Implementação de espaços de tuplas do tipo JavaSpaces

Orlando de Andrade Figueiredo

Orientador: Prof. Dr. Dilvan de Abreu Moreira

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP – São Carlos
Outubro de 2002

Agradecimentos

Ao professor Dilvan, pela acolhida e pelas boas idéias.

A tantos professores e funcionários do ICMC e da USP, por desempenharem suas funções tão bem.

À CAPES pela bolsa de estudos.

À Fundação Educacional de Ituiutaba por ceder equipamentos.

Aos colegas do grupo, Elaine, Werley, Carlos Estombelo, Percival, Flávia, Carbol, Richard, Pat, Gustavo e Vanderley.

À gente boa do Laboratório Intermídia, professora GraçaP, professor Edson, Renato Bulcão, Claudinha, Laércio, Otávio, Juninho, Renan, Rudinei, Dalton, Martins, Alessandra, Toño, Débora, Andréa, Pedro, Wagner, Gusfaria, Rijomar.

E a tantos colegas com que pude conviver: Luciano "Nanuque", Valmir, Rodrigo Elias, Hévila, Gláucia, Kelly, Simone, Adenilso, Kaminsky, Percy, Alex, Ernesto, Klinger, Kiko, Ricardo Siquieri, Ricardo Ribeiro, Mário Meireles, Álvaro, José Flávio, Leandro Henrique, Rodrigo, Giovana, Esdras.

A João Cardeal, Roberto Bittencourt, Paulo Gustavo, José Carlos Oliveira, Wladmir Sybine, Ana Isabela, Antônio César, Luís Edmundo, Caiouby, professores e alunos da Universidade Estadual de Feira de Santana e da Universidade Federal da Bahia, que, lá na Bahia, muito me incentivaram a partir para essa aventura.

A Íris Barcelos, Saulo, Demétrio, Éder, professores e alunos da Fundação Educacional de Ituiutaba, que, lá em Minas, me incentivaram a concluí-la.

A tia Jacy, tio José Augusto e tia Gláucia, pelo apoio em São Paulo.

Um agradecimento especial a meu pai, João Orlando, e a minha mãe, Maria Elena, que estiveram sempre me apoiando nesse período, seja por email ou telefone. Sem falar nas orações...

Resumo

Um espaço de tuplas tem como função criar uma abstração de memória compartilhada sobre um sistema distribuído. Por propiciar modelos de programação muito simples e com baixo acoplamento entre os elementos do sistema, espaços de tuplas têm sido empregados na construção de sistemas distribuídos complexos. O espaço de tuplas JavaSpaces é um dos mais populares espaços de tuplas para a linguagem Java. Ele tem como características relevantes a conformidade a objetos, a persistência e o emprego de transações.

As atuais implementações de JavaSpaces apresentam restrições como: complexidade de configuração, limitação de alcance e não serem abertas. Por "complexidade de configuração" entende-se ter que usar boa parte da infra-estrutura Jini (feita para facilitar o desenvolvimento e administração de sistemas distribuídos) e o *Remote Method Invocation* (mecanismo de chamadas remotas padrão no ambiente Java), mesmo quando eles seriam dispensáveis. Por "limitação de alcance", entende-se não poder usar as implementações sobre redes amplas, como a Internet. Por "não ser aberto" entende-se que: ou o código fonte não está disponível ou o código fonte e o aplicativo são distribuídos por licenças de software proprietárias ou o uso do software requer algum componente proprietário.

Um projeto de espaço de tuplas em conformidade com a especificação JavaSpaces e que busca contornar as restrições acima é apresentado neste trabalho. São destaques do projeto proposto:

1. Dispensar o *Remote Method Invocation* pois utiliza *sockets* diretamente;
2. Implementar a persistência sobre bases de dados relacionais;
3. Suscitar o emprego de um mecanismo direto para obtenção de *proxies* Jini.

As características 1 e 3 simplificam a configuração do espaço de tuplas e viabilizam o seu emprego da Internet. A característica 2 viabiliza uma implementação baseada em software aberto. Um protótipo foi implementado para verificar as idéias propostas.

Abstract

Tuple spaces create an abstraction layer that uses shared memory semantics for data distribution over distributed systems. Tuple spaces are used to implement simple and loosely-coupled programming models. They have been used for building complex distributed systems. JavaSpaces is a popular tuple space specification for the Java language. It is object-oriented, persistent and transactional.

There are some implementations of the JavaSpaces specification, but either they cannot operate over the Internet or they are not open source. To address these issues, this work presents a new JavaSpaces implementation (JuspSpaces), which has three important features:

1. It is completely implemented using sockets; it does not use the Java Remote Method Invocation (RMI);
2. Relational databases are used to implement persistence;
3. It allows alternative ways to get proxies for services in a Jini like system.

The first and third features simplify the JuspSpaces setup and allow it to work over the Internet. The second feature makes this implementation more reliable, by using a proven storage technology (relational database) that is also available as open software. A prototype for JuspSpace was developed to test the proposed ideas.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contexto e Motivação

O desenvolvimento de aplicações distribuídas apresenta dificuldades consideráveis e, ao longo dos anos, definir ferramentas que auxiliem esse desenvolvimento tem sido uma necessidade que ocupou grande parte dos pesquisadores da área. O *Remote Procedure Call* (RPC) e, seu sucessor no mundo da orientação a objetos, o *Common Object Request Broker Architecture* (CORBA), são exemplos popularmente conhecidos. Uma ferramenta menos conhecida, embora histórica, é o espaço de tuplas.

Um espaço de tuplas tem como função criar uma abstração de memória compartilhada sobre um sistema distribuído. Processos distintos do sistema podem executar uma tarefa comum, usando o espaço de tuplas como meio de comunicação, coordenação e sincronização.

O conceito de espaço de tuplas surgiu em meados dos anos 1980 e, nos primeiros anos, a aplicação-alvo de seus idealizadores foi a programação paralela. Somente nos anos 1990, os espaços de tuplas foram explorados como ferramenta de programação distribuída. Um dos atrativos dos espaços de tuplas nesse contexto é a simplicidade, como demonstra a declaração de Ken Arnold, o idealizador do espaço de tuplas JavaSpaces[1]:

> "Uma instituição de pesquisa (cujo nome será omitido) substituiu 20000 linhas de código escritos em Java, que consumiram seis meses de trabalho de três pessoas para ficarem prontas, por 2000 linhas de código escritas em JavaSpaces, que uma pessoa levou seis semanas escrevendo. No mínimo, são 18000 pontos possíveis de falha a menos, sem contar que é mais fácil de entender e depurar."

Outro grande apelo que os espaços de tuplas têm é a característica de baixo acoplamento entre os elementos que participam do sistema. Bishop e Warren (2002)

[1] Mensagem para a lista de discussão *JavaSpaces-Users* no dia 22/1/2002.

comentam a adequação de espaços de tuplas a problemas de sistemas distribuídos no seu prefácio:

> "Tem sido um grata surpresa descobrir que espaços de tuplas resolvem naturalmente muitos dos problemas tradicionais que estão associados à construção de sistemas distribuídos com boa resposta à escala. Isso não quer dizer que espaços de tuplas sejam o "remédio para todos os males", mas quando olhamos para os métodos que utilizamos nas últimas duas décadas para construir e conectar sistemas de software distribuídos, parece impossível pensar em construir um sistema distribuído 'real' sem um JavaSpace.(...)"

> "Obviamente, espaços de tuplas não são apropriados para todos os sistemas, (...) mas, à medida em que os sistemas se tornam mais complexos, JavaSpaces ajudam a contornar problemas como falhas parciais e complexidade crescente."

O primeiro espaço de tuplas recebeu o nome de Linda [Gelernter, 1985]. Ele trabalha em associação com diversas linguagens de programação, em especial, com as linguagens C e Fortran, as mais usadas em análise numérica e computação paralela. Não tardou, surgiram outros espaços de tuplas, quase sempre acrescentando alguma variante ao modelo básico implementado em Linda.

De particular interesse para este trabalho são os espaços de tuplas que operam em consórcio com a linguagem Java. O mais popular entre eles é chamado JavaSpaces, e foi proposto pela Sun Microsystems, a mesma empresa que criou a linguagem Java. Outro exemplo notável de espaço de tuplas em Java é o T-Spaces, da IBM.

A união do modelo de espaço de tuplas com a linguagem Java trouxe benefícios sensíveis. A linguagem Java tem características que favorecem o desenvolvimento de aplicações distribuídas, dentre elas a portabilidade, o suporte natural à movimentação de código e a segurança. Não é à toa que Java vem sendo a linguagem de escolha em algumas categorias de aplicações distribuídas tais como agentes móveis.

JavaSpaces é uma especificação. Isso quer dizer que pode haver mais de uma implementação para JavaSpaces. Até meados de 2002, três implementações eram conhecidas:

- *Outrigger*, desenvolvida pela Sun para servir de referência para as demais implementações e para divulgar o JavaSpaces; é distribuída gratuitamente segundo os termos da licença SCSL (Sun Community Software License),
- *GigaSpaces*, a primeira implementação comercial a surgir e
- *Autevo*, da Intamission, outra implementação comercial.

A especificação JavaSpaces está fortemente vinculada ao sistema Jini, também de autoria da Sun. O Jini consiste em uma infra-estrutura que pretende simplificar o desenvolvimento, instalação e administração de sistemas distribuídos. O Jini é baseado na linguagem Java.

Jini é uma das mais concretas propostas no sentido de se implementar sistemas computacionais que possam ser mantidos com pouca ou quase nenhuma intervenção humana. Vários serviços e protocolos definem a infra-estrutura Jini, como, por exemplo, o serviço de *lookup* e o serviço de transações distribuídas. Colocar no ar essa infra-estrutura implica fazer funcionar vários servidores. Além disso, nas implementações de Jini atuais, o serviço de *lookup* requer uma rede com suporte a *broadcasting*, o que é um limitante. Uma aplicação dependente do servidor de *lookup* tem dificuldades para operar sobre a Internet.

O fato é que não há hoje como executar o espaço de tuplas JavaSpaces sem toda a infra-estrutura de Jini ao redor. É desejável que fosse possível executar apenas o espaço de tuplas, a exemplo do que acontece com T-Spaces, que funciona a partir de um único servidor. Executar JavaSpaces sem alguns dos demais servidores é permitido por Jini, embora isso não esteja tão claro mesmo à comunidade que acompanha essa tecnologia. As implementações existentes nem sequer mencionam essa possibilidade.

É difícil encontrar uma implementação satisfatória de JavaSpaces. As implementações atuais oferecem funcionalidades especiais, cada uma a seu modo, mas mesmo assim, ainda requerem a infra-estrutura Jini. Outra restrição que atinge a todos, é que, como produtos de software, esses espaços de tuplas possuem licenças proprietárias. Atualmente é possível adquirir cópias gratuitas pela Internet, mas não há garantias que isso permanecerá assim indefinidamente.

1.2 Objetivos

O objetivo principal deste trabalho é o desenvolvimento de um espaço de tuplas em conformidade com a especificação JavaSpaces com as seguintes características adicionais:

- ser de fácil instalação e configuração através da redução de serviços Jini envolvidos em sua operação;
- ser capaz de operar em redes de amplo alcance como a Internet; e

- ser baseado exclusivamente em softwares livres de modo a poder ser distribuído sob uma licença de software livre.

Para implementar um espaço de tuplas conforme a especificação JavaSpaces, problemas menores precisam ser tratados:

- quais mecanismos de persistência usar;
- como recuperar a informação armazenada;
- como controlar o acesso concorrente aos dados;
- como implementar a semântica transacional, incluindo a recuperação de falhas;
- como implementar a comunicação entre cliente e servidor.

Na totalidade dos aspectos a serem considerados, implementar um espaço de tuplas do tipo JavaSpaces guarda semelhanças com a implementação de bases de dados orientadas a objetos.

1.4 Organização da dissertação

Este capítulo 1 contém a motivação e os objetivos do trabalho.

O capítulo 2 trata do modelo de espaço de tuplas original. Constam deste capítulo a descrição do modelo, exemplos de aplicações e argumentos a favor do modelo. O espaço de tuplas enfocado é o Linda.

Diversos espaços de tuplas surgiram com o passar dos anos. Cada um deles trouxe contribuições ao modelo original, que precisam ser consideradas quando se projeta a implementação de um espaço de tuplas. O objetivo do capítulo 3 é apresentar essas variações, que giram em torno de questões como segurança, orientação a objetos, novas operações, etc.

O capítulo 4 é uma revisão dos principais problemas associados à implementação de um espaço de tuplas em Java (JavaSpaces), tais como serialização, RMI, *sockets*, persistência de objetos.

No capítulo 5, é apresentada uma implementação para um espaço de tuplas em Java usando JavaSpaces. São mostrados os módulos principais e suas funcionalidades. Alguns detalhes da implementação do protótipo montado e alguns testes realizados também fazem parte deste capítulo.

O capítulo 6 é o capítulo de conclusões do trabalho, que aponta suas principais contribuições e sugere trabalhos futuros.

O apêndice A apresenta as principais modificações na versão corrigida dessa dissertação em relação à versão original que foi defendida peranta a Banca Examinadora.

Capítulo 2

O modelo de espaço de tuplas original

2.1 Considerações iniciais

Linda foi o primeiro espaço de tuplas. No início, Linda [Gelernter, 1985] foi criado para mostrar a viabilidade do modelo de espaço de tuplas, que acabara de ser proposto. Hoje, Linda tornou-se um produto de software comercial maduro. Outros espaços de tuplas surgiram, trazendo variações ao modelo de espaço de tuplas (Capítulo 3), mas Linda manteve-se como uma referência do modelo de espaço de tuplas original.

O objetivo do presente capítulo é apresentar o modelo de espaço de tuplas e discutir seus aspectos mais relevantes. Para tal, será usado o espaço de tuplas Linda, pelos motivos já expostos. Na Seção 2.2, está a definição do modelo. As Seções 2.3 e 2.4 referem-se a Linda. Alguns exemplos de aplicações que podem ser montadas com Linda são encontrados na Seção 2.5. Esses exemplos mostram a propriedade mais importante dos espaços de tuplas, que é o desacoplamento, discutido na Seção 2.6. Há, ainda, uma discussão sobre a implementação do modelo na Seção 2.7, que antecede as considerações finais.

2.2 O modelo de espaço de tuplas

O modelo de espaço de tuplas é um modelo de memória. As operações essenciais de um espaço de tuplas são operações de escrita e leitura.

Um espaço de tuplas possui, sobre um sistema distribuído, a semântica de memória compartilhada, isto é, processos espalhados pelo sistema distribuído têm no espaço de tuplas um dispositivo de memória que é comum a todos e no qual podem ler e escrever. Isso é ilustrado na Figura 2.1, onde os retângulos representam o espaço de memória dos nós dos sistema distribuído, os círculos representam o espaço de memória dos processos, a nuvem representa o espaço de tuplas e as figuras geométricas representam objetos de memória. Cada processo tem seu espaço de memória "estendido" para toda a nuvem e os objetos que ali se

encontram são comuns a todos os processos. O espaço cria essa "ilusão", que é muito conveniente. Mas, na realidade, ele dispõe os objetos de uma maneira bem distinta e gerencia todos os acessos de forma a manter a ilusão, como mostra a Figura 2.2.

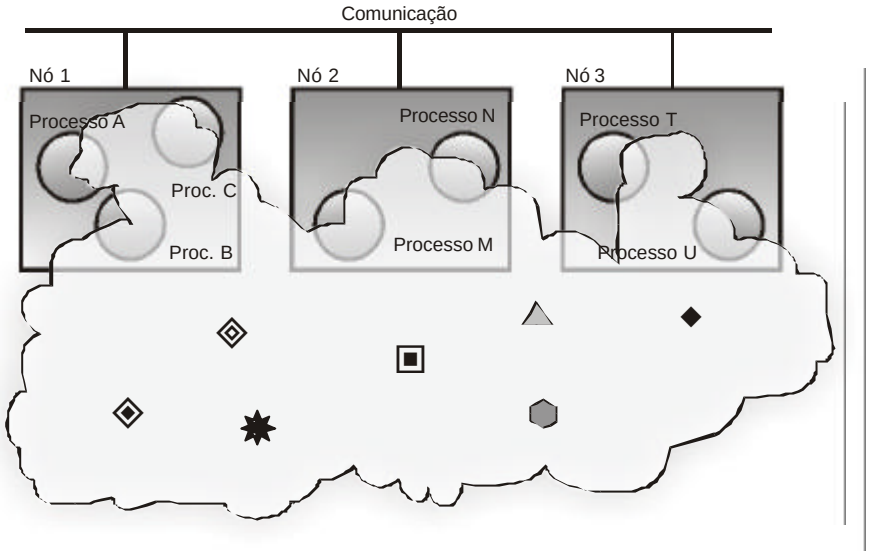

Figura 2.1 – O espaço de tuplas cria a "ilusão" de que existe um espaço de memória compartilhada entre os processos num sistema distribuído. Cada processo enxerga o seu próprio espaço de memória (o círculo) bem como todo o espaço de tuplas (a nuvem).

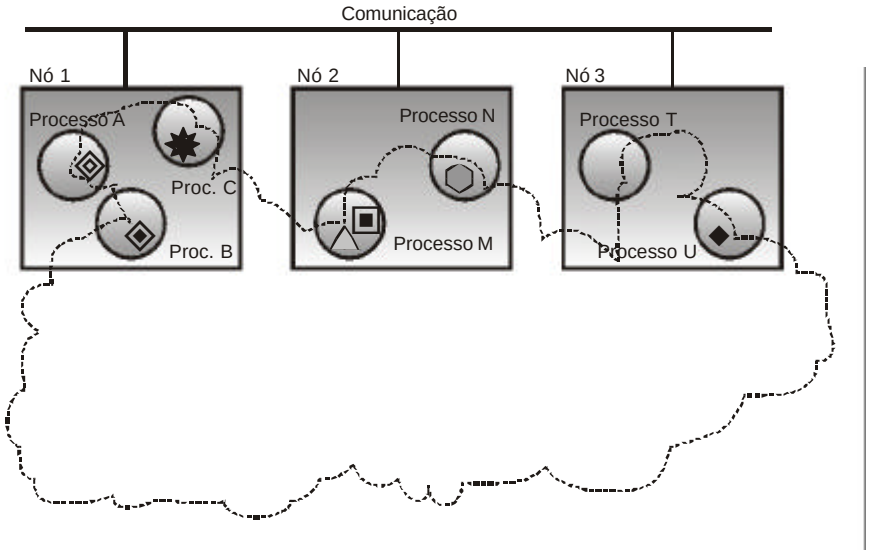

Figura 2.2 – Na realidade, os objetos de memória estão dispostos na memória dos nós do sistema distribuído. O espaço de tuplas mantém a "ilusão" de memória compartilhada movendo os objetos conforme as requisições de acesso (a disposição indicada na figura é apenas uma sugestão, pois há vários esquemas possíveis).

Os dados são armazenados num espaço de tuplas na forma de tuplas. O parâmetro de uma operação de escrita é uma tupla, assim como o retorno de uma operação de leitura. Tuplas são compostas de campos. Cada campo é de um tipo definido em alguma linguagem de programação associada ao espaço de tuplas. Cada campo possui ainda um valor e uma posição definida na seqüência de todos os campos que compõem a tupla. Isso quer dizer que, se a ordem dos campos da tupla for alterada, tem-se uma tupla diferente.

Um espaço de tuplas não possui endereços. Isso quer dizer que as tuplas armazenadas em um espaço de tuplas não podem ser recuperadas a partir de endereços. E, ao serem inseridas, elas jamais são inseridas em endereços. Tuplas são simplesmente escritas *no espaço*.

