INTRODUCTION
Although Java is rapidly becoming the premier tool for building Internet applications, the fact that seemingly simple Internet applications, such as Web browsers, are actually distributed applications is often overlooked. Internet applications carry with them all the complexities associated with any distributed environment, although the severity of problems is admittedly lighter.

The term distributed applications encompasses many technologies and development techniques, and a clear definition of it remains elusive. For clarity, a distributed application is one in which two or more components are cooperatively operating over a process boundary. The process boundary introduces the need for concurrency and shared memory. Concurrency represents the capability to share resources in a mutually exclusive manner with all the guarantees that it implies.

A growing trend in the industry today is transitioning existing enterprise applications to Web applications. Web applications are best defined as those that present a Hypertext Transfer Protocol (HTTP) interface for operation from within a Web browser. Employing this new motif for interacting with the user has given new life to many of these applications by creating a more intuitive user interface and expanding the platforms that can access it.

Inside of the Web pages that represent this new interface are Java applets — code modules that can execute within a Java virtual machine — that enhance the Web browser’s
ability to interact with the user. For many, this defines the extent of Java’s utility in developing distributed applications. This article presents a greater role for Java in the world of distributed computing. Java distributed computing represents a body of Java programming interfaces that enables Java applications to communicate with each other across a process boundary. The simplest form of this type of computing is two Java applications passing data over a Transmission Control Protocol/Internet Protocol (TCP/IP) network connection. The more complex form is two Java applications sending and receiving Java objects.

JAVA'S BENEFITS

Is Java hype or reality? Java is not a panacea for computing, but a very well thought out tool for building applications that need to operate in a networked environment. Like any tool, Java assists with certain jobs and is completely wrong for others. At this stage in Java’s technology adoption, users are trying to see what this new tool can do.

Java has provided some immediate benefits through its use:

1. Owing to the nature of its programming language design, Java suits complex object-oriented development without some of the pitfalls associated with other languages in this category. For example, C++ requires detailed control of memory allocation and de-allocation. Java handles this automatically through garbage collection. This one small change in philosophy adds a tremendous amount of quality to software and reduces the number of problems caused with memory leaks and overruns.

2. The Java virtual machine is a widely accepted standard that is supported on all major commercial operating systems. Java’s design theoretically supports the concept of write-once run-anywhere, but the mass deployment of virtual machines makes the theoretical concept a reality.

3. Java simplifies the deployment of applications in the organization. Some Java applications can be deployed as applets running on Hypertext Markup Language (HTML) pages in Web browsers. Still others can be full-blown stand-alone Java applications that automatically download their new components as a standard practice of the virtual machine. Here again, the promise of write-once run-anywhere is an important one because these types of deployments are unencumbered by hardware and operating system differences.

4. Java offers the promise of consolidated development resources. Today, many IT departments are strangled by the different hardware and system software platforms requiring support. With Java on all these platforms, many of the specialized resources can be combined to work jointly on multiplatform efforts. These combinations also
help to spread understanding of the business’ core applications across the development staff.

5. With companies clamoring to get at and manipulate their legacy data locked away on mainframe computers, a Java virtual machine can be a saving grace. IBM is currently porting the virtual machine for OS/390 and AS/400. Both of these machines store roughly 85% of corporate data today. Java will provide access to this data as well as offering new ways to process and distribute data throughout the organization.

In general, it could be said that Java simplifies the development, deployment, and maintenance of applications in the enterprise. Since most applications in this environment are for data entry or data retrieval, Java offers enough capabilities and performance today. For some specific applications such as computer-aided design or real-time monitoring, Java cannot provide the performance or features required. Java’s strong suit is thus development of distributed applications — applications that are partitioned into multiple tasks running on different machines.

DISTRIBUTED JAVA APPLICATIONS

Network application programming is never a trivial task, but sending and receiving a simple set of data over the network using Java is greatly simplified. However, there is far more utility when the communicating components maintain context along with the data. To accomplish this, not only data needs to be transmitted, but its functional and structural components as well. Existing distributed computing middleware currently supports maintaining context by using references to running executables. Generally, these references are literal strings that identify the machine where the executable is running. Distributed Java also maintains context as references, but the references are fully functional Java objects.

To define Java distributed computing, one must understand the differences between applications running in the same address space and those running in different address spaces. Two Java applications running in the same address space can simply call functions on each other as long as the functions are programmed to be exposed publicly. In this case, there are no barriers stopping this behavior; however, when the applications are running in separate address spaces, there is a virtual barricade that surrounds each application. This barricade stops one application from seeing and being able to call functions into the other address space. The only way to subvert the virtual barricade is to make the function calls by passing the data over defined pathways into and out of it. In Java, the facility that performs this subversion is called Remote Method Invocation, or RMI.

