**Name: Deepak Sanjay Baravkar**

**Roll No: BIT06**

--------------------------------------------------------------------------------------------------------------------------

**Practical No: 1**

--------------------------------------------------------------------------------------------------------------------------

**Title of the Assignment:** To develop any distributed application through implementing client-server communication programs based on Java Sockets and RMI techniques..

**-**-------------------------------------------------------------------------------------------------------------------------

## Objective of the Assignment:

To objective of the assignment is to study, how sockets are used to communicate the Client and Server using thread and using Remote Method Invocation technique.

--------------------------------------------------------------------------------------------------------------------------

**Scope:** The scope of the assignment is to cover the following issues:

1. Client-Server communication implementation.
2. Request will be send between Server to Client for getting services from Server.

1. Use of RMI

## --------------------------------------------------------------------------------------------------------------------------

## Theory:

**I) Socket**

Socket functions: The Sockets mainly work on Client and Server side. Socket is nothing but a request, sending for made connection between Client and Server. The socket has following steps on both sides,

## Client Side Steps:

## 1. Socket() 2. Connect()

## 3. Send()

## 4. Recv()

## 5. Close()

## Server Side steps:

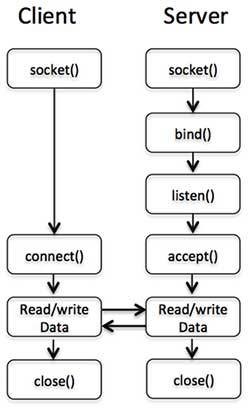
## 1. Socket()

## 2. Bind()

## 3. Listen()

## 4. Accept()

1. **Send**
2. **Recieve**
3. **Close**



## Fig. Socket Communication

## 1. Socket Method:-

The creation of a **Socket** object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

**Socket(String hostName, int port)** - Creates a socket connecting the local host to the namedhost and port; can throw an **UnknownHostException** or an **IOException**.

**Socket (InetAddress *ipAddress*, int *port*)-** Creates a socket using a pre-existing **InetAddress** object and a port; can throw an **IOException**.

## 2. Connect Method:-

public void **connect**() throws **Messaging Exception**

A generic connect method that takes no parameters. Subclasses can implement the appropriate authentication schemes. Subclasses that need additional information might want to use some properties or might get it interactively using a popup window. If the connection is successful, an "open" Connection Event is delivered to any Connection Listeners on this service. Most clients should just call this method to connect to the service.

It is an error to connect to an already connected service. The implementation provided here simply calls the following connect (String, String, String) method with nulls.

# 4. getOutputStream:

3.1 Synopsis

public OutputStream **getOutputStream**() throws IOException

3.2 Description: Returns an output stream for this socket. If this socket has an associated channel then the resulting output stream delegates all of its operations to the channel. If the channel is in non-blocking mode then the output stream's write operations will throw an IllegalBlockingModeException.

# 4. getInputStream:

4.1 Synopsis

public InputStream **getInputStream**()throws IOException

4.2 Description: Returns an input stream for this socket. If this socket has an associated channel then the resulting input stream delegates all of its operations to the channel. If the channel is in non-blocking mode then the input stream's read operations will throw an IllegalBlockingModeException. Under abnormal conditions the underlying connection may be broken by the remote host or the network software (for example a connection reset in the case of TCP connections). When a broken connection is detected by the network software the following applies to the returned input stream :-

* The network software may discard bytes that are buffered by the socket. Bytes that aren't discarded by the network software can be read using read.
* If there are no bytes buffered on the socket, or all buffered bytes have been consumed by read, then all subsequent calls to read will throw an IOException.
* If there are no bytes buffered on the socket, and the socket has not been closed using close, then available will return 0.

Closing the returned InputStream will close the associated socket.

## 5. Close Method:

Closes this socket. Any thread currently blocked in an I/O operation upon this socket will throw a SocketException. Once a socket has been closed, it is not available for further networking use (i.e. can't be reconnected or rebound). A new socket needs to be created. Closing this socket will also close the socket's InputStream and OutputStream.

