

# *Jini<sup>TM</sup> Technology Glossary*



**THE NETWORK IS THE COMPUTER<sup>®</sup>**

901 San Antonio Road  
Palo Alto, CA 94303 USA  
415 960-1300  
fax 415 969-9131

Revision 1.0  
January 25, 1999

Copyright © 1999 Sun Microsystems, Inc.  
901 San Antonio Road, Palo Alto, CA 94303 USA

All rights reserved. Copyright in this document is owned by Sun Microsystems, Inc.

Sun Microsystems, Inc. has patent and other intellectual property rights relating to implementations of the technology described in this Specification ("Sun IPR"). Your limited right to use this Specification does not grant you any right or license to Sun IPR. A limited license to Sun IPR is available from Sun under a separate Community Source License.

THIS SPECIFICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY YOU AS A RESULT OF USING THE SPECIFICATION.

THIS SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE SPECIFICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE SPECIFICATIONS AT ANY TIME, IN ITS SOLE DISCRETION. SUN IS UNDER NO OBLIGATION TO PRODUCE FURTHER VERSIONS OF THE SPECIFICATION OR ANY PRODUCT OR TECHNOLOGY BASED UPON THE SPECIFICATION. NOR IS SUN UNDER ANY OBLIGATION TO LICENSE THE SPECIFICATION OR ANY ASSOCIATED TECHNOLOGY, NOW OR IN THE FUTURE, FOR PRODUCTIVE OR OTHER USE.

#### RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-1(a).

#### TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Jini, JavaSpaces, JavaSoft, JavaBeans, JDK, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, EmbeddedJava, PersonalJava, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, Sun WorkShop, XView, Java WorkShop, the Java Coffee Cup logo, and Visual Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

# *Jini<sup>TM</sup> Technology Glossary*

---



*activation*: The process of transforming a passive object into an active object. Activation requires that an object be associated with a Java<sup>TM</sup> Virtual Machine (JVM), which may entail loading the class for that object into a JVM and the object restoring its persistent state (if any). ([Java Remote Method Invocation Specification](#), Section 7.1.1)

*activation descriptor*: A class instance which holds an activatable object's group identifier (specifies the JVM in which it is activated), the object's class name, a `location` from where to load the object's class code, and object-specific initialization data in marshalled form. ([Java Remote Method Invocation Specification](#), Section 7.2)

*activation group*: The entity which receives a request to activate an object in the JVM and returns the activated object back to the activator. (Section 7.2) A separate JVM is spawned for each activation group. ([Java Remote Method Invocation Specification](#), Section 7.4.7)

*activator*: The entity which supervises activation by being both (1) a database of information that maps activation identifiers to the information necessary to activate an object and (2) a manager of JVMs, that starts up JVM's (when necessary) and forwards requests for object activation (along with the necessary information) to the correct activation group inside a remote JVM. There is usually only one activator per host, started by `rmid`. ([Java Remote Method Invocation Specification](#), Section 7.2)

*active object*: A remote object that is instantiated and exported in a JVM on some system. ([Java Remote Method Invocation Specification](#), Section 7.1.1)

- ancestor transaction*: A transaction that is the parent of a specific nested transaction (a transaction where all its operations are contained, or executed, from within another transaction), or the parent of such a parent, recursively (a grand-parent, a great-grand-parent, and so on). ([Jini™ Transaction Specification](#), Section 3.5)
- attribute set*: A strongly-typed set of fields in a service item (represented by a `net.jini.core.entry.Entry`), that describe the service or provide secondary interfaces to the service. A single attribute is a public field of an `Entry`. ([Jini™ Lookup Service Specification](#), Section 1.3)
- channel*: The abstraction for a conduit between two address spaces, in the RMI transport layer. As such, it is responsible for managing connections between the local address space and the remote address space for which it is a channel. ([Java Remote Method Invocation Specification](#), Section 3.5)
- connection*: The stream-oriented (Section 3.4) abstraction for transferring data (performing input/output) in the RMI transport layer. ([Java Remote Method Invocation Specification](#), Section 3.5)
- discovering entity*: One or more cooperating objects in the Java programming language on the same host, that are about to start, or are in the process of, obtaining references to one or more Jini lookup services. ([Jini™ Discovery and Join Specification](#), Section 1.2)
- discovery request service*: A service which runs on a host in the djinn, and accepts requests for a remote reference to an instance of the Jini lookup service. There are really two discovery request services; one accepts multicast requests, and the other accepts unicast requests. Both instances of the discovery request service are present on every system in a djinn that hosts an instance of the Jini Lookup service.
- discovery response service*: A remote object which runs on a discovering entity, and accepts references to instances of the Jini lookup service. An instance of the discovery response service is hosted on every system that wishes to establish communications with a djinn.
- distributed event adapter*: An event adapter, where the event generator and the event listener instances may exist in different virtual machines, possibly on different hosts. The distributed event adapter is at least a remote event listener, but may also be a remote event generator (see *local event*, *remote event*). ([Jini™ Distributed Event Specification](#), Section 3)
- djinn*: The group of devices, resources, and users joined by the Jini software infrastructure. ([Jini™ Lookup Service Specification](#), Section 1.1) This group, controlled by the Jini system, agrees on basic notions of trust, administration, identification, and policy.

