

UNIT-III

UNIT-III:

Distributed Objects and Remote Invocation: Introduction, Communication between Distributed Objects- Object Model, Distributed Object Model, Design Issues for RMI, Implementation of RMI, Distributed Garbage Collection; Remote Procedure Call, Events and Notifications, Case Study: JAVA RMI

Distributed Objects and Remote Invocation

Introduction:-

Method **invocations** between **objects** in the same process are local method **invocations**. **Remote objects:** **objects** that can receive **remote invocations**. Fundamental concepts of the **distributed object** model: ... ► Every **remote object** has a **remote** interface that specifies which of its methods can be invoked **remotely**.

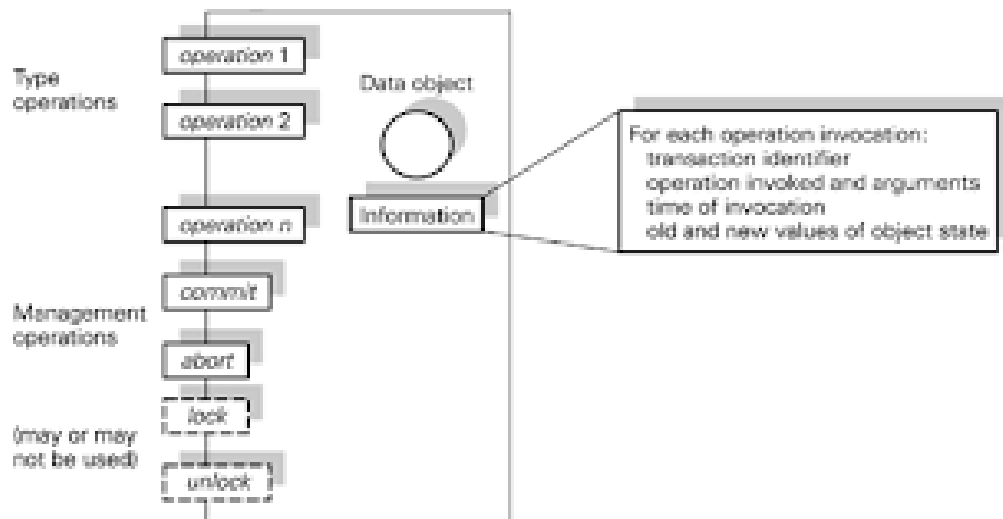
In a **distributed computing** environment, **distributed object communication** realizes **communication between distributed objects**. ...

Invoking a method on a remote **object** is known as remote method invocation (RMI) or remote invocation, and is the **object-oriented** programming analog of a remote procedure call (RPC).

Object Model:-

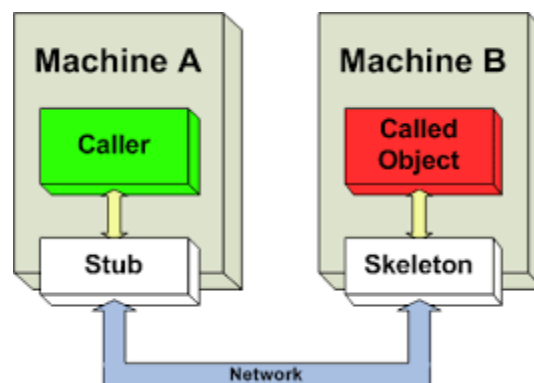
An **object model** for **distributed systems**. ... We assume each **object** is invoked through operations appropriate to its type.

The figure also shows some **object** management operations such as commit, abort, lock and unlock which may be needed for practical implementation of a transaction processing **system** (TPS).



Distributed Object Model:-

In **distributed computing**, **distributed objects** are **objects** (in the sense of **object-oriented programming**) that are **distributed** across different address spaces, either in different processes on the same computer, or even in multiple computers connected via a network, but which work together by sharing data and invoking ...



