ASSIGNMENT NO: 1

Title: Socket Programming

Aim: To develop any distributed application through implementing client-server communication programs based on Java Sockets.

Objectives:

1. To study interprocess communication
2. To learn client server communication using socket

Tools / Environment:

Java Programming Environment, rmiregistry, jdk 1.8, Eclipse IDE.

Related Theory:

Socket: In distributed computing, network communication is one of the essential parts of any system, and the socket is the endpoint of every instance of network communication. In Java communication, it is the most critical and basic object involved.

A socket is a handle that a local program can pass to the networking API to connect to another machine. It can be defined as the terminal of a communication link through which two programs /processes/threads running on the network can communicate with each other. The TCP layer can easily identify the application location and access information through the port number assigned to the respective sockets.

During an instance of communication, a client program creates a socket at its end and tries to connect it to the socket on the server. When the connection is made, the server creates a socket at its end and then server and client communication is established.

Designing the solution:

The **java.net** package provides classes to facilitate the functionalities required for networking. The **socket** class programmed through Java using this package has the capacity of being independent of the platform of execution; also, it abstracts the calls specific to the operating system on which it is invoked from other Java interfaces. The **ServerSocket** class offers to observe connection invocations, and it accepts such invocations from different clients through another socket. High-level wrapper classes, such as **URLConnection** and **URLEncoder**, are more appropriate. If you want to establish a connection to the Web using a URL, then these classes will use the socket internally.

The **java.net** package provides support for the two common network protocols −

- **TCP** − TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.

- **UDP** − UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

Socket programming for TCP:

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

The server instantiates a ServerSocket object, denoting which port number communication is to occur on.

The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.

After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.

The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.

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 two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets.

Socket programming for UDP:

UDP is used only when the entire information can be bundled into a single packet and there is no dependency on the other packet. Therefore, the usage of UDP is quite limited, whereas TCP is widely used in IP applications. UDP sockets are used where limited bandwidth is available, and the overhead associated with resending packets is not acceptable.

To connect using a UDP socket on a specific port, use the following code:

```
DatagramSocket udpSock = new DatagramSocket(3000);
```

A datagram is a self-contained, independent message whose time of arrival, confirmation of arrival over the network, and content cannot be guaranteed. atagramPacket objects are used to send data over DatagramSocket. Every DatagramPacket object consists of a data buffer, a remote host to whom the data needs to be sent, and a port number on which the remote agent would be listened.

Implementing the solution:

Socket Programming for TCP:

Client Programming:

1. **Establish a Socket Connection:** The **java.net Socket** class represents a Socket. To open a socket:

```
Socket socket = new Socket("127.0.0.1", 5000)
```

2. **Communication:** To communicate over a socket connection, streams are used to both input and output the data.

3. **Closing the connection:** The socket connection is closed explicitly once the message to server is sent.

Server Programming:

1. **Establish a Socket Connection:** To write a server application two sockets are needed. A ServerSocket which waits for the client requests (when a client makes a new Socket()). A plain old Socket socket to use for communication with the client.

2. **Communication:** getOutputStream() method is used to send the output through the socket.

3. **Close the Connection:** After finishing, it is important to close the connection by closing the socket as well as input/output streams.

Compilation and Executing the solution:

If you're using Eclipse :

1. Compile both of them on two different terminals or tabs

2. Run the Server program first

3. Then run the Client program

4. Type messages in the Client Window which will be received and showed by the Server Window simultaneously if you are developing echo server application.

5. Close the socket connection by typing something like "Exit".

Conclusion:

In this assignment, the students learned about client-server communication through different protocols and sockets. They also learned about Java support through the socket API for TCP and UDP programming.

Outcome:

1. Students learned about interprocess communication in DS
2. Students develop application through implementing client-server communication programs based on Java Sockets

FAQ:

1. What is interprocess communication?
2. What is socket?
3. Difference between TCP and UDP socket communication?
4. What is shared memory programming?
5. What is port? State application of port.

ASSIGNMENT NO. 2

Title: Remote Method Invocation

Aim: To implement multi-threaded client/server Process communication using RMI.

Objectives:

1. To understand the basics of multi-threaded client/server Process communication.
2. To develop interprocess communication application using RMI.

Tools / Environment:

Java Programming Environment, jdk 1.8, rmiregistry

Related Theory:

RMI provides communication between java applications that are deployed on different servers and connected remotely using objects called **stub** and **skeleton**. This communication architecture makes a distributed application seem like a group of objects communicating across a remote connection. These objects are encapsulated by exposing an interface, which helps access the private state and behavior of an object through its methods.

The following diagram shows how RMI happens between the RMI client and RMI server with the help of the

RMI registry:

RMI REGISTRY is a remote object registry, a Bootstrap naming service, that is used by **RMI SERVER** on the same host to bind remote objects to names. Clients on local and remote hosts then look up the remote objects and make remote method invocations.

Key terminologies of RMI:

The following are some of the important terminologies used in a Remote Method Invocation. **Remote object**: This is an object in a specific JVM whose methods are exposed so they could be invoked by another program deployed on a different JVM.

