7 Application Server

Also called an ***appserver***. A program that handles all application operations between users and an organization's backend business applications or databases. Application servers are typically used for complex transaction-based applications. To support high-end needs, an application server has to have built-in redundancy, monitors for high-availability, high-performance distributed application services and support for complex database access

Application servers, whatever their function, occupy a large chunk of computing territory between database servers and the end user. Most broadly, this "country" is called "middleware" and that tells you something about what application servers do. First and foremost, application servers connect database information (usually coming from a database server) and the end-user or client program (often running in a Web browser). There are many reasons for having an intermediate player in this connection -- among other things, a desire to decrease the size and complexity of client programs, the need to cache and control the data flow for better performance, and a requirement to provide security for both data and user traffic.

But that's not all. In the early days of application servers, it was realized that applications themselves, the programs people were using to get work done, were becoming bigger and more complex -- both to write and maintain. At the same time, pressure was increasing for applications to share more of their data and sometimes functionality. More applications were either located on a network or used networks extensively. It seemed logical to have some kind of program residing on the network that would help share application capabilities in an organized and efficient way -- making it easier to write, manage, and maintain the applications.

The end result of this thinking is what is now called an application server. However, these servers first appeared in client/server computing and on LANs. At first, they were often associated with "tiered" applications, when people described the functionality of

applications as two-tiered (database and client program), three-tiered (database, client program, and application server), or n-tiered (all of the above plus whatever). This was (and still is) a complex model of application development, and it resisted wide-scale implementation. Then along came the World Wide Web.

The Web is automatically three-tiered (database, client program, and Web server) and managing data along with application functionality suddenly became not only an esoteric exercise in better program design, but also a downright necessity. This vaulted the application server from obscurity to the top of a pedestal, and literally scores of companies jumped in to develop products.

Not surprisingly, these companies did not, and still do not, see the role of the application server in the same way. They weren't competing just to make something different. Application servers have different roles, and not every company requires the same functionality. Scalability is a good example. Some companies might want an application server that simply helps them organize their applications for the Web, give them better control over the business logic they contain, and make it easier to monitor and secure the data. They don't need thousands of servers. Other companies, especially big ones, do need to manage thousands of servers. For them, the scalability of an application server is crucial. So some application servers feature scalability, others feature other things, and some try to do everything.

What's most important: security, scalability, business logic management, or database connectivity?

One more thing (yet another complication), application server products belong to a variety of programming regimes. Most, though not all, are written in Java. Some are Microsoft friendly; others are not. This latter distinction shows up in support for either CORBA or Microsoft COM+ (and of course some support both). It's relatively important to consider these servers in the light of an organization's programming preferences.

Java Remote Method Invocation (RMI) allows you to write distributed objects using Java. This paper describes the benefits of RMI, and how you can connect it to existing and legacy systems as well as to components written in Java.

RMI provides a simple and direct model for distributed computation with Java objects. These objects can be new Java objects, or can be simple Java wrappers around an existing API. Java embraces the "Write Once, Run Anywhere model. RMI extends the Java model to be run everywhere."

Because RMI is centered on Java, it brings the power of Java safety and portability to distributed computing. You can move behavior, such as agents and business logic, to the part of your network where it makes the most sense. When you expand your use of Java in your systems, RMI allows you to take all the advantages with you.

RMI connects to existing and legacy systems using the standard Java native method interface JNI. RMI can also connect to existing relational database using the standard JDBC package. The RMI/JNI and RMI/JDBC combinations let you use RMI to communicate today with existing servers in non-Java languages, and to expand your use of Java to those servers when it makes sense for you to do so. RMI lets you take full advantage of Java when you do expand your use.

7.1 Advantages of RMI

At the most basic level, RMI is Java's remote procedure call (RPC) mechanism. RMI has several advantages over traditional RPC systems because it is part of Java's object oriented approach. Traditional RPC systems are language-neutral, and therefore are essentially least-common-denominator systems-they cannot provide functionality that is not available on all possible target platforms.

RMI is focused on Java, with connectivity to existing systems using native methods. This means RMI can take a natural, direct, and fully-powered approach to provide you with a distributed computing technology that lets you add Java functionality throughout your system in an incremental, yet seamless way.

