PUNE INSTITUTE OF COMPUTER TECHNOLOGY
DHANKAWADI, PUNE – 43.

LAB MANUAL

ACADEMIC YEAR: 2012-2013

DEPARTMENT: COMPUTER ENGINEERING

CLASS: B.E **SEMESTER:** II

SUBJECT: COMPUTER LABORATORY-II

INDEX OF LAB EXPERIMENTS

EXPT. NO	Problem statement to be defined using following topic	Revised on
1	Installation, configuration and Administration of HADOOP, Develop front-end for the same. Develop NameNode, TrackerNode, DataNodes (3 to 4 Computers) in loosely coupled Clusters. a. Run a sample job. (word count, terasort,) b. Miniproject using HADOOP for large data set and monitor i. Replication factor ii. Fault recovery iii.Election Algorithm iv.Task Assignment V. Load balancing	15/12/2012
2	Write a multi-threaded Echo server application which will use Socket/RPC communication mechanism.	15/12/2012
3	Write a program to create socket to communicate between two process and remote procedure should read all the ports of the remote machine.	15/12/2012
4	Use the RMI / RPC concept to define the operations on one machine and use those on remotely placed machine's application.	15/12/2012
5	Use IPC Mechanism: shared memory, named pipes and socket to show how distributed shared memory works	15/12/2012
6	Write a program for application running over the HTTP protocol and read the HTTP header of this application	15/12/2012

7	Write a program for finding and using the idle work station in the network with the help of registry based algorithm/ Configuration files.	15/12/2012
8	Write a program to simulate election algorithm to count votes; in a loosely coupled distributed system few machines will act as a voting machines connected to server. For the synchronization using following method a. Ring Algorithm b.Bully's Algorithm	15/12/2012
9	Write program for synchronization of logical clock using Lamport's algorithm.	15/12/2012
10	Write a program for web service creation and consumption of this web services.	15/12/2012

Head of Department
(Computer Engineering)

Subject Coordinator
(Amar Buchade)

TITLE	Multi-threaded Echo server
TOPIC for PROBLEM STATEMENT /DEFINITION	Write a multi-threaded Echo server application which will use communication mechanism. a) Socket b) RPC
OBJECTIVE	Understand system calls for client server communication using socket programming. Understand thread concept Study of RPC mechanism.
S/W PACKAGES AND HARDWARE APPARATUS USED	Ubuntu 10.04,Fedora Portmap package for ubuntu
REFERENCES	1. Unix Network Programming by W. Richard Stevens. Published by Prentice Hall. 2. Beej's Guide to Network Programming **http:/beej.us/guide/bgnet/** 3. Remote procedure call,http://www.linuxjournal.com/article/2204 4. Multithreading," http://www4.comp.polyu.edu.hk/~csajaykr/myhome/teaching/eel358/MT.pdf" 5. How to create thread," http://www.thegeekstuff.com/2012/04/create-threads-in-linux/" 6.Multithreading, "http://users.encs.concordia.ca/~mia/tutorials/coen346/IPC_threads/ch_threads.html"
STEPS	Refer to theory
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Objectives • Theory • Class Diagram/UML diagram • Test cases • Program Listing • Output • Conclusion

Title: Socket programming and RPC

TOPIC FOR PROBLEM STATEMENT:

Objective: To understand system calls for client server communication using socket programming.

Aim: Implementing multithreaded echo server program.

The working of the program can be as below.

1. Multiple clients at a time read string from its standard input and write the line to the server.
2. The server read a line from its network input and echoes the line back to the clients.
3. The clients read the echoed line and print it on its standard output.

Theory:

The standard model for network applications is the client-server model. A server is process that is waiting to be contacted by a client process sot that the server can do something for the client. A typical scenario is as follows:

1. The server process is started on some computer system. It initializes itself, and then goes to sleep waiting for a client process to contact it requesting some service.
2. A client process is started, either on the same system or on another system that is connected to the server's system with a network. Client processes are often initiated by an interactive user entering a command to a time-sharing system. The client process sends a request across the network to the server requesting service of some form. Some examples of the type of server that a server can provide are

   e.g.

   1. Return the time of day to the client.
   2. Print a file on printer for the client
   3. Read or write a file on the server's system for the client
   4. Allow the client to login to server's system.
   5. Execute a command for the client on the server's system.

6. When the server process has finished providing its service to the client, server goes back to sleep, waiting for the next client request to arrive.

Sockets is a method for communication between a client program and a server program in a network, A socket is defined as "the endpoint in a connection." Sockets are created and used with a set of programming requests or "function calls" sometimes called the sockets application-programming interface (API). The most common sockets API is the Berkeley Unix C interface for sockets. Sockets can also be used for communication between processes within the same computer.

System calls allow you to access the network functionality of a Unix. When you call one of these functions, the kernel takes over and does all the work for you automatically.

- **Socket ()** System call –

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

The family is one of

```
AF_UNIX         Unix internal protocols
AF_INET         Internet protocols
AF_NS                   Xerox NS protocols
AF_IMPLINK      IMP link layer
```

The AF_ prefix stands for "address family".

The socket type is one of the following.

```
SOCK_STREAM     stream socket
SOCK_DGRAM      datagram socket
```

The protocol argument to the socket system call is typically set to 0 for most user applications.

2.**bind** System call

bind system call assigns a name to an unnamed socket.

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *myaddr,int addrlen);
```

The second argument is a pointer to a protocol-specific address and the third argument is the size of this address structure .

3. **connect** System call

A client process connects a socket descriptor following the socket system call to establish a connection with a server.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, struct sockaddr *servaddr,int addrlen);
```

The sockfd is a socket descriptor that returned by the socket system call.The second and third arguments are a pointer to a socket address and its size.

4.**listen** System call

This system call is used by a connection-oriented server to indicate that is willing to receive connections.

```
int  listen(int sockfd, int backlog);
```

The backlog argument specifies how many connection requests can be queued by the system while it waits for the server to execute the accept system call. This argument is usually specified as 5, the maximum value currently allowed.

5.**accept** System call

After a connection-oriented server executes the listen system call, an actual connection from some client process is waited for by having the server execute the accept system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *peer,int *addrlen);
```

accept takes the first connection request on the queue and creates another socket with the same properties as sockfd. If there are no connection request pending, this call blocks the caller until on arrives.

The peer and addrlen arguments are used to return the address of the connected peer process (the client).

The system call returns up to three values : an integer return code that is either an error indication or a new socket descriptor, the address of the client process(peer), and the size of this address(addrlen).

6.**read** System call

Data is read from buffer

```
int read(int sockfd,char *buff,unsigned int nbytes);
```

If the read is successful , the number of bytes read is returned- this can be less than the nbytes that was requested. If the end of buffer is encountered, zero is returned. If an error is encountered, -1 is returned.

7.**write** System call

Data is written to buffeR

```
int write(int sockfd, char *buff,unsigned int nbytes);
```

The actual number of bytes written is returned by the system call. This is usually equal to nbytes argument. If an error occurs , -1 returned.

8.**send ,sendto, recv and recvfrom** system calls:

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int sockfd, char *buff, int nbytes,int flags);
int sendto(int sockfd, char *buff, int nbytes,int flags,struct
sockaddr *to,int addrlen);
int recv(int sockfd, char *buff, int nbytes,int flags);
int sendto(int sockfd, char *buff, int nbytes,int flags,struct
sockaddr *from,int addrlen);
```

The first three argument,sockfd,buff, and nbytes , to the four system call are similar to the first three arguments for read and write.

The flags argument is zero.

The to argument for sendto specifies the protocol-sepcific address of where the data is to be sent. Since this address is protocol specific, its length must be specified by addrlen.

The recvfrom system call fills in the protocol-specific address of who sent the data into from.The length of this address is also returned to the caller in addrlen.

Final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value.

All four system calls return the length of the data that was written or read as the value of the function.

9.**close** system call

The normal Unix close system call is also used to close a socket.

int close(int fd);

References:

1.Unix Network Programming by W. Richard Stevens. Published by Prentice Hall.

2.Beej's Guide to Network Programming **http:/beej.us/guide/bgnet/**

ii) Implementation of Multithreaded Echo server using RPC mechanism with RPCGEN **utility.**

Objective: Study of RPC mechanism.

Theory:

RPCGEN is an interface generator pre-compiler for Sun Microsystems RPC. It uses an interface definition file to create client and server stubs in C.

RPCGEN creates stubs based on information contained within an IDL file. This file is written in a language called RPCL - remote procedure call language. This language closely mimics C in style.

An RPC specification contains a number of definitions. These definitions are used by RPCGEN to create a header file for use by both the client and server, and client and server stubs.

RCPL Definitions: Constant, Enumeration, Struct, Union, and TypeDef, Program.

Remote Procedure Call (RPC) is a **protocol** that one program can use to request a service from a program located in another computer in a network without having to understand network details. (A *procedure call* is also sometimes known as a *function call* or a *subroutine call*.) RPC uses the **client/server** model. The requesting program is a client and the service-providing program is the server. Like a regular or local procedure call, an RPC is a **synchronous** operation requiring the requesting program to be suspended until the results of the remote procedure are returned.

When program statements that use RPC are **compiled** into an executable program, a **stub** is included in the compiled code that acts as the representative of the remote procedure code. When the program is run and the procedure call is issued, the stub receives the request and forwards it to a client **runtime** program in the local computer. The client runtime program has the knowledge of how to address the remote computer and server application and sends the message across the network that requests the remote procedure. Similarly, the server includes a runtime program and stub that interface with the remote procedure itself. Results are returned the same way.

Remote procedure calls (RPC) extend the capabilities of conventional procedure calls across a network and are essential in the development of distributed systems.

They can be used both for data exchange in distributed file and database systems and for harnessing the power of multiple processors.

Linux distributions provide an RPC version derived from the RPC facility developed by the Open Network Computing (ONC) group at Sun Microsystems.

RPC and the Client/Server Model:

1. **Caller**: a program which calls a subroutine
2. **Callee**: a subroutine or procedure which is called by the caller
3. **Client**: a program which requests a connection to and service from a network server
4. **Server**: a program which accepts connections from and provides services to a client

There is a direct parallel between the caller/callee relationship and the client/server relationship. With ONC RPC (and with every other form of RPC that I know), the caller always executes as a client process, and the callee always executes as a server process.