Design Issues for RMI:-

- Two design issues that arise in extension of local method invocation for RMI
 - o The choice of invocation semantics
- Although local invocations are executed exactly once, this cannot always be the case for RMI due to transmission error

- o Either request or reply message may be lost
- o Either server or client may be crashed
- o The level of transparency
- Make remote invocation as much like local invocation as possible

RMI Design Issues: Invocation Semantics

- Error handling for delivery guarantees
 - o Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed
 - o Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server
 - o Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations
- Choices of invocation semantics
 - o Maybe: the method executed once or not at all (no retry nor retransmit)
 - o At-least-once: the method executed at least once
 - o At-most-once: the method executed exactly once

Invocation semantics: choices of interest

<i>Fault tolerance measures</i>			<i>Invocation semantics</i>
<i>Retransmit request message</i>	<i>Duplicate filtering</i>	<i>Re-execute procedure or retransmit reply</i>	
No	Not applicable	Not applicable	<i>Maybe</i>
Yes	No	Re-execute procedure	<i>At-least-once</i>
Yes	Yes	Retransmit reply	<i>At-most-once</i>

RMI Design Issues: Transparency

- Transparent remote invocation: like a local call
 - o marshalling/unmarshalling
 - o locating remote objects
 - o accessing/syntax

·

Differences between local and remote invocations

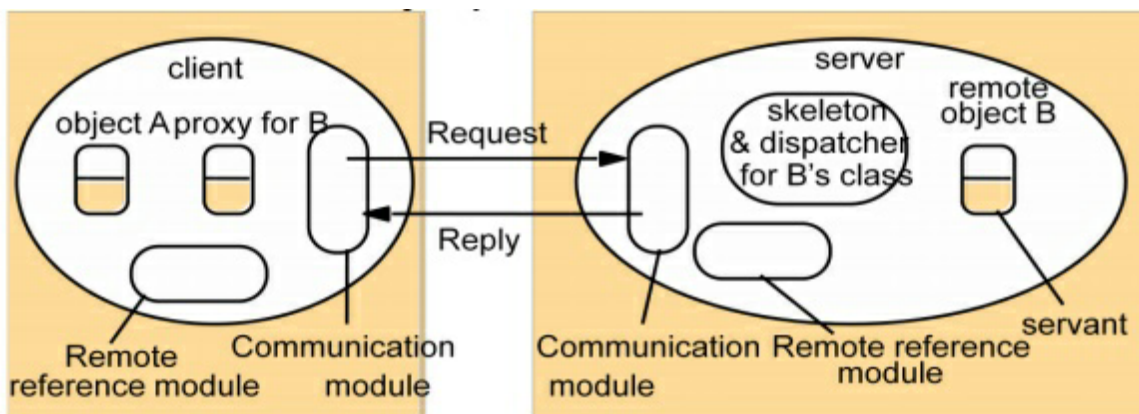
- o latency: a remote invocation is usually several order of magnitude greater than that of a local one
 - o availability: remote invocation is more likely to fail
 - o errors/exceptions: failure of the network? server? hard to tell
- syntax might need to be different to handle different local vs remote errors/exceptions (e.g. Argus)
 - o consistency on the remote machine:
- Argus: incomplete transactions, abort, restore states [as if the call was never made]

Implementation of RMI

- - Communication module
 - o Two cooperating communication modules carry out the request-reply protocols: message type, request ID, remote object reference

- Transmit request and reply messages between client and server
- Implement specific invocation semantics
 - o The communication module in the server
- selects the dispatcher for the class of the object to be invoked,
- passes on local reference from remote reference module,
- returns request

The role of proxy and skeleton in remote method invocation



Remote reference module

- o Responsible for translating between local and remote object references and for creating remote object references
- o remote object table: records the correspondence between local and remote object references
 - remote objects held by the process (B on server)
 - local proxy (B on client)
- o When a remote object is to be passed for the first time, the

module is asked to create a remote object reference, which it adds to its table

Servant

- An instance of a class which provides the body of a remote object
- handles the remote requests

• RMI software

- Proxy: behaves like a local object, but represents the remote object
- Dispatcher: look at the methodID and call the corresponding method in the skeleton
- Skeleton: implements the method

Generated automatically by an interface compiler

Implementation Alternatives of RMI:-

Dynamic invocation

Proxies are static— interface compiled into client code

Dynamic— interface available during run time

Generic invocation; more info in —Interface Repository (COBRA)

Dynamic loading of classes (Java RMI)

• Binder

A separate service to locate service/object by name through table mapping for names and remote object references

Activation of remote objects

- o Motivation: many server objects not necessarily in use all of the time
- § Servers can be started whenever they are needed by clients, similar to inetd
 - o Object status: active or passive
- § active: available for invocation in a running process
- § passive: not running, state is stored and methods are pending
 - o Activation of objects:
- § creating an active object from the corresponding passive object by creating a new instance of its class
- § initializing its instance variables from the stored state
 - o Responsibilities of activator
- § Register passive objects that are available for activation
- § Start named server processes and activate remote objects in them
- § Keep track of the locations of the servers for remote objects that it has already activated

