Government College of Engineering,Jalgaon

(An Autonomous Institute of Govt. of Maharashtra)

Department of Computer Engineering

Lab Manual For
CO455U Distributed Systems Lab
T. Y. B. Tech. (Computer)

Course teacher

Ms. Madhavi Chaudhari

DSL SUBMISSION

Name : KRUSHNA SHIVAJI SASE

Prn No : 1941047

Class : Ty Computer

Bach : T3

Sr. No	Practical Name	DOP	DOC
1	Program for distributed application using RMI	11/09/2021	18/09/2021
2	Program for distributed application using RPC	18/09/2021	25/09/2021
3	To study message passing interface	25/09/2021	02/10/2021
4	Design a Distributed Application using Message passing Interface for remote computation	02/10/2021	02/10/2021
5	Design distributed application which consists of a server and client using threads	16/10/2021	16/10/2021
6	To study java RMI	23/10/2021	23/10/2021
7	Program for distributed chat server using tcp sockets	30/10/2021	30/10/2021
8	RPC mechanism for a file transfer across a network	13/11/2021	13/11/2021
9	To study kerberos	20/11/2021	20/11/2021
10	To study bittorrent and end system multicast	27/11/2021	27/11/2021

GOVERNMENT COLLEGE OF ENGINEERING,JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Krushna Shivaji Sase PRN No: 1941047

Class: T. Y. B.Tech. Computer Batch: T3

Date of Performance: 11/09/2021 Date of Completion: 18/09/2021

Subject:DSL Sign. of Teacher with Date: _____

Experiment No.: 01

Aim: Program for Distributed Application using RMI

Title: Design a distributed application using RMI for remote computation where client submits two strings to the server and server returns concatenation of given strings.

Software Requirements: Ubuntu OS, Eclipse IDE 4.11 **Theory :**

Objective: The main objective of RMI is to provide the facility of invoking methods on the server. This is done by creating a RMI Client and RMI Server. RMI Client invokes a method defined on the RMI Server.

Remote Method Invocation (RMI) is an API which allows an object to invoke a method on an object that exists in another address space, which could be on the same machine or on a remote machine. Through RMI, an object running in a JVM present on a computer (Client side) can invoke methods on an object present in another JVM (Server side). RMI creates a public remote server object that enables client and server side communications through simple method calls on the server object.

Working of RMI

The communication between client and server is handled by using two intermediate objects: Stub object (on client side) and Skeleton object (on server side).

1. **Stub Object**

The stub object on the client machine builds an information block and sends this information to the server. The block consists of

- An identifier of the remote object to be used
- Method name which is to be invoked
- Parameters to the remote JVM

2. **Skeleton Object**

The skeleton object passes the request from the stub object to the remote object. It performs following tasks

- It calls the desired method on the real object present on the server.
- It forwards the parameters received from the stub object to the method.

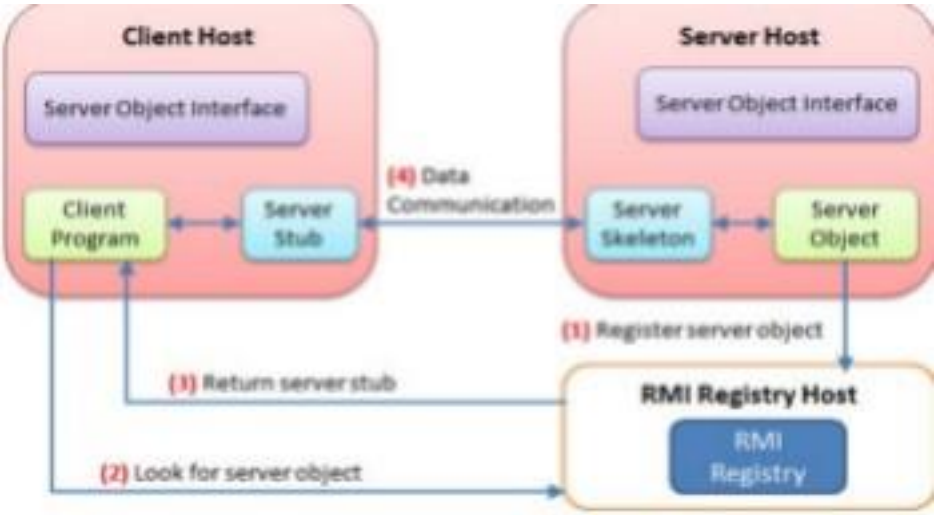


Figure: Working of RMI

Steps to Implement Interface

1. Defining a remote interface
2. Implementing the remote interface
3. Creating Stub and Skeleton objects from the implementation class using rmic (rmi complier)
4. Start the rmiregistry
5. Create and execute the server application program
6. Create and execute the client application program.

Step 1: Defining the remote interface

The first thing to do is to create an interface which will provide the description of the methods that can be invoked by remote clients. This interface should extend the Remote interface and the method prototype within the interface should throw the RemoteException.

```
// Creating a Search interface import
java.rmi.*;
public interface Search extends Remote
{
  // Declaring the method prototype public String query(String
  search) throws RemoteException;
}
```

Step 2: Implementing the remote interface

The next step is to implement the remote interface. To implement the remote interface, the class should extend to the UnicastRemoteObject class of java.rmi package. Also, a default constructor needs to be created to throw the java.rmi.RemoteException from its parent constructor in class.

```
// Java program to implement the Search interface
import java.rmi.*; import java.rmi.server.*;
public class SearchQuery extends UnicastRemoteObject implements
                Search
```

```
{
  // Default constructor to throw RemoteException
  // from its parent constructor
   SearchQuery() throws RemoteException
   {
      super();
   }

  // Implementation of the query interface public
  String query(String search)
                  throws RemoteException
   {
      String result;
      if   (search.equals("Reflection   in   Java"))
         result = "Found";
      else
         result = "Not Found";

      return result;
   }
}
```

Step 3: Creating Stub and Skeleton objects from the implementation class using rmic The rmic tool is used to invoke the rmi compiler that creates the Stub and Skeleton objects. Its prototype is rmic classname. For above program the following command need to be executed at the command prompt rmic SearchQuery

Step 4: Start the rmiregistry

Start the registry service by issuing the following command at the command prompt start rmiregistry

Step 5: Create and execute the server application program

The next step is to create the server application program and execute it on a separate command prompt. The server program uses createRegistry method of LocateRegistry class to create rmi registry within the server JVM with the port number passed as argument.

The rebind method of the Naming class is used to bind the remote object to the new name.

```
//program for server
application import java.rmi.*;
import java.rmi.registry.*;
public class SearchServer
{ public static void main(String args[])
   { try
      {
        // Create an object of the interface
        // implementation class
         Search obj = new SearchQuery();

        // rmiregistry within the server JVM with
        // port number 1900
         LocateRegistry.createRegistry(1900);

        // Binds the remote object by the name
```

```
            // geeksforgeeks
             Naming.rebind("rmi://localhost:1900"+
                        "/geeksforgeeks",obj);
                        }
            catch(Exception ae)
            {
               System.out.println(ae);
            }
        }
    }
```

Step 6: Create and execute the client application program

The last step is to create the client application program and execute it on a separate command prompt .The lookup method of the Naming class is used to get the reference of the Stub object.

```
//program for client
application import java.rmi.*;
public class ClientRequest
{ public static void main(String args[])
    {
       String  answer,value="Reflection  in  Java";
       try
       {
          // lookup method to find reference of remote object Search
          access =
             (Search)Naming.lookup("rmi://localhost:1900"+
                              "/geeksforgeeks");
          answer = access.query(value);
          System.out.println("Article on " + value +
                        " " + answer" at GeeksforGeeks");
       }
       catch(Exception ae)
       {
          System.out.println(ae);
       }
    }
}
```

Algorithm:

Steps:

1. Define a Remote Interface that contains methods which can be remotely invoked on remote object. Each Remote Interface have to extend – Java.rmi.Remote

2. Interface and each remote method declared in the remote interface have to throw java.rmi.RemoteException. **Listing 1**: Define Remote Interface, declaring Remote Method. **Listing 2**: Defining class that extends and implements UnicastRemoteObject and Interface

3. Generate Stub & Skeleton classes using rmic tool:
   rmic <remote-class>

4. compile and generate Stub and Skeleton Classes 5. Start Registry Services using rmiRegistry tool:

rmiRegistry <port-number> or **rmiRegistry**

- Create an object of the RMI server and register its stub in the RMI registry. Java.rmi.Naming class provides a static method for registration in the lookup of rmiStub.

- **bind()** : This method is used to register a remote object (stub of object) in the rmiRegistry. Syntax: public static void bind(String name, Remote object) throws RemoteException, AlreadyBindException

- **rebin()**: This method is used to rebind the remote object in the rmiRegistry.

Syntax: Public static void rebind(String name, Remote object) throws RemoteException **Lookup()** : This method is used to lookup remote stub in the rmi Registry. Syntax: Public static Remote lookup(String name) throws RemoteException

Listing 4: Define a RMI Server Class

6. Now define a RMI Client. A RMI Client has to obtain the reference of remote stub from the RMIregistry and invokes the remote method.

Listing 5: Define a RMI Client Class

If RMI registry is running on non-default port on a different machine, then following steps are required for registration:

Java.rmi.registry.LocateRegistry class is used to create a registry object. This class provide following methods:

Public static Registry getRegistry (int port);

Public static Registry getRegistry(String host, int port);

7. **Listing 6**: Define how to register a server in Registry if RMI Registry run on non-default port Info regarding RMI registry is provided to the lookup () method through “Lookup String”. Lookup string has the following format:

LookupString:

“protocol:hostname:portNo/registeredNameserver”

8. **Listing 7**: If RMI Registry run on non-default port, define a RMI Client

<u>Compile RMI Client and Server Classes and Interface</u>

- First of all compile all the classes and interface using javac command: javac *.java Now generate stub class file from remote method class that implements Remote Methods using rmic tool for remote : rmic hello
- Now start rmiregistry in one console:

- Start rmiregistry in command window using command rmiregistry

- We can also rmiregistry on different ports using rmiregistry <port-no>.
- Star RMI server using java <server-class-name> command Using Java

<rmi-server-class-name> & is placed in the command window to start rmi server.

Now start a RMI client using java <client-class-name> and command line arguments.

Conclusion: In this Practical, we learnt about development of distributed applications using RMI.

DOC: 02/10/2021

……

Mrs. Madhavi Kale

Name & Sign of Course Teacher

Client.java importjava.sql.*;

```
import java.rmi.*;
import java.io.*;
import java.util.*;
import java.util.Vector.*;
import java.lang.*;
import java.rmi.registry.*;
public class Client_Practical_1
{
static String name1,name2,name3;
public static void main(String args[])
{
Client_Practical_1 c=new Client_Practical_1();
BufferedReader b = new BufferedReader(new InputStreamReader(System.in));
int ch;
try { Registry r1 = LocateRegistry.getRegistry ( "localhost", 1030); DBInterface
DI=(DBInterface)r1.lookup("DBServ");
do
```

```
{
System.out.println("1.send input stings\n2.Displayconcatenated string \nEnter ur choice");
ch= Integer.parseInt(b.readLine());
switch(ch)
{ case 1:
System.out.println(" \n Enter first string:");
name1=b.readLine();
System.out.println(" \n Enter second string:");
name2=b.readLine();
name3=DI.input(name1,name2);
break;
case 2:
//display
System.out.println("\n Concatenated String is : ");
int i=0;
System.out.println(" " +name3+"");
break;
}
}while(ch>0);
}
catch (Exception e)
{ // System.out.println("ERROR: " +e.getMessage()); }
}
}
}
```

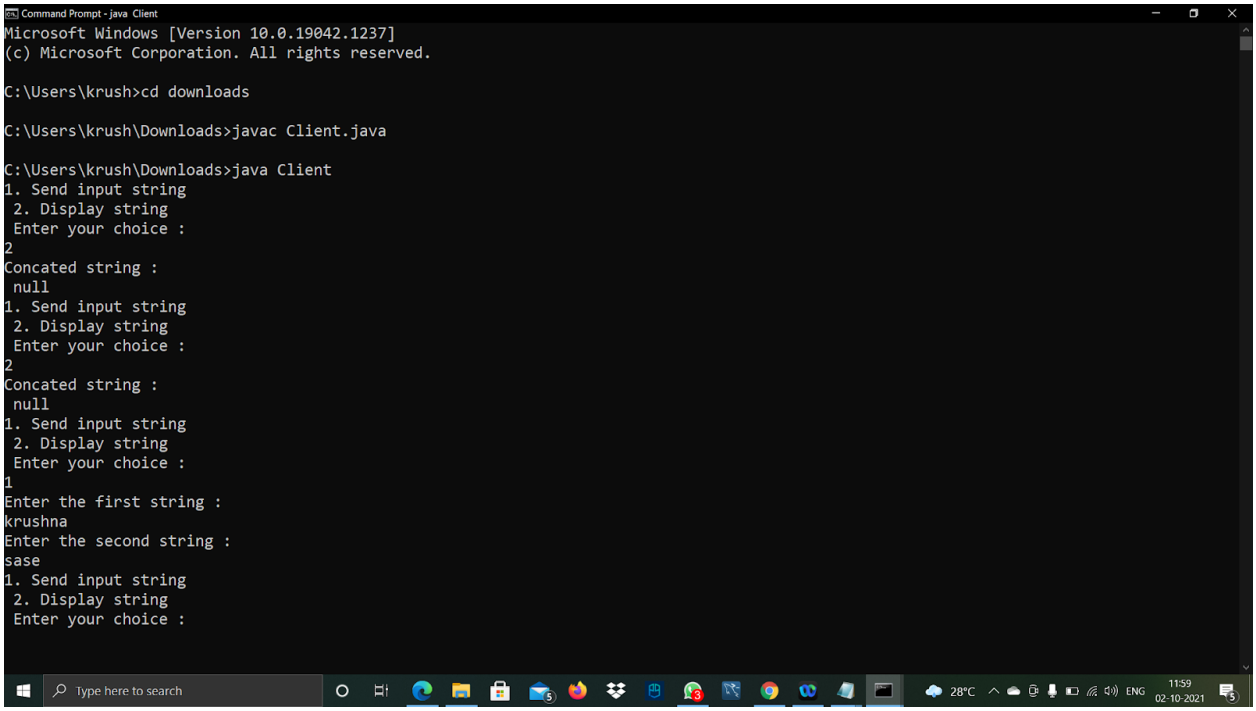
Server.java

```
import java.sql.*;
import java.sql.Connection;
import java.rmi.*;
import java.rmi.Naming.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.Vector;
interface DBInterface extends Remote
```

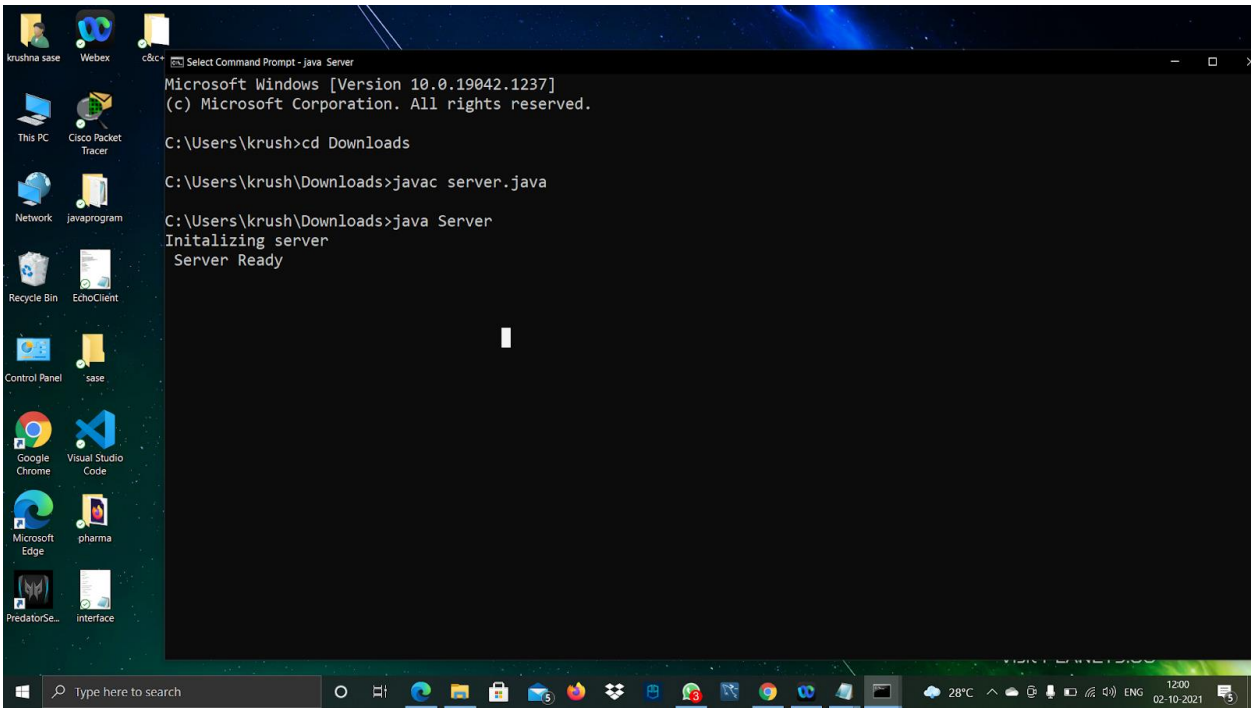
```
{
public String input(String name1,String name2) throws RemoteException; }
public class Server_Practical_1 extends UnicastRemoteObject implements
DBInterface
{
int flag=0,n,i,j;
String name3; ResultSet r;
public Server_Practical_1() throws RemoteException
{ try
{ System.out.println("Initializing Server\nServer Ready"); } catch (Exception e)
{ System.out.println("ERROR: " +e.getMessage());
}
}
public static void main(String[] args)
{ try
{
Server_Practical_1 rs=new Server_Practical_1();
java.rmi.registry.LocateRegistry.createRegistry(1030).rebind("DBServ",rs); }
catch (Exception e)
{
System.out.println("ERROR: " +e.getMessage());
}
}
public String input(String name1,String name2)
{
try{
name3=name1.concat(name2);
} catch (Exception e)
{
System.out.println("ERROR: " +e.getMessage()); }
return name3;
}
}
```

Output:

Client.java



Server.java


Select Command Prompt - java Server
Microsoft Windows [Version 10.0.19042.1237]
(c) Microsoft Corporation. All rights reserved.
C:\Users\krush>cd Downloads
C:\Users\krush\Downloads>javac server.java
C:\Users\krush\Downloads>java Server
Initalizing server
Server Ready
krushna sase
Webex
c&c++
This PC
Cisco Packet Tracer
Network
javaprogram
Recycle Bin
EchoClient
Control Panel
sase
Google Chrome
Visual Studio Code
Microsoft Edge
pharma
PredatorSe...
interface
Type here to search
28°C
ENG
12:00
02-10-2021

GOVERNMENT COLLEGE OF ENGINEERING,JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Krushna Shivaji Sase PRN No: 1941047

Class: T. Y. B.Tech. Computer Batch: T3

Date of Performance: 18/09/2021 Date of Completion: 25/09/2021

Subject:DSL Sign. of Teacher with Date: _____

Experiment No.: 02

Aim: Program for Distributed Application using RPC

Title: Design a distributed application using RPC for remote computation where client submits an integer value to the server and server calculates factorial and returns the result to the client program.

Software Requirements: Ubuntu OS, Eclipse IDE 4.11

Theory :

Remote Procedure Call

Objective:

Remote Procedure Call (**RPC**) is a protocol that one program can **use** to request a service from a program located in another computer on a network without having to understand the network's details. A procedure call is also sometimes known as a function call or a subroutine call. **RPC uses** the client-server model

A remote procedure call is an interprocess communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call.

A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the

required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished.

In distributed computing, a **remote procedure call** (**RPC**) is when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote.[1] This is a form of client–server interaction (caller is client, executor is server), typically implemented via a request–response message-passing system. In the object-oriented programming paradigm, RPC calls are represented by remote method invocation (RMI).

The RPC model implies a level of location transparency, namely that calling procedures is largely the same whether it is local or remote, but usually they are not identical, so local calls can be distinguished from remote calls. Remote calls are usually orders of magnitude slower and less reliable than local calls, so distinguishing them is important.

RPCs are a form of inter-process communication (IPC), in that different processes have different address spaces: if on the same host machine, they have distinct virtual address spaces, even though the physical address space is the same; while if they are on different hosts, the physical address space is different. Many different (often incompatible) technologies have been used to implement the concept.

RPC Implementation Mechanism:

- RPC mechanism uses the concepts of stubs to achieve the goal of semantic transparency.
- Stubs provide a local procedure call abstraction by concealing the underlying RPC mechanism.
- A separate stub procedure is associated with both the client and server processes.
- RPC communication package known as RPC Runtime is used on both the sides to hide existence and functionalities of a network.

- implementation of RPC involves the five elements of program:
    1. Client
    2. Client Stub
    3. RPC Runtime
    4. Server stub
    5. Server

The sequence of events in a remote procedure call are given as follows:

1. Client

- A Client is a user process which initiates a RPC
- The client makes a normal call that will invoke a corresponding procedure in the client stub.