The Remote Procedure Call Mechanism

In order for an RPC to execute successfully, several steps must take place:

1. The caller program must prepare any input parameters to be passed to the RPC. Note that the caller and the callee may be running completely different hardware, and that certain data types may be represented differently from one machine architecture to the next. Because of this, the caller cannot simply feed *raw* data to the remote procedure.
2. The calling program must somehow pass its data to the remote host which will execute the RPC. In local procedure calls, the target address is simply a machine address on the local processor. With RPC, the target procedure has a machine address combined with a network address.
3. The RPC receives and operates on any input parameters and passes the result back to the caller.
4. The calling program receives the RPC result and continues execution.

External Data Representation

As was pointed out earlier, an RPC can be executed between two hosts that run completely different processor hardware. Data types, such as integer and floating-point numbers, can have different physical representations on different machines. For example, some machines store integers (C ints) with the low order byte first while some machines place the low order byte last. Similar problems occur with floating-point numeric data. The solution to this problem involves the adoption of a standard for data interchange.

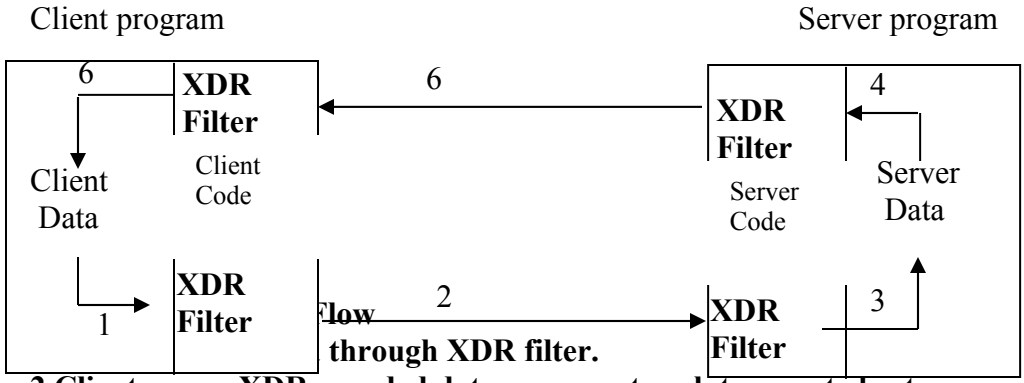
One such standard is the ONC external data representation (XDR). XDR is essentially a collection of C functions and macros that enable conversion from machine specific data representations to the corresponding standard representations and vice versa.

It contains primitives for simple data types such as int, float and string and provides the capability to define and transport more complex ones such as records, arrays of arbitrary element type and pointer bound structures such as linked lists.

Most of the XDR functions require the passing of a pointer to a structure of ``XDR'' type. One of the elements of this structure is an enumerated field called **x_op**. It's possible values are **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**. The **XDR_ENCODE** operation instructs the XDR routine to convert the passed data to XDR format. The **XDR_DECODE** operation indicates the conversion of XDR represented data back to its local representation. **XDR_FREE** provides a means to deallocate memory that was dynamically allocated for use by a variable that is no longer needed.

RPC Data Flow

The flow of data from caller to callee and back again is illustrated in Figure 1. The calling program executes as a client process and the RPC runs on a remote server. All data movement between the client and the network and between the server and the network pass through XDR filter routines. In principle, any type of network transport can be used, but our discussion of implementation specifics centers on ONC RPC which typically uses either Transmission Control Protocol routed by Internet Protocol (the familiar TCP/IP) or User Datagram Protocol also routed by Internet Protocol (the possibly not so familiar UDP/IP). Similarly, any type of data representation could be used, but our discussion focuses on XDR since it is the method used by ONC RPC.



through XDR filter.

2.Client passes XDR encoded data across network to remote host.
3.Server decodes data through XDR Filter.
4.Server encodes function call result through XDR Filter.
5.Server pass XDR encoded data across network back to client.
6.Client decodes RPC result through XDR Filter and continues processing.

Review of Network Programming Theory

In order to complete our picture of RPC processing, we'll need to review some network programming theory. In order for two processes running on separate computers to exchange data, an **association** needs to be formed on each host. An association is defined as the following 5-tuple: *{protocol, local-address, local-process, foreign-address, foreign-process}*

The *protocol* is the transport mechanism (typically TCP or UDP) which is used to move the data between hosts. This, of course, is the part that needs to be common to both host computers. For either host computer, the *local-address/process* pair defines the endpoint on the host computer running that process. The *foreign-address/process* pair refers to the endpoint at the opposite end of the connection.

Breaking this down further, the term *address* refers to the network address assigned to the host. This would typically be an Internet Protocol (IP) address. The term *process* refers not to an actual process identifier (such as a Unix PID) but to some integer identifier required to transport the data to the correct process once it has arrived at the correct host computer. This is generally referred to as a **port**. The reason port numbers are used is that it is not practical for a process running on a remote host to know the PID of a particular server. Standard port numbers are assigned to well known services such as TELNET (port 23) and FTP (port 21).

RPC Call Binding

Now we have the necessary theory to complete our picture of the RPC binding process. An RPC application is formally packaged into a *program* with one or more *procedure* calls. In a manner similar to the port assignments described above, the RPC program is assigned an integer identifier known to the programs which will call its procedures. Each procedure is also assigned a number that is also known by its caller. ONC RPC uses a program called **portmap** to allocate port numbers for RPC programs. It's operation is illustrated in Figure 2. When an RPC **program** is started, it registers itself with the portmap process running on the same host. The portmap process then assigns the TCP and/or UDP port numbers to be used by that application.

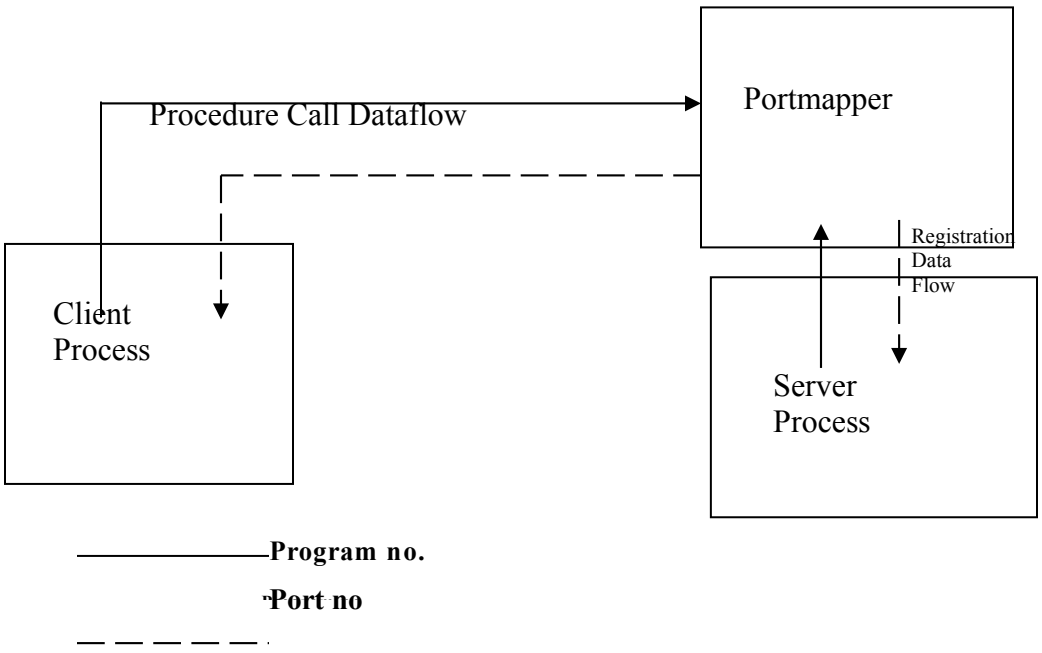


Figure 2. Portmap Operation

The RPC application then waits for and accepts connections at that port number. Prior to calling the remote procedure, the *caller* also contacts portmap in order to obtain the corresponding port number being used by the application whose procedures it needs to call. The network connection provides the means for the caller to reach the correct program on the remote host. The correct procedure is reached through the use of a dispatch table in the RPC program. The same registration process that establishes the port number also creates the dispatch table. The dispatch table is indexed by procedure number and contains the addresses of all the XDR filter routines as well as the addresses of the actual procedures.

MTRPC is a standard linux librpc.a SUN RPC modified to implement a multi threaded execution environment for remote procedure calls. It will make your rpc servers become multi threaded by just compiling them with -lmtrpc) if you use TCP transport).

References: Remote procedure call,http://www.linuxjournal.com/article/2204

2.http://h30097.www3.hp.com

TITLE	Inter process communication
TOPIC for PROBLEM STATEMENT /DEFINITION	Write a program to create socket to communicate between two processes and remote procedure should read all the ports of the remote machine.
OBJECTIVE	Understand system calls for client server communication using socket programming.
S/W PACKAGES AND HARDWARE APPARATUS USED	Ubuntu 10.04,Fedora
REFERENCES	1. Unix Network Programming by W. Richard Stevens. Published by Prentice Hall. 2. Beej's Guide to Network Programming **http:/beej.us/guide/bgnet/**
STEPS	Refer to theory
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Objectives • Theory • Class Diagram/UML diagram • Test cases • Program Listing • Output • Conclusion

Title: Socket programming and RPC
TOPIC FOR PROBLEM STATEMENT:
Objective : To understand how communication takes place between processes using socket programming.
Aim: 1. If there are three machines considered. Process 1 is at Machine 1, Process 2 is at Machine 2. So process 1 will call the procedure whose body is present in process 2 which in turn read the ports of machine 3 and returns it to the Process 1 machine.

Theory:

The standard model for network applications is the client-server model. A server is process that is waiting to be contacted by a client process sot that the server can do something for the client. A typical scenario is as follows:
The server process is started on some computer system. It initializes itself, and then goes to sleep waiting for a client process to contact it requesting some service.
A client process is started, either on the same system or on another system that is connected to the server's system with a network. Client processes are often initiated by an interactive user entering a command to a time-sharing system. The client process sends a request across the network to the server requesting service of some form. Some examples of the type of server that a server can provide are

e.g.

Return the time of day to the client.
Print a file on printer for the client
Read or write a file on the server's system for the client
Allow the client to login to server's system.
Execute a command for the client on the server's system.

When the server process has finished providing its service to the client, server goes back to sleep, waiting for the next client request to arrive.

