JavaSpaces™ Specification

The JavaSpaces™ technology package provides a distributed persistence and object exchange mechanism for code written in the Java™ programming language. Objects are written in entries that provide a typed grouping of relevant fields. Clients can perform simple operations on a JavaSpaces server to write new entries, lookup existing entries, and remove entries from the space. Using these tools, you can write systems to store state, and also write systems that use flow of data to implement distributed algorithms and let the JavaSpaces system implement distributed persistence for you.

Sun microsystems

THE NETWORK IS THE COMPUTER™

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

Revision 1.0
January 25, 1999

Contents

Introduction 1

1.1 Overview

Distributed systems are hard to build. They require careful thinking about problems that do not occur in local computation. The primary problems are those of partial failure, greatly increased latency, and language compatibility[1]. The Java™ programming language has a remote method invocation system called RMI[2] that lets you approach general distributed computation in the Java programming language using techniques natural to the Java programming language and application environment. This is layered on the Java platform's object serialization mechanism[3] to marshal parameters of remote methods into a form that can be shipped across the wire and unmarshalled in a remote server's Java™ Virtual Machine (JVM).

This specification describes the architecture of JavaSpaces technology, which is designed to help you solve two related problems: distributed persistence and the design of distributed algorithms. JavaSpaces services use RMI and the serialization feature of the Java programming language to accomplish these goals.

1.2 The JavaSpaces Application Model and Terms

A JavaSpaces service holds *entries*. An entry is a typed group of objects, expressed in a class for the Java platform that implements the interface `net.jini.core.entry.Entry`. Entries are described in detail in the *Jini™Entry Specification*.

An entry can be *written* into a JavaSpaces service, which creates a copy of that entry in the space[1] that can be used in future lookup operations.

You can look up entries in a JavaSpaces service using *templates,* which are entry objects that have some or all of its fields set to specified *values* that must be matched exactly. Remaining fields are left as *wildcards*—these fields are not used in the lookup.

There are two kinds of lookup operations: *read* and *take*. A *read* request to a space returns either an entry that matches the template on which the read is done, or an indication that no match was found. A *take* request operates like a read, but if a match is found, the matching entry is removed from the space.

You can request a JavaSpaces service to *notify* you when an entry that matches a specified template is written. This is done using the distributed event model contained in the package `net.jini.core.event` and described in the *Jini™Distributed Event Specification.*

All operations that modify a JavaSpaces service are performed in a transactionally secure manner with respect to that space. That is, if a write operation returns successfully, that entry was written into the space (although an intervening take may remove it from the space before subsequent lookup of yours). And if a take operation returns an entry, that entry has been removed from the space, and no future operation will read or take the same entry. In other words, each entry in the space can be taken at most once. Note, however, that two or more entries in a space may have exactly the same value.

The architecture of JavaSpaces technology supports a simple transaction mechanism that allows multi-operation and/or multi-space updates to complete atomically. This is done using the two-phase commit model under the default transaction semantics, as defined in the package `net.jini.core.transaction` and described in the *Jini™Transaction Specification.*

Entries written into a JavaSpaces service are governed by a lease, as defined in the package `net.jini.core.lease` and described in the *Jini™Distributed Lease Specification.*

1. The term “space” is used to refer to a JavaSpaces service implementation.

1.2.1 Distributed Persistence

Implementations of JavaSpaces technology provide a mechanism for storing a group of related objects and retrieving them based on a value-matching lookup for specified fields. This allows a JavaSpaces service to be used to store and retrieve objects on a remote system.

