Experiment No: 1

Socket Programming

Aim:

Implement client Inter-process communication (TCP & UDP) based on socket.

Brief Theory:

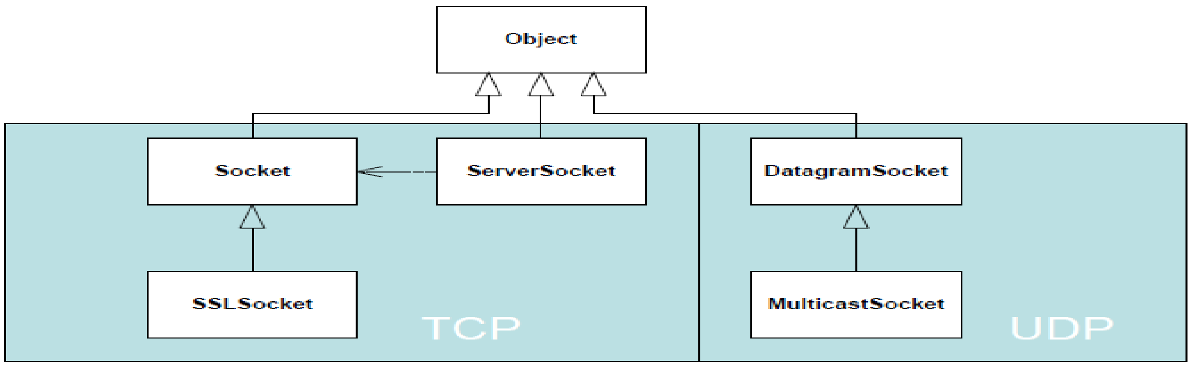
Interprocess communications

When communicating with programs that are running in another process, there are a number of options.

- Use **sockets** for interprocess communication. One program can act as the server program that listens on a socket connection for input from the client program. The client program connects to the server with a socket. Once the socket connection is established, either program can send or receive information.
- Use stream files for communication between programs. To do this, use the System.in, System.out, and System.err classes.
- Use the IBM® Toolbox for Java™, which provides data queues and System i5™ message objects.

There are two types of Internet Protocol (IP) traffic. They are **TCP** or**Transmission Control Protocol** and **UDP** or **User Datagram Protocol**. TCP is connection oriented – once a connection is established, data can be sent bidirectional. UDP is a simpler, connectionless Internet protocol. Multiple messages are sent as packets in chunks using UDP.

Java Socket Hierarchy



Comparison between TCP and UDP

	TCP	UDP
Acronym for	Transmission Control Protocol	User Datagram Protocol or Universal Datagram Protocol
Connection	TCP is a connection-oriented protocol.	UDP is a connectionless protocol.
Function	As a message makes its way across the internet from one computer to another. This is connection based.	UDP is also a protocol used in message transport or transfer. This is not connection based which means that one program can send a load of packets to another and that would be the end of the relationship.
Usage	TCP is suited for applications that require high reliability, and transmission time is relatively less critical.	UDP is suitable for applications that need fast, efficient transmission, such as games. UDP's stateless nature is also useful for servers that answer small queries from huge numbers of clients.
Use by other protocols	HTTP, HTTPs, FTP, SMTP, Telnet	DNS, DHCP, TFTP, SNMP, RIP, VOIP.
Ordering of data packets	TCP rearranges data packets in the order specified.	UDP has no inherent order as all packets are independent of each other. If ordering is required, it has to be managed by the application layer.
Speed of transfer	The speed for TCP is slower than UDP.	UDP is faster because there is no error-checking for packets.

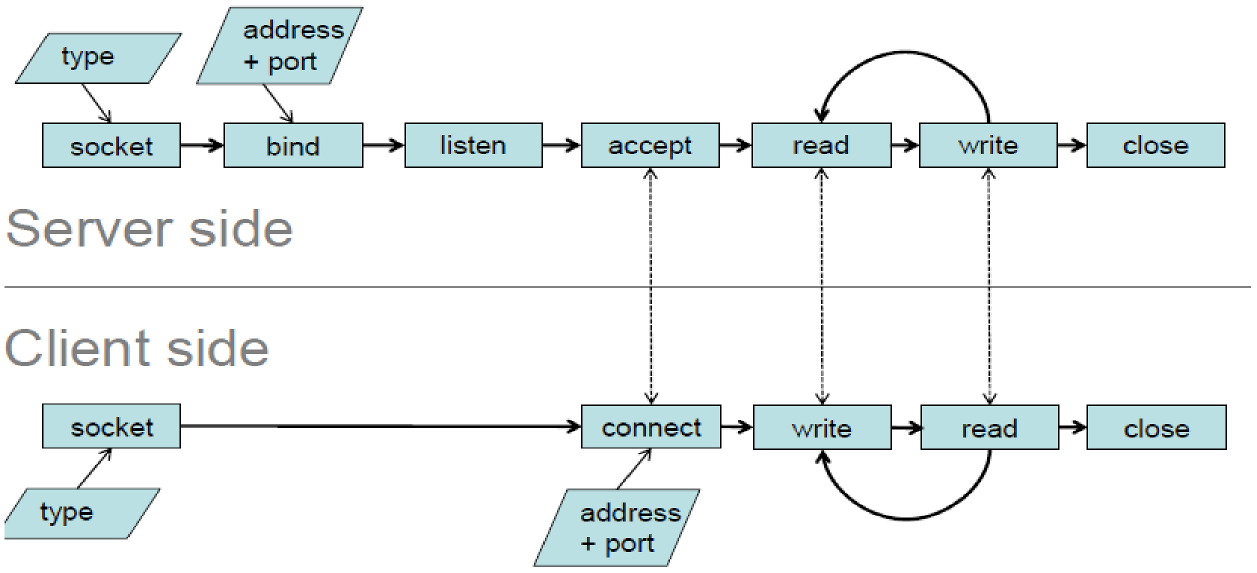
Reliability	There is absolute guarantee that the data transferred remains intact and arrives in the same order in which it was sent.	There is no guarantee that the messages or packets sent would reach at all.
Header Size	TCP header size is 20 bytes	UDP Header size is 8 bytes.
Common Header Fields	Source port, Destination port, Check Sum	Source port, Destination port, Check Sum
Streaming of data	Data is read as a byte stream, no distinguishing indications are transmitted to signal message (segment) boundaries.	Packets are sent individually and are checked for integrity only if they arrive. Packets have definite boundaries which are honored upon receipt, meaning a read operation at the receiver socket will yield an entire message as it was originally sent.
Weight	TCP is heavy-weight. TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control.	UDP is lightweight. There is no ordering of messages, no tracking connections, etc. It is a small transport layer designed on top of IP.
Data Flow Control	TCP does Flow Control. TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control.	UDP does not have an option for flow control
Error Checking	TCP does error checking	UDP does error checking, but no recovery options.

