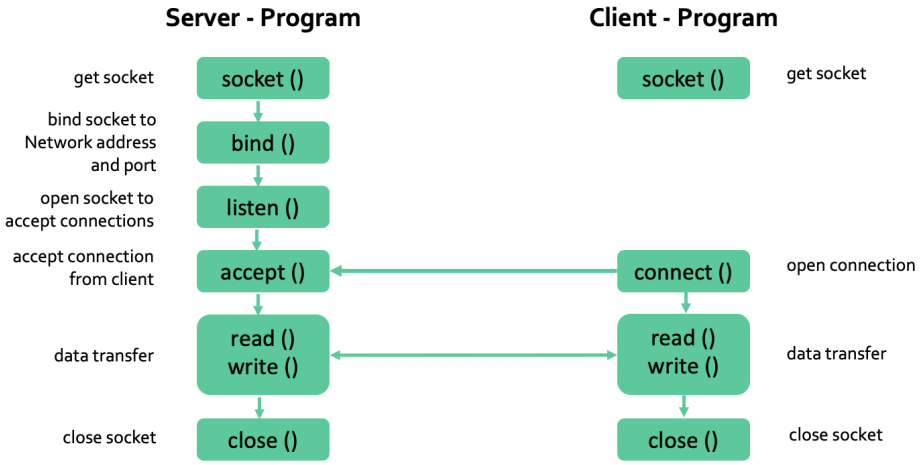
COEN 146: Computer Networks
Lab 3: TCP/IP Socket Programming

Objectives

1. To develop client/ server applications using TCP/IP Sockets
2. To write a C program to transfer file over TCP/IP Socket

TCP/IP Client/ Server[1]

The following figure shows the steps taken by each program:



On the client and server sides:

The `socket()` system call creates an *unbound* **socket** in a communications domain, and return a file descriptor that can be used in later **function** calls that operate on **sockets**.

```
int sockfd = socket(domain, type, protocol)
```

- **sockfd:** socket descriptor, an integer (like a file-handle)
- **domain:** integer, communication domain e.g., AF_INET (IPv4 protocol) , AF_INET6 (IPv6 protocol), AF_UNIX (local channel, similar to pipes)
- **type:** communication type
  SOCK_STREAM: TCP (reliable, connection oriented)
  SOCK_DGRAM: UDP (unreliable, connectionless)
  SOCK_RAW (direct IP service)
- **protocol:** This is useful in cases where some families may have more than one protocol to support a given type of service. Protocol value for Internet Protocol (IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.

```
#include <sys/socket.h>
...
...if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("cannot create socket");
    return 0;
}
```

On the server side:

[1] http://beej.us/guide/bgnet/

After creation of the socket, **`bind()`** system call binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR_ANY to specify the IP address.

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

- **addr***:* Points to a **sockaddr** structure containing the address to be bound to the socket. The length and format of the address depend on the address family of the socket.
- **addrlen***:* Specifies the length of the **sockaddr** structure pointed to by the *addr* argument.

The `listen()` function puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection.

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

- **backlog**: defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

The `accept()` system call extracts the first connection request on the queue of pending connections for the listening socket (sockfd), creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

```
int new_socket= accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

On the client side:

The `connect()` system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

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

Read and write over socket:

```
bzero(buffer, 256);
n = read(newsockfd, buffer, 255);
if (n < 0) error("ERROR reading from socket");
printf("Here is the message: %s\n", buffer);
```

This code initializes the buffer using the `bzero()` function, and then reads from the socket. **Note that the read call uses the new file descriptor, the one returned by `accept()`, not the original file descriptor returned by socket().**

Note also that the `read()` will block until there is something for it to read in the socket, i.e. after the client has executed a `write()`. It will read either the total number of characters in the socket or 255, whichever is less, and return the number of characters read

