Assignment No. 1

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

Objectives: To develop a multi-threaded client/server process communication using Java RMI.

Infrastructure:

Software Used: Java, Eclipse IDE, JDK

Theory:
Remote method invocation (RMI) allows a java object to invoke method on an object running on another machine. RMI provide remote communication between java program. RMI is used for building distributed application.

RMI applications often comprise two separate programs, a server and a client.
1. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.
2. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.

RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application.

Distributed object applications need to do the following:
- Locate remote objects.
- Communicate with remote objects.
- Load class definitions for objects that are passed around.

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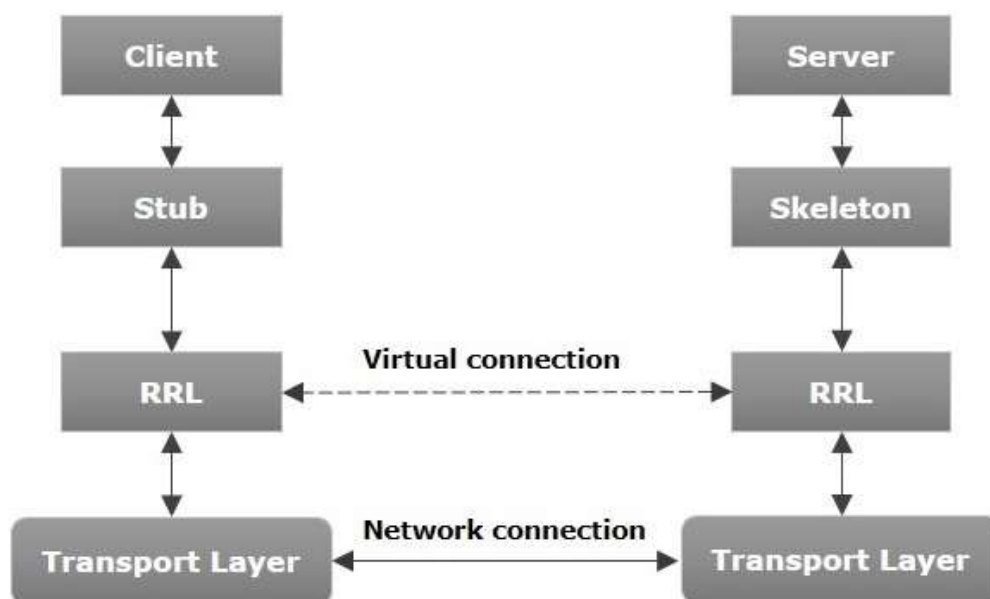
- **Transport Layer** − This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton**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

How RMI provide remote Communication?

The RMI provides remote communication between the applications using two objects stub and skeleton. A remote object is an object whose method can be invoked from another JVM.

1. stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

2. Skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:
1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

Remote Interfaces, Objects, and Methods
The distributed application built using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others. Objects with methods that can be invoked across Java virtual machines are called remote objects.

An object becomes remote by implementing a remote interface, which has the following characteristics:
- A remote interface extends the interface java.rmi.Remote.
- Each method of the interface declares java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a non-remote object when the object is passed from one Java virtual machine to another Java virtual machine. Rather than making a copy of the implementation object in the receiving Java virtual machine, RMI passes a remote stub for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference. The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This property enables a stub to be cast to any of the interfaces that the remote object implements. However, only those methods defined in a remote interface are available to be called from the receiving Java virtual machine.

Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as **marshalling**.

At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

RMI Registry

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

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using **lookup()** method).
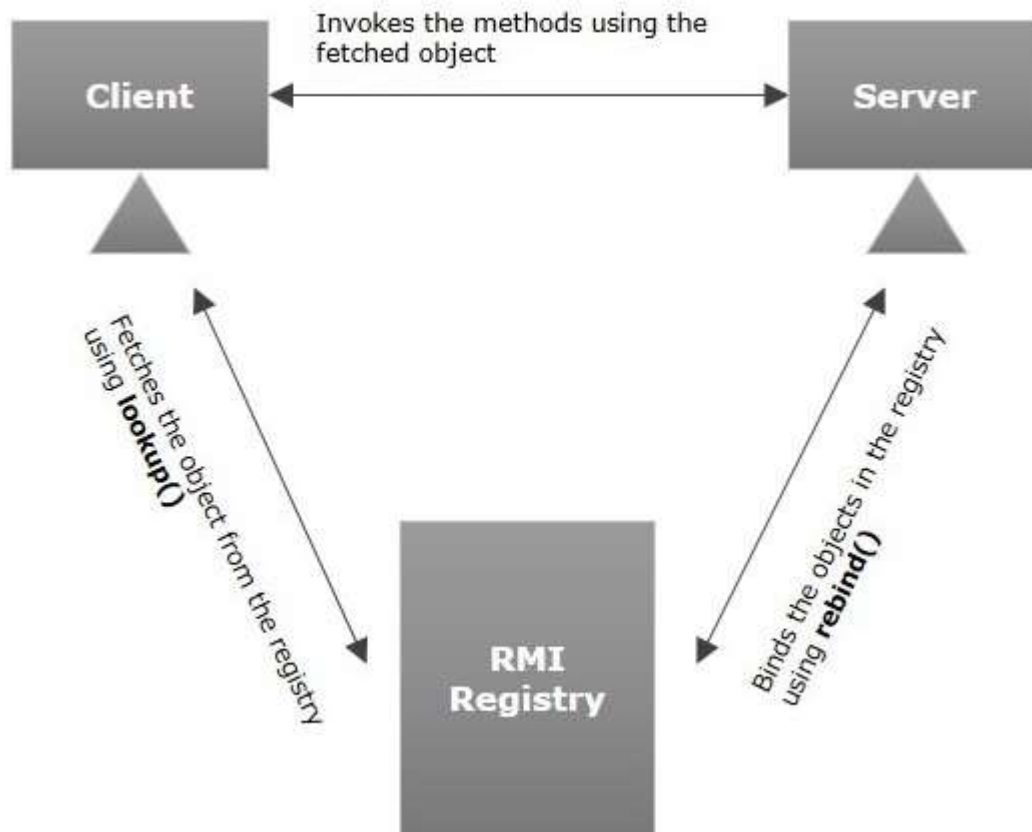
Creating Distributed Applications by Using RMI
Using RMI to develop a distributed application involves these general steps:
1. Designing and implementing the components of your distributed application.
2. Compiling sources.
3. Making classes network accessible.
4. Starting the application.

