Linguagem para desenvolvimento de aplicações distribuídas JavaSpaces

Luthiano Rodrigues Venecian[1]

[1] Programa de Pós-Graduação em Informática
Mestrado em Ciência da Computação
Universidade Católica de Pelotas (UCPEL), RS, Brasil

venecian@gmail.com

Resumo. *A computação distribuída permite que diferentes equipamentos possam ser interligados para realizar uma determinada tarefa em conjunto. Este artigo aborda um estudo sobre a tecnologia para desenvolvimento de aplicações distribuídas JavaSpaces, uma extensão da linguagem Java que permite a criação de repositórios de objetos distribuídos através de redes de computadores.*

1. 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.

A computação distribuída é um tipo de computação, na qual as aplicações são compostas por um conjunto de processos que estão distribuídos através da rede, cooperando para atingir um objetivo comum. Entretanto, a aplicação pode ser vista como uma coleção de componentes e objetos que podem ser alocados em diferentes computadores conectados a uma rede.

As vantagens da construção de aplicações utilizando essa abordagem são muitas [4] [6]:

- É possível conseguir maior desempenho apenas adicionando novos computadores no sistema onde a aplicação esteja rodando, isto é, a aplicação é facilmente escalável;
- Desde que os processos rodando em computadores diferentes estão se comunicando uns com os outros, estes podem compartilhar seus recursos;
- Além disso, a tolerância a falhas é mais facilmente obtida, pois se um processo falhar ou um computador ficar off-line, o resto do sistema continua executando.

Aplicações distribuídas são mais difíceis de projetar e implementar do que aplicações Standalone. Esta complexidade adicional é, na maioria dos casos, devido a grande variedade de plataformas de hardware e software.

Entretanto, esta não é a única dificuldade que deve ser considerada quando são projetadas e implementadas aplicações distribuídas. Os processos de uma aplicação distribuída precisam se comunicar para conseguirem trabalhar em conjunto. Uma vez que a comunicação se dá sobre uma rede, os tempos de transmissão são geralmente altos quando comparados a velocidade do processador dos computadores envolvidos. O tempo

de transmissão de uma origem ao seu destino é denominado latência, e deve ser levado em conta no projeto de aplicações distribuídas.

Outra dificuldade existente no projeto de aplicações distribuídas é a sincronização entre os processos, pois na maioria das vezes um processo tem que esperar por alguma informação de outro processo para prosseguir sua execução.

A tolerância a falhas é outro ponto crucial no projeto de aplicação distribuídas, pois deve-se manter um estado global consistente no caso de falhar parciais

No intuito de se evitar estes tipos de problemas, os desenvolvedores de aplicações distribuídas podem fazer uso do JavaSpaces. A tecnologia JavaSpaces é uma ferramenta simples e expressiva para construção de sistemas distribuídos.

Abordando este contexto, o presente artigo está estruturado em 5 seções. Inicialmente é realizada uma introdução sobre a temática discutida. A seção 2 é apresenta a tecnologia JavaSpaces, sua origem, funcionalidade, vantagens e operações. Os conceitos básicos [1] são apresentados na seção III. Na seção IV é apresentado dois programas exemplos [2] [7] e na última seção (V) são apresentadas as considerações finais.

2. Tecnologia JavaSpaces

JavaSpaces é uma especificação desenvolvida pela Sun Microsystems para o paradigma de programação baseada em espaço [4]. A especificação desenvolvida 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 (é uma arquitetura de rede aberta que habilita desenvolvedores criar serviços centrados na rede - implementados em hardware e software - que são altamente adaptáveis a mudanças).

A construção de aplicações distribuídas geralmente envolve a troca de mensagens entre processos ou invocação de métodos em objetos remotos. Nas aplicações baseadas em JavaSpaces, os processos não se comunicam diretamente entre si.

A tecnologia JavaSpaces provê um modelo de programação que visualiza a aplicação como uma coleção de processos cooperando através de um fluxo de objetos para dentro e fora de um ou mais espaços. O espaço é um repositório de objetos compartilhado. Os processos usam o repositório como um mecanismo de troca e armazenamento persistente de objetos. Ao invés dos processos comunicarem diretamente, eles se coordenam trocando objetos através dos espaços (figura 1).

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.

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 (estado) e inseri-lo no espaço novamente como um novo objeto.