1.2.2 Distributed Algorithms as Flows of Objects

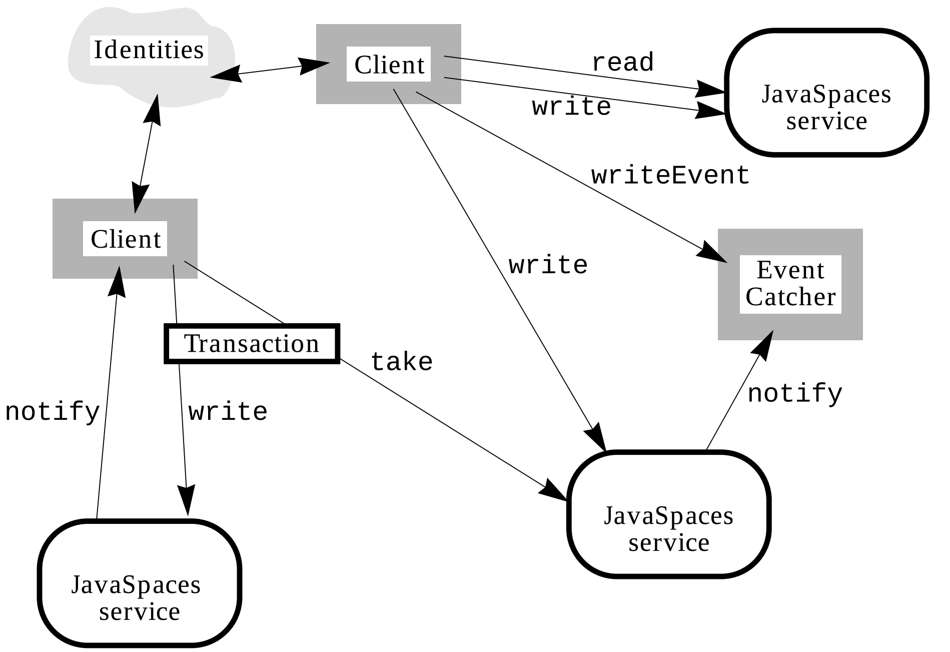
Many distributed algorithms can be modeled as a flow of objects between participants. This is different from the traditional way of approaching distributed computing, which is to create method-invocation-style protocols between participants. In this architecture's "flow of objects" approach, protocols are based on the movement of objects into and out of implementations of JavaSpaces technology.

For example, a book ordering system might look like this:

- A book buyer wants to buy 100 copies of a book. They write a request for bids into a particular public JavaSpaces service.
- The broker runs a server that takes those requests out of the space and writes them into a JavaSpaces service for each book seller who registered with the broker for that service.
- A server at each book seller removes the requests from its JavaSpaces service, presents the request to a human to prepare a bid, and writes the bid into the space specified in the book buyer's request for bids.
- When the bidding period closes, the buyer takes all the bids from the space and presents them to a human to select the winning bid.

A method-invocation-style would create particular remote interfaces for these interactions. With a flow-of-objects approach, only one interface is required—the `net.jini.space.JavaSpace` interface.

In general, the JavaSpaces application world looks like this:



Clients perform operations that map entries or templates onto JavaSpaces services. These can be singleton operations (as with the upper client), or contained in transactions (as with the lower client) so that all or none of the operations take place. A single client can interact with as many spaces as it needs to. Identities are accessed from the security subsystem and passed as parameters to method invocations. Notifications go to event catchers, which may be clients themselves, or proxies for a client (such as a store-and-forward mailbox).

1.3 Benefits

JavaSpaces services are tools for building distributed protocols. They are designed to work with applications that can model themselves as flows of objects through one or more servers. If your application can be modeled this way, JavaSpaces technology will provide many benefits.

JavaSpaces services can provide a reliable distributed storage system for the objects. In the book buying example, the designer of the system had to define the protocol for the participants and design the various kinds of entries that must be passed around. This effort is akin to designing the remote interfaces that an equivalent customized service would require. Both the JavaSpaces system solution and the customized solution would require someone to write the code that presented requests and bids to humans in a GUI. And in both systems, someone would have to write code to handle the seller's registrations of interest with the broker.

The server for the model that uses the JavaSpaces API would be implemented at that point.

The customized system would need to implement the servers. These servers would have to handle concurrent access from multiple clients. Someone would need to design and implement a reliable storage strategy that guaranteed the entries written to the server would not be lost in an unrecoverable or undetectable way. If multiple bids needed to be made atomically, a distributed transaction system would have to be implemented.

All these concerns are solved in JavaSpaces services. They handle concurrent access. They store and retrieve entries atomically. And they provide an implementation of the distributed transaction mechanism.

This is the power of the JavaSpaces technology architecture — many common needs are addressed in a simple platform that can be easily understood, and used in powerful ways.