```
n = write(newsockfd, "I got your message", 18);
if (n < 0) error("ERROR writing to socket");
```

Once a connection has been established, both ends can both read and write to the connection. Naturally, everything written by the client will be read by the server, and everything written by the server will be read by the client. This code simply writes a short message to the client. The last argument of write is the size of the message.

Structures:

Address format
An IP socket address is defined as a combination of an IP interface address and a 16-bit port number. The basic IP protocol does not supply port numbers, they are implemented by higher level protocols like UDP and TCP. On raw sockets sin_port is set to the IP protocol.

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

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

This is defined in `netinet/in.h`

`sin_family` is always set to AF_INET.

`sin_port` contains the port in network byte order. The port numbers below 1024 are called privileged ports (or sometimes: reserved ports). Only privileged processes) may bind to these sockets.

`sin_addr` is the IP host address.

`s_addr` member of struct `in_addr` contains the host interface address in network byte order. `in_addr` should be assigned one of the INADDR_* values (e.g., INADDR_ANY) or set using the `inet_aton, inet_addr, inet_makeaddr` library functions or directly with the name resolver (see `gethostbyname`).

INADDR_ANY allows your program to work without knowing the IP address of the machine it was running on, or, in the case of a machine with multiple network interfaces, it allowed your server to receive packets destined to any of the interfaces.

INADDR_ANY has the following semantics: When receiving, a socket bound to this address receives packets from all interfaces. For example, suppose that a host has interfaces 0, 1 and 2. If a UDP socket on this host is bound using INADDR_ANY and udp port 8000, then the socket will receive all packets for port 8000 that arrive on interfaces 0, 1, or 2. If a second socket attempts to Bind to port 8000 on interface 1, the Bind will fail since the first socket already “owns” that port/interface.

Example:

```
serv_addr.sin_addr.s_addr = htonl (INADDR_ANY);
```

- Note: "Network byte order" always means big endian. "Host byte order" depends on architecture of host. Depending on CPU, host byte order may be little endian, big endian or something else.
- The `htonl()` function translates a long integer from host byte order to network byte order.

To **bind** socket with *localhost*, before you invoke the *bind* function, `sin_addr.s_addr` field of the `sockaddr_in` structure should be set properly. The proper value can be obtained either by

```
my_sockaddress.sin_addr.s_addr = inet_addr("127.0.0.1")
```

or by

```
my_sockaddress.sin_addr.s_addr=htonl(INADDR_LOOPBACK);
```

To convert an address in its standard text format into its numeric binary form use the `inet_pton()` function. The argument `af` specifies the family of the address.

```
#define _OPEN_SYS_SOCK_IPV6
#include <arpa/inet.h>

int inet_pton(int af, const char *src, void *dst);
```

Recap - File transfer:

- Binary file: jpg, png, bmp, tiff etc.
- Text file: txt, html, xml, css, json etc.

You may use functions or system calls for file transfer. C Function connects the C code to file using I/O stream, while system call connects C code to file using file descriptor.

- File descriptor is integer that uniquely identifies an open file of the process.
- I/O stream sequence of bytes of data.

A Stream provides high level interface, while File descriptor provide a low-level interface. Streams are represented as FILE * object, while File descriptors are represented as objects of type int.

C Functions to open and close a binary/text file

fopen(): C Functions to open a binary/text file, defined as:

```
FILE *fopen(const char *file_name, const char *mode_of_operation);
```

where:

- file_name: file to open
- mode_of_operation: refers to the mode of the file access, For example:- r: read , w: write , a: append etc
- fopen() return a pointer to FILE if success, else NULL is returned
- fclose(): C Functions to close a binary/text file.

fclose(): C Functions to close a binary/text file, defined as:

```
fclose( FILE *file_name);
```

Where:

- file_name: file to close
- fclose () function returns zero on success, or EOF if there is an error

C Functions to read and write a binary file

fread(): C function to read binary file, defined as:

```
fread(void * ptr, size_t size, size_t count, FILE * stream);
```

where:

- ptr- it specifies the pointer to the block of memory with a size of at least (size*count) bytes to store the objects.
- size - it specifies the size of each objects in bytes.
- count: it specifies the number of elements, each one with a size of size bytes.
- stream - This is the pointer to a FILE object that specifies an input stream.
- Returns the number of items read

fwrite (): C function to write binary file, defined as:

```
fwrite (void *ptr, size_t size, size_t count, FILE *stream);
```

where:

- Returns number of items written
- *arguments of fwrite are similar to fread. Only difference is of read and write.

For example:
To open "lab3.dat" file in read mode then function would be:

```
FILE* demo; // demo is a pointer of type FILE
```

```
char buffer[100]; // block of memory (ptr)
demo= fopen("lab3.dat", "r"); // open lab3.dat in read mode
fread(&buffer, sizeof(buffer), 1, demo); // read 1 element of size = size of buffer (100)
fclose(demo); // close the file
```

C Functions to read and write the text file.

fscanf (): C function to read text file.