Persistent object stores

- o An object that is guaranteed to live between activations of processes is called a
 - o persistent object
- o Persistent object store: managing the persistent objects
- o stored in marshaled form on disk for retrieval
- o saved those that were modified
 - o Deciding whether an object is persistent or not:
- o persistent root: any descendent objects are persistent (persistent Java, PerDiS)
- o some classes are declared persistent (Arjuna system)
- o Object location

- o specifying a location: ip address, port #, ...
- o location service for migratable objects
- o Map remote object references to their probable current locations

§ Cache/broadcast scheme (similar to ARP)

- o Cache locations
- o If not in cache, broadcast to find it
- o Improvement: forwarding (similar to mobile IP)

Distributed Garbage Collection:-

- Aim: ensure that an object
 - o continues to exist if a local or remote reference to it is still held anywhere
 - o be collected as soon as no object any longer holds a reference to it
- General approach: reference count
- Java's approach
 - o the server of an object (B) keeps track of proxies
 - o when a proxy is created for a remote object
- addRef(B) tells the server to add an entry
 - o when the local host's garbage collector removes the proxy
- removeRef(B) tells the server to remove the entry
 - o when no entries for object B, the object on server is deallocated

Remote Procedure Call (RPC):-

A remote procedure call is an interprocess communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call.

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's

details. A procedure call is also sometimes known as a function call or a subroutine call.

RPC uses the client-server model. The requesting program is a client and the service providing program is the server. Like a regular or local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned. However, the use of lightweight processes or threads that share the same address space allows multiple RPCs to be performed concurrently.

RPC message procedure

When program statements that use RPC framework are compiled into an executable program, a stub is included in the compiled code that acts as the representative of the remote procedure code. When the program is run and the procedure call is issued, the stub receives the request and forwards it to a client runtime program in the local computer.

Explanation of RemoteProcedure Call.

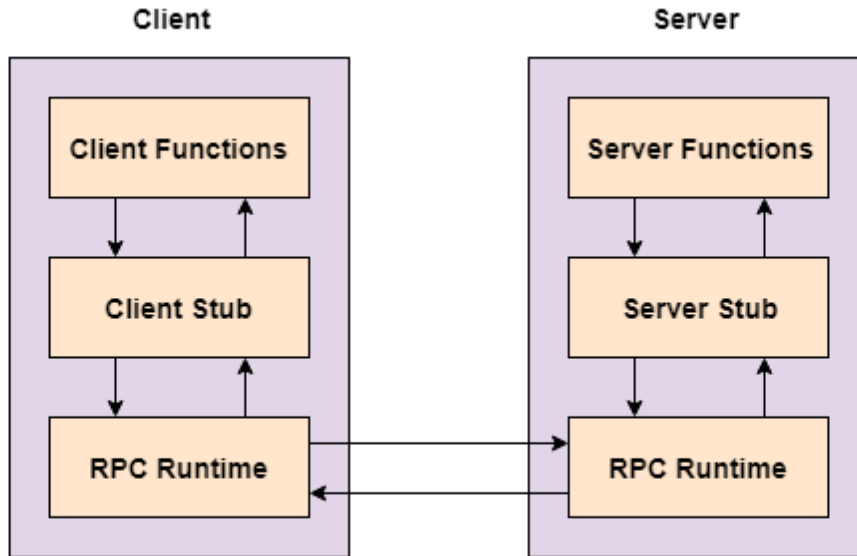
The client runtime program has the knowledge of how to address the remote computer and server application and sends the message across the network that requests the remote procedure. Similarly, the server includes a runtime program and stub that interface with the remote procedure itself. Response-request protocols are returned the same way.

A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished.

The sequence of events in a remote procedure call are given as follows:

- The client stub is called by the client.
- The client stub makes a system call to send the message to the server and puts the parameters in the message.
- The message is sent from the client to the server by the client's operating system.
- The message is passed to the server stub by the server operating system.
- The parameters are removed from the message by the server stub.
- Then, the server procedure is called by the server stub.

A diagram that demonstrates this is as follows:



Advantages of Remote Procedure Call

Some of the advantages of RPC are as follows:

- Remote procedure calls support process oriented and thread oriented models.
- The internal message passing mechanism of RPC is hidden from the user.
- The effort to re-write and re-develop the code is minimum in remote procedure calls.
- Remote procedure calls can be used in distributed environment as well as the local environment.
- Many of the protocol layers are omitted by RPC to improve performance.