JavaSpaces services also help with data that would traditionally be stored in a file system, such as user preferences, email messages, images, and so on. Actually this is not a different use of a JavaSpaces service. Such uses of a file system can equally be viewed as passing objects that contain state from one external object (the image editor) to another (the window system that uses the image as a screen background). And JavaSpaces services enhance this functionality because they store objects, not just data, so the image can have abstract behavior, not just information that must be interpreted by some external application(s).

JavaSpaces services can provide distributed *object* persistence with objects in the Java programming language. Because code written in the Java programming language is downloadable, entries can store objects whose behavior will be transmitted from the writer to the readers, just as in an RMI using Java technology. An entry in a space may, when fetched, cause some active behavior in the reading client. This is the benefit of storing objects, not just data, in an accessible repository for distributed cooperative computing.

1.4 JavaSpaces Technology and Databases

A JavaSpaces service can store persistent data which is later searchable. But a JavaSpaces service is not a relational or object database. JavaSpaces services are designed to help solve problems in distributed computing, not to be used primarily as a data repository (although there are many data storage uses for JavaSpaces applications). Some important differences are:

- Relational databases understand the data they store and manipulate it directly via query languages. JavaSpaces services store entries that they understand only by type and the serialized form of each field. There are no general queries in the JavaSpaces application design, only “exact match” or “don’t care” for a given field. You design your flow of objects so that this is sufficient and powerful.
- Object databases provide an object oriented image of stored data that can be modified and used, nearly as if it were transient memory. JavaSpaces systems do not provide a nearly-transparent persistent/transient layer, and work only on copies of entries.

These differences exist because JavaSpaces services are designed for a different purpose than either relational or object databases. A JavaSpaces service can be used for simple persistent storage, such as storing a user’s preferences that can be looked up by the user’s ID or name. JavaSpaces service functionality is somewhere between that of a filesystem and a database, but it is neither.

1.5 JavaSpaces System Design and Linda[1] Systems

The JavaSpaces system design is strongly influenced by Linda systems, which support a similar model of entry-based shared concurrent processing. Our references (§4) include several that describe Linda-style systems.

No knowledge of Linda systems is required to understand this specification. This section discusses the relationship of JavaSpaces systems with respect to Linda systems for the benefit of those already familiar with Linda programming. Other readers should feel free to skip ahead.

JavaSpaces systems are similar to Linda systems in that they store collections of information for future computation and are driven by value-based lookup. They differ in some important ways:

- Linda systems have not used rich typing. JavaSpaces systems take a deep concern with typing from the Java platform type-safe environment. In JavaSpaces systems, entries themselves, not just their fields, are typed — two different entries with the same field types but with different data types for the Java programming language are different entries. For example, an entry that had a string and two double values could be either a named point or a named vector. In JavaSpaces systems these two entry types would have specific different classes for the Java platform, and templates for one type would never match the other, even if the values are compatible.
- Entries are typed as objects in the Java programming language, so they may have methods associated with them. This provides a way of associating behavior with entries.
- As another result of typed entries, JavaSpaces services allow matching of subtypes — a template match can return a type that is a subtype of the template type. This means that the read or take may return more states than anticipated. In combination with the previous point, this means that entry behavior can be polymorphic in the usual object-oriented style that the Java platform provides.
- The fields of entries are objects in the Java programming language. Any object data type for the Java programming language can be used as a template for matching entry lookups as long as it has certain properties.

1. "Linda" is the name of a public domain technology originally propounded by Dr. David Gelertner of Yale University. "Linda" is also claimed as a trademark for certain goods by Scientific Computing Associates, Inc. This discussion refers to the public domain "Linda" technology.

This means that computing systems constructed using the JavaSpaces API are object-oriented from top to bottom, and behavior-based (agent-like) applications can use JavaSpaces services for co-ordination.

- Most environments will have more than one JavaSpaces service. Most Linda tuple spaces have one tuple space for all cooperating threads. So transactions in the JavaSpaces system can span multiple spaces (and even non-JavaSpaces system transaction participants).
- Entries written into a JavaSpaces service are leased. This helps keep the space free of debris left behind due to system crashes and network failures.
- The JavaSpaces API does not provide an equivalent of "eval" because it would require the server to execute arbitrary computation on behalf of the client. Such a general compute server system has its own large number of requirements (such as security and fairness).

