***try*** to:
  - **Transfer** `nbytes` from the **receive buffer** of **socket** `s`
  - ***Store*** the **data** in the `buf` variable

- **Usually**:
  - `read(s, buf, nbytes)` will **transfer** exactly `nbytes`

  However:
  - **When** the **receive buffer** is **near** ***empty***, the `read` operation ***may*** **return** with ***less*** **than** `nbytes`.

- **A commonly used TCPrecv( ) function**
  - The following **function** is **guaranteed** to **receive** exactly `nBytes`:

```
int TCPrecv( int s, char *buf, int nBytes )
{

   int k;

   k = 0;                  // No bytes received yet....

   /* ------------------------------------------------------------
    Call read( ) as long as we did not receive nBytes....
   ------------------------------------------------------------ */
   while ( k < nBytes )
   {
      n = read( s, &buf[k], nBytes-k );  // Receive "remaining" nBytes-k bytes

      /* ========================================
   Check if there was an error....
   ======================================== */
      if ( n == -1 )
         return -1;                              // Return error....

      k = k + n;             // We sent n more bytes, add to k
   }

   return nBytes;
}
```

- **Closing a TCP connection**
  - **Closing** a **TCP connection**:

```
int s;     // TCP socket used to communicate

close(s);
```

- **For CS450 students**
  - **Programming remark** for **CS450 students**:
    - A **TCP socket** behaves **exactly** like a **pipe** or an **file descriptor** of an open file !
      - The `read( )/write( )` call can ***return*** with ***less*** **than** the **specified amount of data** when we reach the ***end*** **of the file** !!!

  The ***similarity*** is **obvious** in the **behavior**:

  - a **pipe** or an **opened file** in **UNIX** is a ***stream*-oriented device**:
    - the ***unit*** **(smallest amount) of data** that can be **read/written** is ***one*** **byte**

Programming a *serial* server (easy to understand, no self-respecting programmer will do it)

- ***Serial* server: serving *one* client at a time**
  - **Serial server:**

    - A **serial server** can **attend to *one* client** at a time
      - While **processing** the **client**, the **TCP server** **can *not* accept** a ***new* connection request**

    (Because the **server** is ***not* able** to **execute** the `accept( )` call.....)

  - **Structure** of a ***serial* TCP server**:

```
/* ---------------------------
   1. Create a TCP socket
   --------------------------- */
s1 = socket(AF_INET, SOCK_STREAM, 0);

....
/* ------------------------------------------------------------
   2. Bind socekt to a IP network address (IP addr, TCP port#)
      so that the socket can be identified
   ------------------------------------------------------------ */
if ( bind(s1, (struct sockaddr *) &server, sizeof(server) ) < 0 )
 { perror("bind:");
   exit(1);
 }

/* ------------------------------------------------------------
   3. Change socket into a listen socket

   A listen socket run the TCP connection establishment protocol
   that starts a "sliding window session"
   ------------------------------------------------------------ */
listen(s1, 5);

/* ==================================================
   We want to keep the server running forever...
   That's why we use a while-loop...
   ================================================== */
while (1)
{
   len = sizeof(from);

   /* ------------------------------------------------------
      4. Make the server program wait for a connection request
    from a client
------------------------------------------------------------ */
   if ( (s2 = accept(s1, (struct sockaddr *) &from, &len)) < 0 )
   {
      perror("accept: ");
      exit(1);
   }

  // A serial server will continue execution here without creating a new process

   /* ---------------------------------------------
      Server program use socket s2 to
commninicate with the client
--------------------------------------------- */

   /* ---------------------------                  ---+
      Processing loop                                |
      --------------------------- */                 |
   while ( read( s2, buf, .... ) > 0 )               | Change this loop
         {                 |   f o r   d i f f e r e n t   k i n d s
      process  d a t a   i n   b u f          |   o f   s e r v i c e s
      write(s2, reply, ...); // Send reply back         |
           }                                  - - - +

   close(s2);   // Close socket when done

   // We NOW loop back up and
   // accept the next connection request
}
```

