Vanderbilt University

Dept of Computer Science

CS 3281: Operating Systems

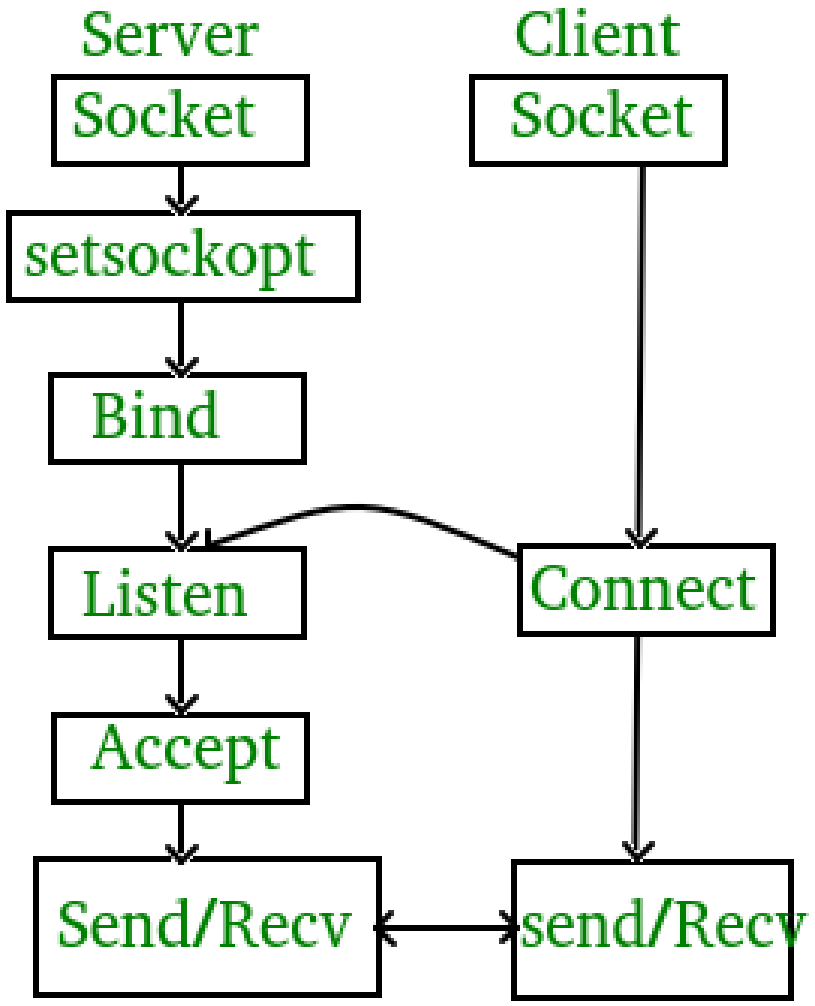
Lecture on Socket Programming

Spring 2024

What is a Socket?

- A socket is an operating system-provided capability for user programs to be able to safely and fairly access the networking hardware resources on the compute machine
- A socket is often represented inside the OS using a file descriptor
- Operations on the socket are then very similar to those of files, e.g., read (or recv) and write (or send) except that instead of reading and writing from a file residing on a local disk, the information is sent and received over a network through a network interface card attached to the computer
- The socket abstraction is generic enough to support a range of protocols, the most famous being the Internet protocols of TCP/IP and UDP/IP
- Specific operations on the socket, such as bind or accept versus connect, determine whether the communicating entity operates as a server or a client

Socket Programming Basics



- Socket programming involves two roles
  - A passive service provider waiting for clients to request a service – a role played by the server
  - An active connection initiator requesting the service – a role played by a client of the service
- These roles are asymmetric
  - Server side uses different set of operations to provide the service than the client who actively initiates a connection

A Note of Endianness (1/2)

- Different computer chipsets maybe one of Little Endian or Big Endian architecture
- In a Little Endian machine, the address of a Word (often 32 or 64 bits long) is the address of the least significant byte of that Word
- In a Big Endian machine, the address of a Word is the address of the most significant byte of that Word
- Thus, when a Little Endian machine naively transfers a data type, such as an integer, to a Big Endian machine, the exact opposite ordering of bytes in memory used by the two architectures will lead to a completely different interpretation of the received value
- Thus, appropriate swapping of bytes must be done before a little endian machine talks to a big endian machine (or vice versa)

