

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**18CSC302J-COMPUTER NETWORKS**

SEMESTER -5

BATCH-2

|  |  |
| --- | --- |
| **REGISTRATION NUMBER.** | **RA1811028010086** |
| **NAME** | **Aayush Mahajan** |

**B. Tech CSE-Cloud Computing, Third Year (Section-J2)**

**Faculty Incharge: Ms. Vaishnavi Moorthy, Asst Prof/Dept of CSE**

**YEAR 2020-21**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **INDEX** | | | | | |
| **Ex.no** | **Date of completion** | **Topic** | **Page No.** | **Marks** |
| 1 | 05/08/2020 | Study of Basic Functions of Socket Programming | 3 | 10 |
| 2 | 12/08/2020 | TCP/IP Client-Server Communication | 9 | 9 |
| 3 | 20/08/2020 | UDP Client-Server Communication | 13 | 9 |
| 4 | 28/08/2020 | Date and time request - TCP/IP communication | 16 | 10 |
| 5 | 04/09/2020 | Half Duplex communication using TCP/IP | 20 | 9 |
| 6 | 11/09/2020 | Full Duplex communication using TCP/IP | 24 | 10 |
| 7 | 25/09/2020 | Implementation Of File Transfer Protocol | 31 | 9 |
| 8 | 17/10/2020 | Remote Command Execution Using UDP | 36 | 9.5 |
| 9 | 23/10/2020 | Arp Implementation Using UDP | 41 | 9 |
| 10 | 4/11/2020 | IPv6 addressing and subnetting | 44 | 9 |
| 11 | 4/11/2020 | Communication Using HDLC/PPP | 50 | 10 |

**Date : 05/08/2020**

**EXPERIMENT 1**

**STUDY OF BASIC FUNCTIONS OF SOCKET PROGRAMMING**

**AIM:** To Study About The Standard Library For Socket Programming And Implement Them On Linux Console.

**TECHNICAL OBJECTIVE:** There are a lot of libraries in the socket IO so figure out what they do using the command line interface in linux based OS

**GIVEN REQUIREMENTS:** Using a linux based OS to find out what different libraries are present in the standard library for socket IO

**Commands:**

1. man stdio - It is library required for input and output functions. The standard I/O library provides a simple and efficient buffered stream I/O interface. Input and output is mapped into logical data streams and the physical I/O characteristics are concealed.
2. man string - library for string manipulation function. The string functions perform operations on null-terminated strings.

Eg:

**int strcasecmp(const char \****s1***, const char \****s2***);**

Compare the strings *s1* and *s2* ignoring case.

1. man net db - database access methods. The supported file formats are btree, hashed and UNIX file oriented. The btree format is a representation of a sorted, balanced tree structure. The hashed format is an extensible, dynamic hashing scheme. The flat-file format is a byte stream file with fixed or variable length records.
2. man time - run programs and summarize system resource usage. Time runs the program COMMAND with any given arguments. When COMMAND finishes, time displays information about resources used by COMMAND. If COMMAND exits with non-zero status, time displays a warning message and the exit status. time determines which information to display about the resources used by the COMMAND from the string FORMAT.
3. man date - print or set the system date and time. Display the current time in the given FORMAT, or set the system date.
4. man stat - display file or file system status,

-f, --file-system

display file system status instead of file status

1. man ioctl - control device. system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files may be controlled with ioctl() requests. The argument fd must be an open file descriptor.
2. man errno - look up errno names and descriptions, look up errno macro names, errno codes, and the corresponding descriptions. Search for errors whose description contains all the given words (case-insensitive). List all errno values.
3. man pcap-filter - packet filter syntax, is used to compile a string into a filter program. The resulting filter program can then be applied to some stream of packets to determine which packets will be supplied to pcap\_loop(), pcap\_dispatch(), pcap\_next(), or pcap\_next\_ex().
4. man system - execute a shell command. The system() library function uses fork(2) to create a child process that executes the shell command.system() provides simplicity and convenience: it handles all of the details of calling fork(2), execl(3), and waitpid(2), as well as the necessary manipulations of signals; in addition, the shell performs the usual substitutions and I/O redirections for command. The main cost of system() is inefficiency: additional system calls are required to create the process that runs the shell and to execute the shell.
5. man inet - converts the Internet host address. converts the Internet host address cp from the IPv4 numbers-and-dots notation into binary form (in network byte order) and stores it in the structure that inp points to.
6. man read - read from a file descriptor. It attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and read() returns zero.
7. man write - send a message to another user. The write utility allows you to communicate with other users, by copying lines from your terminal to theirs.
8. man gethostname - These system calls are used to access or to change the hostname of the current processor.returns the null-terminated hostname in the character array name, which has a length of len bytes. If the null-terminated hostname is too large to fit, then the name is truncated, and no error is returned. On success, zero is returned. On error, -1 is returned, and errno is set appropriately.
9. man gethostbyname - It returns a structure of type hostent for the given host name. Here name is either a hostname or an IPv4 address in standard dot notation. If name is an IPv4 address, no lookup is performed and gethostbyname() simply copies name into the h\_name field and its struct in\_addr equivalent into the h\_addr\_list[0] field of the returned hostent structure.
10. man htons - function converts the unsigned short integer hostshort from host byte order to network byte order.
11. man htonl - function converts the unsigned integer hostlong from host byte order to network byte order.
12. man bind - bind a name to a socket. bind() assigns the address specified by addr to the socket referred to by the file descriptor sockfd. addrlen specifies the size, in bytes, of the address structure pointed to by addr. Traditionally, this operation is called “assigning a name to a socket”.
13. man socket - create an endpoint for communication. creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.
14. man Ifconfig - is used to configure the kernel-resident network interfaces. Ifconfig is used to configure the kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed.
15. man send - send a message on a socket. The send() call may be used only when the socket is in a connected state (so that the intended recipient is known). The only difference between send() and write(2) is the presence of flags.
16. man recv - receive a message from a socket. They may be used to receive data on both connectionless and connection-oriented sockets. This page first describes common features of all three system calls, and then describes the differences between the calls.
17. man accept- accept jobs sent to a destination. The cupsaccept command instructs the printing system to accept print jobs to the specified destinations.

**RESULT:**

The Basic Functions Of Socket Programming Were Studied And Executed Successfully By Using Linux Console.

**Date: 12.08.20**

**EXPERIMENT: 2**

**TCP/IP CLIENT-SERVER COMMUNICATION**

**AIM:** To Study about The Simple TCP/IP Client Server Communication By Establishing A Connection Between Host And Client.

**TECHNICAL OBJECTIVE:** To implement an TCP/IP client-server application, where the client establishes a connection with the server, sends a string to the server. The server reads the string, prints it and echoes it back to the client and the client reads the string.

**GIVEN REQUIREMENTS:** There are two hosts, client and server. The client accepts the message from the user and sends it to the server. The server receives the message, prints it and echoes the message back to the client.

