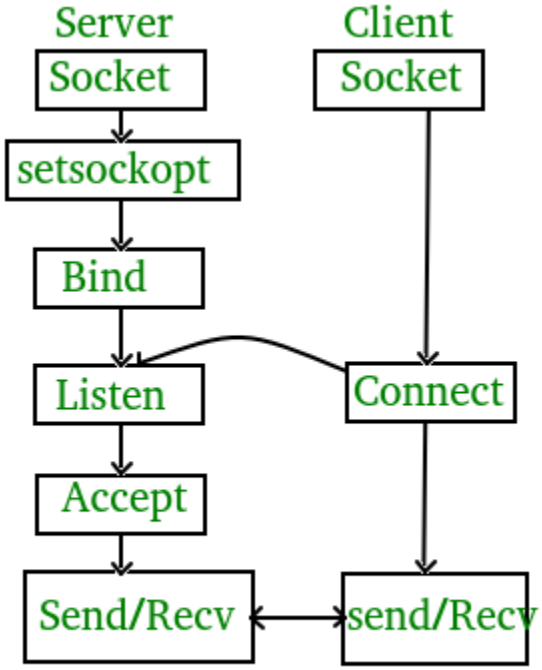
Socket Programming

What is socket programming?
Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

State diagram for server and client model



Stages for server

- **Socket creation:**

int sockfd = socket(domain, type, protocol)

sockfd: socket descriptor, an integer (like a file-handle)
domain: integer, specifies communication domain. We use AF_ LOCAL as defined in the POSIX standard for communication between processes on the same host. For communicating between processes on different hosts connected by IPV4, we use AF_INET and AF_I NET 6 for processes connected by IPV6.
type: communication type
SOCK_STREAM: TCP(reliable, connection oriented)
SOCK_DGRAM: UDP(unreliable, connectionless)
protocol: 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.(man protocols for more details)

- **Setsockopt:**
- int setsockopt(int sockfd, int level, int optname,

  const void *optval, socklen_t optlen);

This helps in manipulating options for the socket referred by the file descriptor sockfd. This is completely optional, but it helps in reuse of address and port. Prevents error such as: “address already in use”.

- **Bind:**
- int bind(int sockfd, const struct sockaddr *addr,

  socklen_t addrlen);

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.

- **Listen:**

int listen(int sockfd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The 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.

- **Accept:**

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

It 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.

Stages for Client

- **Socket connection:** Exactly same as that of server’s socket creation
- **Connect:**
- int connect(int sockfd, const struct sockaddr *addr,

socklen_t addrlen);

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.

Implementation

Here we are exchanging one hello message between server and client to demonstrate the client/server model.

Server.c

```
// Server side C/C++ program to demonstrate Socket programming
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#define PORT 8080
int main(int argc, char const *argv[])
{
	int server_fd, new_socket, valread;
	struct sockaddr_in address;
	int opt = 1;
	int addrlen = sizeof(address);
	char buffer[1024] = {0};
	char *hello = "Hello from server";

	// Creating socket file descriptor
	if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
	{
		perror("socket failed");
		exit(EXIT_FAILURE);
	}
```

```
	// Forcefully attaching socket to the port 8080
	if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
                                                  &opt,
sizeof(opt)))
	{
		perror("setsockopt");
		exit(EXIT_FAILURE);
	}
	address.sin_family = AF_INET;
	address.sin_addr.s_addr = INADDR_ANY;
	address.sin_port = htons( PORT );

	// Forcefully attaching socket to the port 8080
	if (bind(server_fd, (struct sockaddr *)&address,
                                                      sizeof(address))<0)
	{
		perror("bind failed");
		exit(EXIT_FAILURE);
	}
	if (listen(server_fd, 3) < 0)
	{
		perror("listen");
		exit(EXIT_FAILURE);
	}
	if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                                        (socklen_t*)&addrlen))<0)
	{
		perror("accept");
		exit(EXIT_FAILURE);
```

```
        }
        valread = read( new_socket , buffer, 1024);
        printf("%s\n",buffer );
        send(new_socket , hello , strlen(hello) , 0 );
        printf("Hello message sent\n");
        return 0;
}
```

Client.c

```
// Client side C/C++ program to demonstrate Socket programming
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
        int sock = 0, valread;
        struct sockaddr_in serv_addr;
        char *hello = "Hello from client";
        char buffer[1024] = {0};
        if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        {
                printf("\n Socket creation error \n");
                return -1;
        }

        serv_addr.sin_family = AF_INET;
        serv_addr.sin_port = htons(PORT);
```

```
	// Convert IPv4 and IPv6 addresses from text to binary form
	if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
	{
		printf("\nInvalid address/ Address not supported \n");
		return -1;
	}

	if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
	{
		printf("\nConnection Failed \n");
		return -1;
	}
	send(sock , hello , strlen(hello) , 0 );
	printf("Hello message sent\n");
	valread = read( sock , buffer, 1024);
	printf("%s\n",buffer );
	return 0;
}
```

Output:

Client:Hello message sent

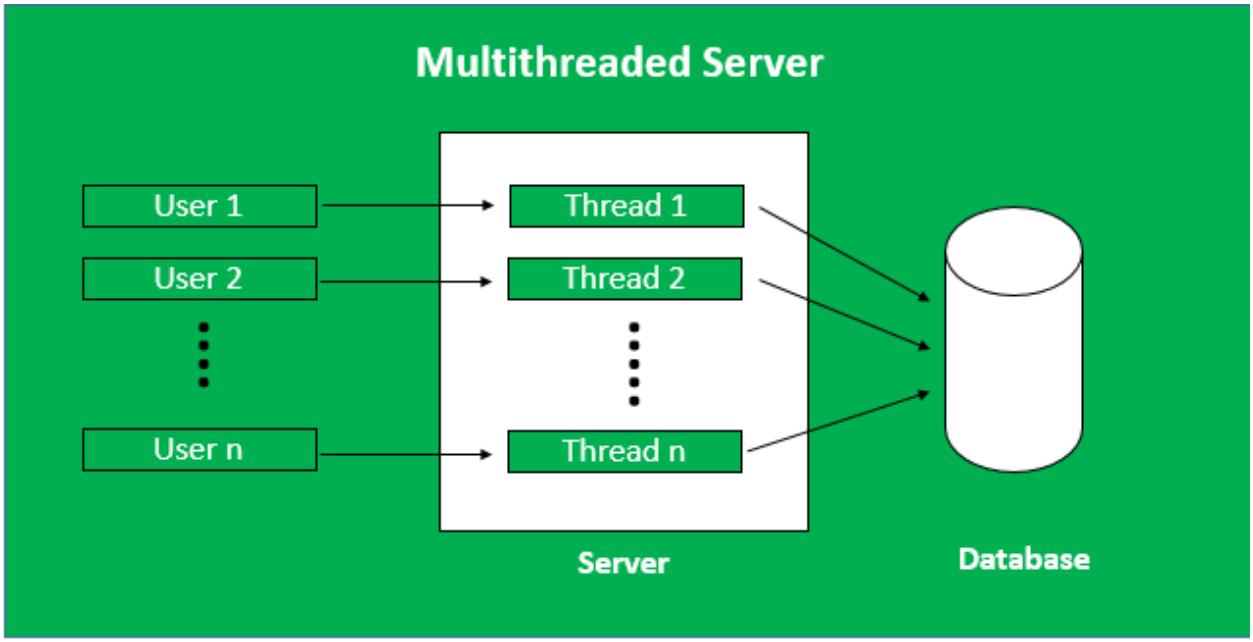
Hello from server

Server:Hello from client

Hello message sent

Multithreaded Server:

A server having more than one thread is known as Multithreaded Server. When a client sends the request, a thread is generated through which a user can communicate with the server. We need to generate multiple threads to accept multiple requests from multiple clients at the same time.



Advantages of Multithreaded Server:

Quick and Efficient: Multithreaded server could respond efficiently and quickly to the increasing client queries quickly.

Waiting time for users decreases: In a single-threaded server, other users had to wait until the running process gets completed but in multithreaded servers, all users can get a response at a single time so no user has to wait for other processes to finish.

Threads are independent of each other: There is no relation between any two threads. When a client is connected a new thread is generated every time.

The issue in one thread does not affect other threads: If any error occurs in any of the threads then no other thread is disturbed, all other processes keep running normally. In a single-threaded server, every other client had to wait if any problem occurs in the thread.

Disadvantages of Multithreaded Server:

Complicated Code: It is difficult to write the code of the multithreaded server. These programs can not be created easily

Debugging is difficult: Analyzing the main reason and origin of the error is difficult.

Server-Side Program: When a new client is connected, and he sends the message to the server.

1. Server class: The steps involved on the server side are similar to the article Socket Programming in Java with a slight change to create the thread object after obtaining the streams and port number.

- **Establishing the Connection:** Server socket object is initialized and inside a while loop a socket object continuously accepts an incoming connection.
- **Obtaining the Streams:** The inputstream object and outputstream object is extracted from the current requests' socket object.
- **Creating a handler object:** After obtaining the streams and port number, a new clientHandler object (the above class) is created with these parameters.
- **Invoking the start() method:** The start() method is invoked on this newly created thread object.

2. ClientHandler class: As we will be using separate threads for each request, let's understand the working and implementation of the **ClientHandler** class implementing Runnable. An object of this class acts as a Runnable target for a new thread.

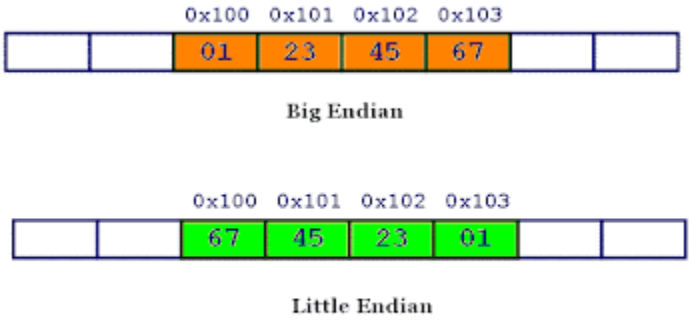
- First, this class implements Runnable interface so that it can be passed as a Runnable target while creating a new Thread.
- Secondly, the constructor of this class takes a parameter, which can uniquely identify any incoming request, i.e. a **Socket**.
- Inside the **run()** method of this class, it reads the client's message and replies.

Little and Big Endian Mystery

What are these?

Little and big endian are two ways of storing multibyte data-types (int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

Suppose integer is stored as 4 bytes (For those who are using DOS-based compilers such as C++ 3.0, integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.



C++

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
	unsigned int i = 1;
	char *c = (char*)&i;
	if (*c)
		cout<<"Little endian";
	else
		cout<<"Big endian";
	return 0;
}
```

C

```
#include <stdio.h>
int main()
{
```

```
unsigned int i = 1;
char *c = (char*)&i;
if (*c)
        printf("Little endian");
else
        printf("Big endian");
getchar();
return 0;
}
```