Linguagens para desenvolvimento de aplicações paralelas e distribuídas

JavaSpaces

Luthiano Rodrigues Venecian

Tópicos

- Introdução
  - Computação distribuída
- Tecnologia JavaSpaces
  - Origem, funcionalidades, vantagens, operações
  - Banco de dados relacionais X JavaSpaces
  - Conceitos básicos
  - Programa Exemplo - Hello Workd
  - Transações
  - Programa Exemplo – uso de Transações
- Considerações Finais
- Referências Bibliográficas

Introdução

As redes de computadores tornaram-se vitais para o mundo da computação, mudando a maneira que as pessoas Usam o computador e, é claro, a maneira como os **projetistas desenvolvem suas aplicações**. É nesse contexto que surge a computação distribuída.

Computação Distribuída

É um tipo de computação onde:

- As aplicações são compostas por processos que estão distribuídos através da rede;
- Pode ser vista como uma coleção de componentes e objetos que podem ser alocados em diferentes computadores da rede.

As vantagens na construção das aplicações:

- É possível conseguir maior desempenho adicionando novos computadores no sistema onde a aplicação está rodando – A aplicação é facilmente escalável;
- Desde que os processos rodando em computadores diferentes estão se comunicando um com os outros, estes podem compartilhar seus recursos;
- Tolerância a falhas – Se um computador para ou fica off-line, o resto do sistema continua executando.

Computação Distribuída

Aplicações distribuídas são mais difíceis de projetar e implementar do que aplicações Standalone.

Entretando, essa não é a única dificuldade:

- Os processos de uma aplicação distribuída, precisam se comunicar para trabalharem em conjunto
  - A comunicação se dá sobre uma rede: Tempos de transmissão são geralmente altos quando comparados a velocidade do processador dos computadores envolvidos. O tempo de uma transmissão de uma origem ao seu destino é a LATÊNCIA;
- Sincronização entre os processos;
- Tolerância a falhas – necessita manter um estado global de consitências.

No intuito de evitar esses tipos de problemas, os desenvolvedores podem fazer o uso da Tecnologia JAVASPACES

A Tecnologia JavaSpaces

- É uma especificação desenvolvida pela Sun Microsystems para o **paradigma de programação baseada em espaço;**
- É baseada na linguagem Java e totalmente orientada a objetos;
- A implementação de referência da Sun foi desenvolvida como um serviço dentro da arquitetura de computação distribuída *Jini*

Jini é uma arquitetura de rede aberta que habilita desenvolvedores criar serviços centrados na rede – implementados em Hardware e Software.

O que é espaço?

- Um espaço é um repositório de objetos compartilhado e acessível via rede.

