

PRACTICAL – 01

AIM: Implement concurrent echo client server application using TCP socket.

Program:

TCPClient.java

```
import java.io.*;
import java.net.*;
class MyTCPClient
{
    public static void main(String argv[]) throws Exception
    {
        String Sentence;
        String GetSentence;

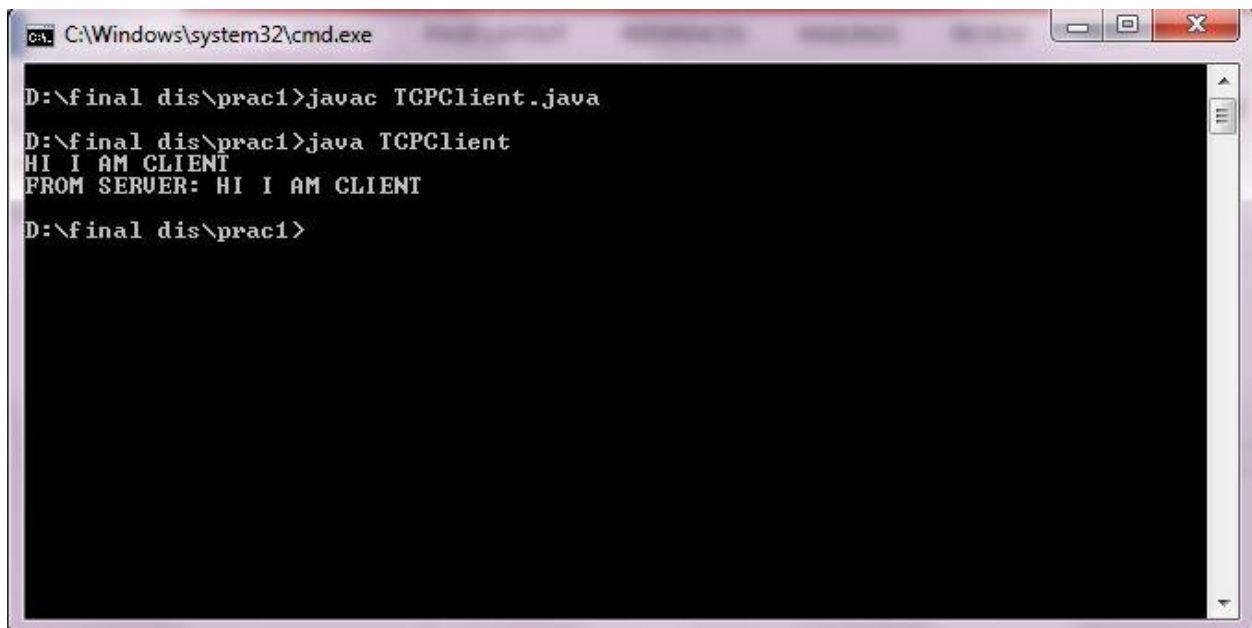
        BufferedReader FromUser = new BufferedReader( new InputStreamReader(System.in));
        Socket cSocket = new Socket("localhost", 6789);
        DataOutputStream ToServer = new DataOutputStream(clientSocket.getOutputStream());
        BufferedReader FromServer = new BufferedReader(new
        InputStreamReader(cSocket.getInputStream()));
        Sentence = getFromUser.readLine();
        ToServer.writeBytes(Sentence + '\n');
        GetSentence = FromServer.readLine();
        System.out.println("FROM SERVER: " + GetSentence);
        clientSocket.close();
    }
}
```

TCPServer.java

```
import java.io.*;
import java.net.*;
class TCPServer
{
    public static void main(String argv[]) throws Exception
    {
        String ClSentence;
        String CpsSentence;
        ServerSocket GetSocket = new ServerSocket(6789);

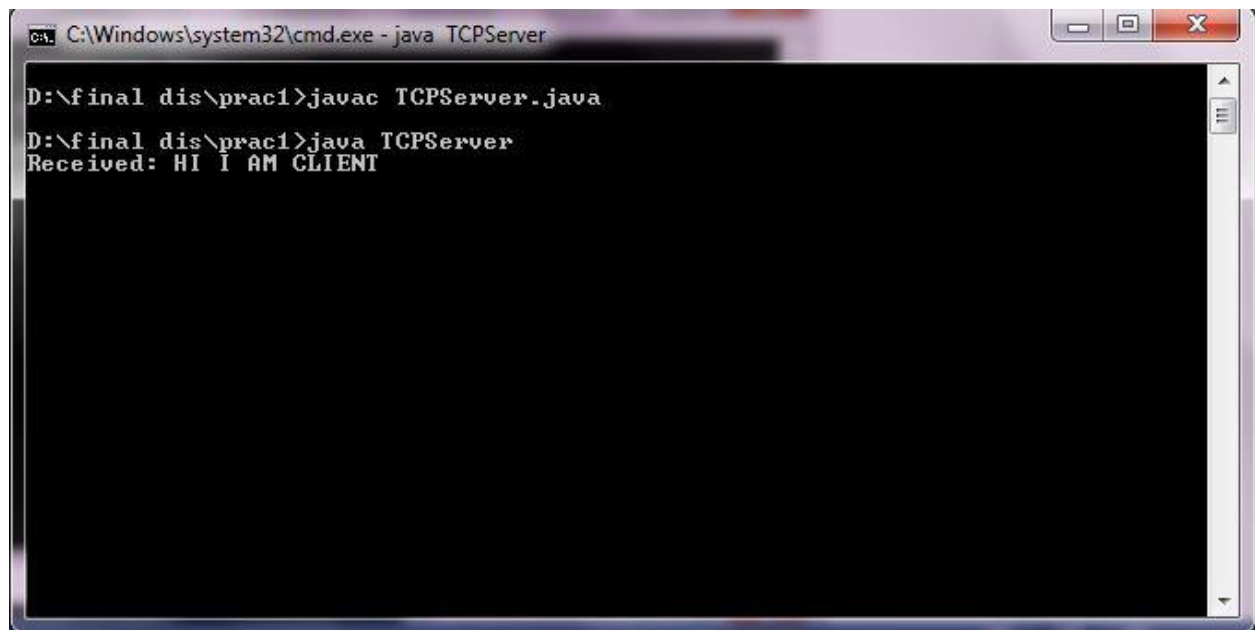
        while(true)
        {
            Socket ConSocket = GetSocket.accept();
```

```
BufferedReader FromClient = new BufferedReader(new  
InputStreamReader(ConSocket.getInputStream()));  
DataOutputStream ToClient=new DataOutputStream(ConSocket.getOutputStream());  
CISentence = FromClient.readLine();  
System.out.println("Received: " + CISentence);  
CpsSentence = CISentence.toUpperCase() + '\n';  
ToClient.writeBytes(CpsSentence);  
    }  
}  
}
```

Output:

The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The command prompt is at the directory "D:\final dis\prac1". The user has entered the command "javac TCPClient.java", which has been executed. Then, the user entered "java TCPClient", which has also been executed. The output of the program is displayed as two lines: "HI I AM CLIENT" and "FROM SERVER: HI I AM CLIENT". The command prompt is now waiting for the next input.

```
C:\Windows\system32\cmd.exe  
D:\final dis\prac1>javac TCPClient.java  
D:\final dis\prac1>java TCPClient  
HI I AM CLIENT  
FROM SERVER: HI I AM CLIENT  
D:\final dis\prac1>
```



A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Windows\system32\cmd.exe - java TCPServer". The command prompt shows the following text:

```
D:\final dis\prac1>javac TCPServer.java
D:\final dis\prac1>java TCPServer
Received: HI I AM CLIENT
```

The window has a standard Windows interface with minimize, maximize, and close buttons in the top right corner. The background is black, and the text is white.

PRACTICAL - 02

AIM: Implement concurrent day/time service using socket programming.

Program:

DateClient.java

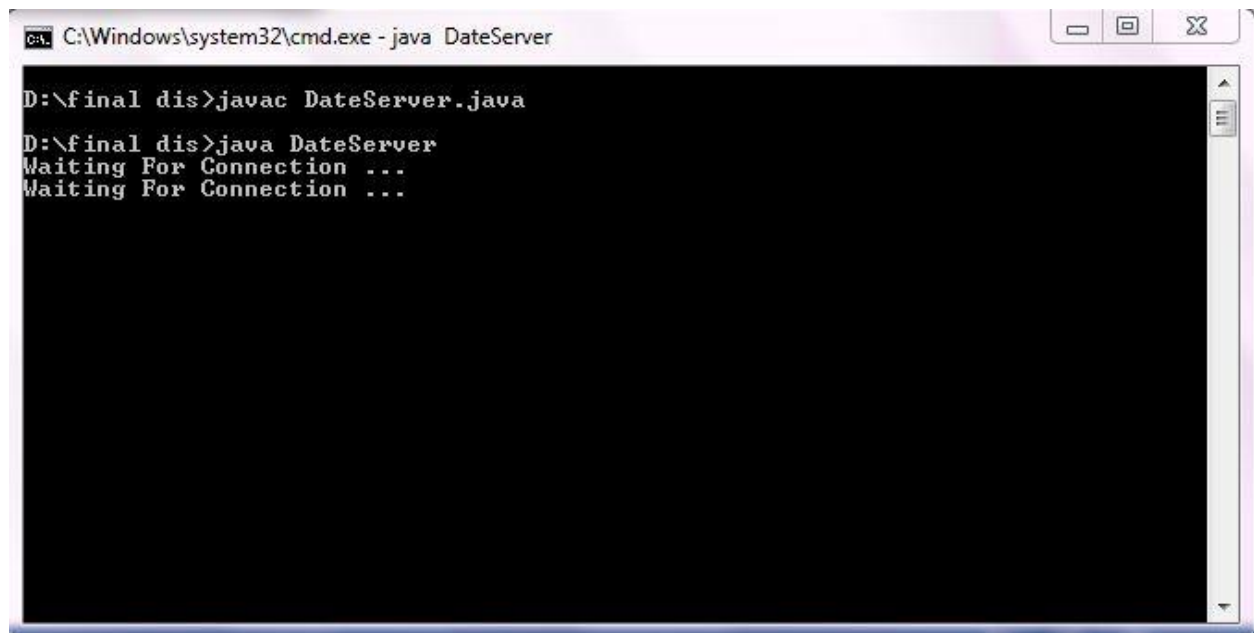
```
import java.io.*;
import java.net.*;
class DateClient
{
    public static void main(String args[]) throws Exception
    {
        Socket NewSocket=new Socket(InetAddress.getLocalHost(),5217);
        BufferedReader Input=new BufferedReader(new InputStreamReader(soc.getInputStream()));

        System.out.println(in.readLine());
    }
}
```

DateServer.java

```
import java.net.*;
import java.io.*;
import java.util.*;
class DateServer
{
    public static void main(String args[]) throws Exception
    {
        ServerSocket SS=new ServerSocket(5677);

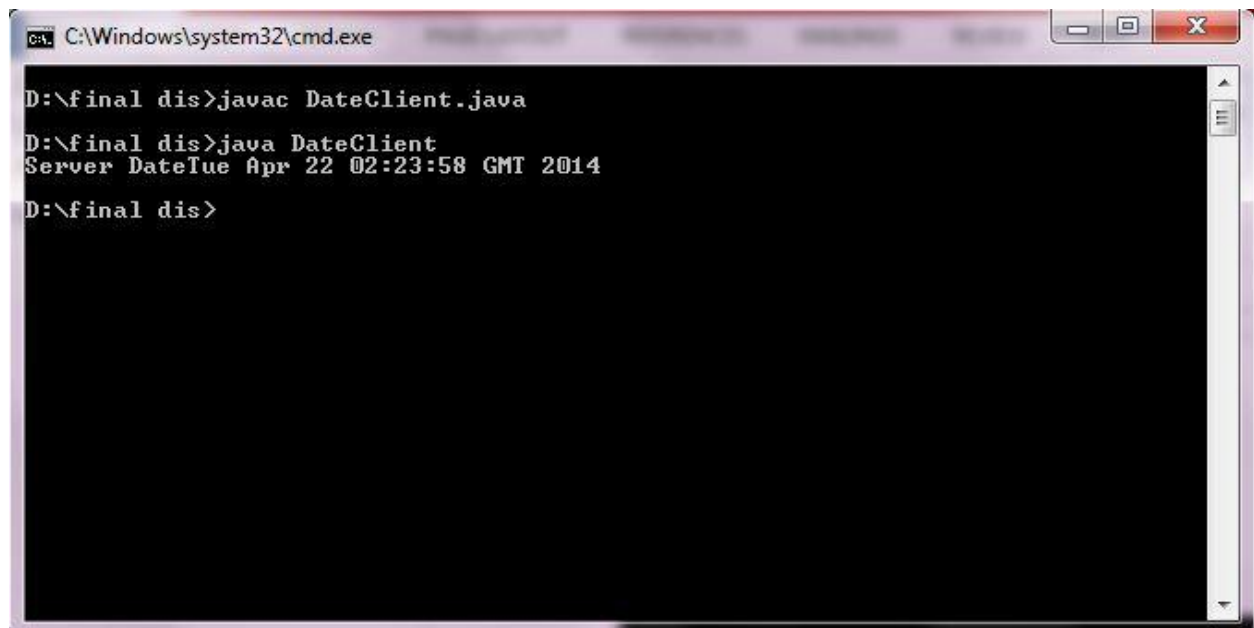
        while(true)
        {
            System.out.println("Waiting For Connection ...");
            Socket S=SS.accept();
            DataOutputStream out=new DataOutputStream(S.getOutputStream());
            out.writeBytes("Server Date" + (new Date()).toString() + "\n");
            out.close();
            S.close();
        }
    }
}
```

Output:

```
C:\Windows\system32\cmd.exe - java DateServer

D:\final dis>javac DateServer.java

D:\final dis>java DateServer
Waiting For Connection ...
Waiting For Connection ...
```



```
C:\Windows\system32\cmd.exe

D:\final dis>javac DateClient.java

D:\final dis>java DateClient
Server DateTue Apr 22 02:23:58 GMT 2014

D:\final dis>
```

PRACTICAL – 03

AIM: Study of Remote Procedure Call

Client Server Architecture

Client-server software architecture is versatile and flexible in today's fast-changing IT landscape. It is modular in structure and relies on messaging services for communication between components. They were designed to improve flexibility, usability, scalability, and interoperability. Software flexibility implies the ability for a program to change easily according to different users and different system requirements.