**Programming Language:** Python

**Algorithm:**

**Server:**

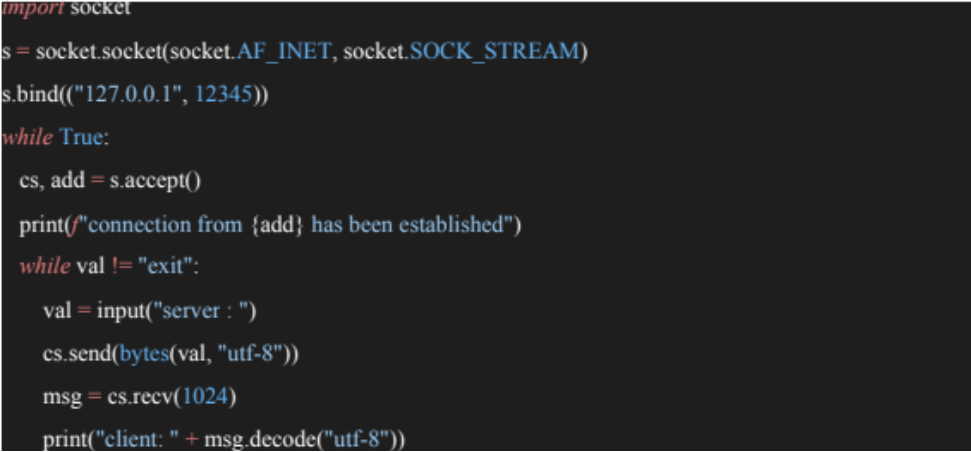
1. Create a socket using the socket() function in c.
2. Initialize the socket address structure as per the server and connect the socket to the address of the server using the connect();
3. Receive and send the data using the recv() and send() functions.  
   Close the connection by calling the close() function.

**Client:**

1. Create a socket using the socket() function in c.
2. Initialize the socket address structure and bind the socket to an address using the bind() function.
3. Listen for connections with the listen() function.
4. Accept a connection with the accept() function system call. This call typically blocks until a client connects to the server.
5. Receive and send data by using the recv() and send() function in c.
6. Close the connection by using the close() function.

**Programming Language:** Python

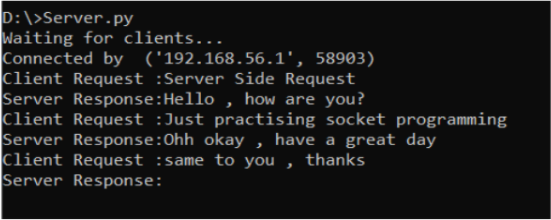
**Server Code:**



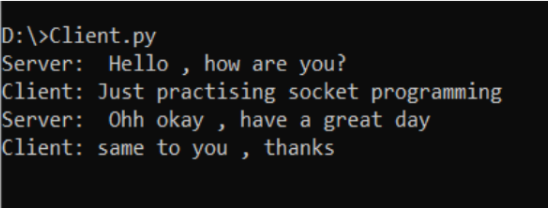
**Client Code:**

****

**Server Side Output:**

****

**Client Side Output:**



**Result:**

The program was executed successfully and a connection was established between the client and server using TCP/IP protocol.

**Date: 20.08.20**

**EXPERIMENT: 3**

**UDP ECHO CLIENT SERVER COMMUNICATION**

**AIM:** To Study The UDP Client Server Communication by establishing a connection between the client and server .

**TECHNICAL OBJECTIVE:** To implement FTP application, where the Client on establishing a connection with the Server sends the name of the file it wishes to access remotely. The Server then sends the contents of the file to the Client, where it is stored.

**GIVEN REQUIREMENTS:** There are two hosts, Client and Server. The Client sends the name of the file it needs from the Server and the Server sends the contents of the file to the Client, where it is stored in a file.

**Algorithm:**

**Server:**

1. Create a socket using socket() system call.

2. Bind socket to port number using bind() system call.

3. Inside a loop waits for the client to send some message.

4. Receive a message using recvfrom() system call.

5. If the received message is ‘bye’, send a message ‘bye’ and stop the chat. If not, send some message using sendto() system call.

**Client:**

1. Create a socket using socket() system call.

2. Inside a loop, type a message to be sent to the server.

3. Send the message using sendto() system call.

4. Receive the message from server using recvfrom() system call.

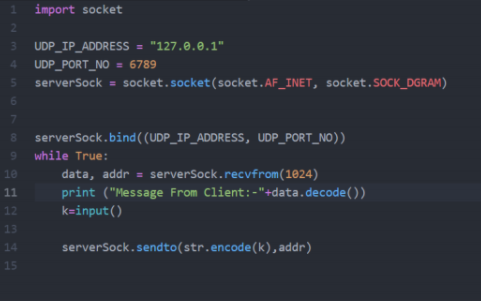
5. If the message received or sent is ‘bye’ then stop.

6. Close the socket.

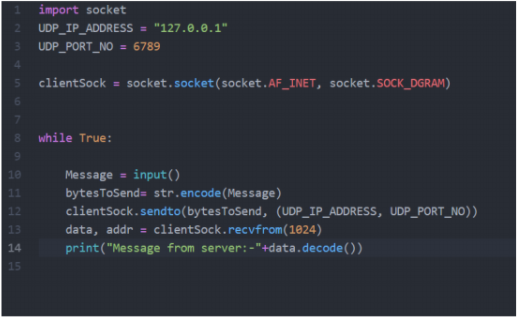
7. Stop.

**Programming Language:** Python

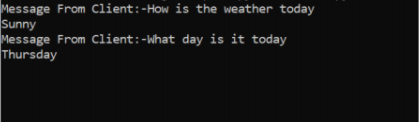
**Server Code:**

****

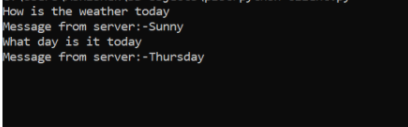
**Client Side Code:**

****

**Server Side Output:**

****

**Client Side Output:**

****

**RESULT:**

The Program Was Executed Successfully And A Connection Was Established Between Client And Server Using UDP Protocol.

**Date: 29.08.20**

**EXPERIMENT: 4**

**Date and Time Request – TCP/IP Communication**

**AIM:** To set up a TCP/IP Client - Server Communication and request for date and time from the server.

**TECHNICAL OBJECTIVE:** The creators of TCP/IP recognized that certain applications might not work properly if there was too much differential between the system clocks of a pair of devices. To support this requirement, they created a pair of ICMP messages that allow devices to exchange system time information. The initiating device creates a Timestamp message and sends it to the device with which it wishes to synchronize.

**GIVEN REQUIREMENTS:** One aspect of this autonomy is that each device maintains a separate system clock. Since even highly-accurate clocks have slight differences in how accurately they keep time, as well as the time they are initialized with, this means that under normal circumstances, no two devices on an internetwork are guaranteed to have exactly the same time.