## 6. Bind Method:

bind() assigns a socket to an address. When a socket is created using socket(), it is only given a protocol family, but not assigned an address. This association with an address must be performed with the bind() system call before the socket can accept connections to other hosts.

Bind() returns 0 on success and -1 if an error occurs.

Prototype int bind(int sockfd, conststructsockaddr \*my\_addr, socklen\_taddrlen);

## 7.Listen Method:

After a socket has been associated with an address, listen() prepares it for incoming connections. However, this is only necessary for the stream-oriented (connection-oriented) data modes,

Once a connection is accepted, it is dequeued. On success, 0 is returned. If an error occurs, -1 is returned.

## 8.Accept Method:

When an application is listening for stream-oriented connections from other hosts, it is notified of such events (cf. select() function) and must initialize the connection using the accept() function. The accept() function creates a new socket for each connection and removes the connection from the listen queue. The accept() function returns the new socket descriptor for the accepted connection, or -1 if an error occurs. All further communication with the remote host now occurs via this new socket. Datagram sockets do not require processing by accept() since the receiver may immediately respond to the request using the listening socket.

1. **Printstream:**

**System.in** is an object of type **InputStream**; **System.out** and **System.err** areobjects of Type **PrintStream.** It used to make output stream.

**Syntax and function:** Output stream that contains print( ) and println( ) **PrintStream** is anoutput stream derived from **OutputStream**, it also implements the low-level method **write( )**. Thus, **write( )** can be used to write to the console

**Problem Statement** This assignment involves the implementation of the following applicationsusing socket

programming:

* 1. Addition of digits of a given number
  2. Find the Factorial of a Number
  3. Perform String Operations

**Lab Environment:** To implement applications using sockets, we use Java programming language. Java socket implementations any operating system plat forms can be used.

**Attacking the problem:** Forth implementations for the assignment problems typical client and server side steps will be performed in the given order. These are basic string and mathematical problems, we solve these problems as standalone problems and later on implement the same using socket based approach.

1. **Thread**

Thread is any process which is executing. A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initiall set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

**Start: start**()is a function from thread class Causes this thread to begin execution; the Java VirtualMachine calls the run method of this thread.

**Run:** run() is a function from thread class If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.

## II) RMI

Java RMI has been introduced to reduce the complexity in developing protocol that relies on UDP and TCP. Java RMI provides a simpler mechanism to invoke method remote. This sample application is layered into 3 tiers: client, RMI server or middleware, and database.Client is very thin and lightweight. Here client is only responsible for user interface while large business processes are handed over to RMI middleware. The middleware contains server objects distributed in registry service. The client will lookup the entire object from the registry service and invoke methods remotely. Database server stands on another layer that will be accessed by the data accessors in RMI middleware.

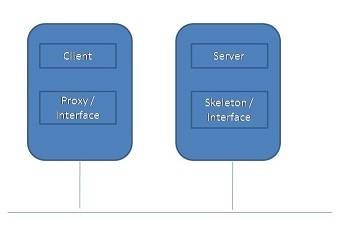


Fig. RMI Communication

## Java RMI deals with the following points:

Remote Procedure Call

Marshalling and Serialization Java

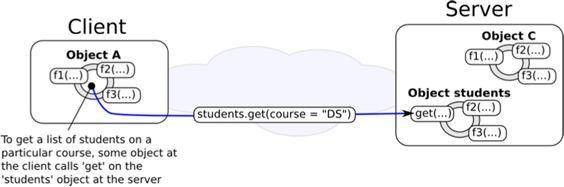
RMI Overview

How to implement Java RMI? How to Run Java RMI?

Stub and Skeleton

**Advantages:** Fair security as the database is hidden from clients Deployment, upgrading and maintenance are easy since the core of business process located in one server

**Disadvantages:** Single point of failure Performance bottle neck if there are too many requests Under saturated network condition, performance of Java RMI is poor compares to TCP and UDP



## Fig. Invoking remote method.

RPC is the middle which allows client to call remote functions located on the server side. RPC is designed bearing in mind to combat the main disadvantage in inter-process communication (UDP and TCP) as it is hard in developing protocol. Moreover, we want to make distributed systems more transparent. In order to call remote functions or methods, client depends on Stub function stimulating client as a real functions or methods locally.