2. Client Stub

Client stub is responsible for the following two tasks:

i. On receipt of a call request from the client, it packs specifications of the target procedureand arguments into a message and asks the local RPCRuntime to send it to the server stub.

ii. On receipt of the result of procedure execution, it unpacks the result and passes it to the client.

3. RPCRuntime

- Transmission of messages between Client and the server machine across the network is handled by RPCRuntime.
- It performs Retransmission, Acknowledgement, Routing and Encryption.
- RPCRuntime on Client machine receives messages containing result of procedure execution from server and sends it client stub as well as the RPCRuntime on server machine receives the same message from server stub and passes it to client machine.
- It also receives call request messages from client machine and sends it to server stub.

4. Server Stub

Server stub is similar to client stub and is responsible for the following two tasks:

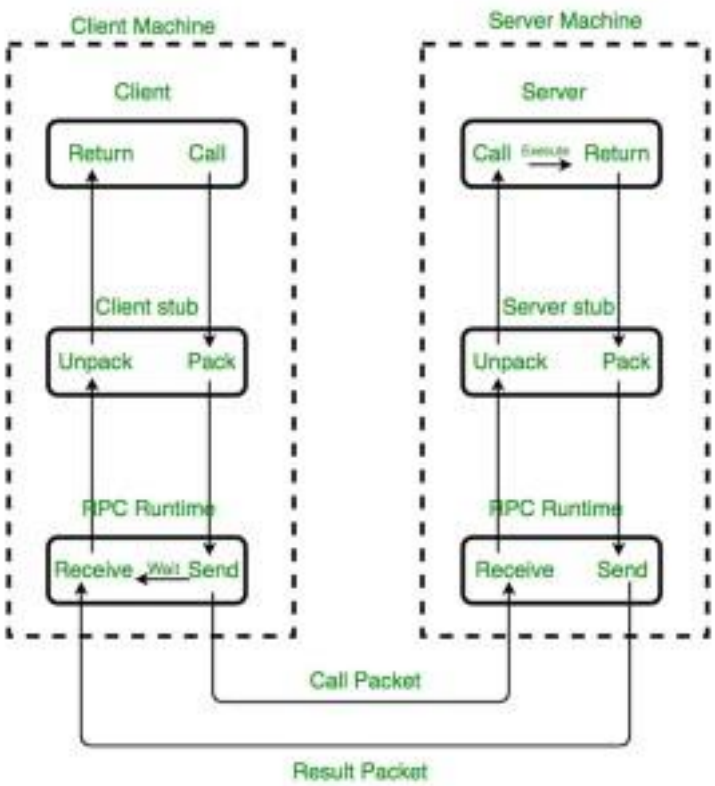
i. On receipt of a call request message from the local RPCRuntime, it unpacks and makes anormal call to invoke the required procedure in the server.

ii. ii. On receipt of the result of procedure execution from the server, it unpacks the result intoa message and then asks the local RPCRuntime to send it to the client stub.

5. Server

When a call request is received from the server stub, the server executes the required procedure and returns the result to the server stub. Thus we have successfully implemented RPC mechanism for a file transfer across a network

- The client stub is called by the client.
- The client stub makes a system call to send the message to the server and puts the parameters in the message. The message is sent from the client to the server by the client's operating system.
- The message is passed to the server stub by the server operating system.
- The parameters are removed from the message by the server stub.
- Then, the server procedure is called by the server stub.



Implementation of RPC mechanism

- **ALGORITHM:**

  Steps:

  1. Create the IDL**,** Open terminal
  - sudo apt-get adv –keyserver pgp.mit.edu –recv-keys A4A9406876FCBD3C456770C88C718D3B5072E1F5
  - sudo apt-get updates
  - sudo apt-get install rpcbind
  - mkdir exp2
  - cd exp2
  - gedit fact.x

  2. add following code in itstruct intpair { int a;
  };
  program FACT_PROG {
  version FACT_VERS {
  int FACT(intpair) = 1;
  } = 1;
  } = 0x23451111;

  3. save and exit the file
  - rpcgen -a -C fact.x
  - gedit Makefile.fact

  4. find the following line in
  the fileCFLAGS += -g
  and change it to:
  CFLAGS += -g -DRPC_SVC_FG

5. find the following line in the same fileRPCGENFLAGS = and change it to:
   RPCGENFLAGS = -C
6. save and exit the filegedit fact_client.c
7. save and exit the filegedit fact_server.c
8. save and exit the file
9. compilemake -f Makefile.fact
10. In one terminal, run:
    sudo ./fact_server
11. In another terminal, run:
    cd exp2
    $exp2/./fact_client localhost

Conclusion: In this Practical, we learnt about development of distributed application using RPC

DOC: 30/03/2021

……

Mrs. Madhavi Kale

Name & Sign of Course Teacher

rpcserver.py

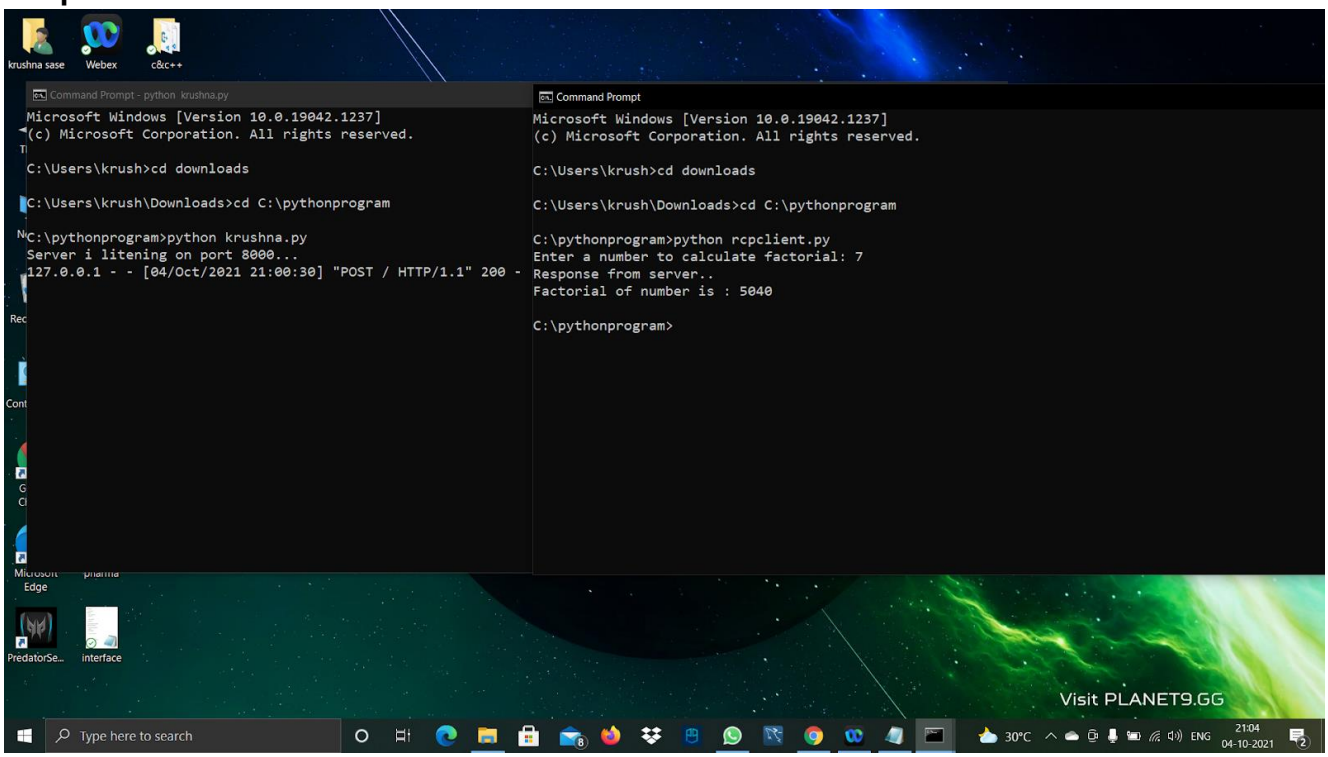
```
from xmlrpc.server import
SimpleXMLRPCServer import
xmlrpc.client

def fact(n):
    if n < 0:
        return 0
    elif n == 0 or n == 1:
        return 1
    else:
        fact = 1
        while(n > 1):
            fact *= n
```

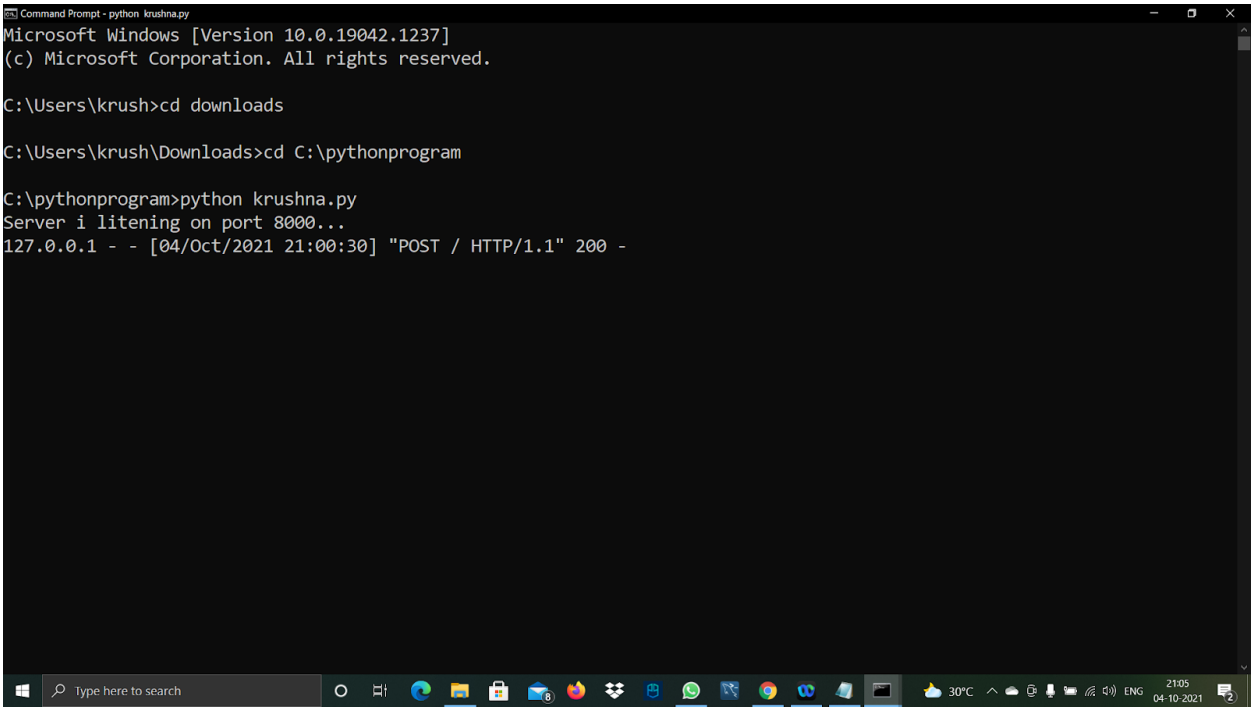
```
        n -= 1
    return fact

server =
SimpleXMLRPCServer(("localhost",8000))
print("Server i litening on port 8000...")

server.register_function(fact,"fact")

server.serve_forever()
```

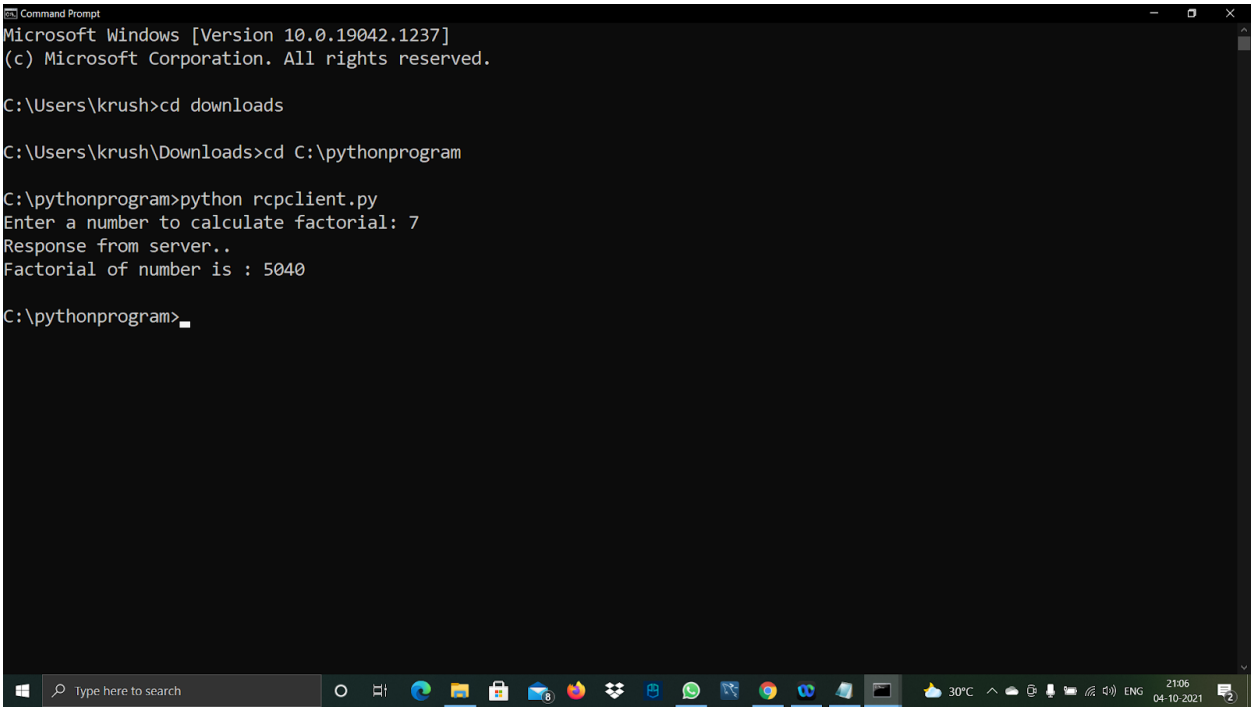
rpcclient.py:

```
import xmlrpc.client
proxy =

xmlrpc.client.ServerProxy("http://localhost:8000/")

n = int(input("Enter a number to calculate factorial:

")) factorial = proxy.fact(n)

print("Response from server..")
print("Factorial of number is :",factorial)
```

Output:




Command Prompt - python krushna.py
Microsoft Windows [Version 10.0.19042.1237]
(c) Microsoft Corporation. All rights reserved.
C:\Users\krush>cd downloads
C:\Users\krush\Downloads>cd C:\pythonprogram
C:\pythonprogram>python krushna.py
Server i litening on port 8000...
127.0.0.1 - - [04/Oct/2021 21:00:30] "POST / HTTP/1.1" 200 -
Type here to search
30°C
ENG
21:05
04-10-2021


Command Prompt
Microsoft Windows [Version 10.0.19042.1237]
(c) Microsoft Corporation. All rights reserved.
C:\Users\krush>cd downloads
C:\Users\krush\Downloads>cd C:\pythonprogram
C:\pythonprogram>python rcpclient.py
Enter a number to calculate factorial: 7
Response from server..
Factorial of number is : 5040
C:\pythonprogram>
Type here to search
30°C
ENG
21:06
04-10-2021

GOVERNMENT COLLEGE OF ENGINEERING, JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Krushna Shivaji Sase PRN No: 1941047

Class: T. Y. B.Tech. Computer Batch: T3

Date of Performance: 25/09/2021 Date of Completion: 02/10/2021

Subject:DSL Sign. of Teacher with Date: _____

Experiment No.: 03

Aim: To Study Message Passing Interface (MPI)

Title: To Study Message Passing Interface (MPI) for Distributed Computing system.

Theory :

Message Passing Interface (MPI)

Objective:

Message Passing Interface (MPI) is a standardized and portable message passing system distributed and developed for parallel computing. MPI provides parallel hardware vendors with a clearly defined base set of routines that can be efficiently implemented.

The message passing interface (MPI) is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory.

In parallel computing, multiple computers -- or even multiple processor cores within the same computer -- are called nodes. Each node in the parallel arrangement typically works on a portion of the overall computing problem. The challenge then is to synchronize the actions of each parallel node, exchange data between nodes and provide command and

control over the entire parallel cluster. The message passing interface defines a standard suite of functions for these tasks.

FEATURES:

1. General
   - Communicators combine context and group for message security-Thread safety

2. Point-to-point communication
   -Structured buffers and derived datatypes, heterogeneity

   - Modes: normal (blocking and non-blocking), synchronous, ready (to allowaccess to fast protocols), buffered

3. Collective
   - Both built-in and user-defined collective operations
   - Large number of data movement routines
   - Subgroups defined directly or by topology
4. Non-message-passing concepts not included:
   - process management
   - remote memory transfers
   - active messages
   - threads
   - virtual shared memory
5. MPI does not address these issues, but has tried to remain compatible with theseideas (e.g. thread safety as a goal, intercommunicators)

Working of MPI in Distributed Computing:

- **Distributed Memory**
  Every processor has its own local memory which can be accessed directly only by its own CPU. Transfer of data from one processor to another is performed over a network. Differs from shared memory systems which permit multiple processors to directly access the same memory resource via a memory bus.

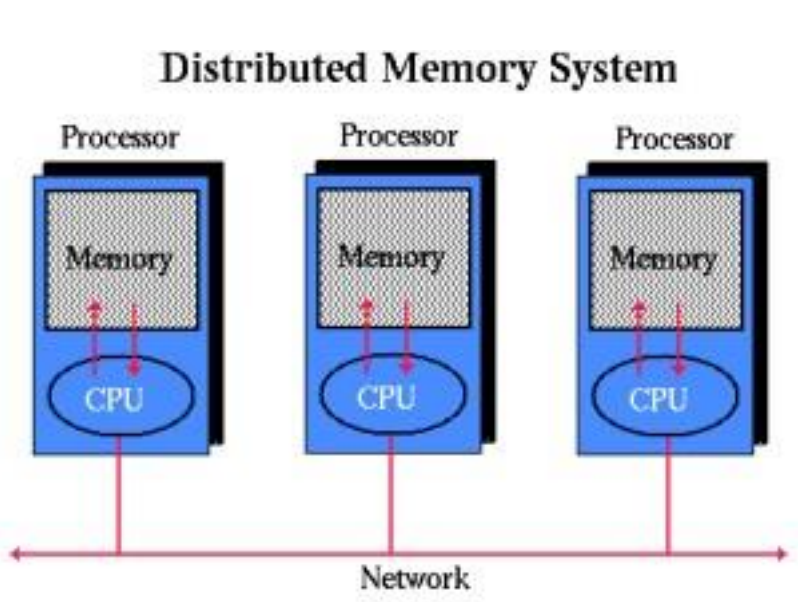


Figure: Message Passing Interface in DS

- **Message Passing**

  The method by which data from one processor's memory is copied to the memory of another processor. In distributed memory systems, data is generally sent as packets of information over a network from one processor to another. A message may consist of one or more packets, and usually includes routing and/or other control information.

  CO455 Distributed Operating System Lab

- **Processor**

  A process is a set of executable instructions (program) which runs on a processor. One or more processes may execute on a processor. In a message passing system, all processes communicate with each other by sending messages - even if they are running on the same processor. For reasons of efficiency, however, message passing systems generally associate only one process per processor.

- **Message Passing Library**

  Usually refers to a collection of routines which are imbedded in application code to accomplish send, receive and other message passing operations.

- **Send / Receive**

  Message passing involves the transfer of data from one process (send) to another process (receive). Requires the cooperation of both the sending and receiving process. Send operations usually require the sending process to specify the data's location, size, type and the destination. Receive operations should match a corresponding send operation.