**ALGORITHM:**

1. Create a TCP server socket.

2. Bind the socket to a server address.

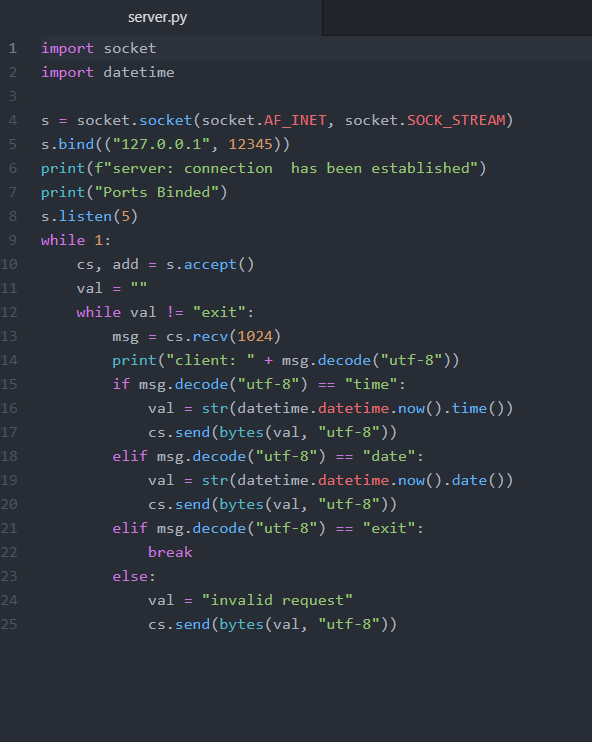
3. Start a client socket while the server socket is passive and waiting for the client to connect.

4. Once connection is established between the client and server socket, send a request for the date or time to the server.

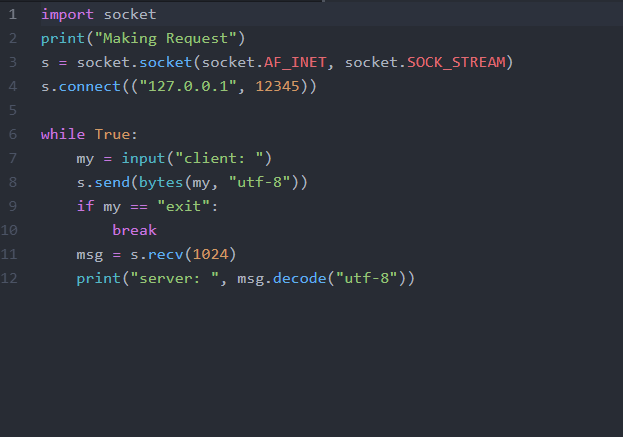
5. Interpret the message from the client and send the required output while the client waits.

**Programming Language:** Python

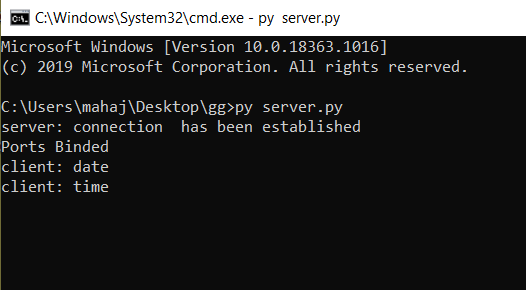
**Server Side Code:**



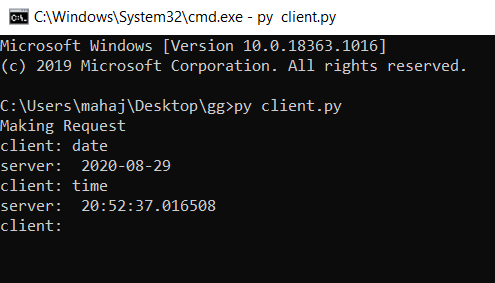
**Client Side Code:**



**Server Side Output:**

****

**Client Side Output:**

****

**Result:**

The connection between the client and server was established and date and time field was noted successfully.

**Date: 5.09.20**

**EXPERIMENT: 5**

**Half Duplex Chat Using TCP/IP Communication**

**AIM:** To implement a half-duplex application, where the Client establishes a connection with the Server. The Client can send and the server well receive messages at the same time.

**TECHNICAL OBJECTIVE:** To implement a half-duplex application, where the Client establishes a connection with the Server. The Client can send and the server well receive messages at the same time.

**GIVEN REQUIREMENTS:** There are two hosts, Client and Server. Both the Client and the Server exchange message i.e. they send messages or receive message from the other. There is only a single way communication between them.

**ALGORITHM:**

**SERVER:**

1. Include the necessary header files.

2. Create a socket using socket function with family AF\_INET, type as SOCK\_STREAM.

3. Initialize server address to 0 using the bzero function.

4. Assign the sin\_family to AF\_INET, sin\_addr to INADDR\_ANY, sin\_port to dynamically assigned port number.

5. Bind the local host address to socket using the bind function.

6. Listen on the socket for connection request from the client.

7. Accept connection request from the Client using accept function.

8. Fork the process to receive message from the client and print it on the console.

9. Read message from the console and send it to the client.

**CLIENT:**

1. Include the necessary header files.

2. Create a socket using socket function with family AF\_INET, type as SOCK\_STREAM.

3. Initialize server address to 0 using the bzero function.

4. Assign the sin\_family to AF\_INET.

5. Get the server IP address and the Port number from the console.

6. Using gethostbyname function assign it to a hostent structure, and assign it to sin\_addr of the server

address structure.

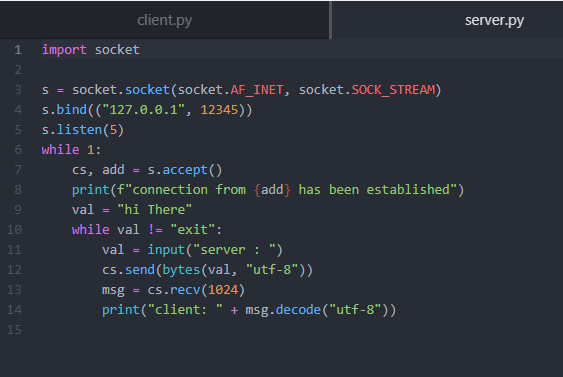
7. Request a connection from the server using the connect function.

8. Fork the process to receive message from the server and print it on the console.

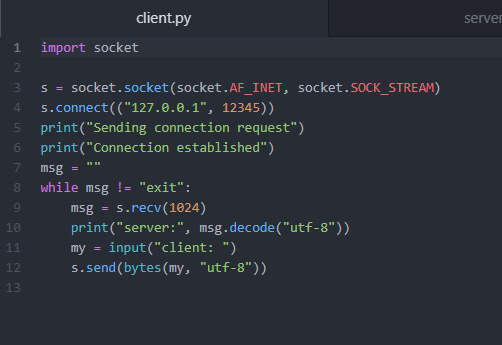
9. Read message from the console and send it to the server.

**Programming Language:** Python

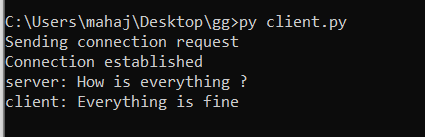
Server Side Code:



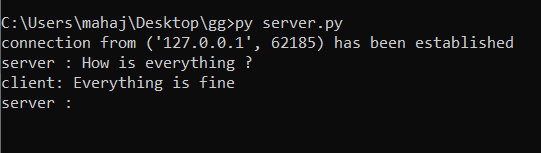
Client Side Code:

****

Client Side Output:



Server Side Output:



**Result:**

Half duplex application, where the Client establishes a connection with the Server. The Client can send and the server well receive messages at the same time, successfully implemented.

**Date: 12.09.20**

**EXPERIMENT: 6**

**Full Duplex Chat Using TCP/IP Communication**

**AIM:** To implement a Full duplex application, where the Client establishes a connection with the Server. The Client can send and the server well receive messages at the same time.