Sockets is a method for communication between a client program and a server program in a network, A socket is defined as "the endpoint in a connection." Sockets are created and used with a set of programming requests or "function calls" sometimes called the sockets application-programming interface (API). The most common sockets API is the Berkeley Unix C interface for sockets. Sockets can also be used for communication between processes within the same computer.
System calls allow you to access the network functionality of a Unix. When you call one of these functions, the kernel takes over and does all the work for you automatically.

- **Socket ()** System call –

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

The family is one of

AF_UNIX	Unix internal protocols
AF_INET	Internet protocols

AF_NS Xerox NS protocols
AF_IMPLINK IMP link layer

The AF_ prefix stands for “address family”.

The socket type is one of the following.

SOCK_STREAM stream socket
SOCK_DGRAM datagram socket

The protocol argument to the socket system call is typically set to 0 for most user applications.

2.**bind** System call

bind system call assigns a name to an unnamed socket.

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *myaddr,int addrlen);
```

The second argument is a pointer to a protocol-specific address and the third argument is the size of this address structure .

3. **connect** System call

A client process connects a socket descriptor following the socket system call to establish a connection with a server.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, struct sockaddr *servaddr,int addrlen);
```

The sockfd is a socket descriptor that returned by the socket system call.The second and third arguments are a pointer to a socket address and its size.

4.**listen** System call

This system call is used by a connection-oriented server to indicate that is willing to receive connections.

```
int  listen(int sockfd, int backlog);
```

The backlog argument specifies how many connection requests can be queued by the system while it waits for the server to execute the accept system call. This argument is usually specified as 5, the maximum value currently allowed.

5.**accept** System call

After a connection-oriented server executes the listen system call, an actual connection from some client process is waited for by having the server execute the accept system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *peer,int *addrlen);
```

accept takes the first connection request on the queue and creates another socket with the same properties as sockfd. If there are no connection request pending, this call blocks the caller until on arrives.

The peer and addrlen arguments are used to return the address of the connected peer process (the client).

The system call returns up to three values : an integer return code that is either an error indication or a new socket descriptor, the address of the client process(peer), and the size of this address(addrlen).

6.**read** System call

Data is read from buffer

int read(int sockfd,char *buff,unsigned int nbytes);

If the read is successful , the number of bytes read is returned- this can be less than the nbytes that was requested. If the end of buffer is encountered, zero is returned. If an error is encountered, -1 is returned.

7.**write** System call

Data is written to buffeR

int write(int sockfd, char *buff,unsigned int nbytes);

The actual number of bytes written is returned by the system call. This is usually equal to nbytes argument. If an error occurs , -1 returned.

8.**send ,sendto, recv and recvfrom** system calls:

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int sockfd, char *buff, int nbytes,int flags);
int sendto(int sockfd, char *buff, int nbytes,int flags,struct
sockaddr *to,int addrlen);
int recv(int sockfd, char *buff, int nbytes,int flags);
int sendto(int sockfd, char *buff, int nbytes,int flags,struct
sockaddr *from,int addrlen);
```

The first three argument,sockfd,buff, and nbytes , to the four system call are similar to the first three arguments for read and write.

The flags argument is zero.

The to argument for sendto specifies the protocol-sepcific address of where the data is to be sent. Since this address is protocol specific, its length must be specified by addrlen.

The recvfrom system call fills in the protocol-specific address of who sent the data into from.The length of this address is also returned to the caller in addrlen.

Final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value.

All four system calls return the length of the data that was written or read as the value of the function.

9.**close** system call

The normal Unix close system call is also used to close a socket.

int close(int fd);

References:

1.Unix Network Programming by W. Richard Stevens. Published by Prentice Hall.

2.Beej's Guide to Network Programming **http:/beej.us/guide/bgnet/**

TITLE	Inter process communication
TOPIC for PROBLEM STATEMENT /DEFINITION	Use the RMI / RPC concept to define the operations on one machine and use those on remotely placed machine's application.
OBJECTIVE	Study RMI/RPC Mechanism
S/W PACKAGES AND HARDWARE APPARATUS USED	Ubuntu 10.04,Fedora
REFERENCES	1. http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html 2.The Complete Reference JAVA 2 –Herbert Schildt
STEPS	Refer to student theory
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Objectives • Theory • Class Diagram/UML diagram • Test cases • Program Listing • Output • Conclusion

Objective: Study of RMI/RPC mechanism.

RMI Mechanism: Java Remote Method Invocation

Theory: Remote Method Invocation, abbreviated as RMI , provides support for

distributed objects in Java, i.e. it allows objects to invoke methods on remote objects. The calling objects can use the exact same syntax as for local invocations
.

An RMI application is often composed of two separate programs, a server and a client . The server creates remotes objects and makes references to those objects

accessible.
Then it waits for clients to invoke methods on the objects. The client gets remote references to remote objects in the server and invokes methods on those remote objects.

The RMI model provides an distributed object application to the programmer. It is a mechanism that the server and the client use to communicate and pass information between each other. A distributed object application has to handle the following properties:

1. Locate remote objects: The system has to obtain references to remote objects. This can be done i two ways. Either by using RMI's naming facility, the rmiregistry, or by passing and returning remote objects.

2. Communicate with remote objects: The programmer doesn't have to handle

communication between the remote objects since this is handled by the RMI system.

The remote communication looks like an ordinary method invocation for the

programmer.

3. Load class bytecodes for objects that are passed as parameters or return values:

All mechanisms for loading an object's code and transmitting data is provided by the

RMI system.

Figure, below, illustrates an RMI distributed application. In this example the RMI

registry is used to obtain references to a remote object. First the server associates a name with
a remote object in the RMI registry. When a client wants access to a
remote object it looks up the object, by its name, in the registry.

Then the client can invoke methods on the remote object at the
server.

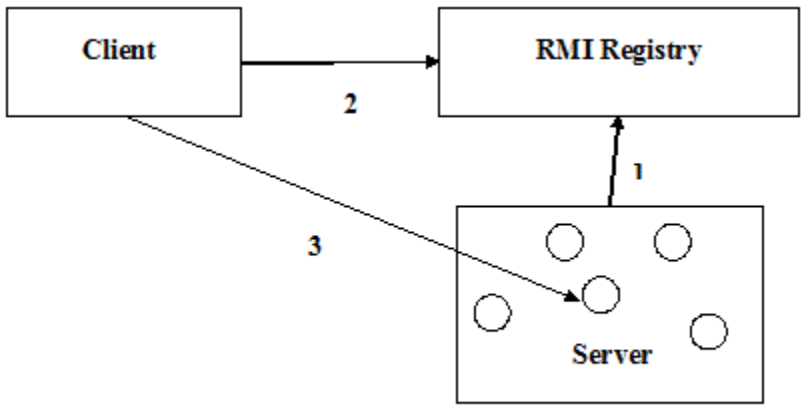


Figure RMI distributed application

Interfaces and classes:
Since Java RMI is a single-language system, the programming of distributed application in
RMI is rather simple. All interfaces and classes for the RMI system are
defined in the java.rmi package . Figure below, illustrates the relationship
between some of the classes and interfaces. The RemoteObject class implements the Remote
interface while the other classes extend RemoteObject.

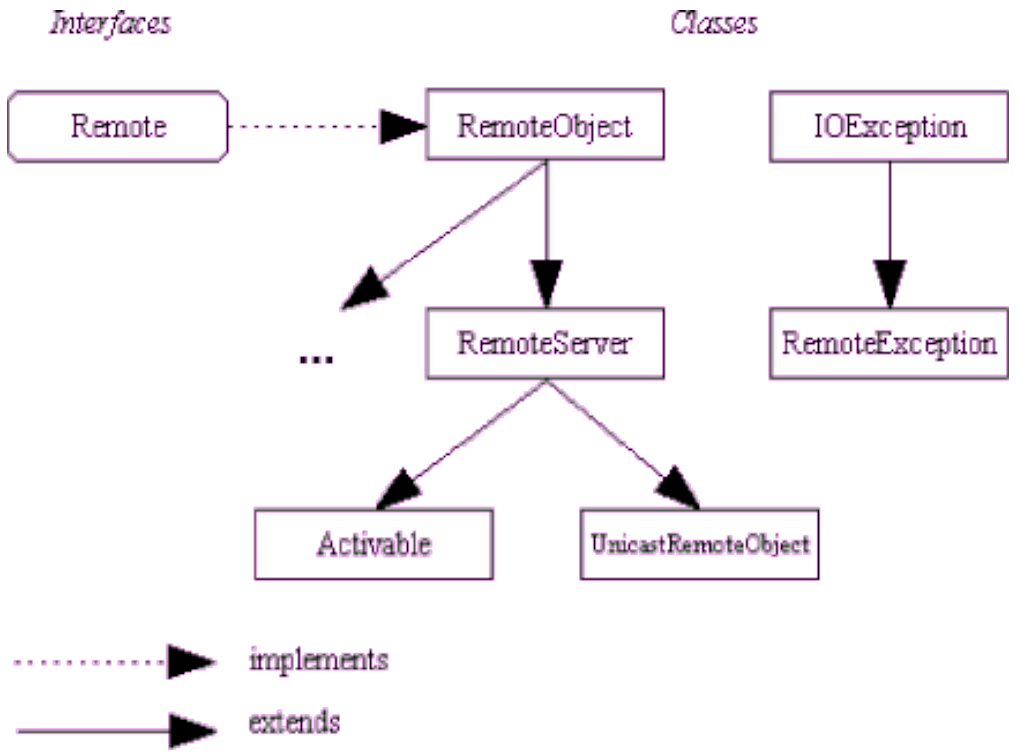


Figure Interfaces and Classes in the java.rmi package.

The Remote Interface
:
A remote interface is defined by extending the Remote interface that is provided in the

java.rmi package. The remote interface is the interface that declares methods that clients can
invoke from a remote virtual machine. The remote interface must satisfy the

following conditions:

1. It must extend the interface Remote.

2. Each remote method declaration in the remote interface must include the exception

RemoteException (or one of it's superclasses) in it's thrown clause.

The RemoteObject Class

RMI server functions are provided by the class RemoteObject and its subclasses

RemoteServer, UnicastRemoteObject and Activatable. Here is a short description of what the
diffrent classes handle:

1. RemoteObject provides implementations of the methods hashCode, equals and

toString in the class java.lang.Object.

2. The classes UnicastRemoteObject and Activatable create remote objects and export

them, i.e. the classes make the remote objects available to remote clients.