A recuperação da tupla se dá pelo conteúdo de seus campos. O parâmetro de uma operação de leitura é um subconjunto de campos e os valores esperados para cada campo desse subconjunto. Os demais campos podem assumir quaisquer valores. Como resultado da operação, o espaço retorna uma tupla que tem os valores nos campos do subconjunto exatamente iguais aos valores especificados. Caso mais de uma tupla atenda a esse critério, qualquer uma delas, mas apenas uma, será devolvida pelo espaço de tuplas como resultado da operação. Esse esquema de recuperação por conteúdo é conhecido como *memória associativa.*

Um caso especial é quando não existe, no espaço de tuplas, uma tupla com os valores especificados na operação de leitura. Um espaço de tuplas pode exibir dois comportamentos: manter bloqueado o processo que invocou a operação, até que uma tupla seja escrita, ou informar imediatamente ao processo de que a tupla não existe.

A semântica bloqueante se assemelha à semântica de semáforos e confere bastante expressividade ao modelo de espaço de tuplas. Graças a da, é possível sincronizar processos através das próprias tuplas, de uma forma muito simples.

A semântica não-bloqueante (também chamada *predicativa*) se justifica porque há casos em que o processo, que invocou a operação de leitura, precisa apenas determinar se a tupla está ou não no espaço. Ele não precisa esperar pela tupla.

Outro aspecto que diferencia as operações de leitura no modelo de espaço de tuplas é que há operações que removem a tupla do espaço de tuplas após a leitura e há operações que a mantêm.

2.3 Linda

Linda [Gelernter, 1985, Carriero e Gelernter, 1989b] é um espaço de tuplas constituído de um conjunto de seis operações e um sistema de *run-time* (que é composto por *daemons* em execução nos diversos nós do sistema).

As operações de Linda são usadas ao longo do código-fonte de um programa que se quer executar e que é escrito em alguma linguagem de programação associada ao Linda (mais comumente C ou Fortran). Um pré-compilador transforma as operações de Linda em instruções de mais baixo nível dirigidas para o sistema *run-time*. Essas instruções são escritas na linguagem associada e são inseridas diretamente no código-fonte. Durante a execução do programa, o sistema de *run-time* responde às requisições devidamente construídas pelo pré-processador.

O pré-compilador Linda possui, *a priori*, informações sobre os componentes, as características e a configuração do sistema distribuído sobre o qual o programa será executado. Desta forma, o pré-compilador pode otimizar alguns parâmetros de execução do sistema de *run-time*, como, por exemplo, a distribuição das tuplas. Portanto, Linda é um espaço de tuplas voltado para sistemas distribuídos fechados (sistemas conhecidos previamente)

2.4 Operações em Linda

Linda possui um conjunto de operações reduzido. São seis operações apenas, duas de escrita (*out* e *eval*) e quatro de leitura (*in, inp, rd* e *rdp*). Nesta seção, a apresentação das operações é acompanhada por exemplos escritos em C-Linda.

A operação *out* insere uma tupla no espaço de tuplas. A palavra *out* ("para fora", em inglês) é usada no sentido de que a tupla sai do processo que a gerou.

A operação *eval* também insere uma tupla no espaço de tuplas, mas, caso a tupla contenha campos com funções a serem avaliadas, a avaliação se inicia quando a tupla entra no espaço. *Eval* é uma abreviação para *evaluate* ("calcular", em inglês). A diferença entre *out* e *eval* fica mais clara se compararmos a ação de duas operações sobre uma mesma tupla, com um campo a ser avaliado.

```
out(2, f(1))
eval(2, f(1))
```

No primeiro caso, o próprio processo que invocou *out* é responsável por executar `f(1)` e preencher o campo com o resultado de retorno da função, antes de inserir a tupla no espaço. Ou seja, a tupla já entra no espaço com o segundo campo calculado. No segundo caso, o processo que invoca *eval* não calcula `f(1)`; a tupla entra no espaço com essa execução pendente, e então um novo processo é criado dentro do espaço para executar a função. A operação *eval* é o mecanismo de criação de processos em Linda.

Enquanto a tupla estiver sendo executada, diz-se que ela é uma *tupla ativa*, ou ainda, uma *tupla de processos*. Essas tuplas não podem ser lidas. Quando a execução termina, a tupla se transforma em uma *tupla passiva* ou *tupla de dados*.

A recuperação de tuplas em Linda se faz com gabaritos (em inglês, *templates*). É a forma encontrada para implementar a memória associativa. Enquanto que, numa tupla, os campos são todos *reais*, num gabarito os campos podem ser *reais* ou *formais*. Um campo formal é aquele que não contém dados, apenas seu tipo é indicado. Sua função é definir "a forma" do campo. Em geral, o campo formal está atrelado a uma variável. Campos reais são aqueles que apresentam forma e conteúdo. Tome-se como exemplo o gabarito

```
("exemplo", i, ?j, ?char*)
```

com quatro campos. O sinal de interrogação no início define um campo como formal. É o caso do terceiro e do quarto campos. Os dois primeiros campos são reais. O tipo do terceiro campo é definido pelo tipo da variável `j`. O quarto campo é do tipo *(char*)*.

Todas operações de leitura requerem um gabarito como parâmetro. O resultado da operação será uma tupla compatível com o gabarito. As regras para que um gabarito seja compatível com uma tupla são as seguintes:

1. A tupla e o gabarito devem ter o mesmo número de campos.
2. Cada campo na tupla e seu respectivo campo no gabarito devem ser do mesmo tipo.
3. Os conteúdos dos campos reais do gabarito devem ser iguais aos conteúdos dos respectivos campos na tupla.

A diferença entre as quatro operações de leitura se dá pela semântica bloqueante e pela permanência da tupla no espaço após a operação. A operação *in* bloqueia se não houver uma tupla compatível com o gabarito e remove a tupla do espaço quando termina. A operação *rd* também bloqueia, mas não remove a tupla do espaço (faz uma cópia). A operação *inp* não bloqueia, mas retira a tupla. Finalmente, a operação *rdp* não bloqueia, nem remove a tupla.

Figura 2.3 – Instância hipotética de um espaço de tuplas em um dado instante. As tuplas estão delimitadas por parênteses. Seus campos separados por vírgulas. As strings são delimitadas por aspas. Esse espaço parece estar servindo a várias aplicações, pela variedade dos tipos de tuplas.

Para exemplificar, considere-se o conjunto de tuplas da Figura 2.3, que representa o conteúdo de um certo espaço de tuplas em um dado instante. A operação `in("S")` tem como resultado a tupla `"S"`, que é apagada do espaço após a operação. Interessante notar que o gabarito, nesse caso, é composto apenas de campos reais. A operação `in("T")` ocasiona um bloqueio no processo que a executou porque não existe uma tupla dessa forma. Esse bloqueio perdura até que uma tupla `"T"` seja inserida no espaço. A operação `rd(A,1,1,?x)`, com a variável `x` sendo de tipo inteiro, faz a variável `x` receber o valor 103. A tupla lida não é apagada. A leitura `inp(?char*,"Cleópatra",?msg)` retorna imediatamente, informando, de alguma forma[2], que não há uma tupla compatível com esse gabarito. Para a

[2] Há uma controvérsia na literatura sobre o mecanismo usado pelas operações não-bloqueantes para informar que a tupla não existe no espaço. Alguns artigos referem-se a um ponteiro nulo, outros a um valor booleano FALSE.

operação `rd("Tarefa",?y)` sendo `y` de tipo inteiro, há quatro tuplas compatíveis com o gabarito. Qualquer uma delas pode ser o resultado. Por exemplo, `y` poderia valer 9 como resultado da operação.

2.5 Aplicações

O modelo de espaço de tuplas, embora simples, é muito flexível. Ao usuário é permitido construir livremente as tuplas para a sua aplicação, ou seja, ele tem total liberdade para definir quantos campos tem a tupla e quais são os tipos desses campos. Nesse "projeto da tupla" está a chave para uma aplicação bem sucedida.

Nesta seção, têm-se exemplos de aplicações dos espaços de tuplas em sincronização, estruturas de dados, programação paralela e comunicação entre processos. Para mais exemplos ver [Carriero e Gelernter, 1989a, Carriero e Gelernter, 1989b, Freeman et al, 1999]

2.5.1 Sincronização distribuída

A sincronização distribuída pode ser feita usando semáforos. Em Linda, um semáforo é uma coleção de tuplas idênticas. Na Figura 2.3, o semáforo S é o conjunto das tuplas com um campo "S". Para inicializar o semáforo com *n* elementos, repete-se a operação `out("S")` *n* vezes. Para decrementar o semáforo S, basta realizar a operação `in("S").` O estado do semáforo é determinado pelo número de tuplas no espaço. Se não houver nenhuma, o processo bloqueia ao tentar decrementar o semáforo. A semântica bloqueante da operação *in* coincide com a semântica bloqueante de semáforos. Pra incrementar o semáforo, o processo deve executar `out("S").`

2.5.2 Estruturas de dados

Uma matriz, ou outras estruturas de dados em que os elementos possuem uma posição definida dentro da estrutura, podem ser implementadas em Linda acrescentando-se campos relativos à posição. Um elemento da estrutura é representado por uma tupla da forma `(nome da estrutura, campos índices, valor)`. Na Figura 2.3, existe uma matriz 2x2, chamada A. Para alterar o elemento na *i*-ésima linha e *j*-ésima coluna, os comandos seriam

```
in(A, i, j, ?ValorAntigo)
Calcula o NovoValor
out(A, i, j, NovoValor)
```

Cestas (*bags*, em inglês) são estruturas de dados com duas operações definidas: "inserir um elemento" e "remover um elemento". Os elementos não precisam ser idênticos, mas são tratados de forma indistinta. Na Figura 2.3, há uma cesta de tarefas e uma cesta de resultados, representadas pelas tuplas `("Tarefa",x)` e `("Resultado",y)` respectivamente.

2.5.3 Programação paralela

O modelo de programação paralela mais popular entre os usuários de espaços de tuplas chama-se "operários replicados" e está esquematizado na Figura 2.4. Consiste de um processo mestre e diversos processos operários, que interagem entre si através do espaço de tuplas. O processo mestre divide uma tarefa em tarefas menores e as distribui para os operários. Estes executam as tarefas menores, encontrando um resultado, que é devolvido ao mestre, que, tendo recolhido todos os resultados, monta a resposta final.

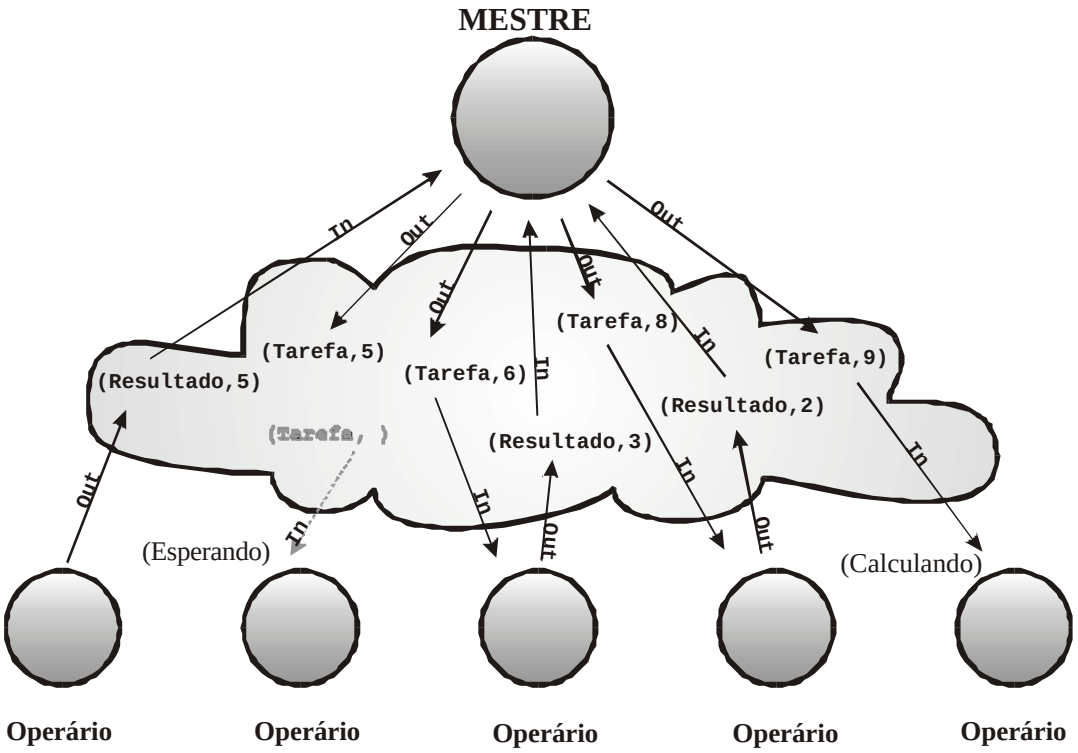

Figura 2.4 – Programação paralela através do modelo de operários replicados. Um processo mestre distribui tarefas para processos operários, que as resolvem e retornam os resultados. A coordenação entre mestre e operários é feita através de um espaço de tuplas.

A coordenação entre mestre e operários é feita com cestas de tarefas e resultados. Isso fica embutido de forma simples e elegante no código. A forma geral do código do operário é:

```
operário() {
     do {
          in("Tarefa", ? a_fazer);
          executa a_fazer e obtém feito
          out("Resultado", feito);
     } while(condição)
}
```

Inicialmente, o operário procura uma tarefa na cesta de tarefas. Se não existir tarefa, ele fica bloqueado esperando, conforme pode-se ver na Figura 2.4. De posse da tarefa, ele a executa por um certo tempo. O resultado é colocado em uma cesta própria para resultados. O operário está pronto para procurar uma outra tarefa.

Pelo lado do mestre, o algoritmo é conceitualmente simples. Ele começa despachando diversas tarefas para a cesta, através da operação `out("Tarefa", Descrição)`e, em seguida, espera pelos resultados com a operação `in("Resultado", ?result)`.

A aplicabilidade desse modelo na resolução de problemas envolvendo paralelismo é evidente [Carriero e Gelernter, 1986]. Diz-se que esse modelo possui um *balanceamento de carga intrínseco*, pois cada operário requisita tarefas na medida de sua disponibilidade. Outros modelos de programação paralela com espaços de tuplas podem ser encontrados em [Carriero e Gelernter, 1989a].

2.5.4 Comunicação entre processos

Uma mensagem escrita no espaço de tuplas na forma de uma tupla com um só campo (campo que contém o corpo da mensagem) não possui destinatário nem remetente. É impossível determinar sequer quem a escreveu. Como, então, realizar a comunicação entre processos? Esta tarefa torna-se simples com a devida construção da tupla. A tupla deve ter, pelo menos, um campo com o endereço do processo destinatário e um campo com a mensagem. O processo pode recuperar as mensagens destinadas a ele, retirando do espaço as tuplas que contenham seu endereço naquele campo. Para o caso da tupla possuir apenas o campo de destinatário e o campo da mensagem, nessa ordem, as tuplas seriam retiradas com a operação `in(endereço, ?mensagem)`.

É possível adicionar um campo para o remetente. Isso permite identificar quem enviou a mensagem. Pode-se pensar ainda em criar canais, nos quais diversos processos trocariam informações, como nos canais de bate-papo da Internet. Seria mais um campo na tupla. A forma da tupla seria então

```
(remetente, destinatário, canal, mensagem).
```

Esse é apenas um dos esquemas possíveis. É intuitivo pensar em mecanismos de *multicasting* e *broadcasting* nesse esquema.

2.6 Desacoplamento

Como foi dito na Seção 2.5.4, em geral, é impossível determinar quem colocou uma dada tupla no espaço de tuplas e quem irá retirá-la. O processo que cria a tupla desconhece o processo que irá consumi-la. Por isso, diz-se que eles estão *desacoplados*. Eles podem nem mesmo estar no mesmo espaço de memória dentro do sistema distribuído (desacoplamento espacial). E mais: podem nem coexistir no mesmo instante de tempo (desacoplamento temporal).

Essa propriedade distingue o modelo de espaço de tuplas do modelo de passagem de mensagens. Embora similares, eles não são idênticos. No modelo de espaço de tuplas, a mensagem tem existência própria, independentemente dos processos que se comunicam através dela. Gelernter (1985) chamou isso de *comunicação geradora (generative communication)*, no sentido de que a comunicação *gera* uma terceira entidade independente. Ao criar essa terminologia, Gelernter queria salientar essa diferença conceitual do seu modelo com o modelo de passagem de mensagens.

A propriedade de desacoplamento é a principal responsável pelo interesse em torno do modelo de espaço de tuplas. Na teoria de projeto de software é extremamente desejável alcançar alto desacoplamento entre os módulos do projeto. Assim, os módulos podem ser trocados ou alterados de forma independente. Em projetos grandes, essa característica é indispensável.

Veja-se, por exemplo, o modelo de operários replicados discutido na Seção 2.5.3. Um sistema que adota esse modelo é relativamente fácil de gerenciar, porque o número de operários não está amarrado a nenhum elemento do sistema. O mestre desconhece quantos e quais são os operários. Assim, o número de operários no sistema pode ser alterado à vontade.

Quando o sistema está sobrecarregado, mais nós de processamento podem ser adicionados, mesmo em tempo de execução, sem comprometer o andamento do trabalho. O sistema passa naturalmente para uma escala maior. E há uma maior tolerância a falhas. Se um nó entrar em pane, sua falta não será sentida, pois nenhum elemento do sistema dependia da informação de quem era aquele nó. Os demais nós prosseguem funcionando normalmente.

> "Comunicação, exclusão mútua, sincronização e gerenciamento de memória do espaço de tuplas são responsabilidade do espaço de tuplas e, portanto, transparentes para o programador. Em outras palavras, todos os processos são dispensados da responsabilidade de saber nomes, endereços, variáveis e estado temporal de qualquer outro processo." [Cannon e Dunn, 1994]

O nível de desacoplamento alcançado com um espaço de tuplas é o que o diferencia de outros mecanismos populares de programação distribuída, tais como chamada remota a procedimentos (*Remote Procedure Call* ou RPC) e filas de mensagens (*Message queues*). O mecanismo de RPC é uma extensão, para o escopo de um sistema distribuído, da semântica de chamada a procedimento local (Seção 4.4). A evolução da RPC são as plataformas para objetos distribuídos como CORBA e DCOM. Filas de mensagem, em sua origem, são mecanismos de comunicação interprocessos internas a um sistema operacional. Um processo pode criar uma fila em que outros processos poderão ler ou escrever mensagens. Esta idéia foi estendida para sistemas distribuídos.

Rogue Wave (2001) faz uma interessante comparação entre os mecanismos, tendo por base o nível de acoplamento em três aspectos: espaço de processo, tempo e canal, como ilustra a Figura 2.5. No aspecto do tempo, o acoplamento mais fraco é assíncrono, isto é, os processos não precisam se comunicar simultaneamente. No aspecto de canal, o acoplamento mais fraco é quando cada processo não precisa conhecer a identidade ou o endereço do outro. E no aspecto espaço de processo, o maior acoplamento é uma chamada a procedimento, que ocorre no mesmo espaço de processo e não permite qualquer flexibilidade em termos de espaço de endereçamento, linguagens e representações heterogêneas.