Goals of RMI

Following are the goals of RMI −

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

Conclusion:
We implemented a multi-thread client/server process communication using RMI.

Assignment No. 2

Aim: Develop any distributed application using CORBA to demonstrate object brokering. (Calculator or String operations).

Objective: Demonstration of object brokering using CORBA in Java.

Infrastructure:

Software used: Java, JDK 1.8, IDE like Eclipse(optional)

Theory:

Common Object Request Broker Architecture (CORBA):
CORBA is an acronym for Common Object Request Broker Architecture, 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.

Why CORBA is used?
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.

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 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. 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.
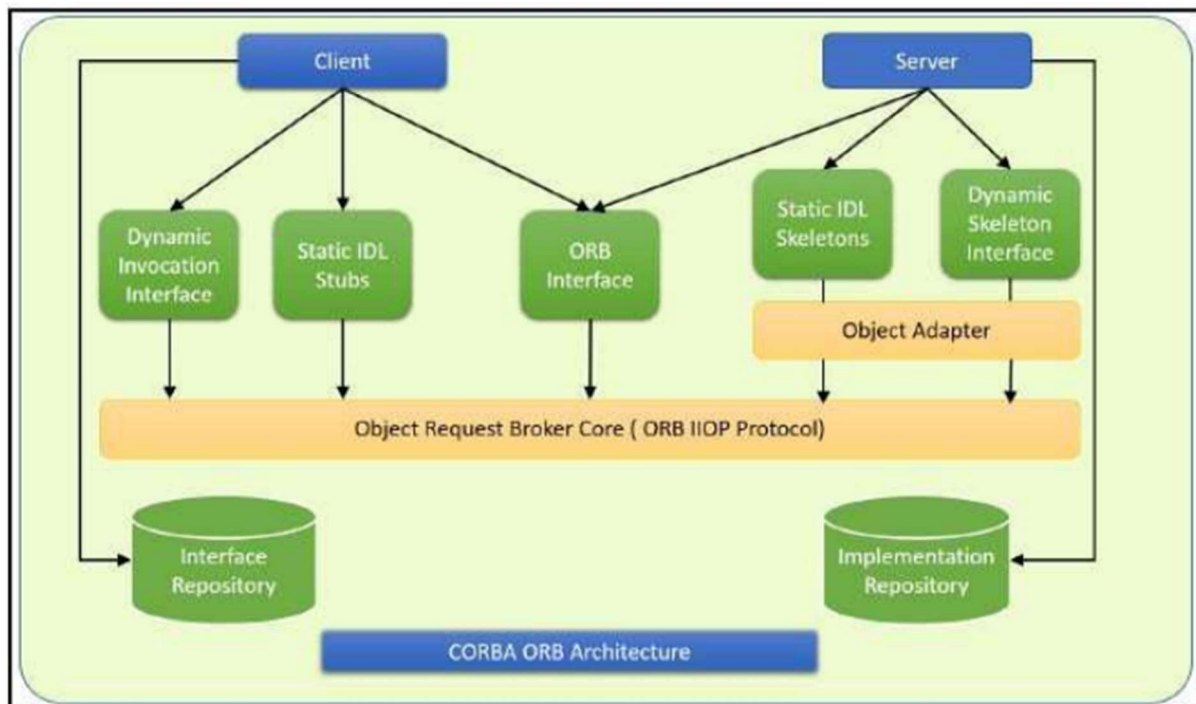
The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons, The objects are written (on the right) and a client for it (on the left), as represented in the diagram.

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.

Java Support for CORBA

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.

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.