Usability refers to human-computer interaction and the ability of a software application to accomplish a user's goal. Some defining features are ease-of-use and a clear, logical process of evolution towards a goal. Scalability refers to a product's (be it hardware or software) ability to change in size or volume gracefully to meet user requests. Interoperability is the ability of software or hardware to function with other systems without requiring human intervention and manpower. Client-server software architecture aims to increase productivity through improvements in all of these categories.

Client-server architecture developed as a response to the limitations of file-sharing architectures, which require tons of bandwidth and can often stall or jam a network causing it to crash. They require low shared usage and low volume of data to be transferred. In client-server architecture, the database server replaced the file server. Relational database management systems (DBMSs) answered user queries directly. Since only specific queries were being answered, only that data was transferred instead of entire files that slow down networks. It also improved consistency in data between users, since all users had access to the same updated information.

The primary languages for structuring queries are SQL and RCP. SQL stand for 'standard query language'. SQL uses a GUI (graphic user interface) to make requests from databases. The newest ANSI (American National Standards Institute) standard is SQL. However, there are several recent versions of SQLs for sale from a variety of major vendors from Microsoft to Oracle. RPC is 'remote procedure call'.

RPC is protocol or set of rules structuring an intelligible request that is used by one program to request data or services from another program on another computer in another network. Full knowledge of network details is unnecessary. It allows an application to be distributed on and accessible from different platforms. Client and server stubs are created respectively so each party has the section it needs for the remote function it requests. Stubs are called to work when a remote function is required by the application and communication between client and server is synchronous. RPCs make it easier to design a client-server software architecture that employs multiple programs distributed over a network.

Two Tier Architecture

Two tier client-server software architectures improve usability and scalability. Usability is increased through user-friendly, form-based interfaces. Scalability is improved because two tiered systems can hold up to 100 users, whereas file server architectures can only accommodate 12. Two tiered architecture is best suited to homogeneous environments processing non-complex, non-time sensitive information.

Two tier architectures consist of three components: user system interfaces, processing management, and database management. User system interface (USI) is a component of an organization's decision support system, which includes human decision-makers. It provides a user friendly layer of communication to make requests of the server and offers multiple forms of input and output. USIs include features like display management services, sessions, text input, and dialog. Processing management includes process development, process implementation, process monitoring, and process resources services. Database management includes database and file services.

Two tier client-server design derives its name from how it distributes work between client and server. Clients access databases through the user system interface. Database management, on the server side, distributes processing between both client and server. Both tiers, the client and the server, are responsible for some of the processing management. Simply put, the client uses the user interface to make requests through database management on the server side.

Most of the application processing takes place on the client side, while the database management system (DBMS), on the server side, focuses on processing data through stored procedures. Connectivity between the tiers can be dynamically altered depending on users' requests and the services they are demanding. Two tier client server architectures work well for groups or businesses of up to 100 users on an LAN (Local Area Network), any more and service would deteriorate. Also, this software architecture offers limited flexibility by requiring the writing of manual code to move program functionality to a different server.

Three Tier Architecture

Three tier client server architecture is also known as multi-tier architecture and signals the introduction of a middle tier to mediate between clients and servers. The middle tier exists between the user interface on the client side and database management system (DBMS) on the server side. This third layer executes process management, which includes implementation of business logic and rules. The three tier models can accommodate hundreds of users. It hides the complexity of process distribution from the user, while being able to complete complex tasks through message queuing, application implementation, and data staging or the storage of data before being uploaded to the data warehouse.

As in two tiered architectures, the top level is the user system interface (client) and the bottom level is performs database management. The database management level ensures data consistency by using features like data locking and replication. Data locking is also referred to as file or record locking. This is a first-come, first-serve DBMS feature used to manage data and updates in a multi-user environment. The first user to access a file or record denies any other user

access or “locks it”. It opens up again and becomes accessible to other users once the update is complete.

The middle tier is also called the application server. It contains a centralized processing logic, which facilitates management and administration. Localizing system functionality in the middle tier makes it possible for processing changes and updates to be made once and be distributed throughout the network available to both clients and servers. Sometimes the middle tier is divided into two or more units with different functions. This makes it a multi-layer model.

For example, in web applications, the client side is usually written in HTML meanwhile the application servers are usually written in C++ or Java. By using a scripting language embedded in HTML, web servers act as translation layers that allow for communication between the client and server layers.

This layer receives requests from clients and generates HTML responses after requesting it from database servers. Popular scripting languages include JavaScript, ASP (Active Server Page), JSP (JavaScript Pages), PHP (Hypertext Preprocessor), Perl (Practical Extraction and Reporting Language), and Python. One of the major benefits of three tier architecture is the ability to partition software and “drag and drop” modules onto different computers in a network.

RPC Overview

The space environment is of critical importance to our nation's technological assets, both to those situated in space as well as on the ground. We are becoming increasingly dependent upon orbiting satellites for applications such as communication networks, global positioning for ship and airline navigation, and monitoring the Earth for research and weather forecasting. Astronauts frequently travel into space on the shuttle, and teams of astronauts will soon be constructing the International Space Station.

As our use of space increases, so must our ability to predict conditions in space, to safeguard human lives, and to protect the national investment in technological systems. In addition, many of our activities on the ground, including communication and electric power distribution, are affected strongly by changing conditions in space.

The near-Earth regions of space are driven by the Sun, and vary from minute-to-minute and day-to-day within the eleven-year cycle of solar activity. Like seasonal variations in the terrestrial weather, each stage of the solar cycle is characterized by its own set of conditions which affect different sectors of human and technological activity. Unlike terrestrial weather conditions which are monitored routinely at thousands of locations around the world, the conditions in space are monitored by only a handful of space-based and ground-based facilities.

Space weather forecasters are required to specify and to predict conditions in space using a minimum of guidance from actual measurements. This extreme under-sampling of the diverse, coupled regions of space, extending all the way from the Sun to the Earth, demands that models be utilized to provide a continuous, quantitative assessment of the geospace environment.

The vision for future space environment services is to utilize a suite of real-time, data-driven, operational models that provide quantitative predictions of conditions throughout near-Earth space. The regions of interest include the Sun, the solar wind, the terrestrial magnetosphere, the ionosphere, and the upper atmosphere. Future operational models, together with the forecasters' expertise, will make possible an accurate and comprehensive set of products to better serve our nation's needs.

This modeling capability will also enable the crucial training of personnel in preparation for the next peak in solar activity, using simulated conditions from previous disturbed periods. Extensive research, modeling, and monitoring efforts directed at understanding the space environment have produced a broad spectrum of data and model resources. In the United States, various aspects of this effort have been funded by NASA, the National Science Foundation, NOAA, the Department of Defense, the Department of Energy, and the Department of Interior.

Most recently, an interagency effort has been initiated to fund research targeted specifically toward understanding and predicting the space environment. This initiative, the National Space Weather Program, stems from the broad interest in space shared by commercial, educational, and governmental organizations. A primary goal of the National Space Weather Program is to focus and to build on our existing resources to produce quantitative predictive models of the space environment.

Research relevant to the national Space Weather Program also benefits from strong international collaboration among scientists. In addition, SWPC Space Weather Operations, jointly operated with the USAF, also interacts directly with the international regional warning centers that operate under the International Space Environment Service. The extensive model and data resources that exist and are being developed, together with our increasing need to monitor and to predict the space environment, present the imperative to transition these resources into operational use in a manner that most efficiently serves national needs. Model and data resources need to be critically evaluated through a competitive process that will insure their rapid and flexible utilization of the best and most useful. This transitioning from research to operations will be accomplished at the SWPCRapidPrototypingCenter, where staff will work directly with modelers, data providers, service providers and end users to insure an efficient environment to test and transition these products.

The ongoing direct interaction between the SWPC staff and the broad user community will influence future research and development efforts funded by various government agencies, and especially those funded through the National Space Weather Program. Because of the broad range of basic research questions that remain, guidance from the space weather operations centers and the end-user community will be required to identify the most immediate needs and the most promising opportunities. The continuous feedback between the users of space environment services and the model and data transitioning at SWPC will provide direction to the funding agencies and to the researchers they support.

RPC Implementation

An implementation of the RPC model usually consists of at least three elements, a language compiler, a client runtime library, and a server runtime library. The language compiler produces suitable client and server stubs from a program written in an RPC language which usually a non-procedural language is providing the capability of declaring remote procedures and their parameters. In conjunction with the client and server applications, the client and server stubs are compiled by a procedure language compiler, such as C, producing object files which are linked to the client and server runtime libraries. This process produces an executable client and an executable server.

The client and server runtime libraries are called by the client and server stubs respectively. These object runtime libraries contain the routines for performing conversion between the local data representations and the common data representation, for creating the network message formats, and for the transmission of these messages between client and server according to user-specified protocols.

With such an implementation, the developer of an RPC application is required to produce the following:

- The RPC language program for the RPC language compiler.
- The client application which calls the client stub.
- The server application which is called by the server stub.

```
long binop_add(a,b)
long a,b;
{
    return(a + b);
}
```

“The procedure *binop_add*”

In this report, a binary addition application is used as an example to illustrate these concepts. Consider a procedure named *binop_add* which adds two binary integer numbers. It has as input parameters two integers *a* and *b*. The return value of the procedure is the sum of *a* and *b*. In C, a call to such a procedure is:

```
c = binop_add(a,b);
```

In C, the procedure *binop_add* is implemented as shown in Figure 2. Following traditional practice, a developer wishing to use the procedure *binop_add* in an application simply calls the procedure and that procedure is linked into the application.

The goal of an RPC implementation is to achieve the same effect as the traditional method of using procedures except that the procedure *binop_add* is run on a separate system. Ideally, when *binop_add* is an RPC, the only additional task required of the developer is to produce

declarations for *binop_add* and its parameters in the RPC language. The RPC language compiler generates the client stub which is called by the statement:

```
c = binop_add(a,b);
```

In addition, the RPC language compiler produces a server stub which calls the *binop_add* procedure in Figure 2. See Figures 3, 4, and 5 for the RPC language needed to define the *binop_add* example for ONC RPC, DCE RPC, and ISO RPC respectively.

The RPCGEN Protocol

The details of programming applications to use Remote Procedure Calls can be overwhelming. Perhaps most daunting is the writing of the XDR routines necessary to convert procedure arguments and results into their network format and vice-versa.

Fortunately, `rpcgen(1)` exists to help programmers write RPC applications simply and directly. `rpcgen` does most of the dirty work, allowing programmers to debug the main features of their application, instead of requiring them to spend most of their time debugging their network interface code. `rpcgen` is a compiler. It accepts a remote program interface definition written in a language, called RPC Language, which is similar to C. It produces a C language output which includes stub versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, and a header file that contains common definitions. The client stubs interface with the RPC library and effectively hide the network from their callers. The server stub similarly hides the network from the server procedures that are to be invoked by remote clients. `rpcgen`'s output files can be compiled and linked in the usual way. The developer writes server procedures—in any language that observes Sun calling conventions—and links them with the server skeleton produced by `rpcgen` to get an executable server program. To use a remote program, a programmer writes an ordinary main program that makes local procedure calls to the client stubs produced by `rpcgen`. Linking this program with `rpcgen`'s stubs creates an executable program. (At present the main program must be written in C). `rpcgen` options can be used to suppress stub generation and to specify the transport to be used by the server stub. Like all compilers, `rpcgen` reduces development time that would otherwise be spent coding and debugging low-level routines. All compilers, including `rpcgen`, do this at a small cost in efficiency and flexibility.