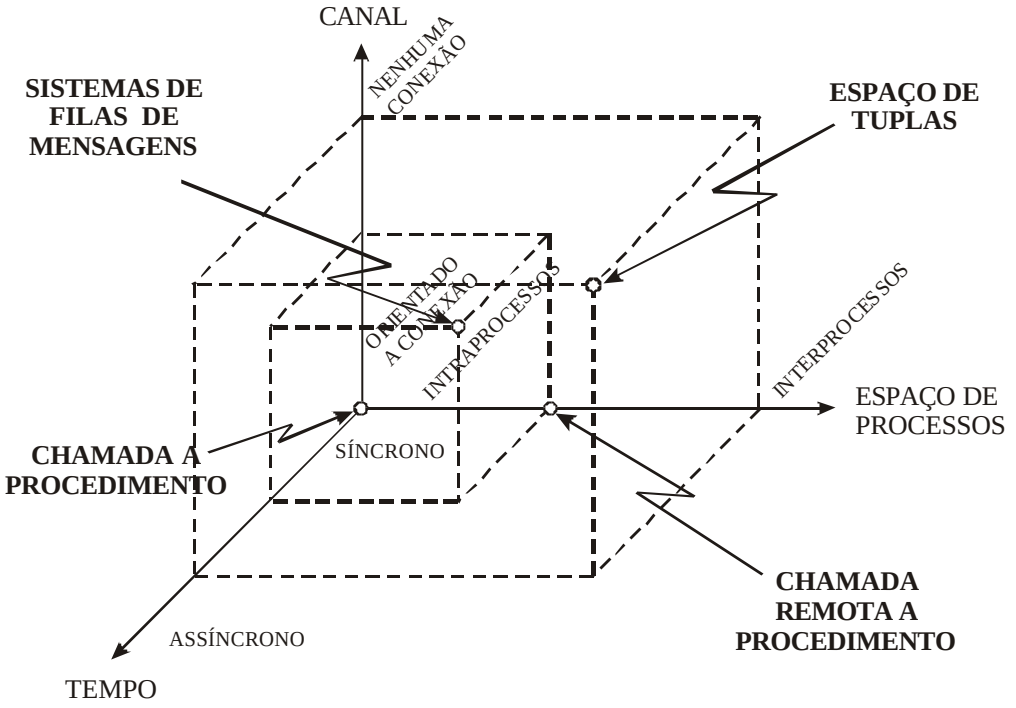

Figura 2.5 [Rogue Wave, 2001] – Comparação entre mecanismos tendo por base o grau de acoplamento. O acoplamento é diferenciado em três aspectos: tempo, espaço de processos e canal. Quanto mais próximo da origem, o acoplamento é maior.

A RPC e seus derivados orientados a objetos, como o CORBA e o DCOM, representam um avanço em termos de desacoplamento no espaço de processos, visto que eles facilitam a construção de sistemas distribuídos. Com o uso de serviços de nomes, o desacoplamento é ainda maior, pois introduzem um nível de indireção entre os endereços das aplicações clientes e servidoras. No entanto, sistemas baseados em RPC, são ainda orientados a conexão e, em geral, síncronos.

Os sistemas de filas de mensagens estão um passo adiante no sentido de um menor acoplamento nos três aspectos analisados. Eles permitem que os processos se comuniquem sem um canal ativo entre os dois. Eles oferecem ainda mecanismos de comunicação interprocessos e mecanismos não-bloqueantes para lidar com atrasos de comunicação.

2.7 Implementação

O modelo de espaço de tuplas é conceitualmente simples e elegante. No entanto, a dúvida quanto à viabilidade de uma implementação com desempenho satisfatório foi a maior objeção que a comunidade científica apresentou inicialmente e que restringiu a disseminação

do modelo. A busca por uma implementação convincente foi o centro dos esforços, nos anos iniciais, dos pesquisadores interessados no modelo.

Há dois aspectos preponderantes no desempenho de uma implementação de espaço de tuplas. O primeiro é a memória associativa. Uma vez que a recuperação dos dados é feita por conteúdo, algum algoritmo de busca tem de ser adotado. Com um volume massivo de dados, o tempo consumido numa operação de leitura pode ser sensível. O segundo aspecto é a distribuição das tuplas no sistema distribuído. Uma boa distribuição minimiza as (custosas) transferências de tuplas entre nós causadas pelas operações de escrita e leitura.

Nos anos 1980, espaços de tuplas (quase sempre Linda) foram implementados nas mais distintas plataformas de hardware. A implementação sobre a rede S/Net conseguiu um desempenho aceitável [Carriero e Gelernter, 1986]. Em outro trabalho, até um coprocessador de hardware próprio para espaço de tuplas foi proposto [Ahuja et al., 1988]. *Meshs* de *transputers* [Faasen, 1991] e redes de *workstations* [Bjornson, 1992] também merecem ser citados.

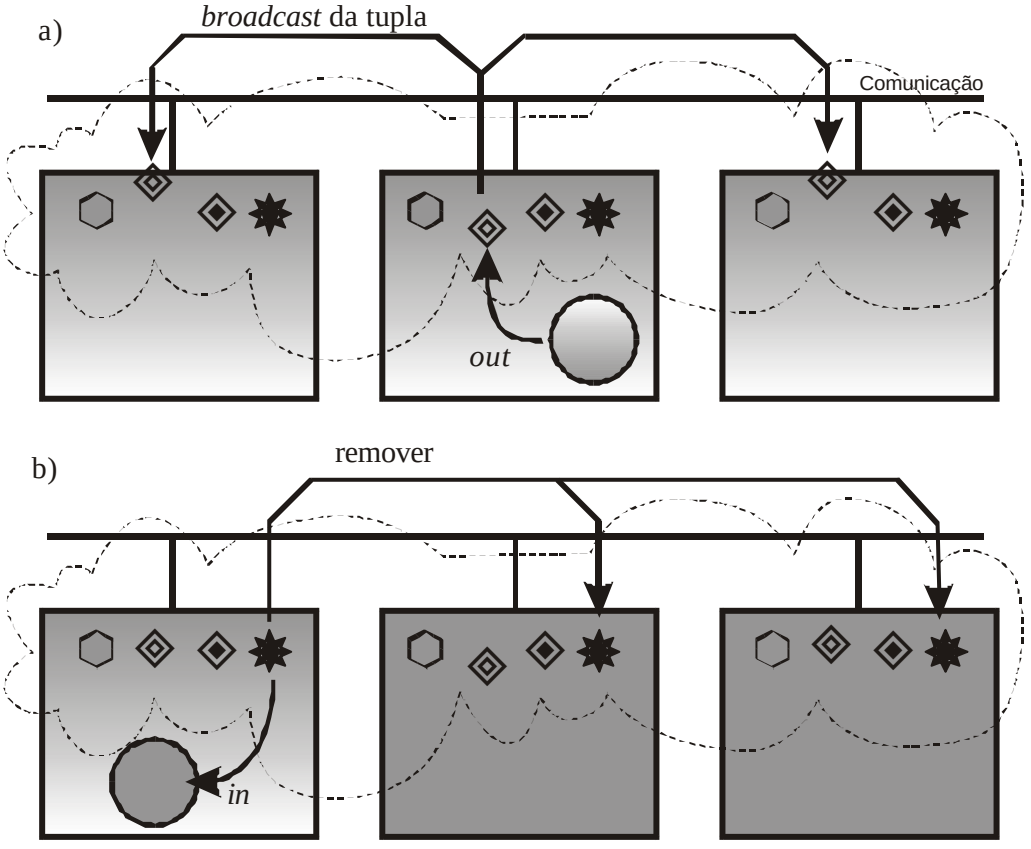

Figura 2.6 – Protocolo com replicação. A operação out é transmitida para todos os nós. A operação in é local, mas a remoção é transmitida para todos os nós.

Nos anos 1990, os esforços se focaram em implementar espaços de tuplas sobre sistemas distribuídos. A implementação de referência dessa fase foi proposta por Bjornson (1992), do grupo de Yale. Ela é a base para a implementação comercial de Linda que se encontra atualmente no mercado [SCA, 2002].

A busca por um desempenho aceitável sempre foi o maior objetivo das implementações. Para isso, inevitavelmente, recorria-se a algum esquema de distribuição. Um exemplo clássico é a implementação sobre a rede S/Net. Ela se valia de uma característica notável desse tipo de rede – o suporte a *broadcasting* confiável. Dois protocolos diferentes foram implementados, que são apresentados a seguir.

O primeiro protocolo (Figura 2.6) consiste em transmitir para todos os nós da rede uma tupla que se pretende escrever no espaço de tuplas. A escrita foi disparada através de uma operação *out* executada em um dos nós. Esse protocolo cria uma cópia integral do espaço de tuplas em cada nó (replicação total). A operação *rd* pode ser feita no próprio nó, sem gerar mensagens na rede. A operação *in*, no entanto, deve remover a tupla de todos os nós. Se dois nós tentarem remover a mesma tupla ao mesmo tempo, o protocolo garante que apenas um deles conseguirá. Graças a isso, a operação *in* torna-se cara e complicada.

O segundo protocolo (Figura 2.7) consiste em gravar a tupla apenas no próprio nó em que foi executada a operação *out*. Esse protocolo não gera replicação das tuplas. Pelo contrário, ocorre uma partição do espaço de tuplas, que fica espalhado. Numa operação de leitura, o gabarito precisa ser transmitido por *broadcasting* para todos os nós. Se algum deles tiver uma tupla compatível, ele a transmite para o interessado. Se mais de um nó encontrar uma tupla, o algoritmo garante que apenas um deles vai conseguir retorná-la. Se nenhum nó encontrar uma tupla, todos os nós passam a comparar cada tupla gravada com o gabarito que receberam. Esse protocolo se mostrou mais eficiente que o primeiro.

Um esquema intermediário é arranjar os nós do sistema em forma de uma matriz bidimensional como proposto por Faasen (1991) (Figura 2.8). Quando acontece uma operação *out* em um nó (A), a tupla é enviada para todos os nós na mesma linha. Esse é um esquema de replicação parcial. Todos os nós da mesma linha possuem uma cópia idêntica de uma partição do espaço de tuplas. O espaço de tuplas é a união dessas partições. Quando acontece uma operação *in* em outro nó (B), o gabarito é enviado para todas as máquinas na mesma coluna. Como o gabarito chegará a todas as linhas (mas não a todas as máquinas), ele será comparado com todas a tuplas. Em alguma máquina (C), a tupla será encontrada.

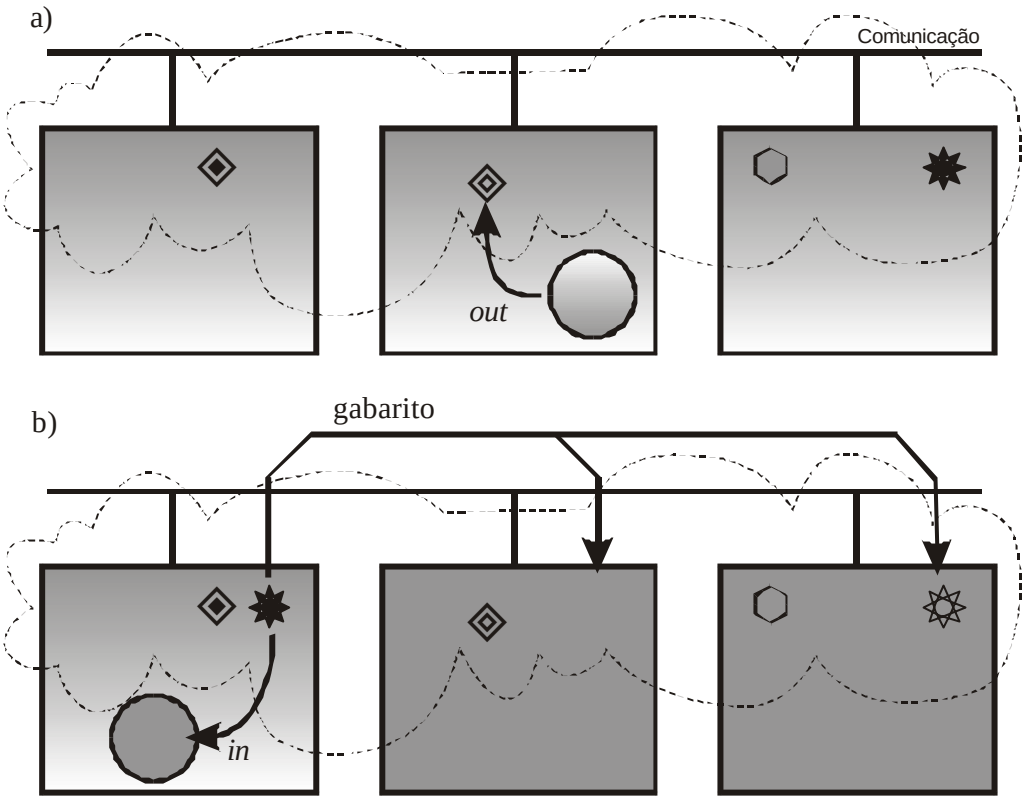

Figura 2.7 – Protocolo sem replicação. A operação out tem efeito local. Numa operação de leitura, o gabarito precisa ser transmitido para todos os nós por broadcasting.

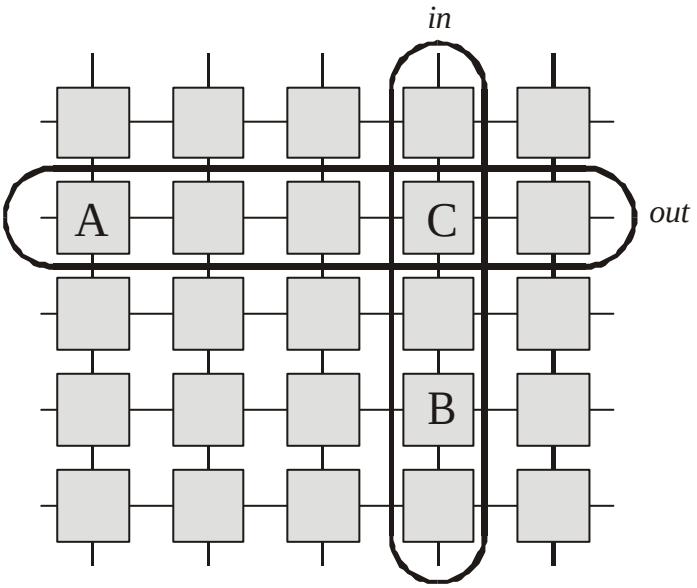

Figura 2.8 – Protocolo com replicação parcial. O espaço de tuplas é dividido em partições, uma em cada linha. Cada nó da linha contém uma cópia da partição.

Em uma arquitetura centralizada, as tuplas são mantidas em um único nó do sistema (Figura 2.9). Todas as operações requisitadas ao espaço de tuplas em outros nós do sistema requerem a comunicação com o nó central. Essa arquitetura tem desvantagens evidentes.

Primeiro, o nó central pode vir a ser um gargalo para o desempenho do sistema se o número de requisições crescer demasiadamente. Segundo, se o nó central tiver uma pane, todo o sistema pára de funcionar.

Carreira et al. (1994) afirmam que o desempenho de arquiteturas centralizadas é aceitável em sistemas pequenos ou médios, da ordem de dezenas de máquinas.

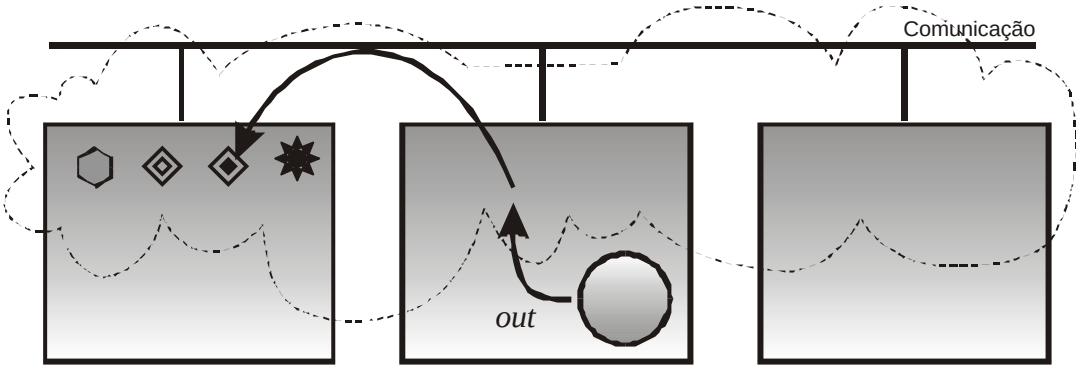

Figura 2.9 – Arquitetura centralizada. As tuplas são mantidas num único nó.

2.8 Considerações finais

O modelo de espaço de tuplas nunca foi uma solução popular. Talvez pese para isso o fato de que, por muitos anos, não tenha existido uma distribuição gratuita do modelo. Ou que o desempenho menor – ainda que na mesma ordem de grandeza – em comparação com as melhores tecnologias tenha desencorajado usuários.

> "Embora não represente a solução mais eficiente em problemas com granulosidade fina e redes com tráfico denso de mensagens, as pesquisas têm mostrado que, quando considerados o desenvolvimento e verificação do software, juntamente com a velocidade de execução, a abordagem de espaços de tuplas para o desenvolvimento de sistemas distribuídos é eficiente e de bom custo-benefício para uma variedade de problemas e redes". [Cannon e Dunn, 1994]

Talvez seja difícil identificar os propósitos e aplicações potenciais do modelo. Na literatura, Linda e o modelo de espaço de tuplas aparecem nas categorias de software mais distintas. Há quem os considere como um mecanismo de memória compartilhada distribuída [Tanenbaum, 1995] (embora pesquisadores da área nem os citem), ambiente de passagem de mensagens para programação paralela [Almasi e Gottlieb, 1994], *middleware* [Tanenbaum, 2000, Wyckoff et all., 1998], linguagem de coordenação (Seção 3.2), quadro-negro (*blackboard*), etc.

Em resumo, o modelo de espaço de tuplas é um mecanismo com semântica de memória compartilhada que pode ser acessado concorrentemente por processos espalhados em um sistema distribuído. O modelo implementa um esquema de memória associativa. Duas propriedades do modelo de espaço de tuplas destacam-se: a simplicidade e o baixo acoplamento entre os processos. Juntas, elas dão ao modelo uma vantagem em termos de facilidade de desenvolvimento e manutenção das aplicações.

Capítulo 3

Variações do modelo de espaço de tuplas original

3.1 Considerações iniciais

A partir de Linda, o modelo de espaço de tuplas se difundiu. Usuários, pesquisadores e desenvolvedores começaram então a criar novos espaços de tuplas, quase sempre expandindo em algum sentido as funcionalidades básicas de Linda. Conhecer essas variações no modelo original ajuda a compreender melhor o próprio modelo e suas limitações. E é imprescindível a quem se habilita a implementar um espaço de tuplas.

Neste capítulo, vários espaços de tuplas são passados em revista, enfocando-se as funcionalidades com que contribuíram para o modelo de espaço de tuplas. Na Seção 3.2, é contado um breve histórico dos espaços de tuplas que são abordados nas seções seguintes. As variações consideradas nesse capítulo são: o modelo de *múltiplos* espaços de tuplas (Seção 3.3), novas operações (Seção 3.4), mecanismos de tolerância a falhas (Seção 3.5), persistência (Seção 3.6) e segurança (Seção 3.6).

3.2 Breve histórico dos espaços de tuplas

O modelo de espaço de tuplas foi proposto pela primeira vez por David Gelernter como conclusão de seu trabalho de doutorado [Gelernter e Bernstein, 1982]. Nesse tempo, ainda não havia as operações não-bloqueantes, que foram introduzidas posteriormente, completando o modelo tal como é conhecido hoje. Gelernter passou a integrar o corpo docente da Universidade de Yale e dedicou-se a desenvolver uma implementação de Linda que apresentasse um desempenho satisfatório juntamente com alunos e pesquisadores de outras instituições. Essa linha de pesquisa culminou com uma tese de doutorado que propunha uma estratégia de implementação sobre sistemas distribuídos [Bjornson, 1992].

A primeira proposição no sentido de uma mudança mais radical no modelo de espaço de tuplas foi o conceito de múltiplos espaços de tuplas [Gelernter, 1989]. A partir de então, poder-se-ia criar espaços de tuplas através de uma operação adicional. E esses espaços poderiam ser incluídos dentro de outros espaços. O primeiro espaço de tuplas a implementar esse modelo foi Linda 3.

