Syllabus: Distributed Objects and Remote Invocation: Introduction, Communication between Distributed Objects- Object Model, Distributed Object Modal, 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 communication between distributed objects

Distributed objects[are objects that are distributed across different address spaces, either in multiple computers connected via a network or even indifferent processes on the same computer, but which work together by sharing data and invoking methods. This often involves location transparency, where remote objects appear the same as local objects.

The main method of distributed object communication is with remote method invocation

Invoking a method on a remote object is known as **remote method invocation**) generally by message-passing

Message-passing: one object sends a message to another object in a remote machine or process to perform some task. The results are sent back to the calling object.

The remote procedure call (RPC) approach extends the common programming Abstraction of the procedure call to distributed environments, allowing a calling Process to call a procedure in a remote node as if it is local.
Remote method invocation (RMI) is similar to RPC but for distributed objects,with
Added benefits in terms of using object-oriented programming concepts in Distributed systems and also extending the concept of an object reference to the Global distributed environments, and allowing the use of object references as Parameters in remote invocations
Remote procedure call – client calls the procedures in a server program that is running in a different process

Remote method invocation (RMI) – an object in one process can invoke methods of objects in another process

Event notification – objects receive notification of events at other objects for which they have registered
Middleware Roles

provide high-level abstractions such as RMI enable location transparency free from specifics of communication protocols

operating systems and communication hardware

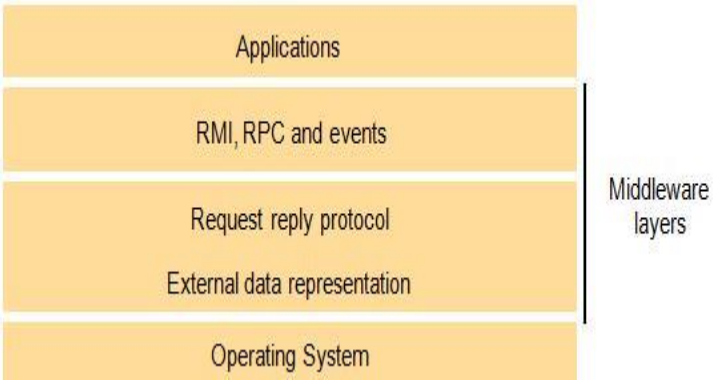


Fig middle ware layer

Communication between distributed objects and other objects

Life cycle : Creation, migration and deletion of distributed objects is different from local objects

Reference : Remote references to distributed objects are more complex than simple pointers to memory addresses

Request Latency : A distributed object request is orders of magnitude slower than local method invocation

Object Activation : Distributed objects may not always be available to serve an object request at any point in time

Parallelism: Distributed objects may be executed in parallel.

Communication : There are different communication primitives available for distributed objects requests

Failure: Distributed objects have far more points of failure than typical local objects.

Security: Distribution makes them vulnerable to attack.

Distributed object model:

The term **distributed objects** usually refers to software modules that are designed to work together, but reside either in multiple computers connected via a network or in different processes inside the same computer.

Distributed objects

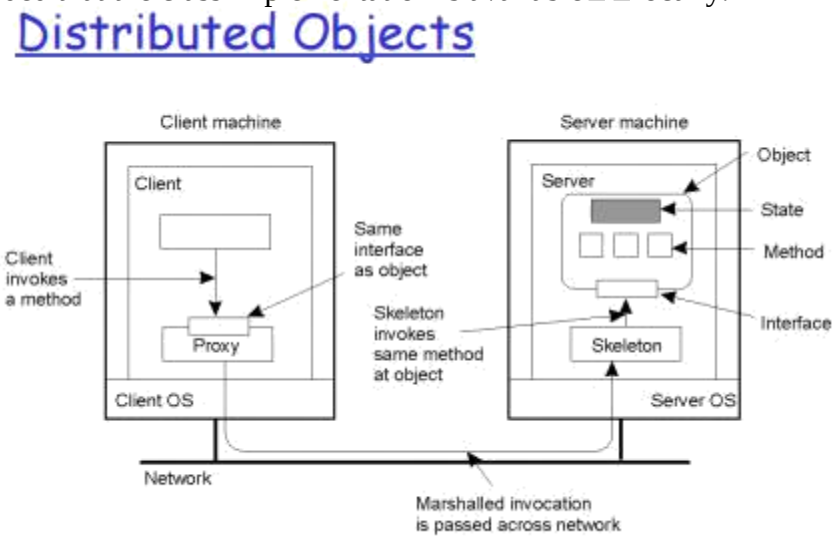
The state of an object consists of the values of its instance variables since object-based programs are logically partitioned, the physical distribution of objects into different

processes or computers in a distributed system. Distributed object systems may adopt the client-server architecture. objects are managed by servers and their clients invoke their methods using remote method invocation.

In RMI, the client‘s request to invoke a method of an object is sent in a message to the server managing the object. The invocation is carried out by executing a method of the object at the server and the result is returned to the client in another message

Distributed objects can assume other architectural models. For example, objects can be replicated in order to obtain the usual benefits of fault tolerance and enhanced performance, and objects can be migrated with a view to enhancing their performance and availability.