Remote interface: This is a Java interface that defines the methods that exist in a remote object. A remote object can implement more than one remote interface to adopt multiple remote interface behaviors.

RMI: This is a way of invoking a remote object's methods with the help of a remote interface. It can be carried with a syntax that is similar to the local method invocation.

Stub: This is a Java object that acts as an entry point for the client object to route any outgoing requests. It exists on the client JVM and represents the handle to the remote object.

If any object invokes a method on the stub object, the stub establishes RMI by following these steps:

1. It initiates a connection to the remote machine JVM.
2. It marshals (write and transmit) the parameters passed to it via the remote JVM.
3. It waits for a response from the remote object and unmarshals (read) the returned value or exception, then it responds to the caller with that value or exception.

Skeleton: This is an object that behaves like a gateway on the server side. It acts as a remote object with which the client objects interact through the stub. This means that any requests coming from the remote client are routed through it. If the skeleton receives a request, it establishes RMI through these steps:

1. It reads the parameter sent to the remote method.
2. It invokes the actual remote object method.
3. It marshals (writes and transmits) the result back to the caller (stub).

The following diagram demonstrates RMI communication with stub and skeleton involved:

Designing the Solution:

The essential steps that need to be followed to develop a distributed application with RMI are as follows:

1. Design and implement a component that should not only be involved in the distributed application, but also the local components.

2. Ensure that the components that participate in the RMI calls are accessible across networks.

3. Establish a network connection between applications that need to interact using the RMI.

Remote interface definition: The purpose of defining a remote interface is to declare the methods that should be available for invocation by a remote client.

Programming the interface instead of programming the component implementation is an essential design principle adopted by all modern Java frameworks, including Spring. In the same pattern, the definition of a remote interface takes importance in RMI design as well.

2. **Remote object implementation**: Java allows a class to implement more than one interface at a time. This helps remote objects implement one or more remote interfaces. The remote object class may have to implement other local interfaces and methods that it is responsible for. Avoid adding complexity to this scenario, in terms of how the arguments or return parameter values of such component methods should be written.

3. **Remote client implementation**: Client objects that interact with remote server objects can be written once the remote interfaces are carefully defined even after the remote objects are deployed.

Let's design a project that can sit on a server. After that different client projects interact with this project to pass the parameters and get the computation on the remote object execute and return the result to the client components.

Implementing the solution:

Consider building an application to perform diverse mathematical operations.

The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.

1. **Creating remote interface, implement remote interface, server-side and client-side program and Compile the code.**

This application uses four source files. The first file, **AddServerIntf.java**, defines the remote interface that is provided by the server. It contains one method that accepts two **double** arguments and returns their sum. All remote interfaces must extend the **Remote** interface, which is part of **java.rmi**. **Remote** defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a **RemoteException**.

The second source file, **AddServerImpl.java**, implements the remote interface. The implementation of the **add()** method is straightforward. All remote objects must extend **UnicastRemoteObject**, which provides functionality that is needed to make objects available from remote machines.

The third source file, **AddServer.java**, contains the main program for the server machine. Its primary function is **to update the RMI registry on that machine**. This is done by using the **rebind()** method of the **Naming** class (found in **java.rmi**). That method associates a name with an object reference. The first argument to the **rebind()** method is a string that names the

server as "AddServer". Its second argument is a reference to an instance of **AddServerImpl**.

The fourth source file, **AddClient.java**, implements the client side of this distributed application. **AddClient.java** requires three command-line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URL uses the **rmi** protocol. The string includes the IP address or name of the server and the string "AddServer". The program then invokes the **lookup( )** method of the **Naming** class. This method accepts one argument, the **rmi** URL, and returns a reference to an object of type **AddServerIntf**. All remote method invocations can then be directed to this object.

The program continues by displaying its arguments and then invokes the remote **add()** method.

The sum is returned from this method and is then printed.

Use **javac** to compile the four source files that are created.

2. Generate a Stub

Before using client and server, the necessary stub must be generated. In the context of RMI, a *stub* is a Java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine.

All of this information must be sent to the remote machine. That is, an object passed as an argument to a remote method call must be serialized and sent to the remote machine. If a response must be returned to the client, the process works in reverse. **The serialization and deserialization facilities are also used if objects are returned to a client.**

To generate a stub the command is RMIcompiler is invoked as follows:

rmic AddServerImpl.

This command generates the file **AddServerImpl_Stub.class**.

3. Install Files on the Client and Server Machines

Copy **AddClient.class**, **AddServerImpl_Stub.class**, **AddServerIntf.class** to a directory on the client machine.

Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_ Stub.class**, and **AddServer.class** to a directory on the server machine.

4. Start the RMI Registry on the Server Machine

Java provides a program called **rmiregistry**, which executes on the server machine. It maps names to object references. Start the RMI Registry from the command line, as shown here:

start rmiregistry

5. Start the Server

The server code is started from the command line, as shown here:

java AddServer

The **AddServer** code instantiates **AddServerImpl** and registers that object with the name "AddServer".

6. Start the Client

The **AddClient** software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. You may invoke it from the command line by using one of the two formats shown here:

java AddClient 192.168.13.14 7 8

Conclusion:

Remote Method Invocation (RMI) allows you to build Java applications that are distributed among several machines. Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This is an important feature, because it allows you to build distributed applications.

