JavaSpaces

by

Jonathan Neitz

Email: neitz@cpsc.ucalgary.ca

Date: February 15, 2000

Abstract:

This paper discusses a new Java technology, JavaSpaces. This technology is based upon the services provided by Sun's jini platform, and Java's serialization and RMI services. Note that although JavaSpaces makes uses of the jini technology it does not require a jini service in order to be utilized. The JavaSpaces technology provides a way to do distributed computing in a practical and easy manner. This paper discusses the JavaSpaces API (basic operations that a JavaSpaces application can perform) along with it's advantages/disadvantages and demonstrates how to use the JavaSpaces technology by providing a detailed example of its use.

Overview:

JavaSpaces technology provides a way for distributed applications to communicate via a space. Objects are either written or read from a space by an application. When an object has been passed into a space it cannot be operated upon. The object once in the space is treated as passive data. In order for an application to modify an object it must be removed from the space, modified and then reinserted. Figure 1 below was taken from the Sun Microsystems website on JavaSpaces. It illustrates the operations that can be preformed on a space using cartoon Dukes.

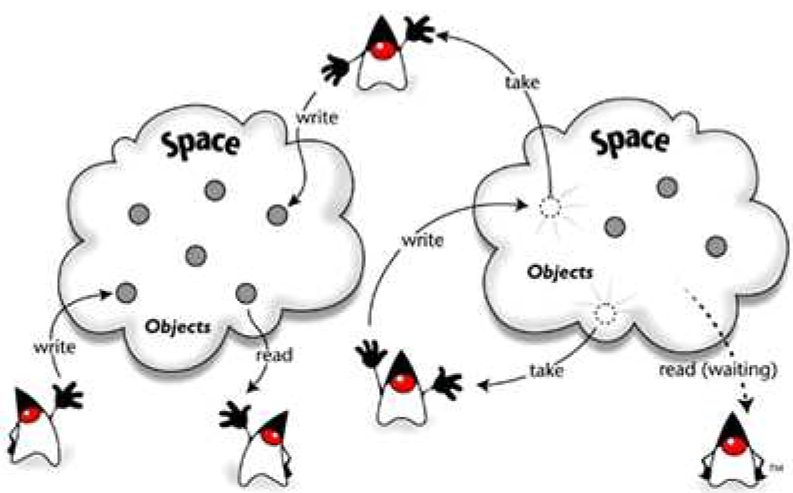

Figure 1: JavaSpaces an overview
Picture from: http://java.sun.com/products/javaspaces/

History:

The concepts and technology behind JavaSpaces are heavily based on previous work done by David Gelenter at Yale University. His concurrent co-ordination programming language, Linda, was based on a concept called a tuple space. A tuple space is a persistent object store upon which a few basic operations can be preformed. The net effect of this approach was that it made distributed computing easy by abstracting away issues like communication between distributed applications. JavaSpaces closely mirrors Linda in its basic operation. Linda has been in use since 1982 and has had a highly successful track record in systems that have made use of it. This bodes well for JavaSpaces as it gives it some needed credibility in the face of other distributed solutions, hopefully making adoption quicker.

JavaSpaces Concepts

JavaSpaces has a few concepts associated with its use. I'm going to give an overview of each of these concepts and explain their role in the JavaSpaces model.

Spaces:

A space is the fundamental concept in JavaSpaces it resembles the Linda concept of tuple spaces. A space acts as a persistent data store for objects and allows applications to communicate across a network by passing objects through this space. One could view it as a kind of shared memory space between different applications. Spaces have a number of properties that facilitate their use in distributing computing environments.

Spaces are:

- *Shared*
  So many remote processes can interact with a space concurrently.

- *Persistent*
  An object written into a space will remain there until a server specified lease time is up (which can be renewed) or the object has been removed from the space.

- *Associative*
  Objects in a space are located by associative lookup. This allows an application to make a request for the properties of an object rather then trying to find an object by certain name. This allows requests for objects to be more general and flexible.

- *Transactionaly secure*
  JavaSpaces used jini transaction mechanisms to allow for atomic operations (transactions will be discussed in more detail later).

- *Executable Objects can be exchanged*
  Because JavaSpaces relies on Java Virtual Machines it is able to pass around objects. Because this model allows for full object orientation throughout the system, once an object has been read out of a space it's methods can be executed, just like other any other Java object.

