JavaSpace™ Specification
Revision 0.4

June 27, 1997 4:28 pm

The JavaSpace™ 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 JavaSpace 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 JavaSpace system implement distributed persistence for you.

Revision 0.4, June 27, 1997 4:28 pm

Table of Contents

About This Document 0

0.1 Status

This document is the fourth release of the specification for JavaSpaces. The primary purpose of publishing at this time is to gather comments from a wider audience on the design and utility of JavaSpaces. As a welcome side effect, we will also gather data on the utility of this document itself—we encourage editorial as well as technical comment. Obviously, all details are subject to change without notice.

This revision has changes marked with change bars (as in this paragraph). Major changes are enumerated below (§0.4).

0.2 Annotations

Note – In this document you will see several paragraphs in this style. These are areas where we specifically invite comment. Consider them as “notes to reviewers.”

0.3 Comments

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

0.4 Overview of Changes From Version 0.3

- A discussion on JavaSpace's relationship to Linda-style systems has been added (§1.5).
- The implementation of `Entry.rep` has been elided, as it was not contractual (§2.1).
- The definition of `UnusableEntryException` has expanded to include how it is used for a completely unusable entry. (§2.1.1).
- A description of `InternalSpaceException` has been added (§2.2.1).
- A `notify` under a non-`null` transaction will get all notifications that would happen under a `null` transaction in addition to those for the provided transaction (§2.7).
- A `renew` call has been so you can renew an existing notification request instead of creating a new one (§2.8).
- The method `cancelInterest` has been renamed `cancel` (§2.9).
- Methods have been added for JavaSpace implementors to get and set an `EntryRep` object's identifier.
- The `AdminIterator` interface is no longer `Remote`, so that a space can implement it with a client-side proxy to a space-specific remote interface. Unfortunately this means that it *must* provide a proxy, even if that proxy just forwards all calls to a remote implementation of `AdminIterator` (§5.5).
- The administrative interface now allows you to shut down a server (§5.6).

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[1] has a remote method invocation system called RMI[2] that lets you approach general distributed computation in Java using techniques natural to the Java language and environment. This is layered on Java's object serialization mechanism[3] to marshall parameters to remote methods into a form that can be shipped across the wire and unmarshalled in the remote server's virtual machine.

These tools, powerful as they are, do not make distributed computation systems easy to design—they merely make them possible to approach. This specification describes the JavaSpace model that is designed to help you solve two related problems: distributed persistence and the design of distributed algorithms.

1.2 The JavaSpace Model and Terms

A JavaSpace holds *entries*. An entry is a typed group of objects, expressed in a Java class that extends the class `jive.javaSpace.Entry`.

1. Hereafter the Java language is called simply "Java", and a JavaSpace server is called simply a "JavaSpace"; "Java" is a trademarks of Sun Microsystems, Inc.

An entry can be *written* into a JavaSpace, which creates a copy of that entry in the JavaSpace that can be used in future lookup operations.

You can look up entries in a JavaSpace 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 JavaSpace 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 JavaSpace.

You can request a JavaSpace to *notify* you when an entry is written that matches a specified template. This is done using the Jive event model contained in the package `jive.events` and described in the Jive event specification (not yet available).

All operations that modify a JavaSpace 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 a 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 JavaSpace can be taken at most once. Note, however, that two or more entries in a JavaSpace may have exactly the same value.

JavaSpaces support a simple transaction mechanism that allow multi-operation and/or multi-space updates to complete atomically. This is done using the Jive transaction model contained in the package `jive.transactions` and described in the Jive transaction specification (not yet available).

Each operation on a JavaSpace can operate under some identity, which is a parameter to the invocation. A space may use this identity to protect access to part or all of the space for some or all operations.

1.2.1 Distributed Persistence

JavaSpaces 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 JavaSpace to be used to store and retrieve objects on a remote system.

1.2.2 Distributed Algorithms as Flows of Objects

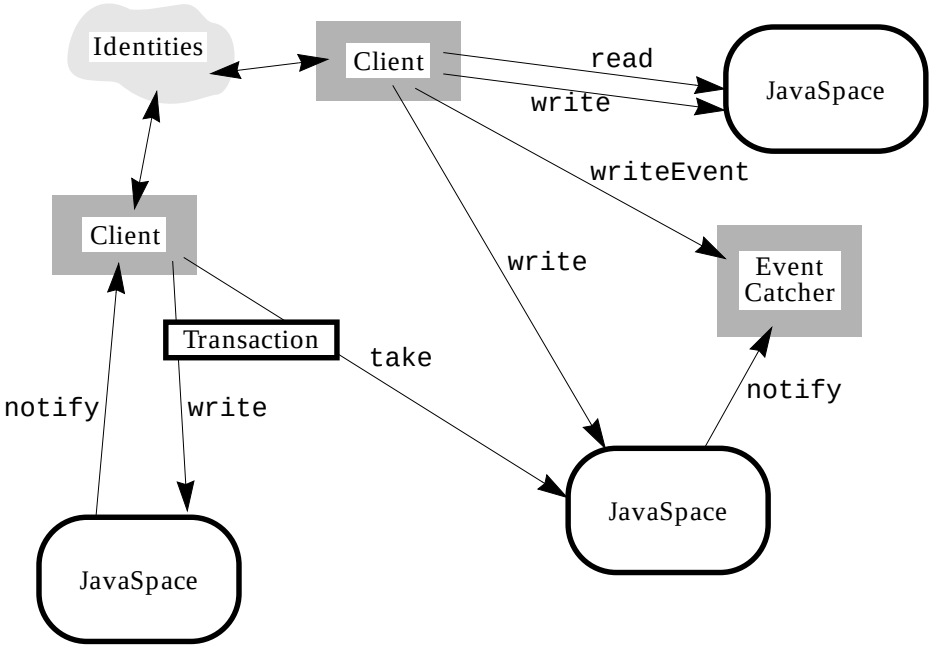
Many distributed algorithms can be modeled as a flow of objects between participants. This is different from the traditional way of approaching distributed computing, which is to create method-invocation-style protocols between participants. In JavaSpace's "flow of objects" approach, it is the movement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