However, many compilers allow escape hatches for programmers to mix low-level code with high-level code. `rpcgen` is no exception. In speed-critical applications, hand-written routines can be linked with the `rpcgen` output without any difficulty. Also, one may proceed by using `rpcgen` output as a starting point, and then rewriting it as necessary.

```
* msg.x: Remote message printing protocol
*/
```

```
program MESSAGEPROG {
```

```

version MESSAGEVERS {
int PRINTMESSAGE(string) = 1;
}=1;
} = 99;

```

Remote procedures are part of remote programs, so we actually declared an entire remote program here which contains the single procedure PRINTMESSAGE. This procedure was declared to be in version 1 of the remote program. No null procedure (procedure 0) is necessary because rpcgen generates it automatically. Notice that everything is declared with all capital letters. This is not required, but is a good convention to follow.

Notice also that the argument type is “string” and not “char *”. This is because a “char *” in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a “string”. There are just two more things to write. First, there is the remote procedure itself. Here’s the definition of a remote procedure to implement the PRINTMESSAGE procedure we declared above:

```

/*
 * msg_proc.c: implementation of the remote procedure "printmessage"
 */

#include <stdio.h>
#include <rpc/rpc.h> /* always needed */
#include "msg.h" /* need this too: msg.h will be generated by rpcgen */
/*
 * Remote version of "printmessage"
 */
int *
printmessage_1(msg)
char **msg;
{
static int result; /* must be static! */
FILE *f;
f = fopen("/dev/console", "w");
if (f == NULL) {
result = 0;
return (&result);
}
fprintf(f, "%s\n", *msg);
fclose(f);
result = 1;
return (&result);
}

```

RPCGEN Programming Guide

Notice here that the declaration of the remote procedure `printmessage_1()` differs from that of the local procedure `printmessage()` in three ways:

1. It takes a pointer to a string instead of a string itself. This is true of all remote procedures: they always take pointers to their arguments rather than the arguments themselves.
2. It returns a pointer to an integer instead of an integer itself. This is also generally true of remote procedures: they always return a pointer to their results.
3. It has “_1” appended to its name. In general, all remote procedures called by RPCGEN are named by the following rule: the name in the program definition (here `PRINTMESSAGE`) is converted to all lower-case letters, an underbar (“_”) is appended to it, and finally the version number (here 1) is appended. The last thing to do is declare the main client program that will call the remote procedure.

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */

#include <stdio.h>
#include <rpc/rpc.h> /* always needed */
#include "msg.h" /* need this too: msg.h will be generated by rpcgen */
main(argc, argv)
int argc;
char *argv[];
{
    CLIENT *client;
    int *rs;
    char *server;
    char *msg;
    if (argc < 3) {
        fprintf(stderr, "usage: %s host message\n", argv[0]);
        exit(1);
    }
    /*
     * Save values of command line arguments
     */
    server = argv[1];
    msg = argv[2];
    /*
     * Create client "handle" used for calling MESSAGEPROG on the
     * server designated on the command line. We tell the RPC package
     * to use the "tcp" protocol when contacting the server.
     */
```

```

client = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
if (client == NULL) {
/*
 * Couldn't establish connection with server.
 * Print error message and die.
 */
clnt_pcreateerror(server);
exit(1);
}
/*
 * Call the remote procedure "printmessage" on the server
 */
rs = printmessage_1(&msg, client);
if (rs == NULL) {
/*
 * An error occurred while calling the server.
 * Print error message and die.
 */
clnt_perror(cl, server);
exit(1);
}
}
Page 6 rpcgen Programming Guide
/*
 * Okay, we successfully called the remote procedure.
 */
if (*rs == 0) {
/*
 * Server was unable to print our message.
 * Print error message and die.
 */
fprintf(stderr, "%s: %s couldn't print your message\n",
argv[0], server);
exit(1);
}
/*
 * The message got printed on the server's console
 */
printf("Message delivered to %s!\n", server);
}

```

There are two things to note here:

1. First a client “handle” is created using the RPC library routine `clnt_create()`. This client handle will be passed to the stub routines which call the remote procedure.

2. The remote procedure `printmessage_1()` is called exactly the same way as it is declared in `msg_proc.c` except for the inserted client handle as the first argument.

Here's how to put all of the pieces together:

```
example% rpcgenmsg.x
example% cc rprintmsg.cmsg_clnt.c -o rprintmsg
example% cc msg_proc.cmsg_svc.c -o msg_server
```

Two programs were compiled here: the client program `rprintmsg` and the server program `msg_server`. Before doing this though, `rpcgen` was used to fill in the missing pieces.

Here is what `rpcgen` did with the input file `msg.x`:

1. It created a header file called `msg.h` that contained `#define`'s for `MESSAGEPROG`, `MESSAGEVERS` and `PRINTMESSAGE` for use in the other modules.
2. It created client "stub" routines in the `msg_clnt.c` file. In this case there is only one, the `printmessage_1()` that was referred to from the `printmsg` client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is `FOO.x`, the client stubs output file is called `FOO_clnt.c`.
3. It created the server program which calls `printmessage_1()` in `msg_proc.c`. This server program is named `msg_svc.c`.

PRACTICAL - 04

Implement Power RPC using RPCGEN.

power.x:

```
struct Variables
{
int a;
int b;
};
program POWER_PROG

{

version POWER_VERSION

    {
        int power1(Variables)=1;
    }=1;

}=0x20000000;
```

power_client.c :

```
#include "power.h"

void power_prog_1(char *host)
{
    CLIENT *clnt;
    int *result_1;
    Variables power1_1_arg;

#ifdef DEBUG
    clnt = clnt_create (host, POWER_PROG, POWER_VERSION, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */
    power1_1_arg.x=5;
    power1_1_arg.y=2;
    result_1 = power1_1(&power1_1_arg, clnt);
    printf("power(a,b)=%d",*result_1);
    if (result_1 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
}
```



```

    }
#endif DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
}

int main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    power_prog_1 (host);
    exit (0);
}

```

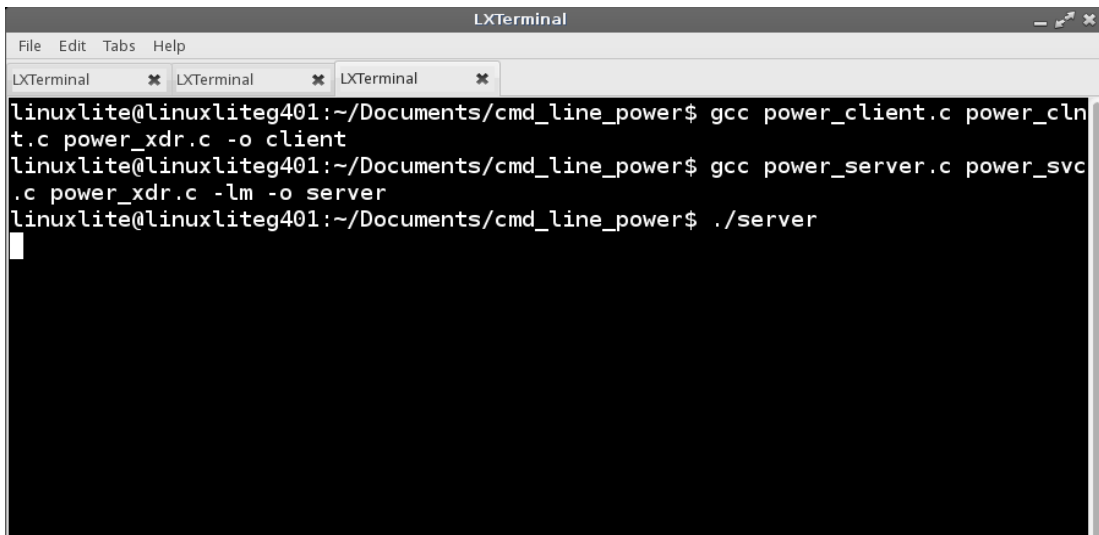
power_server.c :

```

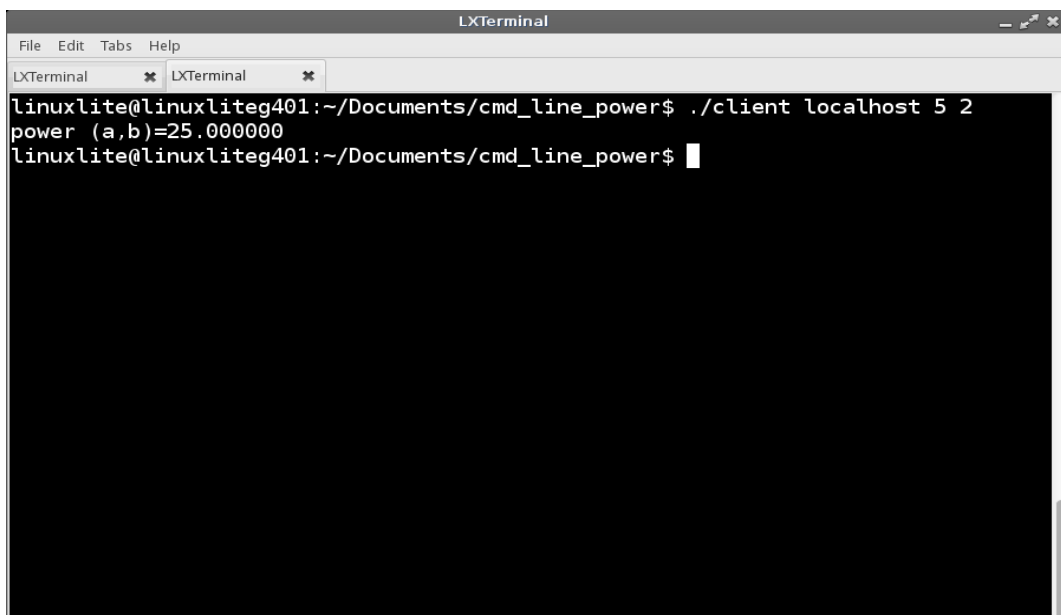
#include "power.h"

int * fpower1_1_svc(Variables *argp, struct svc_req *rqstp)
{
    static int result;
    /*
     * insert server code here
     */
    int i1;
    result=argp->x;
    for(i1=1;i1<argp->y;i1++)
        result*=argp->x;
    return &result;
}

```

OUTPUT:

```
LXTerminal
File Edit Tabs Help
LXTerminal LXTerminal LXTerminal
linuxlite@linuxliteg401:~/Documents/cmd_line_power$ gcc power_client.c power_client.c power_xdr.c -o client
linuxlite@linuxliteg401:~/Documents/cmd_line_power$ gcc power_server.c power_server.c power_xdr.c -lm -o server
linuxlite@linuxliteg401:~/Documents/cmd_line_power$ ./server
```



```
LXTerminal
File Edit Tabs Help
LXTerminal LXTerminal
linuxlite@linuxliteg401:~/Documents/cmd_line_power$ ./client localhost 5 2
power (a,b)=25.000000
linuxlite@linuxliteg401:~/Documents/cmd_line_power$
```

PRACTICAL - 05

Write a program to implement basic calculator using RPCGEN.

calc.x

```
struct Variables
{
    int x;
    int y;
};
program CALC_PROG
{
    version CALC_VERSION
    {
        int add(Variables)=1;
        int sub(Variables)=2;
        int mul(Variables)=3;
        int div(Variables)=4;
    }=1;
}=0x20000000;
```

calc_client.c:

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calc.h"

Void calc_prog_1(char *host)
{
    CLIENT *clnt;
    Variables a1;

    int *ans_1;
    int *ans_2;
    int *ans_3;
    int *ans_4;

#ifdef DEBUG
    clnt = clnt_create(host, CALC_PROG, CALC_VERSION, "udp");
    if (clnt == NULL) {
```

```

        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */

    a1.x=10;
    a1.y=10;

    ans_1 = add_1(&a1, clnt);
    if (ans_1 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("%d + %d = %d",a1.x,a1.y*ans_1);

    ans_2 = sub_1(&a1, clnt);
    if (ans_2 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("%d - %d = %d",a1.x,a1.y*ans_2);

    ans_3 = mul_1(&a1, clnt);
    if (ans_3 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("%d * %d = %d",a1.x,a1.y*ans_3);

    ans_4 = div_1(&a1, clnt);
    if (ans_4 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("%d / %d = %d",a1.x,a1.y*ans_4);

#ifdef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
}

int main (int argc, char *argv[])
{
    char *host;

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

```

```

        host = argv[1];
        calc_prog_1 (host);
    exit (0);
}

```

calc_server.c:

```

/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calc.h"

int * add_1_svc(Data *argp, struct svc_req *rqstp)
{
    static int  ans;

    /*
     * insert server code here
     */
    printf("adding %d of %d",argp->x,argp->y);
    ans=argp->x+argp->y;
    return &ans;
}

int * sub_1_svc(Data *argp, struct svc_req *rqstp)
{
    static int  ans;

    /*
     * insert server code here
     */
    printf("adding %d of %d",argp->x,argp->y);
    ans=argp->x-argp->y;
    return &ans;
}

int * mul_1_svc(Data *argp, struct svc_req *rqstp)
{
    static int  ans;

    /*
     * insert server code here
     */