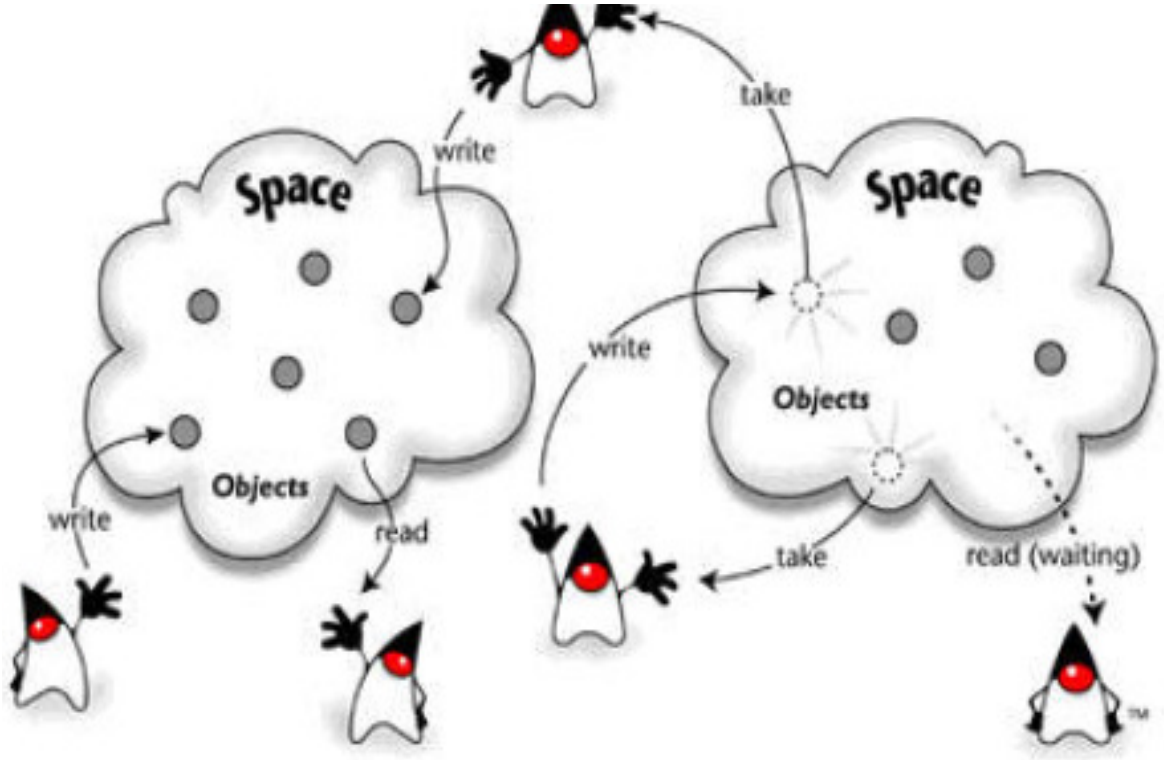
- Processos utilizam esse repositório como um mecanismo persistente para armazenar e trocar objetos. Ao invés de comunicar diretamente, eles comunicam trocando objetos pelo espaço.

Modelo de Programação

Espaço é um repositório de objetos compartilhado e acessível via rede.

- **Processos utilizam esse repositório como um mecanismo persistente para armazenar e trocar objetos.**

- **Ao invés de comunicar diretamente, eles comunicam trocando objetos pelo espaço.**



Modelo de Programação

- Na construção de aplicações distribuídas, geralmente envolve troca de mensagens entre processos ou invocação de métodos em objetos remotos. **Em JavaSpaces, os processos não se comunicam diretamente entre sí.**
- Os processos fazem operações simples, como escrever (write) novos objetos no espaço, retirar (take) objetos do espaço ou ler (read – fazer uma cópia) objetos do espaço.

Modelo de Programação

- Para ler ou retirar objetos do espaço, os processos procuram o objeto desejado a partir de um padrão (template). Se o objeto não é encontrado imediatamente, o processo pode esperar até que o objeto desejado seja colocado no espaço.

- Processos não modificam os objetos que estão no espaço e nem podem chamar seus métodos diretamente.
  - Para modificar um objeto: o processo deve removê-lo do espaço, atualizar seus atributos e inseri-lo no espaço novamente como um novo objeto.

Origem

- JavaSpaces teve origem no início de 1998, é um novo modelo para computação distribuída inspirada nas pesquisas desenvolvidas pelo professor David Gelenter de Ciência da Computação da Universidade de Yale - EUA.

- Gelender desenvolveu uma ferramenta denominada Linda para a construção de aplicações distribuídas. Linda consistia de um pequeno número de operações combinadas com um armazenamento persistente denominado tuple space.

  O resultado foi surpreendente. O uso de um meio de armazenamento de objetos juntamente com um pequeno número de operações simples, permitiu implementar facilmente uma grande classe de problemas paralelos e distribuídos.

Funcionalidades

- **Espaços são compartilhados** - Espaços são "memórias compartilhadas" acessíveis através da rede, onde vários processos remotos podem interagir concorrentemente;

- **Espaços são persistentes** - Espaços provêm armazenamento confiável para os objetos. Uma vez que o objeto é colocado no espaço, ele permanecerá lá até que um processo o remova explicitamente;

Funcionalides

- **Espaços são associativos** - Os objetos de um espaço são localizados através de busca associativa, ao invés de um endereço de memória ou de um identificador.
  - A busca associativa provê um modo simples de localizar objetos de acordo com o seu conteúdo, sem precisar saber como o objeto é chamado, quem o possui, quem o criou, ou onde ele está armazenado;

Funcionalidades

- **Transações em espaços são seguras** - A tecnologia JavaSpaces provê um modelo de transação, que assegura que uma operação no espaço é atômica;

- **Espaços permitem a troca de conteúdo** - Enquanto o objeto está no espaço, ele é apenas um dado passivo. Entretanto, quando o objeto é copiado ou retirado do espaço, uma cópia local do objeto é criada, possibilitando então o acesso aos seus campos privados e a invocação de seus métodos.