The RemoteException Class
:
The class RemoteException is a superclass of the exceptions that the RMI system throws

during a remote method invocation. Each remote method that is declared in a
remote interface must specify RemoteException (or one of it's superclasses) in it's throws
clause to ensure the robustness of applications in the RMI system.
When a remote method invocation fails, the exception RemoteException is thrown.

Communication failure, protocol errors and failure during marshalling or unmarshalling of
parameters or return values are some reasons for RMI failure.

RemoteException is an exception that must be handled by the caller of the remote method, i.e.
it is a checked exception. The compiler ensures that the programmer have handled these
exceptions.

Object Registry

:

The RMI API allows a number of directory services to be used[1] for registering a distributed object.

A simple directory service called the RMI registry, rmiregistry, which is provided with the Java Software Development Kit (SDK)[2]. The RMI Registry is a service whose server, when active, runs on the object server's host machine, by convention and by default on the TCP port 1099.

Implementation of a simple RMI system
:

This is a simple RMI system with a client and a server. The server contains one method

(helloWorld) that returns a string to the client.
To build the RMI system all files has to be compiled. Then the stub and the skeleton, which
are standard mechanisms communication with remote objects, are created with the rmic compiler.

This RMI system contains the following files (the files are shown below):

1. HelloWorld.java: The remote interface.

2. HelloWorldClient.java: The client application in the RMI system.

3. HelloWorldServer.java: The server application in the RMI system.

When all files are compiled, performing the following command will create the stub and the
skeleton:

rmic HelloWorldServer

Then the two classes will be created, HelloWorldServer_Stub.class and

HelloWorldServer_Skel.class, where the first class represents the client side of the RMI

system and the second file represents the server side of the RMI system.

HelloWorld.java

```
/*
Filename: HelloWorld.java
*/
import java.rmi.Remote;
import java.rmi.RemoteException;
/*
Classname: HelloWorld
Comment: The remote interface.
*/
public interface HelloWorld extends Remote {
String helloWorld() throws RemoteException;
}
```

HelloWorldClient.java

```
/*
Filename: HelloWorldClient.java
*/
import java.rmi.Naming;
import java.rmi.RemoteException;
```

```
/*
Classname: HelloWorldClient
Comment: The RMI client.
*/
public class HelloWorldClient {
static String message = "blank";
// The HelloWorld object "obj" is the identifier that is
// used to refer to the remote object that implements
// the HelloWorld interface.
static HelloWorld obj = null;
public static void main(String args[])
{
try {
obj = (HelloWorld)Naming.lookup("//"
+ "kvist.cs.umu.se"
+ "/HelloWorld");
message = obj.helloWorld();
System.out.println("Message from the RMI-server was: \""
+ message + "\"");
}
catch (Exception e) {
System.out.println("HelloWorldClient exception: "
```

```
+ e.getMessage());
e.printStackTrace();
}
}
}
```

HelloWorldServer.java

```
/*
Filename: HelloWorldServer.java
*/
import
import
import
import
java.rmi.Naming;
java.rmi.RemoteException;
java.rmi.RMISecurityManager;
java.rmi.server.UnicastRemoteObject;
/*
Classname: HelloWorldServer
Purpose: The RMI server.
*/
```

```
public class HelloWorldServer extends UnicastRemoteObject
implements HelloWorld {
public HelloWorldServer() throws RemoteException {
super();
}
public String helloWorld() {
System.out.println("Invocation to helloWorld was succesful!");
return "Hello World from RMI server!";
}
public static void main(String args[]) {
try {
// Create an object of the HelloWorldServer class.
HelloWorldServer obj = new HelloWorldServer();
// Bind this object instance to the name "HelloServer".
Naming.rebind("HelloWorld", obj);
System.out.println("HelloWorld bound in registry");
}
catch (Exception e) {
System.out.println("HelloWorldServer error: " + e.getMessage());
e.printStackTrace();
}
}
}
```

RPC Mechanism:

Theory:

RPCGEN is an interface generator pre-compiler for Sun Microsystems RPC. It uses an interface definition file to create client and server stubs in C.

RPCGEN creates stubs based on information contained within an IDL file. This file is written in a language called RPCL - remote procedure call language. This language closely mimics C in style.

An RPC specification contains a number of definitions. These definitions are used by RPCGEN to create a header file for use by both the client and server, and client and server stubs.

RCPL Definitions: Constant, Enumeration, Struct, Union, and TypeDef, Program.

Remote Procedure Call (RPC) is a **protocol** that one program can use to request a service from a program located in another computer in a network without having to understand network details. (A *procedure call* is also sometimes known as a *function call* or a *subroutine call*.) RPC uses the **client/server** model. The requesting program is a client and the service-providing program is the server. Like a regular or local procedure call, an RPC is a **synchronous** operation requiring the requesting program to be suspended until the results of the remote procedure are returned.

When program statements that use RPC are **compiled** into an executable program, a **stub** is included in the compiled code that acts as the representative of the remote procedure code. When the program is run and the procedure call is issued, the stub receives the request and forwards it to a client **runtime** program in the local computer. The client runtime program has the knowledge of how to address the remote computer and server application and sends the message across the network that requests the remote procedure. Similarly, the server includes a runtime program and stub that interface with the remote procedure itself. Results are returned the same way.

Remote procedure calls (RPC) extend the capabilities of conventional procedure calls across a network and are essential in the development of distributed systems.

They can be used both for data exchange in distributed file and database systems and for harnessing the power of multiple processors.

Linux distributions provide an RPC version derived from the RPC facility developed by the Open Network Computing (ONC) group at Sun Microsystems.

RPC and the Client/Server Model:

5. **Caller**: a program which calls a subroutine
6. **Callee**: a subroutine or procedure which is called by the caller
7. **Client**: a program which requests a connection to and service from a network server

8. **Server**: a program which accepts connections from and provides services to a client

There is a direct parallel between the caller/callee relationship and the client/server relationship. With ONC RPC (and with every other form of RPC that I know), the caller always executes as a client process, and the callee always executes as a server process.

The Remote Procedure Call Mechanism

In order for an RPC to execute successfully, several steps must take place:

5. The caller program must prepare any input parameters to be passed to the RPC. Note that the caller and the callee may be running completely different hardware, and that certain data types may be represented differently from one machine architecture to the next. Because of this, the caller cannot simply feed *raw* data to the remote procedure.
6. The calling program must somehow pass its data to the remote host which will execute the RPC. In local procedure calls, the target address is simply a machine address on the local processor. With RPC, the target procedure has a machine address combined with a network address.
7. The RPC receives and operates on any input parameters and passes the result back to the caller.
8. The calling program receives the RPC result and continues execution.

External Data Representation

As was pointed out earlier, an RPC can be executed between two hosts that run completely different processor hardware. Data types, such as integer and floating-point numbers, can have different physical representations on different machines. For example, some machines store integers (C ints) with the low order byte first while some machines place the low order byte last. Similar problems occur with floating-point numeric data. The solution to this problem involves the adoption of a standard for data interchange.

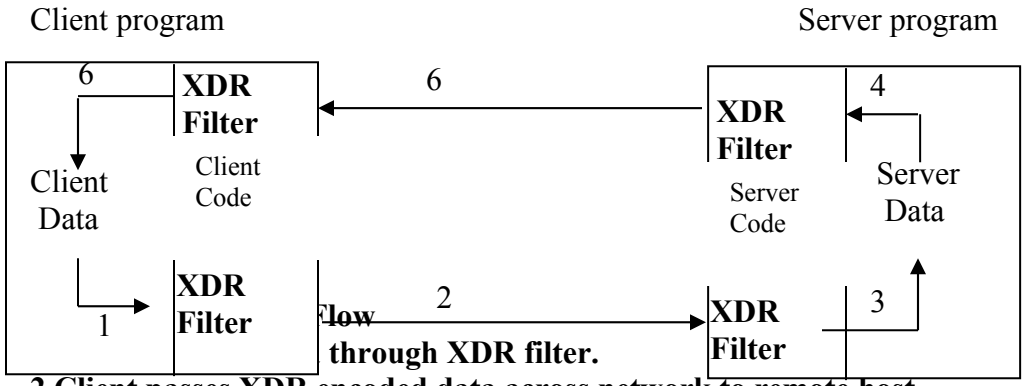
One such standard is the ONC external data representation (XDR). XDR is essentially a collection of C functions and macros that enable conversion from machine specific data representations to the corresponding standard representations and vice versa. It contains primitives for simple data types such as int, float and string and provides the capability to define and transport more complex ones such as records, arrays of arbitrary element type and pointer bound structures such as linked lists.

Most of the XDR functions require the passing of a pointer to a structure of ``XDR'' type. One of the elements of this structure is an enumerated field called **x_op**. It's possible values are **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**. The **XDR_ENCODE** operation instructs the XDR routine to convert the passed data to XDR format. The **XDR_DECODE** operation indicates the conversion of XDR represented

data back to its local representation. **XDR_FREE** provides a means to deallocate memory that was dynamically allocated for use by a variable that is no longer needed.

RPC Data Flow

The flow of data from caller to callee and back again is illustrated in Figure 1. The calling program executes as a client process and the RPC runs on a remote server. All data movement between the client and the network and between the server and the network pass through XDR filter routines. In principle, any type of network transport can be used, but our discussion of implementation specifics centers on ONC RPC which typically uses either Transmission Control Protocol routed by Internet Protocol (the familiar TCP/IP) or User Datagram Protocol also routed by Internet Protocol (the possibly not so familiar UDP/IP). Similarly, any type of data representation could be used, but our discussion focuses on XDR since it is the method used by ONC RPC.



through XDR filter.

2.Client passes XDR encoded data across network to remote host.
3.Server decodes data through XDR Filter.
4.Server encodes function call result through XDR Filter.
5.Server pass XDR encoded data across network back to client.
6.Client decodes RPC result through XDR Filter and continues processing.