- dynamic class loading*: The capability of the Java application environment to download files (classes for the Java platform, audio, and images) from a httpd server at runtime, if they are not already available to the client JVM. Dynamic class loading may be used by the RMI runtime to download: stub classes; skeleton classes; classes that are passed as sub-types of declared method parameters; and classes that are passed as subtypes of declared method return types. (See dynamic stub loading)
- dynamic stub loading*: A subset of dynamic class loading, used to support client-side stubs which implement the same set of remote interfaces as a remote object itself. ([Java Remote Method Invocation Specification](#), Section 3.1)
- endpoint*: The abstraction used to denote an address space or JVM in the RMI transport layer. In the implementation, an endpoint can be mapped to its transport. That is, given an endpoint, a specific transport instance can be obtained. ([Java Remote Method Invocation Specification](#), Section 3.5)
- entry*: An entry is a typed group of object references, expressed as a class for the Java platform that implements the `net.jini.core.entry.Entry` interface. Entry fields must all be references to `Serializable` objects. ([Jini™ Entry Specification](#), Section 1)
- event*: Something that happens in an object, corresponding to some change in the abstract state of the object. Events are abstract occurrences that are not directly observed outside of an object, and may not correspond to a change in the actual state of the object that advertises the ability to register interest in the event. ([Jini™ Distributed Event Specification](#), Section 2.1)
- event generator*: An object that has some kinds of abstract state changes that might be of interest to other objects, and allows other objects to register interest in those events. This is the object that will generate notifications when events of this kind occur, sending those notifications to the event listeners that were indicated as targets in the calls that registered interest in that kind of event. ([Jini™ Distributed Event Specification](#), Section 2.1)
- event listener*: An object that has interest in being notified when a particular event type occurs. The event listener (1) implements the appropriate interface, and (2) registers with an event generator. (See remote event listener)
- export, -ed, -ing*: The process of making a remote object available to accept incoming calls, on a specific port. An object can be exported (1) if the object is a sub-class of `java.rmi.server.UnicastRemoteObject`, through the constructor, (2) by passing the object to the static `UnicastRemoteObject` method, `exportObject` (Section 5.3.1), (3) if the

object is a sub-class of `java.rmi.activation.Activatable`, through the constructor, or (4) by passing the object to the static `Activatable` method, `exportObject`. ([Java Remote Method Invocation Specification](#), Section 7.3)

*faulting remote reference*: A faulting remote reference to a remote object, sometimes referred to as a fault block, “faults in” the active object’s reference upon the first method invocation to the object. Each faulting reference, contained in the remote object’s stub, maintains both a persistent handle (a `java.rmi.activation.ActivationID`) and a transient remote reference to the target remote object. ([Java Remote Method Invocation Specification](#), Section 7.1.2)

*host*: A hardware device that may be connected to one or more networks. An individual host may house one or more JVMs. ([Jini™ Discovery and Join Specification](#), Section 1.2)

*inferior transaction*: The inverse of the transactional ancestor relationship: Transaction  $T_i$  is an inferior of  $T_a$  if and only if  $T_a$  is an ancestor of  $T_i$ . ([Jini™ Transaction Specification](#), Section 3.5)

*joining entity*: One or more cooperating objects in the Java programming language on the same host that have just received a reference to the Jini Lookup service and are in the process of obtaining services from, and possibly exporting services to, a djinn. ([Jini™ Discovery and Join Specification](#), Section 1.2)

*join protocol*: The protocol which allows entities to start communicating usefully with services in a djinn, through the Jini lookup service. ([Jini™ Discovery and Join Specification](#), Section 1.1)