Fields	1. Sequence Number, 2. AcK number, 3. Data offset, 4. Reserved, 5. Control bit, 6. Window, 7. Urgent Pointer 8. Options, 9. Padding, 10. Check Sum, 11. Source port, 12. Destination port	1. Length, 2. Source port, 3. Destination port, 4. Check Sum
Acknowledgement	Acknowledgement segments	No Acknowledgment

How TCP and UDP work

A **TCP** connection is established via a three way handshake, which is a process of initiating and acknowledging a connection. Once the connection is established data transfer can begin. After transmission, the connection is terminated by closing of all established virtual circuits.

UDP uses a simple transmission model without implicit hand-shaking dialogues for guaranteeing reliability, ordering, or data integrity. Thus, UDP provides an unreliable service and datagrams may arrive out of order, appear duplicated, or go missing without notice. UDP assumes that error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level. Unlike TCP, UDP is compatible with packet broadcasts (sending to all on local network) and multicasting (send to all subscribers).



Different Applications of TCP and UDP

Web browsing, email and file transfer are common applications that make use of TCP. TCP is used to control segment size, rate of data exchange, flow control and network congestion. TCP is preferred where error correction facilities are required at network interface level. UDP is largely used by time sensitive applications as well as by servers that answer small queries from huge number of clients. UDP is compatible with packet broadcast - sending to all on a network and multicasting – sending to all subscribers. UDP is commonly used in Domain Name System, Voice over IP, Trivial File Transfer Protocol and online games.

Java Socket Programming

Package for socket programming

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

Classes

InetAddress

Socket

ServerSocket

DatagramSocket

DatagramPacket

Methods of InetAddress Classes

getByName(String host)

getAllByName(String host)

getLocalHost()

```
InetAddress a = InetAddress.getByName(hostname);
```

Socket Classes

- ❑ Corresponds to active TCP sockets only!
  - ❖ client sockets
  - ❖ socket returned by accept();

- Passive sockets are supported by a different class:
  - ServerSocket
- UDP sockets are supported by
  - DatagramSocket

- java.net.Socket
  - Implements client sockets (also called just “sockets”).
  - An endpoint for communication between two machines.
  - Constructor and Methods
    - Socket(String host, int port): Creates a stream socket and connects it to the specified port number on the named host.
    - InputStream getInputStream()
    - OutputStream getOutputStream()
    - close()
- java.net.ServerSocket
  - Implements server sockets.
  - Waits for requests to come in over the network.
  - Performs some operation based on the request.
  - Constructor and Methods
    - ServerSocket(int port)
    - Socket Accept(): Listens for a connection to be made to this socket and accepts it. This method blocks until a connection is made.
- Constructor creates a TCP connection to a named TCP server.
  - There are a number of constructors:

```
Socket(InetAddress server, int port);
Socket(InetAddress server, int port,InetAddress local, int localport);
Socket(String hostname, int port);
```

UDP Socket

- In Package java.net
  - java.net.DatagramSocket
    - A socket for sending and receiving datagram packets.

- Constructor and Methods
  - DatagramSocket(int port): Constructs a datagram socket and binds it to the specified port on the local host machine.
  - void receive(DatagramPacket p)
  - void send(DatagramPacket p)
  - void close()

The following steps occur when establishing a TCP connection between two computers using sockets:

1. The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
2. The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
3. After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.
4. The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.
5. On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a twoway communication protocol, so data can be sent across both streams at the same time.

Conclusion:

Socket programming is used for inter-process communication and as we have used UDP programming which was connectionless and so we could only send one way data.

Experiment No: 2

Implementation of multi thread application

Aim:

Implementation of multi thread application (basic mathematical operations).

Brief Theory:

A thread is a path of execution within a process. A process can contain multiple threads.

A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.