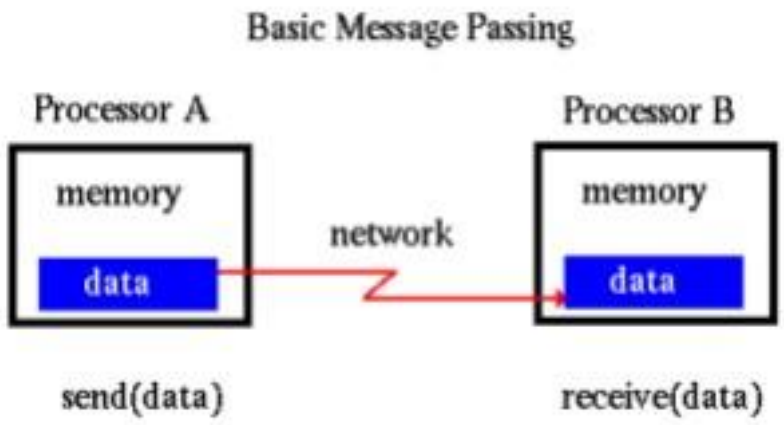


Figure: Send & Receive Msg in MPI in Network

- **Synchronous / Asynchronous**

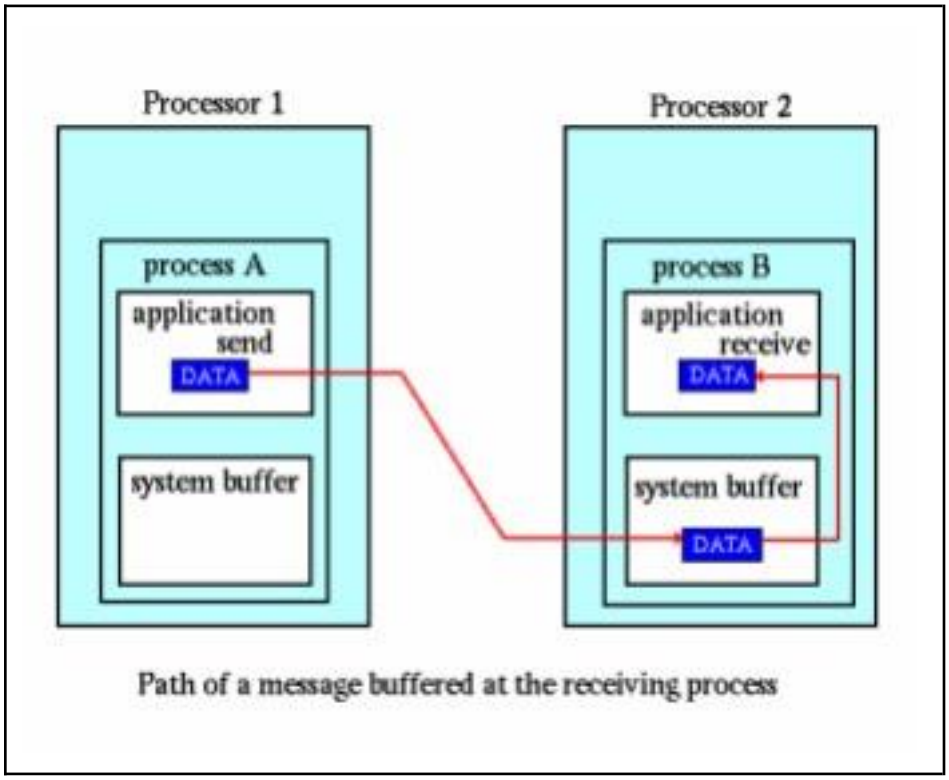
A synchronous send operation will complete only after acknowledgement that the message was safely received by the receiving process. Asynchronous send operations may "complete" even though the receiving process has not actually received the message.

- **Application Buffer**

The address space that holds the data which is to be sent or received. For example, your program uses a variable called, "in msg". The application buffer for in msg is the program memory location where the value of in msg resides.

- **System Buffer**

System space for storing messages. Depending upon the type of send/ receive operation, data in the application buffer may be required to be copied to/from system buffer space.
Allows communication to be asynchronous.



- **Blocking Communication**

  A communication routine is blocking if the completion of the call is dependent on certain "events". For sends, the data must be successfully sent or safely copied to system buffer space so that the application buffer that contained the data is available for reuse. For receives, the data must be safely stored in the receive buffer so that it is ready for use.

- **Non-blocking Communication**

  A communication routine is non-blocking if the call returns without waiting for any communications events to complete (such as copying of message from user memory to system memory or arrival of message).

  It is not safe to modify or use the application buffer after completion of a non-blocking send. It is the programmer's responsibility to insure that the application buffer is free for reuse.

  Non-blocking communications are primarily used to overlap computation with communication to effect performance gains.

Advantage of MPI

1. Standardization: MPI is the only message passing library which can be considered a standard.
2. Portability: no need to modify your source code when you port your application to a different platform.
3. Functionality: Many routines available to use.
4. Availability: A variety of implementations are available.

Disadvantages of MPI:

1. Lack of scalability between memory and CPU. Adding more CPUs can increase the traffic on shared memory associated with cache and memory management.
2. Programmer responsibility for synchronization construct that ensure correct access of global memory.
3. Expense: It becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing number of processors.

- **Case Study: BSD UNIX IPC Mechanism**

  Feature:

1. It is network independent in the sense that it can support communication networks that use different sets of protocols, different naming conversions, different hardware, and so on.

2. It uses a unified abstraction, called socket, for an endpoint of communication. That is, a socket is an abstract object form which message are sent and received.

3. For location transparency, it uses a two-level naming scheme for naming communication endpoints That is socket can be assign a high-level name that is a human readable string. 4.

It is highly flexible in the sense that it uses a typing mechanism for socket to provide the semantic aspect of communication to application in a controlled and uniform manner. **IPC Primitives:**

1. S=socket(domain, type, protocol)

When a process wants to communicate with another process, it must first create a socket by using the socket system call. The first parameter specify the communication domain, second parameter specifies the socket type and third parameter specifies the communication protocol.

2. Bind(s, addr, addrlen)

After creating a socket, the receiver must bind it to a socket address. If two-way communication is desired between two processes, both processes have to receive messages, and hence both must separately bind their sockets to a socket address.

3. Connect(s, server_addr, server_addrlen)

In connection-based communication, two processes first establish a connection between their paires of sockets. The connection establishment pricess is asymmetric because one of the processes keeps waiting for a request for a connection and the other makes a request for a connection.

4. Listen(s, backlog)

The listen system call is used in case of connection-based communication by a server process to listen in its socket for client requests for connections.

5. Snew=accept(s, client_addr, client_addrlen)

The accept system call is used in a connection-based communication by a server process to accept a request for a connection establishment made by a client and to obtain a new socket for a communication with that client.

6. Primitives for Sending and Receiving Data:

nbytes= read(snew, buffer, amount) write(s,
“message”, msg_length) amount=
recvform(s, buffer, sender_address)
sendto(s, “message”, receiver_address)

Conclusion:

In this Practical, we studied about Message Passing Interface.

DOC: 02/10/2021

……

Mrs. Madhavi Kale

Name & Sign of Course Teacher

GOVERNMENT COLLEGE OF ENGINEERING,JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Krushna Shivaji Sase PRN No: 1941047

Class: T. Y. B.Tech. Computer Batch: T3

Date of Performance: 02/10/2021 Date of Completion: 02/10/2021

Subject:DSL Sign. of Teacher with Date: _____

Experiment No.: 04

Aim: Design a distributed application using Message Passing Interface (MPI)
Title: Practical for a distributed application using Message Passing Interface (MPI) for remote computation where client submits a string to the server and server returns the reverse of it to the client.

Theory:

Message Passing Interface (MPI)

Objective:

Message Passing Interface (MPI) is a standardized and portable message- passing system distributed and developed for parallel computing. MPI provides parallel hardware vendors with a clearly defined base set of routines that can be efficiently implemented.

The message passing interface (MPI) is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory.

In parallel computing, multiple computers -- or even multiple processor cores within the same computer -- are called nodes. Each node in the parallel arrangement typically works on a portion of the overall computing problem. The challenge then is to synchronize the actions of each parallel node, exchange data between nodes and provide command and control over the entire parallel cluster. The message passing interface defines a standard suite of functions for these tasks.

FEATURES:

1. General
   - Communicators combine context and group for message security
   -Thread safety
2. Point-to-point communication
   -Structured buffers and derived datatypes, heterogeneity
   - Modes: normal (blocking and non-blocking), synchronous, ready (to allow access to

fast protocols), buffered

3. Collective

- Both built-in and user-defined collective operations
- Large number of data movement routines - Subgroups defined directly or by topology

4. Non-message-passing concepts not included:

· process management

· remote memory transfers

· active messages

· threads

· virtual shared memory

5. MPI does not address these issues, but has tried to remain compatible with these ideas (e.g. thread safety as a goal, intercommunicators)

Working of MPI in Distributed Computing:

· **Distriuted Memory**

Every processor has its own local memory which can be accessed directly only by its own CPU. Transfer of data from one processor to another is performed over a network. Differs from shared memory systems which permit multiple processors to directly access the same memory resource via a memory bus.

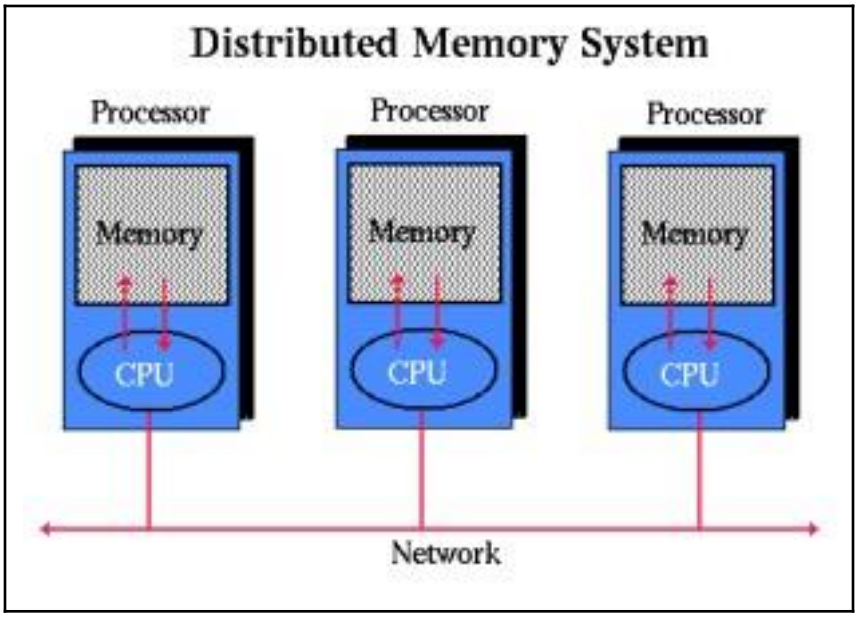


Figure: Message Passing Interface in DS

· **Message Passing**

The method by which data from one processor's memory is copied to the memory of another processor. In distributed memory systems, data is generally sent as packets of information over a network from one processor to another. A message may consist of one or more packets, and usually includes routing and/or other control information.

· **Processor**

A process is a set of executable instructions (program) which runs on a processor. One or more processes may execute on a processor. In a message passing system, all processes communicate with each other by sending messages - even if they are running on the same processor. For reasons of efficiency, however, message passing systems generally associate only one process per processor.

· **Message Passing Library**

Usually refers to a collection of routines which are imbedded in application code to accomplish send, receive and other message passing operations.

· **Send / Receive**

Message passing involves the transfer of data from one process (send) to another process (receive). Requires the cooperation of both the sending and receiving process. Send operations usually require the sending process to specify the data's location, size, type and the destination. Receive operations should match a corresponding send operation.

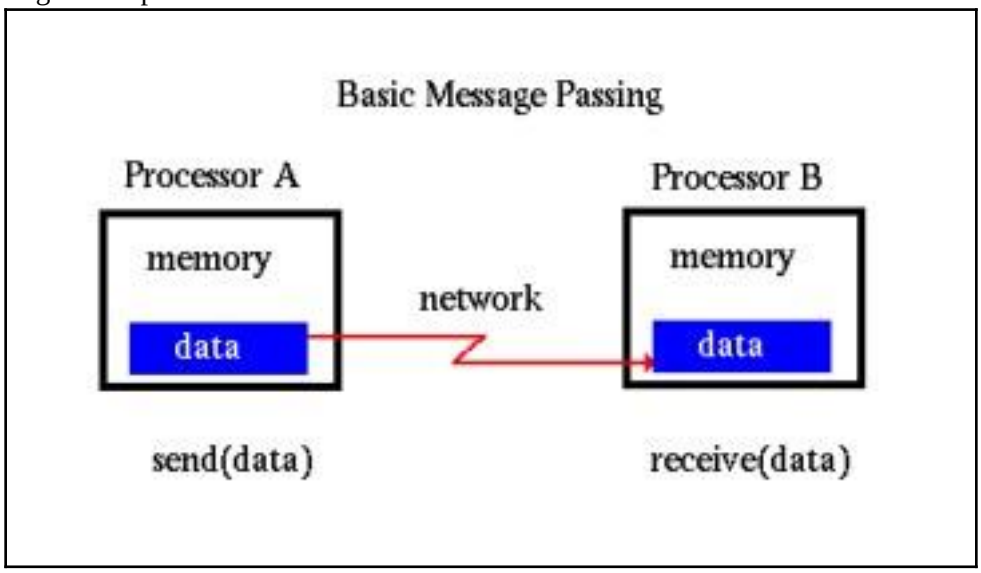


Figure: Send & Receive Msg in MPI in Network

· **Synchronous / Asynchronous**

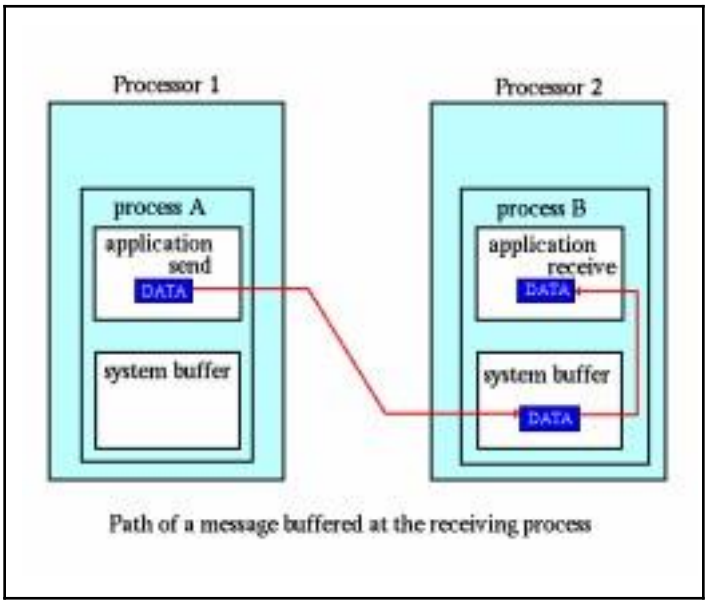
A synchronous send operation will complete only after acknowledgement that the message was safely received by the receiving process. Asynchronous send operations may "complete" even though the receiving process has not actually received the message.

· **Application Buffer**

The address space that holds the data which is to be sent or received. For example, your program uses a variable called, "in msg". The application buffer for in msg is the program memory location where the value of in msg resides.

· **System Buffer**

System space for storing messages. Depending upon the type of send/ receive operation, data in the application buffer may be required to be copied to/from system buffer space. Allows communication to be asynchronous.



· **Blocking Communication**

A communication routine is blocking if the completion of the call is dependent on certain "events". For sends, the data must be successfully sent or safely copied to system buffer space so that the application buffer that contained the data is available for reuse. For receives, the data must be safely stored in the receive buffer so that it is ready for use.

· **Non-blocking Communication**

A communication routine is non-blocking if the call returns without waiting for any communications events to complete (such as copying of message from user memory to system memory or arrival of message).

It is not safe to modify or use the application buffer after completion of a non-blocking send. It is the programmer's responsibility to insure that the application buffer is free for reuse.

Non-blocking communications are primarily used to overlap computation with communication to effect performance gains.

Advantage of MPI

1. Standardization: MPI is the only message passing library which canbe considered a standard.
2. Portability: no need to modify your source code when you portyour application to a different platform.
3. Functionality: Many routines available to use.
4. Availability: A variety of implementations are available. **Disadvantages of MPI:**

1. Lack of scalability between memory and CPU. Adding more CPUs can increase the traffic on shared memory associated with cache andmemory management.

2. Programmer responsibility for synchronization construct that ensure correct access of global memory.

3. Expense: It becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing number of processors. **Steps to Implement Interface**

Server site:

1. # open terminalsudo apt-get update
sudo apt-get install libopenmpi-dev
mkdir exp3
cd exp3 gedit
server.c
2. pass the variablesMPI_Comm client;
MPI_Status status;
char port_name[MPI_MAX_PORT_NAME],
str[50],ch,temp;
int size, again, i,j;
3. MPI_Open_port(MPI_INFO_NULL, port_name);
printf("Server available at port: %s\n", port_name);
4. send the string at server site
5. send the reversed string to client (character by character) ,end the string(tag=1)
6. MPI_Comm_disconnect(&client);
7. # save and exit the file

Client site:

8. gedit client.c
9. accept input string
10. Receive the reversed string from server and display it
11. printf("\nReversed string is : %s\n\n",str);
12. MPI_Comm_disconnect(&server);
13. # save and exit the file
14. Compile and run

Conclusion: Hence We Sucessfully Study a distributed application using Message Passing Interface (MPI)

……...……
…….....

DOC:8/4/21

……

Ms. Madhavi Kale.

Name & Sign of Course Teacher

Server Code:

```
import java.sql.*;
import java.sql.Connection;
import java.rmi.*;
import java.rmi.Naming.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.Vector;
interface DBInterface extends Remote
 {
 public String input(String name1) throws RemoteException; }
```

```
public class Server extends UnicastRemoteObject implements DBInterface
 {
 int flag=0,n,i,j;
 String name3;
 ResultSet r;
 public String reverse_str(String s1){
 char ch[]=s1.toCharArray();
 String rev="";
 for(int i=ch.length-1;i>=0;i--){
 rev+=ch[i];
  }
 return rev;
  }
 public Server()throws RemoteException
  {
 try
  {
 System.out.println("InitializingServer\nServerReady"); }
 catch(Exception e)
  {
 System.out.println("ERROR:"+e.getMessage());
  }
  }
 public static void main(String []args)
  {
 try
  {
 Server rs=new Server();
java.rmi.registry.LocateRegistry.createRegistry(1030).rebind("DBServ",
rs); } catch(Exception e)
  {
 System.out.println("ERROR:"+e.getMessage());
  }
  }
 public String input(String name1)
  {
 try
  {
 name3=reverse_str(name1);
 //name3=name1.concat(name2);
  }
 catch(Exception e)
  {
 System.out.println("ERROR:"+e.getMessage());
```