- **Example 1: "print to terminal" server**
  - This processing loop will ***print*** each character that is **received**:

```
   /* -------------------------------
      Processing loop
------------------------------- */
   while (read(s2, &c, 1) > 0)    // Receive 1 character
   {
      putchar(c);                 // Print it
      fflush(stdout);
   }
```

```
    close(s2);
```

- The **entire** ***TCP server*** **program:**

```
main(int argc, char *argv[])
{
     int s1;    /* s1 = listen socket */
     int s2;    /* s2 = connect socket */
   struct sockaddr_in me; /* My own Internet address */
   struct sockaddr_in myname; /* Used to retrieve my TCP port */
    int   myname_len;

   struct sockaddr_in from; /* Internet address from client */
    int len;    /* length associated with `from' */

    char c;    /* Used to get data from connection */

   if (argc == 1)
    { printf("Usage: %s port\n", argv[0]);
   exit(1);
    }

   /* ------------------------------------------------
   Create a TCP socket
   ------------------------------------------------ */
   s1 = socket(AF_INET, SOCK_STREAM, 0);

   /* ------------------------------------------------------------
  Set up Internet address (IPaddr, Port#) to bind the socket
  ------------------------------------------------------------ */
   me.sin_family = AF_INET;    /* Put in the family */
   me.sin_port = htons(atoi(argv[1]));  /* Put in the port number */
   me.sin_addr.s_addr = htonl(INADDR_ANY); /* Use wildcard address */

   /* ------------------------------------------------
      Make TCP socket "identifiable"
   ------------------------------------------------ */
   if ( bind(s1, (struct sockaddr *) &me, sizeof(me) ) < 0)
    { perror("bind:");
   exit(1);
    }

   /* ========================================
   Change s1 into a "listen" socket
       (with max 5 pending connection reqs)
   ======================================== */
   listen(s1, 5);

   /* ------------------------------------
   Keep the server running forever....
   ------------------------------------ */
   while (1)
   {
   len = sizeof(from);

      /* ====================================================
   Make the server program wait for a connection
    from a client....
   ==================================================== */
     if  (  (s2 = accept(s1, (struct sockaddr *) &from, &len)) < 0)
      {
       perror("accept: ");
        exit(1);
       }

   /* ========================================
      Use socket s2 to get data from client
    and print it
      ======================================== */
    while  (read( s2 , &c, 1) > 0)
      {
         putchar(c);     // Print each character read
       fflush(stdout);
       }

    close( s2 );
   printf("\n\n...Done...\n\n");
    }
}
```

- The **client** simply sends a **serie** of **characters** to the **server**:

```
/* ========================================================
   The socket creation and connection to server
   have been omitted for brevity
   ======================================================== */

char c;

while ((c = getchar()) != EOF )        // Read a character and store in c
{
   write(s, &c, 1);                    // Send the character to server
}
```

- **Example Program:** (Demo above code) ***Example***
  - The **server** Prog file: click here

- The **client** Prog file: click here

How to run the program:

- **Right click** on link and **save** in a scratch directory
- To compile: `gcc -o tcp1-cli tcp1-cli.c`
  `gcc -o tcp1-serv tcp1-serv.c`
- To run: on machine **X**: `tcp1-serv 8000`
  on machine **Y**: `tcp1-cli X 8000`

- **Example 2: addition server**
  - The **server** will ***read*** **2 integers** and then **send** the **sum** back to the **client**:

```
    /* ================================================
       This is the processing loop of the server
    ================================================ */
    int x, y, z;

    while ( 1 )
    {
       int x1, y1, z1;

       if ( read( s2, &x1, 4) <= 0 )   // An int is 4 bytes
          break;

       x = ntohl(x1);                  // Convert to host order

       if ( read( s2, &y1, 4) <= 0 )
          break;

       y = ntohl(y1);                  // Convert to host order

       z = x + y;
       printf("Client data: %d, %d, send sum = %d\n", x, y, z);

       z1 = htonl(z);
       write ( s2, &z1, 4 );
    }

    close(s2);
```