Review of Network Programming Theory

In order to complete our picture of RPC processing, we'll need to review some network programming theory. In order for two processes running on separate computers to exchange data, an **association** needs to be formed on each host. An association is defined as the following 5-tuple: *{protocol, local-address, local-process, foreign-address, foreign-process}*

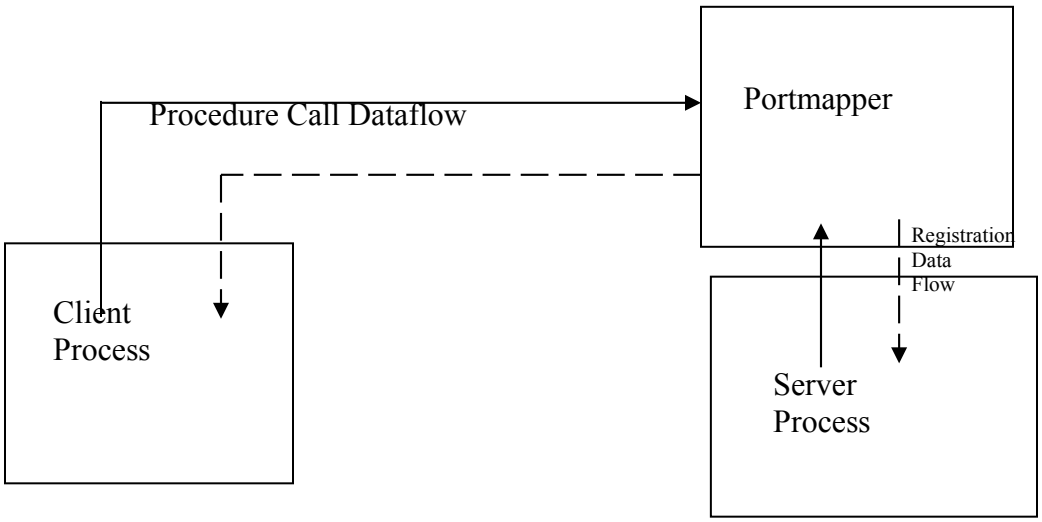
The *protocol* is the transport mechanism (typically TCP or UDP) which is used to move the data between hosts. This, of course, is the part that needs to be common to both host computers. For either host computer, the *local-address/process* pair defines the

endpoint on the host computer running that process. The *foreign-address/process* pair refers to the endpoint at the opposite end of the connection.

Breaking this down further, the term *address* refers to the network address assigned to the host. This would typically be an Internet Protocol (IP) address. The term *process* refers not to an actual process identifier (such as a Unix PID) but to some integer identifier required to transport the data to the correct process once it has arrived at the correct host computer. This is generally referred to as a **port**. The reason port numbers are used is that it is not practical for a process running on a remote host to know the PID of a particular server. Standard port numbers are assigned to well known services such as TELNET (port 23) and FTP (port 21).

RPC Call Binding

Now we have the necessary theory to complete our picture of the RPC binding process. An RPC application is formally packaged into a *program* with one or more *procedure* calls. In a manner similar to the port assignments described above, the RPC program is assigned an integer identifier known to the programs which will call its procedures. Each procedure is also assigned a number that is also known by its caller. ONC RPC uses a program called **portmap** to allocate port numbers for RPC programs. It's operation is illustrated in Figure 2. When an RPC **program** is started, it registers itself with the portmap process running on the same host. The portmap process then assigns the TCP and/or UDP port numbers to be used by that application.



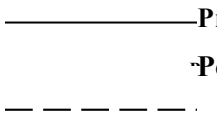


Figure 2. Portmap Operation

The RPC application then waits for and accepts connections at that port number. Prior to calling the remote procedure, the *caller* also contacts portmap in order to obtain the corresponding port number being used by the application whose procedures it needs to call. The network connection provides the means for the caller to reach the correct program on the remote host. The correct procedure is reached through the use of a dispatch table in the RPC program. The same registration process that establishes the port number also creates the dispatch table. The dispatch table is indexed by procedure number and contains the addresses of all the XDR filter routines as well as the addresses of the actual procedures.

MTRPC is a standard linux librpc.a SUN RPC modified to implement a multi threaded execution environment for remote procedure calls. It will make your rpc servers become multi threaded by just compiling them with -lmtrpc) if you use TCP transport).

References: Remote procedure call,http://www.linuxjournal.com/article/2204

2.http://h30097.www3.hp.com

TITLE	Inter process communication
TOPIC for PROBLEM STATEMENT /DEFINITION	Implement IPC Mechanism to show how distributed shared memory works. a) shared memory b) named pipes c) socket
OBJECTIVE	To study IPC Mechanism
S/W PACKAGES AND HARDWARE APPARATUS USED	Ubuntu 10.04,Fedora, C programming
REFERENCES	1. Unix Network Programming by W. Richard Stevens. Published by Prentice Hall. 2. Beej's Guide to Network Programming http:/beej.us/guide/bgnet/ 3. Shared memory,http://tuxthink.blogspot.in/2012/03/inter-process-communication-using_19.html
STEPS	Refer to student activity theory
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Objectives • Theory • Class Diagram/UML diagram • Test cases • Program Listing • Output • Conclusion

What is shared memory?

Shared memory is the fastest interprocess communication mechanism. The operating system maps a memory segment in the address space of several processes, so that several processes can read and write in that memory segment without calling operating system functions. However, we need some kind of synchronization between processes that read and write shared memory.

Consider what happens when a server process wants to send an HTML file to a client process that resides in the same machine using network mechanisms:

- The server must read the file to memory and pass it to the network functions, that copy that memory to the OS's internal memory.
- The client uses the network functions to copy the data from the OS's internal memory to its own memory.

As we can see, there are two copies, one from memory to the network and another one from the network to memory. And those copies are made using operating system calls that normally are expensive. Shared memory avoids this overhead, but we need to synchronize both processes:

- The server maps a shared memory in its address space and also gets access to a synchronization mechanism. The server obtains exclusive access to the memory using the synchronization mechanism and copies the file to memory.
- The client maps the shared memory in its address space. Waits until the server releases the exclusive access and uses the data.

Using shared memory, we can avoid two data copies, but we have to synchronize the access to the shared memory segment.

Creating memory segments that can be shared between processes

To use shared memory, we have to perform 2 basic steps:

- Request to the operating system a memory segment that can be shared between processes. The user can create/destroy/open this memory using a **shared memory**

object: *An object that represents memory that can be mapped concurrently into the address space of more than one process..*

- Associate a part of that memory or the whole memory with the address space of the calling process. The operating system looks for a big enough memory address range in the calling process' address space and marks that address range as an special range. Changes in that address range are automatically seen by other process that also have mapped the same shared memory object.

Once the two steps have been successfully completed, the process can start writing to and reading from the address space to send to and receive data from other processes. Now, let's see how can we do this using **Boost.Interprocess**:

Named Pipe:

A named pipe is really just a special kind of file (a FIFO file) on the local hard drive. Unlike a regular file, a FIFO file does not contain any user information. Instead, it allows two or more processes to communicate with each other by reading/writing to/from this file.

Creating a FIFO File
The easiest way to create a FIFO file is to use the *mkfifo* command. This command is part of the standard Linux utilities and can simply be typed at the command prompt of your shell. You may also use the *mknod* command to accomplish the same thing. This command requires an additional argument but otherwise it works pretty much the same. To learn more about these commands check out the man pages on them (just type *man mkfifo* or *man mknod* at the shell command prompt). The following example shows one way to use the mkfifo command:

prompt> mkfifo /tmp/myFIFO

That's it! It's pretty simple to set a file up for FIFO. There is also a system call that allows you to do this so you can put it in a program. You can read about that on your own if you would like.

Using a FIFO File
Since this named pipe looks like a file, you can use all the system calls associated with files to interact with it. In particular, you can use the *open, read, write*, and *close* system calls. The following are prototypes for each of these calls as you will need to use them.

- *int open(const char *pathname, int flags);*
- *int read(int fd, void *buf, size_t count);*
- *int write(int fd, const void *buf, size_t count);*

- *int close(fd);*

To understand better what each of the return values and parameters are for, check each of these system calls out by looking at their respective man pages. Be sure to use the 2nd level of the man pages (type *man 2 open*, for example).

To give you an idea of how all of these can work together, here is a breif snippet showing how a sender might set up a connection and send a message.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define MAX_LINE 80

int main(int argc, char** argv) {
  char line[MAX_LINE];
  int pipe;

  // open a named pipe
  pipe = open("/tmp/myFIFO", O_WRONLY);

  // get a line to send
  printf("Enter line: ");
  fgets(line, MAX_LINE, stdin);

  // actually write out the data and close the pipe
  write(pipe, line, strlen(line));

        // close the pipe
  close(pipe);
  return 0;
}
```

Reading from the pipe is very similar. You need to open it up, use the *read* call to read information, and then close the pipe when everything is finished. More details on this process can be found on-line or by browsing various man pages. Please note that reading and writing to named pipes are blocking in nature. For example, if a process writes to a named pipe, it will get blocked until there is process willing to read that pipe and vice versa.

TITLE	HTTP Header Analyzer
TOPIC for PROBLEM STATEMENT /DEFINITION	Write a program for application running over the HTTP protocol and read the HTTP header of this application
OBJECTIVE	To study HTTP Request and Response headers used in case of Web Browser request handling and Web Server
S/W PACKAGES AND HARDWARE APPARATUS USED	Ubuntu 10.04,Fedora, C programming
REFERENCES	Hypertext Transfer Protocol -- HTTP/1.1," http://www.w3.org/Protocols/rfc2616/rfc2616.html"
STEPS	Refer to student activity theory
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Theory along-with Mathematical Model • Source Code • Output • Conclusion

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers . A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred. HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification defines the protocol referred to as "HTTP/1.1", and is an update to RFC 2068

Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

```
Request       = Request-Line              ;
                *(( general-header        ;
                 | request-header         ;
                 | entity-header ) CRLF)  ;
                CRLF
                [ message-body ]          ;
```

Request-Line

The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

```
Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

Method

The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.

```
Method         = "OPTIONS"                ;
               | "GET"                    ;
               | "HEAD"                   ;
               | "POST"                   ;
               | "PUT"                    ;
               | "DELETE"                 ;
               | "TRACE"                  ;
               | "CONNECT"                ;
               | extension-method
extension-method = token
```

The list of methods allowed by a resource can be specified in an Allow header field. The return code of the response always notifies the client whether a method is currently allowed on a resource, since the set of allowed methods can change dynamically. An origin server SHOULD return the status code 405 (Method Not Allowed) if the method is known by the origin server but not allowed for the requested resource, and 501 (Not Implemented) if the method is unrecognized or not implemented by the origin server. The methods GET and HEAD MUST be supported by all general-purpose servers.