Disadvantages of Remote Procedure Call

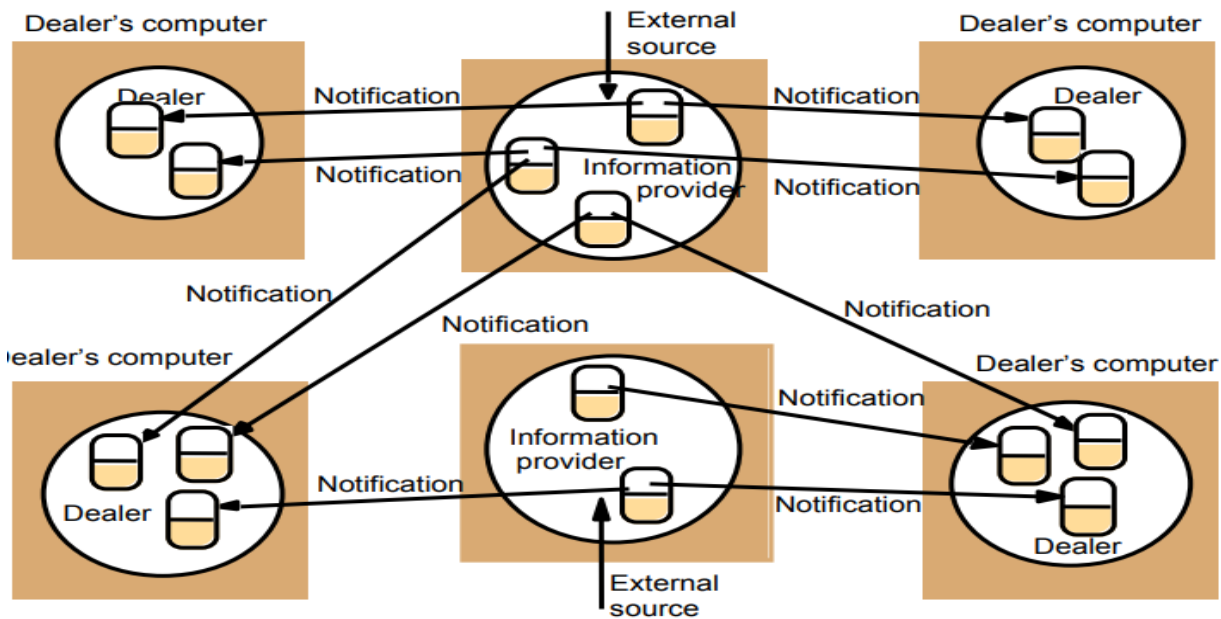
Some of the disadvantages of RPC are as follows:

- The remote procedure call is a concept that can be implemented in different ways. It is not a standard.
- There is no flexibility in RPC for hardware architecture. It is only interaction based.
- There is an increase in costs because of remote procedure call.

Events and notifications:-

- „ Distributed event-based systems extend local event model
 - ◁ Allowing multiple objects at diff. locations to be notified of events taking place at an object
 - ◁ Two characteristics: heterogeneous, asynchronous
- „ Such systems can be useful: stock dealing room
 - ◁ Allow dealers using computers to see latest stock prices
 - ◁ Information provider process: continuously receives new trading information from external source and applies to the stock object
 - ◁ Each such update is regarded as an event
 - ◁ The stock object notifies all the dealers who have subscribed
 - ◁ Dealer process: creates local object to represent stock to display, local object subscribes to stock object in info. provider process, receives notifications and display to user