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 this, 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:

1. Creating CORBA Objects using Java IDL:

1.1. First one has to define CORBA-enabled interface and it implementation, which involves:

> • 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. 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.

```
1. // IDL
2. module jen {
3. module corba {
4. interface NeatExample ...
5. };
6. };
7.
```

1.3. Interfaces

It involves declaration of interface which includes an interface header and an interface body. For example:

```
1. interface PrintServer : Server { ...
2.
```

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.

```
1. readonly attribute string myString;
```

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

```
1. 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:

```
 1. module OS {
 2. module services {
 3. interface Server {
 4. readonly attribute string serverName;
 5. boolean init(in string sName);
 6. };
 7. interface Printable {
 8. boolean print(in string header);
 9. };
10. interface PrintServer : Server {
11. boolean printThis(in Printable p);
12. };
13. };
14. };
15.
```

2. Turning IDL Into Java

Now for implementing the remote interfaces in Java we use 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 helper class whose name is the name of the IDL interface with "Helper" appended to it

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

> 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 serverside 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.

3. Writing the Implementation

The IDL interface is written and generated the Java interface and support classes for it, including the client stub and the server skeleton.

Implementing the solution:

Now create a directory named hello/ where you develop sample applications and create the files in this directory.

1. Defining the Interface (Hello.idl)

The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL). To complete the application,

you simply provide the server (HelloServer.java) and client (HelloClient.java) implementations.

2. Implementing the Server (HelloServer.java)

The servant, HelloImpl, is the implementation of the Hello IDL interface; each Hello instance is implemented by a HelloImpl instance. The servant is a subclass of HelloPOA, which is generated by the idlj compiler from the example IDL. The servant contains one method for each IDL operation, in this example, the sayHello() and shutdown() methods. Servant methods are just like ordinary Java methods; the extra code to deal with the ORB, with marshaling arguments and results, and so on, is provided by the skeleton.

The HelloServer class has the server's main() method, which:

> • Creates and initializes an ORB instance

> • Gets a reference to the root POA and activates the POAManager

> • Creates a servant instance (the implementation of one CORBA Hello object) and tells the ORB about it

> • Gets a CORBA object reference for a naming context in which to register the new CORBA object

> • Gets the root naming context

> • Registers the new object in the naming context under the name "Hello"

> • Waits for invocations of the new object from the client.
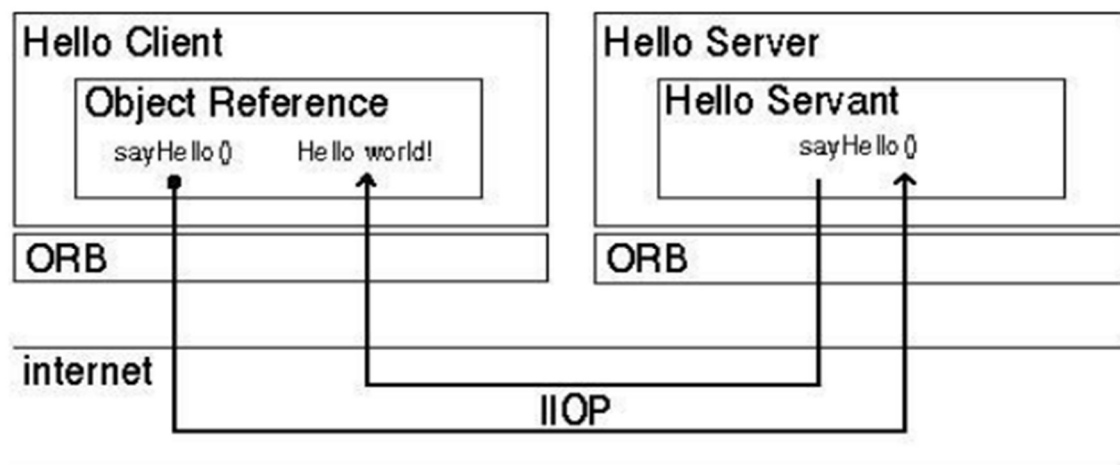
3. Implementing the Client Application (HelloClient.java)

The example application client that follows:

> • Creates and initializes an ORB

> • Obtains a reference to the root naming context

> • Looks up "Hello" in the naming context and receives a reference to that CORBA object

• Invokes the object's sayHello() and shutdown() operations and prints the result.

Building and executing the solution:

Now we use naming service, which is a CORBA service that allows CORBA objects to be named by means of binding a name to an object reference. The name binding may be stored in the naming service, and a client may supply the name to obtain the desired object reference. The two options for Naming Services with Java include orbd, a daemon process containing a Bootstrap Service, a Transient Naming Service,

To run this client-server application on the development machine:

      1. Change to the directory that contains the file Hello.idl.

      2. Run the IDL-to-Java compiler, idlj, on the IDL file to create stubs and skeletons.

This step assumes that you have included the path to the java/bin directory in your path.

idlj -fall Hello.idl

You must use the -fall option with the idlj compiler to generate both client and serverside bindings.

The files generated by the idlj compiler for Hello.idl, with the -fall command line option, are:

- HelloPOA.java
- _HelloStub.java
- Hello.java
- HelloHelper.java
- HelloHolder.java
- HelloOperations.java

3. Compile the .java files, including the stubs and skeletons (which are in the directory directory HelloApp). This step assumes the java/bin directory is included in your path.

