JavaSpaces technology for distributed communication and collaboration

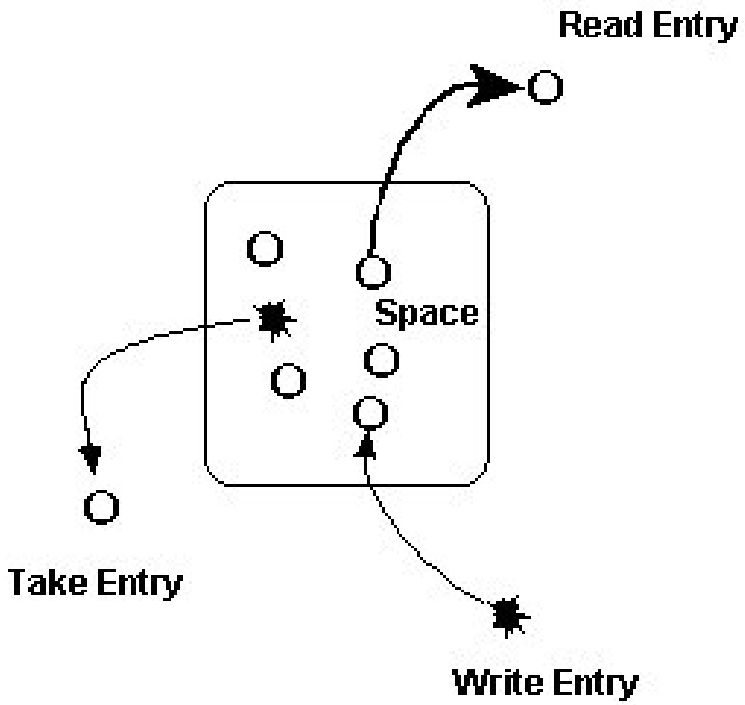
Chih-Yao Hsieh
Computer Science and Engineering
University of Texas at Arlington
chsieh@cse.uta.edu

Abstract

This paper will give an introduction of the JavaSpaces ™ technology and the application of the JavaSpaces™ for distributed communication and collaboration. I will also discuss the way JavaSpaces™ express the data stored in the "Spaces", and the difference between the JavaSpaces™ and other distributed technologies. After the comparison, I will show the pros and cons of the JavaSpaces™ technology.

1 Introduction

1.1 Overview of JavaSpaces [4][14]



What are JavaSpaces ™? JavaSpaces ™ can be considered as a library providing a set of methods to help the developers to build a distributed system that uses the flow of objects model. The system is not based on the traditional approach using the method-invocation-style protocol. Otherwise, it uses the flow of objects approach to accomplish the distributed computing – this means the architecture of the distributed system is based on the movement of objects.

The objects can move in and out of the "spaces". This kind of architecture can simplify the remote interfaces of the distributed system. If the system uses the method-invocation-style design, then the developers will have to design a much more complex remote interface then the remote interface used in the JavaSpaces ™.

1.2 Motivation

If we develop the distributed computing environment in traditional way with RPC, CORBA, or RMI, we have to face the complexity of data consistency issues, message transmission, and fault tolerance issues. The JavaSpaces ™ technology handles most of those issues and hides the complexity from developers. It can simplify the developing process and improve the productivity of the distributed computing programs. So we will take a look at the JavaSpaces ™ technology and consider the pros and cons of it.

1.3 The Traditional Distributed Computing Environment

The traditional distributed computing environments use method-invocation-style designs such as RPC, CORBA, RMI, and others. They provide the methods that make the developers can easily invoke the remote methods without caring the low level stuff such as "create the socket", "binding the socket", and "connect", etc. But they do not make other things such as data consistency, synchronization, and message communication, etc. transparent to the developers. The developers have to handle those things themselves.

1.4 The Advantages of JavaSpaces

JavaSpaces ™ is a technology that makes Java objects easy to do dynamic sharing, communication, and coordination with each other. JavaSpaces ™ hides the low level design of message communication, objects passing, and consistency maintenance from the users. JavaSpaces ™ is developed based on the Jini technology while Jini is based on the RMI of Java. The characteristic of Jini is that Jini can let any process residing in any JVM (Java Virtual Machine) transfer binary code to each other.