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 JavaSpace.
- The broker runs a server that takes those requests out of the space and writes them into a JavaSpace for each book seller who registered with the broker for that service.
- A server at each book seller removes the requests from its JavaSpace, presents the request to a human to prepare a bid, and writes the bid into the JavaSpace 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 JavaSpace interface.

In general, the JavaSpace world looks like this:



Clients perform operations that map entries or templates onto JavaSpaces. 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 JavaSpaces 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 are a tool 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 will provide many benefits.

JavaSpaces 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 JavaSpace 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 JavaSpaces 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 the JavaSpaces servers. They handle concurrent access. They store and retrieve entries atomically. And they provide implementation of the Java standard distributed transaction mechanism.

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

JavaSpaces also help with data traditionally stored in a file system, such as user preferences, email messages, images, and so on. Actually this is not a different use of JavaSpaces. 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 enhance this 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 provide distributed *object* persistence with Java objects. Because Java code is downloadable, JavaSpace entries can store objects whose behavior will be transmitted from the writer to the readers, just as in Java RMI. An entry in a JavaSpace 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 and Databases

A JavaSpace can store persistent data which is later searchable. But a JavaSpace is neither a relational or object database. JavaSpaces are designed to help solve problems in cooperative distributed computing, not primarily as a data repository (although there are many data storage uses for JavaSpaces) Some important differences are:

- Relational databases understand the data they store and manipulate it directly via query languages. JavaSpaces store entries that they understand only by type and the serialized form of each field. There are no general queries in a JavaSpace, 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 do not provide a nearly-transparent persistent/transient layer, and work only on copies of entries.

These differences exist because the JavaSpace are designed for a different purpose than either relational or object databases. A JavaSpace 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. JavaSpace functionality is somewhere between that of a filesystem and a database, but it is neither.

1.5 JavaSpaces and Linda[1] Systems

The JavaSpace model is strongly influenced by Linda systems, which support a similar model of entry-based shared concurrent processing. Our references (§7) 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 to Linda systems for the benefit of those already familiar with Linda programming. Other readers should feel free to skip ahead.

JavaSpaces 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:

1. Linda is a registered trademark of Scientific Computing Associates.

- Linda systems have not used rich typing. JavaSpaces take a deep concern with typing from the Java type-safe environment. Entries themselves, not just their fields, are typed—two different entries with the same field types but with different Java types are different tuples. For example, an entry that had a string and two double values could be either a named point or a named vector. In JavaSpaces these two entry types would have specific different Java classes, and templates for one type would never match the other, even if the values are compatible (§2.3).
- Entries are typed as Java objects, so they may have methods associated with them. This provides a way of associating behavior with entries (§2.3).
- As another result of typed entries, JavaSpaces allow matching of subtypes—a template match can return a type that is a subtype of the template type (§2.3). This means that the read or take may return more state than anticipated. In combination with the previous point, this means that entry behavior can be polymorphic in the usual object-oriented style that Java provides.
- The fields of JavaSpace entries are Java objects. Any Java object type can be used for matching lookups as long as it has certain properties (§2.3). This means that computing systems done using JavaSpaces are object-oriented from top to bottom, and behavior-based (agent-like) applications can use JavaSpaces for coordination.
- Linda systems provide blocking read and take operations for coordination. JavaSpaces cannot do so, since this would require a blocked thread in the server for each blocked client waiting for a read or take to complete, putting too large a burden on the server. Instead we provide notification when potentially matching entries are written, and require the client to attempt a take or read (§2.7).
- Transactions in a JavaSpace are prevented from seeing results of another transaction's modifications by a thrown exception that does not abort the transaction (§3.3). In existing Linda-style systems with transactions (as in almost all database systems), this isolation is achieved by blocking a thread that attempts conflicting access, and unresolvable conflict aborts at least one of the transactions. Non-blocking prevention of transaction conflicts is
- JavaSpaces are multiple—most Linda tuple spaces have one tuple space for all cooperating threads. So JavaSpace transactions span multiple spaces (and even non-JavaSpace transaction participants).

- JavaSpaces do 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, JavaSpace 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 of the JavaSpace design 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, since it will run on computers with limited local memory.
- A variety of server implementations should be possible, including relational database storage, object-oriented database storage.
- It should be possible to create replicated JavaSpace implementations.