**TECHNICAL OBJECTIVE:** To implement a full duplex application, where the Client establishes a connection with the Server. The Client and Server can send as well as receive messages at the same time. Both the Client and Server exchange messages.

**GIVEN REQUIREMENTS:** There are two hosts, Client and Server. Both the Client and the Server exchange message i.e. they send messages to and receive message from the other. There is a two-way communication between them.

**ALGORITHM**

**SERVER SIDE:**

1. Include the necessary header files.

2. Create a socket using a socket function with family AF\_INET, type as SOCK\_STREAM.

3. Bind the local host address to the socket using the bind function.

4. Listen on the socket for connection requests from the client.

5. Accept connection requests from the Client using accept function.

6. Fork the process to receive a message from the client and print it on the console and another process to read messages from the console and send it to the client simultaneously.

**CLIENT SIDE:**

1. Include the necessary header files.

2. Create a socket using a socket function with family AF\_INET, type as SOCK\_STREAM.

3. Get the server IP address and the Port number from the console.

4. Using gethostbyname function assign it to a hostent structure, and assign it to sin\_addr of the server address structure.

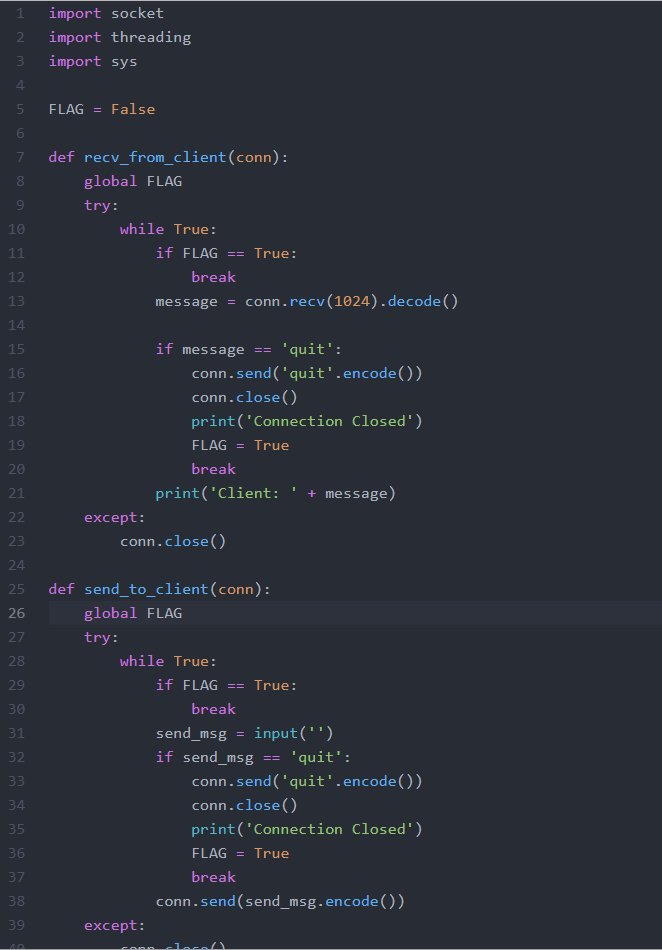
5. Request a connection from the server using the connect function.

6. Fork the process to receive a message from the server and print it on the console and another process to read messages from the console and send it to the server

simultaneously**.**

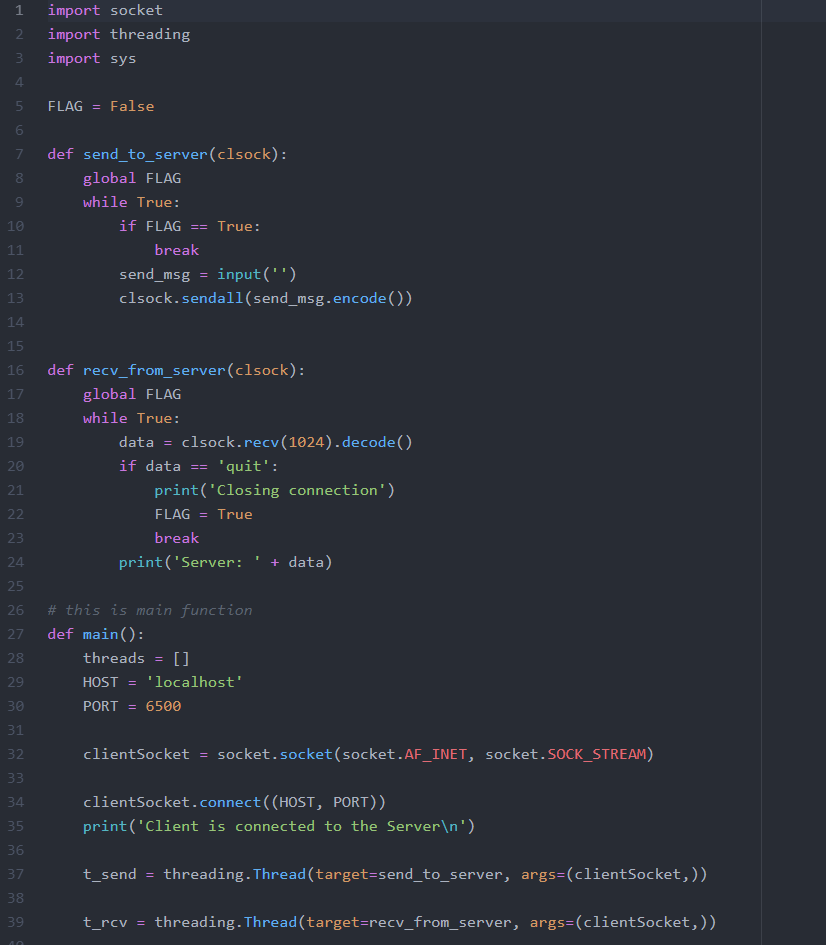
**Programming Language:** Python

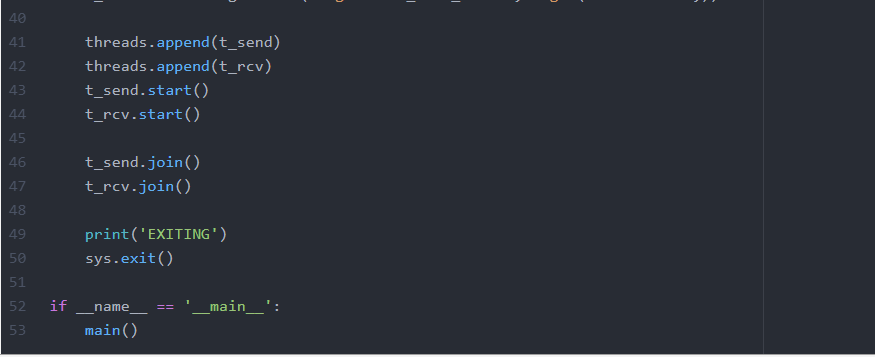
**Server Side Code:**



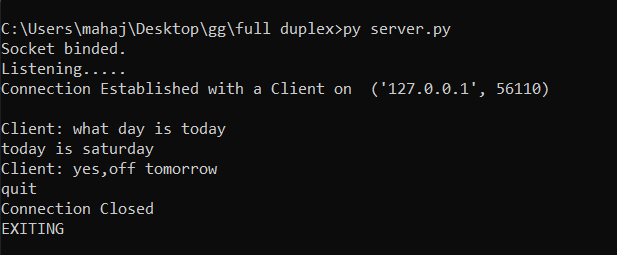


**Client Side Code:**

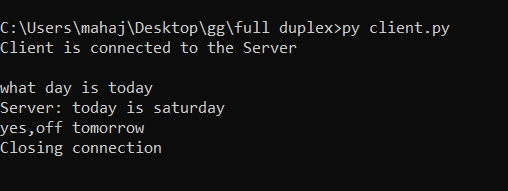




**Server Side Output:**

****

**Client Side Output:**

****

**Result:**

A Full duplex application, where the Client establishes a connection with the Server. The Client can send and the server well receive messages at the same time has been implemented.