To provide some familiar vocabulary that will help provide context for Java applications; applications that expose their functions publicly are
sometimes referred to as servers, and the applications that call them are referred to as clients. Hence, the client/server paradigm that has become so popular applies again with the rise of Web applications. When discussing these terms relative to a pure-Java application, applications that expose their functions publicly are referred to as remote Java objects and the applications using them are referred to as Java clients.

**Remote Method Invocation**

The Remote Method Invocation facility is core to distributed Java and defines a framework for Java-to-Java application communications that extends Java over process boundaries in a natural manner. Designed using techniques learned from the experiences of the Distributed Computing Environment's (DCE) Remote Procedure Calls (RPC) and Common Object Request Broker Architecture (CORBA), Java RMI is an advanced inter-object communications system. The primary difference between inter-application and inter-object communications is the requirement for inter-object communications to support pass-by-value for objects. Pass-by-value will be explained in detail later.

As previously stated, RMI extends Java naturally over process boundaries. This means that Java communicates with remote objects — those in another address space — in a manner that is transparent to the user. That is, remote Java objects will behave in the prescribed manner of all Java objects. By upholding this contract, remote Java objects will support the same behavior of a local Java object providing the desired local/remote transparency that is one of the core focuses of distributed computing. To accomplish this level of local/remote transparency, three important requirements must be met:

1. The two communicating components must agree on a common messaging protocol.
2. The two communicating components must use the same transport mechanism, for example, TCP/IP networking protocol.
3. Code and data must be marshaled — packaged in a byte-oriented stream — in a consistent manner.

The work on behalf of JavaSoft to develop and implement these requirements represents an outstanding body of computing research and technology. Actually, points 1 and 2 are fairly simple if experienced in network programming. However, the third point requires the cooperation of multiple Java subsystems, including Introspection, Object Serialization, Garbage Collection, and Remote Method Invocation itself.

**Introspection**

Java's Introspection facilities allow the virtual machine to identify all of a Java object's methods and fields from the Java class description. With this
knowledge, the virtual machine can “flatten” Java objects from their in-memory state to a sequential stream of bytes. Once in the latter format, the object can be stored on persistent media or transferred over a network. This facility is not exposed directly to the programmer, for this would pose an opportunity to subvert the built-in security mechanisms. Instead, this facility is exposed through the Reflection programming interface and object serialization.

Reflection is a programmatic interface for allowing Java objects to identify public, and sometimes private, methods and fields on Java objects. However, the method calls to the Reflection interface are checked by the virtual machines security manager, thus allowing the security manager to restrict access. Of note, Introspection and Reflection can only be used on local Java objects. While a useful tool for building a remote procedure call mechanism, it cannot be used to examine remote Java objects. Therefore, the contract between the client and the server must be designed before the application is programmed.

Object Serialization
Object serialization uses Java’s powers of introspection to store and retrieve objects from a persistent form without requiring additional programming. To accomplish this task, the serialization layer must be able to identify and read all fields on a Java object. Furthermore, the serialization layer must define a format for flattened objects that allows for identification of class type and simplified retrieval.

The data format chosen for object serialization is publicly distributed from JavaSoft with the Java Development Kit (JDK). This format implements certain required functionality for this facility. For example, if two fields within an object reference the same object, only one copy of the object is serialized along with the two individual references. This provides a level of integrity by ensuring that any changes to the object are reflected via both fields. This format also includes the class name for each object that is serialized so that the corresponding code can be associated when retrieved. Additionally, each serialized object is stored with a unique identifier that represents that object within the stream. This allows the object to be updated within the stream without having to serialize the entire graph again.

A common problem associated with persistent objects is reconciling hine. The Naming class is used by Java objects that wish to use remote Java objects and exposes an interface for finding and retrieving a reference to a remote object.

These classes allow Java applications to implement RMI in a modular manner. That is, it does not make applications that use them reliant on any particular implementation of RMI, thus allowing RMI to operate over a host of networking protocols and vendor independent implementations. The following is a sample RMI transaction.
Sample RMI Transaction

The transaction presented in the steps that follow is based on Sun Microsystems’ implementation of RMI that ships with the JDK release 1.1. Again, the only parts of this transaction that are vendor independent are the parts that use the Registry and Naming classes.