Outcome:

1. Students learn basics of multi-threaded client server process communication.
2. Students develop interprocess communication application using JAVA RMI.

FAQ:

1. What is Heterogeneity?
2. Example of is marshaling and unmarshalling?
3. Explain RMI with diagram.
4. What is binding?
5. What is role of RMI registry? why we start RMI registry first.
6. What is use of "UnicastRemoteObject","lookup(),rebind().
7. What is stub and skeleton?
8. What is difference between Exception and RemoteException

ASSIGNMENT NO. 3

Title: Common Object Request Broker Architecture (CORBA)

Aim: To develop any distributed application with CORBA program using JAVA IDL.

Objective:

1. To understand the basics steps for implementation of CORBA.
2. To develop any distributed application using CORBA to demonstrate object brokering.

Tools / Environment:

Java Programming Environment, JDK 1.8

Related Theory:

Common Object Request Broker Architecture (CORBA):

CORBA is an acronym for Common Object Request Broker Architecture. It is an open source, vendor-independent architecture and infrastructure developed by the **Object Management Group** (**OMG**) to integrate enterprise applications across a distributed network. CORBA specifications provide guidelines for such integration applications, based on the way they want to interact, irrespective of the technology; hence, all kinds of technologies can implement these standards using their own technical implementations.

When two applications/systems in a distributed environment interact with each other, there are quite a few unknowns between those applications/systems, including the technology they are developed in (such as Java/ PHP/ .NET), the base operating system they are running on (such as Windows/Linux), or system configuration (such as memory allocation). **They communicate mostly with the help of each other's network address or through a naming service.** Due to this, these applications end up with quite a few issues in integration, including content (message) mapping mismatches.

An application developed based on CORBA standards with standard **Internet Inter-ORB Protocol** (**IIOP**), irrespective of the vendor that develops it, should be able to smoothly integrate and operate with another application developed based on CORBA standards through the same or different vendor.

Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example. All applications related to, say, "HR&Benefits" maintain an object

model with details of the organization, employees with demographic information, benefits, payroll, and deductions. They are only different in the way they handle the details, based on the country and region they are operating for. For each object type, similar to the HR&Benefits systems, we can define an interface using the **Interface Definition Language** (**OMG IDL**).

The contract between these applications is defined in terms of an interface for the server objects that the clients can call. This IDL interface is used by each client to indicate when they should call any particular method to marshal (read and send the arguments).

The target object is going to use the same interface definition when it receives the request from the client to unmarshal (read the arguments) in order to execute the method that was requested by the client operation. Again, during response handling, the interface definition is helpful to marshal (send from the server) and unmarshal (receive and read the response) arguments on the client side once received.

The IDL interface is a design concept that works with multiple programming languages including C, C++, Java, Ruby, Python, and IDLscript. This is close to writing a program to an interface, a concept we have been discussing that most recent programming languages and frameworks, such as Spring. The interface has to be defined clearly for each object. The systems encapsulate the actual implementation along with their respective data handling and processing, and only the methods are available to the rest of the world through the interface. Hence, the clients are forced to develop their invocation logic for the IDL interface exposed by the application they want to connect to with the method parameters (input and output) advised by the interface operation.

The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons, The objects are written (on the right) and a client for it (on the left), as represented in the diagram. The client and server use stubs and skeletons as proxies, respectively. The IDL interface follows a strict definition, and even though the client and server are implemented in different technologies, they should integrate smoothly with the interface definition strictly implemented.

In CORBA, each object instance acquires an object reference for itself with the electronic token identifier. Client invocations are going to use these object references that have the ability to figure out which ORB instance they are supposed to interact with. The stub and skeleton represent the client and server, respectively, to their counterparts. They help establish this communication through ORB and pass the arguments to the right method and its instance during the invocation.

zInter-ORB communication

The following diagram shows how remote invocation works for inter-ORB communication. It shows that the clients that interacted have created **IDL Stub** and **IDL Skeleton** based on **Object Request Broker** and communicated through **IIOP Protocol**.

To invoke the remote object instance, the client can get its object reference using a naming service. Replacing the object reference with the remote object reference, the client can make the invocation of the remote method with the same syntax as the local object method invocation. ORB keeps the responsibility of recognizing the remote object reference based on the client object invocation through a naming service and routes it accordingly.

Java Support for CORBA

CORBA complements the Java™ platform by providing a distributed object framework, services to support that framework, and interoperability with other languages. The Java platform complements CORBA by providing a portable, highly productive implementation environment, and a very robust platform. By combining the Java platform with CORBA and other key

enterprise technologies, the Java Platform is the ultimate platform for distributed technology solutions.