Java RMI or Remote Method Invocation is an example of Remote Procedure Call (RPC) enabling client program to locate the server object and remotely invoke the methods from server through server’s stub and skeleton function. Java RMI is usually referred to Distributed Object rather than RPC. This is because Java natively is object oriented programming. So we say object as a whole. Moreover, RPC somehow is mentioned to procedural programming.

## The main Java RMI component consists of 3 elements:

Client : -Invoke method on remote object

Server : -Process that owns remote object

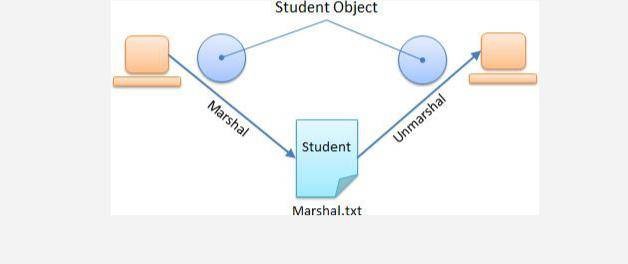
Registry :-Name server that relates objects with unique names

## The process of Java RMI is divided into 4 steps:

1. . Instantiate server object and then register it to RMI registry service with unique name
2. . Client program locates server object from RMI registry service with associated name
3. . Once server object is found is RMI registry, RMI registry returns server stub

13. Server stub handles the data communication with Server skeleton on server side.

## Marshalling and Serialization



**Fig: Marshal object**

**to a file**

Serialization is process of breaking the object(to be sent across network) into form that can be sent across the network (that is converting it into to a sequence of bytes).

**Marshaling** is process of encoding object to put them on the network (on one end) (and thenUnmarshal the object to decode the object and place it in the address space on the other end).

The terms **Marshalling** and **serialization** are used interchangeably to describe the process of squashing data types into a well-defined intermediate format that is suitable for transmission on a network or for storage in a file.

E.g. Remote Procedure Call (RPC) , XML

**Serialization** is the process of converting a set of object instances that contain references to eachother into a linear stream of bytes, which can then be sent through a socket, stored to a file, or simply manipulated as a stream of data. **Serialization** is the mechanism used by RMI to pass objects between JVMs, either as arguments in a method invocation from a client to a server or as return values from a method invocation.

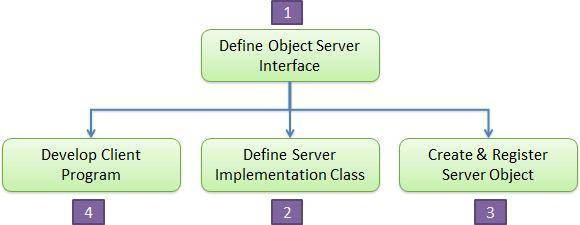
Note that:

Not all data types can be serialized

User -defined class must implement Serializable interface

Attempt to marshal or serialize a non-serializable object will throw an Exception in Java

## How to Implement Java RMI?



## Step 1: Define Object Server Interface

Extend the java.rmi.Remote class

Declare methods with throwing RemoteException

import java.io.\*; import java.rmi.\*; public interface AddI extends Remote

{

public int add(int a ,int b ) throws RemoteException;

}

## Step 2: Define Server Implementation Class

Extend UnicastRemoteObject

Implement interface done in Step 1

Throw RemoteException on constructor

import java.rmi.\*; import java.rmi.server.\*; public class AddServer extends UnicastRemoteObject implements AddI

{

public AddServer() throws RemoteException

{}

public int add(int a, int b)

{ return(a+b); }

}

## Step 3: Create and Register Server Object

Instantiate server object implemented in step 2

Get reference to registry service

Register server object to registry service

import java .rmi.\*; public class RegisterMe

{ public static void main(String args[])

{try

{ AddServer obj=new AddServer(); //create obj addserver

Naming.rebind("add",obj); //binding of obj to name

System.out.println ("Registered successfully");