**Date: 12.10.20**

**EXPERIMENT: 7**

**IMPLEMENTATION OF FILE TRANSFER PROTOCOL**

**AIM:** To implement FTP application, where the Client on establishing a connection with the Server sends the name of the file it wishes to access remotely. The Server then sends the contents of the file to the Client, where it is stored.

**TECHNICAL OBJECTIVE:** The File Transfer Protocol Is a Standard Network Protocol Used for The Transfer of Computer Files Between a Client and Server On a Computer Network. FTP Is Built On a Client-Server Model Architecture Using Separate Control and Data Connections Between the Client and The Server.

**GIVEN REQUIREMENTS:** There are two hosts, Client and Server. The Client sends the name of the file it needs from the Server and the Server sends the contents of the file to the Client, where it is stored in a file.

**Algorithm**

Server

 Import Socket Module.

 Create a socket object and connect to a port on the local machine.

 Accept filename from client and search in the home directory.

 Open the file in read and write mode, encode and send to the client.

 Close the file.

 Close the connection.

Client

 Import Socket Module

 Create a Socket object.

 Enter a filename.

 Open a file to accept incoming file streams and write to the new/existing file.

 Close the file.

 Close the connection.

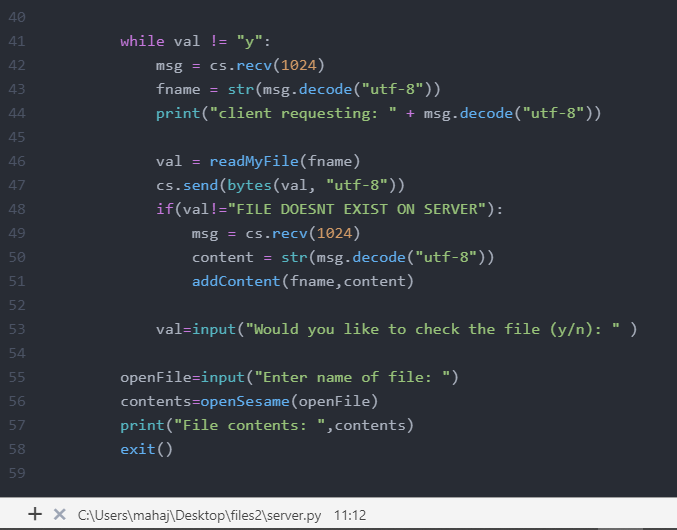
**Programming Language:** Python

Client Side Code:

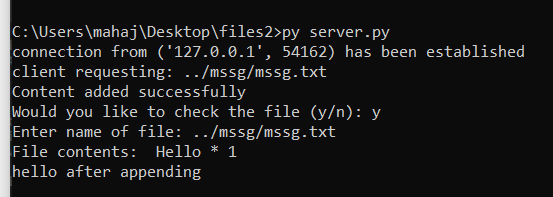


Server Side Code:

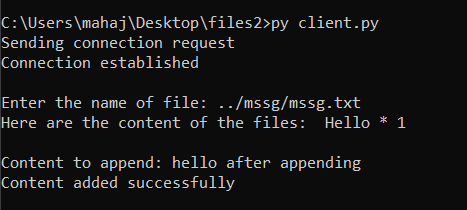




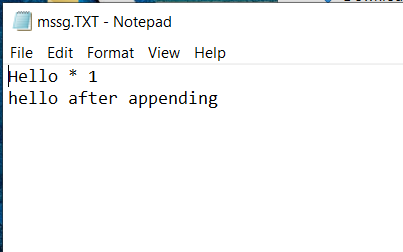
Server Side Output:



Client Side Output:



In The Text File:



**Result:**

Successfully implemented FTP application, where the Client on establishing a connection with the Server sends the name of the file it wishes to access remotely. The Server then sends the contents of the file to the Client, where it is stored.

**Date: 24.10.20**

**EXPERIMENT: 8**

**REMOTE COMMAND EXECUTION USING UDP**

**AIM:** Remote Command execution is implemented through this program using which Client is able to execute commands at the Server. Here, the Client sends the command to the Server for remote execution. The Server executes the command and the send result of the execution.

**TECHNICAL OBJECTIVE:** Remote Command execution is implemented through this program using which Client is able to execute commands at the Server. Here, the Client sends the command to the Server for remote execution. The Server executes the command and the send result of the execution.

**GIVEN REQUIREMENTS**: There are two hosts, Client and Server. The Client sends a command to the Server, which executes the command and sends the result back to the Client.

**ALGORITHM:**

Client Side:

Step1: Start the program and create a client socket that connects to the required host and port.

Step2: Use input streams read message given by server and print it.

Step3: Use input streams; get the message from user to be given to the server.

Step4: Use output streams to write message to the server.

Step5: Stop the program

Server Side:

Step1: Start the program.

Step2: Create a server socket at the server side.

Step3: Create a socket at the client side and the connection is set to accept by the server socket using the accept () method.

Step4: In the client side the remote command to be executed is given as input.

Step5: The command is obtained using the readLine() method of Buffer Reader.

Step6: Get the runtime object of the runtime class using getruntime().

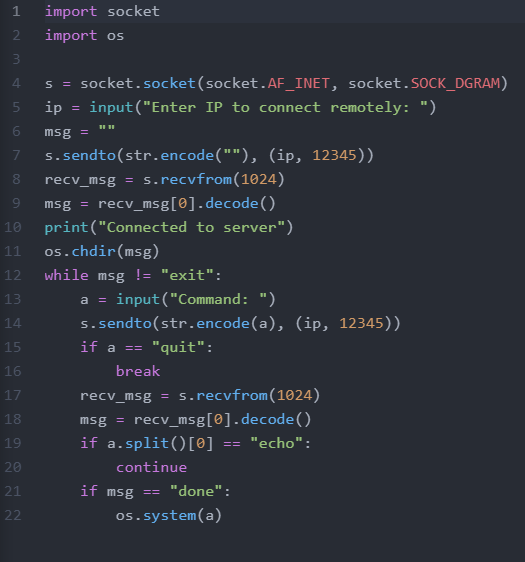
Step7: Execute the command using the exec () method of runtime