A space also preserves the ACID properties of traditional database systems. These properties are key in providing a distributed persistent object store. The ACID properties are:

- *Atomicity*
  All grouped operations either occur or do not occur

- *Consistency*
  Completion of a transaction must leave the system in a consistent state.
- *Isolation*
  Ongoing transactions should not affect each other
- *Durability*
  Results of a transaction should be as persistent as the entity being operated upon.

For each listed operation I have included the relevant page from the Sun's documentation on JavaSpaces and Jini. This documentation is as far as I know not publicly available. All of the documentation is available in the Jini SDK and JavaSpaces SDK which are downloadable from sun's website here: http://developer.java.sun.com/developer/products/jini/ The latest Java 1.2 or higher will also be required to run any of the example code.

Entry:

An entry is the only type of object that can be stored within a space. To be an entry, an object must implement the Entry interface found in the net.jini.core.entry.Entry package. Beyond this an Entry is exactly like any other Java object. Thus you can instantiate it, invoke its methods, and assign values to it's public fields.

Entry specification

Template:

A Template is a kind of Entry that is used to locate Entry objects within a space. A Template can have some or all of its fields set to specific values that must be matched exactly. Unset fields are set to null in which case they act like wildcards and can match any given field. Because of the object oriented behavior of JavaSpaces subtypes of objects can also matched. A simple example of a template is to create an Entry object that has a string field. The original Entry object could have the field set to "Hello World". A matching template can be would either not set that field which would match any object with that particular kind of Entry or it's field could be set to "Hello World" providing for an exact match.

Lease:

A lease is an object associated with an Entry that contains a time, specified in milliseconds, that governs how long an object will remain stored in a space. A lease can be renewed or set to an infinite time such that the object will not be removed from the space unless an application specifically removes the object. A benefit of the lease object is that it helps to keep the space clean but having a way to automatically clean up old objects within the space.

Lease specification

Transactions:

A single operation on a space is always secure, as it will either occur or not. The problem that occurs is when you need several operations to occur and if one fails it will effect the rest. Transactions provide a way to group multiple operations together such that they will act as a single operation. Transactions in

JavaSpaces are quite powerful as they are not limited to just one space they can be enacted over multiple spaces.

Transaction specification

JavaSpaces Operations:

There are four primary kinds of operations that can be invoked on a space.

Write:

A write operation will place a given Entry into a space. The object will not be placed directly in the space rather a copy will be made and placed in the system. The written object can be present multiple times within the space. A write returns a Lease object with a time given in milliseconds by the server. A write command can suggest to the server a lease time for the object but ultimately it is the servers decision on what the lease time will be.

Read:

A Read operation searches the space for an Entry that matches a given template, this is done using associative lookup in trying to match the type of template (or subtype) and it’s corresponding fields. An exact match must be found in order for the corresponding Entry to be found. Once this match is found a copy of the object is made so that the requesting application can make use of the object.

Take:

A Take operation performs identical to the read operation except it is destructive and removes the object from the corresponding space.

Notify:

A Notify operation asks the space to notify an object when an Entry matching the given template is added to the space.

Details on the exact method calls can be found on this page for the JavaSpace Interface specification.

Examples:

Here are some practical examples that can give you an idea of what the JavaSpaces technology can be to use for. These are not code examples but merely application examples to give you a better idea of some useful applications of JavaSpaces Technology.

Chat systems

A simple multi-user chat system could be set up using a space. This space could hold messages in a discussion. A person would talk by placing a message object into the space. All other chat members could view the resulting message. In addition the persistence aspect of the space would allow for new comers to the chat board to see the transpired conversation.

Compute servers

A compute server takes computationally intensive jobs and divides the work over a network of computers. By using the JavaSpaces technology a number of tasks can be placed in a space. Each available computer can then take tasks from the space and perform that part of the calculation and place the result back into the space to be read later. This approach scales naturally: it works the same way whether there are 10 computers available or 1,000. The approach also provides natural load balancing, since each computer picks only as much work as it can handle at a given time, with slow computers doing less work and fast computers doing more.