A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Example of Multi thread:

```
   packagedemotest;

   public class GuruThread1 implements Runnable

   {

   public static void main(String[] args) {

   Thread guruThread1 = new Thread("Guru1");

   Thread guruThread2 = new Thread("Guru2");

   guruThread1.start();

   guruThread2.start();

   System.out.println("Thread names are following:");

   System.out.println(guruThread1.getName());

   System.out.println(guruThread2.getName());

}
   @Override

   public void run() {

   }

    }
```

Conclusion: Multithreading in Java is a process of executing two or more threads simultaneously to maximum utilization of CPU. Multithreaded applications execute two or more threads run concurrently. Each thread runs parallel to each other. Multiple threads don't allocate separate memory area, hence they save memory. Also, context switching between threads takes less time.

Experiment No: 3

Group Communication (Absolute ordering).

Aim: Implementation of group communication in distributed system.

Theory:

Group Communication

A group is an operating system abstraction for a collective of related processes. A set of cooperative processes may, for example, form a group to provide an extendable, efficient, available and reliable service. The group abstraction allows member processes to perform computation on different hosts while providing support for communication and synchronization between them.

The term multicast means the use of a single communication primitive to send a message to a specific set of processes rather than using a collection of individual point to point message primitives. This is in contrast with the term broadcast which means the message is addressed to every host or process.

A consensus protocol allows a group of participating processes to reach a common decision, based on their initial inputs, despite failures.

A reliable multicast protocol allows a group of processes to agree on a set of messages received by the group. Each message should be received by all members of the group or by none. The order of these messages may be important for some applications. A reliable multicast protocol is not concerned with message ordering; only message delivery guarantees. Ordered delivery protocols can be implemented on top of a reliable multicast service.

Multicast algorithms can be built on top of lower-level communication primitives such as point-to-point sends and receives or perhaps by availing of specific network mechanisms designed for this purpose.

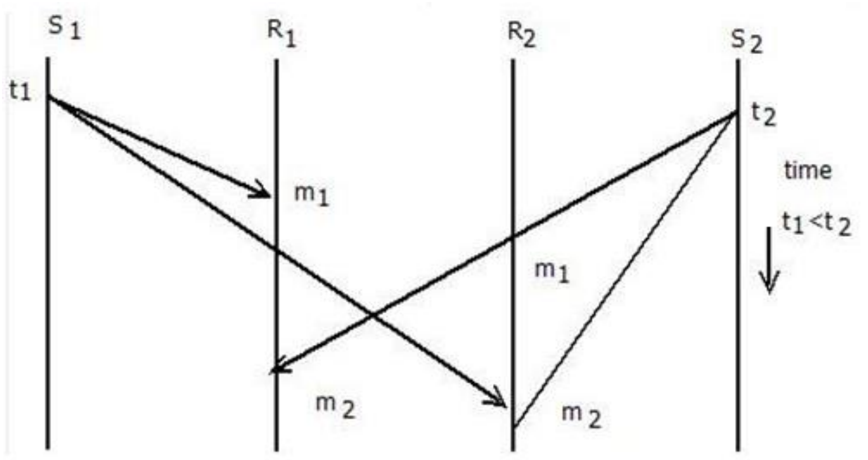
The management of a group needs an efficient and reliable multicast communication mechanism to allow clients obtain services from the group and ensure consistency among servers in the presence of failures. Consider the following two scenarios:-

- A client wishes to obtain a service which can be performed by any member of the group without affecting the state of the service.
- A client wishes to obtain a service which must be performed by each member of the group.

In the first case, the client can accept a response to its multicast from any member of the group as long as at least one responds. The communication system need only guarantee delivery of the multicast to a non-faulty process of the group on a best-effort basis. In the second case, the all-

or-none atomic delivery requirements requires that the multicast needs to be buffered until it is committed and subsequently delivered to the application process, and so incurs additional latency.

Absolute Ordering



- It ensures that all message delivered to all receiver in exact in which they are sent.
- Used to implement is to use global timestamps as message id.
- System is assumed to have clock synchronization and when sender message timestamps is taken as id of message and embedded in message.
- Kernel of receiver machine saves all incoming message in separate queue.
- A sliding window is used to periodically deliver message from receiver i.e. fixed time interval is selected as window size.
- Window size and properly chosen by considering maximum time for message to kernel from one m/c to other in n/w.

Conclusion:

Absolute ordering semantics allows the sender deliver all the messages to all the receiver processes in the exact order in which they were sent.

Experiment No: 4

Remote Method Invocation (RMI)

Aim:

Implement Remote Method Invocation (RMI) based application.

Brief Theory:

Java supports sockets to facilitate flexible communication between client and server application. However, it requires application level protocol to encode and decode messages which are cumbersome and error-prone. In such cases Remote Procedure Call (RPC) which alert the communication interface to the level of procedure call can be used. But RPC is communication mechanism between function and object. Hence to match the semantics of object invocation, remote Method Invocation (RMI) is used in distributed object systems. RMI is action of invoking a method of a remote interface on a remote object. In these systems, a stub object manages the invocation on a remote object.

RMI adds the following facilities to Java programming language:

1. Integrates distributed object model with semantics of Java objects.
2. Facilitates seamless remote invocation on objects.
3. Facilitates usage of applets on client side.
4. Transparency of invocation between local and remote objects.
5. Ease of communication between distributed applications.
6. Reliable communication between distributed applications.
7. Ease of use.

