Remote Method Invocation (RMI)

Agenda

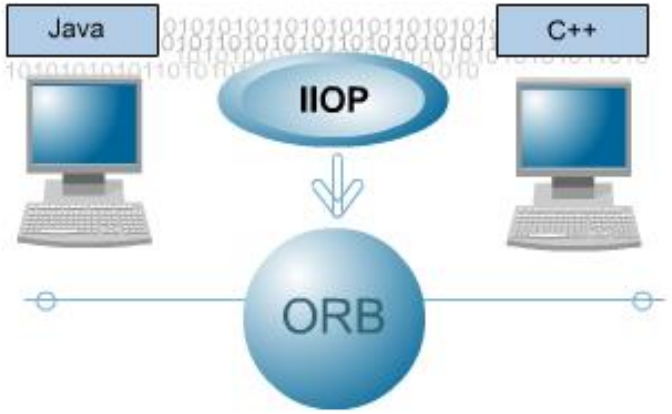
- Introducing RMI
- Using RMI
- Running an RMI application
- RMI communication
- The RMI registry
- Serialization
- Remote method calls
- JNDI

Introducing RMI

- When you create a standalone Java application, its objects can call methods on other objects running in the same Java Virtual Machine (VM).
- However, if you are building an enterprise application, you may need to create a Java application that's distributed across more than one networked computer.
- To do this, you need to write code that can access objects running in a different Java VM.
- A straightforward way to build a Java application is to use Java's Remote Invocation (RMI) system.
- RMI allows you to easily write applications in which your objects can call methods on objects in another Java VM. In RMI this is known as a remote VM, and the objects of a remote VM are known as remote objects.
- RMI was originally designed only to use a Java-to-Java protocol called JRMP (Java Remote Method Protocol) – in other words, to communicate between Java applications.
- However, it's now possible to use RMI to communicate between Java and other languages such as C++, because RMI can now run over a protocol called IIOP (Internet Inter-ORB Protocol) that allows access to an ORB (Object Request Broker).

Introducing RMI

- RMI can work directly with Java objects, which allows an application to easily use an application on a remote machine.
- It also means that your distributed applications can use other features of the Java platform, such as garbage collection and Java security.
- Since Java Developer Toolkit (JDK) 1.1, RMI has been available to developers as part of the core Java Application Programming Interface (API).
- The relevant classes are in the package java.rmi and its subpackages.



Introducing RMI

- The RMI logical architecture is partitioned into the following RMI packages:

java.rmi	The java.rmi package is the core package for RMI containing the remote interface, a few key classes, and many standard exceptions.
java.rmi.server	The java.rmi.server package is the core package for server side RMI interfaces, classes, and interfaces.
java.rmi.registry	The Java.rmi.registry package provides a class and an interface for interfacing with the RMI lookup service.
java.rmi.activation	The java.rmi.activation package provides the interfaces and classes needed to implement server objects that can be activated by client requests.
java.rmi.dgc	The java.rmi.dgc package provides the classes and interface for performing distributed garbage collection.
javax.rmi	The javax.rmi package currently contains a class for implementing portable RMI/IIOP objects.
javax.rmi.corba	The javax.rmi.corba package contains all the CORBA-specific classes and interfaces for implementing and using RMI/ IIOP objects.

Using RMI

- Let's look at a simple example that uses RMI to communicate between two VMs.
- The application consists of
  - an RMI server application that provides a remote object called simple server
  - an RMI client application that calls methods on the remote object