```
fscanf(FILE *ptr, const char *format, ...)
```

Where:

- Reads formatted input from the stream.
- Ptr: File from which data is read.
- format: format of data read.
- returns the number of input items successfully matched and assigned, zero if failure

fprintf(): C function to write a text file.

```
fprintf(FILE *ptr, const char *format, ...);
```

Where:

- *arguments similar to fscanf ()

For example:

```
FILE *demo; // demo is a pointer of type FILE
demo= FILE *fopen("lab3.dat", "r"); // open lab3.dat in read mode
/* Assuming that lab3.dat has content in below format
City
Population
....
*/
char buf[100]; // block of memory
fscanf(demo, "%s", buf); // to read a text file
fclose(demo); // close the file
*to read whole file use while loop
```

System Call to open, close, read and write a text/binary file.

open(): System call to open a binary/text file, defined as:

```
open (const char* Path, int flags [, int mode ]);
```

Where:

- returns file descriptor used on success and -1 upon failure
- Path :- path to file
- flags :- O_RDONLY: read only, O_WRONLY: write only, O_RDWR: read and write, O_CREAT: create file if it doesn't exist, O_EXCL: prevent creation if it already exists

close(): System call to close a binary/text file, defined as:

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

where:

- return 0 on success and -1 on error.
- fd : file descriptor which uniquely identifies an open file of the process

read(): System call to read a binary/text file.

```
read (int fd, void* buf, size_t len);
```

where:

- returns 0 on reaching end of file, -1 on error or on signal interrupt
- fd: file descriptor
- buf: buffer to read data from
- len: length of buffer

write(): System call to write a binary/text file.

```
write (int fd, void* buf, size_t len);
```

where:

- *arguments and return of write are similar to read().

For example:

```
int fd = open("lab3.dat", O_RDONLY | O_CREAT); //if file not in directory, file is
                                                                                created
Close(fd);
```

Implementation steps:

Step 1. [30%] Write a C program for a TCP server that accepts a client connection for file transfer.

Step 2. [25%] Write a C program for a TCP client that connects to the server. In this case
   a. The client connects to the server and request a file to download from the server.
   b. The server accepts the connection and transfers the file to the client

Step 3. Compile and run. Note: you may use the IP address 127.0.0.1 (loop back IP address) for a local host, i.e. both of your client and server run on the same machine.

[20%] Demonstrate your program to the TA:
   a. Your client and server on your same machine
   b. Your client and your classmate's server IP address. You may to discuss with the TA if you run into access problems

Multiple Clients – Concurrent Server

In general, a TCP server is designed as a concurrent server to server multiple clients. This means when a client sends a request for a file transfer, the sever accepts the connection request and spawns a thread to handle this transfer on the connection descriptor. The server will then continue in a loop listening for another client connection request to handle another file transfer.

Step 4. [20%] Write a C program for a concurrent TCP server that accepts and responds to multiple client connection requests, each requesting a file transfer. Modify your TCP server in Step 1 so that when the server accepts a connection from a client it spawns a separate thread to handle this specific client connection for file transfer.

Note: You will have several threads (at the same time) running on the server transferring copies of src.dat files to clients that each will save at their destination as – dst.dat file (possibly needs to be numbered differently on the same host).

[5%] Demonstrate to the TA, multiple clients making file transfer request to the server and that the server makes multiple transfers at the same time. Make N = 5. Upload your source code to Camino.

Note: To be able to see 5 threads handling connections at the same time, you may need to introduce a delay of a few second in the process of data transfer to make it visible. This is due to the fact completing thread file transfer takes a fraction of a millisecond if not a microsecond.

Requirements to complete the lab

1. Demo to the TA correct execution of your programs [recall: a successful demo is 25% of the grade]
2. Submit the source code of your program as .c files on Camino

Please start each program with a descriptive block that includes minimally the following information:

```
/*
* Name: <your name>
* Date: <date> (the day you have lab)
* Title: Lab3 - Part ….
* Description: This program  … <you should
* complete an appropriate description here.>
*/
```