Lab Project 2: **File Transfer over Socket**

- **Number of members per group: 1 or 2**
- **Use C language**

This project consists of a client program and a server program that communicate to transfer one file from the client to the server using TCP.

The client accepts four arguments: server's IP address, server's port number, source file name, and destination file name.
The server starts first and receives one argument: the port number.

The server waits for a connection request (`listen()` and `accept()`). The client requests a connection (`connect()`) and then sends the name of file <output> (the output file) to the server. To simplify, the name of the output file should have at most 9 characters. The server receives the name of the file, sends back to the client the message "received!", opens the file, and waits for data.

The client receives the message "received!" and opens file <input> (input file). Then it reads file <input> and sends the data in chunks of 10 bytes to the server. The server writes the received bytes to file <output>.

After sending the file, the client closes the connection and exits. The server closes the connection when no more data is available.

After executing, <input> and <output> must contain similar content. Your program should be built on TCP. You must use C and Linux. You will need to use the socket library. The man pages in the Unix/Linux systems have a lot of useful information. Start with `man socket`. There is a man page for each function in the socket library.

To handle the files, you must use functions `fread()` and `fwrite()` only, since your program should be able to transfer binary files as well as text files.

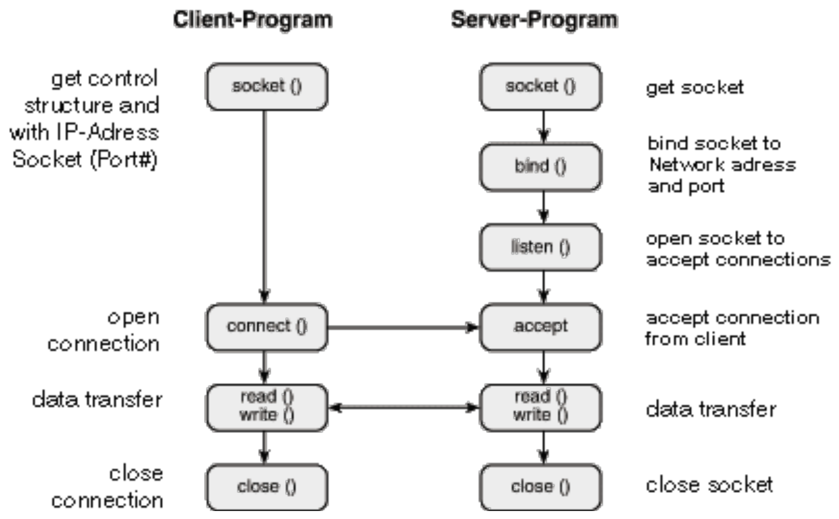
Deliverables:

- **Demo your project to the TA in the lab (using two terminal windows and two computers)**
- **Submit your code to Camino**

Additional Notes:

This project consists of two programs: client, and server.

The following figure shows the steps taken by each program:



On the client and server sides:

The `socket()` **function** 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:

After creation of the socket, **`bind()`** function 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()` function 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. The `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_ANYand 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);
```