JavaSpaces ™ is like a marketplace with some clients, sellers, and brokers, etc. For example, we can use the JavaSpaces ™ to build several PC marketplaces by spaces - just analogous to the physical PC stores in the real world. According to our own design, the sellers can put the objects containing prices, and order forms, etc., and then request the spaces to notify the sellers when the objects they put are being read. Each client can search the prices for the products they specified in those marketplaces.
The design goals of JavaSpaces ™ technology are [2]

- **Simple**

  To achieve this goal, JavaSpaces ™ only have seven major methods that are usually used. Those methods are “write”, “read”, “take”, “notify”, “snapshot”, “readBlocking”, and “takeBlocking”. The JavaSpaces ™ hides many complex issues from developer like message transmission, service lookup mechanism, etc.

- **Write less code**

  Most of the server codes are written by JavaSpaces ™ so the developers can pay less attention to the low level stuff. The object transmission mechanism, serialization process, notification mechanism, and many other procedures are all packed as libraries to be used.

- **Unified mechanism broadens applications**

  The unified structure of the programs can easily replicate the programs and execute them.

- **Legacy interoperability**

  This goal is achieved because we can reuse the legacy codes by wrapping them in objects created by Java APIs.

- **Use Java language to full capabilities**

  This means to exploit the characteristic of Java. Because Java is a strongly typed programming language, JavaSpaces ™ take the advantage of this to find, match, and reference Java objects in the spaces of JavaSpaces ™ service.

- **Real objects**

  The objects stored in JavaSpaces ™ are real objects. This means the objects can be data or classes so the objects can store data, methods, and other objects and so on.

- **Asynchronous**

  The providers and requestors of network resources exchange their information asynchronously.

- **Extensible**

  A variety of implementations should be possible, including relational database and object-oriented database storage.

- **Transparent**

  The same entries and templates must work in the same ways regardless of the implementation. The entries and templates will be considered the same if all the public fields are exactly identical.

- **100% Pure Java**

  This design goal is to maintain the consistency of this library.

2 The Architecture of JavaSpaces

2.1 The JavaSpaces Application Model

As I mentioned above, JavaSpaces ™ uses the flow of objects model to let all the participants of the distributed system to interact with each other by exchanging the objects stored in the "spaces" by participants. The "object" referred here is expressed as a type of Java – Entry. This is an interface stored in *net.jini.core.entry.Entry.* When we want to use the use the Entry, we just need to implement the interface to a class. Because of the Entry is a class of Java the Entry not only can store the data but also the methods.

We can divide the participants into two categories – requester and provider. The provider can put the Entries into the spaces, stores the data and methods in the Entries, and

registers to the spaces so that whenever the Entry matches a specified template is written by any other participant the spaces will notify the provider that somebody modifies the Entry the provider provides. The requester can concurrently “read” or “take” several Entries stored in the spaces. The requester look up entries using templates, which are entry objects that have some or all of its fields set to specified values that must be matched exactly [1].

Entries written into the JavaSpaces ™ services will have a leasing time. We can specify the “WRITE” operation a certain duration such as 5 minutes or so. Besides, the holder of that Entry can renew or cancel the lease before the lease expires. With the leasing time, the JavaSpaces ™ service can know the object being written will only be kept in space for 5 minutes. The leasing time is used to prevent the side effect of partial failure because after the partial failure occurs, the objects held by the failed participants will not be freed. In the worst case, the objects may grow without control. With the leasing time, the JavaSpaces ™ services can remove the objects residing in the space when the leasing time expires.

The leasing time can be certain amount of time, Lease.FOREVER, or Lease.ANY. Lease.FOREVER means the leasing time of this object is indefinitely. The object can stay in the space forever. Lease.ANY means the leasing time of the object is not declared, the actual leasing time will be decided by the JavaSpaces ™ server.