The requirements for JavaSpace are:

- The client side of JavaSpaces must be 100% Pure Java.
- 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:

- RMI

- Object Serialization
- Distributed Events (to be delivered)
- Distributed Transactions (to be delivered)

Entries, Templates, and Operations 2

There are four primary operations that you can invoke on a JavaSpace. These operations have 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 JavaSpace
- `read`—read an entry from this JavaSpace that matches the given template.
- `take`—read an entry from this JavaSpace that matches the given template, removing it from this JavaSpace.
- `notify`—notify a specified object when entries that match the given template are written into this JavaSpace.

2.1 `Entry` and `EntryRep`

An entry is a typed group of object references represented by a Java class that is a direct or indirect subtype of `jive.javaSpace.Entry`. Two different entries have the same type if and only if they are of the same Java class.

All fields in an entry must be public, and they must all be references to `Serializable` objects. Entries may not have fields of primitive type (`int`, `boolean`, ...), although the objects they refer to may have primitive and non-public fields.

Entries are not directly written into JavaSpaces, nor are they directly returned by them. Entries flow to and from a JavaSpace wrapped inside `EntryRep` objects that contain a serialized form of the entry that is suitable for JavaSpace storage and retrieval. The public part of `jive.javaSpace.Entry` looks like this:

```
public class Entry implements java.io.Serializable, Cloneable {
    public EntryRep rep() throws IllegalArgumentException {…}
}
```

Multiple calls to `rep` on the same `Entry` will return different `EntryRep` objects if any object reachable from the entry has changed state between the calls. Absent any changes of reachable state, `rep` may or may not return different objects.

The relevant public part of `EntryRep` is

```
public final class EntryRep
    implements java.io.Serializable, Cloneable
{
    public EntryRep(Entry e) throws IllegalArgumentException {…}
    public Entry entry() throws UnusableEntryException {…}
}
```

`EntryRep` is `final` because it is a direct communication class between clients and any JavaSpace. In other words, an `EntryRep` that is created by a client can be used with any JavaSpace. The full `EntryRep` class is discussed in (§2.11).

An `EntryRep` object can be used more than once. For example, the same `EntryRep` object can be written into the same JavaSpace as many times as desired. This wwill create multiple entries with the same contents. `Entry` and `EntryRep` are `Serializable` and `Cloneable` for your convenience.

Any attempt to create an `EntryRep` from a malformed entry type (one that has non-public or primitive fields) throws an `IllegalArgumentException`.

2.1.1 `UnusableEntryException`

If the serialized fields of the entry cannot be deserialized for any reason, `entry` throws `UnusableEntryException` (§2.1.1). This will only happen if deserializing a field causes an exception to be thrown.

```
public class UnusableEntryException extends Exception {
    public Entry partialEntry;
    public String[] unusableFields;
    public Throwable[] nestedExceptions;
}
```

The `partialEntry` field will refer to an entry of the type that would have been returned, with all the usable fields filled in. Fields whose deserialization caused an exception will be `null` and have their names listed in the `unusableFields` string array. For each element in `unusableFields`, the corresponding element of `nestedExceptions` will refer to the exception that caused the field to fail deserialization.

If the `EntryRep` is corrupt in such a way as to prevent even an attempt at field deserialization, `partialEntry` and `unusableFields` will both be `null`, and `nestedExceptions` will be a single element array with the offending exception.

The kinds of exceptions that can show up in `nestedExceptions` are:

- `ClassNotFoundException`: The actual class of an object that was serialized cannot be found.
- `InstantiationException`: An object could not be created for a given type.
- `IllegalAccessException`: The field in the entry was either inaccessible or `final`.
- `RemoteException`: When a `RemoteException` is the nested exception of an `UnusableEntryException`, it means that a remote reference in the entry's state is no longer valid (more below). Remote errors on the `read` call itself are passed through by `read`, and so arrive as `RemoteException` objects, not `UnusableEntryException` objects.

Generally speaking, writing a remote reference to a non-persistent server into a JavaSpace is risky. Because entries are stored in serialized form, entries in a JavaSpace will not participate in the garbage collection that keeps such references valid. However, if the reference is not persistent because the server does not export persistent references, that garbage collection is the only way to ensure the ongoing validity of a remote reference. If a field contains a reference to a non-persistent remote object, either directly or indirectly, it is possible that the reference is no longer valid when it is deserialized. If this happens, `EntryRep.entry` throws an `UnusableEntryException` (§2.1.1) with the

exception's `nestedException` field set to the `RemoteException` that signalled this condition. Usually this means that the remote object no longer exists. In such a case, the client code must decide whether to `take` the entry out of the space (if the original operation was a `read`), or to `write` the entry back into the JavaSpace (if the original operation was a `take`) or to leave the JavaSpace as it is. This `take` option, of course, assumes that the lookup fields that match the offending entry uniquely identify that entry—otherwise the `take` might remove a different entry.

Unfortunately, in Java 1.1, the only kind of server type available is *not* persistent. Until a persistent type is added, you will have to handle the above problems with remote references. You may choose instead to have your entries store information sufficient to look up the current reference—such as the registry's host name and the name in the registry—rather than putting raw references into the JavaSpace.