  - The **client** will send the **server 2 integer values** and then **receive back** an **integer (sum)** and prints it:

```
  /* ============================================================
     The socket creation and connection to server
     have been omitted for brevity
     ============================================================ */

  int x, y, z;

  while ( 1 )
  {
     printf("Enter first number (0 to exit): ");
     scanf("%d", &x);                       // Read first integer

     if ( x == 0 )
     {
        close(s);
        exit(0);
     }

     printf("Enter second number: ");
     scanf("%d", &y);                       // Read second integer

     int x1, y1, z1;

     x1 = htonl(x);          // Convert them to network byte order first
     y1 = htonl(y);

     write(s, &x1, 4);       // Send the integers
     write(s, &y1, 4);

     /* ===================================
        Receive the answer from the server
  =================================== */
     read(s, &z1, 4);
     z = htonl(z1);                         // Convert answer to host byte order

     printf("%d + %d = %d\n", x, y, z);     // Print it
  }
```

  - **Example Program:** (Demo above code) ***Example***
    - The **server** Prog file: click here
    - The **client** Prog file: click here

How to run the program:

- **Right click** on link and **save** in a scratch directory
- To compile: `gcc -o tcp1a-cli tcp1a-cli.c`
  `gcc -o tcp1a-serv tcp1a-serv.c`
- To run: on machine **X**: `tcp1a-serv 8000`
  on machine **Y**: `tcp1a-cli X 8000`

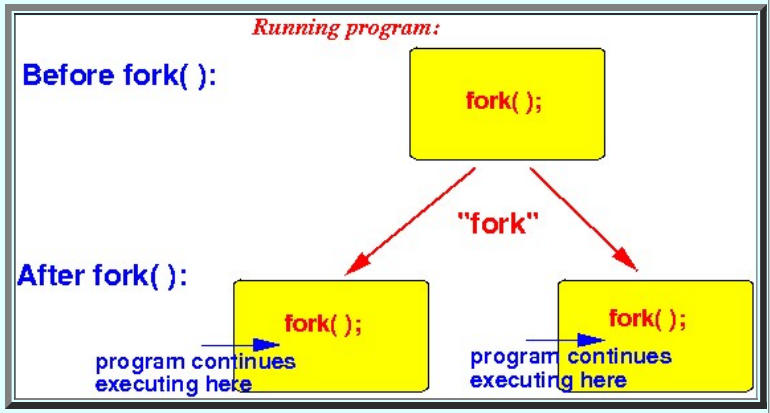
The fork( ) system call

- **The `fork( )` system call (from CS450)**
  - **Syntax** of the `fork( )` system call:

    ```
    int fork( )
    ```

    Effect:

    - The **running program** that executes the `fork( )` function will be ***cloned***:

  - **Example:**

    ```
    int main(int argc, char *argv[] )
    {
       printf("Hello World !\n");

       fork( );

       printf("Good-bye World !\n");
    }
    ```

    Output:

    ```
    Hello World !
    Good-bye World !          // Printed twice !!!
    Good-bye World !
    ```

  - **Example Program:** (Demo above code) ***Example***
    - Prog file: click here

    How to run the program:

    - **Right click** on link(s) and **save** in a scratch directory
    - To compile: `gcc fork1.c`
    - To run: `a.out`

- **Using `fork( )`**
  - **Note:**

    - It may sound **stupid** at first to ***clone*** a **running program** because:
      - ***why*** on earth would you want to run the ***same program*** **twice** ???

  - **Using** the `fork( )` system call:

    - The ***is*** a **way** (discuss soon) to make the **2 identical programs** continue **2** ***different*** **ways**:
      - Use an **`if`-statement**

    - **Then:**

- ***One* copy** of the **program** will execute the **`if`-part**

- The ***other* copy** of the **program** will execute the **`else`-part**

- **Parallel server**
  - **Parallel server**:
    - **Parallel server** = a **server** than can **process *multiple* connection requests** from **clients** at the ***same* time**

  - **Structure** of a **parallel server**:
    - Make a **clone** of **yourself (= the server program)**

    - Have the ***clone*** provide **service** to the **client**

    - **In the mean time**, ***you* (= the original server program)** can ***accept*** other ***connection* requests** made by ***other* clients** !!!