Sequence of Steps

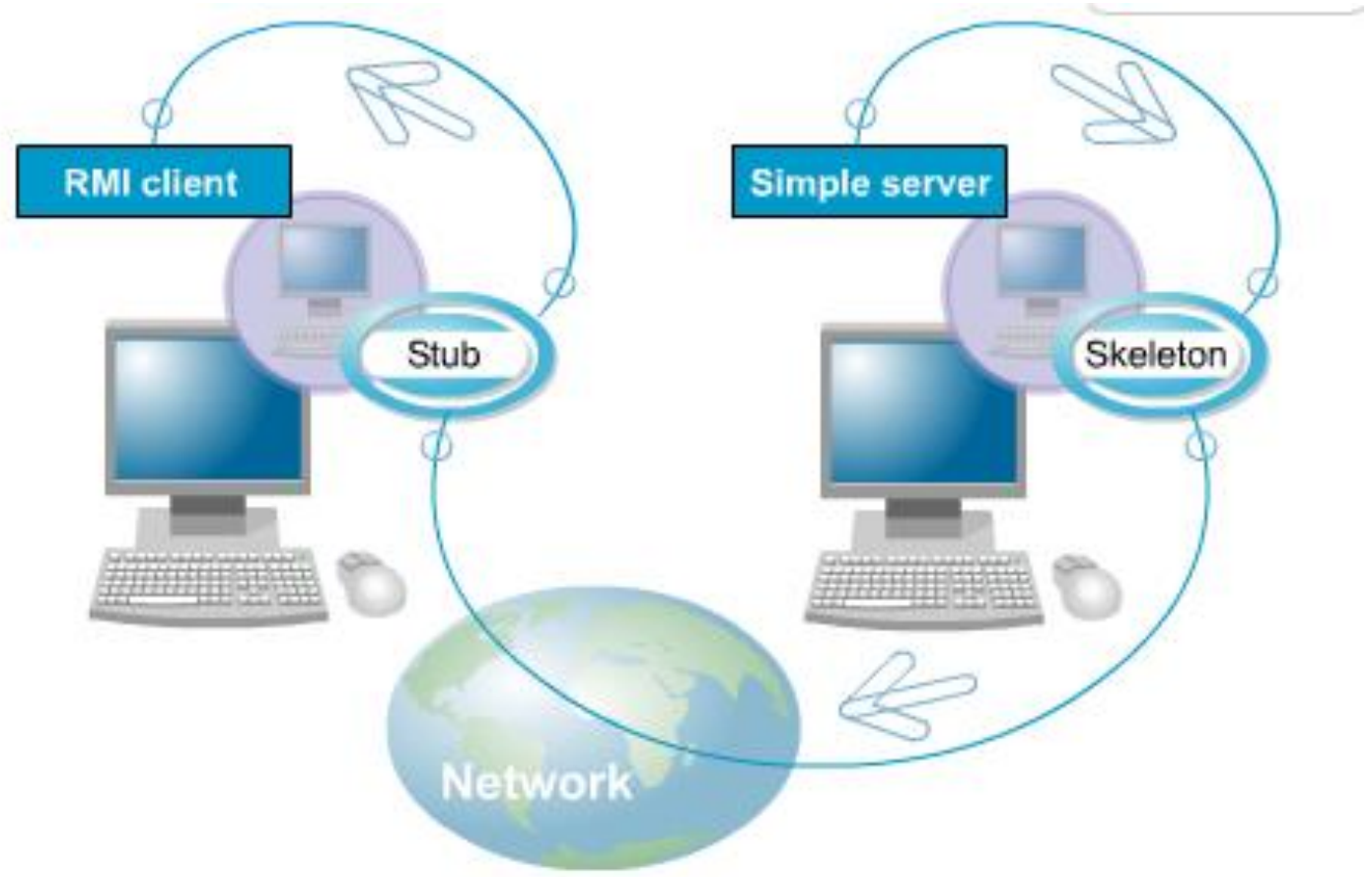
- To successfully run this RMI application (that is, read time remotely), do the following:

1. Define a remote interface, in this example, DateServer.java
2. Create and compile a class that implements the interface, DateServerImpl.java
3. Create stub (for the client) and skeleton (for the server) classes
   - *rmic DateServerImpl*
4. Start the registry (if it hasn't been started yet) by executing the following command
   - *rmiregistry &*
5. Run the server, DateServerImpl, with the following command
   - *java DateServerImpl &*
6. Implement the client, DateClient.java
7. Create, compile and run the client, DateClient, with the following command
   (localhost can be replaced by the machine on which the server runs)
   - *java DateClient localhost*