A Note of Endianness (2/2)

- The networking standardization committee decided to use the Big Endian byte ordering as the standard representation when information is sent on the network
- Operating Systems, such as Linux and others, support helper functions called “htonl” (host to network long) and “htons” (host to network short) to convert long and short integer data types, respectively, to be converted from host ordering to network ordering
- OS also support helper functions for the reverse direction in the form of “ntohl” and “ntohs”
- Since the network byte ordering is Big Endian, the “htonl”, “htons”, “ntohl” and “ntohs” calls are no operations
- But their use is needed to write application code that is portable

A Note on IP Addresses (1/2)

- Every endpoint that uses the Internet protocols will have an address by which it can be reached by other entities subject to the scope of the address (e.g., private versus public)
- The Internet protocols support two forms of addresses: IPv4, which are 32 bits long, and IPv6, which are 128 bits long
- We will use IPv4 for our class and assignments
- An IPv4 address of 32 bits is usually represented using the Classless Inter Domain Routing (CIDR) notation of a.b.c.d/N
  - where, the N after the forward slash indicates the number of the most significant of those 32 bits used to represent the network subnet, while the remaining lower bits represent the host ID within that subnet
  - Thus, a 192.168.1.5/24 => that the subnet is 192.168.1.0 while the host's ID is 5 within that subnet
- IP addresses can be static (or permanent), which are typically used for publicly reachable, well-known servers while most other IP addresses are dynamically assigned and are recyclable, e.g., when we connect to a WiFi network

A Note on IP Addresses (2/2)

- Commands such as “ifconfig” (on Linux/Mac) and “ipconfig” on Windows will display the IP addresses that are associated with the different network interface cards on that machine
- As programmers and users, we typically use IP addresses of the form www.vanderbilt.edu or something like 192.168.5.20 as they are easier to remember and deal with
- An address like 192.168.5.20 uses what is known as the ***Dotted Decimal Notation***
- However, machines do not understand these human-readable representations and hence must be converted to 32-bit integers before they can operate on them
- OS provides helper functions like inet_addr, inet_aton and a slew of other functions in this regard
- Functions going in the reverse direction are also available

A Note on Port Numbers

- A machine may host many different services
- Thus, each service can be reached by the same IP address of that machine
- How then are we to distinguish one service from the other?
- The OS provides one more level of ***demultiplexing*** using the notion of a Port number (which is a 16 bit integer quantity)
- Some services have dedicated and well-known port numbers
  - E.g., 80 for Web, 443 for secure Web, 21 for FTP, 22 for ssh, 25 for email, etc
- Services can be TCP or UDP-based
- Port numbers from 0 to 1023 are reserved for well-known services
- For user services, we are allowed to use anything from 1024 to 65535
- However, several ports from 1024-49151 are registered ports
- Anything from 49152 to 65535 are truly dynamic and can be used by anyone

Steps in Developing Server Code (Passive Entity)