Request-URI

The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request.

```
Request-URI    = "*" | absoluteURI | abs_path | authority
```

The four options for Request-URI are dependent on the nature of the request. The asterisk "*" means that the request does not apply to a particular resource, but to the server itself, and is only allowed when the method used does not necessarily apply to a resource. One example would be

```
OPTIONS * HTTP/1.1
```

The absoluteURI form is REQUIRED when the request is being made to a proxy. The proxy is requested to forward the request or service it from a valid cache, and return the response. Note that the proxy MAY forward the request on to another proxy or directly to the server

specified by the absoluteURI. In order to avoid request loops, a proxy MUST be able to recognize all of its server names, including any aliases, local variations, and the numeric IP address. An example Request-Line would be:

```
GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1
```

To allow for transition to absoluteURIs in all requests in future versions of HTTP, all HTTP/1.1 servers MUST accept the absoluteURI form in requests, even though HTTP/1.1 clients will only generate them in requests to proxies.

The authority form is only used by the CONNECT method

The most common form of Request-URI is that used to identify a resource on an origin server or gateway. In this case the absolute path of the URI MUST be transmitted as the Request-URI, and the network location of the URI (authority) MUST be transmitted in a Host header field. For example, a client wishing to retrieve the resource above directly from the origin server would create a TCP connection to port 80 of the host "www.w3.org" and send the lines:

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host: www.w3.org
```

followed by the remainder of the Request. Note that the absolute path cannot be empty; if none is present in the original URI, it MUST be given as "/" (the server root).

If the Request-URI is encoded using the "% HEX HEX" encoding [42], the origin server MUST decode the Request-URI in order to properly interpret the request. Servers SHOULD respond to invalid Request-URIs with an appropriate status code.

A transparent proxy MUST NOT rewrite the "abs_path" part of the received Request-URI when forwarding it to the next inbound server, except as noted above to replace a null abs_path with "/".

The Resource Identified by a Request

The exact resource identified by an Internet request is determined by examining both the Request-URI and the Host header field.

An origin server that does not allow resources to differ by the requested host MAY ignore the Host header field value when determining the resource identified by an HTTP/1.1 request. An origin server that does differentiate resources based on the host requested (sometimes referred to as virtual hosts or vanity host names) MUST use the following rules for determining the requested resource on an HTTP/1.1 request:

1. If Request-URI is an absoluteURI, the host is part of the Request-URI. Any Host header field value in the request MUST be ignored.

2. If the Request-URI is not an absoluteURI, and the request includes a Host header field, the host is determined by the Host header field value.

3. If the host as determined by rule 1 or 2 is not a valid host on the server, the response MUST be a 400 (Bad Request) error message.

Recipients of an HTTP/1.0 request that lacks a Host header field MAY attempt to use heuristics (e.g., examination of the URI path for something unique to a particular host) in order to determine what exact resource is being requested.

Request Header Fields

The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method invocation.

```
request-header = Accept                ;
               | Accept-Charset        ;
               | Accept-Encoding       ;
               | Accept-Language       ;
               | Authorization         ;
               | Expect                ;
               | From                  ;
               | Host                  ;
               | If-Match              ;
               | If-Modified-Since     ;
               | If-None-Match         ;
               | If-Range              ;
               | If-Unmodified-Since   ;
               | Max-Forwards          ;
               | Proxy-Authorization   ;
               | Range                 ;
               | Referer               ;
               | TE                    ;
               | User-Agent            ;
```

Request-header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields MAY be given the semantics of request- header fields if all parties in the communication recognize them to be request-header fields. Unrecognized header fields are treated as entity-header fields.

Response

After receiving and interpreting a request message, a server responds with an HTTP response message.

```
Response      = Status-Line               ;
                *(( general-header        ;
                 | response-header        ;
                 | entity-header ) CRLF)  ;
                CRLF
                [ message-body ]          ;
```

Status-Line

The first line of a Response message is the Status-Line, consisting of the protocol version followed by a numeric status code and its associated textual phrase, with each element separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Status Code and Reason Phrase

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. These codes are fully defined in 10. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine or display the Reason- Phrase.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- 1xx: Informational - Request received, continuing process
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for HTTP/1.1, and an example set of corresponding Reason-Phrase's, are presented below. The reason phrases listed here are only recommendations -- they MAY be replaced by local equivalents without affecting the protocol.

```
Status-Code    =
      "100"  ;  Continue
    | "101"  ;  Switching Protocols
    | "200"  ;  OK
    | "201"  ;Created
    | "202"  ;  Accepted
    | "203"  ; Non-Authoritative Information
    | "204"  ;No Content
    | "205"  ;  Reset Content
    | "206"  ;Partial Content
    | "300"  ; Multiple Choices
    | "301"  ; Moved Permanently
    | "302"  ; Found
    | "303"  ;  See Other
    | "304"  ; Not Modified
    | "305"  ; Use Proxy
    | "307"  ;Temporary Redirect
    | "400"  ;   Bad Request
    | "401"  ;   Unauthorized
    | "402"  ;   Payment Required
    | "403"  ;   Forbidden
    | "404"  ;  Not Found
    | "405"  ;   Method Not Allowed
    | "406"  ;   Not Acceptable
    | "407"  ;   Proxy Authentication Required
    | "408"  ;  Request Time-out
    | "409"  ;  Conflict
    | "410"  ;   Gone
    | "411"  ;  Length Required
    | "412"  ;  Precondition Failed
    | "413"  ;   Request Entity Too Large
    | "414"  ;  Request-URI Too Large
    | "415"  ;   Unsupported Media Type
    | "416"  ;   Requested range not satisfiable
    | "417"  ;  Expectation Failed
    | "500"  ;  Internal Server Error
    | "501"  ;  Not Implemented
    | "502"  ;  Bad Gateway
    | "503"  ;  Service Unavailable
    | "504"  ;  Gateway Time-out
```

```
    | "505"  ; HTTP Version not supported
    | extension-code
extension-code = 3DIGIT
Reason-Phrase  = *<TEXT, excluding CR, LF>
```

HTTP status codes are extensible. HTTP applications are not required to understand the meaning of all registered status codes, though such understanding is obviously desirable. However, applications MUST understand the class of any status code, as indicated by the first digit, and treat any unrecognized response as being equivalent to the x00 status code of that class, with the exception that an unrecognized response MUST NOT be cached. For example, if an unrecognized status code of 431 is received by the client, it can safely assume that there was something wrong with its request and treat the response as if it had received a 400 status code. In such cases, user agents SHOULD present to the user the entity returned with the response, since that entity is likely to include human- readable information which will explain the unusual status.

Response Header Fields

The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status- Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

