

## Article

# The End of Protocols

[Articles Index](#)

By *Jim Waldo*

The Jini system, announced a little over a year ago, has generated considerable excitement in the world of Java technology and the Internet. The Jini system offers a simple way to construct networks of services that can be accessed by clients. These devices can be implemented either as software services or as hardware devices, and still be accessed in the same way via their Java language interfaces. When a service becomes available, the service can automatically join the network, describing itself by the Java types that it implements. When a service leaves the network, the network can respond to that leaving automatically, so clients will not try to connect to services that are no longer available.

All this is done exploiting the Java environment, with its portable source code, dynamic downloading, and polymorphic object-oriented properties. The Jini system is a Java-centric system on the network, assuming and requiring that the communication between a client and the services used by that client are accomplished through a Java interface. Because of this, many have concluded that Jini requires all components in a Jini system be written completely in Java, and that all of those components communicate with each other using the wire protocol of the Java Remote Method Invocation system. While this is a common case with Jini services and clients, thinking that this is required misses a central point of the Jini approach to distributed computing.

**"...many have concluded that Jini requires all components in a jini system be written completely in Java. ...thinking this is required misses a central point of the Jini approach to distributed computing."**

## Traditional Distributed Systems

To understand this central point, we need to take a step back and understand how communication is accomplished in traditional remote procedure call systems like CORBA or DCOM. In such systems, the central connecting tissue between two programs are the *client-side stub* and the *server-side skeleton*.

The stub is a piece of code that implements an interface to a remote object or service in the address space of a client of that service. The job of the stub is to open up a communication channel to the server, convert all the arguments to be sent to the server into a form that can be transmitted across the wire, and dispatch those converted arguments. The stub code then waits for the response from the server, converting any return values from their wire representation to the internal form used in the process, and handing those results back to the program or thread that made the call.

The skeleton code provides similar functionality on the side of the server. The skeleton code receives the information transmitted by a stub, converts the information that has been transmitted over the network into a form that can be understood by the server program, and makes the appropriate up-call to that server program. The server program will return any result values to the stub code, which will translate those results into a form that can be transmitted over the wire, and send them back to the calling client (where the stub code receives them, as outlined above).

The code for the stub and the skeleton is produced by a compiler that takes as input, a definition of the interface between the client and the service written in some programming language-neutral declarative language (which, seemingly no matter what its form is, is called IDL). These compilers produce source code (sometimes in various languages) that can be compiled by the native compilers for the machines on which the client or service is going to run. The stub and skeleton can then be linked (either statically or at run time) into the client and the service.

**"Protocol design, like any engineering design, is often a trade-off between**

A client, in such a system, uses a single stub to communicate with any service that implements a particular interface, and a service uses a single skeleton to talk to any client that is calling it through a particular interface. This works because the stubs and skeletons produced by the IDL compiler all correspond with a single wire

**that are designed around a one-size-fits-all protocol, such decisions need to favor generality."**

protocol that is defined as part of the overall RPC system. This gives such systems their language- and processor-independence. It doesn't matter what the environment of the client is, or the environment of the server. The system defines what is on the wire, and each system, both client and server, understands that protocol in a way that is appropriate to the language and environment running on

the client or the server.

This independence from language and processor was a requirement when RPC systems were first invented. At that time, every computer company had their own processor, their own operating system, and one or more system languages. Further, the computers of the time were so slow that no one was willing to give up the efficiency of statically compiled binary code to obtain communication between machines. Since the computing world was unable to reach any consensus on a computing environment, communication was enabled by reaching consensus on the communication protocols between those environments.

However, systems based on protocols also have their limitations. Since the protocol needs to be translated into any possible language, the types of information that can be represented in the protocol must be limited to the kinds of data found in all of those languages. If some kind of information needs to be transmitted that is not a part of one of the target languages, conventions for that particular language must be defined to allow artificial representation in the environment, which always adds complexity and often subtly alters the information transmitted.