Dealing Room System

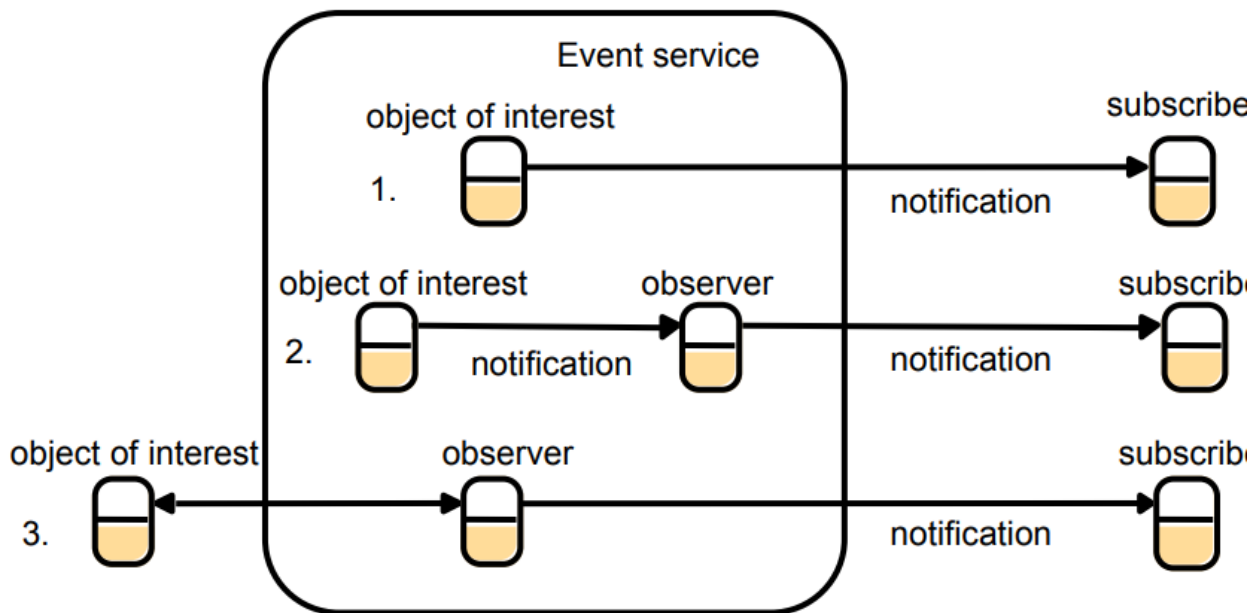


Publish-subscribe paradigm: publisher sends *notifications*, i.e. objects representing events

◁ Subscriber *registers interest* to receive notifications

- „ The object of interest: where events happen, change of state as a result of its operations being invoked
- „ Events: occurs in the object of interest
- „ Notification: an object containing information about an event
- „ Subscriber: registers interest and receives notifications
- „ Publisher: generate notifications, usually an object of interest
- „ Observer objects: decouple an object of interest from its subscribers (not important)

Architecture for distributed event notification



Case study : JAVA 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.

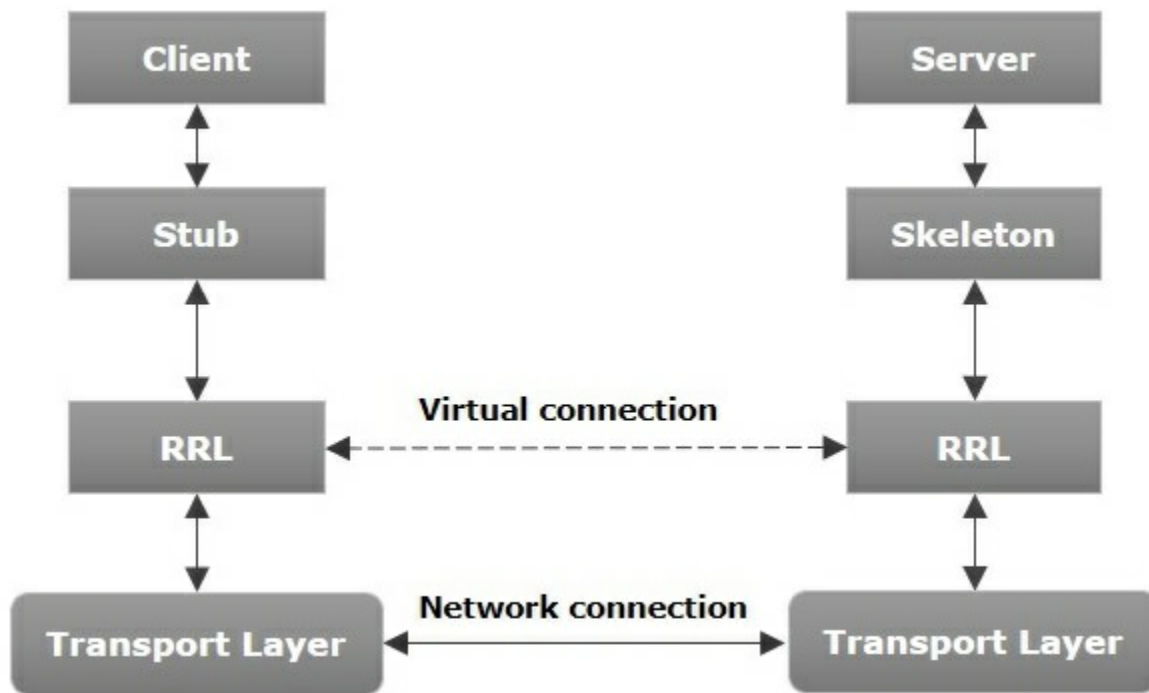
RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

Architecture of an RMI Application

In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

The following diagram shows the architecture of an RMI application.