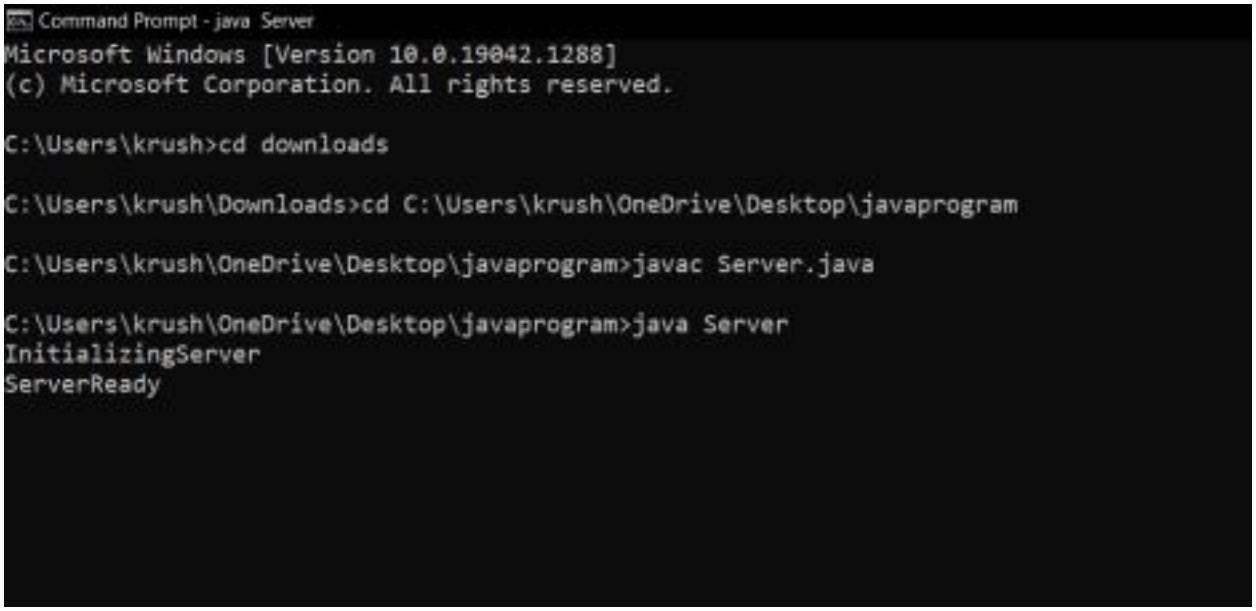
```
 }
 return name3;
 }
 }
```

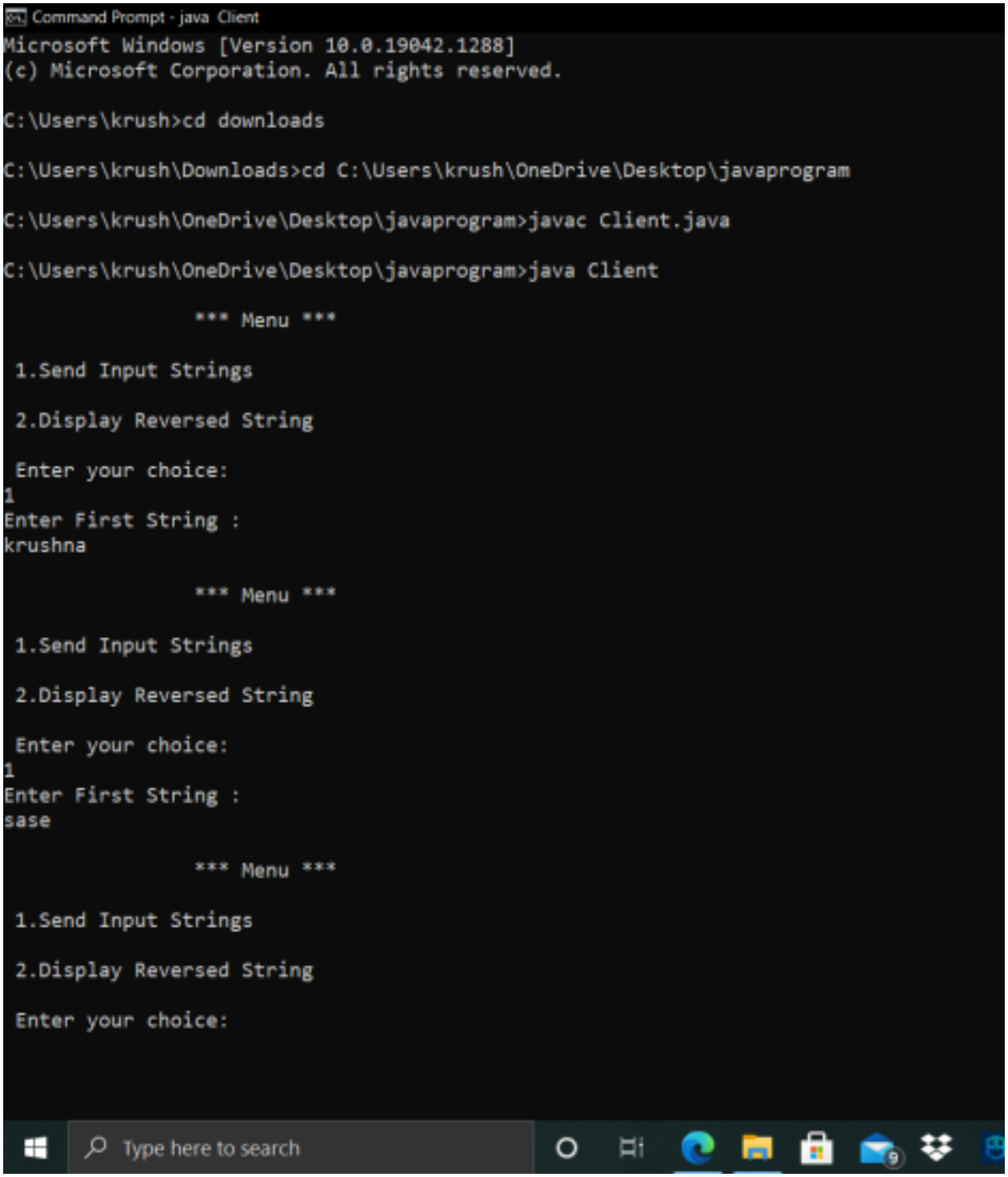
Client Code:

```
import java.sql.*;
import java.rmi.*;
import java.io.*;
import java.util.*;
import java.util.Vector.*;
import java.lang.*;
import java.rmi.registry.*;
public class Client
 {
 static String name1,name2,name3;
 public static void main(String args[])
 {
 Client c=new Client();
 BufferedReader b = new BufferedReader(new
InputStreamReader(System.in)); int ch;
 try
 {
 Registry r1 = LocateRegistry.getRegistry("localhost",1030);
 DBInterface DI = (DBInterface)r1.lookup("DBServ");
 do
 {

 // System.out.println("\n:Menu:" + "\n1.Send Input Strings\n2.Display
ConcatenatedString\n" + "\nEnteryourchoice:");
 System.out.println("\n\t\t*** Menu ***");
 System.out.println("\n 1.Send Input Strings");
 System.out.println("\n 2.Display Reversed String ");
 System.out.println("\n Enter your choice:");

 ch=Integer.parseInt(b.readLine());
 switch(ch)
 {
 case 1:
 {
 System.out.println("Enter First String :");
 name1=b.readLine();
 name3=DI.input(name1);
```

```
break;
 }
case 2:
{System.out.println("\nReversed String = "+name3);
// int i=0;
                        // System.out.println(""+name3+"");
break;}
 }
}while(ch > 0);
}
catch(Exception e)
{
System.out.println("ERROR:"+e.getMessage());
}
}
}
```

Output:




Command Prompt - java Client
Microsoft Windows [Version 10.0.19042.1288]
(c) Microsoft Corporation. All rights reserved.
C:\Users\krush>cd downloads
C:\Users\krush\Downloads>cd C:\Users\krush\OneDrive\Desktop\javaprogram
C:\Users\krush\OneDrive\Desktop\javaprogram>javac Client.java
C:\Users\krush\OneDrive\Desktop\javaprogram>java Client
*** Menu ***
1.Send Input Strings
2.Display Reversed String
Enter your choice:
1
Enter First String :
krushna
*** Menu ***
1.Send Input Strings
2.Display Reversed String
Enter your choice:
1
Enter First String :
sase
*** Menu ***
1.Send Input Strings
2.Display Reversed String
Enter your choice:
Type here to search

GOVERNMENT COLLEGE OF ENGINEERING,JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Krushna Shivaji Sase PRN No: 1941047

Class: T. Y. B.Tech. Computer Batch: T3

Date of Performance: 16/10/2021 Date of Completion: 16/10/2021

Subject:DSL Sign. of Teacher with Date: _____

Experiment No.: 05

Aim: Design a distributed application which consist of a server and client using threads

Title: Program for Distributed Application which consist of a server and client

using threads **Software Requirements**: Ubuntu OS, Eclipse IDE 4.11 **Theory:**

- **PROCESS:**

In a conventional centralized operating system process management deals with mechanisms and policies for sharing the processor the system among all processes .Similarly ,in a distributed operating system ,the main goal-of process management to make the best possible use of the processing resources of the entire system by sharing the among all processes. Three important concepts are used in distributed operating systems to achieve this goal:

1. Processor allocation:

Deals with the process of deciding which process should be assigned to which processor.

2. Process migration:

Deals with the movement of a process from its current location of the processor to which has been assigned. **3. Threads:**

Deals with fine-grained parallelism for better utilization of the processing capability of the system. The processor all location concept ready been described in the previous chapter on resource management. Therefore, this chapter presents a description of the process migration and threads concepts.

- **THREAD:**

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a lightweight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

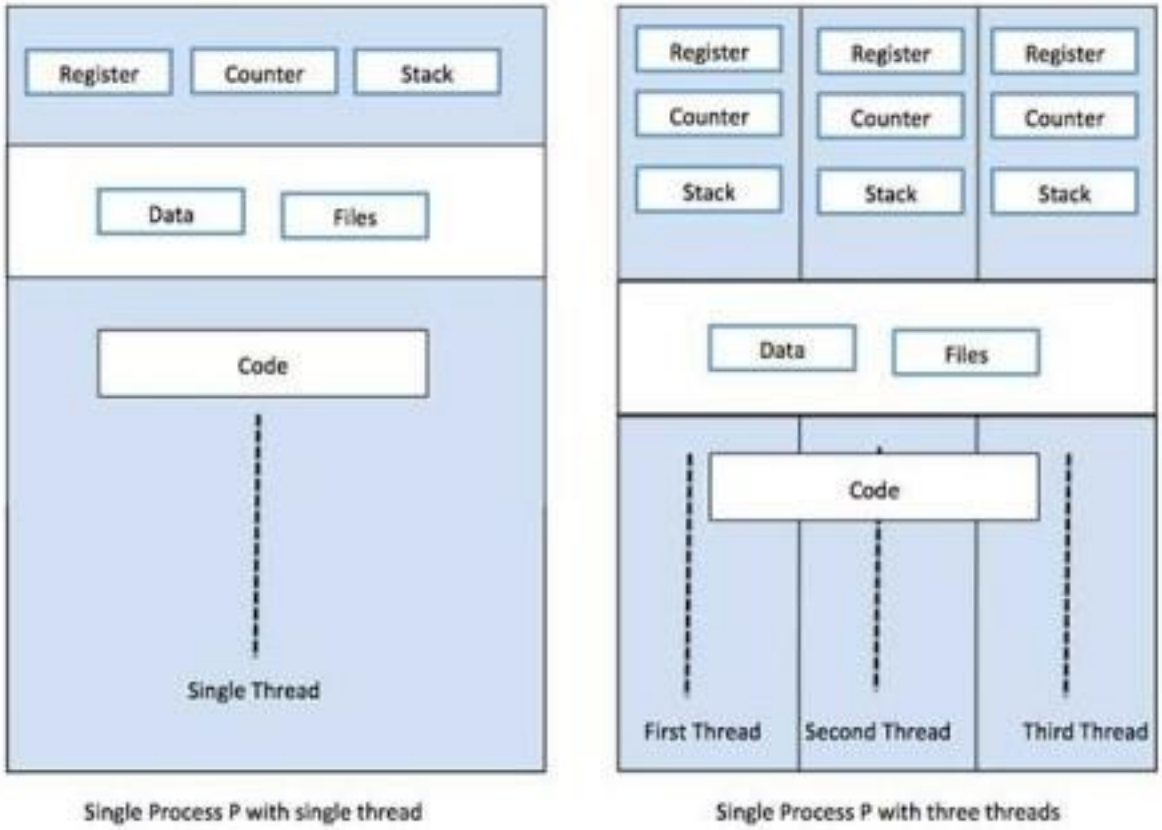


Figure 1: thread concept with Process

ADVANTAGES OF THREAD

1. Threads minimize the context switching time.
2. Use of threads provides concurrency within a process.
3. Efficient communication.
4. It is more economical to create and context switch threads.
5. Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

MULTITHREADING:

Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running in the computer. Each user request for a program or system service (and here a user can also be another program) is kept track of as a thread with a separate identity. As programs

work on behalf of the initial request for that thread and are interrupted by other requests, the status of work on behalf of that thread is kept track of until the work is completed.
The user threads must be mapped to kernel threads, by one of the following strategies:

1. Many to One Model
2. One to One Model
3. Many to Many Model

1. Many to One Model

1. As shown in figure 2, In the **many to one** model, many user-level threads are all mapped onto a single kernel thread.
2. Thread management is handled by the thread library in user space, which is efficient in nature.

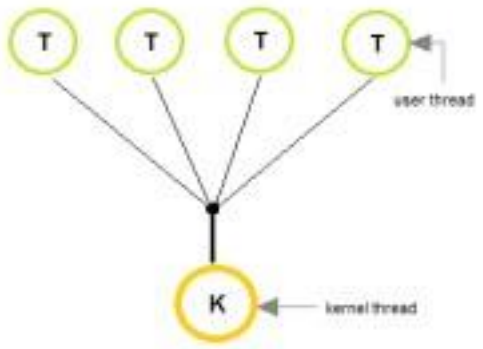
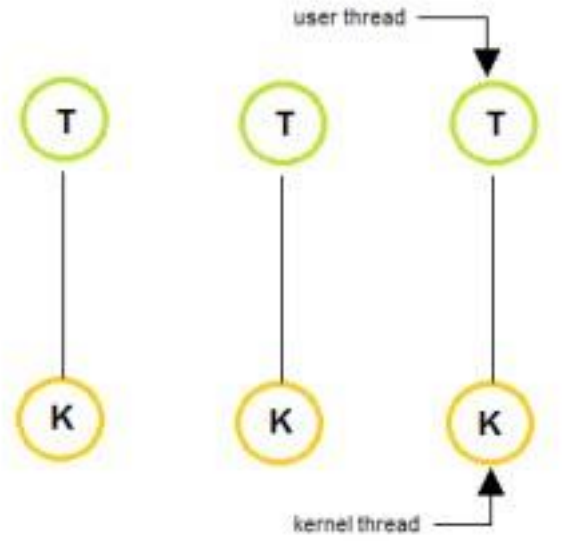


Figure 2: Many to one model

2 .One to One Model:

1. As shown in figure 3 The **one to one** model creates a separate kernel thread to handle each and every user thread.
2. Most implementations of this model place a limit on how many threads can be created.



3 .Many to Many Model

1. The many to many model multiplexes any number of user threads onto an equal or smaller
3. Linux and Windows from 95 to XP implement the one-to-one model for threads.

Difference between Process and

Thread:

Figure 3: One to one model

number of kernel threads, combining

the best features of the one-to-one and

many-to-one models shown in figure 4.

2. Users can create any number of the

threads. 3. Blocking the kernel system

calls does not block the entire process.

4. Processes can be split across multiple processors.

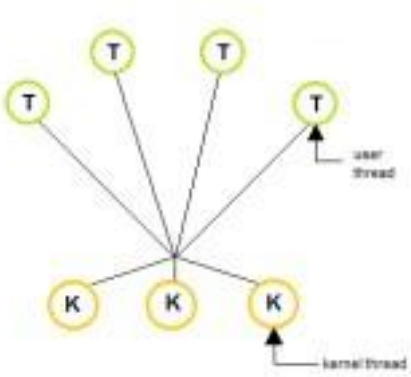


Figure 4- Many to many model

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Methods Used:

1. **public Socket(String host, int port) throws UnknownHostException, IOException.**

This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.

2. **public InputStream getInputStream() throws IOException**

Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket. 3. **public OutputStream getOutputStream() throws IOException**

Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket.

4. **public void close() throws IOException**

Closes the socket, which makes this Socket object no longer capable of connecting again to any server. 5. **public ServerSocket(int port) throws IOException**

Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.

6. **public Socket accept() throws IOException**

Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method.

ALGORITHM:

Steps

Client Program:

1. Initialise input and output stream.
2. Open socket on given port.
3. Open output and input stream.
4. If everything has been initialized then we want to write some data to the socket we have opened aconnection to on the given port
5. Close input and output stream.
6. Close the socket

Server Program:

1. Calling constructor of server final and initializing in object(server) of serverfinal.
2. Calling startserver function
3. Declare a server socket and a client socket for the server
4. Declare the number of connections
5. Try to open a server socket on the given port(Note that we can't choose a port less than 1024 if we are not) 6. Whenever a connection is received, start a new thread to process the connection and wait for the next connection.
7. Close the client connection.

Conclusion:

In this Practical, We learnt about development of distributed application consisting server and client using threads.

DOC: 06/04/2021

……..

Mrs. Madhavi Kale

Name & Sign of Course Teacher

ClientFinal.java

```
import java.io.*;
import java.net.*;

public class Clientfinal
{ public static void main(String[]
args)
{

String hostname = "localhost"; //hostname
int port = 6789; //port numbrt

        // declaration section:
        // clientSocket: our Clientfinal socket
// os: output stream
        // is: input stream
```

```
Socket clientSocket = null;

DataOutputStream os = null;
BufferedReader is = null;

        // Initialization section:
// Try to open a socket on the given port
// Try to open input and output streams

try
{
clientSocket = new Socket(hostname, port);
//initialising client socket os = new
DataOutputStream(clientSocket.getOutputStream()); //output stream
of client socket is = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream())); //input stream of client socket
}
catch (UnknownHostException e)
{
```



```
System.err.println("Don't know about host: " + hostname); }
catch (IOException e)
{
System.err.println("Couldn't get I/O for the connection to: " + hostname);
}

// If everything has been initialized then we want to write some data // to the socket we have
opened a connection to on the given port

        if (clientSocket == null || os == null || is == null) {
System.err.println( "Something is wrong. One variable is null." );  return;
} try { while
( true )
        {
System.out.print( "Enter an integer (0 to stop connection, -1 to stop server): " );
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));//taking input
from terminal  String keyboardInput = br.readLine(); //storing input in keyboard os.writeBytes(
keyboardInput + "\n" );
//writing keyboard into input stream buffer of server

int n = Integer.parseInt( keyboardInput
); if ( n == 0 || n == -1 ) {
```

```
//condiition to get out from loop
break;
                    }

 String responseLine = is.readLine();
//getting input stream in response line
System.out.println("Server returns its square as: " + responseLine);  }
```



```
 // clean up:
 // close the output stream
 // close the input stream
 // close the socket

 os.close();
 is.close();
 clientSocket.close()
 ;
 }
catch (UnknownHostException e)
{
 System.err.println("Trying to connect to unknown host: " + e);  }
catch (IOException e)
 {
 System.err.println("IOException: " + e);
 }
 }
}
```

ServerFinal.java

```
import java.io.*;
import java.net.*;

public class Serverfinal
{

 public static void main(String args[])
          {
```

```
        int port = 6789;//port number
        Serverfinal server = new Serverfinal( port );
//calling constructructor of server final and initialisig in object(server) of serverfina;
server.startServer(); //calling startserver function }
```



```
// declare a server socket and a client socket for the server;  // declare the number of
connections

ServerSocket echoServer = null; //declaring echo server  Socket clientSocket = null;
//declaring client socket  int numConnections = 0;
int port;

public Serverfinal( int port )
        { this.port = port;
        //port is initialised in member variable port
}

public void stopServer() //Switching off the server
{
        System.out.println( "Server cleaning up." );
        System.exit(0);
}

public void startServer() //starting the server
{
        // Try to open a server socket on the given port
        // Note that we can't choose a port less than 1024 if we are not // privileged users (root)

try
        {
echoServer = new ServerSocket(port);
                //initialising server by calling server socket
                function
}
catch (IOException e)

        {
        //if error persist
System.out.println(e);
}

System.out.println( "Server is started and is waiting for connections." );
```

```
System.out.println( "With multi-threading, multiple connections are allowed." );
System.out.println( "Any client can send -1 to stop the server.");

// Whenever a connection is received, start a new thread to process the connection and wait for the
next connection.

 while ( true )
{
 try
          {
 clientSocket = echoServer.accept();
          //clientsocket is intialised
 numConnections ++;
          //number of connectections is increament
    Server2Connection oneconnection = new  Server2Connection(clientSocket, numConnections,
                                                    this)
; //caling constructor of server2 funcction new
Thread(oneconnection).start();
          //run function is called f or each thread
 }
 catch (IOException e)
          {
 System.out.println(e);
 }
 }
 }
 }

class Server2Connection implements Runnable
 {
 BufferedReader is;
 PrintStream os;
 Socket clientSocket;
 int id;
 Serverfinal server;

public Server2Connection(Socket clientSocket, int id, Serverfinal server) {
          this.clientSocket = clientSocket;
```

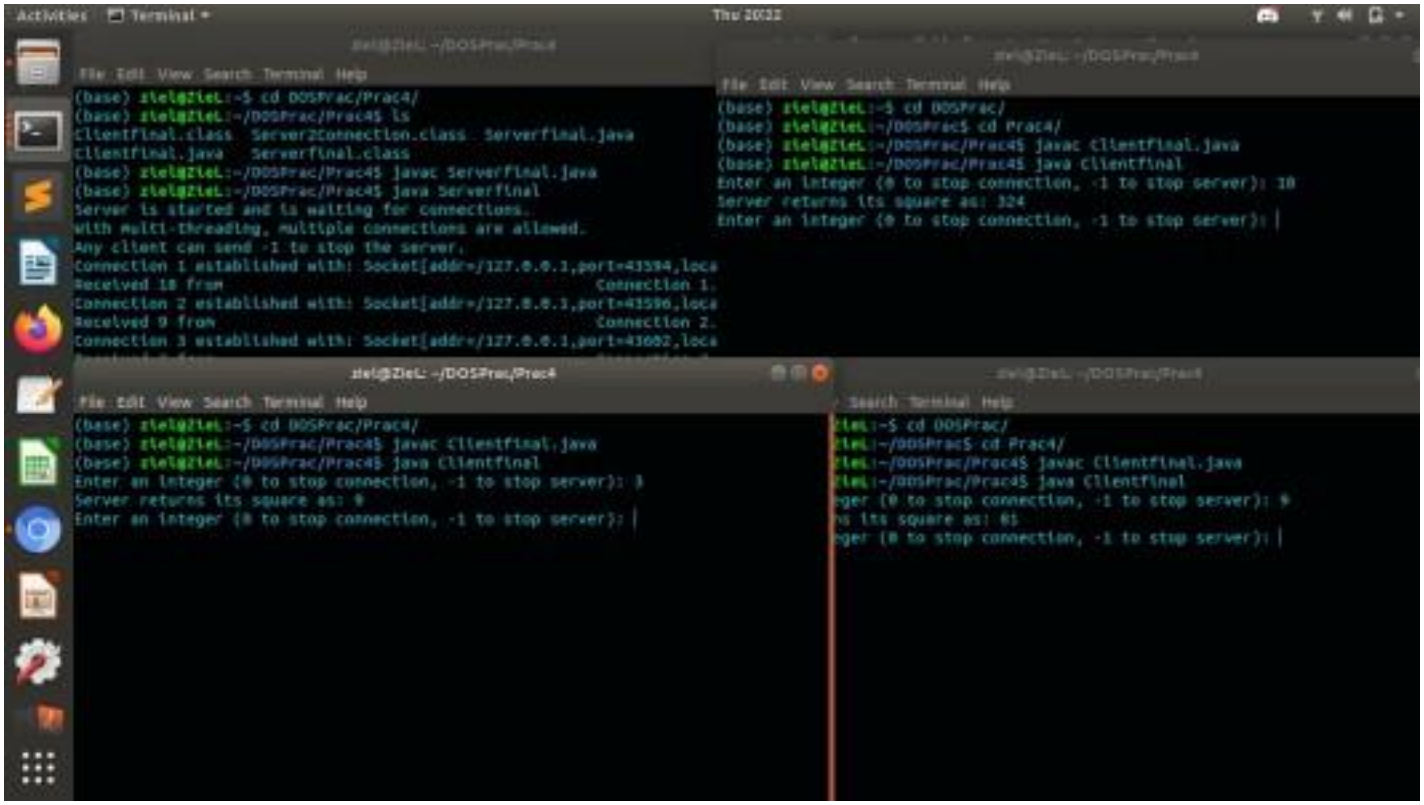
```
        this.id = id;
        this.server = server;
System.out.println( "Connection " + id + " established with: " + clientSocket );
        try
{
is = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
                //data input stream of server
 os = new PrintStream(clientSocket.getOutputStream()); //output stream of serve
 }
catch (IOException e)
 {
 System.out.println(e);
 }
 } public void
 run()
 {
 String line;
         try
         {
 boolean serverStop = false;

 while (true)
                {
  line = is.readLine(); //getting line from client  System.out.println( "Received " + line + "
                    from  Connection " + id + "." );//displaying
 int n = Integer.parseInt(line); if (
                        n == -1 )
                        {
                                                //checking if to stop the server or client
 serverStop = true;
 break;
 } if ( n == 0 ) break;
 os.println("" + n*n
 );
        //writing square in input buffer of client
 }
```



```
 System.out.println( "Connection " + id + " closed." );  //closing client
 connection is.close(); os.close(); clientSocket.close();

 if ( serverStop ) server.stopServer();
```

```
        //this is used to close all the connection
} catch (IOException e)
{
System.out.println(e);//if error occured
}
}
}
```

OUTPUT



GOVERNMENT COLLEGE OF ENGINEERING,JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Krushna Shivaji Sase PRN No: 1941047

Class: T. Y. B.Tech. Computer Batch: T3

Date of Performance: 23/10/2021 Date of Completion: 23/10/2021

Subject:DSL Sign. of Teacher with Date: _____

Experiment No.: 06

Aim: Program for Distributed Application using RMI

Title: Design and develop a distributed Hotel booking application using Java RMI. A distributed hotel booking system consists of the hotel server and the client machines. The server manages hotel rooms booking information. A customer can invoke the following operations at his machine

i) Book the room for the specific guest
ii) Cancel the booking of a guest

Software Requirements : Ubuntu OS, Eclipse IDE 4.11

Theory :

Java RMI - Introduction

RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

Architecture of an RMI Application

In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).