Systems that are based on a protocol also need to fit the needs of all clients and services to the single protocol. The protocol must be completely general, and as is often the case when something needs to be good for everything, these protocols are often less than optimal for particular, specialized communications. Protocol design, like any engineering design, is often a trade-off between efficiency and generality. In systems that are designed around a one-size-fits-all protocol, such decisions need to favor generality.

The most serious problems with such systems, however, is their rigidity once deployed. Since the skeleton code is part of the server, and the skeleton code is part of the client, any change in one has to be reflected in a change in the other. This means that if the server wishes to update its skeleton code, either to change the basic protocol, or to change the information transmitted to the server or returned to the client, the skeleton code in all of the clients using that server needs to be updated simultaneously. When client/server networks consisted of dozens of machines and the changes in services were slow, such simultaneous updates were possible. However, in the current network environment now grown to global proportions, where services are being defined and refined all the time, the requirement for simultaneous update has become a serious problem for protocol-based systems.

## Environmental Change

What went unnoticed during the evolution of protocol-based distributed systems is that the environment requiring such systems has changed dramatically. The number of different microprocessor and operating systems has been dramatically reduced. More important, the speed of our computing systems has gotten to the point where we no longer need to squeeze every efficiency out of our systems. At the same time, dynamic compilation techniques have advanced to the point where dynamically compiled and optimized code is running at speeds that are close to (and in some case faster than) code that has been statically compiled.

The safety of the Java technology environment provides a single, uniform environment in which code can be dynamically loaded into a running process no matter what the underlying processor or operating system. Java's safety means that users are willing to allow such downloaded code to run. The result is a system in which portable binary code is available to the developer of distributed systems. It is this functionality that RMI, and through the semantics of RMI, Jini, exploit to change the protocol-centric nature of distributed systems.

**"The safety of the Java technology environment provides a single, uniform environment in which code can be dynamically loaded into a running process no matter what the underlying processor or operating system."**

The key difference between these Java-based distributed systems and the protocol-based systems we have been looking at is that, in RMI and Jini, the stub code used by the client is not owned by the client. Instead, that stub code comes from the service that the client is wanting to use. In RMI, such stubs are the RMI references, which implement not just the remote interfaces expected by the client but all of the remote interfaces supported by the service. In Jini, these stubs are the proxy objects obtained from the Jini Lookup Service, which are often RMI references but need not be.

In both cases, the Java language equivalent of a stub is dynamically downloaded (if needed) to the client. The code that gets downloaded for the class comes, at root, from the service itself. Different services that implement the same interface may well have different code for their reference or proxy, a fact is hidden from the client of the service who only needs to know the Java interface. But like any other object in a true object-oriented system, the implementation behind an interface can change without the client of that interface knowing or needing to know.

An immediate outcome of this is that the protocol between the client and the server is not the nexus of interaction between those two entities. Since the server provides the stub code to the client on the fly and on demand, that code can change. In particular, an RMI-based server can implement an extension of a previously supported interface without there needing to be any change on the part of the clients. The extension means there is new stub code for the client to use, but the client will receive that new stub code the next time it receives a reference to the service. This allows the service to change, and the clients to automatically update themselves on an as-needed (rather than on a coordinated) basis.

**"Jini proxy code is even less tied to a protocol than RMI."**

This also means that the RMI protocol (sometimes called JRMP) is an accidental, rather than an essential, feature of RMI. The protocol can be expanded and altered (within certain limits, caused by the RMI system's provision for distributed

garbage collection) without changing the RMI system. Again, such changes are possible because the stub code, encapsulated in the RMI reference to the service, is a dynamically loaded and executed piece of code rather than something that has been associated (forever) with the client.