- ***Parent* and *child* process**
  - ***Parent* process** and ***child* process**:
    - ***Parent* process** = the ***original* running program** that called the **`fork( )`** function

      ---

    - ***Child* process** = the ***clone*** of the ***original* running program**

  - ***Very important* fact:**
    - The ***return value*** of the **`fork( )`** function is ***different***:
      - For the ***parent* process**, and
      - for the ***child* process**

    - The **return value** of **`fork( )`** for the ***parent* process** is:
      - The **process id (≠ 0)** of the **child process**

    - The **return value** of **`fork( )`** for the ***child* process** is:
      - **0 (*always*)**

  - **Example:**

```
int main(int argc, char *argv[] )
{
   printf("Hello World !\n");

   int i = fork( );

   printf("i = %d\n", i);            // Show the return value of fork()
}
```

Sample output:

```
Hello World !
i = 10814          <--- This line was printed by the parent process
i = 0              <--- This line was printed by the child  process
```

  - **Example Program:** (Demo above code) ***Example***
    - Prog file: click here

How to run the program:

- **Right click** on link(s) and **save** in a scratch directory

- To compile: `gcc fork2.c`
- To run: `a.out`

- **How to make *parent* process and *child* process run *different* code**
  - `fork( )` is ***always*** used inside an `if`-statement:

```
if ( fork( ) != 0 )
{
   code executed by the parent process
}
else
{
   code executed by the child process
}
```

  - **Example:**

```
int main(int argc, char *argv[] )
{
   printf("Hello World !\n");

   if ( fork( ) != 0 )
   {
   printf("Hello, I am the parent process\n");
   }
   else
   {
   printf("Hello, I am the child process\n");
   }
}
```

  - **Example Program:** (Demo above code) ***Example***
    - Prog file: click here

How to run the program:

- **Right click** on link(s) and **save** in a scratch directory
- To compile: `gcc fork3.c`
- To run: `a.out`

  - A more **advanced example**:
    - The **parent process** will keep created **child processes** when you enter a **line**:

```
int main(int argc, char *argv[] )
{
   char line[1000];

   while ( 1 )
   {
   printf("Enter a line:\n");
      scanf( "%[^\n]", line );
   getchar();

    if  ( fork( ) != 0 )
      {
      printf("Parent process: I will go read a line again\n");
      }
    else
      {  /* ----------------------------------------
      Code that is executed by the child process
      ---------------------------------------------- */
      printf("Child process... your line is:     %s\n", line);
      printf("Child process... here is it again: %s\n", line);
      printf("Child process: Done !!\n");
         exit(0);                      // Child process exits !!
      }
   }
}
```

  - **Example Program:** (Demo above code) ***Example***
    - Prog file: click here

How to run the program:

- **Right click** on link(s) and **save** in a scratch directory
- To compile: `gcc fork4.c`

- To run: `a.out`

Programming a *parallel* server with fork( )

- ***Parallel* server: serving multiple clients**
  - **Parallel server:**

    - A **parallel server** can **attend to *many* client** at the same time

      Actually, the **reality** is that a **parallel server** make a **clone** of itself to **attend to *one* client** at a time...

    - The **next request** can **accepted** by the **original**, while a **clones** service the **other requests**