CORBA standards provide the proven, interoperable infrastructure to the Java platform. IIOP (Internet Inter-ORB Protocol) manages the communication between the object components that power the system. The Java platform provides a portable object infrastructure that works on every major operating system. CORBA provides the network transparency, Java provides the implementation transparency. **An *Object Request Broker (ORB)* is part of the Java Platform.**

The ORB is a runtime component that can be used for distributed computing using IIOP communication. Java IDL is a Java API for interoperability and integration with CORBA. Java IDL included both a Java-based ORB, which supported IIOP, and the **IDL-to-Java**

compiler, for generating client-side stubs and server-side code skeletons. J2SE v.1.4 includes an

Object Request Broker Daemon (ORBD), which is used to enable clients to transparently locate and invoke persistent objects on servers in the CORBA environment.

When using the **IDL programming model,** the interface is everything! It defines the points of entry that can be called from a remote process, such as the types of arguments the called procedure will accept, or the value/output parameter of information returned. Using IDL, the programmer can make the entry points and data types that pass between communicating processes act like a standard language.

CORBA is a language-neutral system in which the argument values or return values are limited to what can be represented in the involved implementation languages. In CORBA, object orientation is limited only to objects that can be passed by reference (the object code itself cannot be passed from machine-to-machine) or are predefined in the overall framework. Passed and returned types must be those declared in the interface.

With RMI, the interface and the implementation language are described in the same language, so you don't have to worry about mapping from one to the other. Language-level objects (the code itself) can be passed from one process to the next. Values can be returned by their actual type, not the declared type. Or, you can compile the interfaces to generate IIOP stubs and skeletons which allow your objects to be accessible from other CORBA-compliant languages.

The IDL Programming Model:

The IDL programming model, known as Java™ IDL, consists of both the Java CORBA ORB and the idlj compiler that maps the IDL to Java bindings that use the Java CORBA ORB, as well as a set of APIs, which can be explored by selecting the org.omg prefix from the Package section of the API index.

Java IDL adds CORBA (Common Object Request Broker Architecture) capability to the Java platform, providing standards-based interoperability and connectivity. Runtime components include a Java ORB for distributed computing using IIOP communication.

To use the IDL programming model, define remote interfaces using OMG Interface Definition Language (IDL), then compile the interfaces using idlj compiler. When you run the idlj compiler over your interface definition file, it generates the Java version of the interface, as well as the class code files for the stubs and skeletons that enable applications to hook into the ORB.

Portable Object Adapter (POA) : An *object adapter* is the mechanism that connects a request using an object reference with the proper code to service that request. The Portable Object Adapter, or POA, is a particular type of object adapter that is defined by the CORBA specification. The POA is designed to meet the following goals:

- Allow programmers to construct object implementations that are portable between different ORB products.
- Provide support for objects with persistent identities.

Designing the solution:

Here the design of how to create a complete CORBA (Common Object Request Broker Architecture) application using IDL (Interface Definition Language) to define interfaces and Java IDL compiler to generate stubs and skeletons. You can also create CORBA application by defining the interfaces in the Java programming language.

The server-side implementation generated by the idlj compiler is the *Portable Servant Inheritance Model*, also known as the POA(Portable Object Adapter) model. This document presents a sample application created using the default behavior of the idlj compiler, which uses a POA server-side model.

1. Creating CORBA Objects using Java IDL:

1.1. In order to distribute a Java object over the network using CORBA, one has to define it's own CORBA-enabled interface and it implementation. This involves doing the following:

- Writing an interface in the CORBA Interface Definition Language
- Generating a Java base interface, plus a Java stub and skeleton class, using an IDL-to-Java compiler
- Writing a server-side implementation of the Java interface in Java

Interfaces in IDL are declared much like interfaces in Java.

1.2. Modules

Modules are declared in IDL using the module keyword, followed by a name for the module and an opening brace that starts the module scope. Everything defined within the scope of this module (interfaces, constants, other modules) falls within the module and is referenced in other IDL modules using the syntax *modulename*::x. e.g.

```
//  IDL
module jen {

   module corba {

      interface NeatExample ...

   };

};
```

1.3. Interfaces

The declaration of an interface includes an interface header and an interface body. The header specifies the name of the interface and the interfaces it inherits from (if any). Here is an IDL interface header:

```
interface PrintServer : Server { ...
```

This header starts the declaration of an interface called PrintServer that inherits all the methods and data members from the Server interface.

1.4 Data members and methods

The interface body declares all the data members (or attributes) and methods of an interface. Data members are declared using the attribute keyword. At a minimum, the declaration includes a name and a type.

```
readonly attribute string myString;
```

The method can be declared by specifying its name, return type, and parameters, at a minimum.

```
string parseString(in string buffer);
```

This declares a method called parseString() that accepts a single string argument and returns a string value.

1.5 A complete IDL example

Now let's tie all these basic elements together. Here's a complete IDL example that declares a module within another module, which itself contains several interfaces:

```
module OS {
   module services {
      interface Server {
         readonly attribute string serverName;
         boolean init(in string sName);
      };

      interface Printable {
         boolean print(in string header);
      };

      interface PrintServer : Server {

         boolean printThis(in Printable p);
      };
   };
};
```

The first interface, Server, has a single read-only string attribute and an init() method that accepts a string and returns a boolean. The Printable interface has a single print()method that accepts a string header. Finally, the PrintServer interface extends the Server interface and adds a printThis() method that accepts a Printable object and returns a boolean. In all cases, we've declared the method arguments as input-only (i.e., pass-by-value), using the in keyword.