```

```

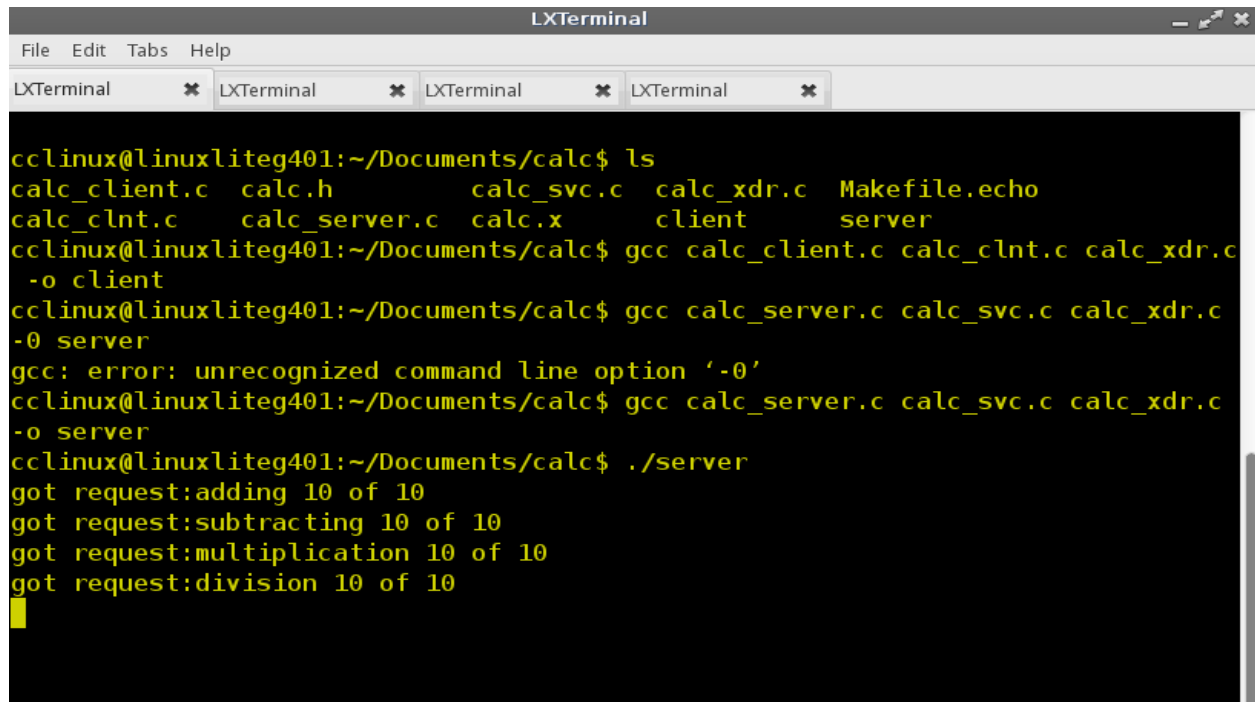
    printf("adding %d of %d",argp->x,argp->y);
    ans=argp->x*argp->y;
    return &ans;
}

int * div_1_svc(Data *argp, struct svc_req *rqstp)
{
    static int ans;

    /*
     * insert server code here
     */
    printf("adding %d of %d",argp->x,argp->y);
    ans=argp->x/argp->y;
    return &ans;
}

```

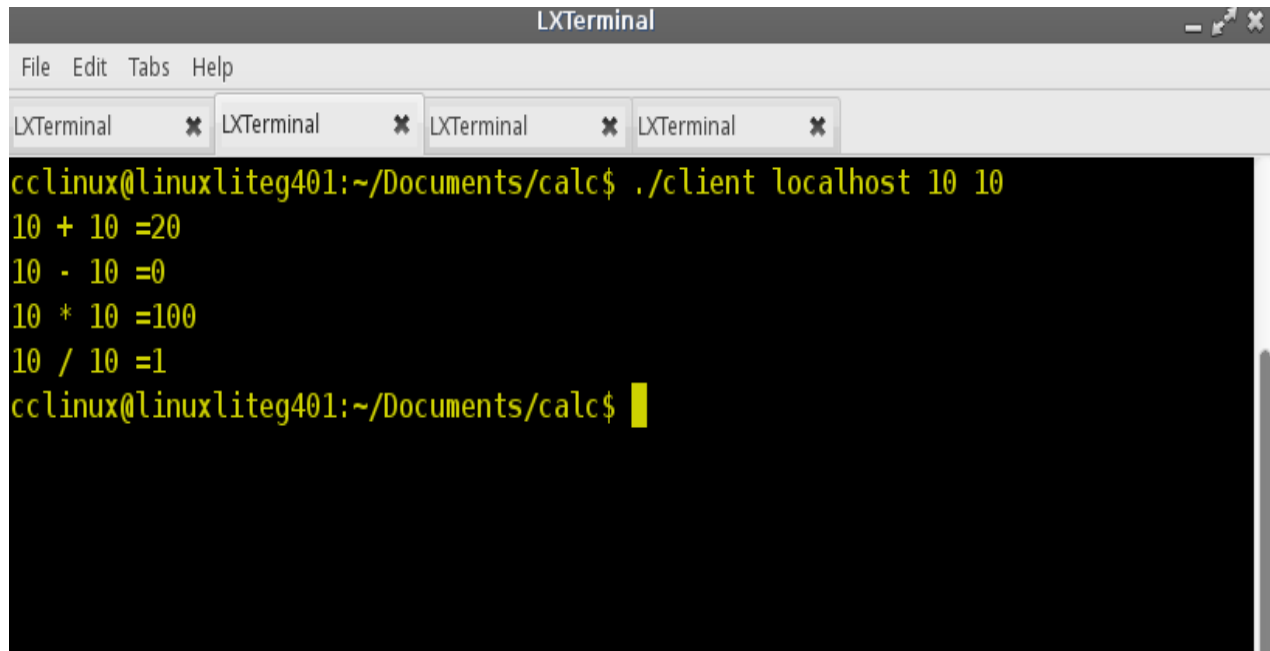
OUTPUT



```

cclinux@linuxliteg401:~/Documents/calc$ ls
calc_client.c  calc.h          calc_svc.c  calc_xdr.c  Makefile.echo
calc_clnt.c   calc_server.c  calc.x      client      server
cclinux@linuxliteg401:~/Documents/calc$ gcc calc_client.c calc_clnt.c calc_xdr.c
-o client
cclinux@linuxliteg401:~/Documents/calc$ gcc calc_server.c calc_svc.c calc_xdr.c
-o server
gcc: error: unrecognized command line option '-o'
cclinux@linuxliteg401:~/Documents/calc$ gcc calc_server.c calc_svc.c calc_xdr.c
-o server
cclinux@linuxliteg401:~/Documents/calc$ ./server
got request:adding 10 of 10
got request:subtracting 10 of 10
got request:multiplication 10 of 10
got request:division 10 of 10

```



The image shows a screenshot of an LXTerminal window. The window has a title bar labeled "LXTerminal" and a menu bar with "File", "Edit", "Tabs", and "Help". Below the menu bar, there are four tabs, each labeled "LXTerminal" with a close button. The terminal content is as follows:

```
cclinux@linuxliteg401:~/Documents/calculator$ ./client localhost 10 10
10 + 10 =20
10 - 10 =0
10 * 10 =100
10 / 10 =1
cclinux@linuxliteg401:~/Documents/calculator$
```

PRACTICAL - 06

Write a program to implement echo server using RPCGEN.

echo.x :

```
struct variables
{
char a[20];
};
program ECHO_PROG
{
version ECHO_VERS
{
rpcecho ECHO(rpcecho)=1;
} = 1;
} = 0x234511;
```

echo_client.c :

```
#include "echo.h"
#include "echo_xdr.c"
#include "echo_clnt.c"
Void echo_prog_1(char *host)
{
CLIENT *clnt;
rpcecho *ans_1;
rpcecho echo_1_arg1;

#ifdef DEBUG
clnt = clnt_create (host, ECHO_PROG, ECHO_VERS, "udp");
if (clnt == NULL) {
clnt_pcreateerror (host);
exit (1);
}
#endif /* DEBUG */

printf("In Client:");
scanf("%s", echo_1_arg1.a);
ans_1 = echo_1(echo_1_arg1, clnt);
if (ans_1 == (rpcecho *) NULL) {
clnt_perror (clnt, "ERROR!!");
}
else
{
```



```

printf("Reply=>%s\n",ans_1);
}
#ifdef DEBUG
clnt_destroy (clnt);
#endif /* DEBUG */
}

int main (int argc, char *argv[])
{
char *host;
if (argc< 2) {
printf ("usage: %s server_host\n", argv[0]);
exit (1);
}
host = argv[1];
echo_prog_1 (host);
exit (0);
}

```

echo_server.c :

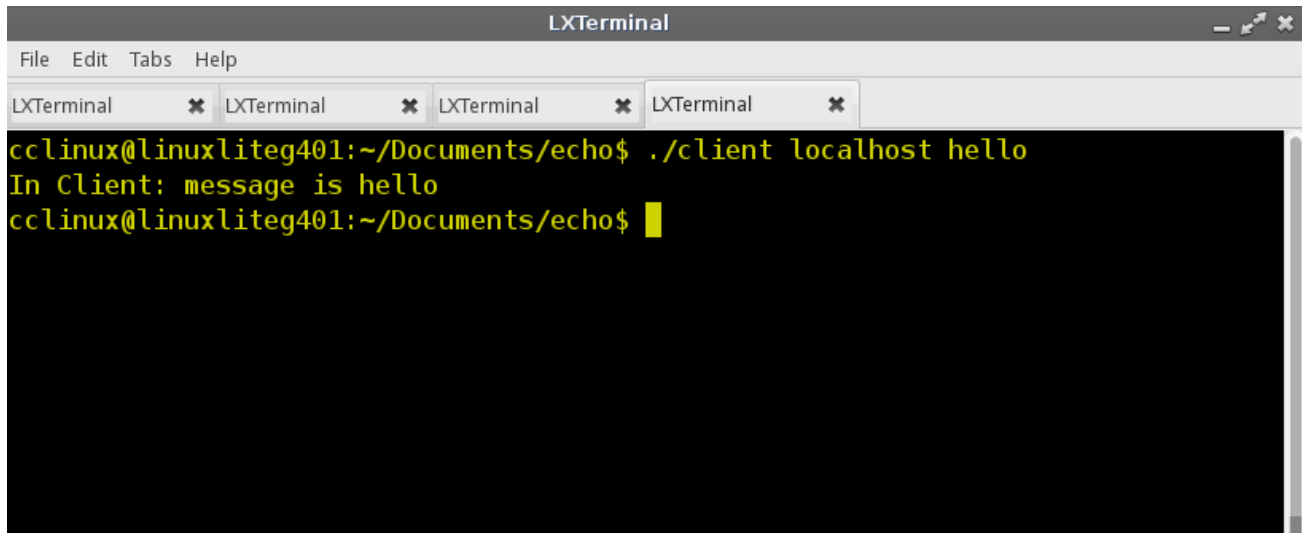
```

#include "echo.h"
#include "echo_xdr.c"
#include "echo_svc.c"

rpcecho * echo_1_svc(rpcecho arg1, struct svc_req *rqstp)
{
static rpcecho ans;
ans=arg1;
printf("In server :"+ans);
return &ans;
}

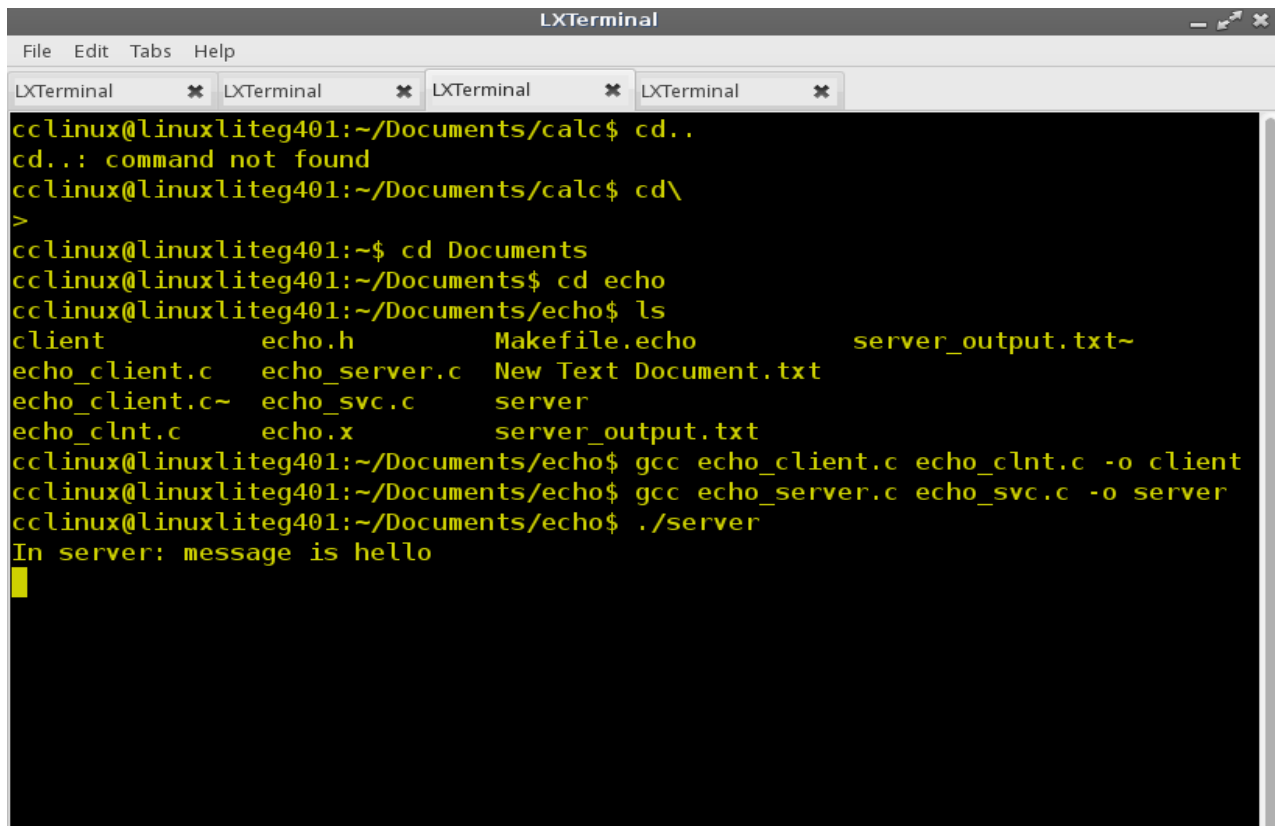
```