- The client program requests the remote objects on the server and tries to invoke its methods. The following Figure 1.1 shows the architecture of an RMI application.

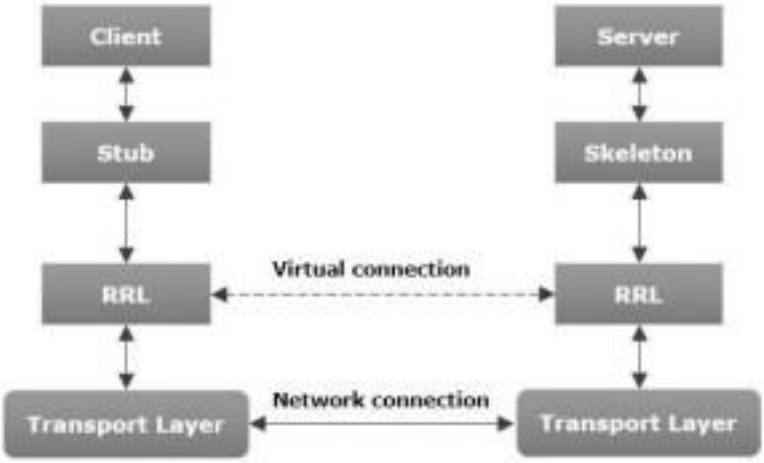


Figure 1.1 - Architecture of an RMI application

The components of this architecture are :

- **Transport Layer** − This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton** − This is the object which resides on the server side. **Stub** communicates with this skeleton to pass request to the remote object.
- **RRL (Remote Reference Layer)** − It is the layer which manages the references made by the client to the remote object.

Working of an RMI Application

The following points summarize how an RMI application works −

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.

- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using **bind()** or **reBind()** methods). These are registered using a unique name known as **bind name**.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using **lookup()** method).

The following illustration explains the entire process −

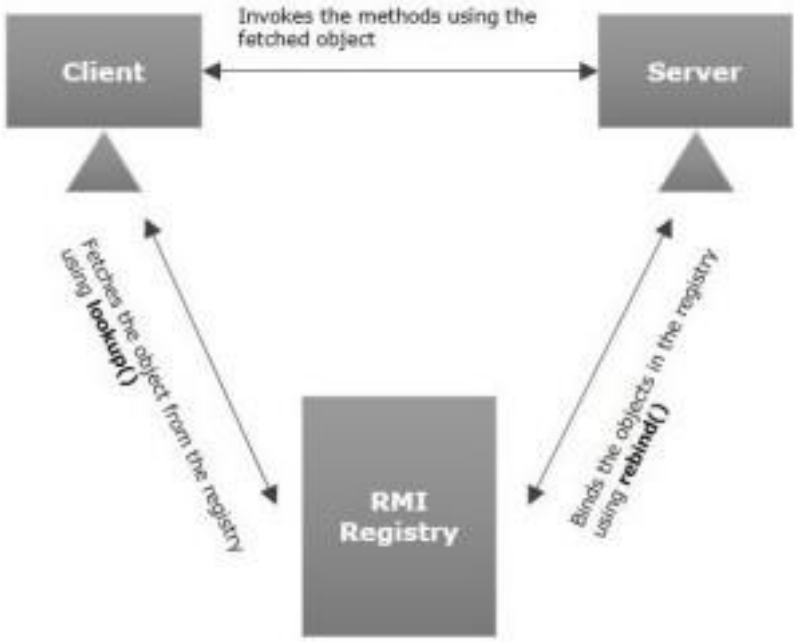


Figure 1.2 - Entire process RMI application

Goals of RMI

Following are the goals of RMI −

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects.

Java RMI Application

To write an RMI Java application, you would have to follow the steps given below −

- Define the remote interface

- Develop the implementation class (remote object)
- Develop the server program
- Develop the client program
- Compile the application
- Execute the application.

Programming Requirement: You are required to write this client-server hotel booking system program, which communicates via RMI. Specifically, your program should consist of two parts: one for the client and another for the server. The client (the customer) will initiate a request by sending request message to the hotel server to execute a specified procedure (e.g. booking) with parameters.

Required files: Included are all files needed to run this application. These files are:

1. RoomBookingClient.java
2. RoomBookingServer.java
3. RoomBookingInterface.java
4. Room.java
5. RoomList.java
6. Connect.policy
7. Rooms.txt
8. rooms.txt

Design Requirement: You can use the following interface and class definitions as a starting point for your project. However, you are free to develop your own interface and class definitions.

1- Interface public interface HotelInterface extends Remote

```
{
public void book (int NumberOfGuests, int NumberOfNights, String
CustomerName, Date checkInDate)throws RemoteException; public boolean
cancelBooking(String CustomerName) throws RemoteException; public Date
inquiry(String CustomerName) throws RemoteException;
}
```

2- Server public class HotelServer extends UnicastRemoteObject implements HotelInterface

```
{
```

```
private Vector Customers; //guests' name private Vector
CheckInDate; //guests' check-in Date private Vector
NumOfNights; //guests' number of nights staying
```



```
private Vector NumbOfGuest; // number of guests boolean done;

public HotelServer() throws RemoteException
{ //implementation
}
public void book (int NumberOfGuests, int NumberOfNights, String CustomerName, Date checkInDate) throws RemoteException
{ //implementation
}
public boolean cancelBooking(String CustomerName) throws RemoteException
{ //implementation
}
public Date inquiry(String CustomerName) throws RemoteException {
//implementation
}
public static void main(String args[]) throws Exception
{ //init Hotel server
}
}
```

The Hotel server has only one input parameter "server_port", which specifies the port of rmiregistry. The default port of rmiregistry is 1099, but we may have to use other ports, if 1099 has already been used by some other programs.

3- Customer (Client)

```
public class Customer
{
public static void (String args[]) throws Exception
{ //get user's input, and perform the operations
}
}
```

The input parameters of a Customer must include:

- server_address: the address of the rmiregistry
- server_port: the port of the rmiregistry
- Operation: one of "book", "cancelBooking", and "Inquiry"
- CustomerName: assume it is unique for every user
- CheckInDate: entered in the book operation
- NumberOfNights: entered in the book operation

- NumberOfGuests: entered in the book operation

Algorithm Steps to run Distributed Application:

1. Compile all java filesjavac *.java

2. Create Stub and Skeleton class file

for RoomBookingServerrmic

RoomBookingServer **3. Run RMIC Registry**

rmiregistry &

4. Run RoomBookingServerjava

RoomBookingServer

5. In new terminal Run

RoomBookingClientjava

RoomBookingClient

Note: Make sure that you are running the programs in same directory

Conclusion:

In this practical we learnt about Designing and Developing an Hotel Booking Distributed application using Java RMI.

DOC: 20/04/2021

……

Mrs. Madhavi Kale

Name & Sign of Course Teacher

RoomBookingClient.java

```
import java.rmi.*;
import
java.rmi.server.*;
import java.io.*;

class RoomBookingClient
{

/**
 *      This is the Client Class. It takes an input from the user, calls the
methodsavailable
 *      to the client from the server class and gives an ouput depending on
theoperation performed.
*/

public static boolean validChoice = true;
static String [] daysOfWeek = { "Monday |", "Tuesday |", "Wednesday|", "Thursday
|" , "Friday |" , "Saturday |" , "Sunday |" };

public static void main (String[] args)
```



```
{
try
{
```

```
//System.setSecurityManager ( new RMISecurityManager ( )); //set up the security
manager

 //String name = "rmi://localhost:9999/RoomBookingSystem"; //connect on local
host on port 9999

String name="rmi://127.0.0.1/RoomBookingSystem";

RoomBookingInterface rbi =(RoomBookingInterface) Naming.lookup (name);


                                rbi.initRooms(); //set up the room booking system


while( validChoice != false )

{

//A small command line interface for the user to use the system.

System.out.println(" ");

System.out.println("**********************Room Booking Service**********************");

System.out.println("");

System.out.println(" Please select a service");

System.out.println("");

System.out.println("1. List of all rooms.");

System.out.println("2. Check availability of a room.");
```



```
System.out.println("3. Book a room.");

System.out.println("4. Display weekly timetable for a room.");

System.out.println("5. Cancel The Booking");

System.out.println("");


//A buffered reader to allow input from the command line from the user.

BufferedReader input = new BufferedReader(new
InputStreamReader(System.in));
```

```
System.out.println("");
System.out.println("Select a number between 1 and 5, 0 to exit");
System.out.println("");
System.out.flush();
String response = input.readLine();

int i = Integer.parseInt(response);
RoomList ListOfAllRooms = new RoomList(); //RoomList Object which stores
//a list of all the rooms available.

try{
switch (i)
{
```



```
case 0: System.out.println("Goodbye"); //User has quit the application.
validChoice =false;
break;

case 1: System.out.println("");
System.out.println("The full list of rooms is as follows");
System.out.println("");
System.out.println("Room|Capacity");  System.out.println("----|--------");
ListOfAllRooms = rbi.allRooms(); //Run the allRooms method which
//ret urns the list of all rooms.
```

```
for(int c = 0; c < 100; c++) //Print the list.

{

if (ListOfAllRooms.RoomList[c] ==null)

{

break;

}

System.out.println(ListOfAllRooms.RoomList[c]);  }

System.out.println("");
```



```
break;


case 2: System.out.println("");

                                        System.out.println("Check a room");

System.out.println("Enter the room name");

                                         String check_room = input.readLine();


                                            System.out.println("Enter the day - ");

System.out.println("0=Mon , 1=Tues, 3=Wed ,4=Thurs , 5=Fri, 6=Sat, 7=Sun");

                                         String check_day = input.readLine();

int real_day = Integer.parseInt(check_day);


System.out.println("Enter the start time - ");

System.out.println("0=8am , 1=9am , 2=10am , 3=11am , 4=12pm , 5=1pm ,
6=2pm , 7=3pm , 8=4pm , 9=5pm , 10=6pm , 11= 7pm");

                                          String check_time = input.readLine();

int real_time = Integer.parseInt(check_time);
```

```
//This checks whether a room is available given the room name, day and time.

String temp = rbi.checkRoom(check_room,real_day,real_time);
```



```
System.out.println(temp);
System.out.println("");

break;


 case 3: System.out.println("Room Booking Service - Rooms can be booked from
8am to 8pm");

System.out.println("");

 System.out.println("Time slots go from 0 for 8am up to 11 for 7pm - Enter a value
in this range");

System.out.println("");

System.out.println("Enter the room name");

                                        String book_room = input.readLine();


System.out.println("");

                                            System.out.println("Enter the day -");

System.out.println("0=Mon , 1=Tues, 3=Wed ,4=Thurs , 5=Fri, 6=Sat, 7=Sun");

                                        String book_day = input.readLine();

int real_day2 = Integer.parseInt(book_day);


System.out.println("");

System.out.println("Enter the start time -");

System.out.println("0=8am , 1=9am , 2=10am , 3=11am , 4=12pm , 5=1pm ,
6=2pm , 7=3pm , 8=4pm , 9=5pm , 10=6pm , 11= 7pm");
```



```
                                        String book_time = input.readLine();

int realb_time = Integer.parseInt(book_time);
```

```
//This checks whether a room is available, if it is it then reserves the room.
String resp = rbi.bookRoom(book_room,real_day2,realb_time);
System.out.println(resp);
System.out.println("");
break;

case 4: System.out.println("Enter the Room name");
String Room1 = new String();
Room1 = input.readLine();

//This checks the timetable for a room. A 2D array containing
//the timetable is returned from the server.

System.out.println("TimeSlot | 0 1 2 3 4 5 6 7 8 9 10 11");
int rtt [][]=(int[] [])rbi.roomTimeTable(Room1).clone();
for(int f=0;f<7;f++)
{
```



```
System.out.println("");
                    System.out.print(daysOfWeek[f]);

for(int j=0;j<12;j++)
{
System.out.print(" ");
System.out.print(rtt[f][j]);  }

}
```

```
System.out.println("");

System.out.println(" ");

System.out.println("The key to start times is as follows... ");

System.out.println("0 = 8am , 1 = 9am , 2 = 10am , 3 = 11am , 4 = 12pm , 5 = 1pm
, 6 = 2pm , 7 = 3pm , 8 = 4pm , 9 = 5pm , 10 = 6pm , 11 =

7pm"); System.out.println(""); break;

 case 5: System.out.println("Room Booking Service - Rooms can be booked from
8am to 8pm");

System.out.println("");

 System.out.println("Time slots go from 0 for 8am up to 11 for 7pm - Enter a value
in this range");

System.out.println("");
```



```
System.out.println("Enter the room name");

String cancel_room = input.readLine();


System.out.println("");

System.out.println("Enter the day -");

System.out.println("0=Mon , 1=Tues, 3=Wed ,4=Thurs , 5=Fri, 6=Sat, 7=Sun");

                                        String can_day = input.readLine();

int can_day2 = Integer.parseInt(can_day);

System.out.println("");

System.out.println("Enter the start time -");

System.out.println("0=8am , 1=9am , 2=10am , 3=11am , 4=12pm , 5=1pm ,
6=2pm , 7=3pm , 8=4pm , 9=5pm , 10=6pm , 11= 7pm");

                                         String can_time = input.readLine();

int can2_time = Integer.parseInt(can_time);
```

```
String canc = rbi.cancelRoom(cancel_room,can_day2,can2_time);

System.out.println(canc);

System.out.println("");
```



```
break;



}

}

catch(Exception e)

{

 System.err.println("Sorry but you have entered one of the fields incorrectly, Please try again ");

}

}

}

catch (Exception ex)

{

System.err.println (ex);

}

}

}
```

RoomBookingServer.java

```
import java.io.*; import

java.rmi.*; import

java.rmi.server.*;
```

```
class RoomBookingServer extends UnicastRemoteObject implements
RoomBookingInterface

{
```



```
/**

 * This is the Server Class. It contains the working methods which can be used by
the client.

*/


protected int day; protected int

time; protected int room;

protected String str = new

String();


public String RoomListTemp [] = new String [100]; //Temporary store for list of

rooms

public String temp = new String(); public Room RoomArray[] = new


Room[100]; //Array of Room Objects RoomList tempList = new


RoomList();


public RoomBookingServer ( ) throws RemoteException  {

super ( );

}


/**
```

```
* This method is called once by the client when the application starts. It reads

* in the input from the text file and creates an Object for each room with the

* name and capacity that was specified in the file.  */


public void initRooms() throws RemoteException

{

String record = null;

String tempRoom = null;

String tempCap = null;


int recCount = 0;

int num;


int capacity;


try

{

//This reads in the text from the file and uses that to create the

//Room Objects. The name is specified first in the text file and the

//capacity is specified last. This is manipulated in order to take in

//these parameters when creating the Rooms.
```



```
BufferedReader b = new BufferedReader(new FileReader("Rooms.txt"));

while((record = b.readLine()) != null)
```

```
{ num = (record.lastIndexOf (" ", record.length ())) + 1; tempRoom =
record.substring (0,num -1); //Reads in the Room name from file

                                    tempCap = record.substring (num,record.length ());
capacity = Integer.parseInt(tempCap); //Reads in the capacity from file

 RoomArray[recCount] = new Room(tempRoom, capacity); //Fills the array with the
created Objects. recCount ++;
}
b.close(); //close the input stream.
} catch (IOException e)
{
System.out.println("Error!" +e.getMessage());  }
}
```



```
/**
* This method is used to return the list of rooms and there capacity to the client.*
  It returns a RoomList Object which contains the arrayList of Rooms. The Client
* can then retrieve a full list of rooms.
*/

public RoomList allRooms() throws RemoteException
{
try
{
```

```
 BufferedReader in = new BufferedReader(new FileReader("rooms.txt")); //read in the text file.

if((str = in.readLine()) != null)

{

tempList.RoomList[0] = str;

for (int i = 1; i< 100; i++)

{

if((str = in.readLine()) != null)

{

tempList.RoomList[i] = str;

}

}

}
```



```
in.close();

}

catch (IOException e)

{

}

return tempList;

}



/**

*	This method takes in a string and then compares that string with the name ofeach Object

*	in the array of Rooms. If it finds the room it returns the index, -1 otherwise.
```

```
*/


public int compareRoom(String str)

{

for(int i=0; i< RoomArray.length; i++)

{

if(RoomArray[i].name.equals (str))

{ return

i;

}

}
```



```
return -1;

}


/**

* This method is used to check whether a room is available or not. Firstly it checks

* for the room in the array, if it finds it it then checks whether the requested

* time slot on the requested day is available. It returns a string to the client

* depending on the value of the timeslot.

*/


public String checkRoom(String r ,int day , int startTime) throws RemoteException

{

int i = compareRoom(r);

 if (RoomArray[i].slotAvailable(day, startTime) == true) //calls methos available to Room Object
```

```
{

String s = "Room is available for booking";

return s;

}

else

{

String s = "Sorry the room is not available for booking";
```



```
return s;

}

}


/**

 *      This method is used to book a Room. Again it checks whether the slot
isavailable and depending

 *      on the result it reserves that slot and informs the client or it informs them that

 *      the slot has already been reserved.

*/


public String bookRoom(String r, int day , int startTime) throws RemoteException

{

int i = compareRoom(r);


if (RoomArray[i].slotAvailable(day, startTime) == true)  {

RoomArray[i].book(day,startTime);

String s = "Room has been successfully booked.";  return s;

}
```

```
else
{
```



```
String s = "Sorry but the Room has already been booked.";
return s;
}
}

public String cancelRoom(String r, int day, int startTime) throws RemoteException
{
int i = compareRoom(r); if
(RoomArray[i].slotAvailable(day, startTime) == false)  {
RoomArray[i].cancel(day,startTime);
String s = "Room has been successfully cancelled";  return s;
}
else
{
String s = "Sorry, You haven't booked this room.";  return s;
}
}

/**
```



```
* This method is used to calculate the timetable for each room. It returns relevant
* the 2D array to the client displaying the weekly timetable for the requested room.
*/
```