Broker systems

An online auction system could be used to bring buyers and sellers of goods and services together. Using this system a potential buyer could describe an item (such as a car) that they would like to purchase. The buyer could then give a description of the item and the price they were willing to pay and put that information into an entry. Then the resulting wanted-to-buy entry would be written to the space. At the same time, potential sellers could be monitoring the space for the arrival of wanted-to-buy entries that match items in their inventory. When a matching request is found a potential seller writes a bid entry into the space, stating an offering price. As a potential buyer, you continually monitor the space for bids on your outstanding requests, and, when you find one that's acceptable, you remove the bids and contact the seller (possibly through the space via another entry).

Code Example - A general Compute Server.

We are already somewhat familiar with a compute server as it was mentioned as one of the practical examples and I did mention it in the lecture. What we are going to do here is look at the code for a very simple compute server whose tasks consist of simple additions and multiplications.

Our compute server will have three basic parts: a Worker, a Task, and a Master. Figure 2 below gives a nice graphical representation of what we will be doing.

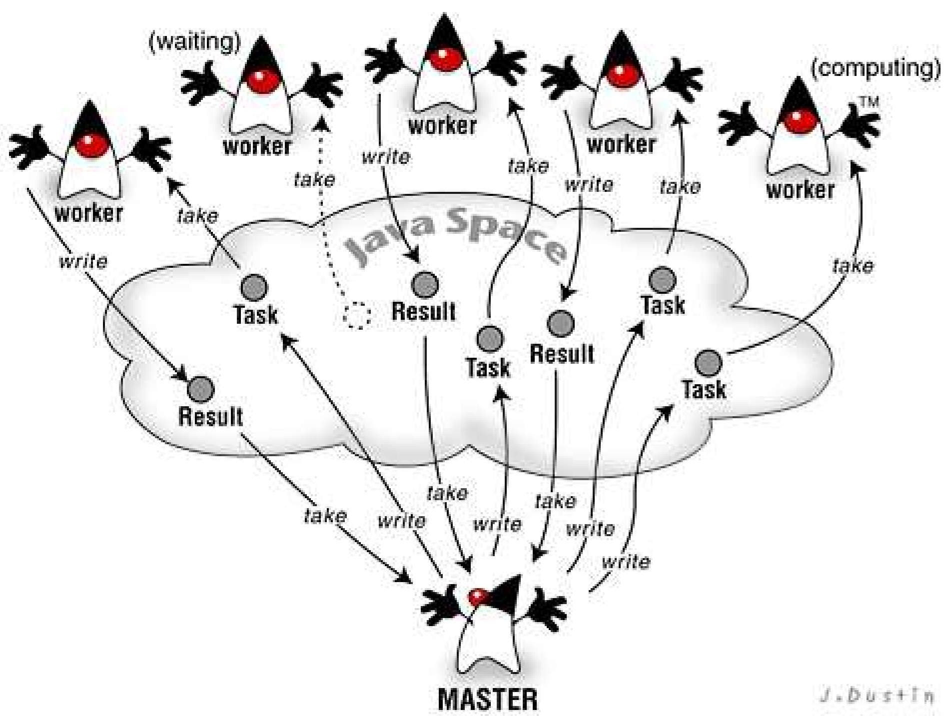

Figure 2: JavaSpaces Compute Server example
Picture from: http://www.sunworld.com/sunworldonline/swol-01-2000/swol-01-jiniology.html

(This example was taken from the Jiniology series in javaworld magazine)

Basically, a compute server is an implementation of a powerful, all-purpose computing engine. The computing engine we will look at uses JavaSpaces. It will function a par the diagram. A master will generate tasks and drop them into the space. A task is simply an entry that describes the task and contains methods needed to perform a computation. A worker will pick up one of the tasks and process it. Then the result will be written back into the space and, at some point, retrieved. The worker then fetches another task and begins the process again. This system has a number of advantages due to the compute server's space-based characteristics. They scale well; the more workers attached to a space the faster tasks are computed. Workers can be added or removed at runtime, and the compute server will continue to operate as long as there is at least one available worker to compute tasks. Compute servers naturally balance loads well a slower CPU worker will compute its tasks more slowly then a fast CPU worker thus faster machines do proportionally more work.

Okay lets look at how each part of the system is coded and functions.

For masters and workers to exchange task objects, they must agree on a common interface for those objects. As this is a simple implementation all that is needed is a method that a worker can call from the task to start computing.

The Command interface shown below does just this.

```
package javaworld.simplecompute;

import net.jini.core.entry.Entry;

public interface Command extends Entry {
    public Entry execute();
}
```

Note that Command extends Entry. This allows the object to be passed into a JavaSpace. The Execute function also returns an Entry allowing for the result to be written back into the JavaSpace.