## Jini and Protocol Independence

Jini proxy code is even less tied to a protocol than RMI. A Jini proxy object is only required to be an implementation of an interface (which is used to identify the object in the Jini Lookup service). This allows the client of the service to know what calls to make to gain access to the service. But how (or even if) the proxy communicates with the service itself is completely up to the proxy and the service from which it comes. The proxy can be an RMI reference (a common, and easily supported case), an object that communicates using some other common and well known protocol (such as CORBA's IIOP protocol), a piece of code that uses a specialized protocol known only to the proxy and the service itself, or a full implementation of the service that runs locally in the client's address space. All of this is transparent to the client, who only sees the Java interface. In the Jini world, the protocol used between a proxy and a service is a private matter between those two objects.

This works because the Jini system uses the RMI semantic notion of associating the proxy with the service and dynamically loading the proxy on demand. In effect, the proxy and the service form a single object that is itself distributed, with part of the object living in the address space of the client and part of the object living at the location of the service.

This approach gives great flexibility to what protocol is actually used. Different services can invent their own specialized protocols that are optimized for that particular pair of proxy and service. Protocols can evolve over time as new ideas are tried out.

But this also explains why the client's access to the network is Java environment-centric. For this approach to be viable, there needs to be a way of dynamically downloading code from the service to the client-code that the client can safely load into its address space and call. Java technology provides this kind of environment, and is the environment of choice for both RMI and Jini.

## Regaining Language Independence

Does this mean that all of the service must be written in the Java programming language? Absolutely not. For an RMI environment, all that is required is that the Java environment be running to allow the exportation of the Java classes that are the implementation of the references to the RMI object. That object can be implemented in any language that can be called by the Java Virtual Machine<sup>1</sup>, using the Java Native Interface (JNI) mechanisms. All that is needed is a simple JNI wrapper, not a complete conversion.

For Jini services, the connection to the Java platform is even more indirect. All that is required of a service that wants to participate in Jini is that the service (be it hardware or software) be able to register a Java object in the Jini lookup service (and keep that registration alive through the renewal of leases) that can communicate with the service. This Java proxy can use any protocol it likes to talk to the service. And the service can have the registration with the lookup service initiated and maintained through other Jini services, like those that are currently available in the Jini 1.1 alpha release.

**"So it is simply not true that only Java code can be used in RMI or Jini environments. Instead, what is required is the use of Java to interact with the network."**

Clients wishing to use these services need to be able to load and run the Java code that is the reference or proxy to the service. But the client application does not need to be written completely in the Java programming language--again, through the native method interfaces, any language that can call or be called by Java code will suffice to allow the client to make use of Jini services.

## Java Platform on the Network

So it is simply not true that only Java code can be used in RMI or Jini environments. Instead, what is required is the use of Java to interact with the network. While this is not without some cost, the benefits are the freedom from any particular wire protocol and the ability to update the stub code used by the client in an asynchronous, on-the-fly fashion. One hope is that this will allow the freedom of experimentation in the area of communication protocols that was impossible in the language-neutral distributed environments that relied on static protocols. At the very least, this approach gives us new techniques to use in scaling our distributed systems in ways that we have not been able to achieve before.

## Related Links

[RMI Home Page](#)  
[Jini Home Page](#)  
[Jini Community Page](#)



## About the Author

*Jim Waldo, Distinguished Engineer with Sun Microsystems, is the lead architect for Jini. Prior to Jini, Jim worked in JavaSoft and Sun Microsystems Laboratories, where he did research on object-oriented programming and systems, distributed computing, and user environments.*

*Before joining Sun, Jim spent eight years at Apollo Computer and Hewlett Packard working on distributed object systems, user interfaces, class libraries, text and internationalization. At HP, he led the design and development of the first Object Request Broker, and was instrumental in getting that technology incorporated into the first OMG CORBA specification. He edited "The Evolution of C++: Language Design in the Marketplace of Ideas" (MIT Press), and was one of the authors of "The Jini Specification" (Addison Wesley).*

*Jim received his Ph.D. in philosophy from the University of Massachusetts Amherst, and holds M.A. degrees in both linguistics and philosophy from the University of Utah. He is a member of the IEEE and ACM; and an adjunct faculty member of Harvard University, where he teaches distributed computing in the department of computer science.*

---

<sup>1</sup> As used on this web site, the terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.