Vantagens

A tecnologia JavaSpaces apresenta as seguintes vantagens:

- **É simples** - A tecnologia não requer conhecimento de uma interface de programação complexa, consistindo de um conjunto de operações simples;
- **É expressiva** - Usando um pequeno conjunto de operações, pode-se construir aplicações distribuídas sem escrever muito código;

Vantagens

- **Suporta protocolos de baixo acoplamento** - Uma vez que remententes e receptores são desacoplados, espaços suportam protocolos simples, flexíveis e confiáveis. O desacoplamento facilita a composição da grandes aplicações, suporta análise global e aumentam o reuso de software;
- **Facilita a tarefa de escrever sistemas cliente/servidor** - Quando se projeta o servidor, características como acesso concorrente de múltiplos clientes, persistência e transação são reinventados a cada vez.

Operações

- **write** – Escreve um objeto no espaço;
- **read** - Pega uma cópia de um objeto do espaço;
- **take** - Remove um objeto do espaço.

- **write:** escreve uma cópia de uma entrada no espaço. Se forem feitas múltiplas chamadas da mesma entrada, então são escritas múltiplas cópias da entrada no espaço;
- **read:** envia uma entrada que é usada como um template , retornando uma cópia de um objeto no espaço compatível com esse template. Se não houver nenhum objeto compatível no espaço, então read pode esperar um certo tempo definido pelo usuário até uma entrada compatível chegar ou até esse tempo se esgotar;

Operações

- **take:** funciona como read, exceto que o objeto compatível com o template é removido do espaço.

Existem ainda as operações **readIfExists** e **takeIfExists**, as quais diferem das operações read e take pelo fato de não esperarem um objeto chegar no espaço caso não encontre um objeto compatível com o template.

Operações

JavaSpaces possui outras duas operações utilizadas em situações específicas:

- **notify:** um template e um objeto são usados de modo que o espaço notifica a esse objeto sempre que entradas compatíveis com o template forem escritas no espaço. É útil em casos onde é necessário interagir com o espaço como em aplicações que utilizam eventos distribuídos.

Operações

- **snapshot:** fornece um método para minimizar a serialização que acontece sempre que entradas ou templates são enviadas ao espaço. É útil em casos onde é necessário realizar uma operação sobre uma mesma entrada sem necessitar fazer alguma modificação.

Banco de dados relacionais X JavaSpaces

- Um serviço JavaSpaces pode armazenar dados persistentes que poderão ser recuperados posteriormente. Porém, ele não é um banco de dados relacional ou orientado a objetos.

- JavaSpaces é um serviço projetado para ajudar a resolver problemas em computação distribuída e não para ser usado primariamente como um repositório de dados .

Banco de dados relacionais X JavaSpaces

- **Banco de dados relacional** armazena e manipula os dados diretamente através de uma linguagem de consulta;

- Entradas em JavaSpaces só podem ser recuperadas pelo tipo e forma serializada de cada campo, já que não possui uma linguagem geral para intermediar uma consulta. Sendo assim, não é possível retornar um conjunto de entradas;

Banco de dados relacionais X JavaSpaces

- **Banco de dados orientado a objetos** fornece uma imagem orientada a objetos do dado armazenado que pode ser modificado e utilizado.

- JavaSpaces não provê um mecanismo de persistência transparente, ou seja, dados não podem ser modificados dentro do espaço, de modo que todas as entradas são cópias dos objetos originais.

Banco de dados relacionais X JavaSpaces

JavaSpaces é projetado para um propósito diferente de um banco de dados relacional ou orientado a objetos.