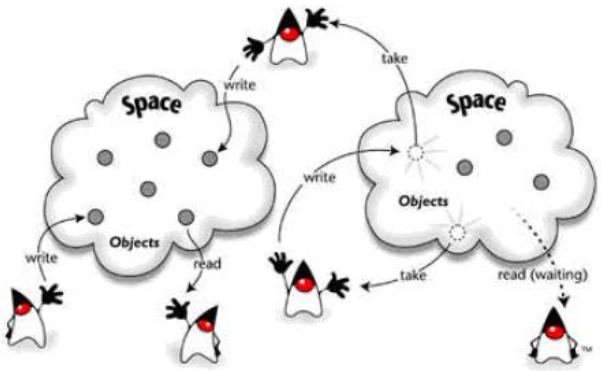

Figure 1. JavaSpaces

2.1. 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 [5].

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.

2.2. O que é um 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 [1].

Diferente de outros repositórios de objetos, processos não podem alterar objetos dentro de um espaço ou invocar seus métodos diretamente, é necessário explicitamente remover ele, alterar e então reinserir no espaço.

2.3. Funcionalidades

A lista abaixo fornece algumas funcionalidades associadas ao JavaSpaces [3]:

- **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.

- **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.
- **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.

2.4. Vantagens da tecnologia JavaSpaces

A tecnologia JavaSpaces apresenta as seguintes vantagens [3] [5]:

- 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.
- 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. A tecnologia JavaSpaces provê estas funcionalidades.

2.5. Operações no espaço

Javaspaces possui um conjunto de operações reduzido. Para manipular qualquer objeto que implementa um serviço JavaSpaces, são fornecidas as seguintes operações [4] [5]:

- **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 - (escreve um objeto 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 - (pega uma cópia de um objeto do espaço);
- **take:** funciona como read, exceto que o objeto compatível com o template é removido do espaço - (remove um objeto 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. Além disso, 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.
- **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.

3. Conceitos Básicos de Aplicações JavaSpaces

3.1. Entries

Entries é o elemento básico de toda aplicação baseda em espaço, é através da troca de entries(entradas) que processos podem se comunicar, sincronizar e coordenar suas atividades.

Entry é um objeto que segue algumas convenções, para tornar a passagem pelo espaço ”segura”. Um Exemplo de uma entry:

```
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!");
    }
}
```

Aqui criamos uma classe chamada KnightOfNi. 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.

3.2. Acesso ao espaço

Existem muitas maneiras de acessar um espaço. Por exemplo, um espaço pode ser registrado como um Jini lookup service, que é consultado para se obter uma referência remota

ao espaço. Ou o espaço pode ser registrado com um RMI registry. Para tornar esta tarefa mais fácil, podemos usar uma classe que usa Jini ou RMI para localizar espaços:

```
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");
   }
}
```

O método estático getSpace retorna um objeto que implementa a interface JavaSpace. Este método pode ter como parâmetro uma String, que é o nome do espaço que queremos acessar. Ou pode não ter parâmetros, que chama getSpace passando o nome default do espaço - ”JavaSpaces”. Primeiro nós setamos o SecurityManager, que neste caso é um RMISecurityManager. Em seguida acessamos a property com.sun.jini.use.registry. Esta property determinha se vamos usar Jini ou RMI para localizar o espaço. Se a property tiver valor null, usamos Jini lookup service, caso contrário, usamos RMI registry.

Se usarmos Jini lookup service, temos que usar duas classes do pacote com.suj.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, e 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.

Se usarmos 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 Microsystem, 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.

3.3. 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 mandar nosso cavaleiro para o espaço facilmente:

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

Nosso método sendKnightOfNiIntoSpace recebe uma entry como parametro, e então escreve ele no espaço. Se este método fosse chamado várias vezes, com a mesma entrie, várias copias do cavaleiro seriam gravadas dentro do espaço. 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).

Olhando um pouco melhor a operação de write, podemos ver que junto com a entry foram passados mais dois parâmetros. O segundo paramêtro é a transação, é a partir dela JavaSpaces suporta multiplas operações serem atômicas. No caso estamos passando uma referência null, isso quer dizer que não estamos utilizando transações, então apenas estamos fazendo uma única operação atômica.

O segundo parâmetro é o lease time. 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). Poderíamos passar como parâmetro 60000, que seria o mesmo que dizer que a entry permaneceria no máximo 1 minuto no espaço.

3.4. Lendo ou retirando do Espaço - Operações de Read e Take

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

3.4.1. 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 cria 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 de associam com o campo correspondente na entry.

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

No primeiro exemplo acima, qualquer entry no espaço que for do tipo KnightOfNi, ou seu filho, será associada com o nosso template. Já no segundo exemplo, apenas a entry com o campo number igual a 1 será associada.

3.4.2. Regras se associação

Aqui temos as regras de associação mais explícitamente:
1. O estiótipo deve ser do mesmo tipo da entry ou um supertipo(pai) da entry.
2. Cada campo do template deve associar com o campo correspondente da entry, onde:

- Se um campo do template for null, então ele se associa com o campo correspondente da entry;
- Se um campo do template for especificado então ele se associa com o campo correspondente da entry caso tiverem os mesmos valores.