2.2 JavaSpace

All operations are invoked on the JavaSpace. Entries are represented by `EntryRep` objects—entries are not stored directly in the JavaSpace. For example, the following code fragment would write an entry of type `AttrEntry` into the JavaSpace referred to by `space`:

```
JavaSpace space = getSpace();
AttryEntry e = new AttrEntry();
e.name = "Duke";
e.value = new GIFImage("dukeWave.gif");
EntryRep erep = e.rep();  // or new EntryRep(e)
space.write(erep, null, null);
```

The `JavaSpace` interface is:

```
import java.rmi.*;
import jive.events.*;
import jive.transactions.*;

public interface JavaSpace
    extends Remote, TransactionParticipant, TransactionMgr
{
    void write(EntryRep rep, Transaction txn, Identity who)
        throws RemoteException, TransactionException,
            SecurityException;
    EntryRep read(EntryRep tmpl, Transaction txn, Identity who)
        throws RemoteException, TransactionException,
```

```
            SecurityException;
    EntryRep take(EntryRep tmpl, Transaction txn, Identity who)
        throws RemoteException, TransactionException,
            SecurityException;
    EventRegID notify(EntryRep tmpl, EventCatcher catcher,
                      Transaction txn, Identity ident, int lease)
        throws RemoteException, TransactionException,
            SecurityException;
    long renew(long cookie, long extension)
        throws RemoteException, NotRegisteredException;
    void cancel(long cookie)
        throws RemoteException, NotRegisteredException;
}
```

The `TransactionMgr`, `TransactionParticipant`, `Transaction` and `TransactionException` types in the above signatures are imported from `jive.transactions`. The `TransactionMgr` interface provides methods for creating transactions. The `TransactionParticipant` interfaces provides methods for participants in these generated transactions. The `Identity` and `SecurityException` types are imported from `jive.security`. The `EventCatcher` and `EventRegID` types are imported from `jive.events`.

Each operation is performed under an identity that is a security principal. The JavaSpace can vet each operation against this identity, throwing `SecurityException` to prevent unwanted access. The details of what is unwanted is up to the JavaSpace implementation.

In all methods that have the parameter, `txn` may be `null`, which means that no `Transaction` object is managing the operation (§3). The `ident` object may also be `null`, which means that the operation is not associated with any principal. Such an operation will only succeed if there are no access control restrictions on that operation of the JavaSpace.

The details of each method are given below.

2.2.1 *InternalSpaceException*

The exception `InternalSpaceException` may be thrown by a JavaSpace implementation 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 `RemoteException`. The exception adds no new fields or methods, and has two exceptions: one that

takes a `String` description and the other that takes a `String` and a nested exception; both constructors simply invoke the equivalent `RemoteException` constructors.

2.3 Templates and Matching

The lookup operations (`read`, `take`, and the notification requests) use entry objects of a given type, whose fields can either have *values* (references to objects) or *wildcards* (`null` references). When considering a template *T* as a potential match against a entry *E* in the space, fields with values in *T* must be matched exactly by the value in the same field of *E*. Wildcards in *T* match any value in the same field of *E*.

The values of two fields *match* if the bytes generated by their serialized form match, ignoring differences of serialization stream implementation (such as blocking factors for buffering). A JavaSpace does not use `Object.equals` or any other form of type-specific value matching.

You can write an entry that has a `null`-valued field, but you cannot match explicitly on a `null` value in that field, since `null` signals a wildcard field. If you have a field that may be variously `null` or not in entries, and want to match on whether that field is set, you can set the field to `null` in your entry. But if you will need to write templates that distinguish between set and unset values for that field, you will have to add a `Boolean` field that indicates whether the field is set, and use a `Boolean` value for that field in templates.

Each field's serialized form is considered as if it were generated separately, that is, if two fields of the entry refer to the same object (directly or indirectly), the serialized form that is compared for each field will have a separate copy of that object. This is to regularize the serialized form between the template, which may have `null` fields where a matching entry has values. If this were not so, then a template which had a wildcard for the first object would have the full, serialized object where the second reference appears, but the entry would have only a back-reference to the first object in that place. For details, see the specification on Object Serialization.

The type of *E* can be a subtype of the type of *T*, in which case all fields added by the subtype are considered to be wildcards. This enables a template to match entries of any of its subtypes.

An entry that has no wildcards is a valid template.

2.4 write

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

Each field of the entry is independent, and is therefore serialized separately. This means that if two fields of an entry refer to the same object, directly or indirectly, the entry actually written into the JavaSpace will have two independent copies of that object. Consequently, when the entry is read back from the JavaSpace, the reconstituted entry will have independent copies of that object. (This is only true of different fields of an entry; if an object graph of a particular field refers to the same object twice, the graph will be serialized and reconstituted with a single copy of that object.)