The primary advantages of RMI are:

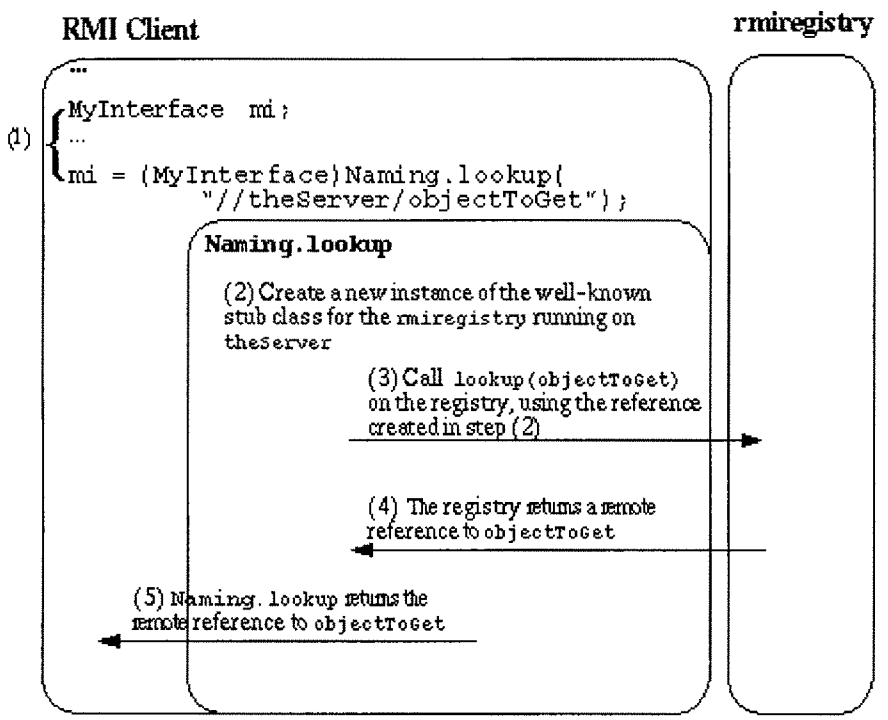
- **Object Oriented**: RMI can pass full objects as arguments and return values, not just predefined data types. This means that you can pass complex types, such as a standard Java hashtable object, as a single argument. In existing RPC systems you would have to have the client decompose such an object into primitive data types, ship those data types, and the recreate a hashtable on the server. RMI lets you ship objects directly across the wire with no extra client code.
- **Mobile Behavior**: RMI can move behavior (class implementations) from client to server and server to client. For example, you can define an interface for examining employee expense reports to see whether they conform to current company policy. When an expense report is created, an object that implements that the client can fetch interface from the server. When the policies change, the server will start returning a different implementation of that interface that uses the new policies. The constraints will therefore be checked on the client side-providing faster feedback to the user and less load on the server-without installing any new software on user's system. This gives you maximal flexibility, since changing policies requires you to write only one new Java class and install it once on the server host.
- **Design Patterns**: Passing objects lets you use the full power of object oriented technology in distributed computing, such as two- and three-tier systems. When you can pass behavior, you can use object oriented design patterns in your solutions. All object oriented design patterns rely upon different behaviors for their power; without passing complete objects-both implementations and type-the benefits provided by the design patterns movement are lost.
- **Safe and Secure**: RMI uses built-in Java security mechanisms that allow your system to be safe when users downloading implementations. RMI uses the security manager defined to protect systems from hostile applets to protect your systems and network from potentially hostile downloaded code. In severe cases, a server can refuse to download any implementations at all.