Persistent Linda (ou simplesmente P-Linda) [Anderson e Shasha, 1991] é talvez o espaço de tuplas que mais tenha contribuído com idéias inovadoras. Fruto do trabalho de pesquisadores da área de banco de dados, P-Linda foi o primeiro espaço de tuplas a incorporar as idéias de persistência e semântica transacional ao modelo básico de espaço de tuplas. As consultas em P-Linda são mais próximas do modo usado em bancos de dados, permitindo outras formas de comparação entre campos além da mera igualdade. P-Linda é também o primeiro a propor novas operações para o modelo. As operações propostas têm dois objetivos: operar sobre conjuntos de tuplas e combinar operações que pudessem ser executadas conjuntamente, para melhorar o desempenho.

O grupo de pesquisa da Universidade de York, na Inglaterra, liderado pelo professor Alan Wood, é um dos mais dedicados grupos de pesquisa sobre espaços de tuplas. Dali saíram várias proposições de alteração no modelo, que ficaram conhecidas como Linda de York. Talvez nenhum outro grupo tenha questionado tão incisivamente os fundamentos teóricos do modelo de espaço de tuplas. Em decorrência do trabalho do grupo de York, hoje se sabe que as operações de Linda não são suficientes para resolver certos problemas de computação distribuída. Para contornar essa limitação, novas operações foram propostas pelo grupo de York, mas não foram incorporadas pelo grupo de Yale.

Carriero e Gelernter (1989b) é a mais polêmica publicação sobre Linda. Nela, os autores comparam Linda com outros modelos de programação concorrente populares à época, afirmando ser Linda a melhor escolha entre eles. A reação veio na seção de cartas do periódico [CORPORATE, 1989]. Os proponentes dos outros modelos argumentavam que Linda não é uma linguagem de programação independente e precisa ser acoplada a uma outra linguagem. A tréplica [Carriero e Gelernter, 1992] consistiu em separar computação de coordenação como duas funções independentes (ortogonais, na linguagem do artigo). Dessa forma, Linda seria uma linguagem de coordenação completa. A computação ficaria por conta de outras linguagens. Coordenação era um termo emergente à época. Estava muito em evidência a "Teoria da Coordenação" [Malone e Crowston, 1992], que congregava

pesquisadores de áreas tão distintas quanto ciência da computação, teoria da organização, pesquisa operacional, economia, lingüística e psicologia. O que há em comum entre todas é que elas precisam gerenciar dependências entre atividades de alguma forma; e isso é coordenação. A partir de então, Linda ganhou a alcunha de linguagem de coordenação. Muitos espaços de tuplas surgiram neste contexto. Até um congresso internacional próprio foi criado [COORDINATION, 1996] (o congresso teve edições em 1996, 1997, 1999, 2000, 2002). Artigos sobre espaços de tuplas têm grande destaque nestes eventos.

O interesse pelo modelo de espaço de tuplas parecia fechado a esse nicho, quando surgiram os espaços de tuplas em Java e projetaram o modelo para outros meios. O primeiro a aparecer foi JavaSpaces [Waldo, 1996] [Sun Microsystems, 1999a], que é um espaço de tuplas para objetos em Java, com suporte a transações, *leasing* (um mecanismo de automanutenção, Seção 4.2.1) e persistência. JavaSpaces apresenta também uma operação adicional para recuperação de tuplas baseada na notificação de eventos. JavaSpaces é apresentado em detalhes no capítulo 4.

T Spaces [Wyckoff et all, 1998] é outro espaço de tuplas em Java muito popular e foi proposto e desenvolvido pela IBM, que tem tradição de pesquisa em bancos de dados. Na visão de seus proponentes, T Spaces é um *middleware* ideal para aplicações distribuídas envolvendo dispositivos móveis com pequena capacidade computacional, como celulares e *Personal Digital Assistents*. Algumas características notáveis de T Spaces são: mais modalidades de consulta além da igualdade, persistência, controle de acesso para fins de segurança e notificação de eventos.

Lime (Linda in a Mobile Environment) [Picco et al, 1998] é um espaço de tuplas voltado para agentes e computação móvel. Esse é um nicho de aplicação propício para os espaços de tuplas. Lime mostra como o conceito básico de espaço de tuplas pode ser adaptado criativamente para aplicações com demandas específicas. Em Lime, cada agente possui espaços de tuplas próprios, que ele leva consigo a tiracolo, chamados ITS (*interface tuple space*). Quando o agente se fixa em um *host*, seus ITSs são fundidos a um espaço de tuplas local e as tuplas passam a ficar visíveis para os demais agentes acoplados ao *host* (Figura 3.1). Ao deixar o host, o agente leva consigo os ITSs com as tuplas.

Globe (*Global Object Exchange*) [Larsen e Spring, 1999] é um espaço de tuplas que usa replicação para atingir níveis maiores de tolerância a falhas e desempenho. Cada réplica de Globe é implementada com um JavaSpaces. Globe é um trabalho acadêmico e seu valor

está na discussão detalhada dos problemas que se pode enfrentar para implementar um espaço de tuplas com replicação.

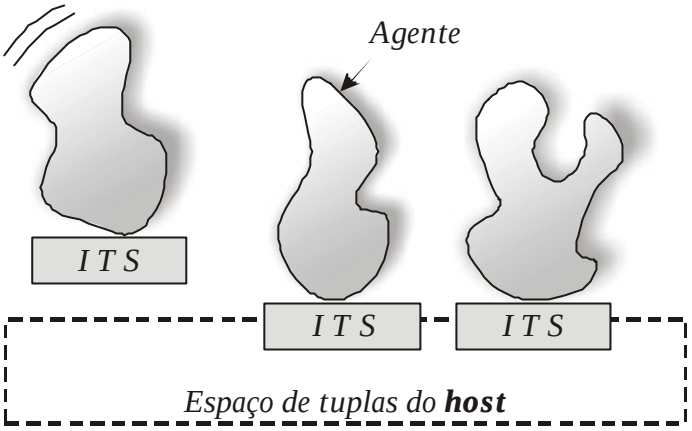

Figura 3.1 – Espaço de tuplas Lime. Os agentes carregam espaços de tuplas próprios. Reproduzido de [Picco et al., 1998]

Ruple [Rogue Wave, 2001] é um espaço de tuplas voltado para a troca de documentos pela Internet. A grande inovação de Ruple é a definição de tupla, que não possui campos como uma tupla de Linda. As tuplas de Ruple são documentos em XML. No entanto, a recuperação das tuplas permanece sendo feita por conteúdo, usando consultas expressas na linguagem XQL. Outras características interessantes são: o protocolo SOAP para comunicação por rede, o uso de *leasing* para automanutenção e certificação X.509 para segurança.

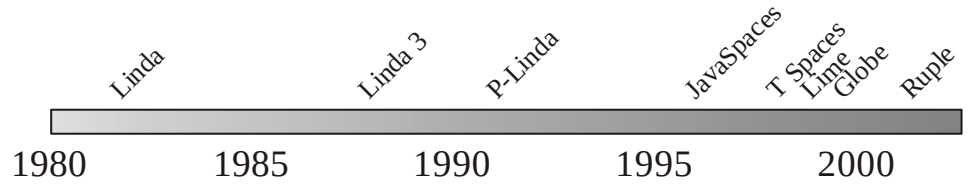

Figura 3.2 – Linha do tempo dos principais espaços de tuplas abordados neste capítulo.

3.3 Múltiplos espaços de tuplas

No modelo de espaços de tuplas original, o espaço de tuplas era monolítico. O modelo não apresentava mecanismos que permitissem modularizar o projeto de um aplicativo. Embora isso fosse possível de alguma forma através de cestas (Seção 2.5.2), esta necessidade levou a criação do conceito de múltiplos espaços de tuplas [Gelernter, 1989].

Dois ingredientes foram acrescentados a Linda para dar suporte ao modelo de múltiplos estados de tuplas: um novo tipo de variável, denotado TS (sigla de *tuple space*) e uma nova operação *tsc* (sigla de *tuple space create*).

Sem entrar em detalhes da semântica desta operação,

> "*tsc* retorna um *handle* para um espaço de tuplas que pode ser armazenado numa variável do tipo TS. Esse *handle* pode ser colocado em outros espaços de tuplas e, assim, ser passado para outros processos. As demais operações em Linda devem ser precedidas pela variável do tipo TS para especificar a qual espaço de tuplas elas se referem, por exemplo:
>
> ```
> TS  tsNew = tsc();
> tsNew.out("Hello");
> tsNew.out("foo",5);
> tsNew.in("foo",int ?);
> ```
>
> Se chamadas sem um espaço de tupla específico, as operações em Linda se referem a um espaço de tuplas local." [Butcher et al., 1994]

3.4 Conjunto de operações

Há um debate em torno do conjunto de operações básicas do modelo de espaço de tuplas e de Linda.

A primeira indagação é se esse conjunto é mínimo ou se há operações redundantes.

> "Note que *rd* e *rdp* são desnecessárias no sentido de que elas podem ser implementadas em termos de *in* e *inp* respectivamente (por exemplo, *rd* poderia ser implementada em termos de um par *in/out*). Embora *rd* e *rdp* melhorem a eficiência, eles não acrescentam nada ao poder de expressão do modelo." [Butcher et al., 1994]

As operações predicativas também são questionadas por Jacob e Wood (2000).

> "Parece que há uma necessidade profunda, por parte de programadores, dessas operações – é como se houvesse algo "errado" em escrever um código que pode bloquear para sempre. Esse imperativo psicológico deve ser conseqüência do fato de que programadores aprendem seu ofício, infelizmente, em um ambiente seqüencial e não conseguem lidar intuitivamente com processos concorrentes".

Os autores postulam que as operações *inp* e *rdp* deveriam ser evitadas e mostram que as situações onde elas poderiam ser desejadas podem ser contornadas com as operações bloqueantes e um bom suporte a *multithreading*. Em seu trabalho, eles analisam as dificuldades de implementação dessas operações e chegam a uma conclusão surpreendente:

– "*inp* não é uma versão não-bloqueante de *in* – ela bloqueia, como *in*, até que uma tupla compatível seja recuperada ou até que um *deadlock* seja detectado.

– *inp* é uma operação cara – é sensivelmente mais cara (em termos de carga do sistema *run-time*) que um *in* que recupera a tupla, visto que o *run-time* também deve fazer detecção de *deadlock* quando necessário. Os efeitos dessa operação cara são sentidos por todo o sistema. De alguma forma, as operações predicativas exigem uma sincronização global do sistema". [Jacob e Wood, 2000]

A segunda indagação, essa mais grave, é quanto à precisão da definição semântica de algumas operações, como *eval*, *inp*, *rdp* e *tsc*. Jacob e Wood (2000) atacam a definição das operações predicativas (não-bloqueantes).

> "O significado informal de *inp* é: 'dê-me uma tupla compatível com o gabarito, ou diga-me se você não pode'. O problema surge com a expressão 'não pode'. Podem existir várias interpretações para isso, variando da ´comprovadamente não vai ser possível para todos os possíveis processos futuros' até 'não estou disposto a fornecer um tupla no momento'. A primeira forma é não-computável em sistemas abertos e a segunda de nada serve".

Os autores criticam a informalidade das definições semânticas destas operações presentes na literatura e nos manuais. Como resposta, eles apresentam uma definição algébrica "útil e implementável".

A terceira indagação, de certa forma relacionada com a primeira, mas numa direção contrária, é se o conjunto das operações é suficiente para resolver qualquer problema e, em não sendo, quais operações deveriam ser adicionadas ao modelo.

Butcher et al (1994) discutem o problema da soma distribuída:

> "A partir de n valores $\{x_1, ..., x_n\}$, deseja-se encontrar sua soma, isto é, $x_1 + ... + x_n$. As adições devem ser executadas em paralelo, preferencialmente extraindo o máximo nível possível de paralelismo.
>
> Um algoritmo bem conhecido que resolve este problema consiste em formar uma árvore binária de processos, cada qual tomando dois números como entradas e produzindo uma saída (a soma das entradas).[...] O cálculo termina quando o processo raiz termina".

A estrutura desta solução é mostrada na Figura 3.3.

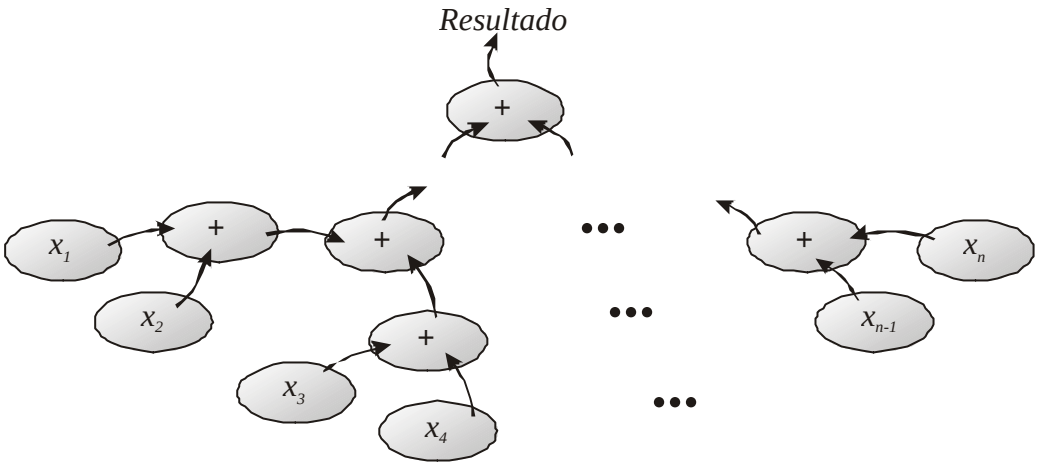

Figura 3.3 – O problema da soma distribuída.

Reproduzido de [Butcher et al., 1994]

A solução em Linda para este problema num primeiro momento poderia ser:

```
void operário (void) {
      int x,y;
      while(true) {
            in(?x);
            in(?y);
            out(x+y);
      }
}
```

Mas essa solução possui dois erros fatais: ela não apresenta um meio de detectar que o cálculo terminou e ela pode levar a um *deadlock*. Butcher et al. (1994) discutem várias soluções alternativas e concluem que o problema deveria ser resolvido com a criação de uma nova operação chamada *collect*. A sintaxe da operação é :

```
int collect(TS destino, <gabarito>)
```

e a semântica informal : todas as tuplas compatíveis com o gabarito são movidos para o espaço de tuplas destino e o número de tuplas movidas é o retorno da operação.

Rowstron e Wood (1996) propõem uma operação parecida, chamada *copy-collect* (a diferença para o *collect* é que ela copia e não remove as tuplas do espaço) para resolver um outro problema: o dos múltiplos *rd*. Esse problema acontece em algoritmos em que a operação *rd* insiste em retornar a mesma tupla como resultado (o que não acontece com *in* porque ele remove a tupla). Embora existam soluções com as operações já existentes em Linda, *copy-collect* é sugerida a título de desempenho.

Assim como as operações *collect* e *copy-collect* de York, muitos espaços de tuplas previram a criação de operações que atuam sobre conjuntos de tuplas, tais como P-Linda (com *rd all, in all, rdp all, inp all*), T Spaces (com *scan* e *consumescan*), GigaSpaces (com *writeMultiple, readMultiple* e *takeMultiple*) e Ruple (com *readMultiple*).

3.5 Tolerância a falhas

A tolerância a falhas no modelo de espaço de tuplas tem sido abordada de várias formas. Num extremo, deixa-se a tolerância a falhas inteiramente sob a incumbência do sistema de *run-time* do espaço de tuplas, tornando-a transparente para o programador. No outro extremo, os mecanismos de tolerância a falhas são explicitamente embutidos na semântica das operações e cabe ao programador usá-los (obviamente, o sistema de *run-time* tem que dar suporte a estes mecanismos).

Um exemplo da primeira abordagem é FT-Linda [Bakken e Schlichting, 1995]. Para implementar tolerância a falhas transparente, FT-Linda usa um esquema de replicação ativa conhecida como abordagem por máquinas de estados.

> "Nessa técnica, uma aplicação é representada como uma máquina de estados que mantém variáveis de estado e sofre modificações em resposta a comandos de outras máquinas ou do meio. Para conseguir imunidade a falhas, a máquina de estado é replicada em múltiplos processadores independentes e um *multicast* atômico ordenado é usado para distribuir comandos para todas as réplicas de forma confiável e na mesma ordem." [Bakken e Schlichting, 1995]

Quanto aos mecanismos embutidos na semântica das operações, vale lembrar que o modelo de espaço de tuplas favorece a tolerância a falhas devido ao baixo acoplamento entre os processos, conforme foi discutido na Seção 2.5.3. Mas torna-se necessário um mecanismo explícito, que permita a recuperação perante a falhas. O mecanismo mais adotado nos diversos espaços de tuplas é o de transações. P-Linda, MOM [Cannon e Dunn, 1994], JavaSpaces e T-Spaces são exemplos de espaços de tuplas que usam transações.

Uma transação é um agrupamento de operações que possui propriedades especiais. Essas propriedades são conhecidas pela sigla ACID, que vem de atomicidade, consistência, isolação e durabilidade. Uma transação é *atômica* (isto é, indivisível) se, vista de fora, ela não apresenta estados intermediários. Para o mundo exterior (isto é, outras transações), parece que todas as operações da transação acontecem de uma única vez. Se algumas operações não puderem ser realizadas, então a percepção será de que nenhuma operação aconteceu. Para

implementar essa característica, o sistema deve ser capaz de desfazer algumas operações e recuperar o estado inicial. A propriedade de *isolação* garante que as transações não interferem nas outras ao longo de sua execução. Em outras palavras, a execução concorrente das transações acontece como se elas fossem executadas uma de cada vez. *Durabilidade* tem a ver com a persistência dos resultados e efeitos das transações. Transações tiveram origem em bancos de dados, mas sua adoção em sistemas distribuídos provou-se eficaz para os fins de preservação da integridade dos sistemas na presença de falhas.

3.6 Persistência

Diversos espaços de tuplas armazenam as tuplas persistentemente, como P-Linda, JavaSpaces e T Spaces. Persistência é um requisito para aqueles que adotam o modelo de transações, pois só assim atendem à propriedade de durabilidade (o "D" em ACID). Mas a persistência abre uma nova aplicação para os espaços de tuplas – eles passam a servir como repositórios de objetos. Isso aproxima um espaço de tuplas de um banco de dados, mas ainda há diferenças significativas. Uma delas é que um espaço de tuplas típico oferece uma linguagem de consulta muito simples, baseada na igualdade de campos. Há, no entanto, espaços de tuplas que oferecem linguagens mais ricas, como P-Linda e T Spaces.

3.7 Segurança

A questão da segurança no modelo de espaços de tuplas foi levantada por diversos pesquisadores [Pinakis, 1992] [Minsky e Leichter, 1995] [Wood, 1999].

> "A falta de segurança [em sistemas como Linda] pode ser demonstrada considerando-se um par de agentes que desejam comunicar-se usando uma tupla. O agente 1 escreve a tupla no espaço de tuplas e o agente 2 remove a tupla através de um gabarito compatível. Infelizmente (para os propósitos de uma conversação privada), não há nada que impeça um terceiro agente de remover a tupla – e isso pode ocorrer acidentalmente ou por má-fé." [Wood, 1999]

Em geral, as soluções propostas envolvem alguma forma de alteração no modelo de espaço de tuplas. Wood (1999) propõe que as tuplas sejam rotuladas com atributos que autorizem sua leitura e remoção do espaço. O esquema é, na verdade, um pouco mais amplo, ele propõe que os campos das tuplas possam ter seus próprios atributos, ou mesmo um espaço de tuplas inteiro tenha atributos. Wood discute os meios de implementar esse modelo, que

incluem *ACLs (access control lists)* e *capabilities*, mecanismos bem conhecidos em sistemas operacionais.