RMI applications comprise the client and server programs. The client application invokes the remote methods using this remote reference whereas the server application creates remote objects and facilitates their access through their references. Thus RMI aids in development of distributed object applications. These distributed object applications have to perform the following activities:

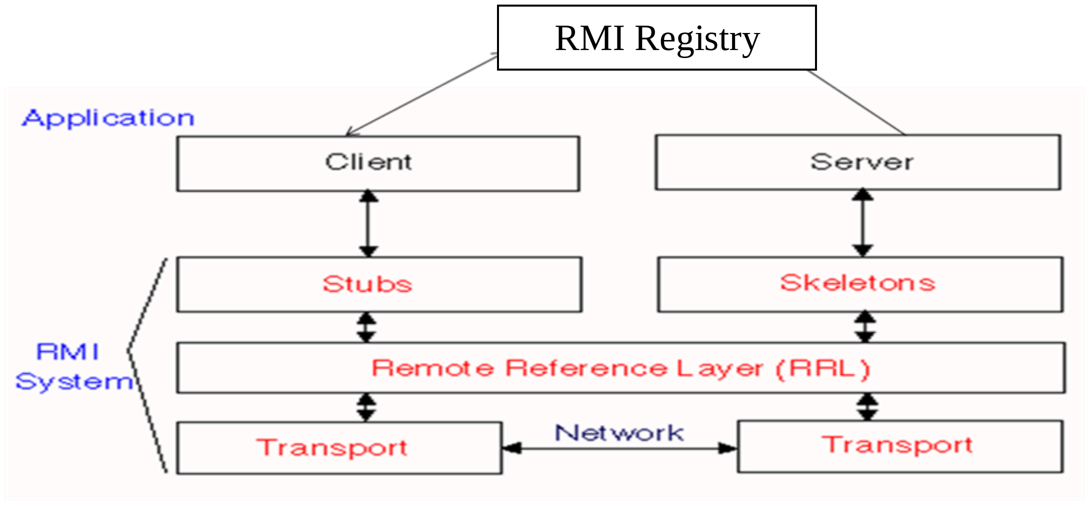


Fig. RMI mechanism

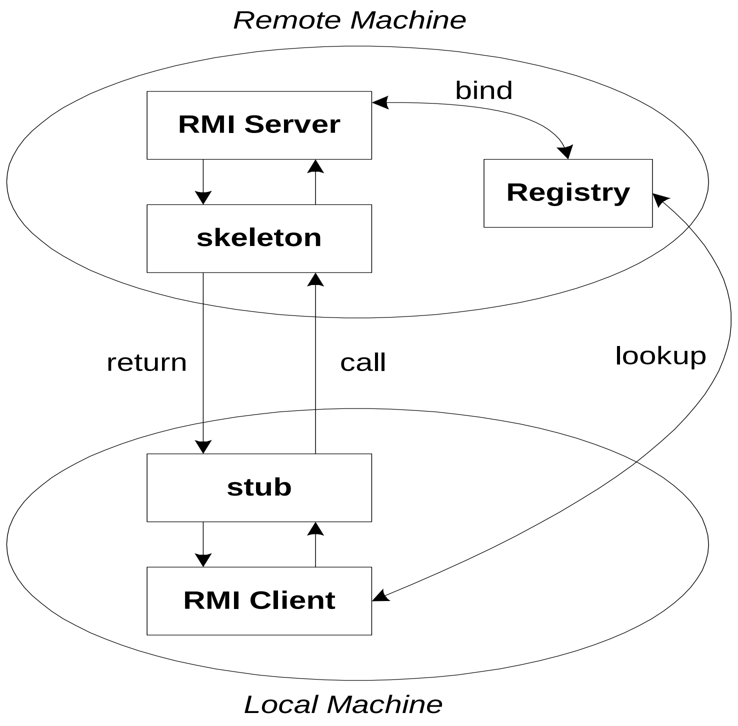
1. Registration and location of remote objects can be done using a simple naming facility called the RMIregistry. Server applications register in this registry and clients can access the objects using the registry. In another method for object referencing, remote OR is passed as a part of the normal operation.
2. Distributed objects can communicate transparently with the client using RMI.
3. RMI allows passing objects between remote applications. Hence it has necessary mechanism for loading an object's code as well as transmitting its data.

Steps to create RMI application:

1. Creating Interface.
2. Creating the implementation class.
3. Creating the server program.
4. Creating the client program to connects to the server object using Naming.lookup().
5. Compile these classes using javac compiler.
6. Generate stubs and skeletons. To create stub and skeleton files, the rmic compiler is run on the class files that have remote implementation.
   rmic implementation-class name
7. Start the RMI registry (using "start rmiregistry" command.) The registry by default runs on port 1099. To start the registry on a different port, specify the port number in the

command. For example, to start the registry on port 2001, it is specified as “start rmiregistry 2001”.

8. Start the server class.
9. Run the client class.



Conclusion: Remote Method Invocation is successfully implemented.

Experiment No: 5

Load balancing

Aim:

Demonstrate load balancing approach in distributed environment.

Brief Theory:

A load is balanced if no processes are idle.

In computing, **load balancing** distributes workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units or disk drives. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource. Using multiple components with load balancing instead of a single component may increase reliability and availability through redundancy. Load balancing usually involves dedicated software or hardware, such as a multilayer switch or a Domain Name System server process.

Load balancing is dividing the amount of work that a computer has to do between two or more computers so that more work gets done in the same amount of time and, in general, all users get served faster. Load balancing can be implemented with hardware, software, or a combination of both.