OUTPUT



The screenshot shows an LXTerminal window with a menu bar (File, Edit, Tabs, Help) and four tabs, all labeled 'LXTerminal'. The terminal content shows a user 'cclinux' at 'linuxliteg401' in the directory '~/Documents/echo'. They run the command './client localhost hello'. The output is 'In Client: message is hello'. The prompt returns to the shell.

```
cclinux@linuxliteg401:~/Documents/echo$ ./client localhost hello
In Client: message is hello
cclinux@linuxliteg401:~/Documents/echo$
```



The screenshot shows an LXTerminal window with a menu bar (File, Edit, Tabs, Help) and four tabs, all labeled 'LXTerminal'. The terminal content shows a user 'cclinux' at 'linuxliteg401' in the directory '~/Documents/echo'. They run the command './server'. The output is 'In server: message is hello'. The prompt returns to the shell.

```
cclinux@linuxliteg401:~/Documents/echo$ ./server
In server: message is hello
cclinux@linuxliteg401:~/Documents/echo$
```

PRACTICAL – 07

AIM: Implement HELLO WORLD services using RMI.

Program:

HelloWorld.java :

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface HelloWorld extends Remote
{
    String helloworld() throws RemoteException;
}
```

HelloWorldServer.java :

```
import java.rmi.*;
import java.net.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;
public class HelloWorldServer extends UnicastRemoteObject implements HelloWorld
{
    public HelloWorldServer() throws RemoteException
    {
        super();
    }
    public String helloworld()
    {
        System.out.println("helloworld : Successful!");
        return "HelloWorld from Server!";
    }
    public static void main(String args[])
    {
        try
        {
            HelloWorldServerobj=new HelloWorldServer();
            Naming.rebind("HelloWorld",obj);
            System.out.println("Bound in registry");
        }
        catch(Exception e)
        {
            ;
        }
    }
}
```

```

System.out.println("HelloWorldServer exception:"+e.getMessage());
e.printStackTrace();
    }
}
}

```

HelloWorldClient.java :

```

import java.rmi.*;
import java.io.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
public class HelloWorldClient
{
    static String message="blank";
    static HelloWorldobj=null;

    public static void main(String args[])
    {
        try
        {
            obj=(HelloWorld)Naming.lookup("//198.168.0.102+"/"HelloWorld");
            message=obj.helloworld();
            System.out.println("Msg from the Server:\\"+message+"\\");
        }
        catch(Exception e)
        {
            System.out.println("HelloWorldClient Error:"+e.getMessage());
            e.printStackTrace();
        }
    }
}

```

Output:**Server Side:**

```
C:\>set PATH="C:\ProgramFiles\Java\jdk1.6.0_06\bin";  
C:\>cd "RMI Old"  
C:\RMI Old>start rmiregistry  
C:\RMI Old>javac HelloWorldServer.java  
C:\RMI Old>java HelloWorldServer
```

Bound in registry.....
helloworld : Successful!

Client Side:

```
C:\>set PATH="C:\ProgramFiles\Java\jdk1.6.0_06\bin";  
C:\>cd "RMI Old"  
C:\RMI Old>Hello>javac HelloWorldClient.java  
C:\RMI Old>Hello>java HelloWorldClient 192.168.0.102  
Msg from the Server:"HelloWorld from Server!"
```

PRACTICAL -08

AIM: Implement Calculator using RMI.

Program:

Calculator.java:

```
public interface Calculator extends java.rmi.Remote {  
    public long add(long a, long b) throws java.rmi.RemoteException;  
    public long sub(long a, long b) throws java.rmi.RemoteException;  
    public long mul(long a, long b) throws java.rmi.RemoteException;  
    public long div(long a, long b) throws java.rmi.RemoteException;  
}
```

CalculatorImp.java:

```
public class CalculatorImpl extends java.rmi.server.UnicastRemoteObject implements Calculator  
{  
    public CalculatorImpl() throws java.rmi.RemoteException {  
        super();  
    }  
    public long add(long w, long q) throws java.rmi.RemoteException {  
        return w + q;  
    }  
    public long sub(long w, long q) throws java.rmi.RemoteException {  
        return w - q;  
    }  
    public long mul(long w, long q) throws java.rmi.RemoteException {  
        return w * q;  
    }  
    public long div(long a, long b) throws java.rmi.RemoteException {  
        return w / q;  
    }  
}
```

CalculatorClient.java:

```
Import java.rmi.Naming;  
Import java.rmi.RemoteException;  
Import java.net.MalformedURLException;  
Import java.rmi.NotBoundException;  
public class CalculatorClient {  
    public static void main(String[] args) {
```

```

try {
    Calculator cal = (Calculator)
Naming.lookup("rmi://localhost/CalculatorService");
System.out.println(Substraction=cal.sub(10, 2) );
System.out.println(Addition=cal.add(64, 46) );
System.out.println(Multiplication=cal.mul(12, 3) );
System.out.println(Division=cal.div(55, 5) );
    }
catch (MalformedURLExceptionmurle) {
System.out.println();
System.out.println("MalformedURLException");
System.out.println(murle);
    }
catch (RemoteException re) {
System.out.println();
System.out.println("RemoteException");
System.out.println(re);
    }
catch (NotBoundExceptionnbe) {
System.out.println();
System.out.println("NotBoundException");
System.out.println(nbe);
    }
catch (java.lang.ArithmeticExceptionae) {
System.out.println();
System.out.println("java.lang.ArithmeticException");
System.out.println(ae);
    }
}

```

CalculatorServer.java:

```

import java.rmi.Naming;
public class CalculatorServer {
public CalculatorServer() {
try {
    Calculator c = new CalculatorImpl();
Naming.rebind("rmi://localhost:1099/CalculatorService", c);
    } catch (Exception e) {
System.out.println("Trouble: " + e);
    }
}
public static void main(String args[]) {
new CalculatorServer();
}
}

```

Output:

```
C:\RMI Old\calculator>start rmiregistry
C:\RMI Old\calculator>javac *.java
C:\RMI Old\calculator>java CalculatorServer

C:\RMI Old\calculator>javac *.java
C:\RMI Old\calculator>javac CalculatorClient.java
C:\RMI Old\calculator>java CalculatorClient
Substraction=8
Addition=100
Multiplication=36
Division=11
```


PRACTICAL – 09

AIM: Implement simple thread creation to print HELLO using pthread.

Program:

```
#include<pthread.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>

#define NUM_THREADS 5

void *printhello(void *threadid)
{
    Long thid;
    thid=(long)threadid;
    printf("Hello How Are You?%ld\n",thid);
    pthread_exit(NULL);
}

int main(intargc,char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0;t<NUM_THREADS;t++)
    {
        printf("in main creating thread %ld \n",t);
        rc=pthread_create(&threads[t],NULL,printhello,(void *)t);
        if(rc)
        {printf("error %d",rc);
        exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Output:

```
[exam2@LINTEL exam2]$ gcc -lpthread tarang.c
[exam2@LINTEL exam2]$ ./a.out
in main creating thread 0
Hello How Are You? 0
in main creating thread 1
Hello How Are You? 1
in main creating thread 2
Hello How Are You? 2
in main creating thread 3
Hello How Are You? 3
in main creating thread 4
Hello How Are You? 4
[exam2@LINTEL exam2]$
```

PRACTICAL - 10

AIM: Implement calculator using pthread library.

Program:

```
#include<stdio.h>
#include<pthread.h>
#define _GNU_SOURCE

void *Add();
void *Sub();
void *Mul();
void *Div();
int main()
{
pthread_t t1,t2,t3,t4;
pthread_attr_t th1,th2,th3,th4;
pthread_attr_init(&th1);
pthread_attr_init(&th2);
pthread_attr_init(&th3);
pthread_attr_init(&th4);
pthread_create(&t1,&th1,&Add,NULL);
pthread_create(&t2,&th2,&Sub,NULL);
pthread_create(&t3,&th3,&Mul,NULL);
pthread_create(&t4,&th4,&Div,NULL);

pthread_join(t1,NULL);
pthread_join(t2,NULL);
pthread_join(t3,NULL);
pthread_join(t4,NULL);
}
void *add()
{
int a,b,sum;
a=25;b=5;
sum=a+b;
printf("\n Addition=%d",sum);
}

void *mul()
{
int a,b,mul;
a=20;b=5;
```

```

mul=a*b;
printf("\n Multiplication=%d",mul);
}
void *sub()
{
int a,b,sub;
a=25;b=5;
sub=a-b;
printf("\n SubstractionN=%d",sub);
}

```

```

void *div()
{
int a,b,div;
a=25;b=5;
div=a/b;
printf("\n Division=%d",div);
}

```

Output:

```

[exam2@LINTEL exam2]$ gcc -lpthread calc.c
[exam2@LINTEL exam2]$ ./a.out

```

```

Addition=30
Subtraction=20
Multiplication=125
Division=5[exam2@LINTEL exam2]$

```

PRACTICAL -11

AIM: Implement Producer Consumer problem using mutex primitive of pthread library.

Program:

```
# include <stdio.h>
# include <pthread.h>
# define BufferSize 10

void *Producer();
void *Consumer();

int BuffIndx=0;
char *BUFF;

pthread_cond_tBuffer_Not_Full=PTHREAD_COND_INITIALIZER;
pthread_cond_tBuffer_Not_Empty=PTHREAD_COND_INITIALIZER;
pthread_mutex_tmutexVar=PTHREAD_MUTEX_INITIALIZER;

int main()
{
pthread_tptid,ctid;

    BUFF=(char *) malloc(sizeof(char) * BufferSize);

pthread_create(&ptid,NULL,Producer,NULL);
pthread_create(&ctid,NULL,Consumer,NULL);

pthread_join(ptid,NULL);
pthread_join(ctid,NULL);

return 0;
}

void *Producer()
{
for(;;)
{
pthread_mutex_lock(&mutexVar);
if(BuffIndx==BufferSize)
{
```

```

pthread_cond_wait(&Buffer_Not_Full,&mVar);
    }
    BUFFER[BuffIndx++]='@';
    printf("Produce : %d \n",BuffIndx);
    pthread_mutex_unlock(&mVar);
    pthread_cond_signal(&Buffer_Not_Empty);
}

}

void *Consumer()
{
for(;;)
{
pthread_mutex_lock(&mVar);
if(BuffIndx==-1)
{
pthread_cond_wait(&Buffer_Not_Empty,&mVar);
}
printf("Consume : %d \n",BuffIndx--);
pthread_mutex_unlock(&mVar);
pthread_cond_signal(&Buffer_Not_Full);
}
}

```

Output:

```

[exam2@LINTEL exam2]$ cc -o Prog08 -lpthread Prog08.c
[exam2@LINTEL exam2]$ ./Prog08
Produce : 1
Produce : 2
Produce : 3
Produce : 4
Produce : 5
Produce : 6
Produce : 7
Produce : 8
Produce : 9
Produce : 10
Consume : 10
Consume : 9
Consume : 8
Consume : 7
Consume : 6
Consume : 5
Consume : 4

```

Consume : 3
Consume : 2
Consume : 1
Consume : 0
Produce : 0
Produce : 1
Produce : 2
Produce : 3
Produce : 4
Produce : 5
Produce : 6
Produce : 7
Produce : 8
Produce : 9
Produce : 10
Consume : 10
Consume : 9
Consume : 8
Consume : 7
Consume : 6
Consume : 5
Consume : 4
Consume : 3
Consume : 2
Consume : 1
Consume : 0
Produce : 0
Produce : 1

BEYOND SYLLABUS – 01

AIM: Case study of Real Time Distributed Systems, Munin DSM, Amoeba, Chorus, Mach, CORBA, DCE Distributed file service, DTS & NTP.