This is unexpected behavior, but it is both logically correct, and practically advantageous. Logically, the fields can refer to object graphs, but the entry is not itself a graph of objects, and so should not be reconstructed as one. An entry (as used with respect to a JavaSpace) is a set of separate fields, not a unit of its own. From a practical standpoint, viewing an entry as a single graph of objects requires a JavaSpace to parse and understand the serialized form, because the ordering of objects in the written entry will be different from that in a template that can match it. Entry is a Serializable class so that client code can store them into files or transmit them across RMI calls. The entry as a whole is not serialized when used with JavaSpaces.

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 JavaSpace may generate notifications to registered objects (§2.7).

2.5 read

A read request will search the JavaSpace for an entry that matches (§2.3) the template provided as an EntryRep. If a match is found, a reference to a copy of that entry is returned. If no match is found, null is returned.

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

(See the discussion in `write` (§2.4) about independent serialization and deserialization of fields.)

2.6 `take`

A `take` request performs exactly like a `read` (§2.5), except that the matching entry is removed from the JavaSpace. Two take operations will never return copies of the same entry, although if two equivalent entries were in the JavaSpace the two takes 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 are commit successfully. If a `RemoteException` is thrown, the `take` may or may not have been successful. 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 JavaSpace 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 taken entry in hand.

(See the discussion in `write` (§2.4) about independent serialization and deserialization of fields.)

2.7 `notify`

A `notify` request invoked on a template registers interest in future incoming entries to the specified JavaSpace that match the template. When matching entries arrive, the specified `EventCatcher` will 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. If the specified lease time is zero, the server uses its prefered lease time. You will get an `IllegalArgumentException` if the lease time requested is negative.

Each `notify` returns an `EventRegID` object that has an event ID that will be passed along with the notification, a cancellation cookie that can be used to unregister interest in the event, and the actual lease length granted by the space. The relevant part of `EventRegID` class looks like this:

```
public class EventRegID implements java.io.Serializable {
    public long getEventID() {…}
    public long getCancelCookie() {…}
    public long getLeaseLength() {…}
    public long getCurrentSeqNo() {…}
}
```

The `jive.events.EventCatcher` interface—the interface that must be supported by the notified object—looks like this:

```
public interface EventCatcher extends Remote {
    void notify(long evID, Remote fromWhom, long seqNo)
        throws EventUnknownException, RemoteException;
}
```

When a matching object is written, the `notify` method is invoked on the catcher, with `evID` being the value returned by the `EventRegID` object's `getEventID` method, `fromWhom` being the JavaSpace, and the `seqNo` being a monotonically increasing number. If you get a notification with a `seqNo` of, say, three, there will have been three previous calls with values zero, one, and two (although you may not have received them, or you may receive call number three more than once).

If the transaction parameter is `null`, the catcher 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, catchers registered under a `null` transaction will not be notified of that entry.

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

It is a good idea to replace a `notify` request with a new one before you get too close to the end of the lease period. Once the new request has been successful, it would be polite to cancel interest in the older request. The client is responsible for tracking lease expiration because it has enough information to do so, and only the client knows whether a given expiration is acceptable. This also reduces the amount of network traffic compared to having the server notify the client that a lease is about to expire.

The server will make a "best effort" attempt to deliver notifications. The server will retry at least until the notification request's lease expires.

Note – We are working on ways to address the starvation and fairness issues when the notification will result in a `take` by the client. If many clients want to take the same entry, clients that are far away and/or far down on the notification list may get starved, while all others will do a failing `take`.

Note – Descriptions of `jive.events` types and their behaviors belong in the event specification when that is available.

2.8 `renew`

The cookie returned in the `EventRegID` object by a notification request (§2.7) can be passed to the JavaSpace's `renew` method to renew the interest in a notification. You can pass the extension time you desire; the space will return the time actually granted. If the time passed is zero, the server will use its preferred extension time. An `IllegalArgumentException` is thrown if the extension time is negative. If the cookie is for an unknown notification request (possibly one that has already expired, or whose transaction has completed), a `NotRegisteredException` will be thrown.

2.9 `cancel`

The cookie returned in the `EventRegID` object by a notification request (§2.7) can be passed to the JavaSpace's `cancel` method to cancel future notifications that would result from the request.

2.10 Operation Ordering

Operations on a JavaSpace 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 JavaSpace. 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).

2.11 Implementing `JavaSpace` using `EntryRep`

The client code will operate primarily in terms of `Entry` objects, but interaction with a JavaSpace requires the use of `EntryRep` objects. For clients, the only interesting parts of `EntryRep` are its constructor (also available via the `Entry.rep` method) and its `entry` method.

The other methods of `EntryRep` are primarily of interest to someone implementing a `JavaSpace` interface. They provide access necessary for building data structures that contain `EntryRep` objects that must be searched by templates. The Java language provides no way to limit access only to such server implementations, however, and so these methods are both visible to client code and invokable by them. The problem created is one of "surface area"—the class is more complex to clients than it needs to be, and programmers of client code must be told that these methods are generally useless to them. However, client code can do no harm by invoking the methods.