- **How?**
  - Partition the computation into units of work (tasks or jobs)
  - Assign tasks to different processors
- **Load Balancing Categories**
  - **Static:** load assigned before application runs
  - **Dynamic:** load assigned as applications run
    - Centralized: Tasks assigned by the master or root process
    - De-centralized: Tasks reassigned among slaves
  - **Semi-dynamic:** application periodically suspended and load balanced
- **Load Balancing Algorithms are:**
  - **Adaptive** if they adapt to system load levels using thresholds.
  - **Stable** if load balancing traffic is independent of load levels.

- **Symmetric** if both senders and receivers initiate action.
- **Effective** if load balancing overhead is minimal.

Conclusion:

Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource. Load balancing can be implemented with hardware, software, or a combination of both.

Experiment No: 6

Name Resolution protocol.

Aim: Implementation of Name Resolution protocol in distributed system.

Theory:

Names play a very important role in all computer systems. They are used to share resources, to uniquely identify entities, to refer to locations, and more. An important issue with naming is that a name can be resolved to the entity it refers to. Name resolution thus allows a process to access the named entity. To resolve names, it is necessary to implement a naming system. The difference between naming in distributed systems and non-distributed systems lies in the way naming systems are implemented. In a distributed system, the implementation of a naming system is itself often distributed across multiple machines.

Name resolution is a method of reconciling an IP address to a user friendly computer name. Originally networks used host files to resolve names to IP addresses. They came in the form of a text file that the computer accessed if name resolution was required. All the computers on the network and their IP address mappings had to be entered manually. The file was then copied to all the machines on the network. When a resource was required, by the user typing its name, the machine referred to the host file to find the IP address.

Name spaces offer a convenient mechanism for storing and retrieving information about entities by means of names. More generally, given a path name, it should be possible to look up any information stored in the node referred to by that name. The process of looking up a name is called name resolution.

The Domain Name System (DNS) is a distributed directory that resolves human-readable hostnames, such as www.dyn.com, into machine-readable IP addresses like 50.16.85.103. DNS is also a directory of crucial information about domain names, such as email servers (MX records) and sending verification (DKIM, SPF, DMARC), TXT record verification of domain ownership, and even SSH fingerprints (SSHFP).

Conclusion:

Name resolution protocol resolves human-readable hostnames, such as www.dyn.com, into machine-readable IP addresses like 50.16.85.103.

Experiment No: 7
Election Algorithm

Aim:

Demonstrate Bully election algorithm.

Brief Theory:

Many distributed algorithms require one process to act as a coordinator or, in general, perform some special role. We consider that it doesn't matter which process is elected; what is important is that one and only one process is chosen (we call this process the

Coordinator) and all processes agree on this decision.

We assume that each process has a unique number (identifier); in general, election algorithms attempt to locate the process with the highest number, among those which currently are up.

Election is typically started after a failure occurs. The detection of a failure (e.g. the crash of the current coordinator) is normally based on time-out. A process that gets no response for a period of time suspects a failure and initiates an election process.

An election process is typically performed in two phases:

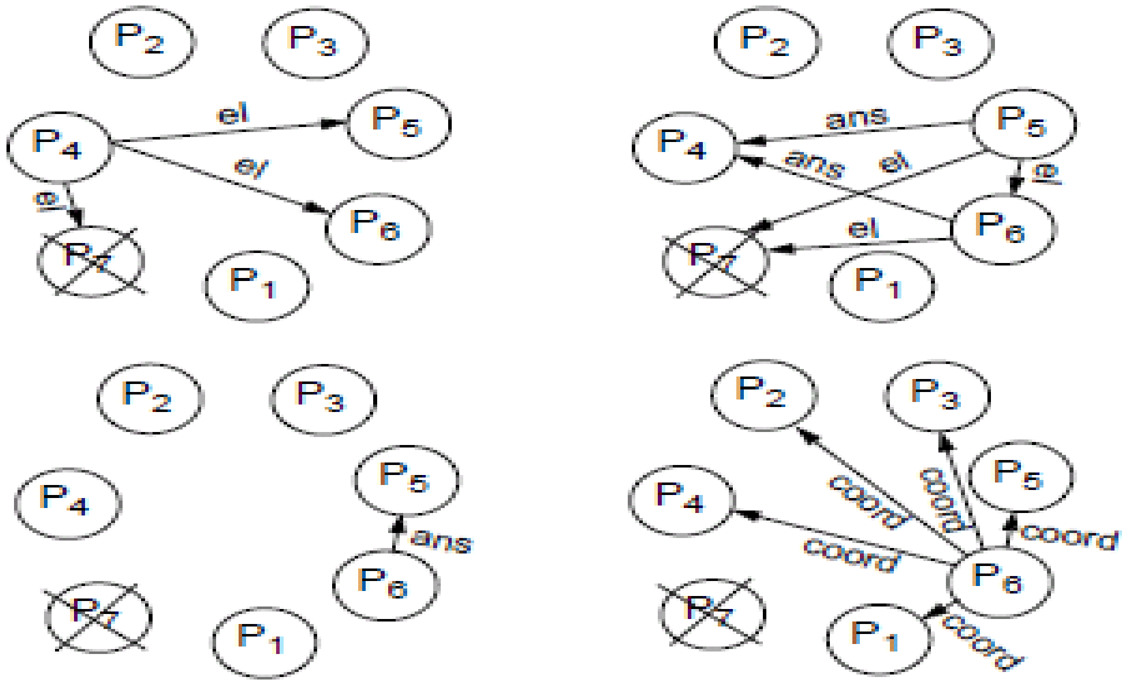
1. Select a leader with the highest priority.
2. Inform all processes about the winner

The Bully Algorithm :

A process has to know the identifier of all other processes (it doesn't know, however, which one is still up); the process with the highest identifier, among those which are up, is selected. Any process could fail during the election procedure.