Running an RMI application

- Let's examine how the DateServerImpl - RMI application works.
  - As you know, when you run this application, the DateClient object appears to obtain a reference to the DateServerImpl object and use its getDate() method, just as if they were in the same VM.
  - However, the DateClient does not communicate directly with the DateServerImpl . Instead the DateClient gets a reference to a stub object from the registry
  - A stub object implements the same remote interfaces as the remote object, so the client application can call methods on it as if it were the required remote object.
  - Stub classes are generated from the relevant server implementation class using the RMI stub/skeleton compiler included in the JDK.
  - When the SimpleClient object wants to call the SimpleServer query method, the method is actually sent to the stub object.
  - The stub object then transmits the message over the network to the remote VM.
  - On the server side, the message is received by a skeleton object that corresponds to the client-side stub object.
  - Like a stub class, a skeleton class is generated from the server implementation class.
  - The skeleton object passes the message to the actual server implementation to invoke the method.
  - Any return values go back to the client in the same way, via the skeleton and the stub.
  - With the Java SE version of RMI, you do not need to use skeletons on the server side if all parts of the distributed application use Java SE.

Running an RMI application

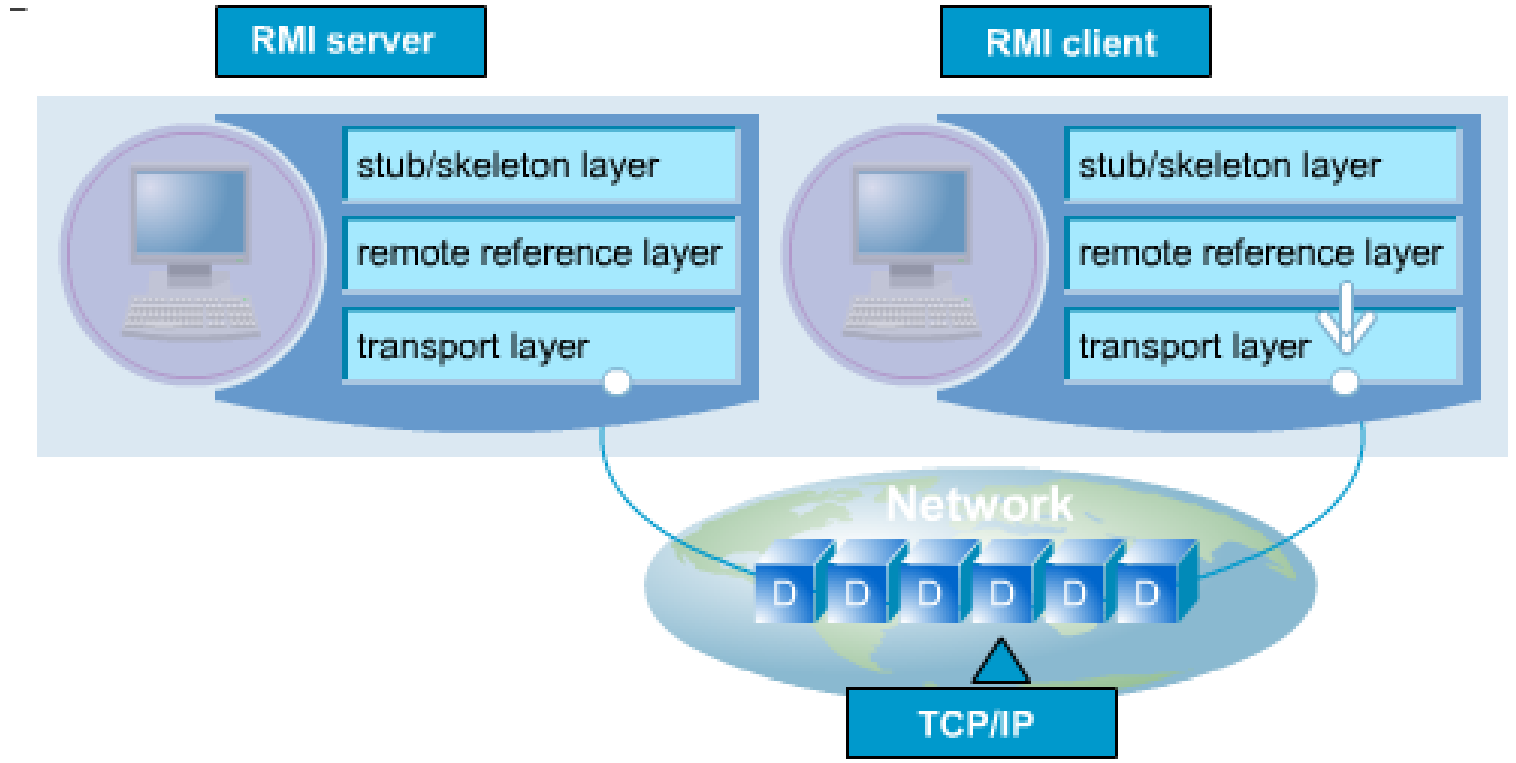


RMI communication

- RMI's communication between VMs is carried out using a three-layered model.
- These layers hide the low-level transport mechanism from the client and server applications.
- The RMI-commuication model is made up of the following:

stub/skeleton layer	The first layer of the RMI-communication model is the stub/skeleton layer. This layer includes the stub or skeleton code that your client and server applications interact with directly. When a stub receives a client message, it packages the message and arguments into a form that can be transmitted across the network. This is known as marshaling.
remote reference layer	The stub passes the data to the remote reference layer. The remote reference layer handles the conversion of the local reference to the stub class to a remote reference to the server object.
transport layer	Finally, the data is passed to the transport layer. This layer deals with opening connections, connection management, and the actual movement of data over the network.

RMI communication



Note: In a Java SE-only environment, RMI libraries unmarshal the message and arguments. In applications using earlier versions of RMI, the unmarshaling is managed by a skeleton object. The method can then be invoked on the server object.

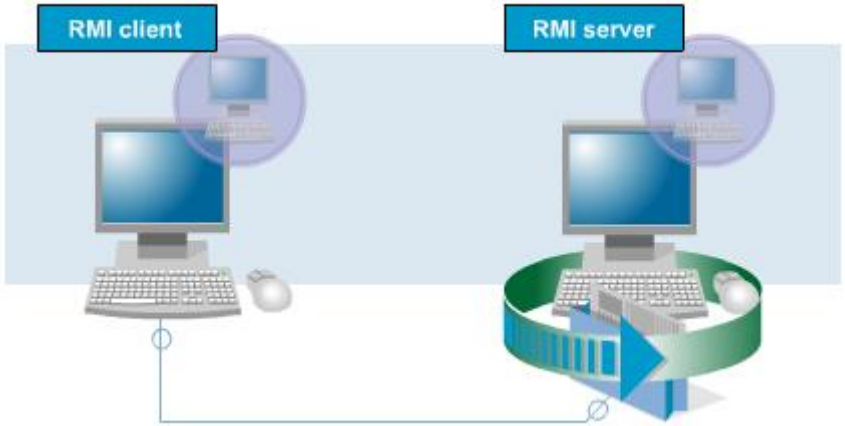
The RMI registry

- In order for an RMI application to work, the server application needs to advertise the location of its objects.
- And the client application needs to be able to obtain references to the stubs that allow it to communicate with those server objects. These services are provided in RMI by the RMI registry
- In an RMI application, the registry runs on the server machine.
- A single RMI registry can be shared by all the server processes running on a particular machine, or each server process can have its own registry.
- The registry can be accessed remotely by the client machine or by a client application on the same machine as the server.
- You can start an RMI registry from the command line using the start rmiregistry command.
  - start rmiregistry
- You can also start a registry in a server application's code using the java.rmi.registry.LocalRegistry class's createRegistry method.
  - public static Registry createRegistry (int port);
- When an RMI registry is running, it listens at a specified port for server and client messages. The registry listens at port 1099 by default.
- Both the server and the client can access the registry's naming services in their code using the static methods provided in the java.rmi.Naming class.

The RMI registry

- When you want to make a server object available to a client, you must register – or bind – the object with the locally running registry.
- To do this, you use the Naming class's bind method to provide the registry with a reference to the object together with a unique URL-formatted string. The string acts as a name that clients can use to locate the object.
  - public static void bind (string name, Object object);
- When a client wants to obtain a reference to the remote object – which is really a local reference to a stub – it uses the Naming class's lookup method.
  - public static Remote lookup (String name);
- This allows the client to connect to the registry and pass it a string name. If there is no object currently bound to that name, the registry will return a NotBoundException. Otherwise, the registry returns a reference to the relevant stub.