On the nomenclature side, the JavaSpaces technology API uses a more accessible set of terms than the traditional Linda terms. The term mappings are "entry" for "tuple", "value" for "actual", "wildcard" for "formal", "write" for "out", and "take" for "in". So the Linda sentence "When you 'out' a tuple make sure that actuals and formals in 'in' and 'read' can do appropriate matching" would be translated to "When you write an entry make sure that values and wildcards in 'take' and 'read' can do appropriate matching."

1.6 Goals & Requirements

The goals for the design of JavaSpaces technology are:

- Provide a platform for designing distributed computing systems that simplifies the design and implementation of those systems.
- The client side should have few classes, both to keep the client-side model simple, and to make downloading of the client classes quick.
- The client side should have a small footprint, because it will run on computers with limited local memory.
- A variety of server implementations should be possible, including relational database storage and object-oriented database storage.
- It should be possible to create a replicated JavaSpaces service.

The requirements for JavaSpaces application clients are:

- It must be possible to write a 100% Pure Java client.
- Clients must be oblivious to the implementation details of the server. The same entries and templates must work in the same ways no matter which server is used.

1.7 Dependencies

This document relies upon the following other specifications:

- Java Remote Method Invocation Specification
- Java Object Serialization Specification
- Jini™ Entry Specification
- Jini™ Entry Utilities Specification
- Jini™ Distributed Event Specification
- Jini™ Distributed Leasing Specification
- Jini™ Transaction Specification

1.8 Changes Since Beta

The following changes have been made to this specification since the beta release:

- The types `Entry` and `UnusableEntryException` have been moved to the `net.jini.core.entry` package, the type `AbstractEntry` has been moved to the `net.jini.entry` package, and the descriptions have been moved into separate specifications (*Jini™ Entry Specification* and *Jini™ Entry Utilities Specification*).
- A timeout of zero in `read` or `take` now means to wait no time at all. The `NO_WAIT` constant has been changed to zero. To wait infinite time, use `Long.MAX_VALUE` as a timeout value (§2.4).
- The semantics of event notification retry have been loosened to allow for implementation variance (§2.7).
- The serialized forms of the relevant classes are now described (§2.9).
- The Introduction chapter has been eliminated.

1.9 Comments

Please direct comments to `js-comments@jse.east.sun.com`

Operations 2

There are four primary kinds of operations that you can invoke on a JavaSpaces service. Each operation has parameters that are entries, including some that are templates, which are a kind of entry. This chapter describes entries, templates, and the details of the operations, which are:

- `write`— Write the given entry into this JavaSpaces service.
- `read`— Read an entry from this JavaSpaces service that matches the given template.
- `take`—Read an entry from this JavaSpaces service that matches the given template, removing it from this space.
- `notify`—Notify a specified object when entries that match the given template are written into this JavaSpaces service.

As used in this document, the term "operation" refers to a single invocation of a method; thus, for example, two different take operations may have different templates.

2.1 Entries

The types `Entry` and `UnusableEntryException` that are used in this specification are from the package `net.jini.core.entry`, and are described in detail in the *Jini™Entry Specification*. In the terminology of that specification, `write` is a store operation; `read` and `take` are combination search and fetch operations, and notify sets up repeated search operations as entries are written to the space.

2.2 net.jini.space.JavaSpace

All operations are invoked on an object that implements the `JavaSpace` interface. For example, the following code fragment would write an entry of type `AttrEntry` into the JavaSpaces service referred to by the identifier `space`:

```
JavaSpace space = getSpace();
AttrEntry e = new AttrEntry();
e.name = "Duke";
e.value = new GIFImage("dukeWave.gif");
space.write(e, null, 60 * 60 * 1000);// one hour
// lease is ignored -- one hour will be enough
```

The `JavaSpace` interface is:

```
package net.jini.space;

import java.rmi.*;
import net.jini.core.event.*;
import net.jini.core.transaction.*;
import net.jini.core.lease.*;

public interface JavaSpace {
    Lease write(Entry e, Transaction txn, long lease)
        throws RemoteException, TransactionException;
    public final long NO_WAIT = 0; // don’t wait at all
    Entry read(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
            RemoteException, InterruptedException;
    Entry readIfExists(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
            RemoteException, InterruptedException;
    Entry take(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
            RemoteException, InterruptedException;
    Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
            RemoteException, InterruptedException;
    EventRegistration notify(Entry tmpl, Transaction txn,
                         RemoteEventListener listener, long lease,
                         MarshalledObject handback)
        throws RemoteException, TransactionException;
    Entry snapshot(Entry e) throws RemoteException;
}
```

The `Transaction` and `TransactionException` types in the above signatures are imported from `net.jini.core.transaction`. The `Lease` type is imported from `net.jini.core.lease`. The `RemoteEventListener` and `EventRegistration` types are imported from `net.jini.core.event`.