The full public part of `EntryRep` looks like this:

```
public final class EntryRep
    implements java.io.Serializable, Cloneable
{
    public EntryRep(Entry e) throws IllegalArgumentException {…}
    public Entry entry() throws UnusableEntryException {…}
```

```
        public long id() {…}
        public void id(long newID) {…}
        public int numFields() {…}
        public Long classFor() {…}
        public Long[] superclasses() {…}
        public boolean matches(EntryRep entry) {…}
        public int hashCode() {…}
        public boolean equals(Object that) {…}
    }
```

The constructor and the `entry` method have already been discussed (§2.1). The `id` methods allow you to get or set the objects ID, which is a space relative identifier for use of the space implementation. The `numFields` method returns the number of fields in the entry. `classFor` returns a `Long` object whose value is the serialize UUID for the type of the `Entry` object from which the `EntryRep` was constructed. The `superclasses` method returns an array of `Long` objects with serialize UUIDs for each superclass, up to and including the UUID for the `Entry` class itself.

The `matches` method determines if the entry object passed as a parameter matches the template object on which it is invoked.

The `hashCode` and `equals` methods have standard behavior.

Transactions 3

The JavaSpace facility uses the package `jive.transactions` to provide basic atomic transactions that group multiple operations across multiple JavaSpaces into a bundle that acts as a single atomic operation. JavaSpaces 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.

Note – The transaction specification is not yet finished, so we do not yet know if transactions can be nested. If so, some obvious modifications will be needed.

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 full JavaSpace.

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 catchers 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 full JavaSpace. A JavaSpace is not required to prefer matching entries written inside the transaction to those in the full JavaSpace. 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. A `read` will throw `TransactionConflictException` if all possible matching entries have been taken in other transactions that have not yet completed (§3.2).
- `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. A `take` will throw `TransactionConflictException` if all possible matching entries have been read or taken in other transactions that have not yet completed (§3.2).
- `notify`: A `notify` performed under a `null` transaction applies to `write` operations that are committed to the full JavaSpace. 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 transactions 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, that operation will terminate with a `TransactionException`.

3.2 `TransactionConflictException`

Each entry in a JavaSpace can be in one of three transactional states:

1. Taken by a transaction that has not yet completed

2. Read by one or more transactions that have not yet completed

3. Not involved in any transaction

The `TransactionConflictException` is thrown when the only way to satisfy a `read` request would be to return an entry in state 1, or when the only way to satisfy a `take` request would be to take an entry in state 1 or 2.

```
public class TransactionConflictException
    extends TransactionException
{
    public TransactionConflictException() {…}
    public TransactionConflictException(String msg) {…}
}
```

A thrown `TransactionConflictException` does not abort either transaction in question. The recipient of the exception is allowed to attempt other operations that may satisfy its needs. Cancelling the transaction will be a very common reaction, but the client makes that decision.

3.3 Transactions and ACID Properties

The ACID properties traditionally offered by database transactions are preserved in transactions on JavaSpaces. 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. Each participant should only see inconsistencies resulting from the operations of its own transactions.
- *Durability:* The results of a transaction should be as persistent as the entity on which the transaction commits.

`TransactionConflictException` is thrown when a requested operation would, if satisfied, violate the isolation property. The exception also preserves consistency because no inconsistent state of a transaction will be made visible.

Persistence is not a required property of JavaSpaces. A transient JavaSpace 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 JavaSpace allows for all its data, which is all that any transaction system can guarantee.

Utilities 4

JavaSpaces are designed to be as simple as possible, but no simpler. This section describes utility classes that will help you use JavaSpaces. These are not architectural—JavaSpaces are perfectly usable without them. They are designed to help in certain common tasks that users of JavaSpaces will face. These classes will live in a `util` subpackage of the JavaSpace package.

4.1 Lease Renewal

Registered notifications time out to prevent junk from accumulating in the server (§2.7). Clients must renew their interest by re-invoking `notify` before the lease expires. We will provide a class that keeps renewing leases on a set of templates until told to stop.

Note – Details of this utility will have to wait until the event specification is available.

4.2 JavaSpaceProxy

Clients using a JavaSpace must translate entries to and from `EntryRep` objects. This may be awkward for small, simple uses of JavaSpaces. We will provide a utility class `JavaSpaceProxy` whose calls use `Entry` where `JavaSpace` uses `EntryRep`. Client code can create a `JavaSpaceProxy` object for the JavaSpace it needs to talk to, and then make all calls on the `JavaSpaceProxy` object, which will create `EntryRep` objects on the fly as needed.

```
public class JavaSpaceProxy {
    public JavaSpaceProxy(JavaSpace space) {…}
    public JavaSpace space() {…}
    public boolean retainReps() {…}
    public void retainReps(boolean retain) {…}
    public int retainLimit() {…}
    public void retainLimit(int limit) {…}

    public void write(Entry entry, Transaction txn, Identity ident)
        throws TransactionException, SecurityException,
                RemoteException
    {…}

    public Entry read(Entry tmpl, Transaction txn, Identity ident)
        throws TransactionException, SecurityException,
            RemoteException
    {—}

    public Entry take(Entry tmpl, Transaction txn, Identity ident)
        throws TransactionException, SecurityException,
            RemoteException
    {…}

    public EventRegID
        notify(Entry tmpl, EventCatcher catcher, Identity ident,
                long lease)
        throws SecurityException, RemoteException
    {…}


    public void renew(long cookie)
        throws NotRegisteredException, RemoteException
    {…}

    public void cancel(long cookie)
        throws NotRegisteredException, RemoteException
    {…}
}
```

A `JavaSpaceProxy` is created with a `JavaSpace` to which it forwards operations. The `space` method returns that value.