The RMI registry



Serialization

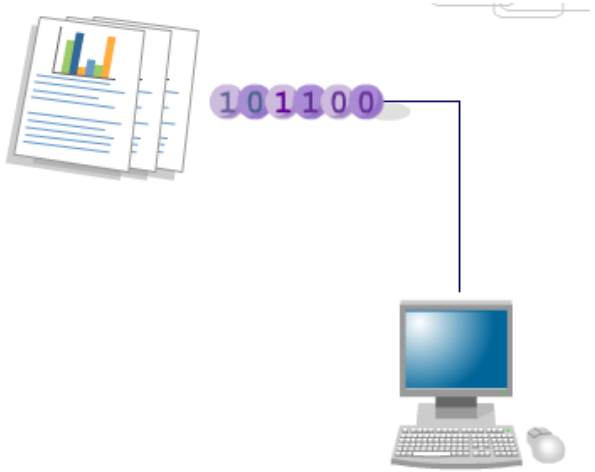
- The stub/skeleton layer in Remote Method Invocation (RMI) is responsible for marshalling method parameters and return values into a form that can be transmitted across a network.
- To do this, it has to convert the parameters or return values into a stream of bytes. And the stub/skeleton layer must be able to retrieve the parameters or return values – including the state of any objects – from the stream when it receives it. This process is carried out by object serialization.
- Java makes it easy to write and read primitive types such as ints or bytes to and from data streams – when writing to a file, for example.
- Although RMI automatically serializes objects, you can also explicitly serialize objects – if you want to save them to a file, for example.

Serialization

- There are many instances where you might want to use serialization to store data in a file. Let's suppose you need to store an application's state data so it's available the next time you run your program application.

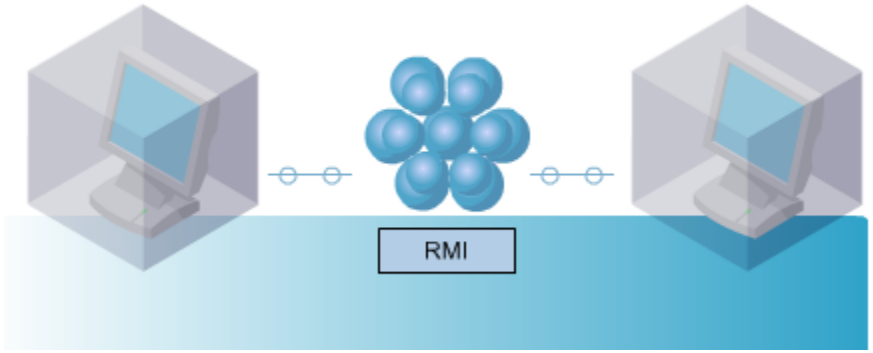
- If you regularly compute tables of values every time a program starts, it could be faster to compute them once, serialize them, and then read it in every time you run the program.

- Serialization uses a binary storage format, so you can't edit the serialized data. Data items that you'll need to edit while the application isn't running, such as user preferences and server configurations, are poor choices for serialization.