The operations of the participants such as “read”, “write”, “take”, etc. all belong to certain transactions. JavaSpaces ™ uses the two-phase commit protocol. If the transaction is not committed by transaction manager, then all the operations included in this transaction will not be executed. This means the transactions in JavaSpaces ™ are all executed atomically.

Transactions in JavaSpaces ™ have the following properties: [5]

- **Atomicity**

Atomicity means the space of JavaSpaces ™ service will not change it state until the transaction has been committed. The operations included in the transaction will only be totally applied or all discarded.

- **Consistency**

After the completion of a transaction, the JavaSpaces ™ service should still stay in a consistent state. It means the service should stay in the correct status after a transac-

tion. For example, if we stipulate that each car should have a label shows its manufacturer, then after the transaction, this rule should not be broken. The enforcement of consistency if not the duty of transaction itself, but the creator of it – a transaction is a tool to allow consistency guarantees, and not itself a guarantor of consistency [1].

- **Isolation**

  Though the transactions may be executed concurrently, each transaction considers other transactions as they will complete either before or after it. In the user's view, the user will only notice that each transaction is executed one after the other sequentially. This means the transactions will not affect each other so when we design a transaction, we can ignore what other transactions are doing.

- **Durability**

  This means as long as the transaction is committed; the changes to the space will remain there while the space crashes. But this characteristic is not guaranteed by Jini Transaction Model, the durability will only be guaranteed when the space is implemented to support persistence space.

2.2 The Operations of JavaSpaces

In the following section, we will discuss the operations of JavaSpaces ™ in detail. The bold font words are the keywords in JavaSpaces ™.

2.2.1 Write [5][7]

The **write** operation will duplicate a copy of entry and place the copy into the space of JavaSpaces ™ service. The original entry object will not be affected by the **write** operation. This means the entry can be passed to several **write** operations without any change. The **write** method takes three parameters – an **Entry** that will be written into the space of the JavaSpaces ™ service, a **Transaction** that specifies which transaction this **write** belongs to, and a **long** value to notify the JavaSpaces ™ service the leasing time of the written object.

If the **write** operation belongs to a certain transaction, the other operations outside the transaction will not be able to see the entry used by this **write** operation until this transaction being committed.

If we specify the leasing time to be Lease.FOREVER, according to Sun's implementation, the actual leasing time will be limited to five minutes. When we want the JavaSpaces ™ service to keep the object forever, the only way is to renew the lease periodically.

2.2.2 Read [1][5][7]

The **read** operation request the JavaSpaces ™ service to search through the space for an entry that matches the template passed to the **read** operation. This action will not remove the entry from the JavaSpaces ™ service. The **read** method takes three parameters – an **Entry** that specifies the template, a **Transaction** that specifies which transaction this operation belongs to, and a **long** value to specify the time the **read** operation should be blocked to wait for the requested object. When the waiting time exceeds, the **read** operation will simply return a **null** value indicating the requested object is not in the JavaSpaces ™ service.

The matching procedure is to match the public fields of the provided **Entry** template against the **Entry** stored in the space of JavaSpaces ™ service. We can use the **null** in the public field of template to specify this field to be a doesn't-care field during matching procedure. The function of **null** here is like a wildcard character. When the JavaSpaces ™ service tries to find the matching entry, the **null** field will all be considered as successful matching.

If the second argument of the **read** is **null**, this means this **read** operation is a standalone operation – it itself is a single operation transaction. There might be several entries match the template, according to the JavaSpaces ™ Specification, the JavaSpaces ™ service will not guarantee each time the **read** operation will get the same entry from the space. We will get one of the matching entries from the spaces either from written entry in the same transaction or the entry outside the transaction. This means we can not design a system relies on assuming the JavaSpaces ™ service will return the same entry whenever we execute the **read** request.

There is a similar operation – **readIfExists**. The main different is the **readIfExists** operation will return immediately no matter the request entry is in the space or not. But there is an exception; if the matching entry is in the same transaction of itself, the **readIfExists** operation will wait if the programmer specifies a waiting period until the waiting period expires.