```
1. javac *.java HelloApp/*.java
```

4. Start orbd.

To start orbd from a UNIX command shell,

      enter: orbd -ORBInitialPort 1050&

5. Start the HelloServer:

      To start the HelloServer from a UNIX command shell, enter:

            java HelloServer -ORBInitialPort 1050 -

            ORBInitialHost localhost&

You will see HelloServer ready and waiting... when the server is started.

6. Run the client application:

java HelloClient -ORBInitialPort 1050 -

ORBInitialHost localhost

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.

Assignment No. 3

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.

Objectives: To learn about MPI, its benefits, and its implementation.

Infrastructure:

Software used: Python, mpi4py library, Microsoft MPI v10.0 (https://www.microsoft.com/en-us/download/details.aspx?id=57467), Numpy

Theory:

With the advent of high-performance multicomputer, developers have been looking for message-oriented primitives that would allow them to easily write highly efficient applications. This means that the primitives should be at a convenient level of abstraction (to ease application development), and that their implementation incurs only minimal overhead.

Sockets were deemed insufficient for two reasons.

- First, they were at the wrong level of abstraction by supporting only simple send and receive primitives.
- Second, sockets had been designed to communicate across networks using general-purpose protocol stacks such as TCPIIP.

They were not considered suitable for the proprietary protocols developed for high-speed interconnection networks, such as those used in high-performance server clusters. Those protocols required an 'interface that could handle more advanced features, such as different forms of buffering and synchronization.

The result was that most interconnection networks and high-performance multicomputers were shipped with proprietary communication libraries. These libraries offered a wealth of high-level and generally efficient communication primitives. Of course, all libraries were mutually incompatible, so that application developers now had a portability problem.

The need to be hardware and platform independent eventually led to the definition of a standard for message passing, simply called the Message-Passing Interface or MPI. MPI is designed for parallel applications and as such is tailored to transient communication. It makes direct use of the underlying network. Also, it assumes that serious failures such as process crashes or network partitions are fatal and do not require automatic recovery.

MPI assumes communication takes place within a known group of processes. Each group is assigned an identifier. Each process within a group is also assigned a (local) identifier. A (group/D, process/D) pair therefore uniquely identifies the source or destination of a message, and is used instead of a transport-level address. There may be several, possibly overlapping groups of processes involved in a computation and that are all executing at the same time.

At the core of MPI are messaging primitives to support transient communication, of which the most intuitive ones are summarized.

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

Figure 4-16. Some of the most intuitive message-passing primitives of MPI.

How MPI Works?

Transient asynchronous communication is supported by means of the MPI_bsend primitive. The sender submits a message for transmission, which is generally first copied to a local buffer in the MPI runtime system. When the message has been copied. the sender continues. The local MPI runtime system will remove the message from its local buffer and take care of transmission as soon as a receiver has called a receive primitive.

There is also a blocking send operation, called MPLsend, of which the sem antics are implementation dependent. The primitive MPLsend may either block the caller until the specified message has been copied to the MPI runtime system at the sender's side, or until the receiver has initiated a receive operation. Syn chronous communication by which the sender blocks until its request is accepted for further processing is available through the MPI~ssend primitive. Finally, the strongest form of synchronous communication is also supported: when a sender calls MPLsendrecv, it sends a request to the receiver and blocks until the latter returns a reply. Basically, this primitive corresponds to a normal RPC.

Both MPLsend and MPLssend have variants that avoid copying messages from user buffers to buffers internal to the local MPI runtime system. These vari ants correspond to a form of asynchronous communication. With MPI_isend, a sender passes a pointer to the message after which the MPI runtime system takes care of communication. The sender immediately continues. To prevent overwrit ing the message before communication completes, MPI offers primitives to check for completion, or even to block if required. As with MPLsend, whether the mes sage has actually been transferred to the receiver or that it has merely been copied by the local MPI runtime system to an internal buffer is left unspecified. Likewise, with MPLissend, a sender also passes only a pointer to the :MPI runtime system. When the runtime system indicates it has processed the message, the sender is then guaranteed that the receiver has accepted the message and is now working on it. The operation MPLrecv is called to receive a message; it blocks the caller until a message arrives. There is also an asynchronous variant, called MPLirecv, by which a receiver indicates that is prepared to accept a message. The receiver can check whether or not a message has indeed arrived, or block until one does.

What is mpi4py?

MPI for Python provides MPI bindings for the Python language, allowing programmers to exploit multiple processor computing systems. mpi4py is is constructed on top of the MPI-1/2 specifications and provides an object oriented interface which closely follows MPI-2 C++ bindings.
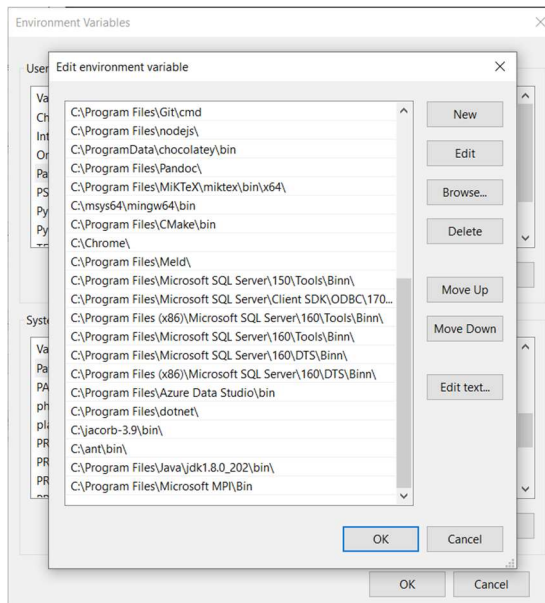
Installing mpi4py