3.4.3. Operações de Read e Take

```
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 exemplo acima 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.

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. No caso do exemplo, utilizamos a contante Long.MAX-VALUE, que significa que a operação de read irá esperar indefinidamente. O tempo de espera é expresso em milisegundos.

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.

3.4.4. Campos Primitivos

Campos primitivos não são utilizados na associação de entries, apenas referências. Como visto anteriormente, nas referências são utilizadas null como wilrdcards, sendo que o mesmo não é possível fazer com os tipos primitivos. Por este e outros motivos, os desenvolvedores do JavaSpaces optaram por não associar tipo primitivos. Entrão, ao invés de tipos primitivos, será necessário uitlizar os objetos correspondentes dos tipos primitivos.

3.5. 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
- Implemente a interface Entry, extenda uma classe que implemente, ou implemente uma interface que implemente a interface Entry
- Cria um construtor sem argumentos
- Utilize campos públicos como referências de objetos (não tipos primitivos)
- Evite campos não-públicos (exceto campos static, transient e final)
- Certifique-se que todos campos são serializáveis

3.6. Banco de dados relacionais e 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 (embora existam muitos usos de armazenagem de dados para aplicações JavaSpaces). Comparando JavaSpaces com banco de dados, tem-se que [4]:

- um 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;

- um 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.

Estas diferenças existem porque 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. Mais importante ainda, 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. Dessa forma, pacotes de dados são tratados exatamente como qualquer outro objeto mantido no espaço. Esta habilidade aumenta muito a capacidade de coordenar diferentes funções e processos em uma rede.

3.7. Transações

Usar transações com, operações baseadas em espaço, tipicamente perguntamos primeiro 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.

3.7.1. 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. Quando escrevemos um entrada (entry) dentro de um spaço sob um 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 dsiponível para leituras, recebimentos e notificações de fora da transação.

3.7.2. Operações de uma Transação

A semântica das operações para se usar transações:

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.

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).

3.7.3. 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.

Agora veremos como criar uma transação:

```
Transaction.Create trc = null;
try {
        trc = TransactionFactory.create( mgr , 300000 );

        } catch ( Exception e ) {
        System.err.println("Could not create transaction" + e );
}
```

Primeiro declaramos 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.

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 transação 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.

3.7.4. Propriedades de transações em espaços

Transações baseadas em espaço aderem a um conjunto de propriedades conhecido como ACID properties.

Essas propriedades são:

- **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.
- **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.

4. Programa exemplo

Nessa sessão serão demostrados 2 programas exemplos. O primeiro uma aplicação bastante simples,um HelloWord e o segundo utilizando JavaSpaces com transações:

4.1. HelloWorld

Message.class

```
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

```
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();
        }
    }
```

4.2. Programa exemplo com Transações

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

        //construtor

        public Message(){}
}
```

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 );
        }
}
```

O próximo passo é escrevermos o método relayMessage() que removerá a mensagem do espaço fonte e escreverá no espaço destino, dentro de uma transação.

A primeira coisa a fazer neste método é chamar o método estático getManager da classe TransactionManagerAccesor, como explicado anteriormente, que irá retornar a TransactionManager. Depois definimos duas variáveis Message: um para servir como um space origem e outra para ser o space destino.

```
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;
}
}
```

5. 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.

References

[1] JavaSpaces(acesso em 06/2008)
http://www.inf.unisinos.br/ barbosa/uniinfo/javaspaces.pdf

[2] Tutorial para iniciantes, explicando todo o processo de instalação e configuração da Tecnologia Jini v1.1 na plataforma Windows, bem como todas as dicas para inicializar o serviço JavaSpaces e rodar aplicações(2001)
http://www.inf.ufrgs.br/procpar/disc/cmp167/trabalhos/sem2001-1/tutorial/Tutorial-Javaspaces/index.htm

[3] JavaSpaces(acesso em 06/2008)
http://tede.ufsc.br/teses/PGCC0786.pdf

[4] Começando com a tecnologia JavaSpaces(2005)
http://java.sun.com/developer/technicalArticles/tools/JavaSpaces

[5] Implementação de espaços de tuplas do tipo JavaSpaces(2002)
http://www.teses.usp.br/teses/disponiveis/55/55134/tde-08032003-012015

[6] Especificação de JavaSpaces(1999)
http://www.inf.ufrgs.br/procpar/disc/cmp157/trabalhos/sem99-2/giovani/JavaSpac.htm

[7] Jini/JavaSpace - Modelo de Transações(2000)
http://www.inf.ufrgs.br/ sawicki/cmp167/T2/T2.html