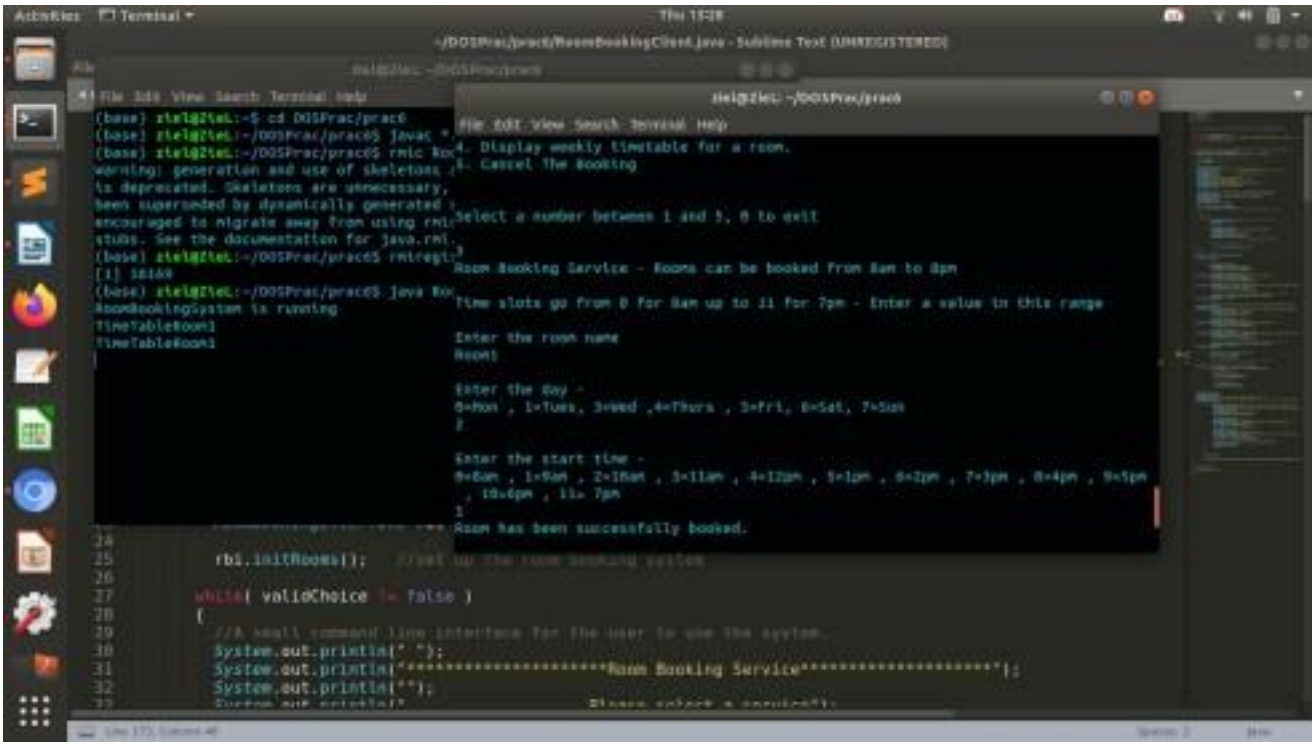
```
public int [][] roomTimeTable(String room) throws RemoteException
{ int
i;
System.out.println("TimeTable" + room);
for ( i = 0; i< RoomArray.length; i++)
{
if(RoomArray[i].name.equals(room))
{
return RoomArray[i].daySlot;
}
else
{
System.out.println("Searching for the room");  }
}

return RoomArray[i].daySlot;
}
```

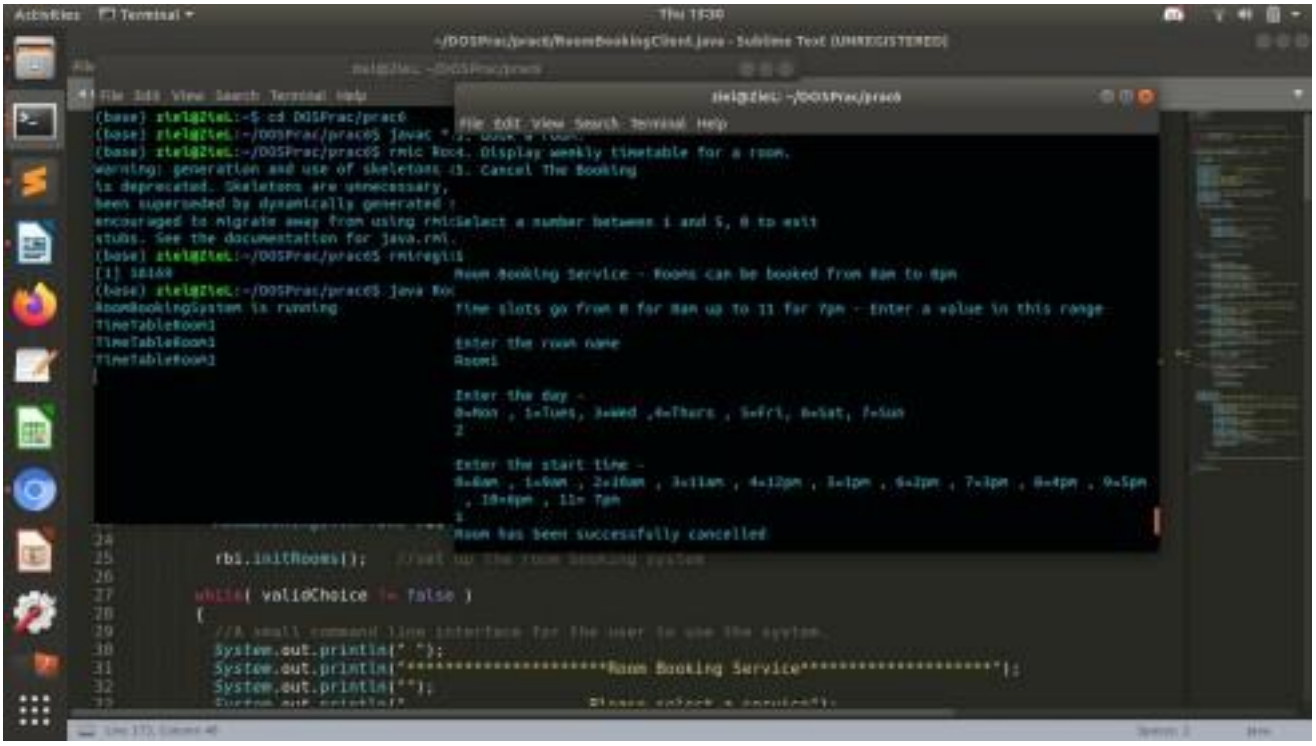


```
//Main Method public static void
main (String[] args)
{
try
{
                    RoomBookingServer server = new RoomBookingServer ();
//String name = "rmi://localhost:9999/RoomBookingSystem";
// Naming.bind (name, server);
```

```
String name = "RoomBookingSystem";
Naming.bind (name, server);
System.out.println (name + " is running");
}
catch (Exception ex)
{
System.err.println (ex);
}
}}
```

OUTPUT


Activities
Terminal
Thu 19:28
~/DOSPrac/prac6/RoomBookingClient.java - Sublime Text (UNREGISTERED)
File Edit View Search Terminal Help
(base) ztel@ZtelL:~$ cd DOSPrac/prac6
(base) ztel@ZtelL:~/DOSPrac/prac6$ javac *.
(base) ztel@ZtelL:~/DOSPrac/prac6$ rmic Roo
warning: generation and use of skeletons
is deprecated. Skeletons are unnecessary,
been superseded by dynamically generated
encouraged to migrate away from using rmi
stubs. See the documentation for java.rmi.
(base) ztel@ZtelL:~/DOSPrac/prac6$ rmiregis
[1] 18169
(base) ztel@ZtelL:~/DOSPrac/prac6$ java Roo
RoomBookingSystem is running
TimeTableRoom1
TimeTableRoom1
ztel@ZtelL: ~/DOSPrac/prac6
File Edit View Search Terminal Help
4. Display weekly timetable for a room.
5. Cancel The Booking
Select a number between 1 and 5, 0 to exit
3
Room Booking Service - Rooms can be booked from 8am to 8pm
Time slots go from 0 for 8am up to 11 for 7pm - Enter a value in this range
Enter the room name
Room1
Enter the day -
0=Mon , 1=Tues, 2=Wed ,4=Thurs , 5=Fri, 6=Sat, 7=Sun
2
Enter the start time -
0=8am , 1=9am , 2=10am , 3=11am , 4=12pm , 5=1pm , 6=2pm , 7=3pm , 8=4pm , 9=5pm
, 10=6pm , 11= 7pm
1
Room has been successfully booked.
rb1.initRooms();
while(validChoice != false)
//A small command line interface for the user to use the system.
System.out.println(" ");
System.out.println("**********Room Booking Service**********");
System.out.println("");


Activities
Terminal
Thu 19:30
~/DOSPrac/prac6/RoomBookingClient.java - Sublime Text (UNREGISTERED)
File Edit View Search Terminal Help
(base) ztel@ZtelL:~$ cd DOSPrac/prac6
(base) ztel@ZtelL:~/DOSPrac/prac6$ javac *.
(base) ztel@ZtelL:~/DOSPrac/prac6$ rmic Roo
warning: generation and use of skeletons
is deprecated. Skeletons are unnecessary,
been superseded by dynamically generated
encouraged to migrate away from using rmi
stubs. See the documentation for java.rmi.
(base) ztel@ZtelL:~/DOSPrac/prac6$ rmiregis
[1] 18169
(base) ztel@ZtelL:~/DOSPrac/prac6$ java Roo
RoomBookingSystem is running
TimeTableRoom1
TimeTableRoom1
TimeTableRoom1
ztel@ZtelL: ~/DOSPrac/prac6
File Edit View Search Terminal Help
4. Display weekly timetable for a room.
5. Cancel The Booking
Select a number between 1 and 5, 0 to exit
2
Room Booking Service - Rooms can be booked from 8am to 8pm
Time slots go from 0 for 8am up to 11 for 7pm - Enter a value in this range
Enter the room name
Room1
Enter the day -
0=Mon , 1=Tues, 2=Wed ,4=Thurs , 5=Fri, 6=Sat, 7=Sun
2
Enter the start time -
0=8am , 1=9am , 2=10am , 3=11am , 4=12pm , 5=1pm , 6=2pm , 7=3pm , 8=4pm , 9=5pm
, 10=6pm , 11= 7pm
1
Room has been successfully cancelled
rb1.initRooms();
while(validChoice != false)
//A small command line interface for the user to use the system.
System.out.println(" ");
System.out.println("**********Room Booking Service**********");
System.out.println("");

GOVERNMENT COLLEGE OF ENGINEERING, JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Krushna Shivaji Sase PRN No: 1941047

Class: T. Y. B.Tech. Computer Batch: T3

Date of Performance: 30/10/2021 Date of Completion: 30/10/2021

Subject:DSL Sign. of Teacher with Date: _____

Experiment No.: 07

Aim: Program for Distributed chat server using TCP sockets.

Title: Design a distributed Program for Distributed chat server using TCP sockets.

Software Requirements: Ubuntu OS, Eclipse IDE 4.11 **Theory :**

Client/Server Communication

At a basic level, network-based systems consist of a server, client, and a media for communication as shown in Fig. 1.1. A computer running a program that makes a request for services is called client machine. A computer running a program that offers requested services from one or more clients is called server machine. The media for communication can be wired or wireless network.

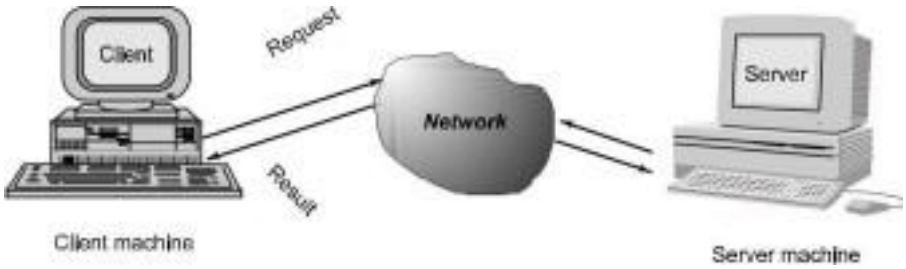


Fig.1.1 Client – Server communication

The client-server model is a distributed communication framework of network processes among service requestors, clients and service providers. The client-server connection is established through a network or the Internet.

The client-server model is a core network computing concept also building functionality for email exchange and Web/ database access. Web technologies and protocols built around the client-server model are:

- Hypertext Transfer Protocol (HTTP)
- Domain Name System (DNS)
- Simple Mail Transfer Protocol (SMTP)
- Telnet

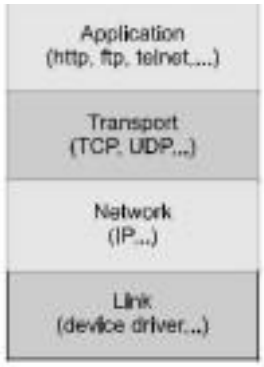
Clients include Web browsers, chat applications, and email software, among others. Servers include Web, database, application, chat and email, etc.

A server manages most processes and stores all data. A client requests specified data or processes. The server relays process output to the client. Clients sometimes handle processing, but require server data resources for completion.

The client-server model differs from a peer-to-peer (P2P) model where communicating systems are the client or server, each with equal status and responsibilities. The P2P model is decentralized networking. The client-server model is centralized networking.

One client-server model drawback is having too many client requests underrun a server and lead to improper functioning or total shutdown. Hackers often use such tactics to terminate specific organizational services through distributed denial-of-service (DDoS) attacks.

Generally, programs running on client machines make requests to a program (often called as server program) running on aserver machine. They involve networking services provided by the transport layer, which is part of the Internet software stack,often called *TCP/IP (Transport Control Protocol/Internet Protocol)*
stack, shown in Fig. 1.2. The transport layer comprises two types of protocols, *TCP (Transport Control Protocol)* and *UDP*
(User Datagram Protocol). The most widely usedprogramming interfaces for these protocols are sockets.



- TCP is a connection-oriented protocol that provides areliable flow of data between two computers. Example applicationsthat use such services are HTTP, FTP, and Telnet.

Fig 1.2 TCP/IP so ware stack

UDP is a protocol that sends independent packets of data, called *datagrams*, from one computer to another with no guarantees about arrival and sequencing. Example applications that use such services include Clock server and Ping. The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer. Port is represented by a positive (16-bit) integer value. Some ports have been reserved to support common/well known services:

∑ FTP 21/tcp
∑ telnet 23/tcp

∑ SMTP 25/tcp
∑ Login 513/tcp
∑ http 80/tcp,udp
443/tcp,ud
∑ https p

User-level process/services generally use port number value >= 1024. **Object-Oriented Programming with Java**

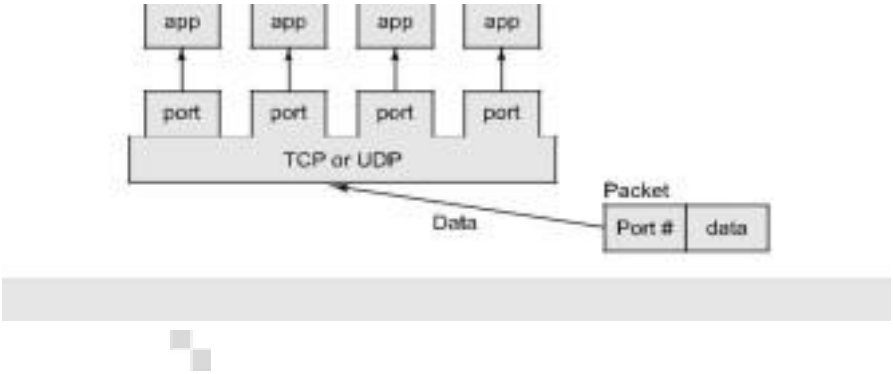


Fig. 1.3 TCP/UDP mapping of incoming packets to appropriate port/process

Object-oriented Java technologies—Sockets, threads, RMI, clustering, Web services—have emerged as leading solutions for creating portable, efficient, and maintainable large and complex Internet applications.

ALGORITHM:

Steps:

TCP Server

Step1: Create a socket
Step2: Bind it to the operating system. Step3:
Listen over it.
Step4: Accept connections.
Step5: Receive data from client and send it back to client.
Step6: Close the socket.

TCP Client Step1:
Create a socket.
Step2: connect to the server using connect().
Step3: send data to server and receive data from the server.
Step4: Close the socket.

Conclusion:

In this practical, We learnt about the implementation of a distributedchat server using

TCP sockets. **DOC: 27/04/2021**

.....

Mrs. Madhavi Kale

Name & Sign of Course Teacher

CLIENT.java

```
/*
 *   To change this license header, choose License Headers in Project Properties.
 *   To change this template file, choose Tools | Templates  * and open thetemplate in
the editor.
 */
import java.io.BufferedReader;
import
java.io.InputStreamReader;
import java.io.PrintStream; import
```

```
java.net.ServerSocket; import

java.net.Socket;

/**

 *

  *   @author Ayush

 */ public class Client { public static void main(String

args[]) throws Exception {

            Socket sk=new Socket("127.0.0.1",5000);

            BufferedReader sin=new BufferedReader(new
InputStreamReader(sk.getInputStream()));

                        PrintStream sout=new PrintStream(sk.getOutputStream());

            BufferedReader stdin=new BufferedReader(new
InputStreamReader(System.in));

            String s; while ( true )

            {

                  System.out.print("Client : ");

                  s=stdin.readLine();

                  sout.println(s);

 if ( s.equalsIgnoreCase("BYE") )  {

 System.out.println("Connection ended by client");

                   break;

 }

                  s=sin.readLine();

                        System.out.print("Server : "+s+"\n");


            } sk.close();

            sin.close();
```

```
            sout.close();
            stdin.close()
            ;
    }

}
```

SERVER.java

```
/*
 *  To change this license header, choose License Headers in Project Properties.
 *  To change this template file, choose Tools | Templates  * and open thetemplate
in the editor.
 */
import java.io.BufferedReader;
import java.io.IOException; import
java.io.InputStreamReader;
import java.io.PrintStream;

import java.net.ServerSocket; import
java.net.Socket; import java.util.Scanner;
import
java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.logging.Level; import
java.util.logging.Logger;
/**
```

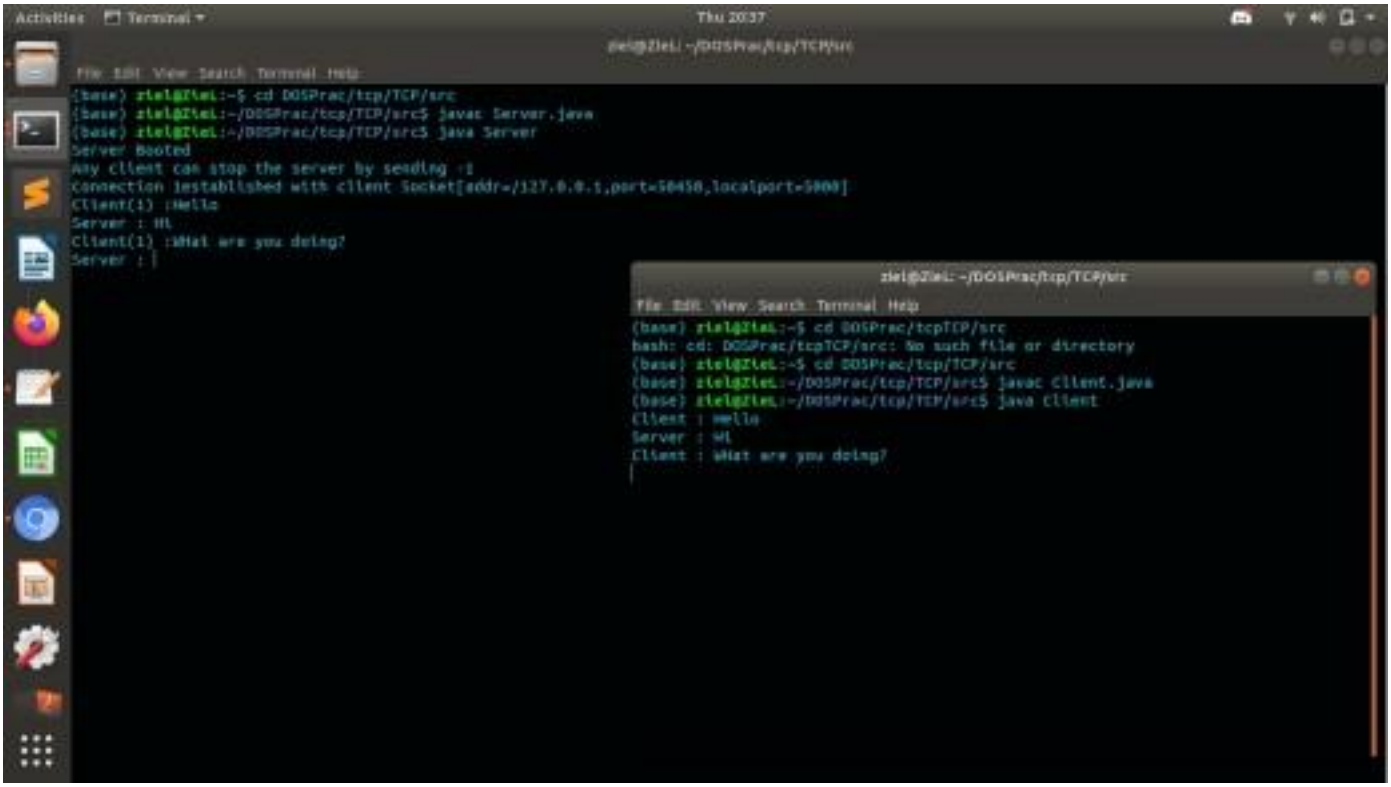
```
 *
 * @author Ayush
 */
public class Server {
 int port;
 ServerSocket server=null;
 Socket client=null;
 ExecutorService pool = null;
 int clientcount=0;

 public static void main(String[] args) throws IOException {  Server serverobj=new
Server(5000);
 serverobj.startServer();
 }

 Server(int port){
 this.port=port; pool =
 Executors.newFixedThreadP
 ool(5);

 }

 public void startServer() throws IOException {
 server=new ServerSocket(5000);
 System.out.println("Server Booted");
 System.out.println("Any client can stop the server by sending -1");
 while(true)
```

```
{

client=server.accept();
clientcount++;

ServerThread runnable= new ServerThread(client,clientcount,this);

pool.execute(runnable);

}


}


private static class ServerThread implements Runnable {

Server server=null;

Socket client=null;

BufferedReader cin;

PrintStream cout;

Scanner sc=new Scanner(System.in);

int id;

String s;

ServerThread(Socket client, int count ,Server server ) throws IOException {


this.client=client;

this.server=server;

this.id=count;

System.out.println("Connection "+id+"established with client "+client);

cin=new BufferedReader(new InputStreamReader(client.getInputStream()));

cout=new PrintStream(client.getOutputStream());

}
```

```
 @Override
 public void run()
 { int x=1; try{
 while(true){
 s=cin.readLine();
System. out.print("Client("+id+") :"+s+"\n");
System.out.print("Server : ");
//s=stdin.readLine();
 s=sc.nextLine(); if
 (s.equalsIgnoreCase("bye"))  {
 cout.println("BYE"); x=0;
 System.out.println("Connection ended by server");
 break;
 }
                        cout.println(s);
                }
 cin.close();
 client.close();
 cout.close();
 if(x==0) {
                    System.out.println( "Server cleaning up." );
                    System.exit(0);
 }
 }
 catch(IOException ex){
 System.out.println("Error : "+ex);
```

```
    }



    }

    }

    }
```

OUTPUT


Activities
Terminal
Thu 20:37
File Edit View Search Terminal Help
(base) ziel@ZieL:~$ cd DOSPrac/tcp/TCP/src
(base) ziel@ZieL:~/DOSPrac/tcp/TCP/src$ javac Server.java
(base) ziel@ZieL:~/DOSPrac/tcp/TCP/src$ java Server
Server Booted
Any client can stop the server by sending :I
Connection iestablished with client Socket[addr=/127.0.0.1,port=60450,localport=5000]
Client(1) :Hello
Server : Hi
Client(1) :What are you doing?
Server :
ziel@ZieL: ~/DOSPrac/tcp/TCP/src
File Edit View Search Terminal Help
(base) ziel@ZieL:~$ cd DOSPrac/tcpTCP/src
bash: cd: DOSPrac/tcpTCP/src: No such file or directory
(base) ziel@ZieL:~$ cd DOSPrac/tcp/TCP/src
(base) ziel@ZieL:~/DOSPrac/tcp/TCP/src$ javac Client.java
(base) ziel@ZieL:~/DOSPrac/tcp/TCP/src$ java Client
Client : Hello
Server : Hi
Client : What are you doing?