Workers

As stated before they take tasks as they become available, compute them, and then write their results back into the space. Results are entries that contain data from the computation's output.

```
                 package javaworld.simplecompute;

                 import jsbook.util.*;
                 import net.jini.core.entry.Entry;
                 import net.jini.core.lease.Lease;
                 import net.jini.space.JavaSpace;

public class Worker {

                         JavaSpace space;

                               public static void main(String[] args) {
                                 worker.startWork();
                               }
```

```
    public Worker() {
        space = SpaceAccessor.getSpace();
    public void startWork() {
        TaskEntry template = new TaskEntry();
        for (;;) {
            try {
                 TaskEntry task = (TaskEntry)
                 space.take(template, null, Long.MAX_VALUE);
                 Entry result = task.execute();
                      if (result != null) {
                      space.write(result, null, 1000*60*10);
                }
            }
             catch (Exception e) {
                System.out.println("Task cancelled");
            }
        }
    }
}
```

In this code we can see that the worker accesses the space by calling the SpaceAccessor function getSpace.(This function is available in the jsbook.util package that comes with the JavaSpaces: Principles, Patterns and Practice book. The code from this book is downloadable from sun's website.) Once it does this it makes a template for TaskEntries and does a continuous loop looking for TaskEntries. Once found the Task is removed from the space using a take operation at which point the tasks execute() function is called which processes the task. When the processing is complete a ResultEntry is written back into the space.

Base task class

This is the base class for all tasks.

```
package javaworld.simplecompute;

import net.jini.core.entry.Entry;

public class TaskEntry implements Command {
    public Entry execute() {
        throw new RuntimeException("TaskEntry.execute()
                                             is not implemented.");
    }
}
```

The TaskEntry implements the Command interface. Since TaskEntry can't be an abstract class (if it were, the worker wouldn't be able to instantiate it and use it as a template), we need to provide an implementation of the execute method. Rather than providing a concrete implementation that does nothing, we instead throw a runtime exception if this method is ever invoked. This encourages subclasses to implement this method, even though they have no compile-time requirement to do so.

The Master

This is a simple master that generates two kinds of tasks: addition and multiplication tasks. In each case, the master initializes the tasks with two integers and then writes the task into a space. For addition tasks, the worker adds the two integers together and returns the result; likewise, for multiplication tasks, the two numbers are multiplied and then the results are written to the space.

Let's start by defining the tasks. Here is the AddTask:

```
package javaworld.simplecompute;

import net.jini.core.entry.Entry;

public class AddTask extends TaskEntry {
    public Integer a;
    public Integer b;

    public AddTask() {}
    public AddTask(Integer a, Integer b) {
        this.a = a;
        this.b = b;
    }

 public Entry execute() {
        Integer answer = new Integer(a.intValue() + b.intValu
        AddResult result = new AddResult(a, b, answer);
        return result;
     }
}
```

The AddTask object takes two integers and adds them together and returns the result as an AddResult object.

ResultEntry

This is the base class for all results to be returned within a system.

```
Here is the definition of ResultEntry:

 package javaworld.simplecompute;

 import net.jini.core.entry.Entry;

public class ResultEntry implements Entry {
    public Integer a;
    public Integer b;
    public Integer answer;

    public ResultEntry() {}
}
```

The AddResult

AddResult entry is subclassed from ResultEntry, and looks like this:

```
package javaworld.simplecompute;

public class AddResult extends ResultEntry {
    public AddResult() {}
    public AddResult(Integer a, Integer b, Integer answer) {
        this.a = a;
        this.b = b;
        this.answer = answer;
    }
    public String toString() {
        return a + " plus " + b + " = " + answer;
    }
```

```
}
```

The MultTask class with be essentially the same as the AddTask class except it will multiply to integers together instead of adding them together.