2.2.3 Take [1][5]

The **take** operation request the JavaSpaces ™ service to search an entry stored in the space that matches the public field specified by the template passed to the **take** operation. If the JavaSpaces ™ service finds the matching entry, it will remove the matching entry from the space of the JavaSpaces ™ service and return the entry.

If there are many matching entries in the space of the JavaSpaces ™ service, the service will choose one from them without any regulation. So if our program takes the entries successively from the space, we can not expect what is the order of entries being returned.

The relationship between **take** and **takeIfExists** are similar to **read** and **readIfExists**. So the **takeIfExists** will return **null** immediately if there is no entry matches the provided template and the only exception is if the **takeIfExists** belongs to a transaction, the **takeIfExists** will wait if the developer specified a waiting time.

2.2.4 Notify [5][7]

The **notify** operation requests the JavaSpaces ™ service to notify the listener when any participant writes an entry that matches the provided template. The **notify** takes five parameters – an **Entry** that specifies the template, a **Transaction** object that indicates that the **notify** operation will not occur within a transaction, a **RemoteEventListener** that is an object that receives the remote events from the space of the JavaSpaces ™ service, a **long** value for the leasing time to specify the period of time the JavaSpaces ™ service should maintain the notification, and a **MarshalledObject** that is an object that the JavaSpaces ™ service provides to the remote listener as part of the notification [7].

2.2.5 Snapshot [5][7]

Originally, every time we pass the template to the methods of JavaSpaces ™ service such as **read**, **write**, **take**, etc., the template must be serialized before it being transferred to the space of JavaSpaces ™ service. If the same template is used so many times, the overhead of serialization can not be disregarded. To avoid this kind of overhead, we can use **snapshot** to “snapshot” an entry we provide and return a specialized entry – a snapshot of entry. This specialized entry can be passed to the **read**, **write**, and **take**, etc. and avoid the overhead of serialization. The restriction of

the usage of **snapshot** is the specialized entry can only be used within the JavaSpaces ™ service that generates it.

3 The Applications of JavaSpaces

3.1 Collaborative Application

I will show an example of the collaborative application here. The application we'll discuss here is the car marketplace. To exploit the characteristics of the JavaSpaces ™ technology, we have to design the system that can gain from the flow of objects model.

There will be a discovery system, a browser system, an order system, and a transaction system. The discovery system will use the Jini "multicast to discovery services" method to add all the reachable market place into the program. The customers can browse the There will also have two classes – client and seller. There will also be a dealer class implement the **Entry** interface. The dealer entry will have the car related info such as car styles, car photos, manufacture, specification, price, etc., and the dealer entry will also contain an order field in order that the customer can **take** the entry out of the space and modify the order field to indicate the willing of order. Then the customer **write** back the entry into the space and the space will notify the dealer that someone has modified the entry he put. The dealer then can **write** a form entry so that the customer can **take** the form entry and fill the form. After all, the transaction is completed.