GOVERNMENT COLLEGE OF ENGINEERING JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Krushna Shivaji Sase PRN No: 1941047

Class: T. Y. B.Tech. Computer Batch: T3

Date of Performance: 13/10/2021 Date of Completion: 13/10/2021

Subject:DSL Sign. of Teacher with Date: _____

Experiment No.: 08

Aim: RPC mechanism for a file transfer across a network.

Title: Implement RPC mechanism for a file transfer across a network.

Software Requirements: Ubuntu OS, Eclipse IDE 4.11 **Theory:**

Distributed File Systems

A file system defines the naming structure, characteristics of the files and the set of operations associated with them.

The classification of computer systems and the corresponding file system requirements are given below. Each level subsumes the functionality of the layers below in addition to the new functionality required by that layer, shown in figure 1.

Distributed File System Issues 1. **Naming**

How are files named? Access independent? Is the name location independent?

- FTP. location and access dependent.
- NFS. location dependent through client mount points. Largely transparent for ordinary users, but the same remote file system could be mounted differently on different machines. Access independent. See

Fig 9-3. Has automount feature for file systems to be mounted on demand. All clients could be configured to have same naming structure.

- AFS. location independent. Each client has the same look within a *cell*. Have a cell at each site. See Fig 13-15.

2. Migration

Can files be migrated between file server machines? What must clients be aware of?

- FTP. Sure, but end-user must be aware.
- NFS. Must change mount points on the client machines.
- AFS. On a per-volume (collection of files managed as a single unit) basis.

3. Directories

Are directories and files handled with the same or a different mechanism?

- FTP. Directory listing handled as remote command.
- NFS. Unix-like. AFS. Unix-like.

Amoeba has separate mechanism for directories and files.

4. Sharing Semantics

What type of file sharing semantics are supported if two processes accessing the same

file? Possibilities:

- Unix semantics - every operation on a file is instantly visible to all processes.
- session semantics - no changes are visible to other processes until the file is closed.

immutable files - files cannot be changed (new versions must be created)

- FTP. User-level copies. No support.
- NFS. Mostly Unix semantics. AFS. Session semantics.

Immutable files in Amoeba.

5. Caching

What, if any, file caching is supported?

Possibilities:

- write-through - all changes made on client are immediately written through to server
- write-back - changes made on client are cached for some amount of time before being written back to server. write-on-close - one type of write-back where changes are written on close (matches session semantics).

- FTP. None. User maintains own copy (whole file)
- NFS. File attributes (inodes) and file data blocks are cached separately. Cached attributes are validated with the server on file open.

Version 3: Uses read-ahead and delayed writes from client cache. Time-based at block level. New/changed files may not visible for 30 seconds. Neither Unix nor session semantics. Non-deterministic semantics as multiple processes can have the same file open for writing.

Version 4: Client must flush modified file contents back to the server on close of file at client. Server can also *delegate* a file to a client so that the client can handle all requests for the file without checking with the

server. However, server must now maintain state about open delegations and recall (with a callback) a delegation if the file is needed on another machine.

- AFS. File-level caching with callbacks (explain). Session semantics. Concurrent sharing is not possible.

6. Locking

Does the system support locking of files?

- FTP. N/A.
- NFS. Has mechanism, but external to NFS in v3. Internal to file system in version 4. AFS. Does support.

7. Replication/Reliability

Is file replication/reliability supported and how?

- FTP. No.
- NFS. minimal support in version 4.
- AFS. For read-only volumes within a cell. For example binaries and system libraries.

8. Scalability

Is the system scalable?

- FTP. Yes. Millions of users.
- NFS. Not so much. 10-100s
- AFS. Better than NFS, keep traffic away from file servers. 1000s.

9. Homogeneity

Is hardware/software homogeneity required?

- FTP. No.
- NFS. No. AFS. No.

10. File System Interface

Is the application interface compatible to Unix or is another interface used?

- FTP. Separate.
- NFS. The same. AFS. The same.

11. Security

What security and protection features are available to control access?

- FTP. Account/password authorization.
- NFS. RPC Unix authentication. Version 4 uses RPCSEC_GSS, a general security framework that can use proven security mechanisms such as Kerberos.
- AFS. Unix permissions for files, access control lists for directories. CODA has secure RPC implementation.

12. State/Stateless

Do file system servers maintain state about clients?

- FTP. No.
- NFS. No. In Version 4 servers maintains state about delegations and file locking.
- AFS. Yes.

File Models

1. Unstructured and Structured files

- In the unstructured model, a file is an unstructured sequence of bytes. The interpretation of the meaning and structure of the data stored in the files is up to the application (e.g. UNIX and MS-DOS). Most modern operating systems use the unstructured file model.
- In structured files (rarely used now) a file appears to the file server as an ordered sequence of records. Records of different files of the same file system can be of different sizes.

2. Mutable and immutable files

- Based on the modifiability criteria, files are of two types, mutable and immutable. Most existing operating systems use the mutable file model. An update performed on a file overwrites its old contents to produce the new contents

Mechanisms for Building Distributed File Systems

1. Mounting

Mount mechanisms allow the binding together of different file namespaces to form a single hierarchical namespace. The Unix operating system uses this mechanism. A special entry, known as a mount point, is created at some position in the local file namespace that is bound to the root of another file name space. From the users point of view a mount point is indistinguishable from a local directory entry and may be traversed using standard path names once mounted. File access is therefore location transparent for the user but not for the system administrator. The kernel maintains a structure called a mount table which maps mount points to appropriate file systems. Whenever a file access path crosses a mount point, this is intercepted by the kernel, which then obtains the required service from the remote server.

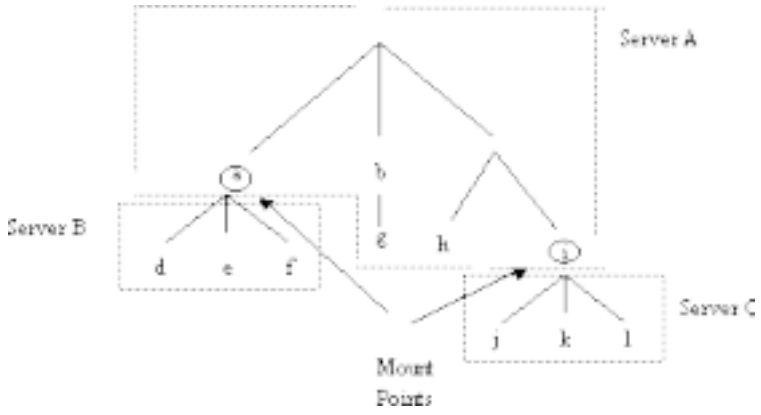


Figure 2- Mounting in file systems

Client machines can maintain mount information individually as is done in Sun's Network File System. Every client individually mounts all the required file systems at specific points in its local file space. Note that each client need not necessarily mount the file systems at the same points. This makes it difficult to write portable code which can locate files in the distributed file system irrespective of where it is executed. Movement of files between servers requires each client to unmount and remount the affected subtree. Note that to simplify the implementation, Unix does not allow hard links to be placed across mount points, i.e. between two separate file systems. Also, servers are unaware of where the subtrees exported by them have been mounted.

Mount information can be maintained at servers in which case it is possible that every client sees an identical namespace. If files are moved to a different server then mount information need only be updated at the servers. Servers each manage domains or volumes which are subtrees of the overall name space. Clients may initially direct file requests to any server which will direct the client to the server which manages the domain containing the required file. This information may be used to guide the client more efficiently for future accesses.

2. Client Caching

Caching is the architectural feature which contributes the most to performance in a distributed file system. Caching exploits temporal locality of reference. A copy of data stored at a remote server is brought to the client. Other metadata such as directories, protection and file status or location information also exhibit locality of reference and are good candidates for caching.

Data can be cached in main memory or on the local disk. A key issue is the size of cached units, whether entire files or individual file blocks are cached. Caching entire files is simpler and most files are in fact read sequentially in their entirety, but files which are larger than the client cache cannot be fetched. Cache validation can be done in two ways. The client can contact the server before accessing the cache or the server can notify clients when data is rendered stale. This can reduce client-server traffic.

3. Bulk Data Transfer

All data transfer in a network requires the execution of various layers of communication protocols. Data is assembled and disassembled into packets, it is copied between the buffers of various layers in the communication protocols and transmitted in individually acknowledged packets over the network. For small amounts of data, the transit time across the network is low, but there are

relatively high latency costs involved with the communication protocols establishing peer connections, packaging the small data packets for individual transmission and acknowledging receipt of each packet at each layer.

Transferring data in bulk reduces the relative cost of this overhead at the source and destination. With this scheme, multiple consecutive data packets are transferred from servers to clients (or vice versa) in a burst. At the source, multiple packets are formatted and transmitted with one context switch from client to kernel. At the destination, a single acknowledgement is used for the entire sequence of packets received.

Caching amortizes the cost of accessing remote servers over several local references to the same information, *bulk transfer* amortizes the cost of the fixed communication protocol overheads and possibly disk seek time over many consecutive blocks of a file. Bulk transfer protocols depend on the spatial locality of reference within files for effectiveness. Remember there is substantial empirical evidence that files are read in their entirety. File server performance may be enhanced by transmitting a number of consecutive file blocks in response to a client block request. ***4. Encryption***

Encryption is used for enforcing security in distributed systems. A number of possible threats exist such as unauthorized release of information, unauthorized modification of information, or unauthorized denial of resources. Encryption is primarily of value in preventing unauthorized release and modification of information.

ALGORITHM:

Steps:

1. Defining a remote interface
2. Implementing the remote interface
3. Creating Stub and Skeleton objects from the implementation class using rmic (rmicomplier) 4. Start the rmiregistry
5. Create and execute the server application program
6. Create the client application program.
7. Read over the file content until either EOF is reached or maximum characters is read and stored incharacter array.
8. Execute the client application program.

Conclusion:

In this Practical, we learnt about the implementation of RPC mechanism for a file transfer across

the network. **DOC: 27/04/2021**

……

Mrs. Madhavi Kale

Name & Sign of Course Teacher

---

tranfer_client.c

```
#include "transfer.h"
#include <time.h>

void transfer_1(char *host, char *filetotransf)
{
    CLIENT *clnt; int
    *result_1; file
    transf_1_arg; FILE
    *ofile; long long int
    total = 0;

                clnt = clnt_create (host, TRANSFER, TRANSFER_1, "tcp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }

    ofile = fopen(filetotransf, "rb");

    if(ofile == NULL) {
        printf("File not found.\n");
        exit(1);
```

```
}

printf("Sending file %s.\n", filetotransf);

strcpy(transf_1_arg.name, filetotransf);

clock_t begin = clock();

while(1) { transf_1_arg.nbytes = fread(transf_1_arg.data, 1, MAXLEN,
        ofile); total += transf_1_arg.nbytes;

        //printf("\r%lld bytes of %s sent to server.", total, transf_1_arg.name);

        result_1 = transf_1(&transf_1_arg, clnt);

        if (result_1 == (int *) NULL) {
                                clnt_perror (clnt, "call failed");
        }

        if(transf_1_arg.nbytes < MAXLEN) {
                printf("\nUpload finished.\n");
                break;

        }
}
```

```
    clock_t end = clock(); double upload_time = (double)(end - begin) /
    CLOCKS_PER_SEC; printf("Upload

    time: %lf\n", upload_time);

    clnt_destroy (clnt);
    fclose(ofile);

} int main (int argc, char

*argv[])

{
    char *host; char
    *filetotransf;

    if (argc < 3) {
                                printf ("usage: %s <server_host> <file>\n", argv[0]);
    exit (1);
    }
    host = argv[1];
    filetotransf = argv[2];

    transfer_1 (host, filetotransf);
exit (0);
}
```

transfer_clnt.c

```
#include <memory.h> /* for memset */
#include "transfer.h"

/* Default timeout can be changed using clnt_control() */ static struct timeval
TIMEOUT = { 25, 0 };

int * transf_1(file *argp, CLIENT
*clnt)
{ static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, TRANSF,

            (xdrproc_t) xdr_file, (caddr_t) argp,
            (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
            TIMEOUT) != RPC_SUCCESS) {
            return (NULL);
    } return
    (&clnt_res);
}
```

transfer_server.c

```
#include "transfer.h"

char opened_file[MAXLEN];
```

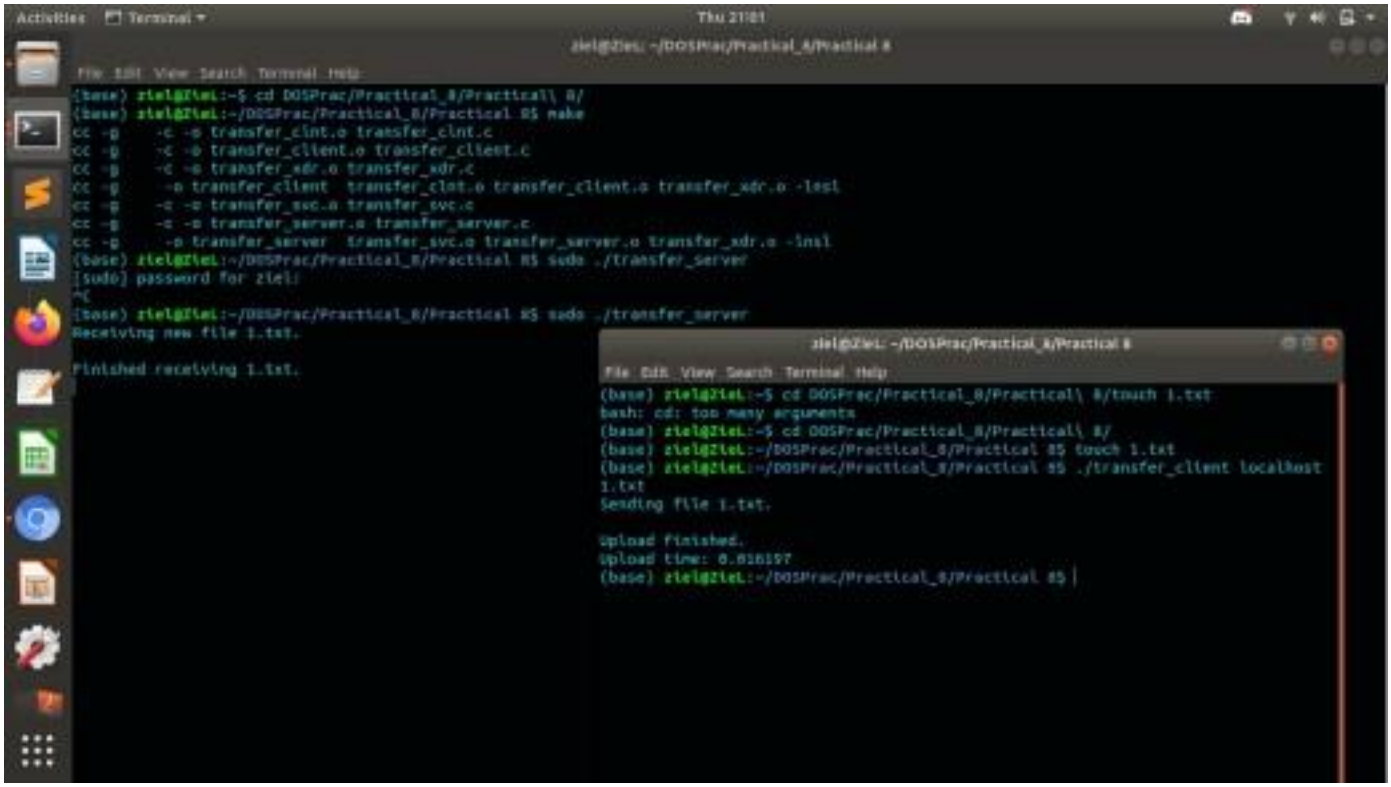
```
FILE *ofile; long long
int total = 0;

int * transf_1_svc(file *argp, struct svc_req
*rqstp)
{ static int result;
      static char tempName[MAXLEN];

      /*
       * insert server code here
       */ /*
      strcpy(tempName, "uploaded_");
      strcat(tempName, argp->name); strcpy(argp-
      >name, tempName);
      */
      total += argp->nbytes;

      if (strcmp(opened_file, "") == 0 && ofile == NULL) {
                                    printf("Receiving new file %s.\n", argp->name);

            strcpy(opened_file, argp->name);
            ofile = fopen(argp->name, "ab+");
      }
      if (strcmp(opened_file, argp->name) == 0) {
            //printf("\r%lld bytes of file %s were received.", total, argp->name);
            fflush(stdout);
```

```
fwrite(argp->data, 1, argp->nbytes, ofile);

        if (argp->nbytes < MAXLEN) {

                printf("\nFinished receiving %s.\n", argp->name);

            total = 0;

            fclose(ofile);
```



```
ofile = NULL; strcpy(opened_file,
"");

}

} return

&result; }
```

OUTPUT



GOVERNMENT COLLEGE OF ENGINEERING,JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Krushna Shivaji Sase PRN No: 1941047

Class: T. Y. B.Tech. Computer Batch: T3

Date of Performance: 20/11/2021 Date of Completion: 20/11/2021

Subject:DSL Sign. of Teacher with Date: _____

Experiment No.: 09

Title: To study Kerberos

Theory:

Kerberos

System security and integrity within a network can be unwieldy. It can occupy the time of several administrators just to keep track of what services are being run on a network and the manner in which these services are used.

Further, authenticating users to network services can prove dangerous when the method used by the protocol is inherently insecure, as evidenced by the transfer of unencrypted passwords over a network using the traditional FTP and Telnet protocols.

Kerberos is a way to eliminate the need for protocols that allow unsafe methods of authentication, thereby enhancing overall network security.

What is Kerberos?

Kerberos is a network authentication protocol created by MIT, and uses symmetric-key cryptography[18] to authenticate users to network services, which means passwords are never actually sent over the network.

Consequently, when users authenticate to network services using Kerberos, unauthorized users attempting to gather passwords by monitoring network traffic are effectively thwarted.