In all methods that have the parameter, `txn` may be `null`, which means that no `Transaction` object is managing the operation (§3).

The `JavaSpace` interface is not a remote interface. Each implementation of a JavaSpaces service exports objects that implement the `JavaSpace` interface locally on the client, talking to the actual JavaSpaces service through an implementation-specific interface. An implementation of any `JavaSpace` method may communicate with a remote JavaSpaces service to accomplish its goal; hence, each method throws `RemoteException` to allow for possible failures. Unless noted otherwise in this specification, when you invoke `JavaSpace` methods you should expect `RemoteException`s on method calls in the same cases where you would expect them for methods invoked directly on an RMI remote reference. For example, invoking `snapshot` may require talking to the remote JavaSpaces service, and so may get a `RemoteException` should the server crash during the operation.

The details of each `JavaSpace` method are given in the sections that follow.

2.2.1 *InternalSpaceException*

The exception `InternalSpaceException` may be thrown by a JavaSpaces service that encounters an inconsistency in its own internal state or is unable to process a request because of internal limitations (such as storage space being exhausted). This exception is a subclass of `RuntimeException`. The exception has two constructors: one that takes a `String` description and the other that takes a `String` and a nested exception; both constructors simply invoke the `RuntimeException` constructor that takes a `String` argument.

```
package net.jini.space;
public class InternalSpaceException extends RuntimeException {
    public final Throwable nestedException;
    public InternalSpaceException(String msg) {…}
    public InternalSpaceException(String msg, Throwable e) {…}
    public printStackTrace() {…}
    public printStackTrace(PrintStream out) {…}
    public printStackTrace(PrintWriter out) {…}
}
```

The `nestedException` field is the one passed to the second constructor, or `null` if the first constructor was used. The overridden `printStackTrace` methods print out the stack trace of the exception, and if `nestedException` is not `null`, print out that stack trace as well.

2.3 `write`

A `write` places a copy of an entry into the given JavaSpaces service. The `Entry` passed to the `write` is not affected by the operation. Each `write` operation places a new entry into the specified space, even if the same `Entry` object is used in more than one `write`.

Each `write` invocation returns a `Lease` object that is `lease` milliseconds long. If the requested time is longer than the space is willing to grant, you will get a lease with a reduced time. When the lease expires, the entry is removed from the space. You will get an `IllegalArgumentException` if the lease time requested is negative.

If a `write` returns without throwing an exception, that entry is committed to the space, possibly within a transaction (§3). If a `RemoteException` is thrown, the `write` may or may not have been successful. If any other exception is thrown, the entry was not written into the space.

Writing an entry into a space may generate notifications to registered objects (§2.7).

2.4 `readIfExists` and `read`

The two forms of the read request search the JavaSpaces service for an entry that matches the template provided as an `Entry`. If a match is found, a reference to a copy of the matching entry is returned. If no match is found, `null` is returned. Passing a `null` reference for the template will match any entry.

Any matching entry can be returned. Successive read requests with the same template in the same JavaSpaces service may or may not return equivalent objects, even if no intervening modifications have been made to the space. Each invocation of read may return a new object even if the same entry is matched in the JavaSpaces service.

A readIfExists request will return a matching entry, or null if there is currently no matching entry in the space. If the only possible matches for the template have conflicting locks from one or more other transactions, the timeout value specifies how long the client is willing to wait for interfering transactions to settle before returning a value. If at the end of that time no value can be returned that would not interfere with transactional state, null is returned. Note that, due to the remote nature of JavaSpaces services, read and readIfExists may throw a RemoteException if the network or server fails prior to the timeout expiration

A read request acts like a readIfExists except that it will wait until a matching entry is found or until transactions settle, whichever is longer, up to the timeout period.