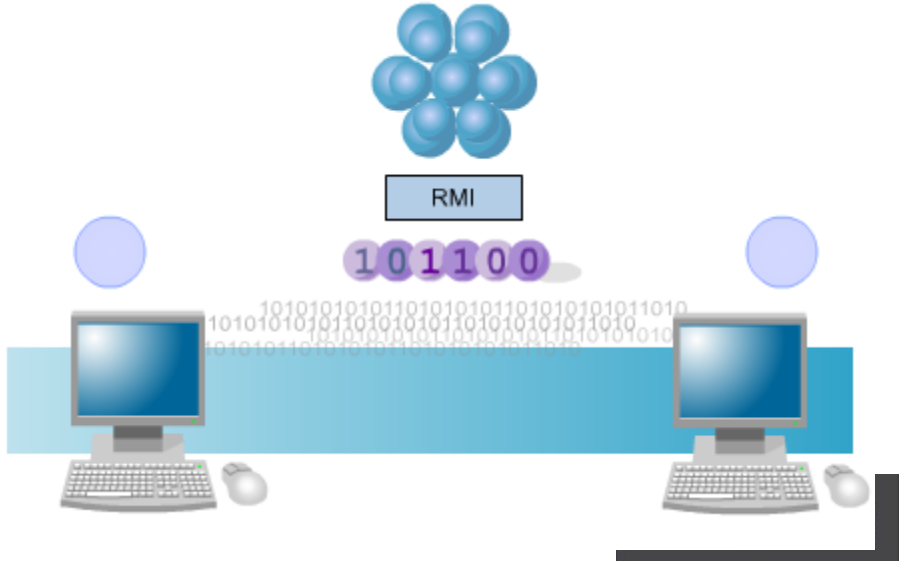
Remote method calls

- In a Java SE Remote Method Invocation (RMI) system, remote method arguments and return values must be transmitted over a network between remote virtual machines (VMs).
- An important part of a remote method call by an RMI client is the marshaling and unmarshaling of the remote method's arguments. And once the remote method has executed, the reverse marshaling and unmarshaling of any return values must occur.
- A feature of Java RMI is that the system takes care of the marshaling and unmarshaling processes automatically.
- Different types of object are transmitted differently over the network, either as remote method arguments or as return values. However, some types of object cannot be used as remote method arguments or return values.



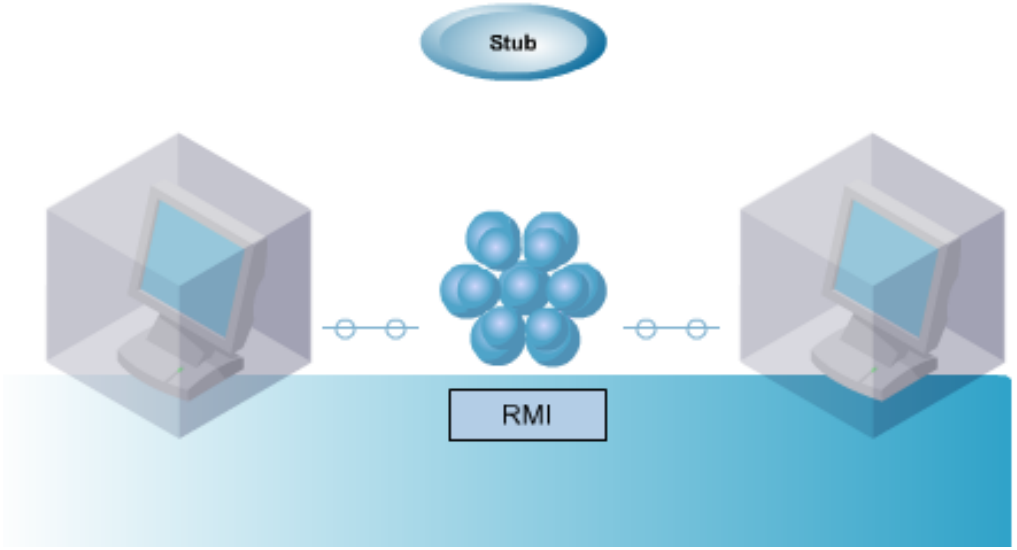
Remote method calls

- A serializable object is an object that implements the Serializable interface or one of its subclasses, such as Externalizable. All primitive datatypes – such as int or char – are also serializable. These datatypes can also be sent over the network as remote method arguments or return values.
- The RMI system passes serializable parameters by copy. When a serializable object is sent over a network to a remote process, the object is serialized and streamed to the process in byte form.
- The remote process constructs a copy of the original object using the information in the stream and the object's bytecode.
- When a copy of a serializable object is made, there is no link between the copy and the original object. So, if a serializable parameter is updated or changed by the server, these updates or changes will not be passed to the original object.



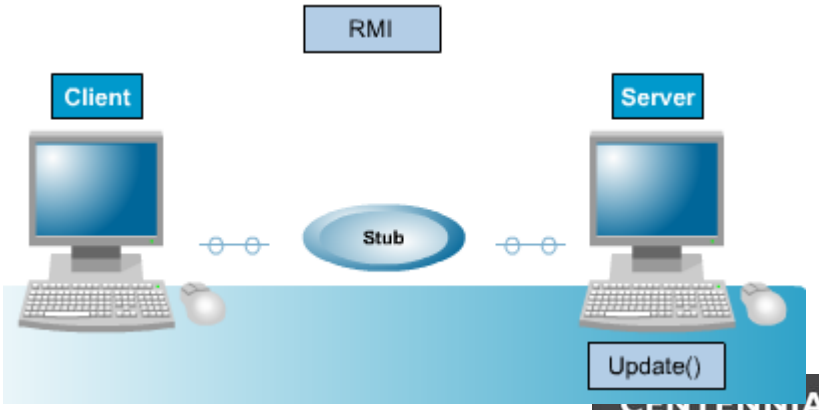
Transmitting remote objects

- The RMI system can send remote objects as remote method arguments or return values over a network. A remote object is an object that implements the Remote interface or one of its subclasses.
- The RMI system passes remote method parameters over the network by reference. When a remote object is a remote method argument or a return value, a remote reference for the remote object – called a stub object – is generated.