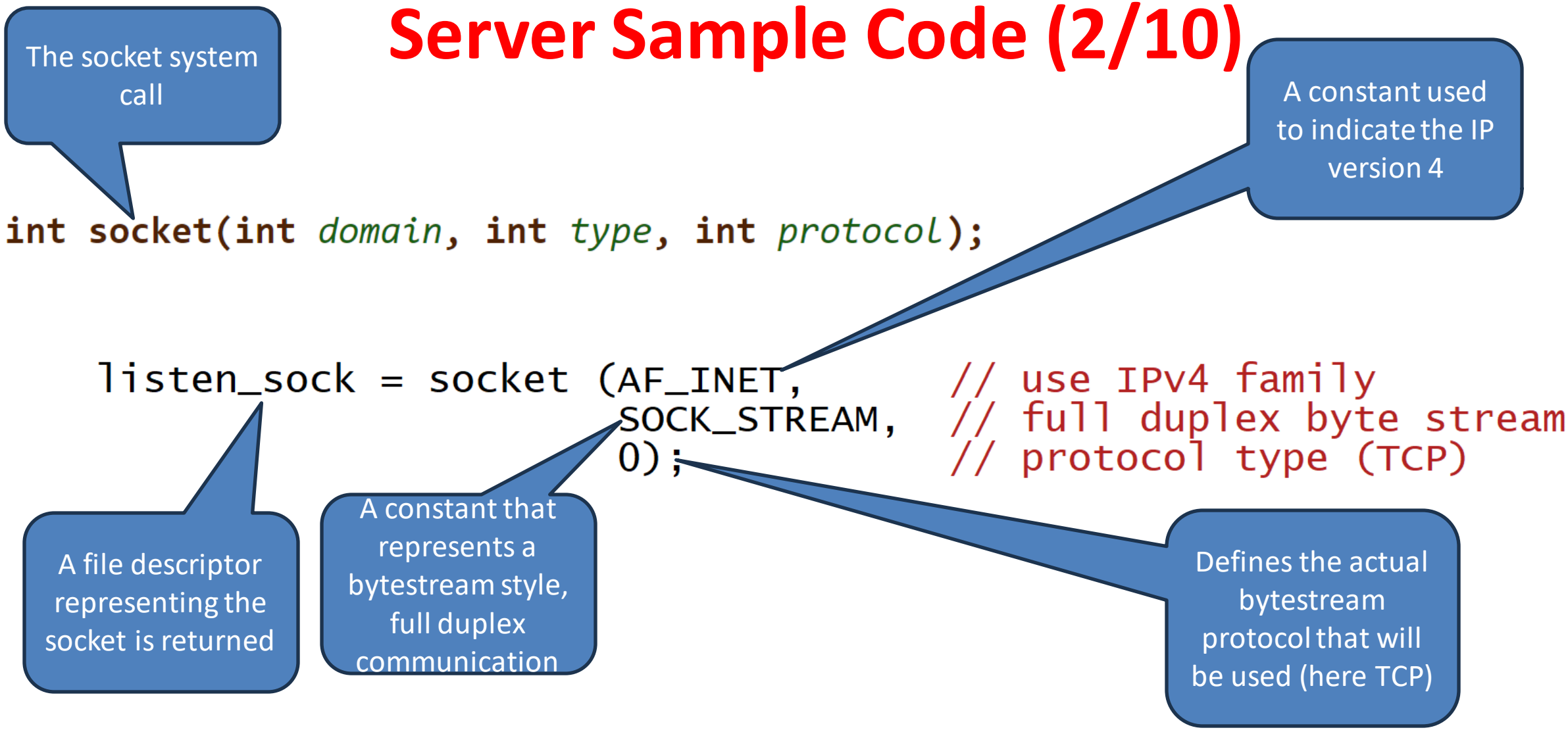
1. Create a socket using “socket” system call
   - Tell the OS what transport and protocol family to use
   - Returns a handle to the socket created by OS
2. Bind an address to this socket using the “bind” system call
   - Tell the OS what network interface to receive requests on, and what port number to associate with the socket
3. Listen for incoming connections using the “listen” system call
   - Allocate resources and get ready to start listening for incoming client requests
4. Accept an incoming client connection via the “accept” system call
   - Block waiting to accept an incoming connection
5. Serve the client on the new socket while continuing to listen on original socket
   - “accept” returns with a new socket handle to be used for I/O with client using “send” and “recv” system calls
   - Server typically continues to listen for new connections on the original socket handle

Server Sample Code (1/10)

```
#include <stdio.h>          // for I/O
#include <string.h>         // basic string API
#include <unistd.h>         // for getpid (), getopt ()
#include <stdlib.h>         // for atoi, etc
#include <sys/types.h>      // for various data types
#include <sys/socket.h>     // for socket API
#include <netinet/in.h>     // for sockaddr_in
#include <arpa/inet.h>      // for inet_addr
```

Typical header files that need to be included

Server Sample Code (2/10)



Care should be taken to ensure that the parameters are compatible with each other. For example, one cannot use UDP for a SOCK_STREAM style protocol.