}

catch(Exception e)

{ System.out.println(e); }}}

## Step 4: Develop Client Program

Get reference to server’s registry

Locate server object from server’s registry service import java.rmi.\*;

public class AddClient

{

public static void main(String args[])

{ try

{int a=10,b=20;

AddI AddIObj=(AddI)Naming.lookup("rmi//localhost/add");

int res=AddIObj.add(a,b); System.out.println("Addition of two number"+res);

}

catch(Exception e)

{

System.out.println(e);

}

}

}

**Stub and Skeleton**

Java RMI is powerful middleware that replaces raw request/reply method as UDP and TCP. The core technology to make it works is Stub and Skeleton function. Stub is located on the client side while Skeleton is located on server side. The role of stub and skeleton is to do marshalling and unmarshalling meaning that it will squash the argument and return value of a method to be a standard format for storing in a file or transferring over the network. Without Stub and Skeleton, java RMI cannot work.

**Code:**

**Client.java**

**package practical;**

**import java.net.\*;**

**import java.io.\*;**

**public class Client {**

**private Socket socket = null;**

**private DataInputStream input = null;**

**private DataOutputStream out = null;**

**public Client(String address, int port) {**

**try {**

**socket = new Socket(address, port);**

**System.out.println("Connected");**

**input = new DataInputStream(System.in);**

**out = new DataOutputStream(socket.getOutputStream());**

**} catch (UnknownHostException u) {**

**System.out.println(u);**

**} catch (IOException i) {**

**System.out.println(i);**

**}**

**String line = "";**

**while (!line.equals("Over")) {**

**try {**

**line = input.readLine();**

**out.writeUTF(line);**

**} catch (IOException i) {**

**System.out.println(i);**

**}**

**}**

**try {**

**input.close();**

**out.close();**

**socket.close();**

**} catch (IOException i) {**

**System.out.println(i);**

**}**

**}**

**public static void main(String args[]) {**

**Client client = new Client("127.0.0.1", 5000);**

**}**

**}**

**Server.java**

**package practical;**

**import java.net.\*;**

**import java.io.\*;**

**public class Server {**

**private Socket socket = null;**

**private ServerSocket server = null;**

**private DataInputStream in = null;**

**public Server(int port) {**

**try {**

**server = new ServerSocket(port);**

**System.out.println("Server started");**

**System.out.println("Waiting for a client ...");**

**socket = server.accept();**

**System.out.println("Client accepted");**

**in = new DataInputStream(new BufferedInputStream(socket.getInputStream()));**

**String line = "";**

**while (!line.equals("Over")) {**

**try {**

**line = in.readUTF();**

**System.out.println(line);**

**} catch (IOException i) {**

**System.out.println(i);**

**}**

**}**

**System.out.println("Closing connection");**

**socket.close();**

**in.close();**

**} catch (IOException i) {**

**System.out.println(i);**

**}**

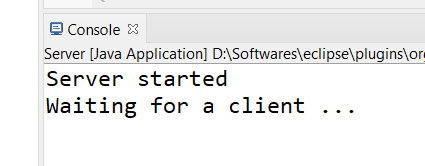
**}**

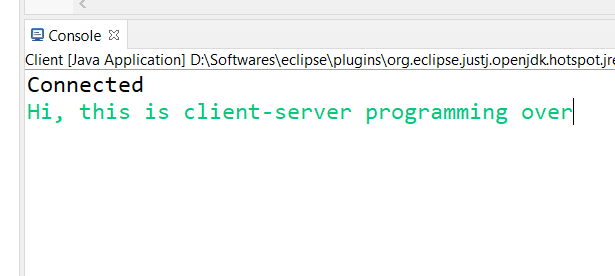
**public static void main(String args[]) {**

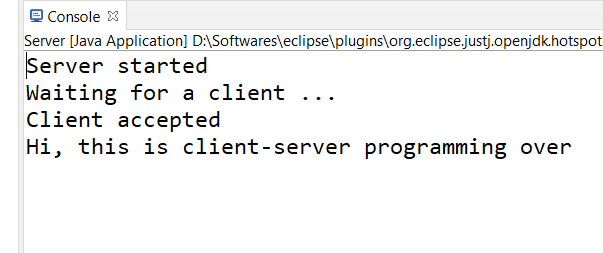
**Server server = new Server(5000);**

**}**

**}**







## CONCLUSION:

In this way we have studied the all methods of communication between client and server in java language. We implemented socket communication program by doing operations on Server side code and implemented RMI Technique for Client and Server communication.