  - **Coding a parallel server:**

```
/* ---------------------------
   Create a TCP socket
   --------------------------- */
s1 = socket(AF_INET, SOCK_STREAM, 0);

....
/* ------------------------------------------------------------
   Bind to a network address so that the server can listen
   for connection requests
   ------------------------------------------------------------ */
if ( bind(s1, (struct sockaddr *) &server, sizeof(server) ) < 0 )
 { perror("bind:");
   exit(1);
 }

/* -----------------------------------------
   Listen for connection requests on s1
   ----------------------------------------- */
listen(s1, 5);

while (1)
{
   len = sizeof(from);

   /* -------------------------------------------------
      Accept connection requests
-------------------------------------------------- */
   if ( (s2 = accept(s1, (struct sockaddr *) &from, &len)) < 0 )
    { perror("accept: ");
      exit(1);
    }

   /* -------------------------------------------------
      You arrive here when the server has accepted
a connection request from one client

Use s2 to communicate with the client
-------------------------------------------------- */

   if ( fork() == 0 )
   {
      /* ---------------------------
   Child process (the clone)
   --------------------------- */
      close(s1);        // Don't need to listen

      /* ---------------------------                  ---+
         Processing loop                                |
   --------------------------- */                      |
      while ( read( s2, buf, .... ) > 0 )             | Change this
               {                              |    t o    d o
            process data in buf                | other tasks
              write(s2, reply, ...);                   |
               }                                - - - +

      close(s2);
 exit(0);
    }
    else
    {
      /* -------------------------------
   Parent process (the original)
   ------------------------------- */
 close(s2);

      // We loop back up and accept the next connection request
    }

}
```

- **Example 1: "null" server**
  - The ***child* process** will ***print*** each character that is **received**:

    (While the **parent process** goes back and **executed** another `accept` call)

```
while (1)
{
    s2 = accept(s1, (struct sockaddr *) &from, &len));
```

```
    /* ---------------------
       fork()
       --------------------- */
    if ( fork() == 0 )
    {
       /* ====================================
          Code executed by the child process
          ==================================== */
       close(s1);                    // Child does not use Listen socket

       /* ========================================
          Use data socket and comm. with client
          ======================================== */
       while (read(s2, &c, 1) > 0)
       {  putchar(c);
          fflush(stdout);
       }

       close(s2);
       printf("\n\n...Done...\n\n");

       exit(0);                                  // Child process exits !!
    }
    else
    {
       close(s2);
    }
 }
```

- **Example Program:** (Demo above code) *Example*
  - The ***parallel* server** Prog file: click here
  - We can use the ***same* client** Prog file: click here

How to run the program:

- **Right click** on link and **save** in a scratch directory
- To compile: `gcc -o tcp1-cli tcp1-cli.c`
  `gcc -o tcp2-serv tcp2-serv.c`
- To run: on machine **X**: `tcp2-serv 8000`
  on machine **Y**: `tcp1-cli X 8000`
  on machine **Z**: `tcp1-cli X 8000`

Both client program can send to the **server** !!!

- **Example 2: addition server**
  - The ***child* process** will ***read* 2 integers** and then **send** the **sum** back to the **client** (until there is no more data):

    (While the **parent process** goes back and **executed** another `accept` call)

```
 while (1)
 {
    s2 = accept(s1, (struct sockaddr *) &from, &len));

    if ( fork() == 0 )
    {
       /* ====================================
          Code executed by the child process
          ==================================== */
       close(s1);                    // Child does not use the Listen socket

       /* ========================================
          Use data socket and comm. with client
          ======================================== */
       while ( 1 )
       {
          int x1, y1, z1;

          if ( read( s2, &x1, 4) <= 0 )
             break;

          x = ntohl(x1);                // Convert to host order

          if ( read( s2, &y1, 4) <= 0 )
             break;

          y = ntohl(y1);                // Convert to host order

          z = x + y;
          printf("Client data: %d, %d, send sum = %d\n", x, y, z);

          z1 = htonl(z);
          write ( s2, &z1, 4 );
       }

       close(s2);
       printf("\n\n...Done...\n\n");
       exit(0);
    }
    else
    {
```

```
            close(s2);
        }
    }
```

- **Example Program:** (Demo above code)

Example

  - The ***parallel*** **addition server** Prog file: click here
  - We can use the ***same*** **client** Prog file: click here

How to run the program:

- **Right click** on link and **save** in a scratch directory
- To compile: `gcc -o tcp1a-cli tcp1a-cli.c`
  `gcc -o tcp2a-serv tcp2a-serv.c`
- To run: on machine **X**: `tcp2a-serv 8000`
  on machine **Y**: `tcp1a-cli X 8000`
  on machine **Z**: `tcp1a-cli X 8000`

Both client program can send to the **server** !!!