Munin DSM:

Software distributed shared memory DSM is a software abstraction of shared memory on a distributed memory machine. The key problem in building a client DSM system is to reduce the amount of communication needed to keep the distributed memories consistent. The Munin DSM system incorporates a number of novel techniques for doing so, including the use of multiple consistency protocols and support for multiple concurrent writer protocols. Due to these and other features, Munin is able to achieve high performance on a variety of numerical applications.

The core of the Munin system is the runtime library that contains the fault handling, thread support, synchronization, and other runtime mechanisms. It consists of approximately 15 lines of C source code that create an 8 kilobyte library that is linked into each Munin program. Each node of an executing Munin program consists of a collection of Munin runtime threads that handle consistency and synchronization operations and one or more user threads performing the parallel computation. Munin programmers write parallel programs using threads as they would on a uniprocessor or shared memory multiprocessor.

AMOEBAS:

The aim of the Amoeba project is to build a [timesharing](#) system that makes an entire network of computers appear to the user as a [single machine](#). Amoeba is a distributed operating system designed to connect together a large number of machines in a transparent way. Its goal is to make the entire system look to the users like a single computer. The system consists of two parts: a microkernel and server processes.

An Amoeba system consists of several components, including a pool of processors where most of the work is done, terminals that handle the user interface, and specialized servers. All these machines normally run the same microkernel.

The microkernel has four primary functions:

1. Manage processes and threads.
2. Provide low-level memory management support.
3. Support communication.
4. Handle low-level I/O.

Amoeba has processes just like most operating systems have. Processes can have multiple threads of control within a single process, all of which share the process address space and

resources. A thread is the active entity within a process. Each thread has a program counter, and its own stack, and executes sequentially.

One of the most unique feature of Amoeba is that it is based on the idea of objects, each of which is named and protected by a 128-bit capability. When a process creates an object, the server managing the object returns a capability for that object. The capability contains bits telling which of the operations on the object the holder of the capability may perform. A typical capability is shown in figure.

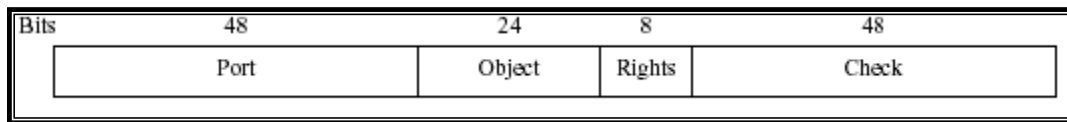


Figure 1 - A typical capability

The *Port* field identifies the server. The *Object* field tells which object is being referred to, since a server normally will manage thousands of objects. The *Rights* field specifies which operations are allowed (e.g., a capability for a file may be read-only). Since capabilities are managed in user space the *Check* field is needed to protect them cryptographically, to prevent users from tampering with them.

Since capabilities are managed by user processes themselves, and can be given away by their owners, the rights are protected from tampering by encryption. As a consequence, different users may have capabilities for the same object, but with different rights.

Design Goals:

Three central design goals were set for the Amoeba distributed operating system:

Network transparency: All resource accesses were to be network transparent. In particular, there was to be a seamless system-wide file system, and processes were to execute at a processor of the system's choosing, without the user's knowledge.

Object-based resource management: The system was designed to be object based. Each resource is regarded as an object and all objects, irrespective of their type, are accessed by a uniform naming scheme. Objects are managed by servers, where they can be accessed only by sending messages to the servers. Even when an object resides locally, it will be accessed by request to a server.

User-level servers: The system software was to be constructed as far as possible as a collection of servers executing at user-level, on top of a standard microkernel that was to run at all computers in the system, regardless of their role. An issue that follows from the last two goals, and to which the Amoeba designers paid particular attention, is that of protection. The Amoeba microkernel supports a uniform model for accessing resources using capabilities.

CHORUS:

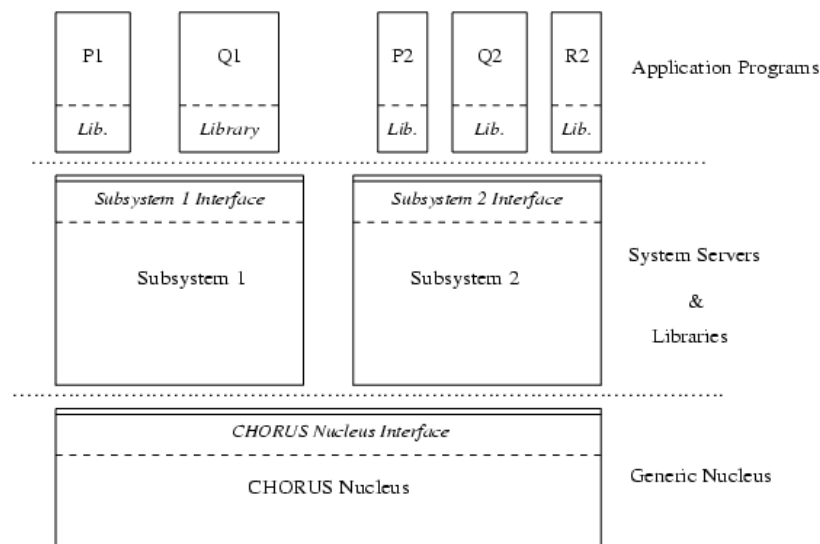
Chorus was a research project on distributed systems at INIRA in France. It was bought by Sun Microsystems in 1997.

The Chorus operating system is a highly scalable and reliable embedded operating system that has established itself among top telecom suppliers. The Chorus operating system is used in public switches and PBXs, as well as within access networks, cross-connect switches, voice-mail systems, cellular base stations, web-phones and cellular telephones.

An open and flexible solution, the Chorus operating system also allows developers to rapidly respond to customer needs and market conditions by quickly and cost-effectively creating and deploying new services and mission-critical applications. With enhanced networking features, the Chorus operating system seamlessly supports third-party protocol stacks, legacy applications, real-time and Java technology-based applications simultaneously on a single hardware platform.

Design Principle

A Chorus system is composed of a small nucleus and a set of system servers, which cooperate in the context of subsystems.



Chorus Nucleus is not the core of a specific operating system; rather it provides generic tools designed to support a variety of host subsystems, which can coexist on top of the Nucleus.

This structure supports application programs, which already run on an existing operating system, by reproducing the operating system's interface within a subsystem.

This classic idea of separating the functions of the operating system into groups of services provided by autonomous servers is central to the Chorus philosophy. In monolithic systems, these functions are usually part of the "kernel". This separation of functions increases modularity, and therefore the portability of the overall system.

Unix is a layered operating system. The innermost layer is the hardware that provides the services for the OS. The operating system, referred to in Unix as the **kernel**, interacts directly with the hardware and provides the services to the user programs. These user programs don't need to know anything about the hardware. They just need to know how to interact with the kernel and it's up to the kernel to provide the desired service. One of the big appeals of Unix to programmers has been that most well written user programs are independent of the underlying hardware, making them readily portable to new systems.

User programs interact with the kernel through a set of standard **system calls**. These system calls request services to be provided by the kernel. Such services would include accessing a file: open close, read, write, link, or execute a file; starting or updating accounting records; changing ownership of a file or directory; changing to a new directory; creating, suspending, or killing a process; enabling access to hardware devices; and setting limits on system resources.

Unix is a **multi-user, multi-tasking** operating system. You can have many users logged into a system simultaneously, each running many programs. It's the kernel's job to keep each process and user separate and to regulate access to system hardware, including cpu, memory, disk and other I/O devices.

MACH:

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional technically advanced system.

Mach is based on five major concepts: processes, threads, ports, messages, and memory objects. A process, as in other systems, is a container for holding threads and other resources that are managed together. A thread is a lightweight process-within-a-process, as in Amoeba. A port is a mailbox that is used for communication. A message is a typed data structure that one thread can send to another thread's port so the receiving thread can read it. Finally, a memory object is a coherent region of memory, all of whose words have certain shared properties and which can be manipulated as a whole.

Mach is an example of an object-oriented system where the data and the operations that manipulate that data are encapsulated into an abstract object. Only the operations of the object are able to act on the entities defined in it. The details of how these operations are implemented are hidden, as are the internal data structures. Thus programmer can use an object only by invoking its defined, exported operations. The object oriented approach supported by Mach allows objects to reside anywhere in a network of Mach systems, transparent to the user. The port mechanism makes all of this possible.

Design Goals:

Multiprocessor operation: Mach was designed to execute on a shared memory multiprocessor, so that both kernel-mode threads and user-mode threads could be executed by any processor. It is designed to run on computer systems ranging from one to thousands of processors.

Operating system emulation: To support the binary-level emulation of UNIX and other operating systems, Mach allows for the transparent redirection of operating system calls to emulation library calls and thence to user-level operating system servers.

Flexible virtual memory implementation: Mach provides the ability to layer emulation of other operating systems as well, and they can even run concurrently.

Portability: Mach was designed to be portable to a variety of hardware platforms. For this reason, machine-dependent code has been isolated as far as possible. In particular, the virtual memory code has been divided between machine independent and machine-dependent parts

Support for diverse architectures: Mach support for diverse architectures including multiprocessors with varying degrees of shared memory access: Uniform Memory Access (UMA), Non-Uniform Memory Access (NUMA), and No Remote Memory Access (NORMA).

Simplified kernel structure with a small number of abstractions: In turn these abstractions are sufficiently general to allow other operating systems to be implemented on top of Mach

Network transparency: Distributed operation, providing network transparency to clients and an object-oriented organization both internally and externally

Integrated memory management and inter-process communication: Integrated memory management and IPC in Mach to provide both efficient communications of large numbers of data, as well as communication-based memory management

Heterogeneous system support: Due to heterogeneous support nature of Mach OS makes Mach widely available and interoperable among computer systems from multiple vendors

Compatibility with UNIX: Mach is better able to satisfy the needs of the masses than the others operating systems because it offers full compatibility with UNIX 4.3BSD.

Ability with varying inter computer network speed: Mach OS has an ability to function with varying inter-computer network speeds, from wide area networks to high-speed local-area networks and tightly coupled multiprocessors.

CORBA:

The Common Object Request Broker Architecture (CORBA) [\[OMG:95a\]](#) is an emerging open distributed object computing infrastructure being standardized by the Object Management Group ([OMG](#)). CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and demarshaling; and operation dispatching.

The following figure illustrates the primary components in the OMG Reference Model architecture. Descriptions of these components are available further below. Portions of these descriptions are based on material from [\[Vinoski\]](#).

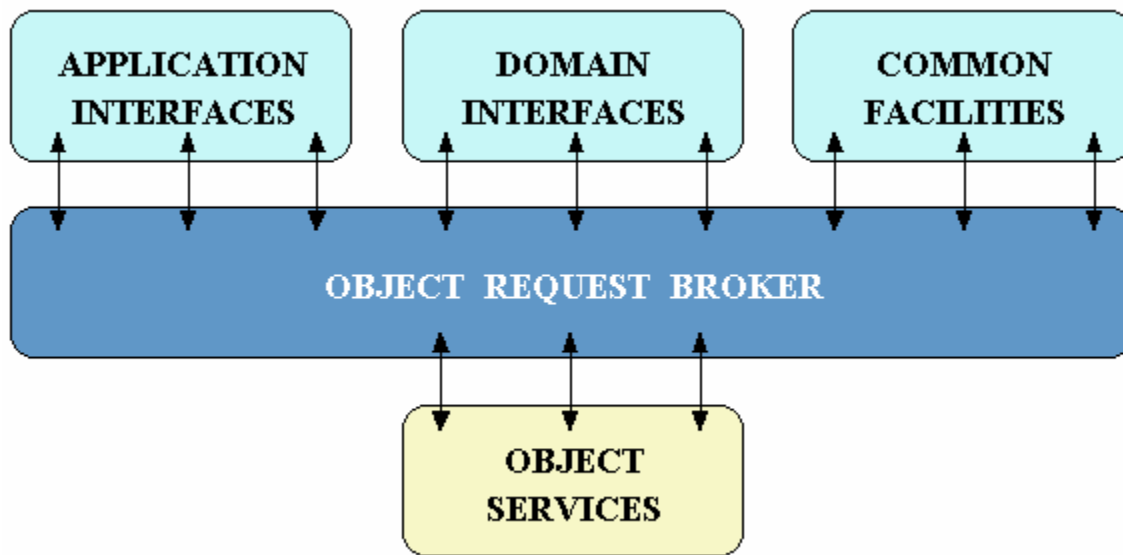


Figure 1 - OMG Reference Model Architecture

Object Services - These are domain-independent interfaces that are used by many distributed object programs. For example, a service providing for the discovery of other available services is almost always necessary regardless of the application domain. Two examples of Object Services that fulfill this role are:

The Naming Service - which allows clients to find objects based on names;

The Trading Service - which allows clients to find objects based on their properties.