*lazy activation*: The activation mechanism that the RMI system uses, which defers activating an object until a client's first use (i.e., the first method invocation). Lazy activation of remote objects is implemented using a faulting remote reference. ([Java Remote Method Invocation Specification](#), Section 7.1.1)

*lease*: A grant to use a resource, offered by one object in a distributed system, to another object in that system for a certain period of time. The duration of the lease is negotiated by the two objects when access to the resource is first requested and given. ([Jini™ Distributed Leasing Specification](#), Section 1) A lease ensures that the lease holder will have access to some resource for a period of time. During the period of a lease, a lease can be cancelled by the entity holding the lease. A lease holder can request that a lease be renewed, or a lease can expire. ([Jini™ Distributed Leasing Specification](#), Section 2.1) In the current RMI implementation, a lease term is not negotiated, as described by the [Jini™ Distributed Leasing Specification](#); the lease term is mandated by the implementation server. Another difference is that in RMI, there is no notion of explicit lease cancellation; lease cancellation is implicit when a remote reference becomes unreferenced by a specific client. ([Java Remote Method Invocation Specification](#), Section 9.1)

*lease grantor*: The object granting access to a resource for some period of time. ([Jini™ Distributed Leasing Specification](#), Section 2)

*lease holder*: The object asking for the leased resource. ([Jini™ Distributed Leasing Specification](#), Section 2)

*live reference*: The concrete representation of a remote object reference (in the RMI transport layer) which consists of an endpoint and an object identifier. Given a live reference for a remote object, a transport can use the endpoint to set up a connection to the address space in which the remote object resides. On the server side, the transport uses the object identifier to look up the target of the remote call. ([Java Remote Method Invocation Specification](#), Section 3.5)

*local event*: An event object that is fired from an event generator to an event listener, where both the generator and the listener instances exist in the same virtual machine. (See event, *remote event*) ([Jini™ Distributed Event Specification](#), Section 1.1)

*lookup discovery protocol*: The protocol that governs the acquisition of a reference to one (or more) instances of the Jini lookup service. ([Jini™ Discovery and Join Specification](#), Section 1.1)

*lookup service*: The Jini lookup service provides a central registry of service items, representing services, available within the djinn. This Jini Lookup service is a primary means for programs to find services within the djinn, and is the foundation for providing user interfaces through which users and administrators can discover and interact with services in the djinn. ([Jini™ Lookup Service Specification](#), Section 1.1)

*marshal streams*: Input/output streams, used by the RMI remote reference layer, that employ object serialization to enable objects in the Java programming language to be transmitted between address spaces. ([Java Remote Method Invocation Specification](#), Section 3.3)

*marshalled object*: A container for an object that allows that object to be passed as a parameter in an RMI call, but postpones deserializing the object at the receiver until the application explicitly requests the object (via a call to the container object). The serializable object contained in the `MarshaledObject` is serialized and deserialized (when requested) with the same semantics as parameters passed in RMI calls ([Java Remote Method Invocation Specification](#), Section 7.4.8), which means that any remote object in the `MarshaledObject` is represented by a serialized instance of its stub. The object contained by the `MarshaledObject` may be a remote object, a non-remote object, or an entire graph of remote and non-remote objects.

*notification filter*: A distributed event adapter which can be used by either the generator of a notification or the recipient to intercept notification calls, do processing on those calls, and act in accord with that processing (perhaps forwarding the notification, or even generating new notifications). ([Jini™ Distributed Event Specification](#), Section 3.2) This filter may be used as an event multiplexer or demultiplexer.

*notification mailbox*: A distributed event adapter which can be used to store the notifications sent to an object until such time as the object for which the notifications were intended desires delivery. Such delivery can be in a single batch, with the mailbox storing any notifications received after the request for delivery until the next request is given. Alternatively, a notification mailbox can be viewed as a faucet, with notifications turned on (delivering any that have arrived since the notifications were last turned off) and then delivering any subsequent notifications to an object immediately, until told by that object to hold the notifications. ([Jini™ Distributed Event Specification](#), Section 3.3)

*object serialization*: The system which allows a bytestream to be produced from a graph of objects, sent out of the Java application environment (either saved to disk or sent over the network) and then used to re-create an equivalent set of objects with the same state. ([Java Object Serialization Specification](#), Section A.1) In RMI, objects transmitted using the object serialization system are passed by copy to the remote address space, unless they are remote objects, in which case they are passed by reference. ([Java Remote Method Invocation Specification](#), Section 3.3)