- Pode-se dizer que JavaSpaces é algo entre um sistema de arquivos e um banco de dados. Um serviço JavaSpaces pode ser usado para armazenar dados persistentes, como armazenar dados preferenciais dos usuários, os quais podem ser recuperados posteriormente através do nome ou um identificador do usuário.
- JavaSpaces pode armazenar mais do que apenas dados. Isto significa que qualquer programa, dispositivo ou informação que seja baseado em objetos pode se juntar a um sistema JavaSpaces.

Conceitos Básicos de Aplicações JavaSpaces

- Entries
- Acesso ao espaço
  - classe SpaceAcessor
- Escrevendo no espaço
  - Operação Write
- Lendo ou retirando
  - Operações de Read e Take

Entries

Entries é o elemento básico de toda aplicação baseda em espaço, é através da troca de entries que processos podem se comunicar, sincronizar e coordenar suas atividades. Uma entry é apenas um objeto que segue algumas convenções, que garantem a segurança do seu trânsito através dos espaços.

```
import net.jini.core.entry.*;
public class KnightOfNi implements Entry{
  public Integer number;
 //um contrutor sem argumentos
 public KnightOfNi(){
}
//um Construtor com um argumento
public KnightOfNi(int i){
        number = new Integer(i);
}
//One Method
public void say(){
        System.out.println("<KnightOfNi+number+"> Ni!");
}
}
```

Esta classe precisa implementar a interface Entry, que é importada do pacote do javaspaces.

Ao invés de criar uma nova classe que implemente a interface Entry, é possível também extender um objeto que já implemente a interface Entry.

Resumo de Convenções para Entries

Aqui temos um resumo das convenções que precisam ser seguidas quando se está fazendo uma entry:

- Declare suas entries como classes públicas;
- Utilize campos públicos como referências de objetos (não tipos primitivos);
- Evite campos não-públicos (exceto campos static, transient e final).

Acesso ao espaço

O processo de achar o espaço e retornar, não é algo muito trivial, e também é muito utilizado em qualquer aplicação JavaSpaces.
Existem muitas maneiras de acessar um espaço. Por exemplo, um espaço pode ser registrado como:

- **Jini lookup service**
- **RMI registry**

Para tornar esta tarefa mais fácil, podemos usar uma classe que usa Jini ou RMI para localizar espaços.

Acesso ao espaço – classe SpaceAcessor

```
public class SpaceAcessor{
   public static JavaSpace getSpace(String name){
      try{
         if (System.getSecurityManager()==null){
            System.setSecurityManager(new RMISecurityManager());
         }
         if (System.getProperty("com.sun.jini.use.registry")==null){
            Locator locator = new
            com.sun.jini.outrigger.DiscoveryLocator();
            Finder finder = new com.sun.jini.outrigger.LookupFinder();
            return (JavaSpace)finder.find(locator, name);
         }
         else {
            RefHolder rh = (RefHolder)Naming.lookup(name);
            return (JavaSpace)rh.proxy();
         }
      }
      catch (Exception e) {
         System.err.println(e.getMessage());
      }
      return null;
   }
   public static JavaSpace getSpace(){
      return getSpace("JavaSpaces");
   }
}
```

Primeiro é setado o SecurityManager, que nesse caso é um RMISecurityManager

Em seguida é acessado a property com.sun.jini.use.regist

Determina se vai ser usado Jini ou RMI:
- **se tiver valor null, é utilizado Jini Lookup Service;**
- **Senão é a RMI registry**

O método estático getSpace= retorna um objeto que implementa a interface JavaSpaces. Esse método pode ter como parâmetro:

- **uma string, que é o nome do espaço que queremos acessar;**
- **Ou não ter parâmetros, passando o default.**