Ruple usa certificados digitais X.509 para implementar segurança. O padrão X.509 se aplica facilmente ao Ruple porque ele é um padrão para documentos e, em Ruple, as tuplas são documentos em XML.

T Spaces adota um esquema de controle de acesso ao espaço de tuplas. O controle é feito em nível de espaço de tuplas e não em tuplas ou campos. Os espaços de tuplas são organizados em domínios, que possuem um conjunto de usuários com acesso permitido. Esses domínios podem ser hierarquizados à moda do Andrew File System.

3.8 Considerações finais

O modelo de espaço de tuplas precisou ser adaptado para as mais diversas aplicações. JavaSpaces e T Spaces são tentativas de se fazer um espaço de tuplas para objetos em Java. Ruple é um espaço de tuplas para documentos compartilhados através da Internet. Lime é um espaço de tuplas para agentes móveis. A variedade de aplicações potenciais do modelo aponta para uma diversificação ainda maior.

Um espaço de tuplas genérico parece não fazer sentido. O modelo de espaço de tuplas é antes um conceito do que um software. E, seja qual for o "sabor" em que for servido, seus "nutrientes" fundamentais deverão estar sempre lá: memória associativa, acesso concorrente com semântica de memória compartilhada, simplicidade conceitual e baixo acoplamento.

Capítulo 4

Sun JINI, JavaSpaces e RMI

4.1 Considerações iniciais

JavaSpaces [Sun Microsystems, 1999a] [Freeman et al., 1999] é o espaço de tuplas para a linguagem Java proposto pela Sun Microsystems, a mesma empresa que criou a linguagem. É curioso que o JavaSpaces tenha sido o principal produto para computação distribuída anunciado pela Sun no primeiro ano de lançamento do Java [Waldo, 1996]. Depois, ele seria atropelado pelo *Remote Method Invocation* (RMI) [Sun Microsystems, 1998] e por Jini [Sun Microsystems, 1999b], que vieram a ser lançados no mercado antes de JavaSpaces. O próprio JavaSpaces foi modificado para se adaptar a esses dois produtos, que, na arquitetura para sistemas distribuídos em Java da Sun, ocupam níveis mais básicos.

O fato é que, para entender e usar JavaSpaces, o usuário deve ter algum conhecimento sobre RMI e Jini, estruturas muito mais genéricas que um simples espaço de tuplas. Se uma das vantagens dos espaços de tuplas é a simplicidade conceitual e até operacional, então o RMI e Jini jogam contra, criando um sobrecarga indesejável, especialmente para iniciantes ou usuários de pequeno porte. Uma das motivações do presente trabalho é divisar uma arquitetura para JavaSpaces que descomplique o máximo possível esse *overhead* operacional. Por outro lado, o uso do RMI e Jini tem suas vantagens e sua remoção criará outras dificuldades. O objetivo do presente capítulo é apresentar minimamente os sistemas Jini (Seção 4.2), JavaSpaces (Seção 4.3) e RMI (Seção 4.4) para que se possa avaliar a arquitetura a ser proposta no próximo capítulo.

4.2 Jini

Jini é uma tecnologia para desenvolvimento, administração e manutenção de sistemas distribuídos, especialmente aqueles que adotam a linguagem Java. A idéia central de Jini é a formação espontânea de comunidades de serviços. O exemplo clássico do funcionamento de

Jini é o de uma câmera fotográfica digital que, ao ser conectada a uma comunidade, busca por todos os serviços de impressão ali disponíveis, escolhe um deles e, em seguida, imprime fotos. Ela poderia, ainda, buscar serviços de armazenamento para enviar os arquivos. O detalhe que torna Jini tão especial é que esses procedimentos – e mesmo a instalação das impressoras e dos serviços de armazenamento – acontecem automaticamente, no melhor estilo *plug-and-play*. (Jini também oferece modelos de programação que conferem maior confiabilidade às aplicações).

São essas as peças-chave de Jini: serviço de *lookup* (com os protocolos associados – descoberta, filiação e *lookup*), *leasing*, eventos remotos e transações distribuídas.

O serviço de *lookup* é a entidade central em Jini (Figura 4.1). Pode haver um ou mais serviços de *lookup* disponíveis no sistema. As incumbências do serviço de *lookup* são manter um registro atualizado dos demais serviços Jini e fornecer, para as aplicações, *proxies* que dão acesso aos serviços.

O protocolo de descoberta é usado por serviços e aplicações para localizar serviços de *lookup* disponíveis. Os serviços de *lookup* que recebem a mensagem, respondem diretamente ao requisitante. Isto completa o protocolo de descoberta.

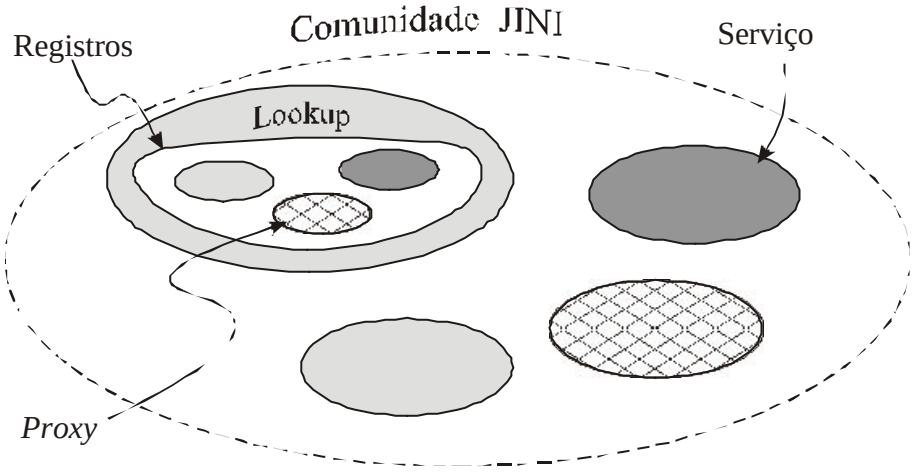

Figura 4.1 – O serviço de lookup mantém os proxies dos serviços.
Adaptado de Jarsen e Spring (1999).

Na seqüência, o protocolo de filiação é usado para os serviços se registrarem no(s) servidor(es) de *lookup*. Ao fim desse processo, eles devem deixar um *proxy* no serviço de

lookup. Um *proxy* (representante, procurador) é um objeto[3] em Java com a capacidade de se comunicar com o serviço, possivelmente através do RMI. O serviço de *lookup* registra os serviços e mantém os *proxies* mediante um *lease* (Seção 4.2.1). Esse mecanismo é especialmente útil para cancelar serviços que apresentam mal-funcionamento. Como o *lease* expira após um certo tempo, ele precisa ser renovado, mas isso só será feito por serviços ativos.

Uma aplicação pode consultar o servidor de *lookup* para encontrar serviços disponíveis na comunidade. Para isso, existe um protocolo de *lookup*, que funciona com gabaritos, à moda dos espaços de tuplas.

4.2.1 O mecanismo de *leasing* de Jini

Para facilitar a manutenção dos sistemas construídos com Jini, existe o mecanismo de *leasing*. Todo serviço Jini deve ser fornecido por um período de tempo negociado por quem requisita e quem fornece o serviço. O requisitante deve informar ao fornecedor o período de tempo durante o qual deseja usufruir do serviço. O fornecedor, ao conceder o serviço, leva em conta o tempo solicitado e sua própria capacidade em fornecer o serviço. Então ele passa para o requisitante um objeto que implementa a interface *Lease*[4]. Através deste objeto, o requisitante pode consultar o tempo concedido, pode pedir renovação do serviço por mais tempo (renovação do *lease*) ou pode mesmo cancelar o serviço antes do tempo.

Com o *leasing*, nada é para sempre. Se um serviço falhar, com o passar do tempo e a expiração do *lease*, ele será cancelado. Em serviços persistentes, objetos gravados e "esquecidos" serão apagados quando seu *lease* expirar.

4.2.2 A arquitetura de *proxies*

As interações cliente-servidor em Jini são modeladas como chamadas a procedimentos. Os *proxies* são os objetos que recebem as chamadas junto aos clientes e acionam os servidores.

---

[3] No presente texto, pressupõe-se que o leitor esteja familiarizado com os conceitos básicos do paradigma de orientação a objetos tais como *objeto*, *classe*, *campo*, *método*, *interface*, *hierarquia*, *herança*, *visibilidade*, etc. bem como a linguagem de modelagem UML e a organização de pacotes em Java.

[4] Do pacote *net.jini.core.lease*

A semântica das chamadas a procedimento em Jini tem duas particularidades: ela permite que objetos reais sejam usados como parâmetros e reconhece as falhas da rede. Por objetos reais, entende-se que não apenas dados, mas também código, seja transmitido entre as máquinas, possibilitando que uma cópia integral de um objeto que originalmente estava na máquina cliente seja executada na máquina servidora ou vice-versa. Por reconhecer as falhas de rede entende-se que todas as operações remotas devem prever a ocorrência de uma exceção relativa às falhas da rede. Esses dois aspectos da semântica também são tratados no RMI e serão vistos com detalhes na Seção 4.4.

Um ponto chave em Jini é a obtenção do *proxy*. Um cliente não deve dispor do *proxy a priori*, ele deve obtê-lo dinamicamente, e esse processo é o primeiro passo no uso do serviço. O *proxy* obtido dinamicamente possui configurações internas atualizadas e está menos propenso a falhas devido a mudanças de configuração. Jini define protocolos muito bem estruturados para que clientes e serviços consigam negociar *proxies*. A peça chave nos protocolos é o serviço de *lookup*, como já foi explicado. A presença de um *proxy* de um serviço no serviço de *lookup* significa a disponibilidade do serviço, visto que os *proxies* são registrados mediante *leasing*.

Do ponto de vista de um serviço, o acesso a um serviço de *lookup* é obrigatório dentro da arquitetura Jini. Isto é, todo serviço Jini deve ser capaz de encontrar um serviço de *lookup* (através do protocolo de descoberta) e registrar seu *proxy* nele (protocolo de filiação). Essa é a forma *default* para um serviço disponibilizar seu *proxy*. *Mas não é a única*. Nada impede que outros meios sejam usados, como, por exemplo, pelo acesso direto a uma porta de escuta em uma URL[5]. Dessa forma, do ponto de vista do cliente, o acesso ao servidor de *lookup* é opcional, isto é, se o serviço oferecer outras formas de contato para a obtenção do *proxy*, o cliente não precisará acessar o serviço de *lookup* (através do protocolo de *lookup*).

Na prática, o serviço de *lookup* tem sido a única forma empregada para fornecimento dos *proxies*. Disponibilizar mecanismos alternativos para obtenção do *proxy* não só é teoricamente previsto no modelo Jini, como tem a vantagem de dispensar o serviço de *lookup* em alguns cenários. O serviço de *lookup* é relativamente pesado, em certos aspectos, e nem

[5] Ken Arnold em mensagem para a lista de discussão *JavaSpaces-Users* no dia 22/1/2002.

sempre é essencial. Não se conhece, no entanto, nenhuma aplicação Jini que funcione sem a presença dele.

Proxies e o acesso a eles são centrais em Jini. Demais elementos do núcleo de Jini (*leasing*, eventos remotos e transações) são modelos de programação. Sua adoção por parte de um serviço é favorecida, mas não exigida.

4.2.3 O modelo de transações Jini

Jini tem suporte a transações distribuídas, cuja conveniência em sistemas distribuídos já foi abordada na Seção 3.5. Jini adota o protocolo de *commit* em duas fases [Tanenbaum, 1995, p. 153] como modelo para transações distribuídas.

Os objetos que participam do protocolo desempenham algum dos seguintes papéis: cliente, participante, gerenciador ou transação. O cliente é qualquer objeto que use um serviço. Participante é o objeto que provê o serviço. Vários participantes podem participar de uma mesma transação. Gerenciador é o objeto responsável por fazer o controle da transação, desde a sua criação até seu termo. O gerenciador é um objeto único no sistema (do ponto de vista de uma transação) e pode-se dizer que ele é o próprio serviço de transações.

A iniciativa das ações sempre cabe ao cliente. Há quatro ações que o cliente pode tomar: criar uma transação, usar uma transação em uma operação, encerrar uma transação com *commit* e abortar a transação.

O cliente cria uma transação junto ao gerenciador. Preliminarmente, o cliente deve ter obtido o *proxy* do gerenciador junto ao servidor de *lookup*. Como resultado da operação, o cliente recebe do gerenciador um objeto transação. Cabe ressaltar que o cliente também pode usar objetos do tipo transação criados e/ou repassados por terceiros.

O objeto transação é usado pelo cliente junto a um serviço oferecido por um participante. O participante oferece operações que têm um parâmetro específico para receber objetos do tipo transação. Ao receber uma transação da qual ainda não participa, o participante busca o gerenciador daquela transação e cadastra-se junto a ele como participante daquela transação. Cada objeto transação possui internamente uma referência para o gerenciador e um número de identificação da transação.

O cliente pode solicitar ao gerenciador que execute um *commit* em uma transação. O gerenciador realiza o pedido em duas fases (daí o nome de protocolo de *commit* em duas

fases). Na primeira fase, ele consulta todos os participantes para saber se estão prontos para execução do *commit*. Na segunda fase, ele envia um comando de *commit* para os participantes, no caso de todos estarem prontos. Se não estiverem, ele envia um comando para abortar a transação.

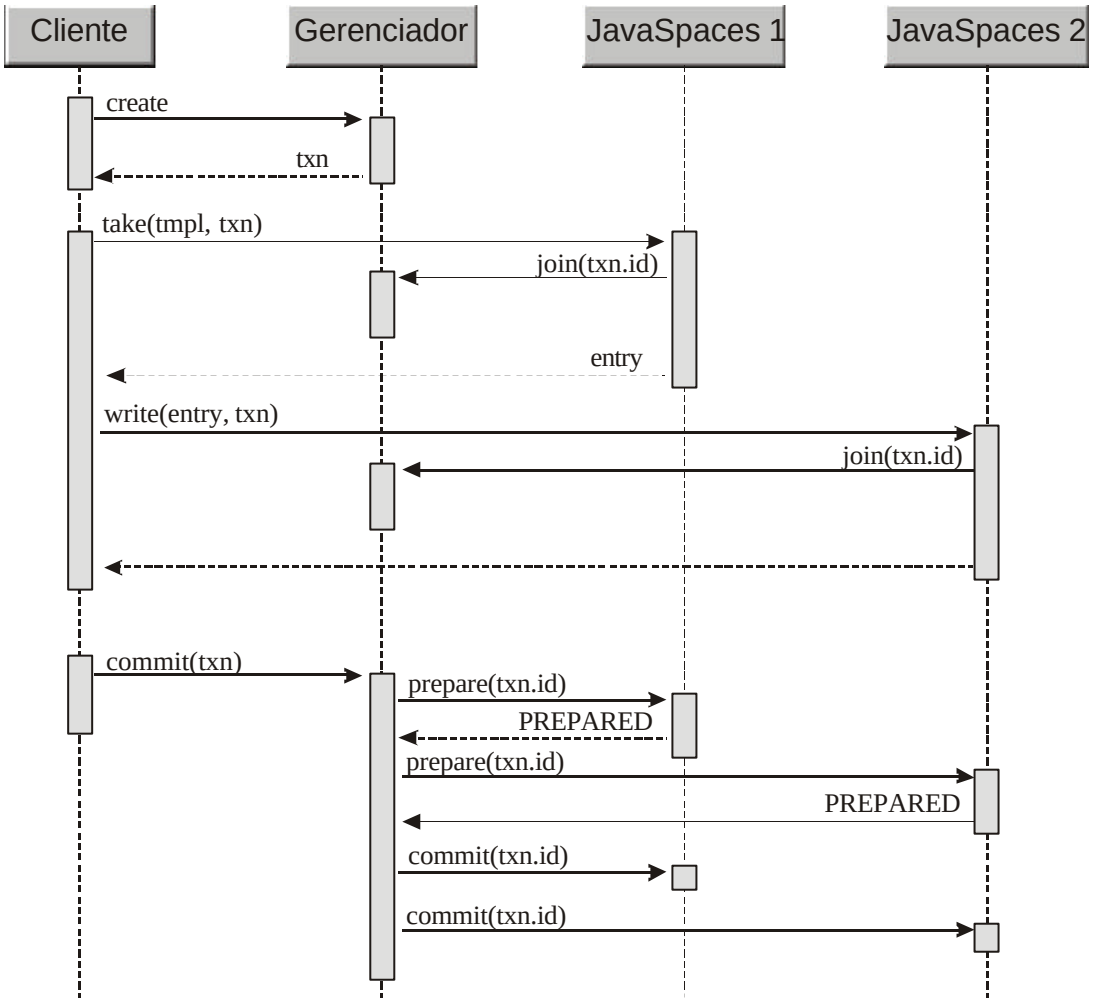

Figura 4.2 – Exemplo de ações que um cliente pode tomar no modelo de transações Jini.

Por fim, o cliente pode solicitar ao gerenciador que aborte a transação. O gerenciador simplesmente repassa a cada participante o comando para abortar.

Para ilustrar o protocolo, a Figura 4.2 apresenta uma seqüência possível envolvendo um cliente e dois serviços JavaSpaces (como participantes). Primeiramente, o cliente cria a transação. Ele executa duas operações sob a transação e depois pede um *commit*.

4.3 JavaSpaces

JavaSpaces é um espaço de tuplas que, em comparação com Linda, adota uma nomenclatura diferente para quase tudo. Em língua inglesa, a terminologia de JavaSpaces é mais próxima do cotidiano do que a terminologia de Linda. Uma tupla é chamada de *entry* (entrada, no sentido de "entrada de uma relação" ou "entrada de uma matriz"). As operações *out*, *in* e *rd* tornam-se *write* (escreva), *take* (remova) e *read* (leia). As operações *inp* e *rdp* tornam-se *takeIfExists* (removaSeExistir) e *readIfExists* (leiaSeExistir). Campos formais são chamados de *wildcards* (coringas).

As diferenças não são apenas de nomenclatura. JavaSpaces apresenta duas novas operações: *notify* e *snapshot*. *Notify* pode ser considerada como uma operação de leitura baseada em notificações e será vista em detalhe na Seção 4.3.4. E *snapshot* não acrescenta nada à semântica do modelo visto que é uma operação de otimização. *Snapshot* é usada quando uma mesma *entry* vai ser escrita muitas vezes no espaço de tuplas e, então, pode-se aproveitar procedimentos usados na gravação da primeira *entry* na gravação das demais. Outras novidades nas operações de JavaSpaces são o vínculo com transações e a duração controlada. O conjunto de operações de JavaSpaces está escrito abaixo, na linguagem Java (esses métodos são da interface *JavaSpace*[6]*)*.

```
Lease write(Entry e, Transaction txn, long lease)
Entry read(Entry tmpl, Transaction txn, long timeout)
Entry readIfExists(Entry tmpl, Transaction txn, long timeout)
Entry take(Entry tmpl, Transaction txn, long timeout)
Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)
EventRegistration notify(Entry tmpl, Transaction txn, long lease)[7]
Entry snapshot(Entry e)
```

Na presente seção, são explicados os principais pontos da especificação JavaSpaces, tais como o conceito de *entry*, a operação *write*, as operações de leitura e a operação de notificação. Também é explicado como o mecanismo de transações afeta a semântica das operações.

---

[6] Do pacote *net.jini.space*.

[7] Dois parâmetros foram suprimidos, por simplicidade.