Advantages of Kerberos

Most conventional network services use password-based authentication schemes. Such schemes require a user to authenticate to a given network server by supplying their username and password. Unfortunately, the transmission of authentication information for many services is unencrypted. For such a scheme to be secure, the network has to be inaccessible to outsiders, and all computers and users on the network must be trusted and trustworthy.

Even if this is the case, a network that is connected to the Internet can no longer be assumed to be secure. Any attacker who gains access to the network can use a simple packet analyzer, also known as a packet sniffer, to intercept usernames and passwords, compromising user accounts and the integrity of the entire security infrastructure.

The primary design goal of Kerberos is to eliminate the transmission of unencrypted passwords across the network. If used properly, Kerberos effectively eliminates the threat that packet sniffers would otherwise pose on a network.

Disadvantages of Kerberos

Although Kerberos removes a common and severe security threat, it may be difficult to implement for a variety of reasons:

- Migrating user passwords from a standard UNIX password database,such as **/etc/passwd** or **/etc/shadow**, to a Kerberos password database can be tedious, as there is no automated mechanism to perform this task. Refer to Question 2.23 in the online Kerberos FAQ: http://www.nrl.navy.mil/CCS/people/kenh/kerberos-faq.html

- Kerberos has only partial compatibility with the Pluggable AuthenticationModules (PAM) system used by most Red Hat Enterprise Linux servers.Refer to Section 42.6.4, “Kerberos and PAM”for more information about this issue.

- Kerberos assumes that each user is trusted but is using an untrusted host on an untrusted network. Its primary goal is to prevent unencrypted passwords from being transmitted across that network. However, if anyone other than the proper user has access to the one host that issues tickets used for authentication — called the *key distribution center* (*KDC*) — the entire Kerberos authentication system is atrisk. o For an application to use Kerberos, its source must be modified to make the appropriate calls into the Kerberos libraries. Applications modified in this way are considered to be *Kerberos-aware*, or *kerberized*. For some applications, this can be quite problematic due to the size of the application or its design. For other incompatible applications, changes must be made to the way in which the server and client communicate. Again, this may require extensive programming. Closed-source applications that do not have Kerberos support by default are often the most problematic.

- Kerberos is an all-or-nothing solution. If Kerberos is used on the network,any unencrypted passwords transferred to a non-Kerberos aware service is at risk. Thus, the network gains no benefit from the use of Kerberos. To secure a network with Kerberos, one must either use Kerberos-aware versions of *all*

client/server applications that transmit passwords unencrypted, or not use *any* such client/server applications atall.

Kerberos Terminology

Kerberos has its own terminology to define various aspects of the service. Before learning how Kerberos works, it is important to learn the following terms.

authentication server (AS)

A server that issues tickets for a desired service which are in turn given to users for access to the service. The AS responds to requests from clients who do not have or do not send credentials with a request. It is usually used to gain access to the ticket-granting server (TGS) service by issuing a ticket-granting ticket (TGT). The AS usually runs on the same host as the key distribution center (KDC).

ciphertext

Encrypted data.

client

An entity on the network (a user, a host, or an application) that can receive a ticket from Kerberos.

credentials

A temporary set of electronic credentials that verify the identity of a client for a particular service. Also called a ticket.

credential cache or ticket file

A file which contains the keys for encrypting communications between a user and various network services. Kerberos 5 supports a framework for using other cache types, such as shared memory, but files are more thoroughly supported.

crypt hash

A one-way hash used to authenticate users. These are more secure than using unencrypted data, but they are still relatively easy to decrypt for an experienced cracker.

GSS-API

The Generic Security Service Application Program Interface (defined in RFC 2743 published by The Internet Engineering Task Force) is a set of functions which provide security services. This API is used by clients and services to authenticate to each other without either program having specific knowledge of the underlying mechanism. If a

network service (such as cyrus-IMAP) uses GSS-API, it can authenticate using Kerberos. **hash**

Also known as a hash value. A value generated by passing a string through a hash function. These values are typically used to ensure that transmitted data has not been tampered with.

hash function

A way of generating a digital "fingerprint" from input data. These functions rearrange, transpose or otherwise alter data to produce a hash value.

key

Data used when encrypting or decrypting other data. Encrypted data cannot be decrypted without the proper key or extremely good fortune on the part of the cracker.

key distribution center (KDC)

A service that issues Kerberos tickets, and which usually run on the same host as the ticket-granting server (TGS).

keytab (or key table)

A file that includes an unencrypted list of principals and their keys. Servers retrieve the keys they need from keytab files instead of using kinit. The default keytab file is /etc/krb5.keytab. The KDC administration server, /usr/kerberos/sbin/kadmind, is the only service that uses any other file (it uses /var/kerberos/krb5kdc/kadm5.keytab).

kinit

The kinit command allows a principal who has already logged in to obtain and cache the initial ticket-granting ticket (TGT). Refer to the kinit man page for more information.

principal (or principal name)

The principal is the unique name of a user or service allowed to authenticate using Kerberos. A principal follows the form root[/instance]@REALM. For a typical user, the root is the same as their login ID. The instance is optional. If the principal has an instance, it is separated from the root with a forward slash ("/"). An empty string ("") is considered a valid instance (which differs from the default NULL instance), but using it can be confusing. All principals in a realm have their own key, which for users is derived from a password or is randomly set for services. **realm**

A network that uses Kerberos, composed of one or more servers called KDCs and a potentially large number of clients.

service

A program accessed over the network.

ticket

A temporary set of electronic credentials that verify the identity of a client for a particular service. Also called credentials.

ticket-granting server (TGS)

A server that issues tickets for a desired service which are in turn given to users for access to the service. The TGS usually runs on the same host as the KDC.

ticket-granting ticket (TGT)

A special ticket that allows the client to obtain additional tickets without applying for them from the KDC.

unencrypted password

A plain text, human-readable password.

How Kerberos Works

Kerberos differs from username/password authentication methods. Instead of authenticating each user to each network service, Kerberos uses symmetric encryption and a trusted third party (a KDC), to authenticate users to a suite of network services. When a user authenticates to the KDC, the KDC sends a ticket specific to that session back to the user's machine, and any Kerberos aware services look for the ticket on the user's machine rather than requiring the user to authenticate using apassword.

When a user on a Kerberos-aware network logs in to their workstation, their principal is sent to the KDC as part of a request for a TGT from the Authentication Server. This request can be sent by the log-in program so that it is transparent to the user, or can be sent by the kinit program after the user logs in.
The KDC then checks for the principal in its database. If the principal is found, the KDC creates a TGT, which is encrypted using the user's key and returnedto thatuser.

The login or kinit program on the client then decrypts the TGT using the user's key, which it computes from the user's password. The user's

key is used only on the client machine and is not transmitted over the network.

The TGT is set to expire after a certain period of time (usually ten to twenty four hours) and is stored in the client machine's credentials cache. An expiration time is set so that a compromised TGT is of use to an attacker for only a short period of time. After the TGT has been issued, the user does not have to re-enter their password until the TGT expires or until they log out and log in again.

Whenever the user needs access to a network service, the client software uses the TGT to request a new ticket for that specific service from the TGS. The service ticket is then used to authenticate the user to that service transparently.

Conclusion :-

In this Practical, we studied about Kerberos

DOC: 04/05/2021

Mrs. Madhavi Kale

Name & Sign of Course Teacher

GOVERNMENT COLLEGE OF ENGINEERING,JALGAON
(An Autonomous Institute of Government of Maharashtra)
Department of Computer Engineering

Name: Krushna Shivaji Sase PRN No: 1941047

Class: T. Y. B.Tech. Computer Batch: T3

Date of Performance: 27/11/2021 Date of Completion: 27/11/2021

Subject:DSL Sign. of Teacher with Date: _____

Experiment No.: 10

Title: To Study BitTorrent and End System Multicast.

Theory:

INTRODUCTIO N T O BIT T ORRENT

BitTorrent is a technology/protocol which makes the distribution of files, especially large files, easier and less bandwidth consuming for the publisher. This is accomplished by utilizing the upload capacity of the peers that are downloading a file. A considerably increase in downloaders will only result in a modest increase in the load on the publisher hosting the file.

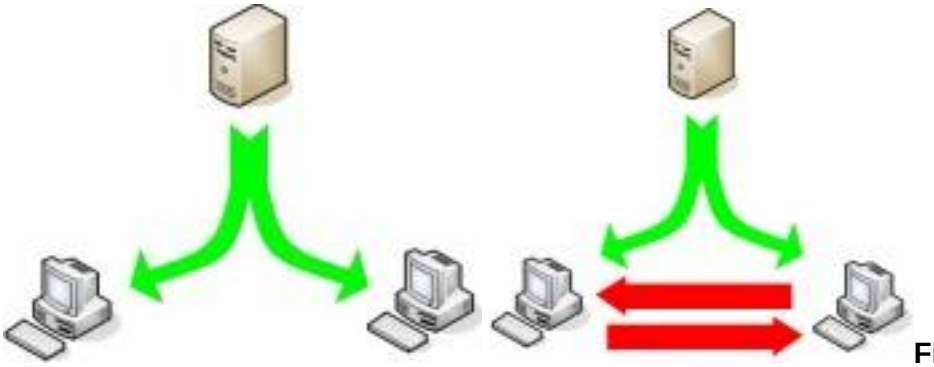


Figure 1 – The basic flow of the BitTorrent protocol.

The illustration in 1 shows the basic flow of BitTorrent. The figure on the left shows a client-server approach to download. The peers download from the server simultaneous. If we assume the upload capacity of the server is the same as the download capacity of a peer, the time for the download to finish will be two times the time if only one peer where downloading from the server. The figure on the right shows an approach similar to BitTorrent. By splitting the file and send one part to each peer, and let the peers download the part they are missing from each other, both download time and load on the server is reduced. Of course, the BitTorrent protocol is much more sophisticated than this simple example, but this shows the basic idea.

THE HISTORY OF BITTORRENT

BitTorrent is by far the most popular peer-to-peer programs ever. Analysis shows that it accounts for about 35% of all Internet traffic. How did it become so popular, and what makes it so special? In the summer of 2001, Cohen released his first beta version of BitTorrent. In 2002 he presented it at a conference. His goal with this software was to give people a quick and simple way of distributing and swapping Linux software online. But, as we all know, the movie-geeks soon saw the potential in the BitTorrent technology. In 2004, pirate copies of movies and TV-shows began dominating the BitTorrent traffic, and after that the growth has been explosive. It is projected that by the end of this year about 40 million

AREAS OF USAGE

Piracy and illegal distribution is the first thing brought to mind when you hear about peer-to peer file sharing. Not unreasonable, since the majority of traffic is illegal exchange of copyrighted material. But the BitTorrent protocol has features which makes it usable for totally legitimate purposes.

large software and patches can be done at higher speeds using BitTorrent .How often haven't you waited too long for the newest security update from Windows to be finished downloading? Using the BitTorrent protocol the spreading of this kind of software can be much quicker and more satisfactory for the end users. Within an organization one can also use the protocol to distribute applications and updates more rapidly. Improvements in the protocol facilitate this usage

THE BITTORRENT ARCHITECTURE:

The BitTorrent architecture normally consists of the following entities: - a static metainfo file (a "torrent file") - a 'tracker' - an original downloader ("seed") - the end user downloader ("leecher") It's all about the torrent file, the centralized tracker and the associated swarm of peers. The centralized tracker provides the different entities with an address list over the available peers. These peers will then contact each other to download pieces of the file from each other. The first step in publishing a file using BitTorrent is to create a metainfo file from the file that you want to publish. The metainfo file is called a "torrent". The torrent file contains the filename, size, hashing information and the URL of the "tracker". The "torrent" is needed by anyone who wants to download the file the torrent is created from. The torrent file can be distributed by e-mail, IRC, http etc. The torrent is created by using a free program. This functionality is also commonly included in the BitTorrent clients. To download or "seed" a file, you need a BitTorrent client. The BitTorrent client is a free application that administrates the download procedure. There are several different BitTorrent clients available. They all support the standard BitTorrent protocol, but may differ and be incompatible with each other regarding certain features. A BitTorrent download is started by opening the torrent file in the BitTorrent client. The tracker keeps a log of peers that are currently downloading a file, and helps them find each other. The tracker is not directly involved in the transfer of data and CO455 Distributed Operating System Lab does not have a copy of the file. The tracker and the downloading users exchange information using a simple protocol on top of HTTP. First, the user gives information to the tracker about which file it's downloading, ports it's listening on etc. The response from the tracker is a list of other users which are downloading the same file and information on how to contact them. This group of peers that all share the same torrent represents a 'swarm'. When creating the torrent file from the original file, the original file is cut into smaller pieces, usually 512 KB or 256Kb in size. The SHA-1 hash codes of the pieces are included in the torrent file. The downloaded data are verified by computing the SHA-1 hash code and comparing it to the SHA-1 code of the corresponding piece in the torrent file. This way the data is checked for errors and it guarantees to the users that they are downloading the real thing. Whenever a piece is downloaded and verified, the downloading peer reports to the other peers in the swarm about its new piece. This piece is now available for other peers. We will now go deeper into piece selection and other details of the protocol.

DECENTRALIZED TRACKER

With one centralized tracker the BitTorrent network is not very fault tolerant. If the tracker goes down the file will no longer be available, since there are no way the peers could know about each other. One centralized tracker also makes the network vulnerable to denial of service attacks. In May 2005 a version 4.1 was released by Cohen and the big newsflash was support for a decentralized tracker. In this new version (a beta release), all you have to do is to publish your .torrent-file on a webpage, blog etc., and no tracker specification is necessary. Use of a dedicated tracker is still supported, and the publisher can choose the preferred mode. The tracker is distributed in the sense that every client or node in the network now acts a lightweight tracker. The solution is based on distributed hash tables (DHTs). This makes it possible to share files with minimal resources, but no guarantees can be made as respect to reliability. In addition to making it easier for a more novice user to share his files through a simple website or a blog, one of the other driving forces for the trackerless mode was economy. A website owner might have to pay (to the web server owner) according to the traffic on his website. The more people who want to download your file, the larger the traffic the website generates the lager the bill will be for the owner. Cohen thinks this is unfair as compared to broadcast by a traditional TV-station: whether

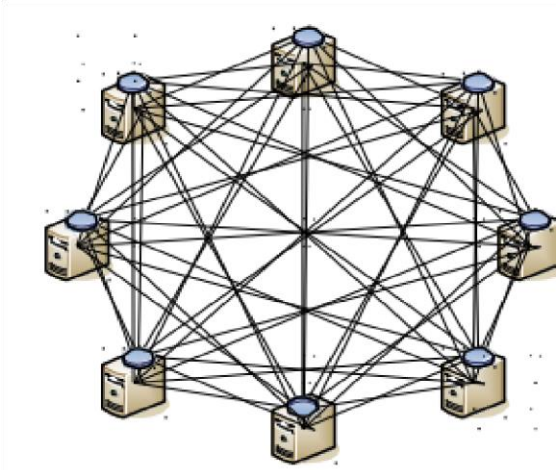


Figure 2.1 – BitTorrent with a Decentralized tracker structure

five people

DISTRIBUTED HASH TABLES (DHTS)

A DHT is a decentralized distributed system. The system consists of a set of participating nodes and a set of keys. The DHT performs the function of a hash table. Key and value pair can be stored and a value can be looked up if the correct key is provided. What separates a DHT from an ordinary hash table is that the storage and lookups are distributed among the nodes (machines) in the network. All nodes are peers that can join and leave the network freely. DHTs makes provable guarantees about performance despite the seemingly chaos created by random joining and leaving peers. Any DHT protocol emphasizes the following characteristics;

- **Decentralization**: the system is collectively created and maintained by the nodes without any central coordination.
- **Scalability**: the system performs efficiently even with thousands or millions of nodes. **Fault tolerance**: the system should be reliable even with nodes continuously joining, leaving or failing.

HOW TO USE BITTORRENT

BitTorrent is a file-sharing protocol that lets you download content directly from other groups of people. Unlike HTTP, the download speeds vary, because you aren't receiving data from a dedicated server, instead you're downloading from other users' computers.

Here’s a simple 3-step guide:

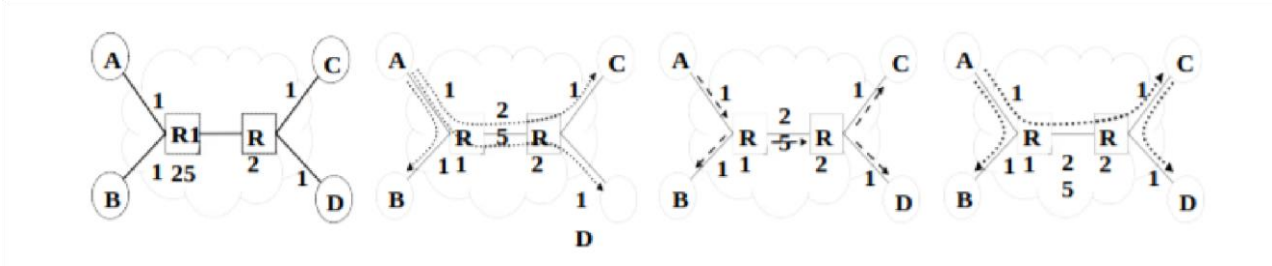
1. **Get a BitTorrent client**: First you need a BitTorrent client, a program that will enable you to connect to other users (or peers) and thereby download the files you want to. Our favorites are: Windows – uTorrent, Mac – Azureus, Linux – BitTorrent. Click through one of the links and download and install the client.
Tip: Want to download Torrents anonymously? Try BTguard, a great way to download torrents securely.

2. **Find a torrent**: Now that you have a BitTorrent client installed, you'll want to find torrents you like and download them. For that you have to visit a BitTorrent directory or use a BitTorrent search engine like Torrentz.
For example, open www.torrentz.com in your browser and search for anything (Linux for example). If you see something you like, click on one of the 'Download Locations' and download the .torrent file from the site.

3. **Start the download:** Once downloaded, double-click the file and it will open and start downloading in your BitTorrent client. Some clients like Azureus will ask you to specify a location to save the file.

END SYSTEM MULTICAST

Figure 3: Example to illustrate End System Multicast



ESM used a peer-to-peer network to distribute video data across all viewers of a video stream. It constructs an overlay tree to distribute data, and continuously optimizes this tree to minimize end-to-end latency. The root of the tree is the source of the broadcast. This is typically the machine that encodes the video data. This machine sends a stream of data packets to the nodes at the first level of the tree. Each of those nodes then forwards the data to the nodes connected to them, and so on, such that all nodes in the system receive the data stream. ESM allowed any user with a DSL or broadband connection or higher to broadcast good quality video to a large number of people. Since it is a peer to peer network, a broadcaster need only broadcast the video to one person for any number of people to view it. Due to the nature of peer to peer multimedia networks, skips in playback or buffering can occur.

To illustrate the di erences between IP Multicast, naive unicast and End System Multicast using Figure 3. Figure 3(a) depicts an example physical topology, where R1 and R2 are routers, while A, B, C and D are end systems. Link delays are as indicated. R1 − R2 represents a costly transcontinental link, while all other links are cheaper local links. Further, let us assume 'A' wishes to send data to all other nodes. Figure 3(b) depicts naive unicast transmission. Naive unicast results in significant redundancy on links near the source (for example, link A − R1 carries three copies of a transmission by A), and results in duplicate copies on costly links (for example, link R1 − R2 has two copies of a transmission by A). Figure 3(c) depicts the IP Multicast tree constructed by DVMRP. DVMRP is the classical IP Multicast protocol, where data is delivered from the source to recipients using an IP Multicast tree composed of the unicast paths from each recipient to the source. Redundant transmission is avoided, and exactly one copy of the packet traverses any given physical link. Each recipient receives data with the same delay as though A were sending to it directly by unicast Figure 3(d) depicts an "intelligent" overlay tree that may be constructed using the End System Multicast architecture. The number of redundant copies of data near the source is reduced compared to naive unicast, and just one copy of the packet goes

Given that End System Multicast tries to push functionality to the edges, there are two very different ways this can be achieved: peer-to-peer architectures and proxy-based architectures. In a peer to-peer architecture, functionality is pushed to the end hosts actually participating in the multicast group. Such architectures are thus completely distributed with each end host maintaining state only for those groups it

is actually participating in. In a proxy-based architecture on the other hand, an organization that provides value added services de- ploys proxies at strategic locations on the Internet. End hosts attach themselves to proxies near them, and receive data using plain unicast, or any available multicast media. While these architectures have important di erences, fundamental to both of them are concerns regarding the performance penalty involved in disseminating data using overlays as compared to solutions that have native multicast support. Thus, an end system in our paper refers to the entity that actually takes part in a self- organization protocol, and could be an end host or a proxy.

Our evaluation of End System Multicast targets a wide range of group communication applications such as audio and video conferencing, virtual classroom and multi-party network games. Such applications typically involve small (tens of members) and medium sized (hundreds of members) groups. While End Sys- tem Multicast may be relevant even for applications which involve much larger group sizes such as broadcasting and content distribution - particularly in the context of proxy-based architectures.

Conclusion:

In this Practical, we studied about the Bittorrent and End System Multicast

DOC: 04/05/2021

Mrs. Madhavi Kale Name & Sign of Course Teacher