Acesso ao espaço – classe SpaceAcessor

```
public class SpaceAcessor{
   public static JavaSpace getSpace(String name){
      try{
         if (System.getSecurityManager()==null){
            System.setSecurityManager(new RMISecurityManager());
         }
         if (System.getProperty("com.sun.jini.use.registry")==null){
            Locator locator = new
            com.sun.jini.outrigger.DiscoveryLocator();
            Finder finder = new com.sun.jini.outrigger.LookupFinder();
            return (JavaSpace)finder.find(locator, name);
         }
         else {
            RefHolder rh = (RefHolder)Naming.lookup(name);
            return (JavaSpace)rh.proxy();
         }
      }
      catch (Exception e) {
         System.err.println(e.getMessage());
      }
      return null;
   }
   public static JavaSpace getSpace(){
      return getSpace("JavaSpaces");
   }
}
```

Usando Jini lookup service:

É usado duas classes do pacote com.sun.jini.outrigger(esse pacote encapsula o código necessário para fazer buscas):

A classe DiscoveryLocator sabe como localizar um Jini lookup service;

• A classe LookupFinder tem o método finder, que tem como parâmetros: um Locator e o nome do serviço. Retorna uma interface para o serviço.

No caso do serviço JavaSpace, esta interface é um proxy local, que implementa a interface JavaSpace e usa um protocolo privado para se comunicar como o espaço remoto.

Acesso ao espaço – classe SpaceAcessor

```
public class SpaceAcessor{
   public static JavaSpace getSpace(String name){
      try{
         if (System.getSecurityManager()==null){
            System.setSecurityManager(new RMISecurityManager());
         }
         if (System.getProperty("com.sun.jini.use.registry")==null){
            Locator locator = new
            com.sun.jini.outrigger.DiscoveryLocator();
            Finder finder = new com.sun.jini.outrigger.LookupFinder();
            return (JavaSpace)finder.find(locator, name);
         }
         else {
            RefHolder rh = (RefHolder)Naming.lookup(name);
            return (JavaSpace)rh.proxy();
         }
      }
      catch (Exception e) {
         System.err.println(e.getMessage());
      }
      return null;
   }
   public static JavaSpace getSpace(){
      return getSpace("JavaSpaces");
   }
}
```

Usando RMI registry:

Temos que chamar o método estático Naming.lookup do pacote java.rmi para procurar o nome no registro.

Se for achado, a referência para o objeto remoto é retornada.

No caso da implementação da SUN, esta referência é do tipo com.sun.jini.mahout.binder.RefHolder, e não um proxy do espaço.

O método proxy do objeto RefHolder retorna o proxy local do espaço.

Escrevendo no Espaço - Operação Write

- Entries são escritas no espaço, para que elas possam ser compartilhadas com outros processos, fazendo parte da Computação distribuída.
- Portanto, apenas objetos escritos no espaço estão visíveis por outras aplicações, caso contrário continuarão objetos locais sem uso para outras aplicações.
- Assumindo que temos uma referencia para o espaço na variável space (obtida pela classe SpaceAccessor), podemos escrever no espaço:

```
public void sendKnightOfNiIntoSpace(KnightOfNi kon){
try{
        space.write(kon, null, Lease.FOREVER);
   } catch(Exception e){
         e.printStackTrace();
   }
}
```

Escrevendo no Espaço - Operação Write

```
public void sendKnightOfNiIntoSpace(KnightOfNi kon){
try{
    space.write(kon, null, Lease.FOREVER);
     } catch(Exception e){
       e.printStackTrace();
     }
}
```

Método, recebe uma entry como parâmetro(kon)

Após ser escrita no espaço, a entrie é tratada como dado passivo. Não podemos invocar seus métodos ou examinar seus campos, sem primeiro obter uma cópia dela (com read ou take, como será visto mais adiante)

Operação write: O segundo paramêtro é a transação, no caso estamos passando uma referência null;
O terceiro parâmetro é o lease time, é o tempo no qual o objeto vai permanecer no espaço, especificado em milisegundos. No caso do nosso exemplo utilizamos uma constante, que diz que o objeto vai ficar para sempre no espaço(ou até ser removido explicitamente por uma aplicação).

Lendo ou retirando do espaço

As operações de read e take nos permitem conseguir cópias de entries ou então remover objeto do espaço. Para isso precisamos localizar este objeto dentro do espaço, e para isso é utilizado Associative Lookup.