```
pip install mpi4py
```

Microsoft MPI V10

Installation Process

1. Download the MPI exe file from the URL provided above.

2. Install the exe file.

3. Setup the path in the System Variable to the installed locations bin folder.

   i.e. As visible in the last line.

   Note: This path may differ according to the path set at the installation time.

4. To verify the MPI software is installed correctly, open a new command prompt ant type.

```
1. mpiexec -help
```

   If everything is installed correctly, it will show the list of options available in the MPI.

Developing the code

1. For distributed addition of the array of numbers we are going to use Reduce method available in the MPI operation.
2. For this first of all we are going to initialize the MPI, followed by the accepting the number of processes for performing the computation.
3. Then we assign rank to each process.
4. Initialize the array, and its value.
5. Send the sub-array of equal size to each process, where it gets computed.
6. This computed value from each process is added together to get the global value or the total sum of the array.

Compiling the Code

We run the code by the following command.

```
1. mpiexec -np 3 python sum.py
```

Here, 3 represent the number of processes.

Conclusion:

We learnt about MPI, how it was introduced and how to implement distributed computing with the help of MPI.

Assignment No. 4

Aim: Implement Berkeley algorithm for clock synchronization.

Objective: To understand Berkeley algorithm for clock synchronization and its implementation.

Infrastructure: Python environment.

Software Requirements: Python 3.0.

Theory:

Berkeley Algorithm

Berkeley's Algorithm is a distributed algorithm for computing the correct time in a network of computers. The algorithm is designed to work in a network where clocks may be running at slightly different rates, and some computers may experience intermittent communication failures.

The basic idea behind Berkeley's Algorithm is that each computer in the network periodically sends its local time to a designated "master" computer, which then computes the correct time for the network based on the received timestamps. The master computer then sends the correct time back to all the computers in the network, and each computer sets its clock to the received time.

There are several variations of Berkeley's Algorithm that have been proposed, but a common version of the algorithm is as follows –

- Each computer starts with its own local time, and periodically sends its time to the master computer.
- The master computer receives timestamps from all the computers in the network.
- The master computer computes the average time of all the timestamps it has received and sends the average time back to all the computers in the network.
- Each computer sets its clock to the time it receives from the master computer.
- The process is repeated periodically, so that over time, the clocks of all the computers in the network will converge to the correct time.

Benefit: It is relatively simple to implement and understand.
Limitation: The time computed by the algorithm is based on the network conditions and time of sending and receiving timestamps which can't be very accurate and also it has a requirement of a master computer which if failed can cause the algorithm to stop working.

More advance algorithm such as the Network Time Protocol (NTP) which use a more complex algorithm and also consider the network delay and clock drift to get a more accurate time.

Scope of Improvement

There are several areas where Berkeley's Algorithm can be improved –

- **Accuracy** – The algorithm calculates the average time based on the timestamps received from all the computers in the network, which can lead to inaccuracies, especially if the network has a high degree of jitter or delay.