Transmitting remote objects

- In an RMI system, it is the stub object – and not the remote object that it represents – that is serialized and sent over the network to a remote process. This stub maintains a link to the original remote object, so changes can be made to the original remote object through the stub.
- The differences between serializable and remote objects must be taken into account when developing RMI clients and servers.
- Let's say a server object calls update methods on an argument to a remote method. In addition, you want the updates to be made to the original object on the client-side, which was passed to the server as the argument.
- In this case, the object should be a remote object because any updates called on the server-side are passed to the original remote object on the client-side through the stub object.
- If an object is not serializable or remote, it cannot be used as an argument to, or return value of, a remote method call. In such cases, a marshal exception will be thrown.

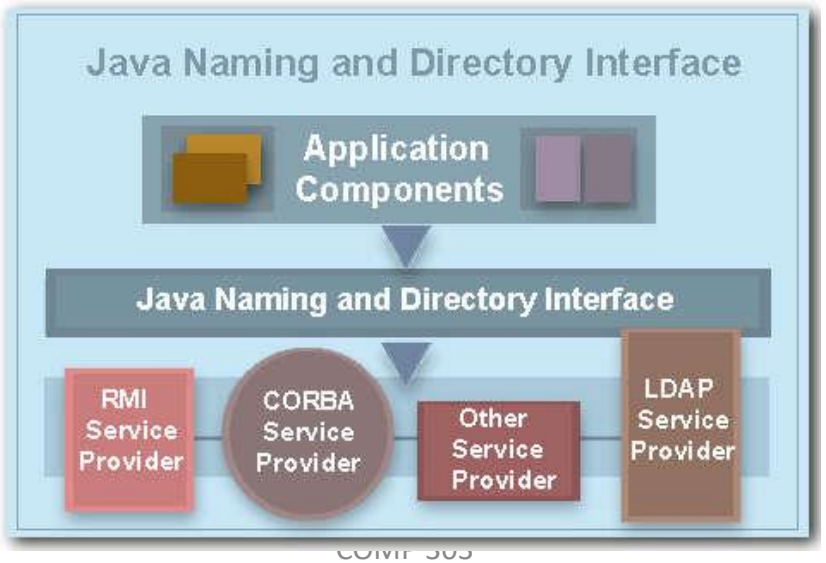


An overview of JNDI

- JNDI Definition
  - The Java Naming and Directory Interface (JNDI) is part of Java EE platform that provides a Java programming interface (API) for connecting Java programs to naming and directory services such as CORBA, LDAP, and RMI. JNDI distinguishes between naming and directory services so, to understand how JNDI works, it is important to differentiate between naming services and directory services.
- Differentiating naming services and directory services
  - The terms "naming service" and "directory service" are often used interchangeably, however there is a difference between them. A naming service associates a single name with a specific resource so, when you search a name service, you can only search for a specific name. A naming service allows you to look up an object by its advertised name. Having located the object, you can then reference it or manipulate it using the host's services – for example, using a file server to move, delete, or rename files remotely.
  - A directory service also finds objects by name, but it can also associate a name with a set of attributes and resources. So when you search a directory service, you can search for items matching a specific set of attributes. Not all directory services have the same structure; some have a flat name space, while others have a tree structure for names. With some directory services, you can store specific types of objects, whereas you can store almost any kind of object with some others.

The JNDI service

- JNDI provides a uniform way of accessing naming and directory services using Java technology. With JNDI, you can read data from any kind of directory, provided there is a JNDI service provider for that directory. Some of the available name service implementations include
  - File naming system
  - Remote Method Invocation (RMI)
  - CORBA
  - Lightweight Directory Access Protocol (LDAP)
- JNDI sits between the application and each of these name services and buffers the application from the underlying service implementations. Enterprise APIs, such as Enterprise JavaBeans, Java Message Service, and JDBC, make use of JNDI for their naming and directory needs.