The method `retainReps` indicates whether the `JavaSpaceProxy` object will remember `EntryRep` objects it has created in the past and reuse them for future invocations of the same `Entry` object. If the `Entry` objects you use as parameters to the JavaSpace are only reused with the same value (for example,

a template that never changes which is used to read entries), you can invoke `retainReps` with `true` and the `EntryRep` objects created will be reused. When you invoke `retainReps` with `false`, any existing cached `EntryRep` objects will be dropped. You can reset the cache of retained objects with two calls to `retainReps`, the first with `false` and the second with `true`.

You can limit the number of cached `EntryRep` objects to a maximum using `retainLimit`. Objects are evicted from the cache on a “least recently used” basis. If a call to `retainLimit` provides a value less than the number currently retained, the cache size will be reduced. You can reset the cache of retained objects with a two calls to `retainLimit`, the first with zero as a parameter and the second with the original value. The default limit is 1000. You can make the cache of effectively unlimited size by using a limit of `Integer.MAX_VALUE`.

Note – When weak references are added to Java we should probably do something with them here.

Note – Other ideas?

Administration 5

The administration of a JavaSpace is dependent upon the implementation of the server. No administrative requirements are true of all JavaSpaces. This chapter discusses administration for our initial implementation of JavaSpace servers.

5.1 Requirements

An administrator of a JavaSpace installation should be able to:

- Freeze operations on the JavaSpace
- Iterate through all entries that match a certain template
- Delete specific entries returned by that iteration
- Get a time stamp on an entry so that staleness can be judged
- Shutting down the server
- Security policy for applying the `Identity` parameters to allowed behavior

5.2 Approach

Administrative tasks will be done using a `JavaSpaceAdmin` interface. Each space can export such an object to those entities who should be allowed to invoke the methods. Each method will also take an `Identity` to authorize each operation.

```
package jive.javaSpace;

import java.rmi.*;
import jive.transactions.*;
import java.security.acl.Acl;

public interface JavaSpaceAdmin
    extends Remote, TransactionParticipant
{
    JavaSpace space() throws SecurityException, RemoteException;

    Transaction freeze(Identity ident)
        throws SecurityException, RemoteException;
    Transaction freeze(Identity ident, long waitFor)
        throws SecurityException, RemoteException;

    AdminIterator
    contents(EntryRep tmpl, Transaction txn, Identity ident)
        throws TransactionException, SecurityException,
               RemoteException;

    void shutDown(Identity ident)
        throws SecurityException, RemoteException;
    void shutDown(Identity ident, long waitFor)
        throws SecurityException, RemoteException;

    Acl acl(Identity ident)
        throws SecurityException, RemoteException;
    void acl(Acl, acl, Identity ident)
        throws SecurityException, RemoteException;
}
```

5.3 space

The `space` method returns the JavaSpace which the `JavaSpaceAdmin` object manages.

5.4 freeze

The `freeze` method has two forms—one that waits indefinitely for existing operations to complete, and one that takes a `long` that says how long to wait for existing operations to complete, after which any operations still in progress will be terminated abruptly. Each invocation of `freeze` returns a transaction

within which normal JavaSpace operations are allowed. Multiple freezing transactions may be in progress simultaneously, but all must be under the same identity as the `freeze` operation that caused the freeze. When all freezing transactions have completed, the JavaSpace will be unfrozen.

If you attempt to freeze a JavaSpace, and generally have permission to do so, but the space is already frozen under another identity, you will get a `FreezeConflictException`, which lists the identity of the freezer. This exposure of identity is necessary so that one admin trying to fix up a broken space will know which other admin is already attempting the fix. The identity is only exposed to other administrators—in order to receive this exception you must be authorized to freeze the space (§5.6).

When `freeze` returns, the JavaSpace server will block incoming requests, and cease sending notifications. When the server is unfrozen, pending incoming requests will continue, and notification delivery will resume, including for operations undertaken during the freezing transactions.

5.5 `contents` and `AdminIterator`

The `contents` method returns an `AdminIterator` object. This will be implemented by each space to perform the following methods:

```
package jive.javaSpace;

import java.rmi.*;

public interface AdminIterator {
    public EntryRep[] nextReps(int maximum) throws RemoteException;
    public void delete(EntryRep) throws RemoteException;
    public long timeStamp(EntryRep rep) throws RemoteException;
}
```

The `nextReps` method takes a count of the maximum number of entries to return, and returns an array of `EntryRep` objects that has up to that many elements. A return value of `null` indicates the end of iteration.

The `delete` method deletes from the space the entry correlating to the given `EntryRep`, which must have been returned by an `AdminIterator` object. A JavaSpace implementation is allowed to throw `IllegalArgumentException` if the `EntryRep` object passed to `delete` was not returned by that same iterator's `nextReps` method and is not a clone or deserialized copy of such an `EntryRep`. An implementation may choose not to throw the exception if every

`EntryRep` object it returns is sufficiently unique to specify a single entry in the JavaSpace. Passing an `EntryRep` object of one space into the `delete` method of another space's `AdminIterator` object generates undefined behavior—if you are lucky, it will throw `IllegalArgumentException`, but it may result in some entry being deleted.