- **Easy to Write/Easy to Use**: RMI makes it simple to write remote Java servers and Java clients that access those servers. A remote interface is an actual Java interface. A server has roughly three lines of code to declare itself a server, and otherwise is like any other Java object. This simplicity makes it easy to write servers for full-scale distributed object systems quickly, and to rapidly bring up prototypes and early versions of software for testing and evaluation. And because RMI programs are easy to write they are also easy to maintain.
- **Connects to Existing/Legacy Systems**: RMI interacts with existing systems through Java's native method interface JNI. Using RMI and JNI you can write your client in Java and use your existing server implementation. When you use RMI/JNI to connect to existing servers you can rewrite any parts of you server in Java when you choose to, and get the full benefits of Java in the new code. Similarly, RMI interacts with existing relational databases using JDBC without modifying existing non-Java source that uses the databases.
- **Write Once, Run Anywhere**: RMI is part of Java's "Write Once, Run Anywhere" approach. Any RMI based system is 100% portable to any Java Virtual Machine*, as is an RMI/JDBC system. If you use RMI/JNI to interact with an existing system, the code written using JNI will compile and run with any Java virtual machine.
- **Distributed Garbage Collection**: RMI uses its distributed garbage collection feature to collect remote server objects that are no longer referenced by any clients in the network. Analogous to garbage collection inside a Java Virtual Machine, distributed garbage collection lets you define server objects as needed, knowing that they will be removed when they no longer need to be accessible by clients.
- **Parallel Computing**: RMI is multi-threaded, allowing your servers to exploit Java threads for better concurrent processing of client requests.
- **The Java Distributed Computing Solution**: RMI is part of the core Java platform starting with JDK 1.1, so it exists on every 1.1 Java Virtual Machine. All RMI systems talk the same public protocol, so all Java systems can talk to each other directly, without any protocol translation overhead.

7.2 Passing Behavior

When we described how RMI can move behavior above, we briefly outlined an expense report program. Here is a deeper description of how you could design such a system. We present this to show how you can use RMI's ability to move behavior from one system to another to move computing to where you want it today, and change it easily tomorrow.

The examples below do not handle all cases that would arise in the real world, but instead give a flavor for how the problem can be approached.
For an RMI client to contact a remote RMI server, the client must first hold a reference to the server. The Naming.lookup method call is the most common mechanism by which



sclients initially obtain references to remote servers. Remote references may be obtained by other means, for example: all remote method calls can return remote references. This is what Naming.lookup does; it uses a well-known stub to make a remote method call to the rmiregistry, which sends back the remote reference to the object requested by the lookup method.

Every remote reference contains a server hostname and port number that allow clients to locate the virtual machine that is serving a particular remote object. Once an RMI client has a remote reference, the client will use the hostname and port provided in the reference to open a socket connection to the remote server.
Please note that with RMI the terms *client* and *server* can refer to the same Java program. A Java program that acts as an RMI server contains an exported remote object. An RMI client is a program that invokes one or more methods on a remote object in another Java Virtual Machine (JVM). If a JVM performs both of these functions, it may be referred to as an RMI client and an RMI server.

7.3 Object Oriented Code Reuse and Design Patterns

Object oriented programming is a powerful technique for allowing code reuse. Many organizations are using object oriented programming to reduce the burden of creating programs and to increase the flexibility of their systems. RMI is object oriented at all levels-messages are sent to remote objects, and objects can be passed and returned.

The Design Patterns movement has been very successful in describing good practices in object oriented design. First made popular by the seminal work Design Patterns, these patterns of programming are a way to formally describe an overall approach to a particular kind of problem. All of these design patterns rely upon creating one or more abstractions that allow varying implementations, thereby enabling and enhancing software reuse. Software reuse is one of the core promises of object oriented technology, and design patterns are one of the most popular techniques for promoting reuse.

All design patterns rely upon object oriented polymorphism-the ability of an object (such as Task) to have multiple implementations. The general part of the algorithm (such as the compute method) does not need to know which particular implementation is present, it need only know what to do with such an object when it gets one. In particular, the compute server is an example of the Command pattern, which lets you represent a request (task) as an object, letting it be dispatched.

This polymorphism is only available if the full objects, including implementations, can be passed between client and server. Traditional RPC systems, such as DCE and DCOM, and object based RPC systems, such as CORBA, cannot download and execute implementations because they cannot pass real objects as arguments, only data.

RMI passes full types, including implementations, so you can use object oriented programming-including design patterns-every where in your distributed computing solutions, not just in local computation. Without RMI's fully object oriented system, you need to abandon design patterns-along with other forms of object oriented software reuse-in much of your distributed computing systems.