Server Sample Code (3/10)

The API then requires us to fill this data structure for the Internet protocols

Use exactly the same value used in the socket system call

Populate with the port number on which a server listens. Should be in network order

This is a nested structure. See below.

```
struct sockaddr_in {
  sa_family_t    sin_family; // address family: AF_INET
  in_port_t      sin_port;   // port in network byte ord
  struct in_addr sin_addr;   // internet address
};


where, the Internet address specified in the struct in_addr is of the
following form:

struct in_addr {
  uint32_t       s_addr; // address in network byte order
}
```

Since our server will listen on all interfaces, we populate this with INADDR_ANY

Care should be taken to first zero out this data structure so that there are no garbage values in it, and then to ensure that the family parameter matches the one used in the socket call, and the other fields are in network byte order.

Server Sample Code (4/10)

The next step is to invoke the bind system call

To support a range of protocols, bind uses a generic data structure

```
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

```
bind_status =
    bind (listen_sock,      // this was the listen socket handle we created
                            // earlier
          // the second arg is supposed to be of type "sockaddr *" which is
          // a typedef declared in the OS headers to  represent a generic
          // network family. However, we  are using the IPv4 family and
          // hence had initialized the "sockaddr_in" structure. Therefore,
          // now we must cast the sockaddr_in to sockaddr, which is the
          // parameter used by the bind call.
          //
          // In C language, therefore, we cast it to the right type
          (struct sockaddr *)&server_addr,
          // indicate the size of the structure
          sizeof (struct sockaddr));
```

Use our listening socket that we created earlier

We must now cast the type from the Internet data structure to the generic data structure

The number of bytes

Care should be taken in using the cast operator and number of bytes.

Note that the structure should have information in network byte order

Server Sample Code (5/10)

The next step is to invoke the listen system call

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

```
listen_status = listen (listen_sock,
                        5);
```

Our listening socket where we expect an incoming connection

Size of queue of pending/outstanding connections

Server Sample Code (6/10)

The next step is to invoke the accept system call

```
int accept(int sockfd, struct sockaddr *_Nullable restrict addr,
           socklen_t *_Nullable restrict addrlen);
```

- The accept system call is a factory method (remember the Factory Method pattern from CS 3251)
- It creates a new socket, which is to be used for communication with the client (and hence it is a factory method)
- Details of the client can be obtained in the second parameter
- Since a server is meant to be long running and should accept and serve multiple clients, such an accept system call is often made inside a ***forever*** loop
- A server can be architected to be iterative, i.e., serve one client at a time, or concurrent, i.e., serve multiple clients concurrently - - but this will need either multiple threads or multiple processes using fork system call

Server Sample Code (7/10)

The forever loop

```
for (;;) {
    printf ("Server: WAITING TO ACCEPT A NEW CONNECTION\n");

    // the accept command shown below actually does the job
    // the TCP/IP 3-way handshaking protocol when a client
    // requests a connection establishment.
    //
    // Note that the accept command creates a new socket handle as the
    // return value of "accept". Understand that this is necessary because
    // in order to serve several clients simultaneously, the server needs
    // to distinguish between the handle it uses to listen for new
    // connections requests and the handle it uses to exchange data with
    // the client. Thus, the newly created socket handle is used to do the
    // network I/O with client whereas the older socket handle  continues
    // to be used for listening for new connections.
    conn_sock = accept (listen_sock, // our l      handle
                        0,   // we don't      s of client
                        0); // hence leng
```

Our listening socket

The newly produced socket for data communication

In this sample code, we are not interested in getting the details of the client

Server Sample Code (8/10)

Variety of recv system calls

```
ssize_t recv(int sockfd, void buf[.len], size_t len,
             int flags);
ssize_t recvfrom(int sockfd, void buf[restrict .len], size_t len,
             int flags,
             struct sockaddr *_Nullable restrict src_addr,
             socklen_t *_Nullable restrict addrlen);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

Receive data on the newly created socket

Receive it in a buffer for which memory is allocated

```
// block until something is received
int recv_status
    = recv (conn_sock,   // client I/O to be done with new socket
                         // second parameter is a buffer into which you
                         // receive data. Linux needs a
(void *)data_buff,
// third parameter is the size of the buff
sizeof (data_buff),
// last parameter is a flag that we
// will ignore.
0);
```

Indicate how many bytes maximum to receive

Various flags to control the behavior

Care should be taken to receive the data on the new data socket and not on the listening socket. Also, the receiving buffer should have enough memory preallocated