Associative Lookup= A procura associativa é um método de procura bem interessante. Ao invés de ser dado um endereço ou localização dentro do espaço, é criada um template (esteriótipo) do objeto que queremos. Para criar este template, criamos uma instancia da classe que estamos procurando e então especificamos os valores de alguns campos que queremos, e deixamos os outros como null. Os valores null servem como wirldcards, ou seja, vão se associar com qualquer valor.

Uma entry se associa com um esterótipo quando cada um dos campos especificados no template se associam com o campo correspondente na entry.

```
KnightOfNi template = new KnightOfNi();
template.number = null;
KnightOfNi template = newKnightOfNi();
template.number = new Integer(1);
```

No exemplo, qualquer entry no espaço que for do tipo KnightOfNi, será associada com o nosso template.

Já no segundo exemplo, apenas a entry com o campo number igual a 1 será associada.

Operações de Read e Take

No exemplo criamos um novo template, instanciando a classe KnightOfNi, e colocando o campo number como null. Assim, a primeira entry do espaço que for encontrada que se associar com este template será retornada pela função read.

```
public int letOneKnightSay(){
KnightOfNi template = new KnightOfNi();
template.number = null;
try{
  KnightOfNi knight = space.read(template, null, Long.MAX_VALUE);
 knight.say();
 return knight.number.intValue();
}
catch(Exception e){
 e.printStackTrace();
return -1;
}
}
```

No caso do exemplo, utilizamos a contante Long.MAX_VALUE, que significa que a operação de read irá esperar indefinidamente.

Assim como a função de write, a função de read contêm mais dois parâmetros.
O segundo parâmetro tem a mesma função que na operação de write, é a transação.
O terceiro parâmetro especifica o tempo de espera da operação de read. Caso não achar uma entry que se associe com o seu template, ela vai bloquear pelo tempo especificado ou até que uma nova entry que se associe com o template seja adicionada no espaço.

Operações de Read e Take

A operação de take tem a mesma sintaxe e funcionamento da operação de read, a única diferença, é que ao invés de criar uma cópia da entry do espaço, a entry é removida do espaço e retornada pelo método.

Programa Exemplo - HelloWorld

Message.class

```
import net.jini.core.entry.Entry;  // Declaração da classe Message, que obrigatoriamente deve implementar Entry

public class Message implements Entry {
    public String content;  // Declaração da string content

    public Message() {  // Declaração do método Message()
    }
}
```

HelloWorld.class

São importadas as classes:

```
import jsbook.util.SpaceAccessor;  // SpaceAccessor (Para localizar o JavaSpace)
import net.jini.core.lease.Lease;  // Lease (Para indicar o tempo de espera)
import net.jini.space.JavaSpace;   // JavaSpace (Para utilizar os objetos do JavaSpace)

public class HelloWorld {  // Declaração da classe HelloWorld
    public static void main(String[] args) {

        try {       // Tratamento de Exceções

            Message msg = new Message();    // Instanciação de uma entry msg
            msg.content = "Hello World";    // content da entry msg é setado com "Hello World"

            JavaSpace space = SpaceAccessor.getSpace(); // Chamada do método GetSpace(classe SpaceAccessor), que retorn
                                                        // a instância de um objeto que implementa a interface JavaSpac

            space.write(msg, null, Lease.FOREVER); // Grava uma cópia da entry no space
                                                   // Parâmetros: 1 - entry gravada
                                                   //             2 - tipo de transação
                                                   //             3 - tempo que a entry deve ser armazenada no space
                                                   //                 Lease.FOREVER indica tempo indefinido

            Message template = new Message();   // Para ler a entry é usado um template

            Message result =(Message)space.read
                (template, null, Long.MAX_VALUE);  // Faz a leitura do space
                                                   // Parâmetros: 1 - entry que receberá os valores da leitura
                                                   //             2 - tipo de transação
                                                   //             3 - tempo que a operação de leitura deve esperar até
                                                   //                 conseguir ler (TIMEOUT)
                                                   //                 Long.MAX_VALUE indica tempo indefinido de espera

            System.out.println(result.content);  // Resultado é impresso na tela

        } catch (Exception e) {         // Tratamento de Exceções
            e.printStackTrace();
        }
    }
```

Transações

- Definição
- Participantes de uma transação
- Operações
- Usando a Transaction Manager
- Criando uma transação
- Propriedades de transações em espaços
- Programa Exemplo

Transações

A primeira operação a fazer para utilizar transações com operações baseadas em espaço é perguntar a **Transaction Manager** para criar a transação e gerenciá-la por um determinado tempo de lease.