Let us now discuss the components of this architecture.

- **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.

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.

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**.

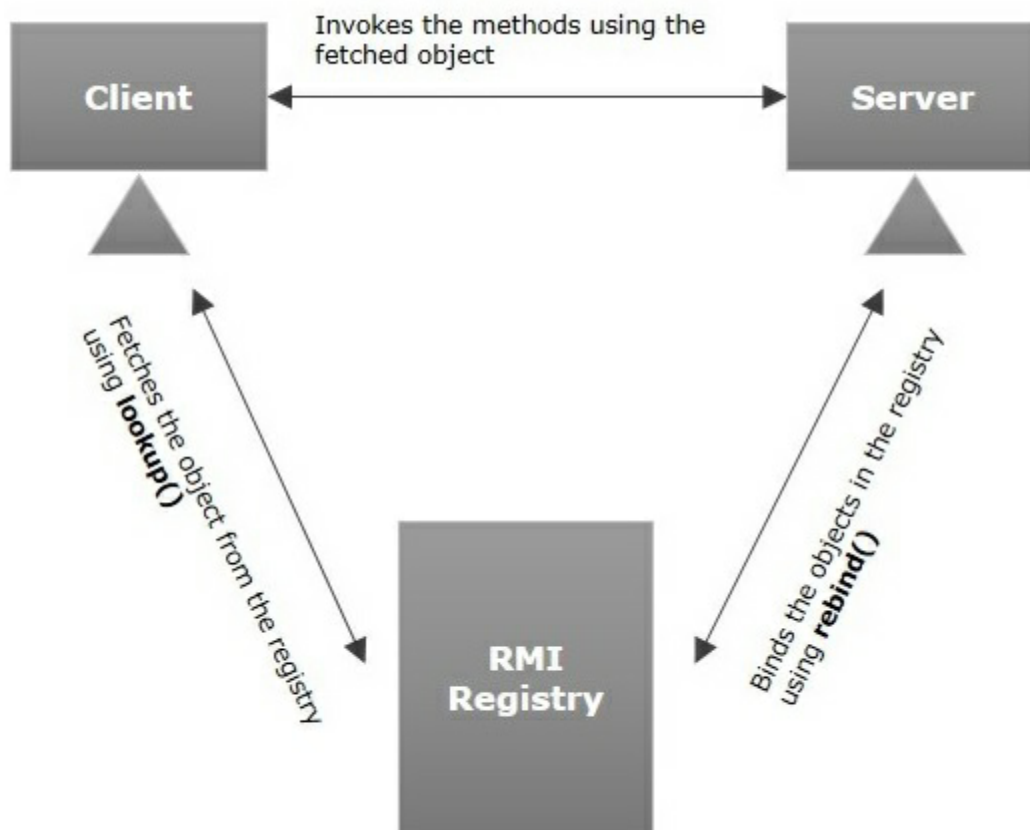
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

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 RMI registry (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).

The following illustration explains the entire process –



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.

Extends Java object model to support distributed objects

„ Allows objects to invoke methods in remote objects using the same syntax as for local invocations

- „ Distributed object model is integrated into Java naturally, but the semantics of parameter passing differ because invoker and target are remote
 - ◁ *Input* parameters and *output* parameters
- „ Single language advantage: remote interfaces are defined in Java, do not have to learn IDL (CORBA)

Shared whiteboard example:

- ◁ Allows a group of users to share a common view of a drawing surface containing graphical objects, such as rectangles, lines, circles.
- ◁ Each of the graphical objects is drawn by one of the users ◁ Server maintains current state of a drawing
 - provides an operation for clients to inform it about the latest shape
 - allows user to retrieve the latest shapes drawn by other users
 - allows enquiry about version number so as to avoid fetching shapes they already have
- each shape has a version number attached

Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*; //package providing Remote interface, must
throw RemoteException import java.util.Vector;
public interface Shape extends Remote { //Shape is a
    remote interface int getVersion() throws
    RemoteException; //returns version number
    GraphicalObject getAllState() throws RemoteException; //GraphicalObject
    serializable }
public interface ShapeList extends Remote { //ShapeList is a remote interface
    Shape newShape(GraphicalObject g) throws RemoteException; // return
    remote object Vector allShapes() throws RemoteException;
    int getVersion() throws
    RemoteException; }
```

Both ordinary and remote objects can be passed as arguments and results

◁ Ordinary object (*GraphicalObject*) is passed by value: all non-remote objects

New object is created in the receiver's process, whose methods can be invoked locally,

possibly having states changed

◁ Remote object (***Shape***) is passed by reference (remote object reference),

on which remote methods can be invoked

newShape: client uses the remote object reference for a remote object of

ShapeList, calls its method *newShape*, passes to it a *GraphicObject* instance, the

server makes a remote object of type *Shape* containing the state of the graphical object and return a remote object reference to it... so that ...

Downloading of classes: Java is designed to allow classes to be downloaded from one virtual machine to another

Non-remote object: passed by value. If the recipient does not already have the class of the object, its code is downloaded automatically

Remote object: passed by reference. If the recipient does not already have the class for a proxy, its code is downloaded automatically

Recall, we need a proxy as the local representative for each remote object such that remote methods can be invoked

And, proxy implements the remote interface of the remote object, for local invokers, the proxy looks as if it is the remote object

Advantage:

- (1) No need for every user to keep the same set of classes in their working environment
- (2) both client and server can make transparent use of instances of new classes (automatically)

Building client and server programs

server program

main program: binding instances of servant classes

main method needs to create a security manager to enable Java security. A default security manager, `RMISecurityManager`, is provided

Note: if an RMI server sets no security manager, proxies and classes can only be loaded from the local classpath, in order to protect the program from code that is downloaded as a result of remote method invocations.

servant classes: `ShapeListServant` and `ShapeServant`, implementing `ShapeList` and `Shape` interfaces respectively

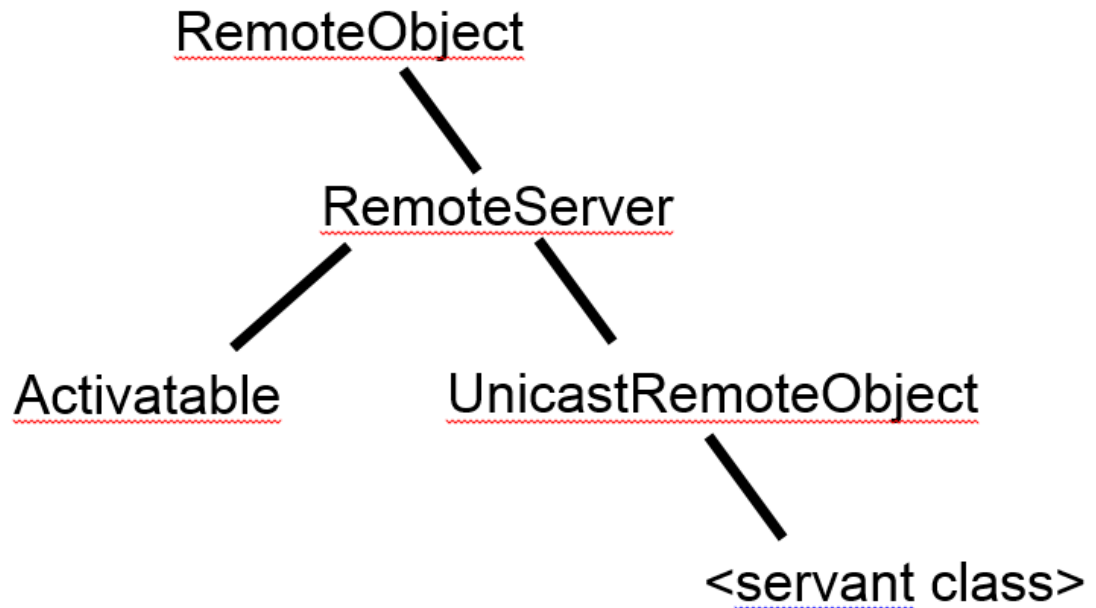
servant classes need to extend `UnicastRemoteObject`, which provides remote object that live only as long as the process in which they are created