In both read methods, a timeout of NO_WAIT means to return immediately, with no waiting, which is equivalent to using a zero timeout.

2.5 takeIfExists and take

The take requests perform exactly like the corresponding read requests (§2.4), except that the matching entry is removed from the space. Two take operations will never return copies of the same entry, although if two equivalent entries were in the JavaSpaces service the two take operations may return equivalent entries.

If a take returns a non-null value, the entry has been removed from the space, possibly within a transaction (§3). This modifies the claims to once-only retrieval—A take is only considered to be successful if all enclosing transactions commit successfully. If a RemoteException is thrown, the take may or may not have been successful. If an UnusableEntryException is thrown, the take removed the unusable entry from the space; the contents of the exception are as described in the *Entry Specification*. If any other exception is thrown, the take did not occur, and no entry was removed from the space.

With a RemoteException, an entry can be removed from a space and yet never returned to the client that performed the take, thus losing the entry in between. In circumstances where this is unacceptable, the take can be wrapped inside a transaction that is committed by the client when it has the requested entry in hand.

2.6 snapshot

The process of serializing an entry for transmission to a JavaSpaces service will be identical if the same entry is used twice. This is most likely to be an issue with templates that are used repeated to search for entries with `read` or `take`. The client-side implementations of `read` and `take` cannot reasonably avoid this duplicated effort, since they have no efficient way of checking whether the same template is being used without intervening modification.

The `snapshot` method gives the JavaSpaces service implementor a way to reduce the impact of repeated use of the same entry. Invoking `snapshot` with an `Entry` will return another `Entry` object that contains a *snapshot* of the original entry. Using the returned snapshot entry is equivalent to using the unmodified original entry in all operations on the same JavaSpaces service. Modifications to the original entry will not affect the snapshot. You can `snapshot` a `null` template — `snapshot` may or may not return `null` given a `null` template.

An `Entry` object returned from `snapshot` on a particular space is only guaranteed to work with that space. Using the snapshot with any other JavaSpaces service will generate an `IllegalArgumentException` unless the other space can use it because of knowledge about the original JavaSpaces service. The entry returned from `snapshot` will only be equivalent to the original unmodified object when used with the space. It will be a different object from the original, and may or may not have the same hash code, and `equals` may or may not return `true` when invoked with the original object, even if the original object is unmodified.

A snapshot is only guaranteed to work within the virtual machine in which it was generated. If a snapshot is passed to another virtual machine (for example, in a parameter of an RMI call), using it — even with the same JavaSpaces service — may generate an `IllegalArgumentException`.

We expect that an implementation of JavaSpaces technology will return a specialized `Entry` object that represents a pre-serialized version of the object, either in the object itself, or as an identifier for the entry that has been cached on the server. Although the client may cache the snapshot on the server it must guarantee that the snapshot returned to the client code is always valid—the implementation may not throw any exception that indicates that the snapshot has become invalid because it has been evicted from a cache. An implementation that uses a server-side cache must, therefore, guarantee that the snapshot is valid as long as it is reachable (not garbage) in the client, such as by storing enough information in the client to be able to re-insert the snapshot into the server-side cache.

No other method returns a snapshot. Specifically, the return values of the `read` and `take` methods are not snapshots, and are usable with any implementation of JavaSpaces technology.

2.7 *notify*

A `notify` request invoked on a template registers interest in future incoming entries, to the specified JavaSpaces service, that match the template. Matching is done as it is for `read`. The `notify` method is a particular registration method under the *Jini™Distributed Event Specification*. When matching entries arrive, the specified `RemoteEventListener` will eventually be notified. When you invoke `notify` you provide an upper bound on the lease time, which is how long you want the registration to be remembered by the server. The server decides the actual time for the lease. You will get an `IllegalArgumentException` if the lease time requested is not `Lease.ANY` and is negative. The lease time is expressed in the standard millisecond units, although actual lease times will usually be of much larger granularity. A lease time of `Lease.FOREVER` is a request for an indefinite lease; if the server chooses not to grant an indefinite lease it will return a bounded (non-zero) lease.