2. Turning IDL Into Java

Once the remote interfaces in IDL are described, you need to generate Java classes that act as a starting point for implementing those remote interfaces in Java using an IDL-to-Java

compiler. Every standard IDL-to-Java compiler generates the following 3 Java classes from an IDL interface:

- A Java interface with the same name as the IDL interface. This can act as the basis for a Java implementation of the interface (but you have to write it, since IDL doesn't provide any details about method implementations).

- A holder class whose name is the name of the IDL interface with "Holder" appended to it

  (e.g., ServerHolder). This class is used when objects with this interface are used as out or inout arguments in remote CORBA methods. Instead of being passed directly into the remote method, the object is wrapped with its holder before being passed. When a remote method has parameters that are declared as out or inout, the method has to be able to update the argument it is passed and return the updated value. The only way to guarantee this, even for primitive Java data types, is to force out and inout arguments to be wrapped in Java holder classes, which are filled with the output value of the argument when the method returns.

The idltoj tool generate 2 other classes:

- **A client stub class,** called _interface-nameStub, that acts as a client-side implementation of the interface and knows how to convert method requests into ORB requests that are forwarded to the actual remote object. The stub class for an interface named Server is called _ServerStub.

- **A server skeleton class,** called _interface-nameImplBase, that is a base class for a server-side implementation of the interface. The base class can accept requests for the object from the ORB and channel return values back through the ORB to the remote client. The skeleton class for an interface named Server is called _ServerImplBase.

So, in addition to generating a Java mapping of the IDL interface and some helper classes for the Java interface, the *idltoj* compiler also creates subclasses that act as an interface between a CORBA client and the ORB and between the server-side implementation and the ORB.

This creates the five Java classes: a Java version of the interface, a helper class, a holder class, a client stub, and a server skeleton.

Conclusion:

CORBA provides the network transparency, Java provides the implementation transparency. CORBA complements the Java™ platform by providing a distributed object framework, services to support that framework, and interoperability with other languages. The Java platform complements CORBA by providing a portable, highly productive implementation environment.

The combination of Java and CORBA allows you to build more scalable and more capable applications than can be built using the JDK alone.

Outcome:

1. Students understand architecture and basics steps for implementation of CORBA
2. Students develop distributed application using CORBA to demonstrate object brokering for string and arithmetic operations.

FAQ :

1. What is CORBA?
2. How CORBA works?
3. Does it synchronous/Asynchronous application?
4. What is ORB?
5. What is IDL interface?
6. What is Object Request Broker Daemon (ORBD).
7. What is middleware.
8. List the use of middleware.
9. List the applications of CORBA.

ASSIGNMENT NO. 4

Title: Message Passing Interface (MPI)

Aim: Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors

Objective:

1. To introduce you to the fundamentals of MPI.
2. To understand widely used standard for writing message passing programs.

Tools / Environment:

Java Programming Environment, JDK1.8 or higher, MPI Library (mpi.jar), MPJ Express (mpj.jar)

Related Theory:

Message passing is a popularly renowned mechanism to implement parallelism in applications; it is also called MPI. The MPI interface for Java has a technique for identifying the user and helping in lower startup overhead. It also helps in collective communication and could be executed on both **shared memory and distributed systems**. MPJ is a familiar Java API for MPI implementation. mpiJava is the near flexible Java binding for MPJ standards.

Currently developers can produce more efficient and effective parallel applications using message passing.

A basic prerequisite for message passing is a good communication API. Java comes with various ready-made packages for communication, notably an interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. The parallel computing world is mainly concerned with `symmetric' communication, occurring in groups of interacting peers. This symmetric model of communication is captured in the successful Message Passing Interface standard (MPI).

Message-Passing Interface Basics:

Every MPI program must contain the preprocessor directive:

```
#include <mpi.h>
```

The mpi.h file contains the definitions and declarations necessary for compiling an MPI program.

MPI_Init initializes the execution environment for MPI. It is a "share nothing" modality in which the outcome of any one of the concurrent processes can in no way be influenced by the intermediate results of any of the other processes. Command has to be called before any other MPI call is made, and it is an error to call it more than a single time within the program. **MPI_Finalize** cleans up all the extraneous mess that was first put into place by MPI_Init.

The principal weakness of this limited form of processing is that the processes on different nodes run entirely independent of each other. It cannot enable capability or coordinated computing. **To get the different processes to interact, the concept of communicators is needed.** MPI programs are made up of concurrent processes executing at the same time that in almost all cases

are also communicating with each other. To do this, an object called the "communicator" is provided by MPI. Thus the user may specify any number of communicators within an MPI program, each with its own set of processes. "**MPI_COMM_WORLD**" communicator contains all the concurrent processes making up an MPI program.

The size of a communicator is the number of processes that makes up the particular communicator. The following function call provides the value of the number of processes of the specified communicator:

int MPI_Comm_size(MPI_Comm comm, int _size).