## FAQs:

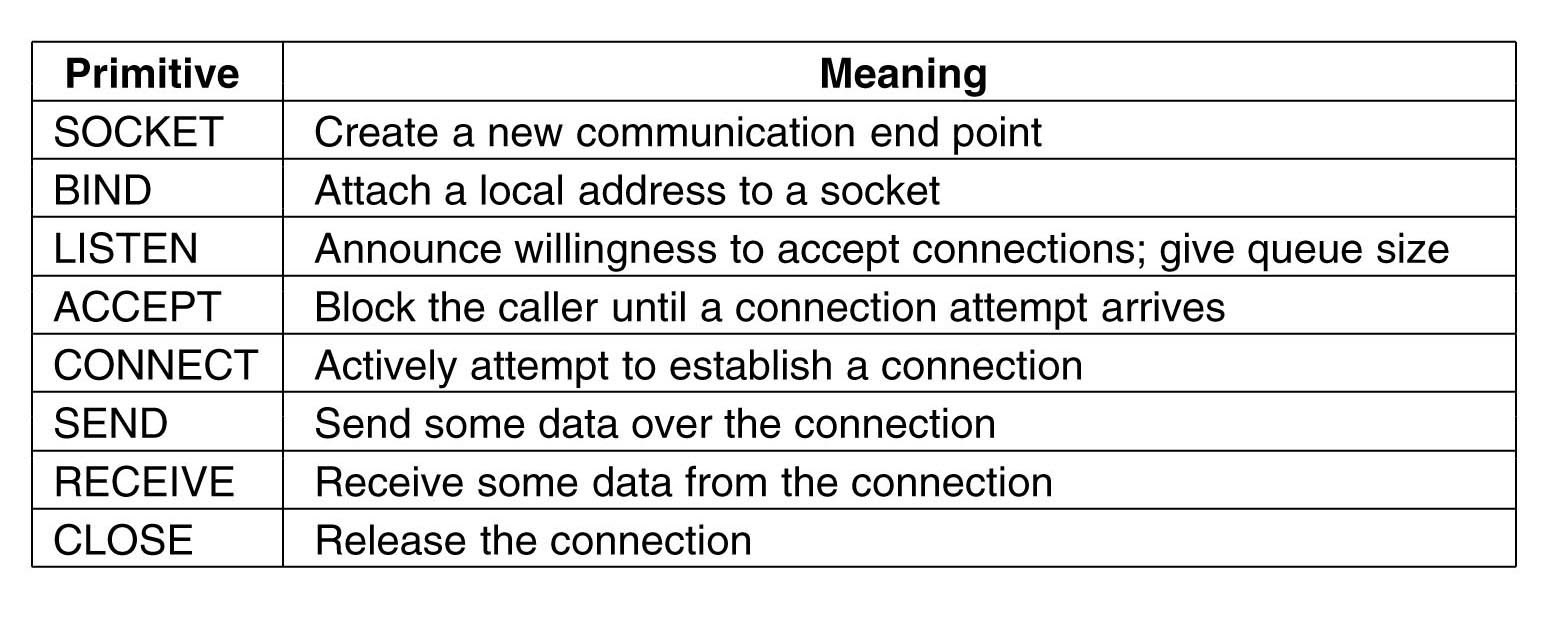
1. What is Socket?

A socket is **one endpoint of a two-way communication link between two programs running on the network**. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to. An endpoint is a combination of an IP address and a port number

1. What is the Process of Socket Client Server Communication?

Clients and servers **exchange messages in a request–response messaging pattern**. The client sends a request, and the server returns a response. This exchange of messages is an example of inter-process communication.