*passive object*: A remote object that is not yet instantiated (or exported) in a JVM, but which can be brought into an active state (see *active object*). ([Java Remote Method Invocation Specification](#), Section 7.1.1)

*pure transaction*: A transaction in which all access to shared mutable state is performed under transactional control. ([Jini™ Transaction Specification](#), Section 3)

*reference list*: A reference list for a remote object is a list of client JVMs that hold references to that remote object. A client JVM is removed from the object's reference list when that client no longer references that object. ([Java Remote Method Invocation Specification](#), Section 9.1)

*registry*: A remote object that maps names to remote objects. The `java.rmi.Naming` class provides methods for lookup, binding, rebinding, unbinding, and listing the contents of a registry. (Section 6.1) A registry can be used in a virtual machine with other server classes or standalone. The methods of `java.rmi.registry.LocateRegistry` may be used to get a registry operating on a particular host or host and port. ([Java Remote Method Invocation Specification](#), Section 6)

*remote event*: An object that is passed from an event generator to a remote event listener to indicate that an event of a particular kind has occurred. The remote event generator and the remote event listener instances may exist in different virtual machines, possibly on different hosts. ([Jini™ Distributed Event Specification](#), Section 2.1)

*remote event generator*: An object that is the source of remote events.



- remote event listener*: An object that has implemented the `net.jini.core.event.RemoteEventListener` interface, which is interested in the occurrence of remote events in some other object. The major function of a remote event listener is to receive notifications of the occurrence of a remote event in some other object (or set of objects). ([Jini™ Distributed Event Specification](#), Section 2.1)
- remote interface*: An interface written in the Java programming language that extends `java.rmi.Remote`, either directly or indirectly, which declares the methods of a remote object. ([Java Remote Method Invocation Specification](#), Section 2.1)
- remote method invocation (RMI)*: The action of invoking a method of a remote interface on a remote object. ([Java Remote Method Invocation Specification](#), Section 2.1)
- remote object*: An object whose methods can be invoked from another JVM, potentially on a different host. An object of this type is described by one or more remote interfaces. ([Java Remote Method Invocation Specification](#), Section 2.1)
- remote reference layer (RRL)*: The layer of the RMI system that supports remote reference behavior (such as invocation to a single object or to a replicated object) and carries out the semantics of method invocation. This layer sits between the RMI stub/skeleton layer and the RMI transport layer. Also handled by the remote reference layer are the reference semantics for the server. ([Java Remote Method Invocation Specification](#), Section 3.2)
- rmic*: The stub and skeleton compiler used to generate the appropriate stubs and skeletons for a specific remote object implementation. The compiler is invoked with the package-qualified class name of the remote object class. The class must previously have been compiled successfully. ([Java Remote Method Invocation Specification](#), Section 5.11)
- rmid*: The activation system daemon which provides an implementation of the activation system interfaces. In order to use activation, you must first run `rmid`. This is the JVM with which activation descriptions get registered. ([Java Remote Method Invocation Specification](#), Section 7.2)
- rmiregistry*: The RMI system command that provides an implementation of the `java.rmi.registry.Registry` interface. The `rmiregistry`, run on a remote host, can be accessed by calling methods of the `java.rmi.Naming` class.
- semantic transaction*: A transaction with specific, associated semantics, as opposed to the protocol specified by the `TransactionManager` interface which does not specify transaction semantics. A semantic transaction is contractual in nature, and implies a particular usage pattern, so if a program operates within the constraints of the contract, assumptions can be safely made about the transaction's behavior or state. ([Jini™ Transaction Specification](#), Section 1.2)