When a process *Pi* detects a failure and a coordinator has to be elected, it sends an *election message* to all the processes with a higher identifier and then waits for an *answer message*:

- If no response arrives within a time limit, *Pi* becomes the coordinator (all processes with higher identifier are down) it broadcasts a *coordinator message* to all processes to let them know.
- If an *answer message* arrives, *Pi* knows that another process has to become the coordinator it waits in order to receive the *coordinator message*. If this message fails to arrive within a time limit (which means that a potential coordinator crashed after sending the *answer message*) *Pi* resends the *election message*.
- When receiving an *election message* from *Pi*, a process *Pj* replys with an *answer message* to *Pi* and then starts an election procedure itself, unless it has already started one, it sends an *election message* to all processes with higher identifier.
- Finally all processes get an *answer message*, except the one which becomes the coordinator.
- If*P6* crashes before sending the coordinator message, *P4* and *P5* restart the election process.

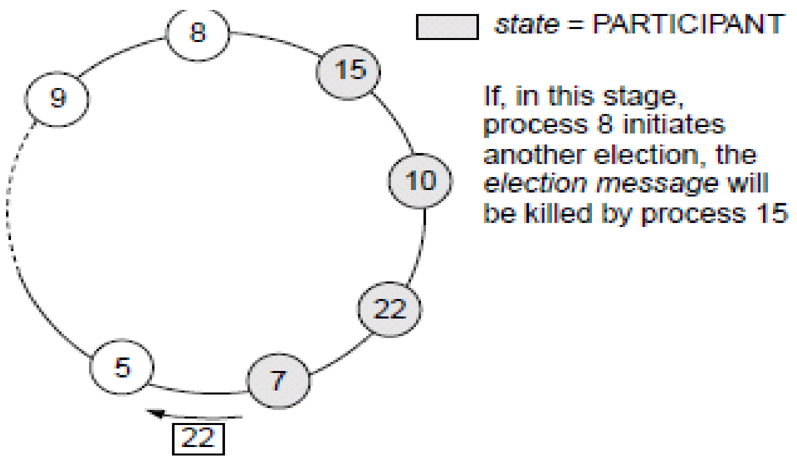


- The best case: the process with the second highest identifier notices the coordinator's failure. It can immediately select itself and then send *n*-2 coordinator messages.

- The worst case: the process with the lowest identifier initiates the election; it sends *n*-1 election messages to processes which themselves initiate each one an election ⇒O(*n2*) messages.

The Ring-Based Algorithm

- We assume that the processes are arranged in a logical ring; each process knows the address of one other process, which is its neighbour in the clockwise direction.
- The algorithm elects a single coordinator, which is the process with the highest identifier.
- Election is started by a process which has noticed that the current coordinator has failed. The process places its identifier in an *election message* that is passed to the following process.
- When a process receives an *election message* it compares the identifier in the message with its own. If the arrived identifier is greater, it forwards the received *election message* to its neighbour; if the arrived identifier is smaller it substitutes its own identifier in the *election message* before forwarding it.
- If the received identifier is that of the receiver itself, this will be the coordinator. The new coordinator sends an *elected message* through the ring.


8
15
9
10
22
5
7
22
▭ *state* = PARTICIPANT
If, in this stage, process 8 initiates another election, the *election message* will be killed by process 15

With one single election started:

- On average: *n*/2 (election) messages needed to reach maximal node; *n* (election) messages to return to maximal node; *n* messages to rotate *elected message*.

Number of messages: 2*n* + *n*/2.

- Worst case: *n*-1 messages needed to reach maximal node; Number of messages: 3*n* - 1.
- The ring algorithm is more efficient on average then the bully algorithm

Conclusion : We have simulated the working of election algorithms in distributed system.

Experiment No: 8

Clock Synchronization

Aim:

Demonstrate the Clock Synchronization using Berkeley algorithm.

Brief Theory:

Synchronization is important if we want to

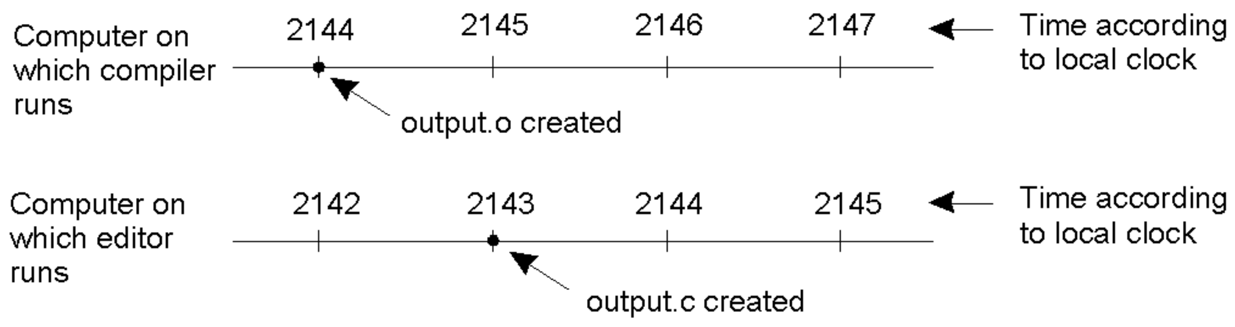
- *control access* to a single, shared resource
- agree on the *ordering of events*