The JNDI service

File Naming System	File Naming System
RMI	As a distributed computing technology, RMI provides the means of binding objects to names and looking up references via a name. The RMI registry provides a simple mechanism for interfacing with a naming service. RMI is designed for objects written only in Java. RMI over IIOP applications are an extension to Sun's RMI procedure call and they enable Java programs to use JNDI to access the CORBA (COS) naming service and execute CORBA objects, allowing Java access to non-Java processes via CORBA.
CORBA	The CORBA naming service – also known as the Object Naming Service and CosNaming – presents the primary and standard way for mapping between names and objects on a CORBA Object Request Broker (ORB).
LDAP	LDAP is a directory service and its context structure is described as a hierarchical, treelike format, which follows a top-down geographical and organizational description.

The JNDI architecture

- The JNDI architecture consists of an Application Programming Interface (API) and a Service Provider Interface (SPI). Java applications use the JNDI API to access naming and directory services. The JNDI SPI allows a variety of naming and directory services to plug in transparently so the Java application can see the JNDI API to access their services. The JNDI class libraries are contained in five packages:
  - javax.naming
  - javax.naming.directory
  - javax.naming.event
  - javax.naming.ldap
  - javax.naming.spi
- The javax.naming package contains the core Java classes and interfaces for accessing naming services. The SPI classes and interfaces are located in the javax.naming.spi package.
- As the javax.naming package contains the interfaces and the API classes used to interact with the naming service, you'll need to import the package to use the services provided by JNDI. You may need to import other packages as required. For instance, Javax.naming.directory contains the classes required for more advanced directory service functionality; javax.naming.event contains the classes and interfaces required to provide event notification services for naming and directory systems; and javax.naming.ldap contains the classes and interfaces for more advanced LDAP3 management features.

The naming context

- A naming context defines a set of name-to-object bindings in the name service that all share the same naming convention. A name in one context object can be bound to another context object, called a subcontext.
- The Context interface
  - The Context interface is used to define the methods to look up, bind, unbind, and rename objects in the namespace. It can also be used to create and destroy naming sub-contexts for naming services that support sub-contexts.
- The InitialContext class
  - implements the Context interface
  - is the root of a client's naming system
  - serves as the starting point for name resolution
- Names are relative in the JNDI architecture and all JNDI operations are performed in relation to a context or one of its sub-contexts. You'll need to define the starting point of a name space before performing any operations on either a name or a directory service. To do this, you'll need to obtain an InitialContext.
- Obtaining an InitialContext

  To acquire an InitialContext, you need to
  - write the code to select the appropriate service provider
  - provide any additional configuration parameters required by the service
  - instantiate an InitialContext object

Locating a service using JNDI

- Having obtained a naming context, you then use the context to look up a service. Using the lookup method, you can locate an object in the namespace. The lookup method creates a parameter based on the name of the desired object and returns the object reference bound to the name. Let's say you want to look up an EJB in the runtime namespace. The lookup code in the client specifies the bean name as a parameter. The bean name used in the lookup method for standalone clients residing outside the Java EE environment, must take the form:
  - ejb/ [<module_name >] /bean_name
  - For example:
  - Object result = ic.lookup ("ejb/MyBean");
- The bean name used in the lookup method for clients running within the Java EE environment needs the java:comp preface. For example:
  - Object result=ic.lookup("java:comp/env/ejb/MyBean");
- The method Java EE clients use to perform type-narrowing of client-side representations of an EJB's home and remote interface is the javax.rmi.PortableRemoteObject.narrow method. For example:
  - MyBeanHome h = MyBeanHome PortableRemoteObject.narrow (result,MyBeanHome.class);
- To access the attributes of a directory object, you use the DirContext.getAttributes. For example:
  - Attributes result = dc.getAttributes(cn="Sam Brenner", ou = "SalesGroup")
- Other methods such as Context.list( ), Context.bind( ), Context.unbind( ), and Context.rename( ) can be used to list an entire context, and bind, unbind, and rename objects. The DirContext.modifyAttributes( ) method can be used to modify directory objects, while the DirContext.search( ) method can be used to perform a basic directory search.