Another advantage of treating the shared state of a distributed program as a collection of objects is that an object may be accessed via RMI, or it may be copied into a local cache and accessed directly, provided that the class implementation is available locally.



Design issues of RMI

RMI Invocation Semantics:

Invocation semantics depend upon implementation of Request Reply Protocol used by RMI

It maybe, used At-least-once, At-most-once
Transparency:

Partial failure, higher latency, Different semantics for remote objects,

For e.g. wait/notify Current consensus: remote invocations should be made transparent in the sense that syntax of a remote invocation is the same as the syntax of local invocation (access

transparency) but programmers should be able to distinguish between remote and local objects by looking at their interfaces, e.g. in Java RMI, remote objects implement the Remote interface

Issues in implementing RMI

Parameter passing

Request reply protocol (handling failures at client and server)

Supporting constant objects, object adapters, dynamic invocations, etc

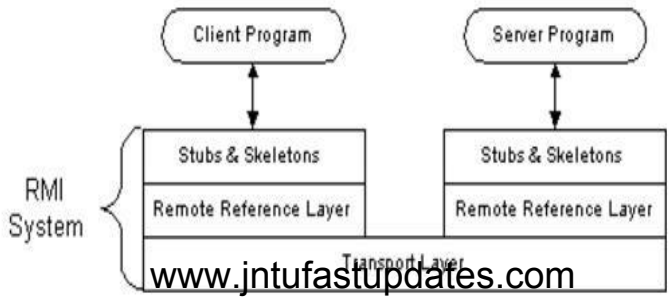
The design goal for the RMI architecture was to create a Java distributed object model that integrates naturally into the Java programming language and the local object model. RMI architects have succeeded; creating a system that extends the safety and robustness of the Java architecture to the distributed computing world.

The RMI architecture is based on one important principle: the definition of behavior and the implementation of that behavior are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.
This fits nicely with the needs of a distributed system where clients are concerned about the definition of a service and servers are focused on providing the service.
Specifically, in RMI, the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class.

Therefore, the key to understanding RMI is to remember that interfaces define behavior and classes

Implementation of RMI:

The RMI implementation is essentially built from three abstraction layers. The first is the Stub and Skeleton layer, which lies just beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.
The next layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one link. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via *Remote Object Activation*.
The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.



Distributed Garbage collection

Distributed garbage collection (DGC) in computing is a particular case of **garbage collection** where references to an object can be held by a remote client.

One of the joys of programming for the Java platform is not worrying about memory allocation. The JVM has an automatic garbage collector that will reclaim the memory from any object that has been discarded by the running program.
One of the design objectives for RMI was seamless integration into the Java programming language, which includes garbage collection. Designing an efficient single-machine garbage collector is hard; designing a distributed garbage collector is very hard.
The RMI system provides a reference counting distributed garbage collection algorithm based on Modula-3's Network Objects.
This system works by having the server keep track of which clients have requested access to remote objects running on the server. When a reference is made, the server marks the object as "dirty" and when a client drops the reference; it is marked as being "clean."

DGC uses some combination of the classical garbage collection (GC) techniques, tracing and reference counting. It has to cooperate with local garbage collectors in each process in order to keep global counts, or to globally trace accessibility of data.

In general, remote processors do not have to know about internal counting or tracing in a given process, and the relevant information is stored in interfaces associated with each process.

DGC is complex and can be costly and slow in freeing memory. One cheap way of avoiding DGC algorithms is typically to rely on a time lease set or configured on the remote object; it is the stub's task to periodically renew the lease on the remote object.

If the lease has expired, the server process (the process owning the remote object) can safely assume that either the client is no longer interested in the object, or that a network partition or crash obstructed lease renewal, in which case it is "hard luck" for the client if it is in fact still interested.

Hence, if there is only a single reference to the remote object on the server representing a remote reference from that client, that reference can be dropped, which will mean the object will be garbage collected by the local garbage collector on the server at some future point in time.

Distributed systems typically require distributed garbage collection. If a client holds a proxy to an object in the server, it is important that the server does not garbage-collect that object until the client releases the proxy. Most third-party distributed systems, such as RMI, handle the distributed garbage collection, but that does not necessarily mean it will be done efficiently. The overhead of distributed garbage collection and remote reference maintenance in RMI can slow network communications by a significant amount when many objects are involved.