4.3.1 Entry

As *entries* que preenchem o espaço de tuplas JavaSpaces e seus campos são objetos[8] em Java. Os campos da *entry*, se tiverem visibilidade pública, funcionam como os campos de uma tupla em Linda, com uma sutil diferença: o campo de uma *entry* possui tipo, valor e *nome*, enquanto o campo de uma tupla possui tipo, valor e *posição*. Os métodos das *entries* são uma novidade que as tuplas não têm, dando *comportamento* às *entries*, mas sem efeito algum no funcionamento do espaço de tuplas.

As classes das *entries* podem ser hierarquizadas segundo relações de herança. Na raiz da hierarquia deve estar a interface *Entry*[9]. Para exemplificar, considere a Figura 4.3. *Produto* e *Importado* implementam a interface *Entry*. (*Importado* é uma subclasse de *Produto)*. A interface em si não define nem campos nem métodos; ela apenas rotula o objeto, informando ao sistema *run-time* Java que o objeto está apto a passar por alguns processamentos (como serialização).

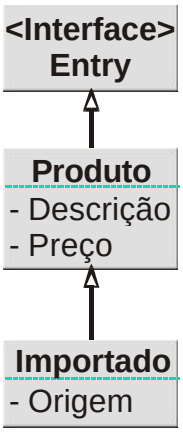

Figura 4.3 – Exemplo de hierarquia de classes de entries.

4.3.2 A operação de escrita

Uma *entry* pode ser escrita no espaço através da operação *write*. O primeiro parâmetro da operação é, obviamente, uma referência para a *entry*. Além disso, a operação *write* tem como parâmetro um tempo de *lease* (tempo requerido para o armazenamento da tupla) e como

[8] Não é possível usar tipos primitivos (*int*, *long*, *float*, *double*, etc) como campos das *entries*.

[9] Do pacote *net.jini.core.entry*

retorno um objeto do tipo *Lease*. O espaço de tuplas pode conceder o tempo requerido ou um tempo menor, se não for possível atender a requisição.

Do ponto de vista da implementação do JavaSpaces, o *leasing* quebra o desacoplamento entre a *entry* e o processo que a escreveu, pois o sistema terá, implicitamente, de relacionar os dois de alguma forma.

Um processo que escreveu 500 *entries* no espaço de tuplas terá 500 objetos do tipo *Lease* para gerenciar. Felizmente, pelo modelo de *leasing* de Jini, um outro objeto pode ser incumbido de gerenciar (renovar ou cancelar) os *leases*; basta que ele receba as referências para os mesmos.

4.3.3 Operações de leitura

Em JavaSpaces, gabaritos são usados na leitura de *entries*, assim como acontece em Linda. Um gabarito é uma *entry* como outra qualquer, exceto que ele é usado numa operação de leitura. Os gabaritos podem possuir *wildcards* (campos formais), que são campos cujo valor é uma referência *null*.

A compatibilidade entre gabarito é similar à de Linda, exceto que, além dos campos, a classe da *entry* e do gabarito também são comparados.

> "Quando se considera a compatibilidade de um gabarito G com uma *entry* E, campos com valores em G devem ter o mesmo valor do mesmo campo de E. *Wildcards* em G combinam com qualquer valor do mesmo campo em E.
>
> O tipo de E deve ser o mesmo de G ou um subtipo de G, caso em que todos os campos adicionais do subtipo são considerados como *wildcards*. Isso habilita um gabarito a ser compatível com *entries* de seus subtipos. A tupla removida deve ser do mesmo tipo de E." [Sun Microsystems, 1999a]

A Figura 4.4 ilustra o processo. À direita, existe um espaço de tuplas com cinco objetos (*entries)*, denotados por retângulos com os cantos arredondados. Estes objetos são das classes *Produto* e *Importado*, mostradas na Figura 4.3. Os retângulos são divididos para indicar os campos dos objetos. O valor do campo está escrito dentro da divisão. Objetos da classe *Produto* possuem duas divisões, correspondentes aos campos *Descrição* e *Preço*, de cima para baixo. Objetos da classe *Importado* possuem três divisões, referentes aos campos

[10] Do pacote *net.jini.core.lease*

Descrição, *Preço* e *Origem*. À esquerda da figura, estão três gabaritos. O primeiro gabarito é da classe *Produto* e possui um *wildcard* no campo *Preço*. Esse gabarito poderia ser usado numa consulta para saber o preço do produto cuja descrição é "Chapinha". No espaço de tuplas ilustrado na figura, há uma *entry* compatível com esse gabarito. O segundo gabarito é do tipo *Importado* e poderia ser usado numa consulta para achar um produto importado que seja produzido na África do Sul ou mesmo saber seu preço. A *entry* cuja descrição é "Amarula" é compatível com esse gabarito. Por fim, o terceiro gabarito poderia ser usado para saber o preço do produto descrito por "White Horse". O gabarito é da classe *Produto* e a *entry* compatível com ele é aquela com a descrição "White Horse", que é da classe *Importado*. Isso é possível porque *Importado* é subclasse de *Produto* (isto é, todo *Importado* é um *Produto*).

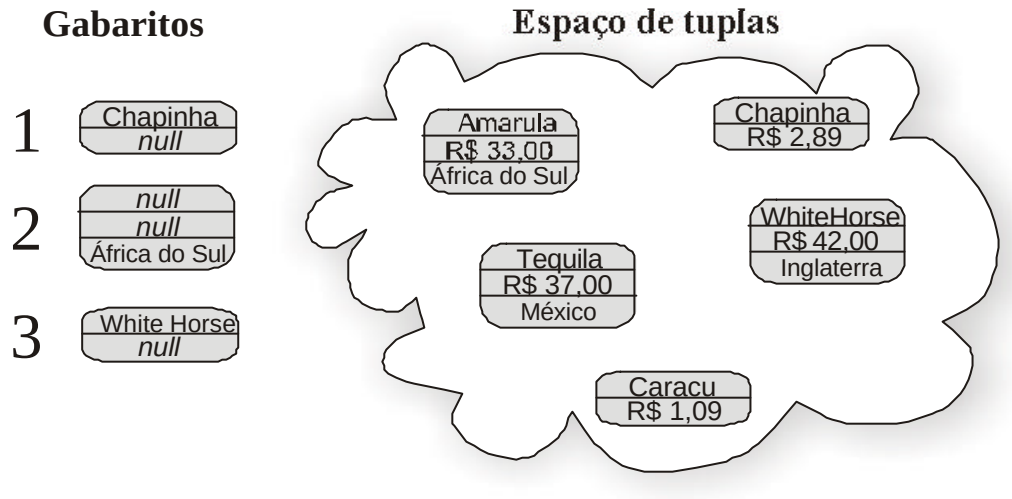

Figura 4.4 – Compatibilidade entre gabaritos e tuplas em JavaSpaces.

As operações de leitura em JavaSpaces possuem um parâmetro que especifica o tempo máximo de bloqueio (todas operações de leitura podem bloquear por influência de transações; além disso, as operações *read* e *take* bloqueiam quando não há uma tupla compatível).

4.3.4 Operação de notificação

A operação de notificação (*notify*) baseia-se no modelo de eventos remotos de Jini. Trata-se de uma versão do modelo de eventos em Java, com a diferença que os eventos não estão confinados a uma mesma máquina virtual.

Em JavaSpaces, a operação *notify* é usada por um objeto para registrar seu interesse em receber notificações (na forma de eventos remotos) sempre que o espaço de tuplas receber

uma *entry* compatível com um dado gabarito (que é parâmetro da operação *notify*). O retorno da operação é uma estrutura de dados que contém vários objetos, inclusive um *Lease*, que delimita o tempo de duração do serviço de notificação.

Toda vez que uma *entry* compatível com o gabarito for escrita no espaço de tuplas, um evento remoto será enviado para um objeto *RemoteEventListener*[11] indicado inicialmente por quem invocou a operação.

4.3.5 Semântica das operações sob transações

As operações *write, take, read, takeIfExists, readIfExists* e *notify* têm como parâmetro um objeto da classe *Transaction*[12] (Seção 4.2.3). Este objeto corresponde a uma transação que está em andamento sob a coordenação de um gerenciador de transações.

O parâmetro *Transaction* afeta o funcionamento das operações da seguinte forma:

- As operações *write, read* e *take*, se o parâmetro for uma referência *null*, funcionam como se estivessem numa transação de uma só operação.
- A operação *notify*, se o parâmetro for uma referência *null*, habilita um objeto a receber notificações de *entries* que foram escritas sem estar vinculadas a nenhuma transação.
- Uma *entry* que é escrita dentro de uma transação só passa a ser visível fora da transação quando esta termina com sucesso. Se a transação for abortada, a *entry* é descartada.
- A operação *read* realizada dentro de uma transação recupera uma *entry* vinculada à mesma transação ou vinculada a nada. A *entry* assim recuperada passa pertencer ao grupo de *entries* lidas pela transação. *Entries* desse conjunto podem ser lidas em outras transações, mas não removidas.
- A operação *take* realizada dentro de uma transação tem comportamento similar à operação *read*. *Entries* removidas por uma transação não podem ser lidas nem removidas por outras transações.

---

[11] Do pacote *net.jini.core.event*

[12] Do pacote *net.jini.core.transaction*

- A operação *notify* realizada sob uma transação implica que somente *entries* escritas sob aquela transação irão gerar notificações. Quando a transação termina, as notificações cessam.

4.4 Remote Method Invocation

O *Remote Method Invocation* (RMI) é uma versão para Java do modelo de chamada remota a procedimentos. Esse modelo é uma das mais populares técnicas de desenvolvimento para aplicações distribuídas. A idéia central é permitir que objetos, possivelmente localizados em nós diferentes de um sistema distribuído, interajam da mesma forma como fariam se estivessem num mesmo nó, ou seja, invocando métodos diretamente uns nos outros.

A técnica mais usada para implementar chamada remota a procedimentos lembra o uso de *proxies* em Jini. Para acessar um objeto remoto, usa-se um outro objeto, que é local e tem a mesma interface do objeto remoto. Em RMI, ele é chamado de *stub*. O *stub* aceita a chamada e seus parâmetros e repassa tudo para o objeto remoto. Existe na máquina remota a contraparte do *stub*, conhecida como *skeleton*, que atua como um cliente. O *skeleton* chama a operação junto ao objeto efetivamente capaz de executá-la, recebe o resultado e o envia para o *stub*, que o devolve a quem executou a chamada.

O RMI tem características peculiares que o diferenciam de outros esquemas de chamada remota a procedimentos. A primeira delas é que o RMI, desde a sua concepção, previu a necessidade de transmitir código, além da transmissão de dados que normalmente é feita. O RMI usa os mecanismos da linguagem Java para carregamento de código através da rede. O código de uma dasse pode ser descarregado de um nó para outro se estiver disponível em um servidor que atenda a requisições em um protocolo conhecido (HTTP, FTP, etc.).

O RMI prevê falhas de comunicação na rede e as torna explícitas em sua semântica. A questão de explicitar ou não falhas de comunicação na semântica de chamadas remotas foi levantada pelo próprio grupo de sistemas distribuídos da Sun, ainda antes da linguagem Java [Waldo et al., 1994]. Segundo os autores, a semântica de chamadas remotas jamais poderia ser idêntica à de chamadas locais, porque a latência e os problemas da rede são inevitáveis e o modelo de programação não pode simplesmente escondê-los (torná-los "transparentes"). O

[13] Ken Arnold em mensagem para a lista de discussão *JavaSpaces-Users* no dia 22/1/2002.

modelo deve permitir que esses problemas intrínsecos à comunicação entre máquinas sejam identificados e tratados. Coerentemente, esse mesmo grupo, quando do surgimento da linguagem Java e da demanda por um mecanismo de chamadas remotas para a linguagem, incluiu no RMI a exceção *RemoteException*, que pode eventualmente ser lançada em qualquer operação de natureza remota.

O RMI possui mecanismos para localizar e ativar objetos remotos. A localização de um serviço é feita com auxílio de um serviço de nomes, que mantém uma lista dos serviços disponíveis e suas URLs. Ele torna a configuração do sistema independente de endereços físicos, que podem ser mudados sem alterar o código dos clientes. A ativação se dá quando o objeto é acessado, mas, por algum motivo, talvez uma pane, não esteja em execução. Um ativador (na verdade, o *daemon rmid*), então, inicia a execução do objeto em uma máquina virtual apropriada. A ativação confere persistência aos serviços, porque eles podem ser recuperados automaticamente.

O desenvolvimento de uma aplicação usando RMI envolve novas etapas, como a geração do *stub* e do *skeleton*. Executar uma aplicação desenvolvida em RMI requer que um servidor de nomes e os *daemons* sejam preliminarmente executados.

4.4.1 Jini e RMI

Entre os usuários Jini, especialmente aqueles que tomaram contato com a tecnologia através de livros de divulgação, estabeleceu-se uma confusão permanente entre Jini e RMI, que já é aceita como fato[14]. Até que ponto, para desenvolver uma aplicação Jini é necessário usar o RMI?

O fato é que a especificação Jini não exige a adoção de nenhum protocolo em particular, nem mesmo o RMI. Esse é um princípio essencial, garantido pela arquitetura de *proxies* [Waldo, 2000]. Um cliente de posse de um *proxy*, que representa um serviço remoto, desconhece completamente o mecanismo pelo qual o *proxy* e o serviço se comunicam. Aliás, se esse for o caso, pois é permitido que o *proxy*, ele mesmo, sem comunicação com nenhuma outra parte, possa implementar o serviço localmente (na mesma máquina virtual Java do cliente), se isso for possível, como acontece em alguns serviços muito simples. No caso do

[14] Rick Kitts, em mensagem para a lista de discussão *Jini-Users*, em 9/8/2002.

proxy precisar comunicar-se com outra máquina, o protocolo usado entre eles é completamente livre, estando assim o projetista do serviço à vontade para escolher a estratégia que lhe for mais conveniente. Para citar exemplos de protocolos conhecidos, TCP, IIOP (CORBA), HTTP com XML ou mesmo o RMI.

A origem da confusão está numa estratégia de divulgação pedagogicamente mal formulada por parte da Sun. Primeiro, por incluir na especificação algumas classes que pertencem ao pacote *java.rmi* e, segundo, por fornecer uma implementação de Jini fortemente dependente do RMI.

As classes *Remote, RemoteException* e *MarshalledObject* são as classes do pacote *java.rmi* que aparecem em diversos pontos da especificação de Jini. Elas estão no pacote do RMI, mas poderiam estar em um outro lugar qualquer, pois os propósitos a que servem são mais gerais e poderiam ser usados em outros contextos que não só aqueles em que o RMI é aplicado. Entretanto, pela menção do pacote *java.rmi* na especificação, pode-se concluir, erroneamente, que a adoção do RMI seja mandatória.

O propósito da classe *RemoteException* é incluir a semântica de falhas de comunicação nas operações realizadas com *proxies* que fazem uso da rede. A possibilidade de falhas, como já discutido na Seção 4.4, tem que estar explícita na arquitetura distribuída, e não escondida (transparente). Essa é uma exigência também para as operações realizadas em *proxies* que não usam o RMI. Por isso, elas também poderiam lançar *RemoteExceptions*.

A interface *Remote* é uma daquelas interfaces muito usadas na linguagem Java para "marcar" um objeto como de um certo tipo (ou com uma certa propriedade), mas que não define operações. Um objeto do tipo *Remote* é aquele que pode estar uma outra máquina e, portanto, deve ter um tratamento diferenciado em certas situações.

Um objeto da classe *MarshalledObject* contém dentro de si um objeto serializado (Seção 4.4.2) e uma URL, onde o código da classe do objeto pode ser obtido. Além de especialmente útil para armazenamento, um *MarshalledObject* é extremamente conveniente por já ter um mecanismo de carregamento de classe automático. Ao extrair o objeto serializado de dentro do *MarshalledObject*, desserializando-o (*unmarshalling*), se o código de sua classe não estiver disponível na própria máquina onde ocorre o *unmarshalling*, um carregador de classe por rede usa a URL embutida para conseguir o código. Por natureza, essa funcionalidade não é de uso apenas em conjunção com o RMI, mas mais geral. (O

MarshalledObject tem outras funcionalidades específicas para o RMI, mas que podem ser ignoradas.)

Quanto à implementação de Jini oferecida pela Sun, os serviços de *lookup* e transações distribuídas, ambas de grande utilização pelas aplicações, foram implementados com RMI. Dessa forma, qualquer negociação com esses serviços requer o RMI. No entanto, seria completamente plausível pensar em uma implementação que utilizasse outros protocolos.

4.4.2 Serialização de objetos em Java

Serialização de objetos é um procedimento utilizado em diversas circunstâncias, não apenas pelo RMI. Mas como o RMI faz uso intensivo de serialização, ela aparece neste documento como um tópico da seção de RMI.

Dispositivos que recebem ou devolvem dados em *série* (isto é, pedaço a pedaço, um após o outro) não comportam naturalmente os objetos, que, em sua forma mais genérica, possuem estruturas não-lineares. Mas há situações importantes em que os objetos precisam estar em dispositivos seriais, como na transmissão de objetos através de redes – que é o caso do RMI – e no armazenamento de objetos em discos (persistência).

Os processos de transformação da estrutura de um objeto em uma estrutura linear e vice-versa são conhecidos como serialização e desserialização. Em Java, existe uma arquitetura bem definida para serialização de objetos [Sun Microsystems, 1999c]. Ela prevê tanto a transmissão de objetos através de redes quanto a persistência em discos. Há um formato padrão para o objeto serializado, mas outros formatos podem ser usados.

O RMI usa o sistema de serialização de Java para transmitir parâmetros entre o *stub* e o *skeleton* e transmitir resultados na direção inversa. Na terminologia de chamadas remotas a procedimentos, a serialização e desserialização são chamadas de *marshaling* e *unmarshaling*.

Pode acontecer de um objeto ser transmitido para ser armazenado em uma máquina remota. Neste caso, o objeto não precisaria ser desserializado na máquina remota pois ele já estaria pronto (serializado) para gravação. A classe *MarshalledObject* é usada nesses casos. Um objeto dessa classe contém um objeto serializado, juntamente com a URL para acessar seu código.

O processo de serialização em Java é genérico o bastante para suportar transmissão e persistência de dados. A persistência é das duas a mais exigente. Por isso, aplicações em que

o desempenho da serialização é crítico, mas que não precisam do suporte à persistência, ganhariam com um processo de serialização mais simples. Esse é o caso, por exemplo, de aplicações de processamento numérico em sistemas distribuídos [Java Grande Forum, 1998].

Nem todo objeto pode ser serializado, por diversos motivos. Por exemplo, um objeto do tipo *Socket* encapsula um recurso do sistema que só vale para aquela máquina naquele instante. Portanto, não faz sentido transmiti-lo ou armazená-lo. Mas há motivos mais frouxos. Para estender a serialização a mais classes, outros esquemas têm sido propostos, inclusive usando a linguagem XML [JSX, 2002].

4.5 Considerações finais

As implementações atuais de JavaSpaces, como a maioria dos serviços Jini, não oferecem mecanismos alternativos para que o cliente obtenha o *proxy*. Devido a isso, tornou-se impossível pensar num cenário simples para a aplicação de JavaSpaces. O cenário possível atualmente inclui executar o seguinte: um servidor HTTP simples, o *daemon rmid*, o servidor de *lookup*, o servidor de transações, o JavaSpaces e, por fim, a aplicação. Para o iniciante, essa tarefa pode ser cansativa e frustrante, pois uma série de conceitos e dispositivos têm de ser bem entendidos para executá-la – conceitos que pouca relação têm com espaços de tuplas. O RMI, em particular, exige um aprendizado considerável. Esta questão está na origem e motivação do presente trabalho.