```
response-header = Accept-Ranges          ;
                | Age                    ;
                | ETag                   ;
                | Location               ;
                | Proxy-Authenticate     ;
                | Retry-After            ;
                | Server                 ;
                | Vary                   ;
                | WWW-Authenticate       ;
```

Response-header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields MAY be given the semantics of response- header fields if all parties in the communication recognize them to be response-header fields. Unrecognized header fields are treated as entity-header fields.

TITLE	**Finding and using the idle work station in the network.**
PROBLEM STATEMENT /DEFINITION	Write a program for finding and using the idle work station in the network with the help of registry based algorithm/Configuration files.
OBJECTIVE	1. Understand what is idle workstation. 2. Use system CPU usage statistics. 3. List of idle workstations.
S/W PACKAGES AND HARDWARE APPARATUS USED	Editors and compilers on linux gcc/cc Linux Tools like sysstat Network and workstations with the configuration as Pentium IV 2.4 GHz. 1 GB RAM, 40 G.B HDD, 15''Color Monitor, Keyboard, Mouse
REFERENCES	1. "Distributed operating systems", Andrew S.Tanenbaum 2. "Advanced Linux Programming", Mark Mitchell, Jeffrey Oldham,and Alex Samue
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	Title Problem Definition Objectives Theory Class Diagram/UML diagram Test cases Program Listing Output Conclusion

Aim: Finding and using the idle work station in the network.
Pre requisite:

1. Linux commands and system calls such as top, uptime, mpstat, sar etc

2. Socket programming APIs.

Learning Objectives:

1. To find when the workstation is idle.
2. To understand approaches of finding idle workstations.
3. To find the list of idle workstations.

Learning Outcomes:

The students will be able to

- Use Linux commands to get CPU usage statistics.
- Analyze systems CPU usage statistics to identify idle workstation.
- Prepare the list of idle workstations.

Theory:
What is an idle workstation?

If no one has touched the keyboard or mouse for several minutes and no user-initiated processes are running, the workstation can be said to be idle.

The algorithms used to locate idle workstations can be divided into two categories:

Server driven- if a server is idle, it registers in registry file or broadcasts to every machine.

Client driven - the client broadcasts a request asking for the specific machine that it needs and wait for the reply.

A. Server Driven:
When a workstation goes idle, and thus becomes a potential compute server, it announces its availability. It can do this by entering its name, network address, and properties in a registry file (or data base), for example.
Later, when a user wants to execute a command on an idle workstation, he types something like `remote command` and the *remote* program looks in the registry to find a suitable idle workstation. For reliability reasons, it is also possible to have multiple copies of the registry.

B. Client Driven:
In client driven approach, the newly idle workstation announce the fact that it has become unemployed by putting a broadcast message onto the network.

All other workstations then record this fact. In effect, each machine maintains its own private copy of the registry.
The advantage of doing it this way is less overhead in finding an idle workstation and greater redundancy.
The disadvantage is requiring all machines to do the work of maintaining the registry.
Whether there is one registry or many, there is a potential danger of race conditions occurring. If two users invoke the *remote* command simultaneously, and both of them discover that the same machine is idle, they may both try to start up processes there at the same time. To detect and avoid this situation, the *remote* program can check with the idle workstation, which, if still free, removes itself from the registry and gives the go-ahead sign. At this point, the caller can send over its environment and start the remote process, as shown in Fig..

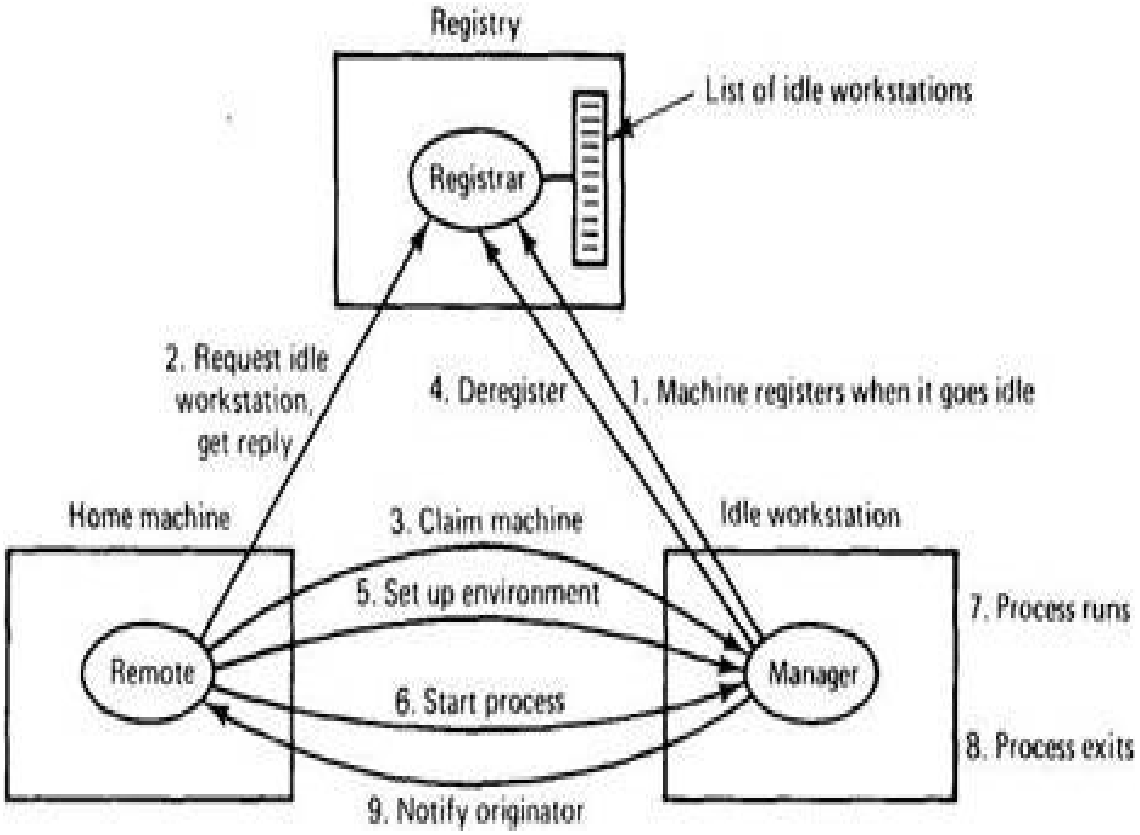


Fig. A registry-based algorithm for finding & using idle workstation.

The other way to locate idle workstations is to use a client-driven approach.
When *remote* is invoked, it broadcasts a request saying what program it wants to run, how much memory it needs, whether or not floating point is needed, and so on.
These details are not needed if all the workstations are identical, but if the system is heterogeneous and not every program can run on every workstation, they are essential.
When the replies come back, *remote* picks one and sets it up.

One twist is to have "idle" workstations delay their responses slightly, with the delay being proportional to the current load.
In this way, the reply from the least heavily loaded machine will come back first and be selected.

Description using set theory:

Let 'S' be set which represents a system S={I,O,N, Succ , Fail}

where,

I=Input (Request to find idle workstation)
O=Output (list of idle workstations)
N=Number of workstations in the network.

O={Wn,n}

Where,

Wn=List of IP addresses of idle workstations
n=number of idle workstations

Success Succ={x |x is set of all cases that are handled in program}

Succ= {All workstations are overloaded | All workstations are idle | more than one request to find idle workstation}

Failures Fail={x |x is set of all cases that are not handled in program}

Fail = {workstation which is initially found idle but later on becomes overloaded}

Registry based approach:

Steps to do /algorithm:

Write program to register when workstation becomes idle. Run this code on all workstations in the network.

Write server program using registry based approach to accept registry requests and keep a list of all registered workstations.

Run the server.

Write a client program requesting for list of idle workstations.

Run client code and get the list from registry server.

FAQs/Review Questions:

1. What are the approaches to find idle workstation in a distributed systems?

2. How registry based algorithm works?

3. How client based algorithm works?

- What are the advantages and disadvantages of each of the approaches?
- What is load balancing in distributed systems?
- What is load sharing?

TITLE	Election algorithm
TOPIC for PROBLEM STATEMENT /DEFINITION	Write a program to simulate election algorithm a. Ring Algorithm b. Bully's Algorithm
OBJECTIVE	To study the use of coordinator
S/W PACKAGES AND HARDWARE APPARATUS USED	Ubuntu 10.04,Fedora, C programming
REFERENCES	[1] Tanenbaum A.S, Distributed Operating System, Pearson Education, 2007. [2] Sinha P.K, Distributed Operating Systems Concepts and Design, Prentice-Hall of India private Limited, 2008. [3] Sandipan Basu / Indian Journal of Computer Science and Engineering (IJCSE)," An Efficient Approach of Election Algorithm in Distributed Systems"
STEPS	Refer to the theory
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Theory along-with Mathematical Model • Source Code • Output • Conclusion

Title: **Simulation of election algorithms**
a. Bully
b. Ring

Objective: Deciding Coordinator for accessing shared resource.

Aim: Simulation of Bully and Ring election algorithms.

Theory:

Many distributed algorithms require one process to act as coordinator, initiator, or to perform some other special role. In the centralized mutual exclusion algorithm, one process is elected as the coordinator. For instance, the process running on the machine with the highest network address might be selected. Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that region, the coordinator sends back a reply granting permission.

It does not matter which process takes on this special responsibility of coordinator, but one of them has to do it. In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator. It is assumed that every process knows the process number of every other process. What the processes do not know is which ones are currently up and which ones are currently down. The goal of election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be.

Bully Election Algorithm

The Bully Algorithm was devised by Garcia-Molina in 1982. When a process notices that the coordinator is no longer responding to requests, it initiates an election. Process P, holds an election as follows:

1) P sends an ELECTION message to all processes with higher numbers.
2) If no one responds, P wins the election and becomes coordinator.
3) If one of the higher-ups answers, it takes over. P's job is done.

At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that it is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and

that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and will take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "Bully Algorithm".

Ring Election Algorithm

This election algorithm is based on the use of a ring. We assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next number along the ring, or the one after that, until a running process is located. At each step, the sender adds its own process number to the list in the message.

Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is (designated by the list member with the highest number) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.

TITLE	**Lamport's algorithm for synchronization of logical clocks.**
PROBLEM STATEMENT /DEFINITION	Write program for synchronization of logical clock using Lamport's algorithm.
OBJECTIVE	• Understand the logical clock implementation in DS • Understand the need for logical clock in DS • Understand the Lamport's clock synchronization algorithm
S/W PACKAGES AND HARDWARE APPARATUS USED	Editors and compilers for C/C++ on Linux Linux/Fedora/Ubuntu PC with the configuration as Pentium IV 2.4 GHz. 1 GB RAM, 40 G.B HDD, 15''Color Monitor, Keyboard, Mouse
REFERENCES	1. Distributed O.S Concepts and Design, P.K.Sinha, PHI 2. Advanced concepts in Operating Systems , Mukesh Singhal & N.G.Shivaratri, TMH 3. Distributed System Principles and Paradigms, Andrew S. Tanenbaum, 2nd edition , PHI
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Objectives • Theory • Class Diagram/UML diagram • Test cases • Program Listing • Output • Conclusion

Aim: Implementation of Lamport's algorithm for synchronization of logical clocks.
Pre requisite:
Clock synchronization in distributed systems.

Physical clock synchronization.

Programming in Linux environment.

Learning Objectives:

- Understand the logical clock implementation in DS
- Understand the need for logical clock in DS
- Understand the Lamport's clock synchronization algorithm

Learning Outcomes:

The students will be able to

- Demonstrate the need for logical clocks in DS
- To demonstrate the need for logical clock synchronization in DS
- To implement happened before relationship among the various events in DS
- To implement Lamport's algorithm for synchronization of logical clocks

Theory:
1.Assume no central time source/global clock.
2. Each system maintains its own local clock.
3.No total ordering of events.
4.No concept of happened-when.
5. Solution proposed by Leslie Lamport was the concept of
logical clocks.

1.To implement "->" in a distributed system, Lamport introduced the concept of logical clocks, which captures"->"numerically.
2. Each process Pi has a logical clock Ci.
3. Clock Ci can assign a value Ci (a) to any event a in process Pi.
4.The value Ci (a) is called the timestamp of event a in process Pi.
5.The value C(a) is called the timestamp of event a in whatever process it occurred.

6. The timestamps have no relation to physical time, whichleads to the term logical clock.
7. The logical clocks can be implemented by simple counters.

Conditions satisfied by Logical Clocks

1. Clock condition: if a->b, then C(a)<C(b).
2. If event a happens before event b, then the clock value (timestamp) of a should be less than the clock value of b.
3. Note that we cannot say: if C(a)<C(b), then a->b.
4. Correctness conditions (must be satisfied by the logical clocks to meet the clock condition above):
5. [C1]: For any two events a and b in the same process Pi , if a happens before b, then Ci (a) < Ci (b).
6. [C2]: If event a is the event of sending a message m inprocess Pi ,and event b is the event of receiving that samemessage m in a different process Pk, then Ci (a) < Ck (b).

Implementation of Logical Clocks

1.Implementation Rules (guarantee that the logical clockssatisfy the correctness conditions):
2.[IR1]: Clock Cimust be incremented between any two successive events in process Pi :Ci := Ci + 1.
3.[IR2]: If event a is the event of sending a message m in process Pi ,then message m is assigned a timestamp tm =Ci (a).
4. When that same message m is received by a different process Pk, Ck is set to a value greater than current value of the counter and the timestamp carried by the message is, Ck := max(Ck , tm+1).

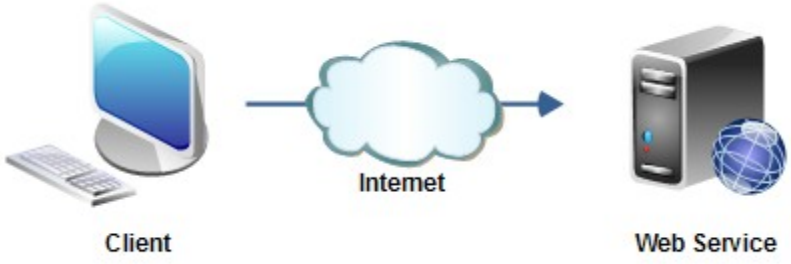
FAQs/Review Questions:

Why logical clocks are implemented in distributed systems?
What is happened-before relationship among the events?
What is total ordering?
What are conditions to be satisfied by the logical clock?
What are the simple implementation rules in Lamport's algorithm?
State the applications where logical synchronization is essential?
Demonstrate the correctness of Lamport's algorithm.

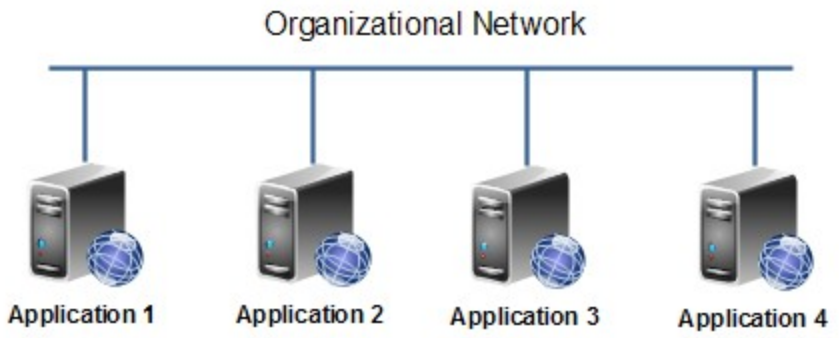
TITLE	**Web Service**
PROBLEM STATEMENT /DEFINITION	Implement any web service
OBJECTIVE	**To publish application through web service**
S/W PACKAGES AND HARDWARE APPARATUS USED	Web service package: gsoap package
REFERENCES	www.w3schools.com/webservices/default.asp
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Objectives • Theory • Class Diagram/UML diagram • Test cases • Program Listing • Output • Conclusion

Theory:

The term "web service" is often used to describe a service that a client (a computer) can call remotely over the internet, via web protocols like HTTP. Like calling a method, procedure or function which is running on a different machine than the client. As such web services are very similar to "remote procedure call" (or just "remoting") protocols, like Java's RMI, Windows DCOM, Corba's IIOP etc. The basic web service principle is illustrated here:



Web services can also be used internally in an organizations network, as a way to allow many different applications to interact with each other. The standardized web service protocols then makes it easier to integrate the various applications. This principle is illustrated her:



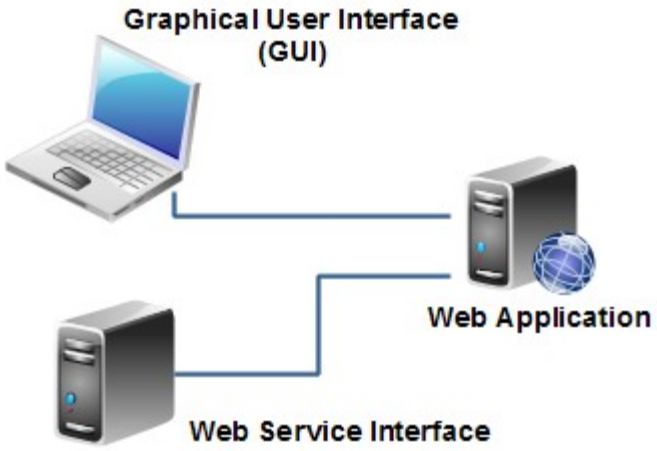
Independent applications on an intranet communicating with each other via web services.

Web Services vs. Web Sites / Web Applications

The main difference between a web service and a web site is, that a web site is typically intended for human consumption (human-to-computer interaction), whereas web services are typically intended for computer-to-computer interaction.

Of course this distinction is somewhat blurred. A web application can contain both a graphical user interface for human users, as well as a set of web services for computer "users" (clients). For instance, a payment service like Paypal has both a graphical user interface for human users, as well as a set of web services through which you can have your own backend systems access the Paypal services.

This illustration shows a web application that contains both a graphical user interface, and a web service interface (a set of web services exposing selected functions of the web application):



A web application with a GUI for human users, and web services for computerized clients.

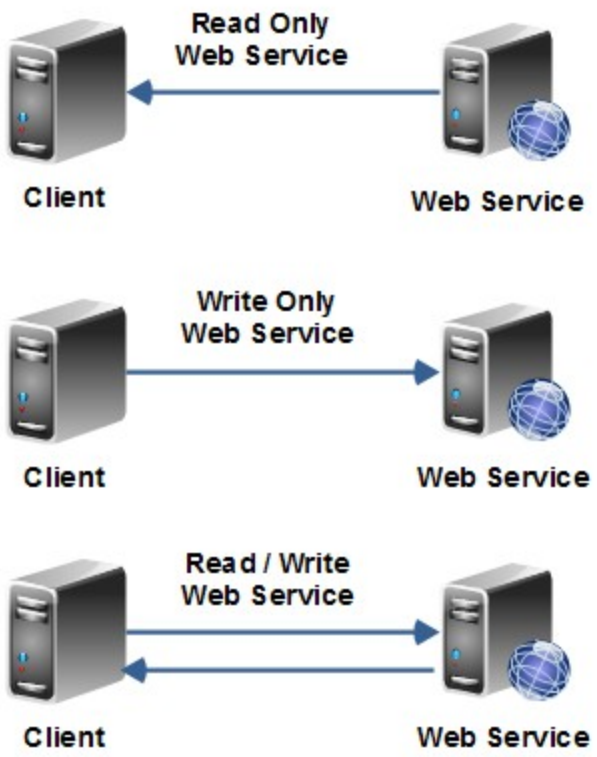
Web Service Message Exchange Patterns

There are multiple types of web services. Some web services a client calls to obtain some information. For instance, a client may call a weather web service to read weather information. These are typical read-only web services. A read-only web service may in practice send an empty request to the web service, which then sends the data back. So, even if a web service is read-only, the client might actually have to send some data (a minimal request) to the web service to obtain the data it wants to read.

Other web services are more write-only kind of web services. For instance, you may transfer data to a web service at regular intervals.

And then others are read-write services where it makes sense to both send data to the web service, and receive data back again.

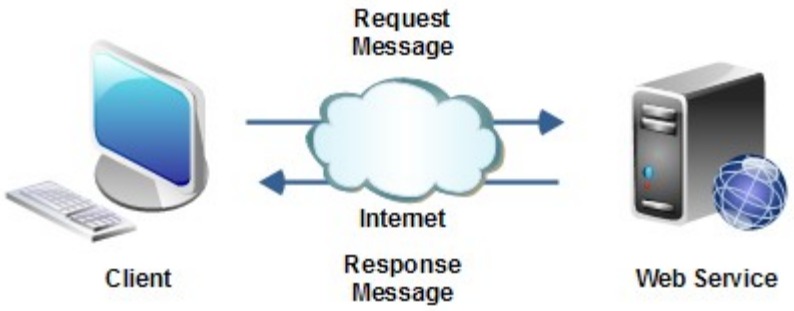
All three communication patterns are illustrated here:



Three message exchange patterns of web services: Read-only, Write-only and Read / Write .

When a client and a web service communicate they exchange messages. A request message is sent from the client to the web service. The web service responds with a response message. This is just like in ordinary HTTP, where a web browser sends an HTTP request to a web server, and the web server replies with an HTTP response.

In the beginning the only web service message format available was SOAP. Later came REST type web services, which uses plain XML and HTTP. Following the REST movement came a wave of people using JSON (JavaScript Object Notation) as message format. Another very simple remoting protocol is called XML-RPC (XML Remote Procedure Call). Of these, the most common is SOAP and I will not get into details with these message formats here, since they will get their own tutorial trails later. I will just briefly mention what they look like.



A clients sends a request message to a web service, and receives a response message.

SOAP

SOAP (Simple Object Access Protocol) is an XML based message format. Here is a simple SOAP message:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Header>
  </soap:Header>

  <soap:Body>

    ... message data ...

    <soap:Fault>
    </soap:Fault>

  </soap:Body>

</soap:Envelope>
```