Então, passamos a transação para cada espaço da operação que gostaríamos que ocorresse sob a transação. Admitindo não haver nenhum problema ao longo do caminho, então explícitamente damos um commit na transação, que resultará em todas as operações completadas.

Participantes de uma transação

Jini provê um serviço de transação que gerencia um conjunto de participantes por meio de um processo de transação.

O serviço de transação comanda os participantes até o ”protocolo commit 2-fases”, um protocolo padrão que garante que todos os participantes completem suas respectivas operações na transação; no caso de falha em um participante, nenhum deles a completará.

- Se qualquer problema ocorrer, a transação é abortada, e deixará o espaço inalterado;
- A transação também pode ser abortada por meio da Transaction Manager se, por exemplo, o lease da transação vença.

Participantes de uma transação

Quando escrevemos um entrada (entry) dentro de um espaço sob uma transação, o entry é invisível a qualquer cliente que tente ler, pegar ou notificar ela fora da transação.

Se o entry é levado para o interior da transação, ele nunca será visto de fora de uma transação. Se a transação abortar, o entry é descartado.

Uma vez que a transação executa o commit, o entry estará disponível para leituras, recebimentos e notificações de fora da transação.

Operações de uma Transação

- **write:** escreve uma entrada em um serviço JavaSpace. Uma entra escrita não é visível de fora da transação, até que a transação execute com sucesso.

  Space.write(Entry entry, Transaction txt, Lease lease);

- **read:** lê uma entrada do serviço JavaSpace que compatibiliza com o dado modelo. Um read pode combinar qualquer entrada escrita sob essa transação ou o espaço inteiro.

Operações de uma transação

- **Take:** lê uma entrada do serviço JavaSpace que compatibiliza com o dado modelo, removendo-o do espaço.
- **notify:** notifica um objeto especificado quando entradas que compatibilizam o dado modelo são escritas em um serviço JavaSpaces.(passa uma entrada e espera a notificação de que houver escrita a partir dessa entrada).

Usando a Transaction Manager

Para fazer uso de transações, primeiro precisamos acessar a transaction manager, para criar e preservar as transações. Como todos serviços JINI, o serviço lookup retorna um proxy objeto para a transaction manager; neste caso específico ficaremos esperando por um serviço que execute a interface TransactionManager.

```
TransactionManager mgr = TransactionManagerAcessor.getManager();
```

Aqui chamamos o método estático getManager() da classe TransactionManagerAcessor, a qual retorna para a TransactionManager o objeto proxy.
Como esse proxy em mãos, podemos criar a transação que irá gerenciar um conjunto de operações, a qual executa a interface TransactionParticipant.

Como criar uma transação:

Declara-se a variável do tipo Transaction.Create, a qual é o tipo do objeto que será retornado quando perguntarmos a Transaction Manager para criar uma nova transação.

```
TransactionManager mgr = TransactionManagerAcessor.getManager();
Transaction.Create trc = null;
try {
        trc = TransactionFactory.create(mgr , 300000);
    } catch ( Exception e ) {
        System.err.println("Could not create transaction"+e);
}
```

Para criarmos a transação, usamos a classe TransactionFactory que chama o método estático create, ao qual leva a transaction manager e um tempo de lease(milisegundos) como parâmetro e cria a transacão que irá ser gerenciada pelo manager que também cuidará do tempo de lease dado.
Se a chamada para create é bem sucedida, o objeto Transaction.Create é retornado e nomeado para a variável trc.
Se alguma coisa houver de errado durante a criação da transação, uma exceção, é lançada no lugar.

Propriedades de transações em espaços

- **Atomicidade** - as mudanças no espaço, ocorridas sob uma transação, são atômicas. Todas as mudanças ocorrem, ou nenhuma ocorre. Estas mudanças incluem as operações read, write, take e notify, que ocorrem sob uma transação.