1. What are the Primitives of Socket at Client & Server Side?



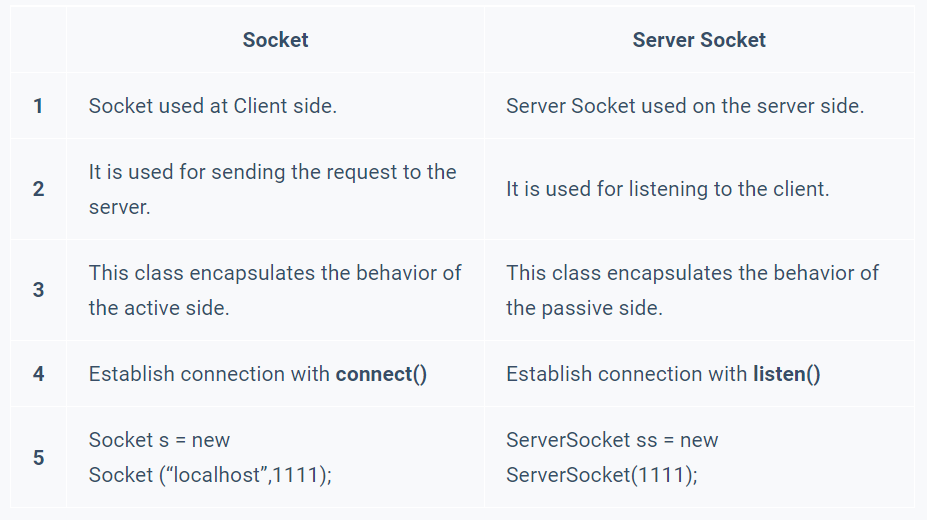
1. What is Use of Bind in Socket?

The bind() function **binds a unique local name to the socket with descriptor socket**. After calling socket(), a descriptor does not have a name associated with it. However, it does belong to a particular address family as specified when socket() is called. The exact format of a name depends on the address family.

1. What is Use of Listen in Socket?

Listen causes a **connection-oriented Socket to listen for incoming connection attempts**. The backlog parameter specifies the number of incoming connections that can be queued for acceptance. To determine the maximum number of connections you can specify, retrieve the MaxConnections value. Listen does not block.

1. What is difference between Socket & ServerSocket?



1. What is DataInputStream & DataOutputStream?

**DataInputStream:**

Class DataInputStream. A **data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way**. An application uses a data output stream to write data that can later be read by a data input stream.

**DataOutputStream:**

DataOutputStream in Java is **a filter output stream that provides methods for writing Java's standard data types**. It enables you conveniently to write strings and all primitive data types such as boolean, int, float, long, etc to a stream.

1. What is Thread ?

Thread series cover designations of diameter/pitch combinations that are measured by the number of threads per inch (TPI) applied to a single diameter. ... 8 – Thread Series (8UN) is **the specified thread forming method for several ASTM standards** including A193 B7, A193 B8/B8M, and A320.

1. What is RMI?

RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

1. **How RMI Works?**

Working of an RMI Application

The following points summarize how an RMI application works −

* When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
* When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.
* The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
* The result is passed all the way back to the client.

1. **What is Stub and Skeleton?**

The stub hides the serialization of parameters and the. network-level communication in order to present a simple invocation mechanism to the caller.

The skeleton is responsible for dispatching the call to the actual remote object implementation.

1. How to Implement Java RMI?

The RMI (Remote Method Invocation) is an API that **provides a mechanism to create distributed application in java**. The RMI allows an object to invoke methods on an object running in another JVM. The RMI provides remote communication between the applications using two objects stub and skeleton.

1. What is Marshaling & Unmarshaling?

Marshalling is the process of transforming the memory representation of an object to a data format suitable for the storage and transmission.

Unmarshalling refers to the process of transforming a representation of an object that is used for storage or transmission to a representation of the object that is executable.

1. What is RMI Registry?

rmiregistry & A remote object registry is **a bootstrap naming service** that is used by RMI servers on the same host to bind remote objects to names. Clients on local and remote hosts can then look up remote objects and make remote method invocations.