As you can see a SOAP message consists of:

- Envelope
  - Header
  - Body
    - Message Data
    - Fault (optional)

The same SOAP message structure is used to send both requests and responses between client and web service.

The `Fault` element inside the `Body` element is optional. A `Fault` element is only sent back if an error occurs inside the web service. Otherwise the normal message data is sent back.

SOAP doesn't specify **how** a message gets from the client to the web service, although the most common scenario is via HTTP.

REST + XML

REST (REpresentational State Transfer) style web services work a bit different from SOAP web services. A REST request is a simple HTTP request just like a regular browser would send to a web server. There is typically no XML request sent. A REST response is typically an XML document sent back in a regular HTTP response, just as if a browser had requested it.

In REST you don't think so much in terms of "services", but rather in "resources". A resource has a given URL, just like an HTML page in a web site. An example of a resource could be a user profile in a social application. Such a resource could have the URL:

http://social.jenkov.com/profiles/jakobjenkov

This URL might return an XML document (resource) describing me. Here is how the XML file returned could look:

```
<profile>
   <firstName>Jakob</firstName>
   <lastName>Jenkov</lastName>
   <address>
      <street>The Road 123</street>
      <zip>12345</zip>
      <city>Copenhagen</city>
   </address>
</profile>
```

REST also naturally supports collections of resources. For instance, this URL might represent a list of all public user profiles:

http://social.jenkov.com/profiles/

Here is an example of how such a profile list in XML could look:

```
<profiles>
   <profile>...</profile>
   <profile>...</profile>
   <profile>...</profile>
</profiles>
```

The two URL's above only reads a resource or resource collection. If you need to modify a resource in REST, you do so by sending different HTTP request to the server. When you read a resource you send an HTTP GET request. If you need to write a resource, you would send an HTTP PUT instead. If you need to delete a resource you would send an HTTP DELETE etc.

If a web service is to be "callable" for clients from the outside world, the client need a description of the service interface. Without a description of the interface, how would the client know what data to send to the service?

You can think of a service interface like an interface in Java or C#. The only extra information needed is where the service is located (IP address), and the message format used by the service. Here is what a service description should contain:

- Interface Name
- Operation Name(s) (if the service has more than one operation).
- Operation Input Parameters
- Operation Return Values
- Service Message Format
- Service Location (IP Address / URL)

How a web service interface description looks depends on the message format used by the web service. Currently, only SOAP web services has a standardized interface description - the Web Service Description Language (WSDL).