Synchronization in Distributed Systems is much more difficult than in uniprocessor systems

Lack of Global Time in DS

- It is impossible to guarantee that physical clocks run at the same frequency
- Lack of global time, can cause problems
- Example: UNIX make
  - Edit output.c at a client
  - output.o is at a server (compile at server)
  - Client machine clock can be lagging behind the server machine clock

Lack of Global Time – Example


Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created

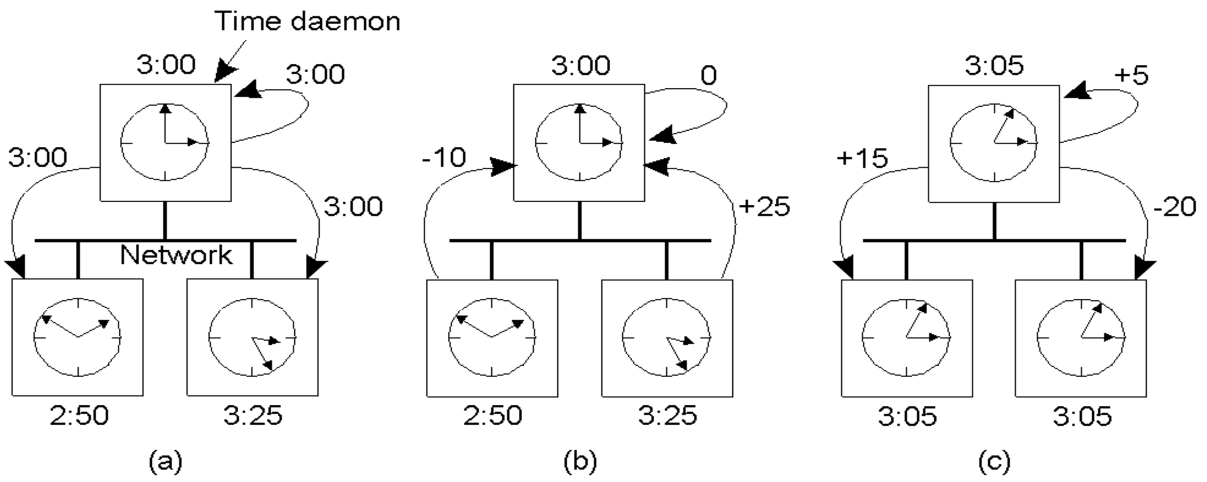
- Distributed systems often require roughly consistent notions of time
- Usually the requirement isn't that time is accurate (UTC), only that it is synchronized
- However, synchronizing machines with UTC automatically synchronizes all machines with each other

- **Two well-known methods:**
  - Cristian's algorithm (UTC time-server based)
  - Berkeley algorithm (no UTC signal, but master)

Clock Synchronization (Cristian)

- Client polls time server (which has external UTC source)
- Gets time from server
- Adjusts received time by adding estimate of one-way delay
  - Estimates travel time as 1/2 of RTT
  - Adds this to server time
  - Errors introduced not by delays, but by asymmetry in delays (path to server and path from server)

Clock Synchronization (Berkeley)



a) The time daemon asks all the other machines for their clock values
b) The machines answer

c) The time daemon tells everyone how to adjust their clock

Logical Clocks

- Most algorithms don't require tightly synchronized clocks, but they often require a common notion of causality
- That is, events can be ordered arbitrarily, as long as causality isn't violated
- For example, it doesn't matter whether I updated my password in Japan before or after someone saved a file in Chile, as long as no messages or other interactions occurred between the two systems
- Lamport captured this notion of causality

Lamport Timestamps

- When message arrives, if process time is less than timestamp s, then jump process time to s+1
- Clock must tick once between every two events
- If $A \rightarrow B$ then must have $L(A) < L(B)$
  - logical clock ordering never violates causaility
  - If $L(A) < L(B)$, it does NOT follow that $A \rightarrow B$
  - Lamport clocks leave some causal ambiguity

Vector Timestamps Definition

- $V_I[I]$: number of events occurred in process I
  - Not using Lamport's rule of jumping clocks ahead!
- $V_I[J] = K$: process I knows that K events have occurred at process J
- All messages carry vectors
- When J receives vector v, for each K it sets
  $V_J[K] = v[K]$ if it is larger than its current value
- It then updates $V_J[J]$ by one (to reflect recv event)

Conclusion: Thus the physical clock synchronization algorithm was simulated.

Experiment No: 9

Mutual Exclusion

Aim:

Demonstrate the working of Lamport's algorithm for mutual exclusion.

Brief Theory:

Mutual exclusion: Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner. Only one process is allowed to execute the critical section (CS) at any given time. In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion.

Distributed mutual exclusion algorithms must deal with unpredictable message delays and incomplete knowledge of the system state.

Three basic approaches for distributed mutual exclusion:

- Token based approach
- Non-token based approach
- Quorum based approach

Token-based approach:

A unique token is shared among the sites. A site is allowed to enter its CS if it possesses the token. Mutual exclusion is ensured because the token is unique.

Non-token based approach:

Two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next.

Quorum based approach:

Each site requests permission to execute the CS from a subset of sites (called a quorum).

Any two quorums contain a common site.