- **Robustness** – The algorithm requires a master computer, which if it fails, can cause the algorithm to stop working. If the master computer is a single point of failure, it can make the algorithm less robust.
- **Synchronization Quality** – The algorithm assumes that all the clocks in the network are running at the same rate, but in practice, clocks may drift due to temperature, aging, or other factors. The algorithm doesn't consider this drift and may fail to achieve a high degree of synchronization between the clocks in the network.
- **Security** – There is no security measures in the algorithm to prevent malicious computers from tampering with the timestamps they send to the master computer, which can skew the results of the algorithm.
- **Adaptability** – The algorithm doesn't adapt well to changes in the network, for example, if a new computer is added to the network or if the network topology changes.

How to use Berkeley's Algorithm

To use Berkeley's Algorithm, you would need to implement the algorithm on each computer in a network of computers. Here is a general overview of the steps you would need to take to implement the algorithm –

1. Designate one computer in the network as the master computer. This computer will be responsible for receiving timestamps from all the other computers in the network and computing the correct time. The master node is chosen using an election process/leader election algorithm.
2. On each computer in the network, set up a timer to periodically send the computer's local time to the master computer.
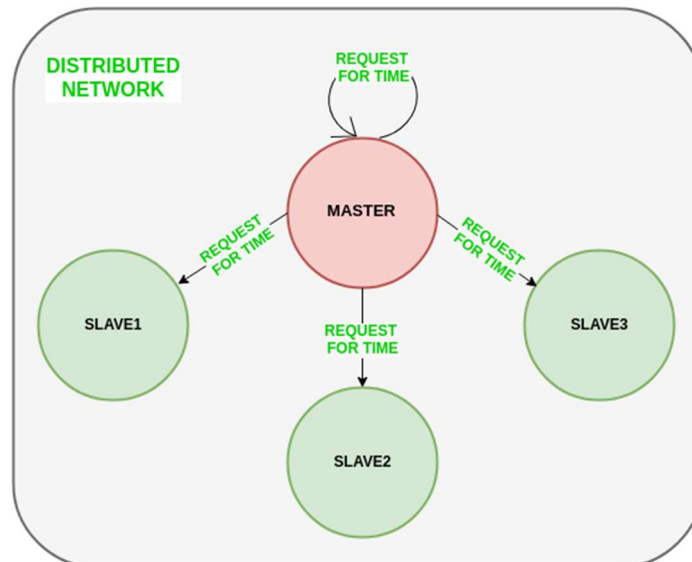
3. On the master computer, set up a mechanism for receiving timestamps from all the computers in the network.
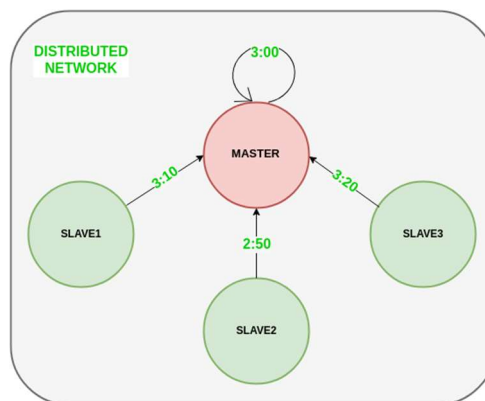
4. On the master computer, implement the logic for calculating the average time based on the received timestamps.
5. On the master computer, set up a mechanism for sending the calculated average time back to all the computers in the network.
6. On each computer in the network, set up a mechanism for receiving the time from the master computer and setting the computer's clock to that time.
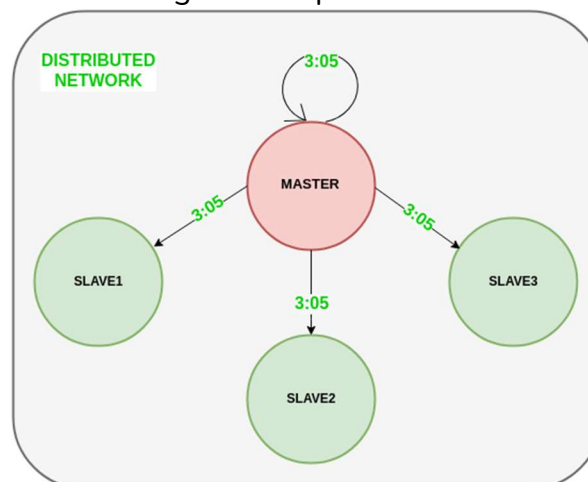
7. Repeat the process periodically, for example, every 30 seconds or 1 minute, to ensure that the clocks in the network stay synchronized.

Conclusion:
We learnt about Berkeley's algorithm for clock synchronization and its implementation.

Assignment No.5

Aim: Implement Token Ring based Mutual Exclusion Algorithm.

**Tools and Environment**: Java Runtime Environment, Java JDK

Theory:

Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process can not enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time.

Mutual exclusion in single computer system Vs. distributed system:

In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved.

In Distributed systems, we neither have shared memory nor a common physical clock and there for we can not solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.

A site in distributed system do not have complete information of state of the system due to lack of shared memory and a common physical clock.
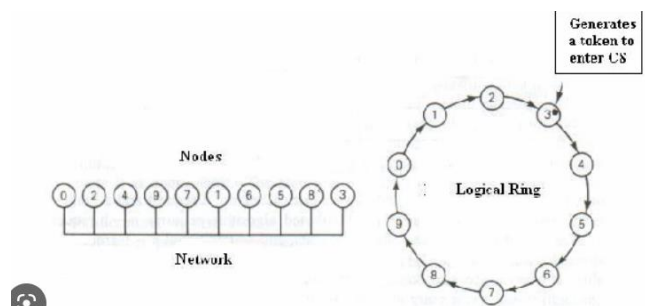
Requirements of Mutual exclusion Algorithm:

No Deadlock:

Two or more site should not endlessly wait for any message that will never arrive.

No Starvation:

Every site who wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other site are repeatedly executing critical section

Fairness:

Each site should get a fair chance to execute critical section. Any request to execute critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system.

Fault Tolerance:

In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

Solution to distributed mutual exclusion:

As we know shared variables or a local kernel can not be used to implement mutual exclusion in distributed systems. Message passing is a way to implement mutual exclusion. Below are the three approaches based on message passing to implement mutual exclusion in distributed systems:

Token Based Algorithm:

A unique token is shared among all the sites.

If a site possesses the unique token, it is allowed to enter its critical section

This approach uses sequence number to order requests for the critical section.

Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.

This approach insures Mutual exclusion as the token is unique

Example:

Suzuki-Kasami's Broadcast Algorithm

Conclusion: Hence Token Ring algorithm achieves mutual exclusion in a distributed system by creating a bus network of processes

Assignment No. 6

Problem Statement: Implementing Bully and Ring Algorithm for Leader Election.

Tools and environment: C++ Programming environment.

Theory:

Election algorithms choose a process from a group of processors to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on other processor. Election algorithm basically determines where a new copy of the coordinator should be restarted. Election algorithm assumes that every active process in the system has a unique priority number. The process with highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects that active process which has highest priority number.Then this number is send to every active process in the distributed system. We have two election algorithms for two different configurations of a distributed system.
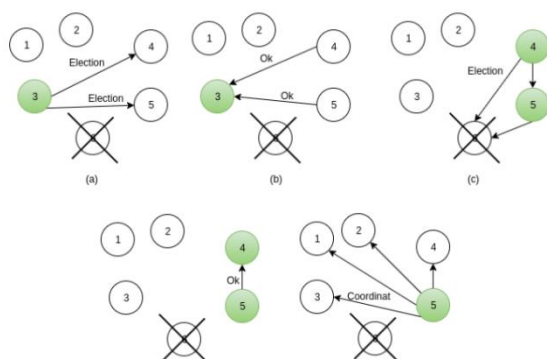

1. **The Bully Algorithm** – This algorithm applies to system where every process can send a message to every other process in the system.

Algorithm –

Suppose process P sends a message to the coordinator.If the coordinator does not respond to it within a time interval T, then it is assumed that coordinator has failed.Now process P sends an election messages to every process with high priority number.It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator.Then it sends a message to all lower priority number processes that it is elected as their new coordinator.However, if an answer is received within time T from any other process Q,

(I) Process P again waits for time interval T' to receive another message from Q that it has been elected as coordinator.

(II) If Q doesn't responds within time interval T' then it is assumed to have failed and algorithm is restarted.

2. **The Ring Algorithm** – This 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. Data structure that this algorithm uses is active list, a list that has a priority number of all active processes in the system.

Algorithm –

If process P1 detects a coordinator failure, it creates new active list which is empty initially. It sends election message to its neighbour on right and adds number 1 to its active list.

If process P2 receives message elect from processes on left, it responds in 3 ways:

(I) If message received does not contain 1 in active list then P1 adds 2 to its active list and forwards the message.

(II) If this is the first election message it has received or sent, P1 creates new active list with numbers 1 and 2. It then sends election message 1 followed by 2.

(III) If Process P1 receives its own election message 1 then active list for P1 now contains numbers of all the active processes in the system. Now Process P1 detects highest priority number from list and elects it as the new coordinator.

Time Complexity : O(n^2) in the worst case scenario, where n is the number of processes.

Space complexity : O(n)
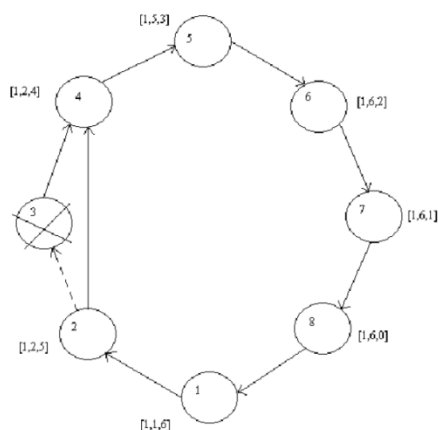
Conclusion: The bully algorithm is a type of Election algorithm which is mainly used for choosing a coordinate. Hence in a distributed system, we need some election algorithms such as bully and ring to get a coordinator that performs functions needed by other processes.