O JavaSpaces é um espaço de tuplas especificado com a arquitetura e os modelos de Jini. Como tal, ele herda as virtudes e vícios do mesmo. As virtudes ficaram evidentes ao longo deste capítulo. Dos vícios, pode-se citar, por exemplo, a omissão quanto a questões de segurança. As implementações de JavaSpaces devem ter uma visão mais ampla dos problemas da computação distribuída.

Assim como há uma confusão estabelecida entre Jini e RMI, Jini e JavaSpaces também têm sido confundidos. Uma coisa é um espaço de tuplas (JavaSpaces), outra é uma arquitetura para chamadas remotas a procedimento (Jini). Tanto é que seria possível imaginar uma implementação de Jini em cima de um espaço de tuplas. Bakken (2002) – um profundo conhecedor dos espaços de tuplas, mas nem tanto de Jini – chegou a pensar (e publicar) que JavaSpace fosse a base para a implementação de Jini. Tanenbaum (2002) também se confundiu. Para ele, JavaSpaces é um elemento essencial (quase que a definição) de Jini.

Capítulo 5

Implementação do JuspSpaces

5.1 Considerações iniciais

JavaSpaces é, antes de tudo, uma *especificação*. Empresas interessadas podem se candidatar a fornecer *implementações* desta especificação. Esse modelo de negócio foi usado pela Sun também na linguagem Java (existe o Java da Sun, IBM, Microsoft, etc.). Inicialmente, a Sun fornece uma implementação gratuita com os objetivos de divulgar a especificação e servir como referência para as demais implementações. No caso do JavaSpaces, a implementação da Sun foi disponibilizada publicamente na mesma época da divulgação da especificação (início de 1999) e recebeu o codenome de *Outrigger*. A primeira implementação fora da Sun apareceu somente dois anos depois e seu nome é *GigaSpaces*[15](inicialmente chamada de J-Spaces). Há ainda uma segunda implementação, de nome *Autevo*[16].

A especificação JavaSpaces não faz restrições quanto a estratégias de implementação do espaço de tuplas. O único aspecto da arquitetura que é amarrado pela especificação é o uso de *proxies* para acesso ao JavaSpaces. No mais, a liberdade é total – a implementação pode ser persistente ou transiente, centralizada ou distribuída, etc.

Um dos objetivos do presente trabalho é discutir aspectos relevantes para um projeto de implementação da especificação JavaSpaces. Por serem produtos comerciais, o *GigaSpaces* e o *Autevo* não apresentam abertamente suas técnicas. O *Outrigger* tem o código aberto, mas sob restrições da licença SCSL (Sun Community Source License), que, embora mais aberta, ainda é proprietária. O *GigaSpaces* incentiva a adoção de seu produto por universidades, liberando o seu código. O *Autevo* não faz menção a acesso a código. Nenhuma

[15] http:://www.gigaspaces.com

[16] http://www.intamission.com

das implementações publicou as técnicas utilizadas de forma sistematizada. Em alguns fóruns da Internet, é possível encontrar discussões de alguns aspectos sobre a implementação do *Outrigger*. Por outro lado, T Spaces [Wyckoff et al., 1999], que não é uma implementação de JavaSpaces, mas é similar em muitos aspectos, faz uma breve apresentação de sua arquitetura.

Este capítulo contém uma descrição do projeto usado para implementar JavaSpaces. A proposta em si está divididas entre as Seções 5.2 (Visão Geral), 5.3 (Módulo *MadLeaser*), 5.4 (Módulo *AcidChoir*) e 5.5 (Módulo *JuspSpaces*). Em seguida, trabalhos relacionados são comparados na Seção 5.6. O protótipo construído para testar a proposta e os resultados obtidos nos testes são discutidos na Seção 5.7. Por fim, as considerações finais.

5.2 Visão geral

O projeto de implementação de JavaSpaces proposto no presente trabalho teve por base os seguintes requisitos:

a. estar em conformidade com a especificação JavaSpaces,

b. ser uma implementação persistente, usando para este fim, se possível, tecnologias que sejam de fácil acesso,

c. possibilitar a implantação sem a presença de um serviço de *lookup* Jini,

d. utilizar tecnologias mais leves, em termos de implantação, do que o RMI.

O ítem *b* se justifica porque a implementação *Outrigger*, que é a implementação de referência da Sun, utiliza um mecanismo de persistência que é proprietário.

O ítem *c* acarreta que um procedimento para fornecimento dos *proxies*, alternativo ao serviço de *lookup*, seja adotado.

O ítem *d* reflete uma opção em favor de uma implantação mais simples (mesmo que isso possa dificultar o desenvolvimento de um protótipo).

Além do espaço de tuplas, um serviço de transações distribuídas precisou ser implementado, em decorrência dos itens *c* e *d*, visto que as demais implementações de serviços de transações disponíveis requerem o serviço de *lookup* e o RMI. Da mesma forma, foi necessário implementar e fornecer objetos que fazem parte do mecanismo de *leasing* e eventos remotos.

O projeto de implementação da especificação JavaSpaces proposto neste trabalho e seu protótipo receberam o codinome de *JuspSpaces*. Há uma divisão em módulos conforme as principais funcionalidades. Os módulos são os seguintes:

- *JuspSpaces*. É o módulo principal, responsável pela funcionalidade de espaço de tuplas.
- *Acid Choir*. É a implementação do serviço de transações distribuídas Jini.
- *Mad Leaser*. Contém várias classes responsáveis pelo mecanismo de *leasing* no *Acid Choir* e *JuspSpaces*.

As seções seguintes contêm uma visão geral dos módulos listados e detalham os principais pontos de cada um.

5.3 Módulo Mad Leaser

A classe *MadLease* e a interface *MadGrantor* são os personagens centrais no projeto do *MadLeaser*. *MadLease* é um *proxy* que implementa a interface *Lease* definida por Jini. *MadGrantor* é uma interface que deve ser implementada por qualquer serviço que ofereça a funcionalidade de *leasing* de Jini. Em associação com *MadGrantor*, existem duas classes utilitárias importantes: *MadDoor* e *Collector*. *MadDoor* é um *thread* que recebe conexões e *Collector* é um *thread* que aciona o *MadGrantor* periodicamente para que ele cancele os *leases* que tiverem expirados.

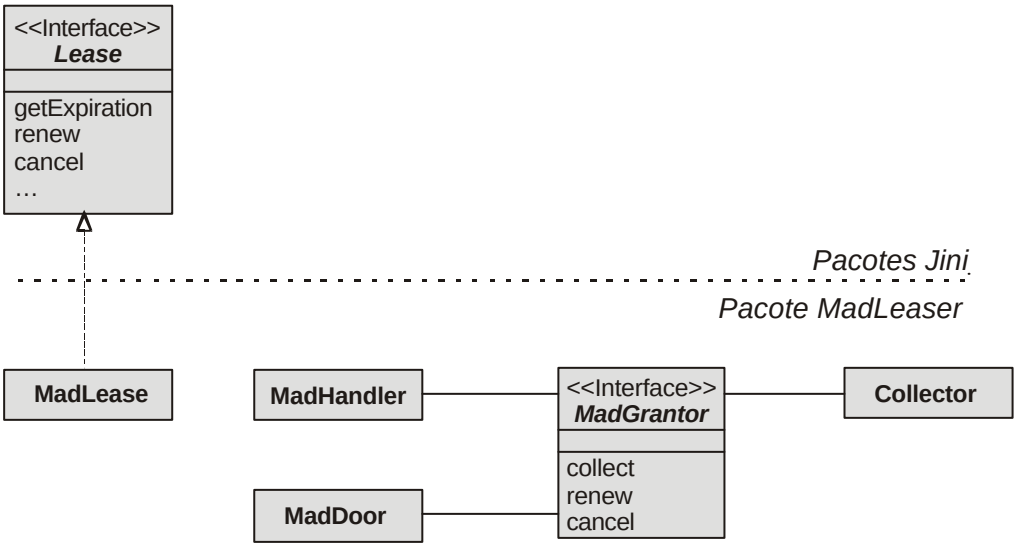

Figura 5.1 – Diagrama de classes do pacote MadLeaser.

O serviço que implementa *MadGrantor*, quando ele é iniciado, cria um objeto do tipo *MadDoor* (que passa a ter uma URL e uma porta de conexão) e outro objeto do tipo *Collector*.

Quando o serviço precisa retornar um objeto *Lease* para um cliente, ele cria um *proxy MadLease*. Internamente, o *MadLease* possui a URL e a porta de escuta do *MadDoor*. Quando o cliente chama um método no *proxy MadLease*, esse, por sua vez, contacta o *MadDoor*, que cria um *thread MadHandler* para tratar da requisição. O *MadHandler* identifica o método, recolhe os parâmetros e, em seguida, invoca o respectivo método no *MadGrantor* (o *MadHandler* possui uma referência para o *MadGrantor*, que lhe foi passada pelo *MadDoor*). O serviço deve implementar os métodos previstos na interface *MadGrantor*, e que têm a ver com a interface *Lease* de Jini. Ao terminar a requisição, a conexão entre *proxy* e serviço é desfeita e o thread *MadHandler* é encerrado.

Collector e o método *collect* do *MadGrantor* são a concretização, em termos de projeto de software, da funcionalidade de cancelamento de *leases* especificado em Jini. O objeto da classe *Collector* tem um *thread* que invoca o método *collect* no *MadGrantor* periodicamente. O período do *Collector* é definido no ato de sua criação. O efeito do método *collect* deve ser o de cancelar todos os *leases* que estiverem expirados no serviço que implementa *MadGrantor*. A duração efetiva dos *leases* é função, portanto, desse período e, provavelmente, difere da duração nominal que foi contratada com o cliente. Mas essa diferença será sempre no sentido de aumentar o *lease* e nunca diminuí-lo, atendendo, assim, à semântica de *leasing* de Jini.

5.4 Módulo Acid Choir

Acid Choir é uma implementação do serviço de transações distribuídas especificado por Jini, que, em essência, é o protocolo de *commit* em duas fases para sistemas distribuídos. O protocolo está ilustrado na Seção 4.2.3. Jini define interfaces para os personagens do protocolo.

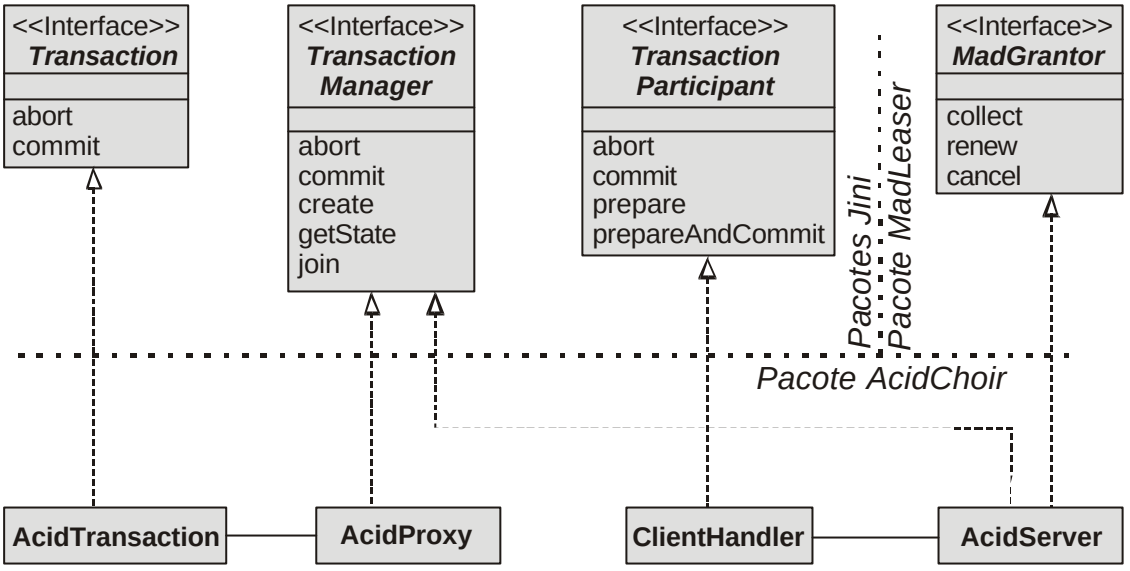

Figura 5.2 – Diagrama de classes do pacote AcidChoir.

TransactionManager é a interface do gerenciador. Ela define operações para criar, completar e abortar (*create*, *commit* e *abort*) uma transação. Além delas, há uma operação para um participante se filiar à transação (*join*) e outra para saber o status da transação (*getState*).

A interface *TransactionParticipant* deve ser implementada pelos participantes do protocolo. Ela define operações que o gerente usa para consultar o participante (*prepare*) e instruir o participante a completar ou abortar (*commit* ou *abort*) a transação. A operação *prepareAndCommit* é usada quando há apenas um único participante envolvido nas transações e a consulta e a instrução podem ser feitas de uma vez só.

A interface *Transaction* representa uma transação, que pode ser completada ou abortada através dela (nos métodos *commit* ou *abort*). Em *AcidChoir*, a interface *Transaction* é implementada pela classe *AcidTransaction*. Existe, internamente a ela, um objeto *AcidProxy*, que representa o *TransactionManager* no nó do cliente (*AcidProxy* implementa a interface *TransactionManager*). Quando o cliente invoca uma operação num objeto *AcidTransaction*, este, por sua vez, invoca o método correspondente no *AcidProxy*. Como o *AcidProxy* conhece a URL e a porta do gerenciador, ele o contacta.

Na máquina do gerenciador, a primeira ação é criar um *ClientHandler* para tratar da requisição. De posse dos parâmetros e do método, o *ClientHandler* invoca o método no

AcidServer, que é do tipo *TransactionManager* e é a real implementação do gerenciador de transações.

Internamente, *AcidServer* mantém uma lista de transações, com o status e os participantes de cada uma.

O objeto *AcidTransaction*, além de ser acionado pelo cliente, pode também ser acionado por um participante, para se filiar junto ao gerenciador. Quando isso acontece, o *AcidProxy* interno ao objeto *AcidTransaction* abre uma porta de escuta na máquina do participante, para receber as consultas e instruções do gerenciador. Do lado do gerenciador, o *ClientHandler* que atendeu a requisição de cadastramento do participante passa a funcionar como um representante do participante. Ele toma conhecimento da URL e da porta em que o *AcidProxy* espera por conexões. Para o gerenciador (o *AcidServer*), o *ClientHandler* é o meio de acesso ao participante. É por isso que o *ClientHandler* implementa a interface *TransactionParticipant*.

A Figura 5.5 mostra um diagrama de objetos envolvendo instâncias do *JuspSpace* e do *AcidChoir*.

5.5 Módulo JuspSpaces

JuspSpaces é o módulo responsável pela funcionalidade de espaço de tuplas. É a implementação propriamente dita da especificação JavaSpaces, visto que os demais módulos implementam funcionalidades Jini requeridos por JavaSpaces.

JuspSpaces adota uma arquitetura centralizada, assim como as demais implementações de JavaSpaces[17] e T Spaces, que não é uma implementação JavaSpaces, mas é similar. Essa arquitetura é muito mais simples de implementar. Arquiteturas distribuídas possuem dificuldades inerentes como a manutenção da consistência do sistema. Jarsen e Spring (1999) desenvolveram um espaço de tuplas com replicação – o Globe – e seu trabalho é uma excelente exposição das dificuldades que surgem em sistema distribuídos. Carreira et al. (1994) afirmam que o desempenho de arquiteturas centralizadas é aceitável em sistemas pequenos ou médios, da ordem de dezenas de máquinas. Vale lembrar que espaços de tuplas

[17] Recentemente, a implementação *GigaSpaces* passou a adotar mecanismos de replicação.

centralizados servem como unidades básicas para a construção de espaços de tuplas replicados, como no caso do Globe, em que cada réplica é uma instância do JavaSpaces.

No JuspSpaces, o armazenamento e a recuperação das tuplas é feito mediante um mecanismo de chaves. Cada campo de uma tupla ou gabarito tem uma chave correspondente, que é calculada segundo um algoritmo. O algoritmo de obtenção da chave deve ser tal que a probabilidade de que dois objetos distintos consigam a mesma chave seja extremamente pequena. Optou-se por fazer o cálculo da chave no cliente em vez do servidor, porque isso pode refletir num melhor desempenho das operações de leitura, particularmente nos casos em que o custo da transmissão do objeto é muito maior que o custo de transmissão da chave. Por outro lado, se a máquina cliente tiver uma capacidade computacional muito menor que o servidor, o cálculo da chave no cliente pode resultar num pior desempenho. Vale lembrar que, como cada campo é armazenado juntamente com sua chave, o cálculo da chave deve ser feito também nas operações de escrita.

A Figura 5.3 mostra o diagrama de classes do pacote *JuspSpace* com algumas classes omitidas por simplicidade. Do lado do cliente, a classe principal é *Proxy*, que implementa a interface *JavaSpace*. Tem ainda a classe acessória *KeyGenerator* com os métodos para geração da chaves a partir dos campos das *entries*.

Quando a aplicação precisa de um serviço de espaço de tuplas, ela aciona no *Proxy* o método desejado. O *Proxy*, por sua vez, contacta o servidor em sua porta de operação. Em seguida, um *thread Handler* é criado no servidor para tratar as requisições provenientes do *Proxy*. O thread *Handler* tem a função de meramente empacotar e desempacotar parâmetros e reconhecer as operações solicitadas. Tão prontamente ele consiga essas informações, ele trata de invocar a operação correspondente no objeto *Space*. Se a operação tiver retorno, o *Handler* o empacota e envia para o *Proxy*.

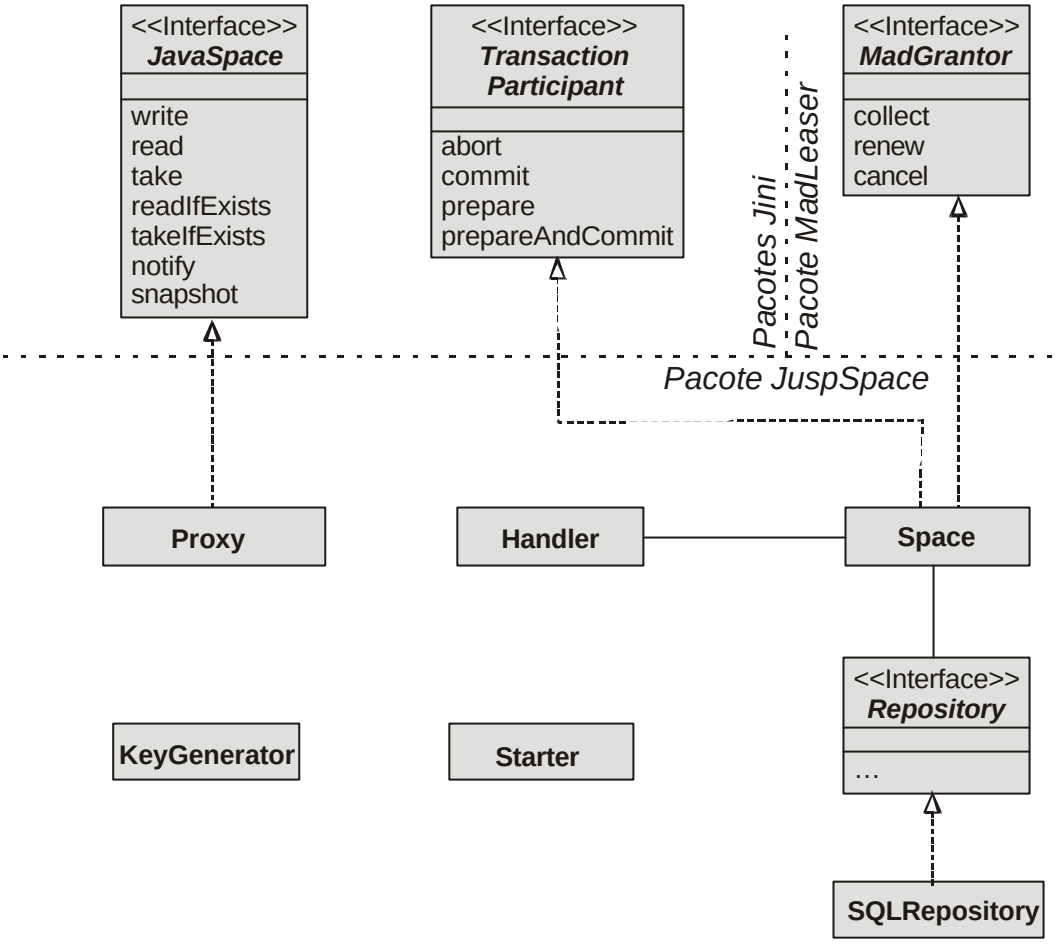

Figura 5.3 – Diagrama de classes do pacote JuspSpace.