Server Sample Code (9/10)

Variety of send system calls

```
ssize_t send(int sockfd, const void buf[.len], size_t len, int flags);
ssize_t sendto(int sockfd, const void buf[.len], size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Server Sample Code (9/)

```
int send_status =
    send (conn_sock, // I/O is done on thi
          // second parameter is the buffer you send
          (void *) data_buff,
          // 3rd param is the length of the b
          sizeof (data_buff),
          // we ignore the flags argument
          0);
```

send reply to the client on the newly created socket

Send the buffer that has the reply to send

Indicate how many bytes maximum to send

Various flags to control the behavior

Care should be taken to send the data on the new data socket and not on the listening socket.

Server Sample Code (10/10)

```
int close(int fd);
```

Closing the socket using the close system call. Usually initiated by the client

```
(void) close (conn_sock);
```

Close our connection socket when closing connection with the established connection with the client

```
(void) close (listen_sock);
```

Close the listening socket when wanting to stop the server entirely

Server Implementation Choices

- Iterative Server
  - Our sample code illustrates an iterative server because we accept one connection and serve that client before we accept the next connection
  - The logic is very simple
  - However, such a design cannot scale to serve multiple clients and offer them good response time
- Concurrent Server
  - An alternate choice is to decouple the accepting of incoming connection request from data communication with the client
  - We can achieve this through multiple different models
    - Process-per-connection, where we fork a child process to perform the data communication with the client while the parent process continue to accept new connections
    - Thread-per-connection, where we spawn a new thread to perform the data communication while the parent thread continues to accept new connections
    - Thread-pool, where we create a pre-allocated set of threads and pick an idle thread from the pool handing it the data communication responsibility while the parent continues to accept connections. The data communication thread returns to the pool after it is done with the client. Such a design requires inter thread communication using some IPC approach

Steps in Developing Client Code (Active Entity)

1. Create a socket using “socket” call
   - Tell the OS what transport and protocol family to use
   - Returns a handle to the socket created by OS
2. Connect to the server
   - Indicate the IP address and Port number of server to connect to
   - Associate the socket with this information
   - Ask the OS to connect to the server and use this socket as the channel for communication.
3. Initiate I/O with server
   - After “connect” returns successfully, continue with the application-specific information exchange with the server
4. Close the connection
   - When information exchange is over, close the connection with the server via the “close” function.

Client Sample Code (1/7)

```
#include <stdio.h>          // for I/O
#include <string.h>         // basic string API
#include <unistd.h>         // for getpid (), getopt ()
#include <stdlib.h>         // for atoi, etc
#include <sys/types.h>      // for various data types
#include <sys/socket.h>     // for socket API
#include <netinet/in.h>     // for sockaddr_in
#include <arpa/inet.h>      // for inet_addr
```

Typical header files that need to be included. Very similar to the server code.

Client Sample Code (2/7)

The socket system call

A constant used to indicate the IP version 4

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

```
conn_sock = socket (AF_INET,      // use IPv4 family
                    SOCK_STREAM,  // full duplex byte stream
                    0);           // protocol type (TCP)
```

A file descriptor representing the socket is returned

A constant that represents a bytestream style, full duplex communication

Defines the actual bytestream protocol that will be used (here TCP)

Socket creation is very similar to that of the server. Care should be taken to ensure that the parameters are compatible with each other. For example, one cannot use UDP for a SOCK_STREAM style protocol. Moreover, the communicating entities must use the same protocols to talk to each other.

Client Sample Code (3/7)

The API then requires us to fill this data structure for the Internet protocols

Use exactly the same value used in the socket system call

Populate with the port number on which a server listens. Should be in network order

This is a nested structure. See below.