- serializable*: Any data type that may be read from `java.io.ObjectInputStreams` and written to `java.io.ObjectOutputStreams`. This includes primitive data types in the Java programming language, remote objects in the Java programming language, and non-remote objects in the Java programming language that implement the `java.io.Serializable` interface. ([Java Remote Method Invocation Specification](#), Section 2.6)
- service*: Something that can be used by a person, a program, or another service. It can be computational, storage, a communication channel to another user, or another service. Examples of services include devices such as printers, displays, disks; software such as applications or utilities; information such as databases and files; and users of the system. Services will appear programmatically as objects in the Java programming language, perhaps made up of other objects in the Java programming language. A service will have an interface, which defines the operations that can be requested of that service. The type of the service determines the interfaces that make up that service. ([Jini™ Architecture Specification](#), Section 2.1)
- service items*: Each service item represents an instance of a service available within the djinn. The item contains the stub (if the service is implemented as a remote object) or serialized object (if the service makes use of a local proxy) that programs use to access the service, and an extensible collection of attribute sets that describe the service or provide secondary interfaces to the service. A new service item is created in the Jini Lookup service when a new service is added to the djinn. ([Jini™ Lookup Service Specification](#), Section 1.2)
- service registrar*: A synonym for Jini Lookup service. (See *lookup service*) ([Jini™ Lookup Service Specification](#), Section 2.5)
- skeleton*: The server-side entity that reads parameters from incoming method requests and dispatches calls to the actual remote object implementation. Note that in the Java Development Kit 1.2, skeleton functionality is now handled by the remote object stub, but skeletons may still be used for compatibility with earlier releases of the JDK. ([Java Remote Method Invocation Specification](#), Section 3.3)
- store-and-forward agent*: A distributed event adapter that enables the object generating a notification to hand the actual notification of those who have registered interest off to a separate object. This agent can implement various policies for reliability. ([Jini™ Distributed Event Specification](#), Section 3.1)
- stub*: The proxy for a remote object, which implements all the interfaces that are supported by the remote object implementation and forwards method invocations to the actual remote object instance. ([Java Remote Method Invocation Specification](#), Section 3.3)

*stub/skeleton layer*: The layer of the RMI system that aids in carrying out method invocation. The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. ([Java Remote Method Invocation Specification](#), Section 3.3) This layer does not deal with specifics of any transport, but transmits data to the remote reference layer via the abstraction of marshal streams. This layer contains client-side stubs (proxies) and server-side skeletons. ([Java Remote Method Invocation Specification](#), Section 3.2)

*template*: An entry object that has some or all of its fields set to specified values. Templates may be used to find matching entries. A template will match an entry if and only if the template's non-null public fields match the entry's non-null public fields *exactly*. Remaining fields (those set to null) are not used in the matching process, but are left as *wildcards*. ([Jini™ Entry Specification](#), Section 1.5)

*transaction*: In general, a transaction is a tool that allows a set of operations to be grouped in such a way as to make them all appear to either all succeed or all fail; further, the operations in the set appear from outside the transaction to occur simultaneously. *In this model*, the concrete representation of a transaction is encapsulated in an object. ([Jini™ Transaction Specification](#), Section 1.1)

*transaction client*: An object that does either, or both, of the following: (1) requests that a transaction manager create a transaction (2) invokes the `commit` or `abort` method to complete a transaction. A single transaction may have more than one client, since the object that completes a transaction may be different from the object that requested its creation. An object that is a transaction client may also be a transaction manager or participant. ([Jini™ Transaction Specification](#), Section 1.2)

*transaction manager*: An object that (1) services requests from transaction clients to create transactions and (2) tracks and manages the completion state of those transactions, by implementing the `TransactionManager` interface. An object that is a transaction manager may also be a transaction client or participant. ([Jini™ Transaction Specification](#), Section 1.2)

*transaction participant*: An object that executes operations of a transaction, and is able to interact with the manager to complete transactions properly. An object providing this service may implement the `TransactionParticipant` interface. An object that is a transaction participant may also be a transaction manager or client. ([Jini™ Transaction Specification](#), Section 1.2)

*transport*: The abstraction that manages channels in the RMI transport layer. Each channel is a virtual connection between two address spaces. Within a transport, only one channel exists per pair of address spaces (the local address space and a remote address space). Given an endpoint to a remote address space, a transport sets up a channel to that address space. The transport

abstraction is also responsible for accepting calls on incoming connections to the address space, setting up a connection object for the call, and dispatching to higher layers in the system. ([Java Remote Method Invocation Specification](#), Section 3.5)

*transport layer*: The layer of the RMI system that is responsible for connection set up, connection management, and remote object tracking. ([Java Remote Method Invocation Specification](#), Section 3.2) The transport layer sits below the RRL.

*weak reference*: When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference. The weak reference allows the JVM's garbage collector to discard the object if no other strong references to the object exist. The distributed garbage collection algorithm interacts with the local JVM's garbage collector in the usual ways by holding normal or weak references to objects; thus, a weak reference allows the RMI runtime to reference a remote object, but not prevent the object from being garbage collected. ([Java Remote Method Invocation Specification](#), Section 3.7)