implementation of servant classes are straightforward, no

concern of communication details

Classes supporting Java RMI:

Every single servant class needs to extend UnicastRemoteObject



- ④ Callbacks: server's action of notifying clients about an event
Client creates a remote object with method for the server to call

callback object

Server provides an operation allowing interested client to inform it of the remote object references of their callback objects

When event occurs, the server notifies the interested client

Use of reflection: used to make a generic dispatcher and to avoid the need for skeletons.

Deprecated. *Skeletons are no longer required for remote method calls in the Java 2 platform v1.2 and greater.*

public interface **Skeleton**

The Skeleton interface is used solely by the RMI implementation...

Client proxies are generated by a compiler called *rmic* from the compiled server classes

Reflection: the class of an object can be determined at runtime, and this class can be examined to determine which methods are available, and even invoke these methods with dynamically created arguments

The key to reflection is the *java.lang.Class*, which allows much information to be determined about a class. This leads onto the other reflection classes such as *java.lang.reflect.Method*. For instance, one can:

- Determine the class of an object.

- Get information about a class's modifiers, fields, methods, constructors, and superclasses.

- Find out what constants and method declarations belong to an interface. Create an instance of a class whose name is not known until runtime.

- Get and set the value of an object's field, even if the field name is unknown to your program until runtime.

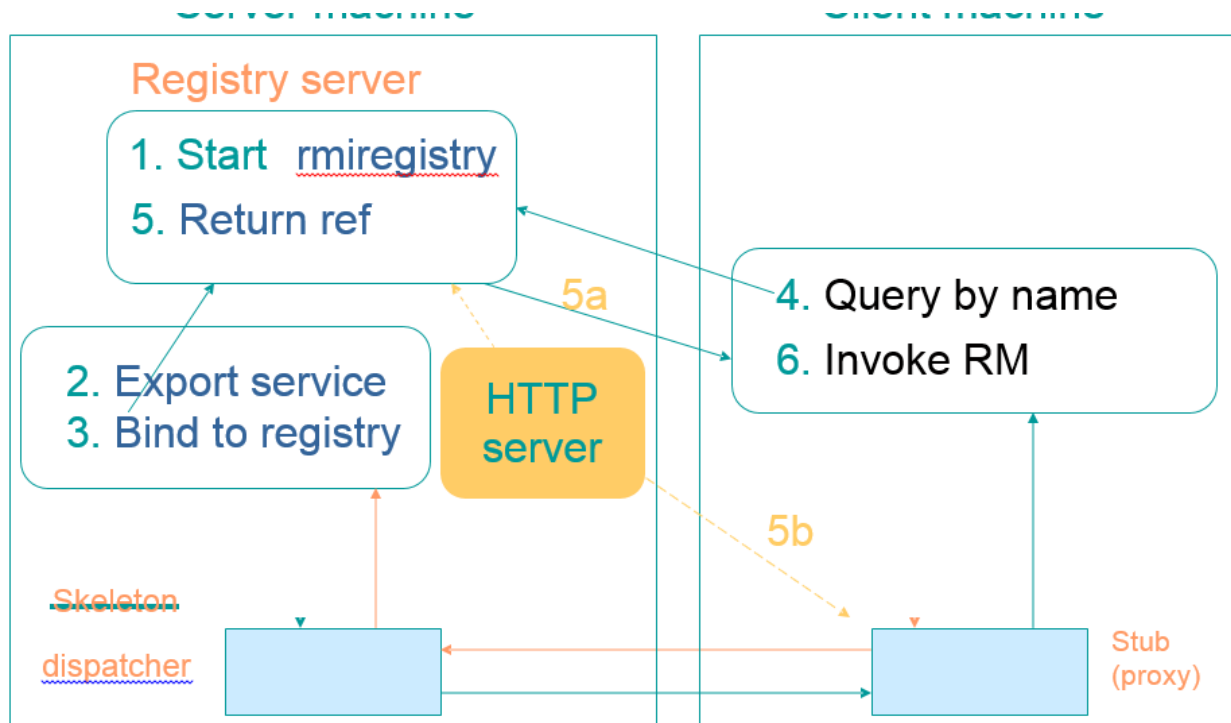
- Invoke a method on an object, even if the method is not known until runtime.

- Create a new array, whose size and component type are not known until runtime, and then modify the array's components.

Java Registry Overview:-

Client machine

Server machine



Heterogeneity is an important challenge to designers :

Distributed systems must be constructed from a variety of different networks, operating systems, computer hardware and programming languages.

The Internet communication protocols mask the difference in networks and middleware can deal with the other differences.

External data representation and marshalling

CORBA marshals data for use by recipients that have prior knowledge of the types of its components. It uses an IDL specification of the data types

Java serializes data to include information about the types of its contents, allowing the recipient to reconstruct it. It uses reflection to do this.

RMI

each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely.

local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once

Middleware components (proxies, skeletons and dispatchers) hide details of marshalling, message passing and object location from programmers.