Of course, if you need distributed reference maintenance, you cannot eliminate it, but you can

reduce its impact. You can do this by reducing the number of temporary objects that may have

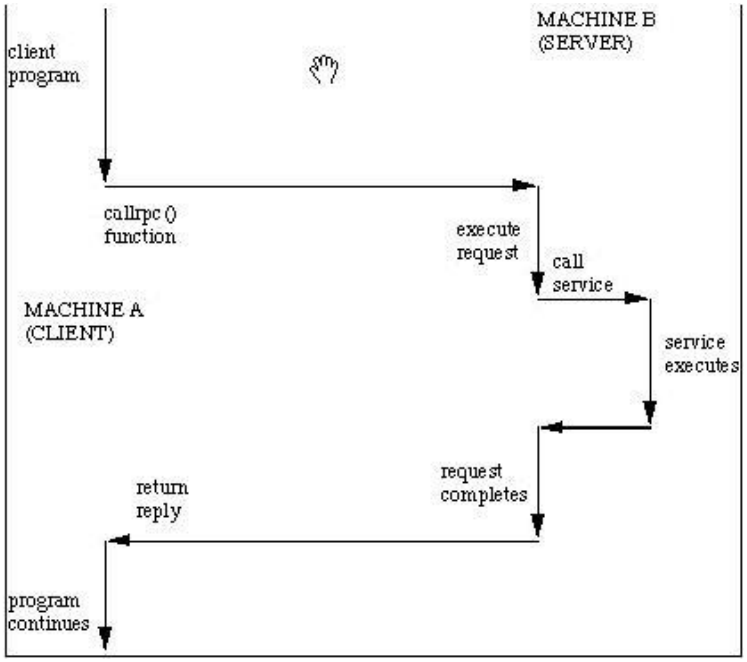
distributed references. The issue is considerably more complex in a multiuser distributed environment, and here you typically need to apply special optimizations related to the products you use in order to establish your multiuser environment.

Remote Procedure Call (**RPC**) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. (A procedure call is also sometimes known as a function call or a subroutine call.) **RPC** uses the client/server model.

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure.

The flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out.

When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.



Remote Procedure Calling Mechanism A remote procedure is uniquely identified by the triple: (program number, version number, procedure number) the program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available simultaneously. Each version contains a number of procedures that can be called remotely. Each procedure has a procedure number.

Events and notification

Events of changes/updates...

notifications of events to parties interested in the events

publish events to send

subscribe events to receive

main characteristics in distributed event-based systems:

a way to standardize communication in heterogeneous systems (not designed to communicate directly)

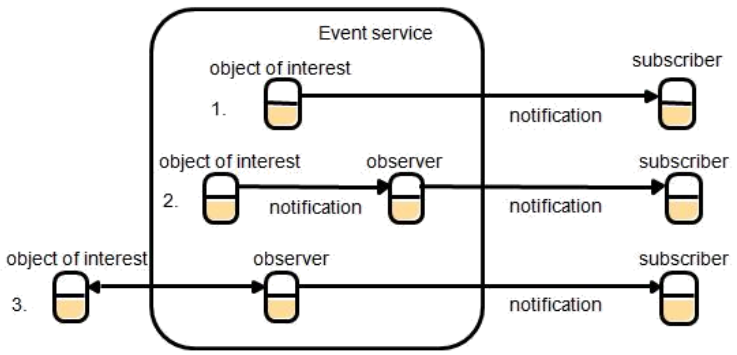
asynchronous communication (no need for a publisher to wait for each subscriber-- subscribers come and go)

event types

each type has attributes (information in it)

subscription filtering: focus on certain values in the attributes (e.g. "buy" events, but only "buy car" events)

Events and Notifications (4):



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)

Case study JAVA RMI

server program main program: binding instances of servant classes

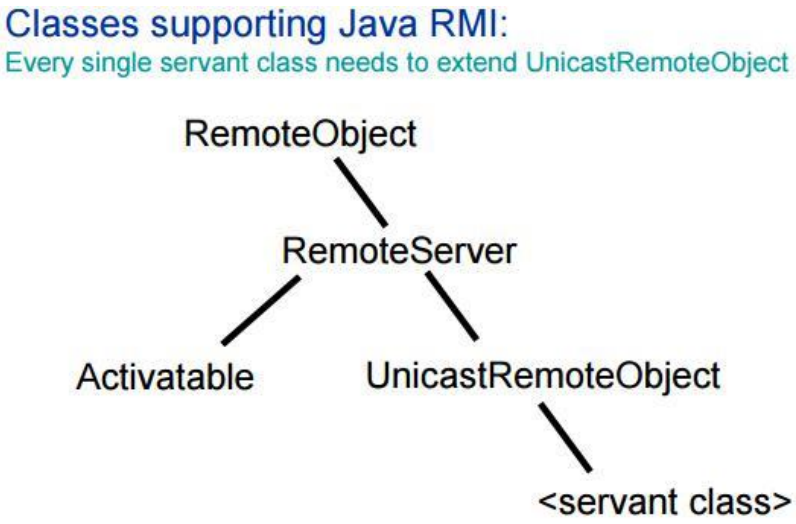
main method needs to create a security manager to enable Java security. A default security manager, RMISecurityManager, is provided

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: ShapeList Servant and Shape Servant, 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



UnicastRemote Object:

automatically creates socket and listens for network requests, and make its services available by exporting them.

RMISecurityManager (): Needed to download objects from network. The downloaded objects are allowed to communicate only with sites they came from.

Default security manager, when none is explicitly set, allows only loading from local file system

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

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