Each `notify` returns a `net.jini.core.event.EventRegistration` object. When an object is written that matches the template supplied in the `notify` invocation, the listener's `notify` method is eventually invoked, with a `RemoteEvent` object whose `evID` is the value returned by the `EventRegistration` object's `getEventID` method, `fromWhom` being the JavaSpaces service, `seqNo` being a monotonically increasing number, and whose `getRegistrationObject` being that passed as the `handback` parameter to `notify`. If you get a notification with a sequence number of 103 and the `EventRegID` object's current sequence number is 100, there will have been three matching entries written since you invoked `notify`. You may or may not have received notification of the previous entries due to network failures or the space compressing multiple matching entry events into a single call.

If the transaction parameter is `null`, the listener will be notified when matching entries are written either under a `null` transaction or when a transaction commits. If an entry is written under a transaction and then taken under that same transaction before the transaction is committed, listeners registered under a `null` transaction will not be notified of that entry.

If the transaction parameter is not `null`, the listener will be notified of matching entries written under that transaction in addition to the notifications it would receive under a `null` transaction. A `notify` made with a non-`null` transaction is implicitly dropped when the transaction completes.

The request specified by a successful `notify` is as persistent as the entries of the space. They will be remembered as long as an un-taken entry would be, until the lease expires, or until any governing transaction completes, whichever is shorter.

The server will make a "best effort" attempt to deliver notifications. The server will retry at most until the notification request's lease expires. Notifications may be delivered in any order.

See the *Jini™Distributed Event Specification* for details on the event types.

2.8 Operation Ordering

Operations on a space are unordered. The only view of operation order can be a thread's view of the order of the operations it performs. A view of inter-thread order can be imposed only by cooperating threads that use an application-specific protocol to prevent two or more operations being in progress at a single time on a single JavaSpaces service. Such means are outside the purview of this specification.

For example, given two threads *T* and *U*, if *T* performs a `write` operation and *U* performs a `read` with a template that would match the written entry, the `read` may not find the written entry even if the `write` returns before the `read`. Only if *T* and *U* cooperate to ensure that the `write` returns before the `read` commences would the `read` be ensured the opportunity to find the entry written by *T* (although it still may not do so because of an intervening `take` from a third entity).

2.9 Serialized Form

The `serialVersionUID` of `InternalSpaceException` is -4167507833172939849L. The only serialized field is the declared public field.

Transactions 3

The JavaSpaces API uses the package `net.jini.core.transaction` to provide basic atomic transactions that group multiple operations across multiple JavaSpaces services into a bundle that acts as a single atomic operation. JavaSpaces services are actors in these transactions; the client can be an actor as well, as can any remote object that implements the appropriate interfaces.

Transactions wrap together multiple operations. Either all modifications within the transactions will be applied or none will, whether the transaction spans one or more operations and/or one or more JavaSpaces services.

The transaction semantics described here conform to the default transaction semantics defined in the *Jini™ Transaction Specification*.

3.1 Operations Under Transactions

Any `read`, `write`, or `take` operations that have a `null` transaction act as if they were in a committed transaction that contained exactly that operation. For example, a `take` with a `null` transaction parameter performs as if a transaction was created, the `take` performed under that transaction, and then the transaction was committed. Any `notify` operations with a `null` transaction apply to `write` operations that are committed to the entire space.

Transactions affect operations in the following ways:

- `write`: An entry written is not visible outside its transaction until the transaction successfully commits. If the entry is taken within the transaction, the entry will never be visible outside the transaction and will not be added to the space when the transaction commits. Specifically, the entry will not generate notifications to listeners not registered under the writing transaction. Entries written under a transaction that aborts are discarded.
- `read`: A `read` may match any entry written under that transaction or in the entire space. A JavaSpaces service is not required to prefer matching entries written inside the transaction to those in the entire space. When read, an entry is added to the set of entries read by the provided transaction. Such an entry may be read in any other transaction to which the entry is visible, but cannot be taken in another transaction.
- `take`: A `take` matches like a `read` with the same template. When taken, an entry is added to the set of entries taken by the provided transaction. Such an entry may not be read or taken by any other transaction.
- `notify`: A `notify` performed under a `null` transaction applies to `write` operations that are committed to the entire space. A `notify` performed under a non-`null` transaction additionally provides notification of writes performed within that transaction. When a transaction completes, any registrations under that transaction are implicitly dropped. When a transaction commits, any entries that were written under the transaction (and not taken) will cause appropriate notifications for registrations that were made under a `null` transaction.