In a formal sense one can view `delete` as more of an efficiency than an enabling technology—clearly the administrative requirement could be met by doing a `take` of appropriate entries. But once administrative code has a direct reference in hand to an entry in the JavaSpace, it is clear that for most implementations it could be made orders of magnitude faster to directly remove elements.

The `timeStamp` method returns a server-relative approximate time stamp of when the entry was written into the space. This time stamp is expressed in the standard Java units of milliseconds, but the space need not be so precise. Time stamps are designed to allow administrative functions to remove stale entries from the space, and so are not required to have millisecond accuracy. The behavior of `timeStamp` with another space's `EntryRep` object is undefined, with the same behavior as `delete` in the same circumstances.

Note that `AdminIterator` is not a remote interface—`contents` must return a local object that communicates with its server on a server-defined remote interface. Many implementations will be able to have `contents` return an object that implements at least `timeStamp` locally, and forwards the other request back to the space.

5.6 `shutDown`

The server can be told to shut down. If the specified identity is authorized to execute this request, the server will immediately stop accepting new requests. If `shutDown` is given only an identity, the shutdown will be immediate, terminating any existing requests. If a time is also specified, existing requests will be given up to `waitFor` milliseconds to complete requests. If `waitFor` is zero, `shutDown` will wait indefinitely for ongoing requests to complete.

5.7 `acl` and Access Control

The `Identity` parameter to the operations will allow the JavaSpace to check operations against a set of principals who are permitted to perform operations. The access control is defined by a `java.security.acl.Acl` object, which has the following permissions:

- `write`—the principal is allowed to write entries into the JavaSpace and request notification of `write` operations.
- `read`—the principal is allowed to read entries from the JavaSpace
- `take`—the principal is allowed to take entries from the JavaSpace and request notification of `take` operations.
- `admin`—the principal is allowed to perform all administrative functions except setting the ACL.
- `acl`—the principal is allowed to change the ACL for the space.

There are two overloads of the `acl` method—one which returns a copy of the space's ACL and the other which provides a replacement ACL which will be copied to be the new ACL object.

Cookbook 6

This chapter shows examples of several common usage patterns for JavaSpace programming. Although not required for the specification, the purpose is to help people use JavaSpaces well, and to provide more background on the expected uses of JavaSpace. As this document matures this chapter will be split off into a separate document

6.1 Write Then Take

Many algorithms are focused around singleton objects that provide a current value for something. Updating singleton objects requires both writing in a new entry with new values and taking out the old entry. The order is important. If you write, then take, there will be no visible inconsistent state—if a client reads the value while both new and old entries are present it will get either the new or the old entry. If the client had read moment earlier it would have gotten the old entry—if it had read moments later it would have gotten the new entry. Which happens is not relevant for singleton entries. If you had, on the other hand, taken before writing, there would be a window where a client's read could fail because no matching entry was available.

For such singleton entries, "write then take" avoids a transaction altogether.

6.2 Entry Splitting

Some entries require a large amount of data, and must be processed by several players in the protocol before they are removed. Such entries can benefit from header entries that contain the modifiable parts of the entry and refer to the rest of the data. For example, a travel expense report may contain a lot of state (records of each expense, including where, why, what type, and so on) that is generally unchanged as the report moves through the approval process. You can create an `ExpenseReport` entry type that holds the report details, and an `ExpenseReportStatus` entry type that refers to a unique key in the `ExpenseReport` and that will hold the current state of the report in the approval process. Then as various players make their approval or rejection marks on the report, only the `ExpenseReportStatus` object must be taken and replaced with updated state, leaving the large `ExpenseReport` object untouched.

6.3 Generations

You may want to update many entries consistently. You can wrap a collection of takes of the old entries and writes of the new entries, but this has a cost of making the entries unavailable during the course of the transaction. You could instead have a generation marker for the entry collection. A `Generation` entry would hold a single marker (say a `String`) that would be stamped on each element of the generation. New entries would then be written under a new generation identifier, and when the new generation was complete, the `Generation` entry would be updated (possibly using “Write Then Take” (6.1)). The older generation could be kept around for a while to allow clients with the older generation marker to work, or these clients could know that a new generation existed when they were unable to get a complete set of entries under the old generation. (If they succeed in getting a complete set of entries under the old generation, they can be considered to have completed before the new generation arrived.)

References and Further Reading 7

7.1 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://chatsubo.javasoft.com/current

[3] *Java Object Serialization Specification*, http://chatsubo.javasoft.com/current

Note – Need references to events and transactions specifications when they are completed.

7.2 Further Reading

7.2.1 Linda Systems

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

[5] *Generative Communication in Linda*, David Gelernter, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80-112 (January 1995)

[6] *Persistent Linda: Linda + Transactions + Query Processing*, Brian G. Anderson, Dennis Shasha,

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

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

7.2.2 The Java Platform

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

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

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

Note – Put in the upcoming RMI book

7.2.3 Distributed Computing

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

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

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