Now we can define the Master class:

```
package javaworld.simplecompute;

import jsbook.util.*;
import net.jini.core.entry.*;
import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

public class Master {
    private JavaSpace space;

    public static void main(String[] args) {
        Master master = new Master();
        master.startComputing();
    }

    private void startComputing() {
        space = SpaceAccessor.getSpace();
        generateTasks();
        collectResults();
    }
    private void generateTasks() {
        for (int i = 0; i < 10; i++) {
             writeTask(new AddTask(new Integer(i), new Integer
        }
        for (int i = 0; i < 10; i++) {
            writeTask(new MultTask(new Integer(i), new Integer
        }
    }
    private void collectResults() {
        for (int i = 0; i < 20; i++) {
            ResultEntry result = takeResult();
            if (result != null) {
                System.out.println(result);
            }
        }
    }
    private void writeTask(Command task) {
        try {
            space.write(task, null, Lease.FOREVER);

        }catch (Exception e) {
            e.printStackTrace();
        }
    }
    protected ResultEntry takeResult() {
        ResultEntry template = new ResultEntry();

        try {
            ResultEntry result = (ResultEntry)
                space.take(template, null, Long.MAX_VALUE);
            return result;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
```

```
}
```

The Master class first gets access to the space by calling the SpaceAccessor.getSpace() function. The Master then calls the two functions one to generate the Add and Multiply tasks and another to collect the results.

So in roughly 30 lines of code we have written a Compute Server. It doesn't do anything spectacular but does demonstrate some of the power and easy of use of the JavaSpaces technology.

Note I have not actually run this code. If you are trying to get this code to run I would suggest looking at the original article in sunworld online magazine here

Advantages

JavaSpaces has a number of advantages when it comes building distributed solutions. JavaSpaces is simple. The space based programming concept it is designed around requires learning only a few operations, read, write and notify. The space concept allows for expressive power as a large base of distributed applications can be tackled using a small amount of code. Because of it's simplicity and expressiveness using JavaSpaces cuts down on the amount of effort needed to design and code distributed systems. It does this through eliminating the complexity of different hardware/software platforms, network protocols, and presenting an interface that can be grasped quickly.

Disadvantages

JavaSpace technology is still a relatively new framework that up to this point has had no real world testing. It does rely greatly on the proven work of Linda but there will no doubt arise issues when it is put to use into actual problem solving roles. One issue that is already being asked is about security. At the time of this writing security issues remain unresolved. One such questions being asked is:

Can one prevent hosts from seeing certain objects in a space?

So far the security issues of JavaSpaces have not been well defined. These issues will only become more important as companies consider using JavaSpaces in their business applications.

Summary

In Conclusion, we see that JavaSpaces provides Simple Interfaces for programming. Only four basic operations can be performed on a space. JavaSpaces is flexible as was shown by the compute server example. One could easy envision how the same kind of system could be used for a variety of processing tasks. JavaSpaces is scalable, this was also demonstrated in the compute server example as Workers could be added or subtracted from the space to increase or decrease computational power. An finally the real power of the JavaSpaces approach was demonstrated by it's loosely coupled nature that allows the client and server to operate while knowing virtually nothing about each other.

Resources

Sun's website dedicated to JavaSpaces information http://java.sun.com/products/javaspaces/

The complete source code from the compute server example can be found here: http://www.javaworld.com/jw-01-2000/jini/code.zip

The source code for the jsbook.util package can be found here:
http://java.sun.com/docs/books/jini/javaspaces/JSBookExamples.zip

References

Freeman, Eric., Hupfer, Susanne., and Arnold, Ken. (December, 1999), [Online]. *Chapter 1 - Introduction, JavaSpaces TM Principles, Patterns, and Practice*.http://developer.java.sun.com/developer/Books/JavaSpaces/introduction.html

Freeman, Eric., & Hupfer, Susanne.(November 11, 1999), [Online] *Make room for JavaSpaces, part 1.* Sunworld Online Magazine.
http://www.sunworld.com/sunworldonline/swol-11-1999/swol-11%20jiniology.html(2000, January 25).

Freeman, Eric.(January, 2000), [Online].*Make room for JavaSpaces, Part 2.* Sunworld Online Magazine.http://www.sunworld.com/sunworldonline/swol-01-2000/swol-01-jiniology.html(2000,January 25).

Laird, Cameron.(September 17, 1999), [Online].*JavaSpaces: Making Distributed Computing Easier*. Byte Online Magazine. http://www.byte.com/feature/BYT19990915S0001(2000, January 25).

JavaSpaces Specification. (1999,January),[Online].

Possible Exam Questions

1. What are the fundamental properties of a Space?
   Briefly describe each property.
2. What are the advantages of JavaSpaces over other distributed solutions?
3. What are the four main operations that can be performed on a Space?
   Briefly describe each operation.