```
struct sockaddr_in {
  sa_family_t    sin_family; // address family: AF_INET
  in_port_t      sin_port;   // port in network byte ord
  struct in_addr sin_addr;   // internet address
};


where, the Internet address specified in the struct in_addr is of the
following form:

struct in_addr {
  uint32_t       s_addr; // address in network byte order
}
```

Populate with the 32 bit integer corresponding to the dotted decimal IP address of the server in network byte order

This code is similar to that of the server. Care should be taken to first zero out this data structure so that there are no garbage values in it, and then to ensure that the family parameter matches the one used in the socket call, and the other fields are in network byte order.

Client Sample Code (4/7)

The next step is to invoke the connect system call

To support a range of protocols, connect uses a generic data structure

```
int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

```
int conn_status;
conn_status =
    connect (conn_sock,      // this was the connect
                             // created earlier
             // the second arg is supposed to be of type "sockaddr *" which
             // is a typedef declared in the OS headers to  represent a
             // generic network family. However, we  are using the IPv4
             // family and hence had initialized the "sockaddr_in"
             // structure. Therefore, now we must cast the sockaddr_in to
             // sockaddr
             (struct sockaddr *)&server_addr,
             // indicate the size of the structure
             sizeof (struct sockaddr));
```

Use our connecting socket that we created earlier

We must now cast the type from the Internet data structure to the generic data structure

The number of bytes

Care should be taken in using the cast operator and number of bytes.

Note that the structure should have information in network byte order

Client Sample Code (5/7)

Variety of send system calls

```
ssize_t send(int sockfd, const void buf[.len], size_t len, int flags);
ssize_t sendto(int sockfd, const void buf[.len], size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Server Sample Code (9/)

```
int send_status =
    send (conn_sock, // I/O is done on thi
          // second parameter is the buffer you send
          (void *) send_buff,
          // 3rd param is the length of the bu
          sizeof (send_buff),
          // we ignore the flags argument
          0);
```

send request to the server using our socket

Send the buffer that has the request to send

Indicate how many bytes maximum to send

Various flags to control the behavior

Client Sample Code (6/7)

Variety of recv system calls

```
ssize_t recv(int sockfd, void buf[.len], size_t len,
             int flags);
ssize_t recvfrom(int sockfd, void buf[restrict .len], size_t len,
             int flags,
             struct sockaddr *_Nullable restrict src_addr,
             socklen_t *_Nullable restrict addrlen);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

Receive data on the socket we created

Receive it in a buffer for which memory is allocated

```
int recv_status = recv (conn_sock,  // I/O done wi  new sock handle
                        // second parameter is a buffer into which you
                        // receive data
                        //
                        // Linux needs a (void *) buffer
                        (void *) rcv_buff,
                        // third parameter is the size o
                        sizeof (rcv_buff),
                        // last parameter is a flag that
                        0);
```

Indicate how many bytes maximum to receive

Various flags to control the behavior

Care should be taken to receive the data on a buffer that has enough memory pre-allocated

Client Sample Code (7/7)

Closing the socket using the close system call. Usually initiated by the client

```
int close(int fd);
```

Close our connection socket

```
(void) close (conn_sock);
```

Demo Screeshots (Start Server First)