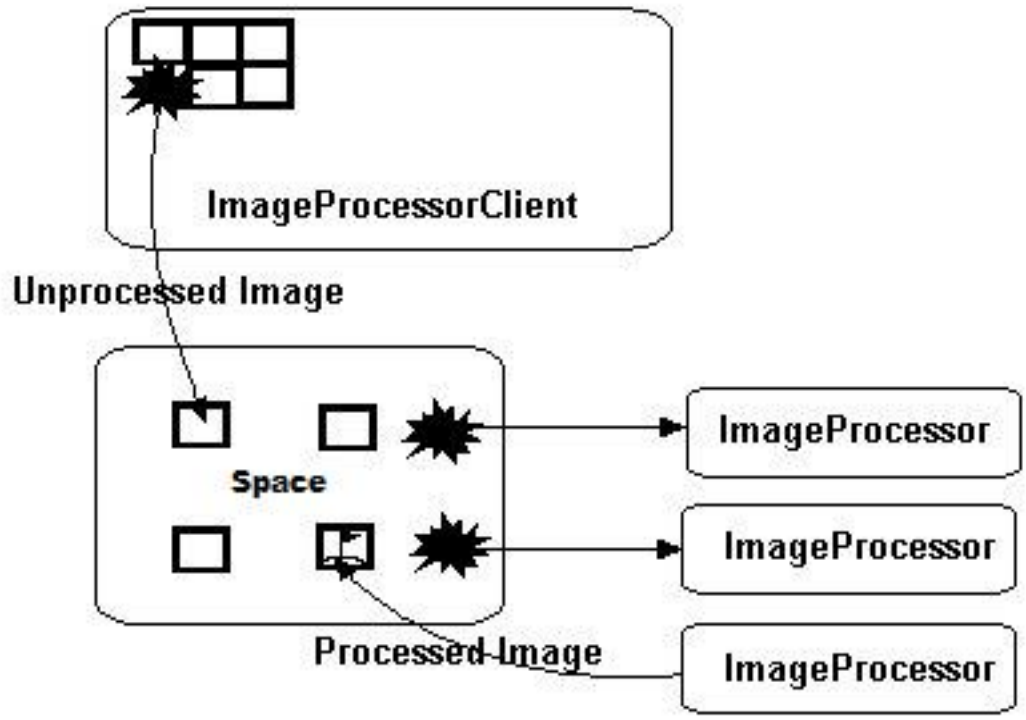
Of course, we can also add some price negotiating system. The way to implement this by JavaSpaces ™ technology is to implement a negotiating entry so that the customer and dealer can discuss the price back and forth by this entry.

Before starting the JavaSpaces ™ service, we have to launch several underlying services – Jini lookup service, Jini transaction service, Web server, and **rmid**. Whenever the JavaSpaces ™ service starts, the service will register itself to the local Jini lookup services. Jini lookup service has to be launched because the JavaSpaces ™ technology depends on it to let the client to discover the JavaSpaces ™ services. The Jini lookup service can also support the information that where the **TransactionManager** is. The **TransactionManager** is needed when transaction is executed. The Web server is used for the binary code transference. The usage of **rmid** will be discussed in the following paragraph.

To start the JavaSpaces ™ service, we have to start **outrigger** – the Sun's implementation of the JavaSpaces ™ service. The service has two modes. One is transient JavaS-

paces ™ Service, another is persistent JavaSpaces ™ Service. The transient service will not need a RMI activation daemon (**rmid**). This means the server will lose all the state information and the **rmid** is unable to restart the JavaSpaces ™ service. Otherwise the persistent JavaSpaces ™ service will store all the state information in a log file when the service terminates. The **rmid** can restart the service later.

3.2 Parallel Application [7]



The application to be discussed in the following section is a parallel image processing application that can processes the images such as sharpen, blur, soften, etc. in a parallel way.

The main components of this program are ImageProcessorClient, ImageProcessor, and ImageEntry. ImageProcessorClient is responsible to get the image and separate the image into small pieces, wrap the image into the ImageEntry, and **write** the unprocessed ImageEntry into the space of the JavaSpaces ™ service. ImageProcessor is responsible to get the unprocessed ImageEntry out of the space, process the Im-

ageEntry, and then write back to the space. The ImageProcessorClient then take the processed ImageEntry back and combine the pieces into a complete processed image.

3.3 Discussion

3.3.1 Pros and Cons

The advantages of the JavaSpaces technology are:

- **Accelerate the develop time of the distributed applications significantly.**
  The JavaSpaces ™ technology has taken care of the transaction, data consistency, and fault tolerance problems. The developer can concentrate on application level design.

- **Flexibility**
  Participants can discover the JavaSpaces ™ dynamically by multicast service lookup or unicast service lookup. Participants can leave the JavaSpaces ™ service at anytime and with some kind of design, the participants can resume the jobs not finished when they come back to service.

- **Consistency**
  The consistency issues are maintained by the JavaSpaces ™ itself.

- **Concurrency transparency**
  JavaSpaces ™ handles the participating processes concurrently. The developers can get rid of the complexity of controlling the concurrent distributed computing.

- **Robustness**
  With the persistent server option, the JavaSpaces ™ service can store the current status to allow the server to resume without losing the data after crash.