**Programming Language:** Python

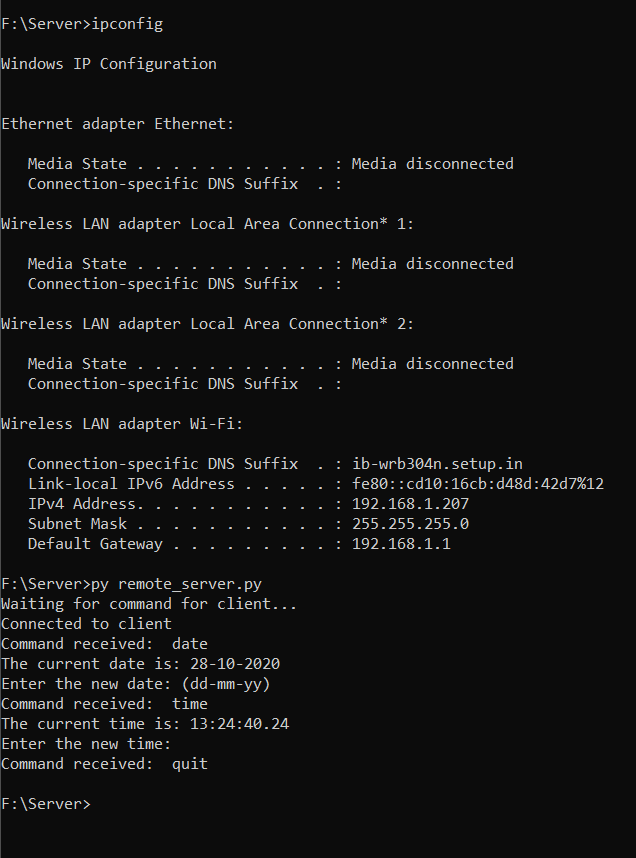
**Server Side Code:**



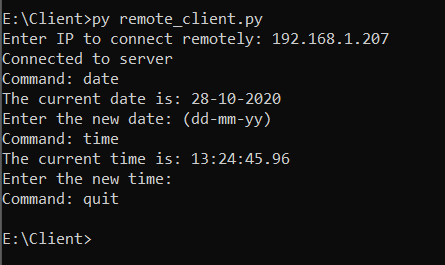
**Client Side Code:**



**Server Side Output:**

****

**Client Side Output:**



**Result:**

Remote Command execution is implemented through this program using which Client is able to execute commands at the Server. Here, the Client sends the command to the Server for remote execution. The Server executes the command and the send result of the execution.

**Date: 29.10.20**

**EXPERIMENT: 9**

**ARP implementation using UDP.**

**AIM:** To carry out the concept of ARP implementation using UDP and get the MAC address as output.

**TECHNICAL OBJECTIVE:** Address Resolution Protocol (ARP) is implemented through this program. The IP address of any Client is given as the input. The ARP cache is looked up for the corresponding hardware address. This is returned as the output. Before compiling that Client is pinged.

**GIVEN REQUIREMENTS:** There is a single host. The IP address of any Client in the network is given as input and the corresponding hardware address is got as the output.

**ALGORITHM:**

1. Include the necessary header files.

2. Create a socket using a socket function with family AF\_INET, type as SOCK\_STREAM.

3. Bind the local host address to the socket using the bind function.

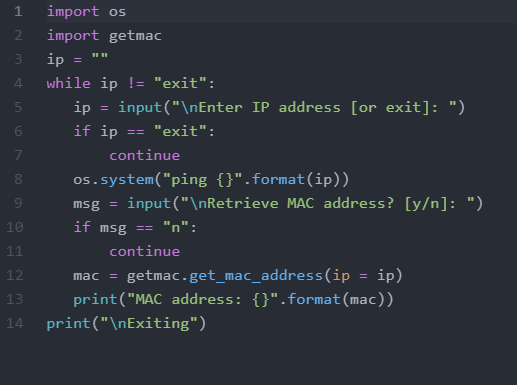
4. Listen on the socket for connection requests from the client.

5. Accept connection requests from the Client using accept function.

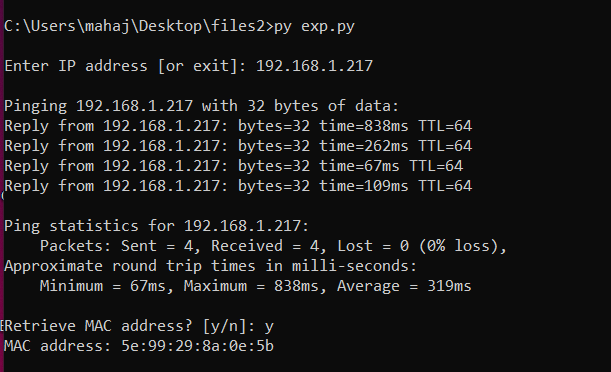
6. Fork the process to receive a message from the client and print it on the console and another process to read messages from the console and send it to the client simultaneously.

**Programming Language:** Python

Code:



Output:



**RESULT:**

Address Resolution Protocol (ARP) is implemented through this program. The IP address of any Client is given as the input. The ARP cache is looked up for the corresponding hardware address. This is returned as the output. Before compiling that Client is pinged.

**Date: 04.11.20**

**EXPERIMENT: 10**

**IPV6 ADDRESSING AND SUBNETTING**

**AIM:** To study IPv6 addressing and subnetting IPv6 Addressing Modes Addressing mode refers to the mechanism of hosting an address on the network. IPv6 offers several types of modes by which a single host can be addressed. More than one host can be addressed at once or the host at the closest distance can be addressed.

**TECHNICAL OBJECTIVE:** To study and understand the types of addressing in IPv6 and subnetting and implementing it in a simulation.

**TECHNICAL REQUIREMENTS:** To Study Ipv6 addressing and subnetting, Ipv6 Addressing mode refers to the mechanism of hosting an address on the network. Ipv6 offers several types of modes by which a single host can Be addressed. More than one host can be addressed at once or the host at the closest distance can be addressed.

**THEORY:**

**IPv6 Addressing**

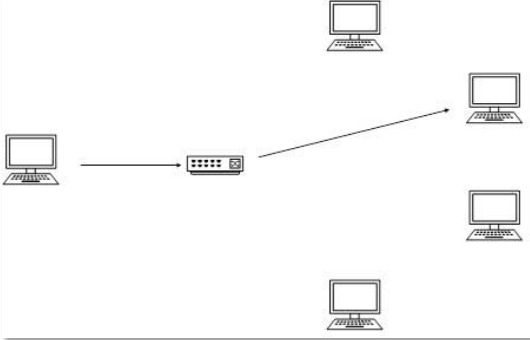
An IPv6 address is 128 bits long

An IPv6 address is represented as eight groups of four hexadecimal digits, each group representing 16 bits (two octets, a group sometimes also called a hextet). The groups are separated by colons (:).