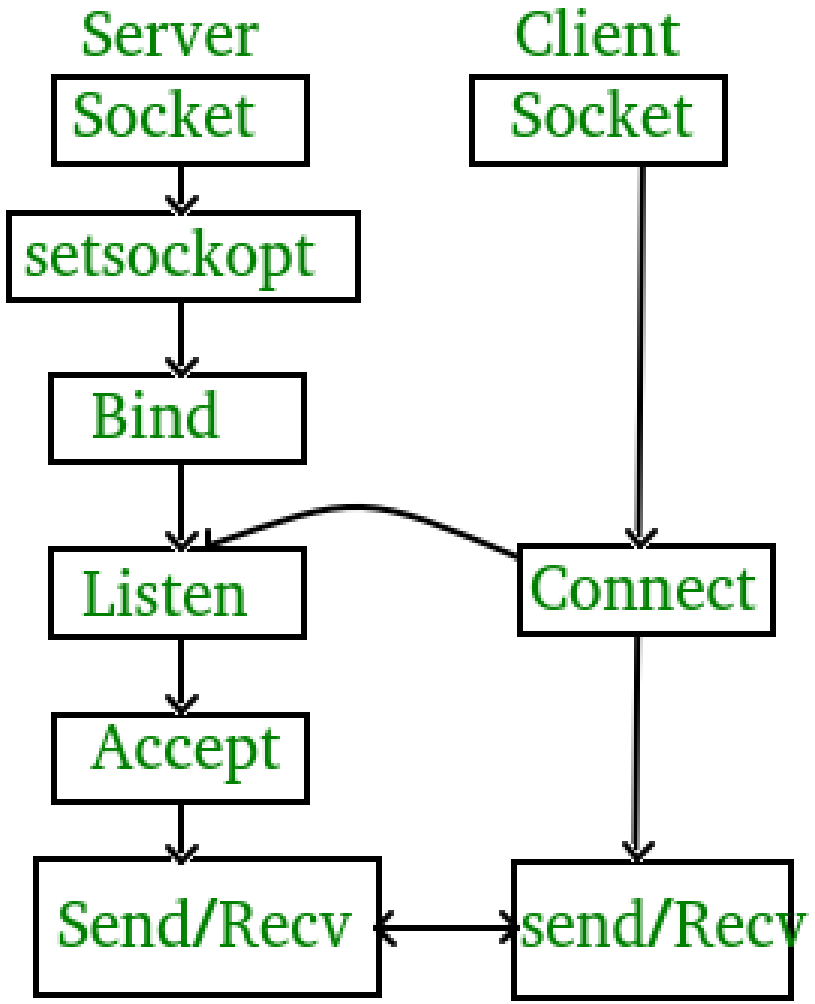
```
gokhale@coursevm:~$ cd Documents/Socket_Code/
gokhale@coursevm:~/Documents/Socket_Code$ ./server -h
Usage: ./server [-h] [-p <port num>]
gokhale@coursevm:~/Documents/Socket_Code$ ./server -p 30000
Server listening on port num: 30000
Server: WAITING TO ACCEPT A NEW CONNECTION
Server: ACCEPTED A NEW CONNECTION
Server: Received data: "This is client"
Server: WAITING TO ACCEPT A NEW CONNECTION
Server: ACCEPTED A NEW CONNECTION
Server: Received data: exit
Server is EXITING
```

- Here we are making the server listen on port 30,000 instead of its default port of 10,000
- Notice how the server exits after it receives an “exit” command from the client
  - This is how we have coded the logic

- Since our server and client run on the same machine, our client does not supply an IP address as it is by default 127.0.0.1
- However, the client uses the non default port used by the server

```
gokhale@coursevm:~/Documents/Socket_Code$ ./client -h
Usage: ./client [-h] [-a <IP addr>] [-p <port num>]
gokhale@coursevm:~/Documents/Socket_Code$
gokhale@coursevm:~/Documents/Socket_Code$ ./client -p 30000
Client is connecting to server at IP addr: 127.0.0.1 and port num: 30000
Client: ESTABLISHED A CONNECTION
Enter some data to send to server (type exit to kill server): "This is client"
Client received: "This is client"
gokhale@coursevm:~/Documents/Socket_Code$
gokhale@coursevm:~/Documents/Socket_Code$ ./client -p 30000
Client is connecting to server at IP addr: 127.0.0.1 and port num: 30000
Client: ESTABLISHED A CONNECTION
Enter some data to send to server (type exit to kill server): exit
Client received: exit
```

Perils of Using Low-level Sockets



- The asymmetric structure is one cause for confusion
- Second, each step shown is a low-level system call that takes in various arguments
  - When used in C/C++ languages, these parameters are pointers to structures that need to be cast from one type to another
  - Other parameters should be provided in such a way that they are all consistent with each other
  - The send-receive loop has to be carefully crafted
  - When used in the context of multi threading, this task becomes even harder
- Third, writing code that is portable across OS is yet another problem area
- All of this leads to very high possibility of committing errors that are hard to debug

See scaffolding code to convince yourself as to how hard it is to write low-level socket code

Emergence of Higher-level Frameworks

- To address these problems, several different middleware frameworks have emerged
- The ACE framework (developed by Prof. Doug Schmidt) is a widely used C++ framework with number of design patterns
- Languages like Java provide portable code
- CORBA like frameworks provide language-agnostic, object-oriented style distributed remote procedure call capabilities along with its own serialization capabilities
- More recently, frameworks like ZeroMQ and gRPC have become more widely used to write networking/distributed systems code using more intuitive and higher-level abstractions
  - These frameworks also support bindings to multiple different programming languages
  - They can leverage different serialization frameworks