The function "MPI_Comm_size" required to return the number of processes; int size. MPI_Comm_size(MPI_COMM_WORLD,&size); This will put the total number of processes in the MPI_COMM_WORLD communicator in the variable size of the process data context. Every process within the communicator has a unique ID referred to as its "rank". MPI system automatically and arbitrarily assigns a unique positive integer value, starting with 0, to all the processes within the communicator. The MPI command to determine the process rank is:

int MPI_Comm_rank (MPI_Comm comm, int _rank).

The send function is used by the source process to define the data and establish the connection of the message. The send construct has the following syntax:

int MPI_Send (void _message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

The first three operands establish the data to be transferred between the source and destination processes. The first argument points to the message content itself, which may be a simple scalar or a group of data. The message data content is described by the next two arguments. The second operand specifies the number of data elements of which the message is composed. The third operand indicates the data type of the elements that make up the message.

The receive command (MPI_Recv) describes both the data to be transferred and the connection to be established. The MPI_Recv construct is structured as follows:

int MPI_Recv (void _message, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status _status)

The source field designates the rank of the process sending the message.

Communication Collectives: Communication collective operations can dramatically expand interprocess communication from point-to-point to n-way or all-way data exchanges.

The scatter operation: The scatter collective communication pattern, like broadcast, shares data of one process (the root) with all the other processes of a communicator. But in this case it partitions a set of data of the root process into subsets and sends one subset to each of the processes. Each receiving process gets a different subset, and there are as many subsets as there are processes. In this example the send array is A and the receive array is B. B is initialized to 0. The root process (process 0 here) partitions the data into subsets of length 1 and sends each subset to a separate process.

MPJ Express is an open source Java message passing library that allows application developers to write and execute parallel applications **for multicore processors and compute clusters / clouds.** The software is distributed under the MIT (a variant of the LGPL) license. MPJ Express is a message passing library that can be used by application developers to execute their parallel Java applications on compute clusters or network of computers.

MPJ Express is essentially a middleware that supports communication between individual processors of clusters. **The programming model followed by MPJ Express is Single Program**

Multiple Data (SPMD).

The multicore configuration is meant for users who plan to write and execute parallel Java applications using MPJ Express on their desktops or laptops which contains shared memory and multicore processors. In this configuration, users can write their message passing parallel application using MPJ Express and it will be ported automatically on multicore processors. We except that users can first develop applications on their laptops and desktops using multicore configuration, and then take the same code to distributed memory platforms

Designing the solution:

While designing the solution, we have considered the multi-core architecture as per shown in the diagram below. The communicator has processes as per input by the user. MPI program will execute the sequence as per the supplied processes and the number of processor cores available for the execution.

Implementing the solution:

1. For implementing the MPI program in multi-core environment, we need to install MPJ express library.
    a. Download MPJ Express (mpj.jar) and unpack it.
    b. Set MPJ_HOME and PATH environment variables:

c. export MPJ_HOME=/path/to/mpj/

d. export PATH=$MPJ_HOME/bin:$PATH

2. Write Hello World parallel Java program and save it as HelloWorld.java (Asign2.java).
3. Compile a simple Hello World (Asign) parallel Java program
4. Running MPJ Express in the Multi-core Configuration.

Conclusion:

Thus Student develop a distributed system application, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.

Outcome:

1. Students learn fundamentals of parallel programming and MPI.
2. Students design parallel algorithms and write parallel programs using the MPI library.

FAQ:

1. What is use of MPI?
2. Application in which we are using MPI?
3. Why we are providing rank to process in MPI.
4. Explain MPI operations.
5. Explain Different data types of MPI.
6. Draw MPI architecture.
7. What is MPI_ABORT?
8. What is MPI_FINALIZE
9. What is difference MPI_ABORT and MPI_FINALIZE

ASSIGNMENT NO. 5

Title: Clock Synchronization

Aim: To implement Berkeley algorithm for clock synchronization.

Objective:

1. To understand the basics of physical and Logical clock in DS.
2. To develop an n-node distributed system that implements Berkeley's time synchronization algorithm.

Related Theory:

Berkeley‟s Algorithm is a clock synchronization technique used in distributed systems. The algorithm assumes that each machine node in the network either doesn‟t have an accurate time source or doesn‟t possess an UTC server.

Algorithm:

- An individual node is chosen as the master node from a pool nodes in the network. This node is the main node in the network which acts as a master and rest of the nodes act as slaves. Master node is chosen using a election process/leader election algorithm.

- Master node periodically pings slaves nodes and fetches clock time at them using Cristian‟s algorithm.

- Master node calculates average time difference between all the clock times received and the clock time given by master‟s system clock itself. This average time difference is added to the current time at master‟s system clock and broadcasted over the network. Scope of Improvement

- Improvision in accuracy of cristian‟s algorithm.

- Ignoring significant outliers in calculation of average time difference

- In case master node fails/corrupts, a secondary leader must be ready/pre-chosen to take the place of the master node to reduce downtime caused due to master‟s unavailability.

- Instead of sending the synchronized time, master broadcasts relative inverse time difference, which leads to decrease in latency induced by traversal time in the network while time of calculation at slave node.