There are also Object Service specifications for lifecycle management, security, transactions, and event notification, as well as many

Common Facilities -- Like Object Service interfaces, these interfaces are also horizontally-oriented, but unlike Object Services they are oriented towards end-user applications. An example of such a facility is the *Distributed Document Component Facility* (DDCF), a compound document Common Facility based on OpenDoc. DDCF allows for the presentation and interchange of objects based on a document model, for example, facilitating the linking of a spreadsheet object into a report document.

Domain Interfaces -- These interfaces fill roles similar to Object Services and Common Facilities but are oriented towards specific application domains. For example, one of the first OMG RFPs issued for Domain Interfaces is for Product Data Management (PDM) Enablers for the manufacturing domain. Other OMG RFPs will soon be issued in the telecommunications, medical, and financial domains.

DCE:

DCE DFS is a distributed client/server application built on the underlying DCE services. It takes full advantage of the lower-level DCE components (such as RPC, the security service, and the directory service). DFS Data Organization DFS data is organized at three levels. (See Figure) The three levels of DFS data are as follows, from smallest to largest:

Files and directories

The unit of user data. Directories organize files (and other directories) into a hierarchical tree structure.

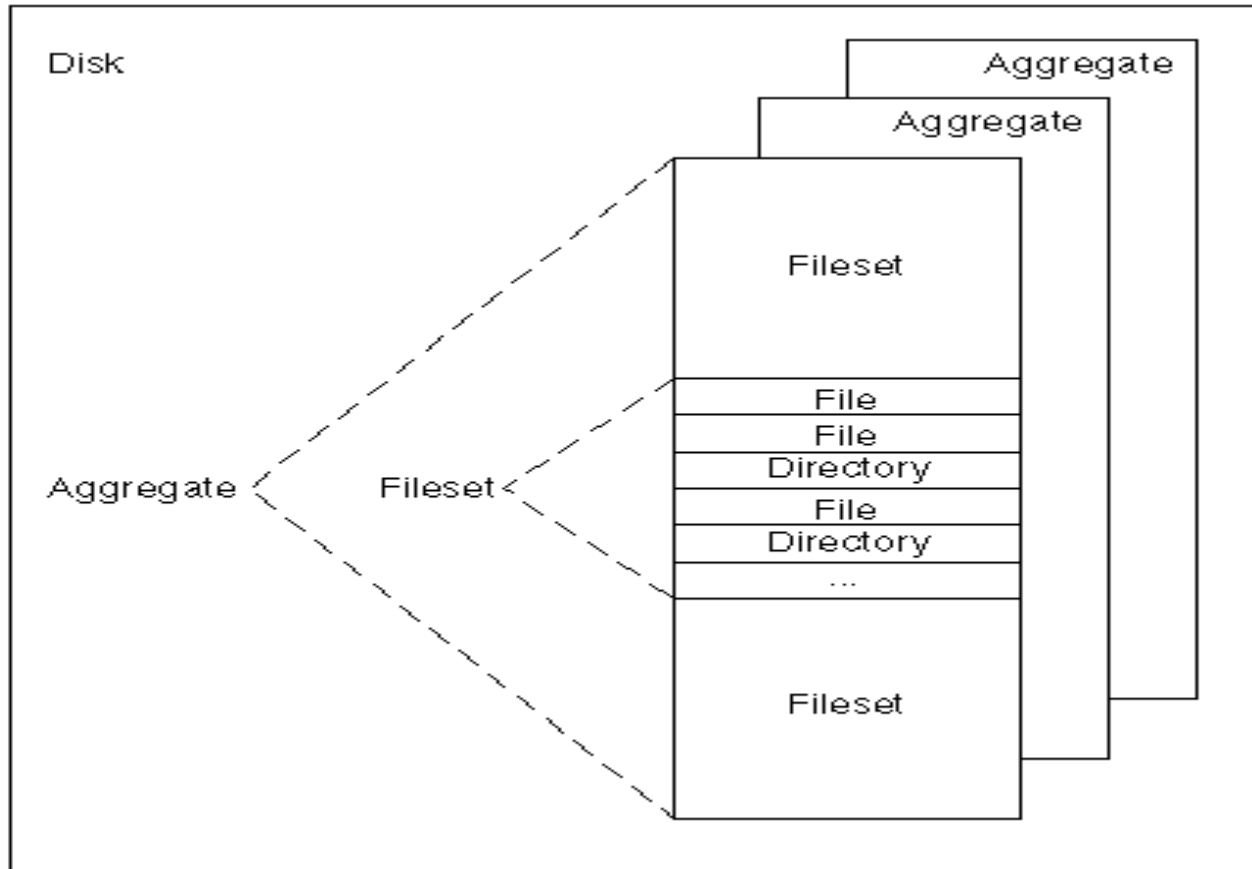
Filesets

The unit of administration. A fileset is a subtree of files and directories that is no larger than a disk or partition (or logical volume, if supported). The fileset is a convenient grouping of files for administrative purposes; for example, the subtree of files pertaining to a particular project can be grouped in the same fileset.

Aggregates

The unit of disk storage, similar to a disk partition. It is also the unit of fileset exporting, which makes the data in filesets available to users of DFS. It can contain one or more filesets.

Figure 34 - Files, Directories, Filesets, and Aggregates



Features of DCE DFS

DCE DFS has the following features:

1. Uniform file access:DFS is based on a global namespace. A DFS file is accessed by the same name no matter where in the distributed system it is accessed from. Users do not need to know the network address or name of the file server machine on which the file is located to name and access the file. For example, the file `/.../cs.univ.edu/fs/usr/ziggy/thesis` can be addressed by that name from anywhere in DCE, including from foreign cells.
2. Intracell location transparency:Data can move from one location to another within a cell without a user or programmer being affected by the move. Because of this transparency, an administrator can move a fileset from one file server machine to another for load balancing, for example, without disturbing users.
3. Performance:DFS is a high-performance file service. Fast response is achieved in part through the caching of file and directory data on the DFS client machine. This reduces the time it takes for a user to access a file, and it also reduces the traffic on the network and the load on the file server machine. The first time a user on a machine accesses a file, the cache manager gets a copy of the file from the file server machine and caches it on the client machine. Subsequent access to

the file can then be made to the copy on the client machine rather than to the copy on the file server machine.

4. **Availability:** DFS makes its services and data highly available in several ways. One way is through replication, in which a read-only copy of a file can be stored on more than one file server machine. This way, if the file server machine that houses one copy of the file is down, another copy of the file may still be available on another file server machine. DFS replication is especially useful for files that are accessed by many users but change infrequently (for example, binary files).

Another way DFS achieves high availability is through caching. Copies of files are cached on DFS clients. Even if a client is temporarily disconnected from the network, users of the client may be able to access copies of files that reside in the local cache.

DFS administration can occur while users continue to access DFS files, which is another means of providing high availability. Both backups and relocation of DFS filesets can be done without making the data in the filesets unavailable to users.

The physical file system portion of DFS, DCE LFS, is designed for fast recovery (yielding high availability) after failures. DCE LFS is a log-based file system; that is, DCE LFS keeps a record of actions taken that affect the file system structure so that, in the case of a system crash, the record can be replayed to bring the file system to a consistent state.

5. **Support for distributed application programming:**DFS is itself a distributed application, but it in turn supports the development of other distributed applications. Programmers can use DFS to share data or to communicate in a distributed application. DFS takes care of network communications and the movement, synchronization, and storage of shared data.
6. **Ease of administration and scalability:** DFS files are grouped into units called *filesets*, which are convenient to administer. The processes that implement DFS, such as the FL server and the backup server, are monitored and maintained automatically by the BOS server, resulting in less work and a more scalable system for a DFS administrator. Because of the high performance mentioned previously, DFS has a high client-to-server ratio. This leads to a scalable system in which clients can be added with low impact on other clients and the rest of the system. Finally, DFS includes tools such as the update server to automate time-consuming administrative tasks.
7. **Integration:**DFS is fully integrated with other DCE components, including RPC, the security service, the directory service, and threads.
8. **Interoperation:**DFS interoperates with other file systems; for example, a UFS can be exported to users of DFS.
9. **Standards:**DFS maintains POSIX single-site read/write semantics. DCE LFS adheres to POSIX 1003.1

DCE Distributed Time Service:

distributed computing system has many advantages but also brings with it new problems. One of them is keeping the clocks on different nodes synchronized. In a single system, there is one clock that provides the time of day to all applications. Computer hardware clocks are not completely accurate, but there is always one consistent idea of what time it is for all processes running on the system.

In a distributed system, however, each node has its own clock. Even if it were possible to set all of the clocks in the distributed system to one consistent time at some point, those clocks would drift away from that time at different rates. As a result, the different nodes of a distributed system have different ideas of what time it is. This is a problem, for example, for distributed applications that care about the ordering of events. It is difficult to determine whether Event A on Node X occurred before Event B on Node Y because different nodes have different notions of the current time.

DCE DTS addresses this problem in two ways:

1. DTS provides a way to periodically synchronize the clocks on the different hosts in a distributed system.
 2. DTS also provides a way of keeping that synchronized notion of time reasonably close to the *correct* time. (In DTS, correct time is considered to be UTC, an international standard.)
- These services together allow cooperating nodes to have the same notion of what time it is, and to also have that time be meaningful in the rest of the world.

Distributed time is inherently more complex than time originating from a single source since clocks cannot be continuously synchronizing, there is always some discrepancy in their ideas of the current time as they drift between synchronizations. In addition, indeterminacy is introduced in the communications necessary for synchronization since clocks synchronize by sending messages about the time back and forth, but that message passing itself takes a certain (unpredictable) amount of time. So in addition to being able to express the time of day, a distributed notion of time must also include an *inaccuracy* factor; that is, how close the timestamp is to the real time. As a result, keeping time in a distributed environment requires not only new synchronization mechanisms, but also a new form of expression of time--one that includes the inaccuracy of the given time. In DTS, distributed time is therefore expressed as a range, or interval, rather than as a single point.

There are several different components that constitute DCE DTS:

- Time clerk
- Time servers
- Local time server
- Global time server
- Courier time server
- Backup courier time server

- DTS API
- Time-Provider Interface (TPI)
- Time format, which includes inaccuracy

Time Clerk

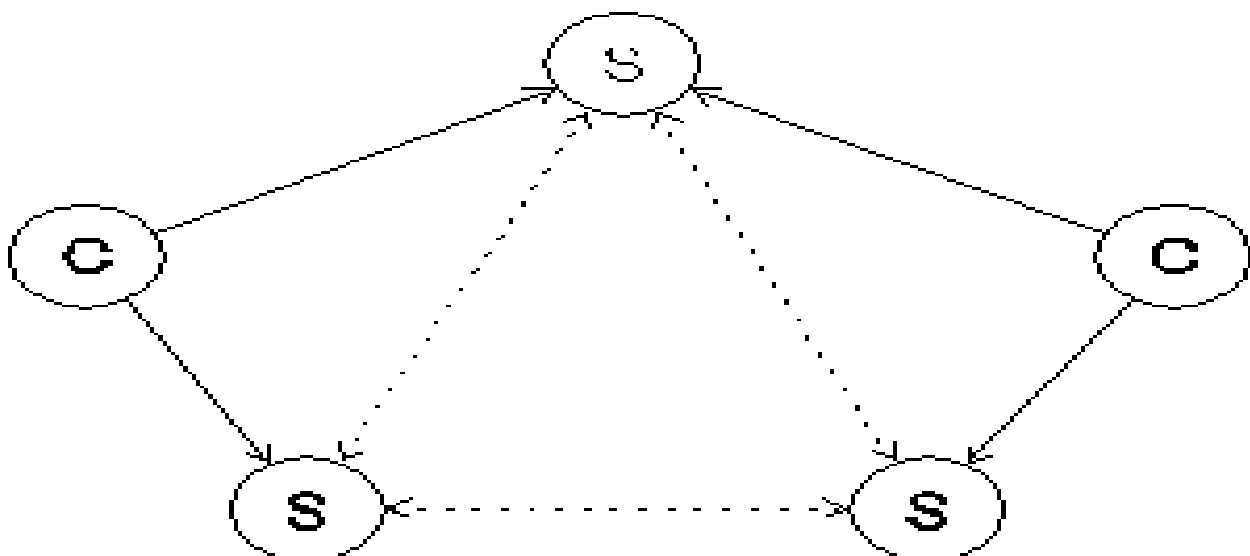
The time clerk is the client side of DTS. It runs on a client machine, such as a workstation, and keeps the machine's local time synchronized by asking time servers for the correct time and adjusting the local time accordingly.

The time clerk is configured to know the limit of the local system's hardware clock. When enough time has passed that the system's time is above a certain inaccuracy threshold (that is, the clock may have drifted far enough away from the correct time), the time clerk issues a synchronization. It queries various time servers for their opinion of the correct time of day, calculates the probable correct time and its inaccuracy based on the answers it receives, and updates the local system's time.