This common site is responsible to make sure that only one request executes the CS at any time.A condition in which there is a set of processes, only one of which is able to access a given resource or perform a given function at any time.

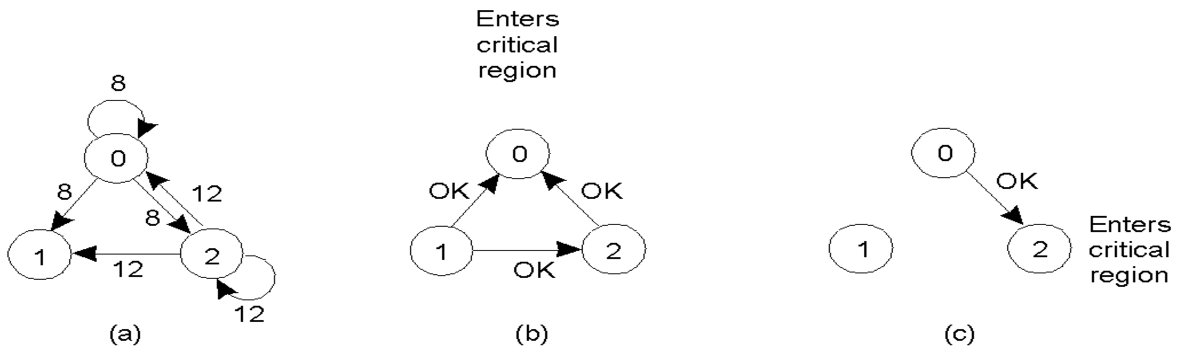
Systems involving multiple processes are often most easily programmed using critical regions. When a process has to read or update certain shared data structures, it first enters a critical region to achieve mutual exclusion and ensure that no other process will use the shared data structures at the same time. In singleprocessor systems, critical regions are protected using semaphores, monitors, and similar constructs. We will now look at a few examples of how critical regions and mutual exclusion can be implemented in distributed systems.

Distributed Mutual Exclusion:

- Centralized Algorithm
- Token Ring Algorithm
- Distributed Algorithm
- Decentralized Algorithm

Distributed Algorithms :

- Contention-based Mutual Exclusion
    - Timestamp Prioritized Schemes
    - Voting Schemes
- Token-based Mutual Exclusion
    - Ring Structure
    - Tree Structure
    - Broadcast Structure



A> Two processes want to enter the same critical region .

B> Process 0 has the lowest timestamp, so it wins.

C> When process 0 is done, it sends an OK also, so 2 can now enter the critical region[2]

Lamport's "logical clock" is used to generate timestamps. These are the general properties for the method:

The general mechanism is that a process P[i] has to send a REQUEST (with ID and time stamp) to all other processes.

- When a process P[j] receives such a request, it sends a REPLY back.
- When responses are received from all processes, then P[i] can enter its Critical Section.
- When P[i] exits its critical section, the process sends RELEASE messages to all its deferred requests. The total message count is 3*(N-1), where N is the number of cooperating processes.
- Number of network messages; 2*(N-1)
- Synchronization Delays: One message propagation delay
- Mutual exclusion: Site Pi enters its critical section only after receiving all reply messages.
- Progress: Upon exiting the critical section, Pi sends all deferred reply messages.

Conclusion: Distributed Mutual Exclusion has been implemented successfully.

Experiment No. 10

Aim:
Demonstrate CORBA application using JAVA.

Theory:
The Common Object Request Broker Architecture (CORBA) is a standard developed by the Object Management Group (OMG) to provide interoperability among distributed objects. CORBA is the world's leading middleware solution enabling the exchange of information, independent of hardware platforms, programming languages, and operating systems. CORBA is essentially a design specification for an Object Request Broker (ORB), where an ORB provides the mechanism required for distributed objects to communicate with one another, whether locally or on remote devices, written in different languages, or at different locations on a network.
In CORBA Programming we have to create:

1. Simple Remote Object.
2. Server to instantiate (create) and bind a remote object.
3. Client to invoke remotely an object

CORBA Architecture:

The major components that make up the CORBA architecture include the:

Interface Definition Language (IDL), which is how CORBA interfaces are defined,
Object Request Broker (ORB), which is responsible for all interactions between remote objects and the applications that use them,

The Portable Object Adaptor (POA), which is responsible for object activation/deactivation, mapping object ids to actual object implementations.

Naming Service, a standard service in CORBA that lets remote clients find remote objects on the networks, and

Inter-ORB Protocol (IIOP).
This figure shows how a one-method distributed object is shared between a CORBA client and server to implement the classic "Hello World" application.

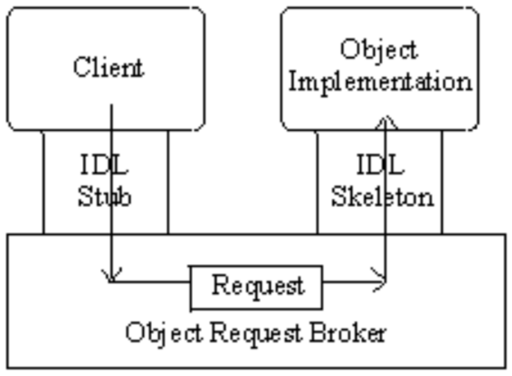


Figure 1: A request passing from client to object implementation

A one-method distributed object shared between a CORBA client and server.

Conclusion:

The CORBA principle simplifies the design of distributed object computing applications.