Scope of Improvement

- Improvision in accuracy of cristian‟s algorithm.
- Ignoring significant outliers in calculation of average time difference
- In case master node fails/corrupts, a secondary leader must be ready/pre-chosen to take the place of the master node to reduce downtime caused due to master‟s unavailability.
- Instead of sending the synchronized time, master broadcasts relative inverse time difference, which leads to decrease in latency induced by traversal time in the network while time of calculation at slave node.

Conclusion:

The Berkeley algorithm is a simple yet effective solution for clock synchronization in distributed systems. Its decentralized approach allows for resilience against failures, and it remains a relevant and widely used tool in ensuring accurate timekeeping between machines.

Outcome:

1. Students learn fundamental of clock synchronization in DS.
2. Students implemented Berkeley algorithm for clock synchronization.

FAQ:

1. What is difference between logical clock and physical clock?
2. Why is it necessary to synchronize the clocks in distributed real time system?
3. How the principle of Berkeley algorithm is used to synchronize time in distributed system?
4. What are other algorithms for clock synchronization in DS?

ASSIGNMENT NO. 6

Title: Mutual Exclusion

Aim: Implement token ring based mutual exclusion algorithm.

Objective:

1. To understand the concept of starvation, Mutual Exclusion in DS.
2. To Achieve Mutual Exclusion In Distributed System based on Token Exchange.

Related Theory:

Token Ring algorithm achieves mutual exclusion in a distributed system by creating a bus network of processes. A logical ring is constructed with these processes and each process is assigned a position in the ring. Each process knows who is next in line after itself. When the ring is initialized, process 0 is given a token. The token circulates around the ring. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token to the next process in the ring. It is not allowed to enter the critical region again using the same token. If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes the token along to the next process.

Advantages:

- The correctness of this algorithm is evident.
- Only one process has the token at any instant, so only one process can be in a CS o Since the token circulates among processes in a well-defined order, starvation cannot occur.

Disadvantages

- Once a process decides it wants to enter a CS, at worst it will have to wait for every other process to enter and leave one critical region.
- If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is not a constant. The fact that the token has not been spotted for an hour does not mean that it has been lost; some process may still be using it.
- The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can pass the token to the next member down the line

Conclusion:

The token ring-based mutual exclusion algorithm is a well-known solution for coordinating access to shared resources in distributed systems. Its simple and efficient design ensures that only one process can access a shared resource at a time, thus preventing conflicts and ensuring consistency. While the algorithm can suffer from potential delays and network congestion, it remains a widely used and effective solution for achieving mutual exclusion in distributed systems.

Outcome:

1. Students learn concept of Mutual exclusion to prevent Race conditions.
2. Students have implemented token ring based mutual exclusion algorithm.

FAQ:

1. What is race condition?
2. What is deadlock and starvation?
3. What is Mutual Exclusion?
4. How to avoid mutual exclusion using

ASSIGNMENT NO. 7

Title: Election Algorithms

Aim: To Implement Bully and Ring algorithm for leader election.

Objective:

1. To enable distributed systems to select a leader in a decentralized manner, without requiring a centralized control mechanism.
2. To ensure that the leader selection process is reliable and efficient, even in the presence of failures, network delays, and other forms of uncertainty.
3. To provide a fair and deterministic method for selecting the leader, such that all nodes in the system have an equal chance of being chosen.

Tools / Environment:

Java Programming Environment, JDK 1.8, Eclipse Neon(EE).

Related Theory: Election Algorithm:

1. Many distributed algorithms require a process to act as a coordinator.
2. The coordinator can be any process that organizes actions of other processes.
3. *A* coordinator may fail.
4. How is a new coordinator chosen or elected?

Assumptions:

Each process has a unique number to distinguish them. Processes know each other's process number.\

There are two types of Distributed Algorithms:

1. Bully Algorithm
2. Ring Algorithm

Bully Algorithm:

A. When a process, P, notices that the coordinator is no longer responding to requests, it initiates an election.

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

B. When a process gets an ELECTION message from one of its lower-numbered colleagues:

1. Receiver sends an OK message back to the sender to indicate that he is alive and will take over.

2. Eventually, all processes give up apart of one, and that one is the new coordinator.

3. The new coordinator announces *its* victory by sending all processes a **CO-ORDINATOR** message telling them that it is the new coordinator.

C. If a process that *was* previously down comes back:

1. It holds an election.
2. If it happens to be the highest process currently running, it will win the election and take over the coordinators job.

"Biggest guy" always wins and hence the name bully algorithm.

Ring Algorithm

Initiation:

1. When a process notices that coordinator is not functioning:

2. Another process (initiator) initiates the election by sending "ELECTION" message (containing its own process number)

Leader Election:

3. Initiator sends the message to it's successor (if successor is down, sender skips over it and goes to the next member along the ring, or the one after that, until a running process is located).

4. At each step, sender adds its own process number to the list in the message.

5. When the message gets back to the process that started it all: Message comes back to initiator. In the queue the **process with maximum ID Number wins**.

Initiator announces the winner by sending another message around the ring.

Designing the solution: A. For Ring Algorithm

Initiation:

1. Consider the Process 4 understands that Process 7 is not responding.

2. Process 4 initiates the Election by sending "ELECTION" message to it's successor (or next alive process) with it's ID.