- **Consistência** - as mudanças ocorridas não violam a correturde do estado do espaço. O programa deve ser correto. Se o algoritmo é correto, a transação garante que o espaço não será colocado num estado inconsistente.

Propriedades de transações em espaços

- **Isolamento** - quando transações executam concorrentemente, uma transação não afeta a outra. Quando você escreve o seu programa, não precisa se preocupar com o que está acontecendo em outras transações.

- **Durabilidade** - quando uma transação commita, suas mudanças sobreviverão a falhas. Em espaços persistentes, o resultado de uma transação sobreviverá a falhas do espaço. Se a implementação do espaço não provê persistência, as mudanças de uma transação não irão sobreviver, se houver uma falha no espaço após o commit.

Programa exemplo com transações

Para demonstrar uma transação baseada em espaço podemos remover uma Entry de um JavaSpace e escrever em outro JavaSpace, dentro de uma transação.

Programa exemplo com transações

Primeiro, definimos uma classe chamada Message que será usada para instanciar a entrada (Entry) a qual será movida.

```
public class Message implements Entry {
      public String content;
      //constructor vazio
      public Message(){}
   }
```

Programa exemplo com transações

Agora escrevemos o método que irá gravar uma entrada (Entry Mesage) em um JavaSpace.

```
private void createMessage() {
Message msg = new Message();
msg.content = “test”;
try {
        sourceSpace.write(msg,null,Lease.FOREVER);
    } catch ( Exception e ) {
        System.err.println(“Cant write message” + e );
    }
}
```

Programa exemplo com transações

O próximo passo é escrevermos o método relayMessage():

- remove a mensagem do space fonte;
- escreve no space destino, dentro de uma transação.

```
private void relayMessage() {
   TransactionManager mgr = TransactionManagerAccessor.getManager();
   Message template = new Message();
   Message msg = null;
   Transaction.Created trc = null;
   try {
            trc = TransactionFactory.create( mgr, 300000 );
       } catch ( Exception e ) {
            System.err.println("Could not create transaction" + e );
       }
  Transaction txt = trc.transaction;
  try {
       try {
          template.content = "test" ;
          msg = (Message)sourceSpace.take(template,txn,Long.MAX_VALUE);
           targetSpace.write(msg,txn,Lease.FOREVER);
        } catch ( Exception e ) {
           txt.abort();
           return;
       }
          txn.commit();
       } catch ( Exception e ) {
          System.err.println("Transaction Failed");
          return;
       }
    }
```

Primeira coisa é chamar o método estático getManager da class TransactionManagerAcesso (retorna a TransactionManager.

Depois define duas variáveis Message:

- **Uma pra servir como espaço origem**
- **Outra pra espaço destino**

Considerações Finais

- JavaSpace é uma tecnologia decididamente inovadora. A comunicação através de um espaço compartilhado permite dinamizar sistemas distribuídos, abertos e dinâmicos em que requerentes e fornecedores de um serviço são altamente prescindidos de acoplamento;
- A escalabilidade é transparente, sendo possível adicionar ou remover um ou mais nós a qualquer momento, mesmo com o processamento em andamento;
- Por fim, espera-se que a proposta desse artigo possa contribuir com novos estudos e implementações de aplicações distribuídas com a tecnologia JavaSpaces.

Referências Bibliográficas

- http://java.sun.com/developer/technicalArticles/tools/JavaSpaces
- http://www.inf.ufrgs.br/procpar/disc/cmp167/trabalhos/sem2001-1/tutorial/Tutorial_Javaspaces/index.htm
- http://www.inf.ufrgs.br/procpar/disc/cmp167/trabalhos/mp2000-1/FernandaBocoli/JavaSpace.html
- http://www.teses.usp.br/teses/disponiveis/55/55134/tde-08032003-012015
- http://tede.ufsc.br/teses/PGCC0785.pdf
- http://www.inf.ufrgs.br/procpar/disc/cmp157/trabalhos/sem99-2/giovani/JavaSpac.htm
- http://www.inf.unisinos.br/~barbosa/uniinfo/javaspaces.pdf