Assignment No. 7

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

Objective: To understand web services, and how distributed applications can be developed to consume web services.

Infrastructure: Python environment.

Software Requirements: Python 3.0, Flask, request library

Theory:
What are Web Services?
A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer.

Components of Web Services
The basic web services platform is XML + HTTP. All the standard web services work using the following components −
   • SOAP (Simple Object Access Protocol)
   • UDDI (Universal Description, Discovery and Integration)
   • WSDL (Web Services Description Language)
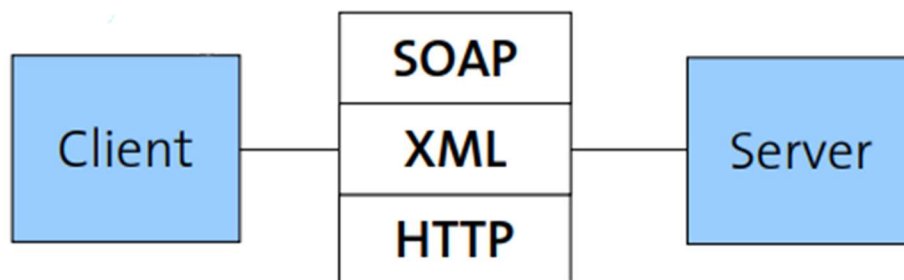
How Does a Web Service Work?
A web service enables communication among various applications by using open standards such as HTML, XML, WSDL, and SOAP. A web service takes the help of -
   • XML to tag the data
   • SOAP to transfer a message
   • WSDL to describe the availability of service.

Types of Web Services
There are mainly two types of web services.
   (I)   SOAP web services.
   (II)   RESTful web services.

Distributed Systems and Web Services
   • Web services provide standards for developing large scale distributed system.
   • Web services on the path of success while CORBA distributed objects failed (This is nothing technical, only a matter of widespread industry acceptance)
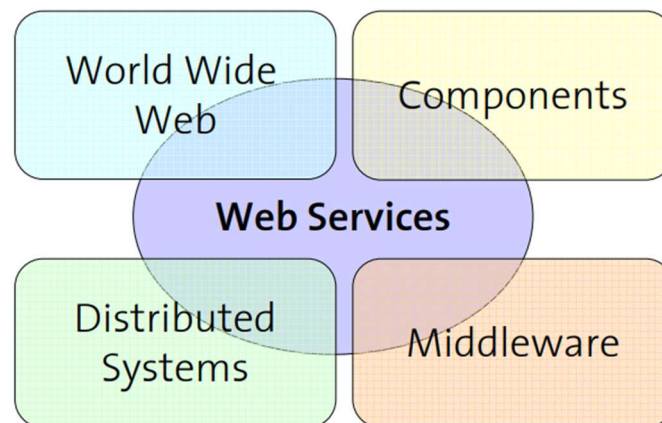
Layers in Distributed Systems
- Client is any user or program that wants to perform an operation over the system. To support a client, the system needs to have a presentation layer through which the user can submit operations and obtain a result.
- The application logic establishes what operations can be performed over the system and how they take place. It takes care of enforcing the business rules and establish the business processes. The application logic can be expressed and implemented in many different ways: constraints, business processes, server with encoded logic ...
- The resource manager deals with the organization (storage, indexing, and retrieval) of the data necessary to support the application logic. This is typically a database but it can also be a text retrieval system or any other data management system providing querying capabilities and persistence.
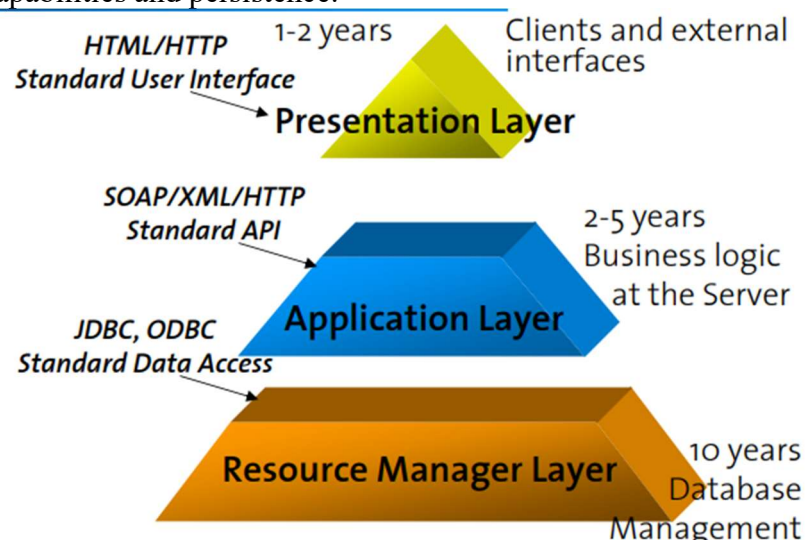
Steps involved in development of Web service, and a distributed application to utilize this Web services are:
1. Setting up the Web Service:
- Choose a programming language and framework for your web service.
- Define the functionality and endpoints of your web service. For simplicity, let's assume you want to create a basic calculator API with two endpoints: POST /add and POST /multiply.
2. Deploying the Web Service:
   You can deploy the web service on any cloud platform.

Alternatively, you can run a local server of the web service.
3. Building the Distributed Application:
   - Define the functionality of your distributed application. In this case, you'll create an application that consumes the calculator API endpoints.
   - Implement the logic to make HTTP requests to the web service endpoints. You can use libraries like axios in JavaScript or requests in Python to send HTTP requests.
   - Parse the responses from the web service and handle any errors that may occur.
4. Test and Run the Distributed Application:
   - Set up the development environment for your distributed application.
   - Run the distributed application and ensure it consumes the web service correctly.
   - Debug and fix any issues that may arise.

Conclusion:
We learnt:
- how the web service works,
- how to use web service in a distributed application,
- implementation of web service and distributed application.