If a transaction aborts while an operation is in progress under that transaction, the operation will terminate with a `TransactionException`. Any statement made in this chapter about `read` or `take` apply equally to `readIfExists` or `takeIfExists`, respectively.

3.2 Transactions and ACID Properties

The ACID properties traditionally offered by database transactions are preserved in transactions on JavaSpaces systems. The ACID properties are:

- *Atomicity:* All the operations grouped under a transaction occur or none of them do.

- *Consistency:* The completion of a transaction must leave the system in a consistent state. Consistency includes issues known only to humans, such as that an employee should always have a manager. The enforcement of consistency is outside of the transaction—a transaction is a tool to allow consistency guarantees, and not itself a guarantor of consistency.
- *Isolation:* Ongoing transactions should not affect each other. Any observer should be able to see other transactions executing in some sequential order (although different observers may see different orders).
- *Durability:* The results of a transaction should be as persistent as the entity on which the transaction commits.

The timeout values in `read` and `take` allow a client to trade full isolation for liveness. For example, if a `read` request has only one matching entry, and that entry is currently locked in a `take` from another transaction, `read` would block indefinitely if the client wanted to preserve isolation. Since completing the transaction could take an indefinite amount of time, a client may choose instead to put an upper bound on how long it is willing to wait for such isolation guarantees, and instead proceed to either abort its own transaction or ask the user whether to continue or whatever else is appropriate for the client.

Persistence is not a required property of JavaSpaces technology implementations. A transient implementation that does not preserve its contents between system crashes is a proper implementation of the `JavaSpace` contract, and may be quite useful. If you choose to perform operations on such a space, your transactions will guarantee as much durability as the JavaSpaces service allows for all its data, which is all that any transaction system can guarantee.

References and Further Reading 4

4.1 References

[1] *A Note on Distributed Computing*, Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. Sun Microsystems Laboratories technical report SMLI TR-94-29, http://www.sunlabs.com/technical-reports/1994/abstract-29.html

[2] *Java Remote Method Invocation Specification*, http://java.sun.com/products/jdk/rmi

[3] *Java Object Serialization Specification*, http://java.sun.com/products/jdk/rmi

[4] *Jini™ Distributed Event Specification*

[5] *Jini™ Distributed Leasing Specification*

[6] *Jini™ Transaction Specification*

4.2 Further Reading

4.2.1 Linda Systems

[8] *How to Write Parallel Programs: A Guide to the Perplexed*, Nicholas Carriero and David Gelernter, *ACM Computing Surveys*, Sept., 1989.

[9] *Generative Communication in Linda,* David Gelernter, *ACM Transactions on Programming Languages and Systems,* Vol. 7, No. 1, pp. 80-112 (January 1985)

[10] *Persistent Linda: Linda + Transactions + Query Processing,* Brian G. Anderson, Dennis Shasha, *Proceedings of the 13th Symposium on Fault-Tolerant Distributed Systems,* 1994.

[11] *Adding Fault-tolerant Transaction Processing to LINDA,* Scott R. Cannon, David Dunn, *Software—Practice and Experience,* Vol. 24(5), pp. 449-446 (May 1994)

[12] *ActorSpaces: An Open Distributed Programming Paradigm,* Gul Agha, Christian J. Callsen, University of Illinois at Urbana-Champaign, UILU-ENG-92-1846,

4.2.2 The Java Platform

[13] *The Java Language Specification*, James Gosling, Bill Joy, Guy Steele, Addison-Wesley

[14] *The Java Virtual Machine Specification*, Tim Lindholm, Frank Yellin, Addison-Wesley

[15] *The Java Class Libraries,* Patrick Chan, Rosanna Lee, Addison-Wesley

4.2.3 Distributed Computing

[16] *Distributed Systems,* Sape Mullender, Addison-Wesley

[17] *Distributed Systems: Concepts and Design,* George Coulouris, Jean Dollimore, Tim Kindberg, Addison-Wesley

[18] *Distributed Algorithms,* Nancy A. Lynch, Morgan Kaufmann