**Step 1.** A Java client object wishing to obtain a reference to a remote Java object running on a particular machine uses the Naming class to perform a lookup on that object.

**Step 2.** If the remote object is running and properly registered in a RMI Registry on that machine, the Registry will return a reference. A reference in RMI is a Java object called a stub, which implements all publicly exposed methods on the remote Java object and maps the calls over the network.

**Step 3.** The Java client object makes a method call on the remote Java object that requires an integer parameter. The stub object uses the object serialization facility to serialize the integer object into a stream, along with the method call signature, and delivers it to a dedicated listening network service called the skeleton.

**Step 4.** The skeleton parses the serialized data from the client and builds a properly formatted method call on the remote Java object, which is local to itself. Any return values are serialized and passed back to the Java client. Exhibit 1 clarifies how the transaction operates. On machine 1, a client application uses the Java Naming class to access the Registry located on machine 2. This is done over a lookup operation. Upon successfully locating the requested object in the Registry, a Stub object is dynamically downloaded onto the client. The Stub and the Skeleton work in tandem to marshal data and method calls from the client application to the server application. The line from the Server to the Registry represents a registration process that must occur before the object can be located.

To simplify the transaction description, only an integer was passed from client to server. However, RMI implements the functionality to pass entire objects in this manner. To do this requires pass-by-value functionality for objects. This is a Herculean feat, requiring a system that can encapsulate code and data together; without both there is no assurance that an object’s data will have the proper coherence. That is, pass-by-value transmits entire objects between process boundaries, including the explicit object being passed and all of its implicit objects defined as fields. When designing distributed systems, coherence will be maintained for explicitly passed objects, but implicitly passed objects may require code definition inside the remote address space.
To accomplish this feat, Java builds on the object serialization facility that stores inside the object's stream the name of the Java classes. These names are then used to request for the Java class files to be transferred if they do not exist locally. The capability to pass classes in this manner is not unusual for Java as this is exactly how Web browsers retrieve Java applets from Web servers. Indeed, the logic inside of RMI to accomplish this uses the class loader functionality associated with automatically downloading Java applets.

Ongoing debates in the industry illustrate the lack of appreciation by programmers for this capability: a severe problem for the industry's overall growth. This is most noticeable in technical discussions at industry events and over the Internet in which developers argue that Java-to-Java communications could have been handled by existing inter-object messaging protocols. However, these existing protocols do not inherently support pass-by-value for objects or distributed garbage collection — both requirements of distributed Java.

AGENT TECHNOLOGY
The new capability to pass entire objects, and the objects they contain, has bred a new type of application called the agent. Agent technology is a rapidly growing field within the Java community. The primary reason for this rise is the solution that agents provide to the problem of possible disconnected network states. Java RMI, as well as all remote procedure call mechanisms, are highly synchronous. That is, the client issues a method call and waits for a response. If the network connection is broken at any time during this waiting period, the client application will never receive its response. This could cause the applications to “hang” or to enter into exception handling logic.
Agents allow clients to send processing code to the server when the connection is up that will execute on the server. If the network connection breaks after the agent is delivered, the entire process can still continue normally. This is because agents work in an asynchronous manner; when the client wants the response to the processing, it will make a separate request to the server to return the agent and all of its collected data. Until the client receives this response, the agent will continue to exist, allowing the client to retry multiple times until it receives it successfully.

Java enables agents such as the Web browser to download an applet and call a method on it to start it running. It requires a contract between client and server to send and receive agents, but this is required of all distributed applications. The combination of synchronous and asynchronous programming allows one to design and build robust distributed applications that operate seamlessly across heterogeneous networked environments.

CONCLUSION
Many different solutions exist for distributing a task over multiple address spaces and over a network. This article has presented a method for inter-object communications within a pure Java environment. For some, the simplicity provided by the distributed Java platform may be reason enough to use it over other distributed computing technologies, such as the Common Object Request Broker Architecture (CORBA), OSF Distributed Computing Environment (DEC), and Distributed Component Object Model (COM). However, even if companies choose not to use this platform, the perceptions of how to build distributed applications have been forever altered.

As this article was being written, Oracle, Netscape, IBM, and SunSoft submitted a proposal to the Object Management Group (OMG), requesting that the introspection and pass-by-value be incorporated into CORBA. Interestingly, these are the same features that provide the distributed Java platform with its power.

J. P. Morgenthal is Java computing analyst at nc.focus, a New York-based research and analysis firm that specializes in Java and network computing. He is the publisher of the industry newsletter, *In the Final Analysis*, and can be reached at jp@ifta.com or http://www.ifta.com.