- **Gain benefits from the Java, Jini, RMI technologies.**
  JavaSpaces ™ is based on Java, Jini, and RMI. So the JavaSpaces ™ can take the advantages from the excellences of those technologies such as cross-platform, security mechanism, easy to design, etc.

- **Can easily cooperate with the C++, C programs by using JNI (Java Native Interface), and CORBA technology to accelerate the performance.**

The disadvantages of the JavaSpaces technology are:

- **The scalability is restricted**

Because all the transmission is centralized in one machine, the bottleneck of the performance is obviously there. This problem can be overcome by doing a load balancing system by developers themselves.

- **Overhead of transmitting of the entries**
  JavaSpaces ™ use the flow of objects model to substitute the traditional method invocation model and the message passing model. In some cases, it is not worthy to abandon the traditional way. For example, the messenger system, if we convey the message by entries, it will be more expensive then using the traditional way.

- **Overhead of the matching process of the entries**
  Like the car market example cited above, if there is lots customers there, the JavaSpaces ™ service will be burdened by doing lots of comparison jobs.

- **Efficiency is an issue**
  The JavaSpaces ™ technology will be less efficient then traditional message passing method and method invocation model while there are lots of objects and object matching process is heavily used or the object transmission is very frequently but the objects are only packed with small amount of data. Those conditions should be considered by the developer to avoid.

3.4 Conclusions

Although the JavaSpaces ™ technology has many advantages, it will not be appropriate to use JavaSpaces ™ in any circumstances. We should use it wisely to keep away from the shortcomings of the JavaSpaces ™.

References:

[1] Sun Microsystems Inc., JavaSpaces ™ Service Specification.
http://www.sun.com/jini/specs/js1_1.pdf

[2] Sun Microsystems Inc., White Paper: JavaSpaces ™
http://java.sun.com/products/javaspaces/whitepapers/jspaper.pdf

[3] Sun Microsystems Inc., Data Sheet: JavaSpaces ™
http://java.sun.com/products/javaspaces/datasheets/jssheet.pdf

[4] JavaSpaces ™ FAQ
http://java.sun.com/products/javaspaces/faqs/jsfaq.html

[5] Freeman, E., S. Hupfer, and K. Arnold. JavaSpaces Principles, Patterns, and Practice. Reading, MA: Addison Wesley Publishing, 1999.

[6] Make room for JavaSpaces, Part 1 - Ease the development of distributed apps with JavaSpaces
http://www.javaworld.com/javaworld/jw-11-1999/jw-11-jiniology_p.html

[7] H.M. Deitel, P.J. Deitel, and S.E. Santry. Advanced Java2 platform: how to program. Prentice-Hall, 2002. pp. 1259-1316

[8] JavaSpaces: Making Distributed Computing Easier
http://www.byte.com/documents/s=146/byt19990915s0001/index.htm

[9] Sun's JavaSpaces is foundation for future distributed systems
http://www.javaworld.com/javaworld/jw-09-1997/jw-09-idgns.javaspaces.html

[10] JavaSpaces Technology – Object Oriented Approach in Distributed and Parallel Systems
http://www.cs.utexas.edu/users/cai/projects/JavaSpaces/JavaSpaces.html

[11] JavaSpaces Promises Distributed Computing Breakthrough
http://www.byte.com/documents/s=146/byt19990921s0001/

[12] Make room for JavaSpaces, Part 5
Make your compute server robust and scalable with Jini and JavaSpaces
http://www.javaworld.com/javaworld/jw-06-2000/jw-0623-jiniology.html

[13] Concurrency of Components Using JavaSpaces
http://eewww.eng.ohio-state.edu/~khan/khan/Presentations/Taiwan-9-00/ConcurrencyCompUsingJavaSpaces/ConcurrencyCompUsingJavaSpaces.htm

[14] JavaSpaces Introduction
http://eewww.eng.ohio-state.edu/~khan/khan/Presentations/Taiwan-9-00/PDFfiles/JavaSpaces_BW.PDF

[15] Jini Architecture Specification
http://www.sun.com/jini/specs/jini1_2.pdf