* In hex, 4 bits (also called a ‘nibble’) is represented by a hex digit
* So 128 bits is reduced down to 32 hex digits
* Hexadecimal values of eight 16 bit fields – X:X:X:X:X:X:X:X (X=16 bit number, ex: A2FE) – 16 bit number is converted to a 4 digit hexadecimal number
* Double colons (::) representation – RFC5952 recommends that the rightmost set of :0: be replaced with :: for consistency :-
* 2001:db8:0:2f::5 rather than 2001:db8::2f:0:0:0:5

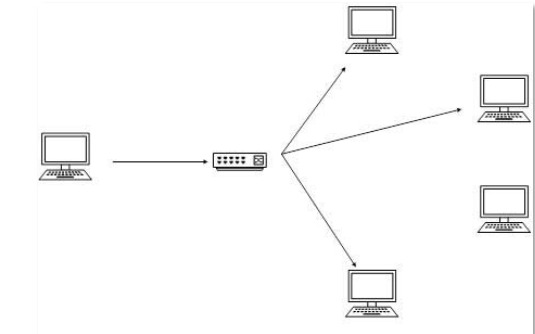
**IPv6 addressing model**

**Unicast**

****

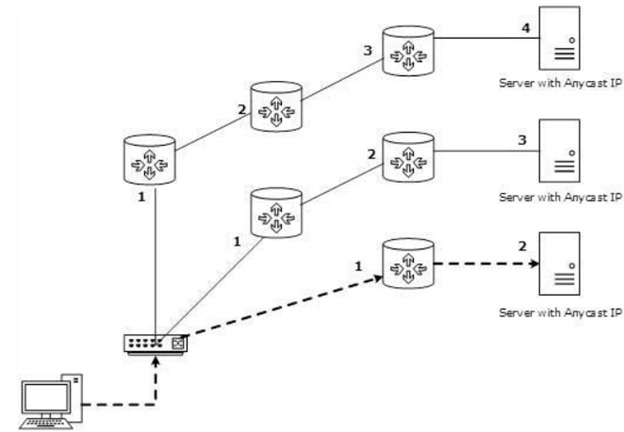
In unicast mode of addressing, an IPv6 interface (host) is uniquely identified in a network segment. The IPv6 packet contains both source and destination IP addresses. A host interface is equipped with an IP address which is unique in that network segment. When a network switch or a router receives a unicast IP packet, destined to a single host, it sends out one of its outgoing interface which connects to that particular host.

**Multicast**

****

The IPv6 multicast mode is same as that of IPv4. The packet destined to multiple hosts is sent on a special multicast address. All the hosts interested in that multicast information, need to join that multicast group first. All the interfaces that joined the group receive the multicast packet and process it, while other hosts not interested in multicast packets ignore the multicast information.

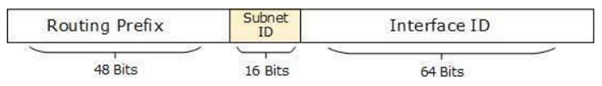
**Anycast**

****

IPv6 has introduced a new type of addressing, which is called Anycast addressing. In this addressing mode, multiple interfaces (hosts) are assigned same Anycast IP address. When a host wishes to communicate with a host equipped with an Anycast IP address, it sends a Unicast message. With the help of complex routing mechanism, that Unicast message is delivered to the host closest to the Sender in terms of Routing cost.

Subnetting: -

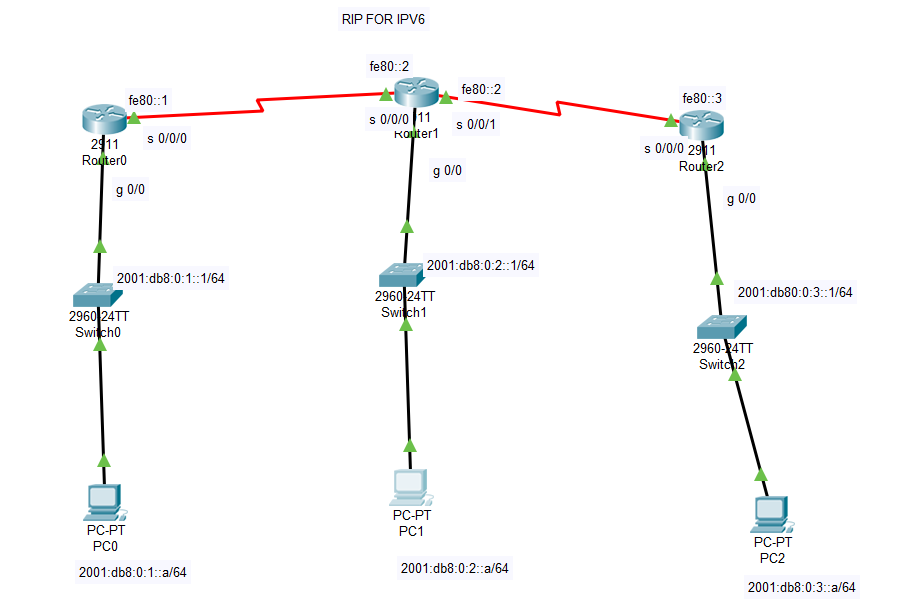
* In IPv4, addresses were created in classes. Class-ful IPv4 addresses clearly define the bits used for network prefixes and the bits used for hosts on that network. To subnet in IPv4, we play with the default classful netmask which allows us to borrow host bits to be used as subnet bits. This results in multiple subnets but less hosts per subnet. That is, when we borrow host bits to create a subnet, it costs us in lesser bit to be used for host addresses.
* IPv6 addresses use 128 bits to represent an address which includes bits to be used for subnetting. The second half of the address (least significant 64 bits) is always used for hosts only. Therefore, there is no compromise if we subnet the network.



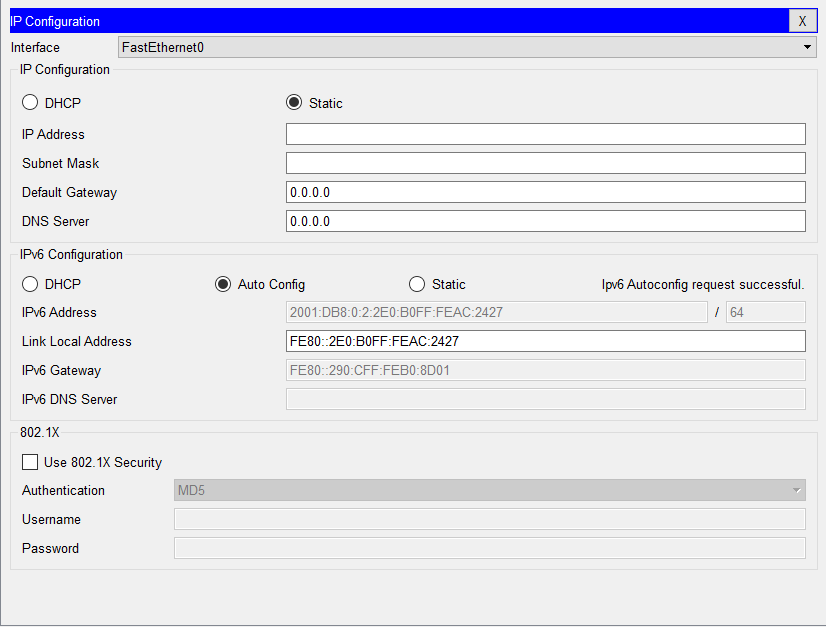
* 16 bits of subnet is equivalent to IPv4’s Class B Network. Using these subnet bits, an organization can have another 65 thousands of subnets which is by far, more than enough.
* Thus routing prefix is /64 and host portion is 64 bits. We can further subnet the network beyond 16 bits of Subnet ID, by borrowing host bits; but it is recommended that 64 bits should always be used for hosts addresses because auto-configuration requires 64 bits.
* IPv6 subnetting works on the same concept as Variable Length Subnet Masking in IPv4.
* /48 prefix can be allocated to an organization providing it the benefit of having up to /64 subnet prefixes, which is 65535 sub-networks, each having 264 hosts. A /64 prefix can be assigned to a point-to-point connection where there are only two hosts (or IPv6 enabled devices) on a link.