Leader Election:

3. Messages comes back to initiator. Here the initiator is 4.

4. Initiator announces the winner by sending another message around the ring. Here the process with highest process ID is 6. The initiator will announce that Process 6 is Coordinator.

Implementing the solution:

For Ring Algorithm:

1. Creating Class for Process which includes
   i) State: Active / Inactive

   ii) Index: Stores index of process.
   iii) ID: Process ID

2. Import Scanner Class for getting input from Console
3. Getting input from User for number of Processes and store them into object of classes.
4. Sort these objects on the basis of process id.
5. Make the last process id as "inactive".
6. Ask for menu 1.Election 2.Exit
7. Ask for initializing election process.

8. These inputs will be used by Ring Algorithm.

Conclusion:

Election algorithms **are designed to choose a coordinator.** We have two election algorithms for two different configurations of distributed system. **The Bully** algorithm applies to system where every process can send a message to every other process in the system and **The Ring** algorithm applies to systems organized as a ring (logically or physically). In this algorithm we assume that the link between the process are unidirectional and every process can message to the process on its right only.

Outcome:

1. Students learned the fundamentals of process coordinator election algorithms in DS
2. Students developed Bully and Ring algorithm for leader election

FAQ:

1. Who is process coordinator? What are its responsibilities?
2. Need of Election Algorithm?
3. What is centralized and decentralized algorithm?
4. Explain Election working of algorithm for Ring & Bully?
5. What is „Token“?
6. Why algorithm is known as “Bully”?

ASSIGNMENT NO. 8

Title: Web Services

Aim: To create a simple web service and write any distributed application to consume the web service.

Objective:

1. To understands the fundaments of web services, architecture and its types.
2. To create a simple web service.

Tools / Environment:

Java Programming Environment, JDK 8, Netbeans IDE with GlassFish Server

Related Theory:

Web Service:

A web service can be defined as a collection of open protocols and standards for exchanging information among systems or applications.

A service can be treated as a web service if:

The service is discoverable through a simple lookup

It uses a standard XML format for messaging

It is available across internet/intranet networks.

It is a self-describing service through a simple XML syntax

The service is open to, and not tied to, any operating system/programming language

Types of Web Services:

There are two types of web services:

1. **SOAP**: SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services. Since it‟s XML

based, it‟s platform and language independent. So, our server can be based on JAVA and client can be on NET, PHP etc. and vice versa.

2. **REST**: REST (Representational State Transfer) is an architectural style for developing web services. It‟s getting popularity recently because it has small learning curve when compared to SOAP. Resources are core concepts of Restful web services and they are uniquely identified by their URIs.

Web service architectures:

As part of a web service architecture, there exist three major roles.

Service Provider is the program that implements the service agreed for the web service and exposes the service over the internet/intranet for other applications to interact with.

Service Requestor is the program that interacts with the web service exposed by the Service Provider. It makes an invocation to the web service over the network to the Service Provider and exchanges information.

Service Registry acts as the directory to store references to the web services.

The following are the steps involved in a basic SOAP web service operational behavior:

1. The client program that wants to interact with another application prepares its request content as a SOAP message.
2. Then, the client program sends this SOAP message to the server web service as an HTTP POST request with the content passed as the body of the request.
3. The web service plays a crucial role in this step by understanding the SOAP request and converting it into a set of instructions that the server program can understand.
4. The server program processes the request content as programmed and prepares the output as the response to the SOAP request.
5. Then, the web service takes this response content as a SOAP message and reverts to the SOAP HTTP request invoked by the client program with this response.
6. The client program web service reads the SOAP response message to receive the outcome of the server program for the request content it sent as a request.

SOAP web services

Simple Object Access Protocol (**SOAP**) is an XML-based protocol for accessing web services. It is a W3C recommendation for communication between two applications, and it is a platform- and language-independent technology in integrated distributed applications.

While XML and HTTP together make the basic platform for web services, the following are the key components of standard SOAP web services:

Universal Description, Discovery, and Integration (**UDDI**)*:* UDDI is an XMLbased framework for describing, discovering, and integrating web services. It acts as a directory of web service interfaces described in the WSDL language.

Web Services Description Language (**WSDL**)*:* WSDL is an XML document containing information about web services, such as the method name, method parameters, and how to invoke the service. WSDL is part of the UDDI registry. It acts as an interface between applications that want to interact based on web services. The following diagram shows the interaction between the UDDI, Service Provider, and service consumer in SOAP web services:

RESTful web services

REST stands for **Representational State Transfer**. RESTful web services are considered a performance-efficient alternative to the SOAP web services. REST is an architectural style, not a protocol. Refer to the following diagram:

While both SOAP and RESTful support efficient web service development, the difference between these two technologies can be checked out in the following table :

1. Students understand the concept of web services.
2. Students developed web service and write any distributed application to consume the web service.

FAQ

1. What Is a Web Service?
2. Explain Architecture of web services w. r. to Provider, Requestor, Service registry and Broker?
3. What is WSDL?
4. List types of Web services?
5. Differentiate between SOAP and REST?
6. List the examples of web services?
7. List applications of web services?

ASSIGNMENT NO. 9