The update can be gradual or abrupt. If an abrupt update is made, the software register holding the current time is modified to reflect the new time. Usually, however, it is desirable to update the clock gradually and, in this case, the tick increment is modified until the correct time is reached. In other words, if a clock is normally incremented 10 milliseconds at each clock interrupt, and the clock is behind, then the clock register will instead be incremented 11 milliseconds at each clock tick until it catches up.

Figure shows a LAN with two time clerks (C) and three time servers (S). Each of the time clerks queries two of the time servers when synchronizing. The time servers all query each other.

Figure 30 - DTS Time Clerks and Servers



Time Servers

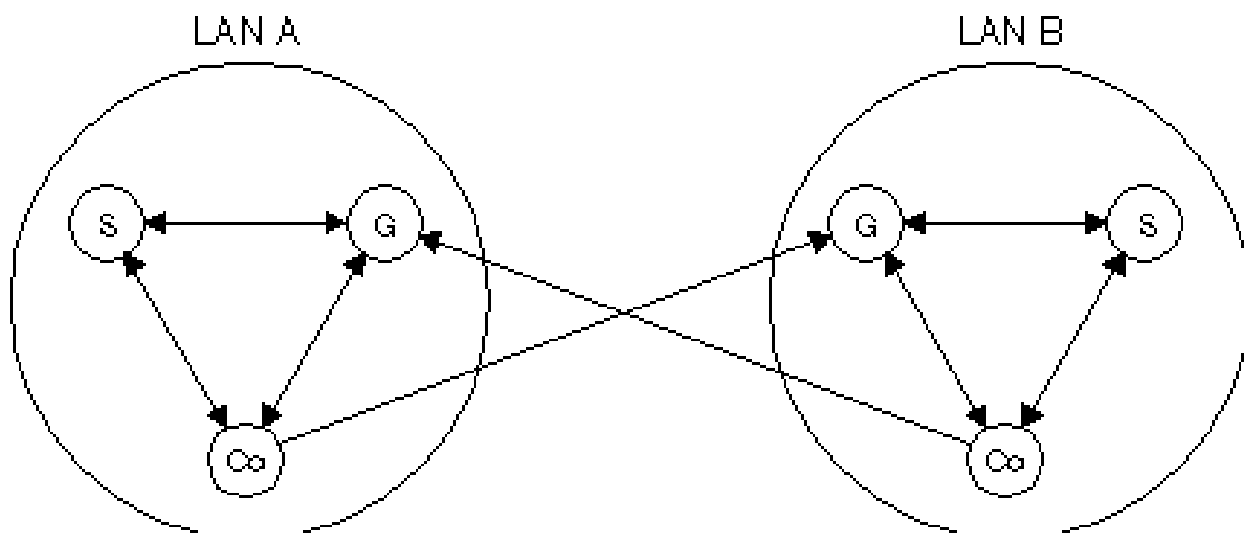
A time server is a node that is designated to answer queries about the time. The number of time servers in a DCE cell is configurable; three per LAN is a typical number. time clerks query these time servers for the time, and the time servers query one another, computing the new system time and adjusting their own clocks as appropriate. One or more of the time servers can be attached to an external time-provider (described later in this section).

A distinction is made between local time servers (time servers on a given LAN) and global time servers. This is because they are located differently by their clients. A client may need to contact a global time server if, for example, the client wants to get time from three servers, but only two servers are available on the LAN. In addition, it may be desirable to configure a DTS system to have two LAN servers and one global time server synchronizing with each other, rather than just having time servers within the LAN synchronizing with each other. This is where couriers are needed.

A courier time server is a time server that synchronizes with a global time server; that is, a time server outside the courier's LAN. It thus imports an outside time to the LAN by synchronizing with the outside time server. Other time servers in the LAN can be designated as backup courier time servers. If the courier is not available, then one of the backup couriers serves in its place.

Figure shows two LANs (LAN A and LAN B) and their time servers (S). In each LAN, one of the time servers acts as a courier time server (Co) by querying a global time server (G) for the current time.

Figure 31 - Local, Courier, and Global Time Servers



DTS Application Programming Interface

DTS provides an API library that allows programmers to manipulate timestamps. For example, programmers can obtain a timestamp representing the current time, translate timestamps to different formats, and compare two timestamps.

Time-Provider Interface

So far, all the components described are those supporting the synchronization of a distributed system's clocks. There must also be a way to ensure they are synchronized to the *correct* time. The notion of the correct time must come from an outside source, which is the external time-provider. This may be a hardware device such as one that receives time from radio or telephone sources. This external time is given to a time server, which then communicates it to other servers. Such an external time-provider can be very accurate. If no such device is available, the external time source can be the system administrator, who consults a trustworthy time source and enters it into the system. This cannot, of course, be as accurate as an automatic time source, but it may be sufficient in some cases.

DTS supports the ability to interface with an external time-provider through the time-provider interface. The external time-provider itself, however, is a hardware device (or a person), and is therefore outside the scope of DCE.

DTS Time Format

The time format used in DTS is a standard one: UTC, which notes the time since October 15, 1582, the beginning of the Gregorian calendar. This time is interpreted using the Time Differential Factor (TDF) for use in different time zones. For example, the TDF in New York City is -5 hours. The TDF for Greenwich, England is 0.

NTP:

The public domain software package called NTP (Network Time Protocol) is an implementation of the same named TCP/IP network protocol. NTP has been initiated in the 1980's by [Dave L. Mills](#) who was trying to achieve a high accuracy time synchronization for computers across the network. The protocol and related algorithms have been specified in several [RFCs](#). Since then NTP has continuously been optimized and is at present time widely used around the world. The protocol supports an accuracy of time down to nanoseconds. However, the maximum achievable accuracy also depends on the operating system and the network performance.

Currently there are two versions of NTP which can be used intermixed: NTP v3 is the latest released version which runs very stable on many operating systems. NTP v4 has some improvements over NTP v3 and has better support for some operating systems. Additionally, there's also a simplified version of the protocol called SNTP (Simple Network Time Protocol). SNTP uses the same TCP/IP packet structure like NTP but due to the simpler algorithms, it provides only very reduced precision. The NTP package contains a background program

(*daemon* or *service*) which synchronizes the computer's system time to one or more external reference time sources which can be either other devices on the network, or a radio clock which is connected to the computer.

Each NTP source distribution contains the NTP daemon itself, plus some utility programs. Earlier versions of the NTP distribution and some of the programs included in the package had names starting with *xntp* (e.g. xntpd) while other utilities in the same package had names starting with *ntp* (e.g. ntpq).

Beginning with NTP version 4, the naming conventions were changed to be more straightforward, so now the name of the NTP distribution itself and the names of all the programs included start with *ntp* (e.g. ntpd, ntpq).

Some Unix-like operating systems use a **script to start the NTP daemon** at system start-up. Sometimes the script still has a name starting with xntp even though the real name of the daemon is ntpd. This is the case, for example, for SuSE Linux.

The Time Synchronization Hierarchy:

The NTP daemon can not only adjust its own computer's system time. Additionally, each daemon can be a client, server, or peer for other NTP daemons:

- As **client** it queries the reference time from one or more servers.
- As **server** it makes its own time available as reference time for other clients.
- As **peer** it compares its system time to other peers until all the peers finally agree about the "true" time to synchronize to.

These features can be used to set up a hierarchical time synchronization structure. The hierarchical levels of the time synchronization structure are called *stratum* levels. A smaller *stratum* number means a higher level in the hierarchy structure. On top of the hierarchy there is the daemon which has the most accurate time and therefore the smallest *stratum* number.

By default, a daemon's *stratum* level is always one level below the level of its reference time source. The top level daemon often uses a radio clock as reference time source. By default, radio clocks have a *stratum* number of 0, so a daemon who uses that radio clock as reference time will be a ***stratum 1 time server***, which has the highest priority level in the NTP hierarchy. In large networks it is a good practice to install one or more stratum 1 time servers which make a reference time available to several server computers in each department. Thus the servers in the departments become stratum 2 time servers which can be used as reference time source for workstations and other network devices of the department.

Unlike in telecom applications where the word *stratum* is used e.g. to classify oscillators according to their absolute accuracy, the term *stratum* in the NTP context does not indicate a certain class of accuracy, it's just an indicator of the hierarchy level.

BEYOND SYLLABUS – 02

AIM: Study of Hadoop Linux.

Apache Hadoop is an open-source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures (of individual machines, or racks of machines) are commonplace and thus should be automatically handled in software by the framework.

The core of Apache Hadoop consists of a storage part (Hadoop Distributed File System (HDFS)) and a processing part (MapReduce). Hadoop splits files into large blocks and distributes them amongst the nodes in the cluster. To process the data, Hadoop MapReduce transfers packaged code for nodes to process in parallel, based on the data each node needs to process. This approach takes advantage of data locality—nodes manipulating the data that they have on hand—to allow the data to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are connected via high-speed networking.

The base Apache Hadoop framework is composed of the following modules:

- Hadoop Common – contains libraries and utilities needed by other Hadoop modules;
- Hadoop Distributed File System (HDFS) – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- Hadoop YARN – a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications; and
- Hadoop MapReduce – a programming model for large scale data processing.

The term "Hadoop" has come to refer not just to the base modules above, but also to the "ecosystem", or collection of additional software packages that can be installed on top of or alongside Hadoop, such as Apache Pig, Apache Hive, Apache HBase, Apache Spark, and others. Apache Hadoop's MapReduce and HDFS components were inspired by Google papers on their MapReduce and Google File System.

The Hadoop framework itself is mostly written in the Java programming language, with some native code in C and command line utilities written as Shell script. For end-users, though MapReduce Java code is common, any programming language can be used with "Hadoop Streaming" to implement the "map" and "reduce" parts of the user's program. Other related projects expose other higher-level user interfaces.

Prominent corporate users of Hadoop include Facebook and Yahoo. It can be deployed in traditional on-site datacenters but has also been implemented in public cloud spaces such as Microsoft Azure, Amazon Web Services, Google App Engine and IBM Bluemix.

Apache Hadoop is a registered trademark of the Apache Software Foundation.

Hadoop distributed file system

The Hadoop distributed file system (HDFS) is a distributed, scalable, and portable file-system written in Java for the Hadoop framework. A Hadoop cluster has nominally a single namenode plus a cluster of datanodes, although redundancy options are available for the namenode due to its criticality. Each datanode serves up blocks of data over the network using a block protocol specific to HDFS. The file system uses TCP/IP sockets for communication. Clients use remote procedure call (RPC) to communicate between each other.

HDFS stores large files (typically in the range of gigabytes to terabytes) across multiple machines. It achieves reliability by replicating the data across multiple hosts, and hence theoretically does not require RAID storage on hosts (but to increase I/O performance some RAID configurations are still useful). With the default replication value, 3, data is stored on three nodes: two on the same rack, and one on a different rack.

Data nodes can talk to each other to rebalance data, to move copies around, and to keep the replication of data high. HDFS is not fully POSIX-compliant, because the requirements for a POSIX file-system differ from the target goals for a Hadoop application. The trade-off of not having a fully POSIX-compliant file-system is increased performance for data throughput and support for non-POSIX operations such as Append.

HDFS added the high-availability capabilities, as announced for release 2.0 in May 2012, letting the main metadata server (the NameNode) fail over manually to a backup. The project has also started developing automatic fail-over.

The HDFS file system includes a so-called secondary namenode, a misleading name that some might incorrectly interpret as a backup namenode for when the primary namenode goes offline. In fact, the secondary namenode regularly connects with the primary namenode and builds snapshots of the primary namenode's directory information, which the system then saves to local or remote directories. These checkpointed images can be used to restart a failed primary namenode without having to replay the entire journal of file-system actions, then to edit the log to create an up-to-date directory structure.

Because the namenode is the single point for storage and management of metadata, it can become a bottleneck for supporting a huge number of files, especially a large number of small files. HDFS Federation, a new addition, aims to tackle this problem to a certain extent by allowing multiple namespaces served by separate namenodes.

An advantage of using HDFS is data awareness between the job tracker and task tracker. The job tracker schedules map or reduce jobs to task trackers with an awareness of the data location. For example: if node A contains data (x,y,z) and node B contains data (a,b,c), the job tracker schedules node B to perform map or reduce tasks on (a,b,c) and node A would be scheduled to perform map or reduce tasks on (x,y,z).

This reduces the amount of traffic that goes over the network and prevents unnecessary data transfer. When Hadoop is used with other file systems, this advantage is not always available. This can have a significant impact on job-completion times, which has been demonstrated when running data-intensive jobs.

HDFS was designed for mostly immutable files and may not be suitable for systems requiring concurrent write-operations.

HDFS can be mounted directly with a Filesystem in Userspace (FUSE) virtual file system on Linux and some other Unix systems.

File access can be achieved through the native Java API, the Thrift API to generate a client in the language of the users' choosing (C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk, and OCaml), the command-line interface, browsed through the HDFS-UI webapp over HTTP, or via 3rd-party network client libraries.