**Diagram:**

Cisco Packet Tracer Configuration:



Auto-Configuration for Subnetting:



**RESULT:**

IPv6 addressing and subnetting was carried out in the Cisco packet tracer successfully and the concept was studied.

**Date: 04.11.20**

**EXPERIMENT: 11**

**COMMUNICATION USING HDLC/PPP**

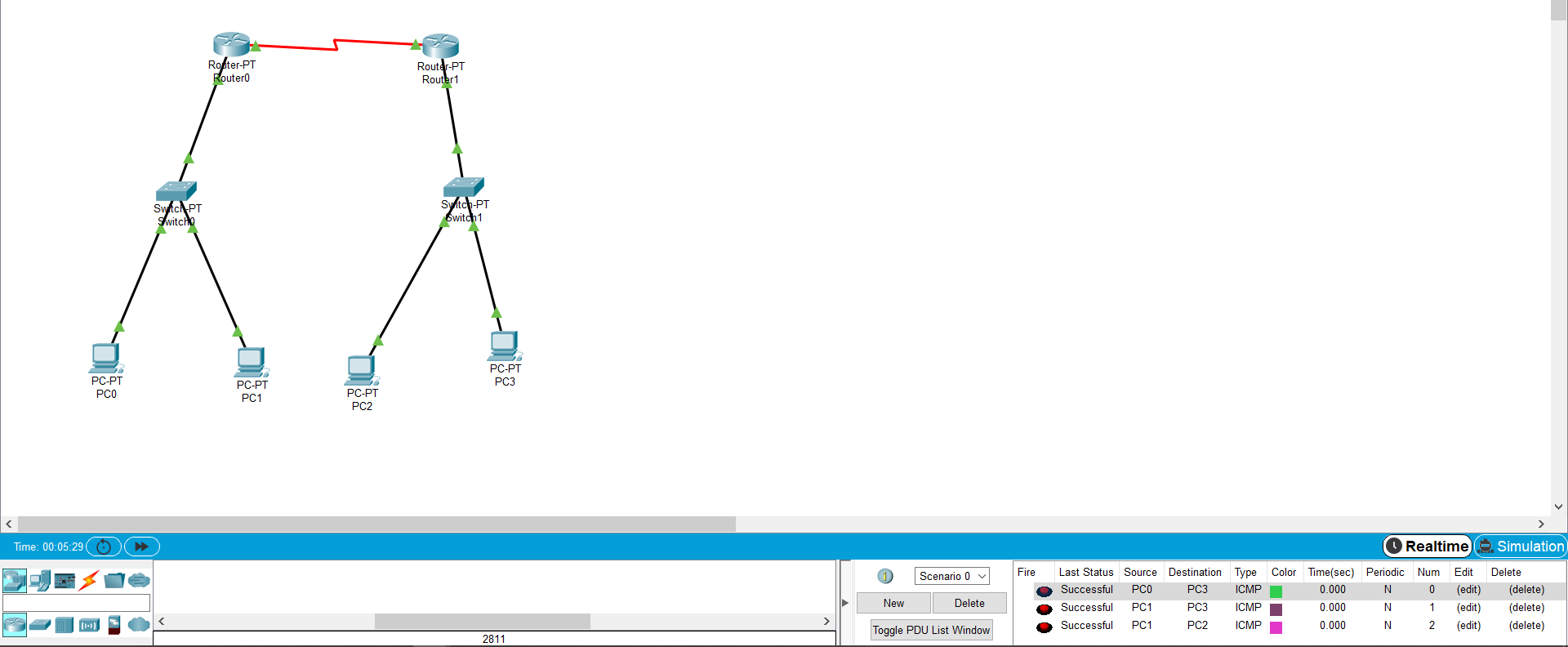
**AIM:** To implement the process of communication using the concept of HDLC and PPP.

**TECHNICAL OBJECTIVE:** HDLC (High-Level Data Link Control) is a Layer 2 WAN Encapsulation Protocol that we use it on Syncronous data links. It is the simplest WAN Protocol that can connect your remote offices over leased lines. PPP (Point to Point Protocol) is also a WAN Encapsulation Protocol that is based on HDLC but we can say that it is the enhanced version of HDLC. There are many additional features in PPP if we compare with HDLC.

**GIVEN REQUIREMENTS:** Communicate between two different networks using HDLC and PPP encapsulation and send packets from both ends.

**Network Diagram:**

Cisco Packet Tracer Configuration:



**STEPS TO CONFIGURE HDLC:**

1) Enable router configuration.

2) Go to interface f0/0, add IP address and subnet mask and initiate no shut down.

3) Interface state is changed to up.

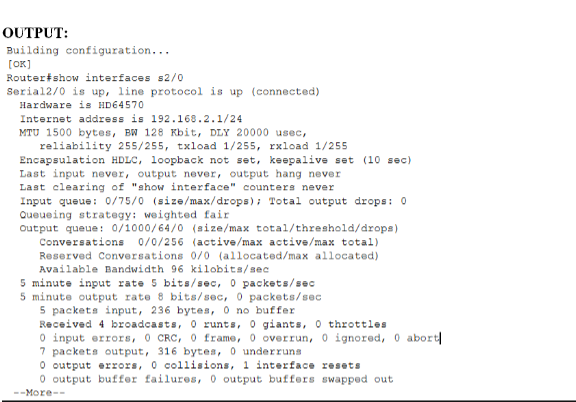
4) Go to interface s2/0 and repeat the steps in 2 and give the clock rate as 64000.

5) Give encapsulation hdlc command in s2/0 interface.

6) Save the configuration.

7) Give the show interface command to check HDLC encapsulation.

Output for HDLC:



**STEPS TO CONFIGURE PPP:**

1) Enable router configuration.

2) Give username hostname and password.

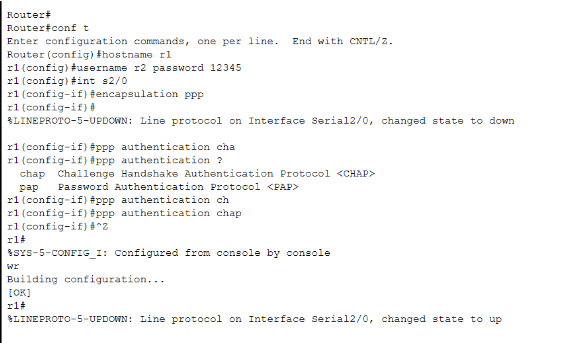
3) Give encapsulation ppp command in s2/0 interface.

4) Save the configuration.

5) Give ppp authentication chap command as shown in figure.

6) Repeat steps for both routers.

Output for PPP:



**RESULT:**

The communication using HDLC and PPP was successful. Both of them were successfully carried out in the Cisco Packet Tracer and the concept was studied.