Apenas um objeto da classe *Space* é instanciado. Como o próprio nome já diz, ele se confunde com o próprio espaço de tuplas. Os principais métodos oferecidos por *Space* referem-se às operações do espaço de tuplas. Ele só não implementa a interface *JavaSpace* porque as operações aparecem levemente alteradas, pois estruturas de dados mais convenientes são usadas para representar a tupla e não mais a classe *Entry*. O objeto da classe *Space* implementa a interface *MadGrantor*. Isso quer dizer que todas as operações de *lease* chegam a ele, bem como a chamada para recolher as *entries* com *lease* expirado, que é feita pelo objeto *Collector* de tempos em tempos. Por fim, *Space* é um *TransactionParticipant*, portanto ele filia-se a um gerenciador de transações quando uma nova transação é usada como parâmetro em alguma operação e, a partir daí, responde às requisições provenientes do gerenciador, como instruções de *prepare*, *commit* ou *abort*.

Internamente, o objeto da classe *Space* possui um objeto que implementa a interface *Repository*. Esse objeto do tipo *Repository* é responsável pelo armazenamento e pela

persistência de todas as *entries* do espaço de tuplas. Obviamente, ele também é único. A interface *Repository* e a classe *SqlRepository* que a implementa serão vistas em detalhes nas Seções 5.5.1 e 5.5.2.

A classe *Starter* é invocada no *shell* do sistema e põe a rodar todas as classes fundamentais para o funcionamento do espaço de tuplas, tais como *Space*, *Repository*, *MadDoor* e *Collector*. Ela recebe como parâmetros no comando de linha o nome do espaço e a porta de operação de *Space*. A porta para o *MadDoor* fica uma unidade acima.

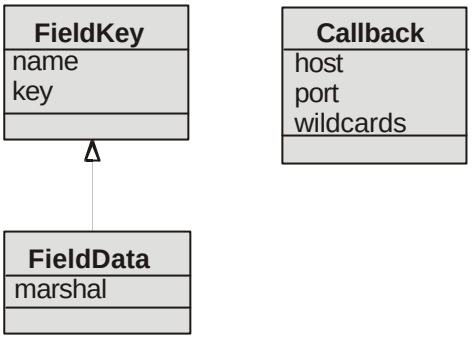

Figura 5.4 – Estruturas de dados usadas na representação de tuplas e gabaritos.

Foi mencionado que estruturas de dados especiais são usadas para representar uma *Entry*. Elas são *FieldKey*, *FieldData* e *Callback*, como mostra a Figura 5.4. *FieldKey* contém o nome do campo e sua classe. Ela é utilizada nas operações de leitura. Um gabarito torna-se um *array* de *FieldKeys*. *FieldData* é uma estrutura de dados com o nome do campo, a chave e o valor do campo (na forma de um *MarshalledObject*). Uma *entry* torna-se um *array* de *FieldDatas*.

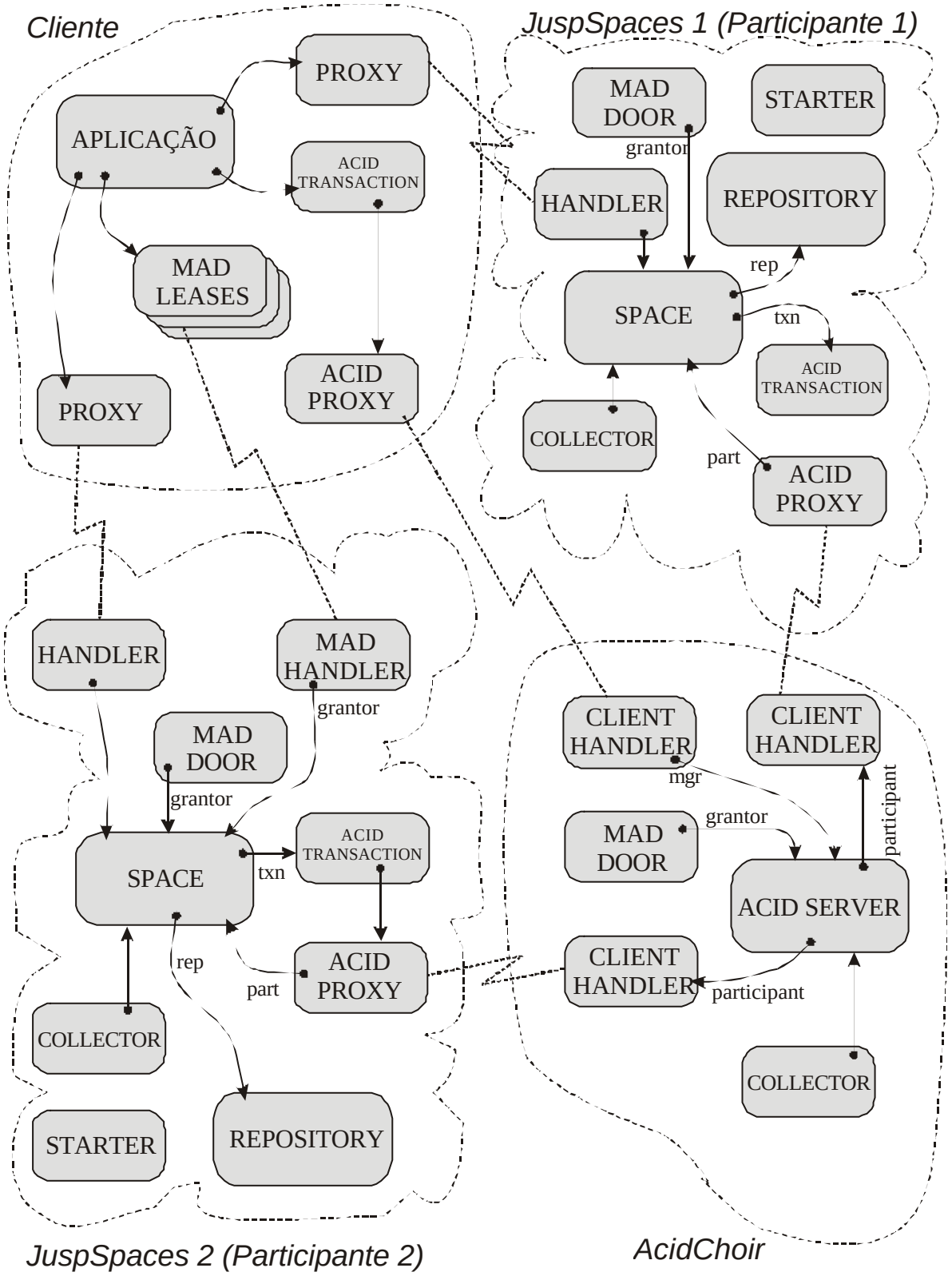

Figura 5.5 – Diagrama de objetos de uma operação em que participam uma aplicação, dois espaços de tuplas e um serviço de transações.

Callback é usado nas operações bloqueantes. Ele contém o nome da máquina em que a aplicação cliente está aguardando, a porta de escuta, um *array* de campos a serem retornados (os *wildcards*).

A Figura 5.5 representa um cenário hipotético em que uma aplicação cliente acessa dois espaços de tuplas distintos, tudo isso sob a mediação de um gerenciador de transações. Trata-se de um diagrama de objetos, no qual os objetos são representados por retângulos de cantos arredondados.

O conjunto de objetos relacionados é limitado por uma linha tracejada ou por uma nuvem, no caso de um espaço de tuplas. Objetos que se comunicam por *sockets* são relacionados por intermédio de uma linha tracejada. Referências são representadas por setas. O papel que o objeto referenciado exerce para o objeto que o referencia está indicado como texto junto à seta.

5.5.1 A interface *Repository*

A interface *Repository* (e suas implementações) é a peça mais importante de todo o pacote *JuspSpace*, pois as principais funcionalidades do sistema são implementadas por ela. Por exemplo, o armazenamento da *entry* juntamente com informações sobre transações e *leasing* é requisitado, em última instância, à interface *Repository*. E as etapas mais importantes da recuperação mediante gabaritos também são feitas por ela.

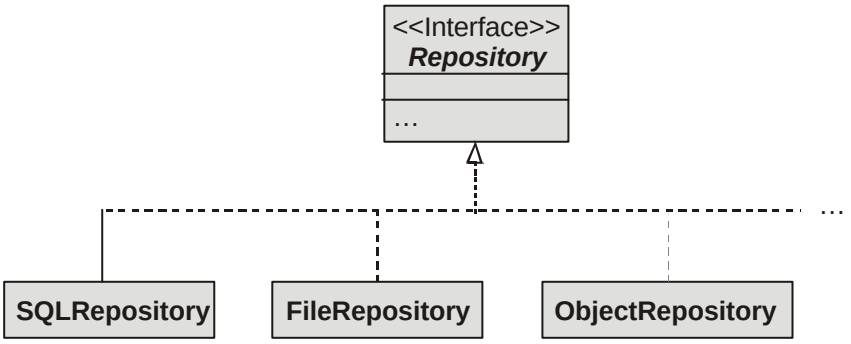

Figura 5.6 – Diferentes implementações para Repository. Somente SqlRepository foi implementada no protótipo.

Repository é a única entidade persistente no projeto. Toda informação que deve ser armazenada de forma persistente, passa pela interface *Repository*. Por isso, optou-se por definir essa classe como uma interface, para que diferentes estratégias de implementação da persistência fossem tentadas. A primeira estratégia prevista, e a única implementada no

protótipo do JuspSpaces, é usar um banco de dados relacional para armazenar e recuperar as informações. A classe que implementa *Repository* segundo essa estratégia ganhou o nome de *SqlRepository*, como menção à linguagem SQL, que é extensivamente usada no acesso a bancos de dados relacionais. Outras estratégias podem ser cogitadas: a gravação das informações diretamente no sistema de arquivos; o uso de novas tecnologias para persistência de objetos, como JDO [Sun MicroSystems, 2002]. A Figura 5.6 ilustra essa idéia.

O conjunto de operações da interface *Repository* é relativamente extenso. São, ao todo, 31 operações. Uma representação do conjunto pode ser vista na Figura 5.7.

```
-- lock methods --
Lock getLock()
void releaseLock(Lock)

-- group methods --
Group getGroup(Lock, Class)
Group[] getAllGroups(Lock)
Group[] getGroupsForClassAndSuperclasses(Lock, Class)
Group[] getGroupsForClassAndSubclasses(Lock, Class)

-- entry methods --
void insertEntryAndMark(Lock, Group+Id, FieldData[], Transaction, Lease)
Group+Id findEntry(Lock, Group, FieldKey[], Transaction)
FieldData[] getFields(Lock, Group+Id, Wildcards)
void removeEntry(Lock, Group+Id)
Id[] getEntriesWrittenUnderTransaction(Lock, Group, Transaction)
Id[] getEntriesTakenUnderTransaction(Lock, Group, Transaction)
Id[] getEntriesReadUnderTransaction(Lock, Group, Transaction)
void putTakenMark(Lock, Group+Id, Transaction)
void putReadMark(Lock, Group+Id, Transaction)
void removeWrittenMark(Lock, Group+Id)
void removeTakenMark(Lock, Group+Id)
void removeReadMark(Lock, Group+Id, Transaction)
void incrementEntryLease(Lock, Group+Id, Duration)
void removeExpiredEntries(Lock, Group)

-- template methods --
void insertTemplate(Lock, Group+Id, FieldKey[], Transaction, Timeout, ...)
Group+Id[] findTemplates(Lock, Group, FieldKey[], Transaction)
TemplateData getTemplateData(Lock, Group+Id)
void removeTemplate(Lock, Group+Id)
Id[] getExpiredTemplates(Lock, Group)

-- transaction methods --
void registerTransaction(Lock, Transaction)
void unregisterTransaction(Lock, Transaction)
State getTransactionState(Lock, Transaction)
void setTransactionState(Lock, Transaction, State)
boolean isTransactionRegistered(Lock, Transaction)
boolean isTransactionActive(Lock, Transaction)
```

Figura 5.7 – Conjunto de operações da interface Repository.

Essa representação sofreu simplificações para facilitar a leitura. Os parâmetros não aparecem com tipo e nome como exige a sintaxe Java, mas com um meio-termo. Alguns parâmetros foram suprimidos. As operações aparecem agrupadas por similaridade de função.

Um conjunto tão extenso não é desejável. Talvez, as funcionalidades oferecidas pela interface pudessem ser rearranjadas em algumas interfaces menores, mas os esquemas pensados eram inviáveis de implementar de acordo com a estratégia adotada para a classe *SqlRepository*. A intenção do projeto da interface *Repository* era chegar a um resultado genérico, mas ele foi fortemente influenciado pelas dificuldades encontradas na implementação da *SqlRepository*. Teme-se que outras estratégias possam exigir alterações no projeto da interface.

Todas as operações da interface *Repository* têm como parâmetro um *lock* (exceto, obviamente, a operação que cria um novo *lock*, chamada *getLock*). *Lock* é um termo mal empregado aqui. A função do *lock* é agrupar operações em uma única transação, de forma que elas sejam executadas atomicamente. Não se trata, no entanto, das transações que foram definidas no escopo do cliente e do gerenciador de transações. São transações internas do espaço de tuplas e se aplicam somente no contexto das operações de *Repository*. Isso se justifica porque as "macro" operações no nível do espaço de tuplas (*write, take, read, etc.*) são quebradas em diversas pequenas "micro" operações no nível de *Repository* e essas "macro" operações, mesmo quando não envolvem transações, precisam ser atômicas. A "macro" operação só é validada quando o *lock* é liberado através da operação *releaseLock*.

A implementação *SqlRepository* está baseada na tecnologia JDBC, a forma mais popular de acesso a bancos de dados relacionais em Java. Em JDBC, só é possível iniciar e finalizar uma transação dentro da mesma "conexão", que é um conceito do JDBC que se materializa na forma de um objeto do tipo *Connection*. Daí a necessidade de que o objeto *Connection* seja repassado a todas as "micro" operações que são realizadas conjuntamente. Numa tentativa de generalizar a idéia, surgiu a idéia do *lock*.

Internamente a uma implementação de *Repository*, as *entries* são agrupadas de acordo com a classe a que pertencem. Na verdade, essa é mais uma influência do projeto da *SqlRepository*, no qual optou-se por dividir as *entries* em tabelas distintas. Cada conjunto de *entries* da mesma classe é denominado de *grupo*. Toda operação que envolve uma *entry* ou um gabarito tem, como um dos passos iniciais, de requisitar a informação sobre qual grupo corresponde à classe daquela *entry* ou gabarito. Para isso, existe a operação *getGroup*. Os

grupos devem ser organizados de forma a refletir a hierarquia das classes que representam. Essa informação é importante porque a recuperação de *entries* pode requerer uma busca em toda a hierarquia de classes. A operação *getGroupsForClassAndSubclasses* tem como resultado um vetor de grupos que corresponde a toda a árvore hierárquica que se inicia numa dada classe, percorrida em largura. Já a operação *getGroupsForClassAndSuperclasses* tem como resultado um vetor de grupos que corresponde a todos os ancestrais que implementam a interface *Entry* de uma dada classe e a ela mesma. Toda essa informação é registrada no momento da criação dos grupos. Se qualquer uma dessas operações para obtenção dos grupos recebe como parâmetro uma classe desconhecida (ou seja, que não possui um grupo correspondente ainda) então um novo grupo é criado. Se as suas superclasses também são desconhecidas, então os grupos correspondentes são criados. Uma relação entre classes e respectivas superclasses deve ser mantida pela classe que implementa a interface *Repository*. A operação *getAllGroups* é utilizada no cancelamento de *leases* expiradas, que se aplica a todos os grupos no *Repository*.

Entries e gabaritos são identificados por *ids*. A criação e a garantia de unicidade do id *não* é de responsabilidade do *Repository*, mas sim do objeto *Space*. Quase sempre, o id tem que ser usado juntamente com o grupo a que pertence a *entry* ou o gabarito.

Entries podem receber marcas de "escrita pela transação X", "removida pela transação Y" e "lida pelas transações Z, W, ...". Essas marcas são o mecanismo de *locking* das transações (transações aqui são aquelas definidas pelo gerenciador de transações). Existem várias operações para colocar e remover as marcas referentes a uma dada *entry*.

A operação *insertEntryAndMark* armazena uma *entry* no *Repository*. Se o parâmetro correspondente à transação não for *null*, a *entry* é marcada como escrita por aquela transação. Ela tem um parâmetro referente ao tempo de *lease* requerido pelo cliente.

A operação *findEntry* é usada para encontrar uma *entry* dentro de um dado grupo a partir das chaves de um gabarito. Seu resultado é, se houver uma *entry* compatível com o gabarito, o id da *entry* ou, caso contrário, *null*. Um aspecto importante sobre essa operação é que: *boa parte da semântica de transações definida na especificação JavaSpaces está definida na operação* findEntry. Ou seja, está sob o encargo de quem implementa esta operação cuidar para que a semântica de transações esteja correta. Dependendo da transação que *findEntry* receber como parâmetro (a transação sob a qual se dá a leitura) a visibilidade sobre o conjunto de *entries* muda. *Entries* escritas em outras transações certamente não

servem como resultado para *findEntry*. Da mesma forma, *entries* lidas e removidas são tratadas de forma diferenciada.

A operação *findEntry* retorna apenas o id da *entry*. Outras operações são usadas para obter a *entry* inteira (*getFields*) ou apagá-la.

As três operações *getEntriesWrittenUnderTransaction*, *getEntriesReadUnderTransaction* e *getEntriesTakenUnderTransaction* são usadas quando uma transação termina, seja com um *commit*, seja com um *abort*.

Há operações relacionadas com os *leases* das *entries*, como *incrementEntryLease*, que serve para renovar o *lease*, e *removeExpiredEntries*, que apaga todas as *entries* vencidas de um grupo.

Os gabaritos (*templates*) também são passíveis de armazenamento devido às operações bloqueantes. Eles devem ser armazenados separados das *entries*, mas ainda no mesmo grupo. Internamente à implementação de *Repository*, deve haver estruturas de dados distintas. O conjunto de operações para lidar com gabaritos é em boa parte similar ao das *entries*, embora menor: uma operação para inserção (*insertTemplate*), uma para busca (*findTemplate*), outra para remoção (*removeTemplate*) e uma última para obter os dados (*getTemplateData*). Sem falar na operação *getExpiredTemplates* que é usada para avisar aos clientes bloqueados que o tempo de espera por uma *entry* acabou.

Por fim, a interface *Repository* possui um conjunto de operações para lidar com transações. Todas as transações distribuídas em que o espaço de tuplas estiver envolvido têm informações mantidas em *Repository*.

5.5.2 A classe *SqlRepository*

A classe *SqlRepository* é uma implementação da interface *Repository* sobre um banco de dados relacional. Uma vantagem dessa estratégia é utilizar todo o ferramental disponível no banco de dados como, por exemplo, estruturas e algoritmos para recuperação de informação.

Na organização das tabelas, há apenas duas tabelas gerais. As demais são específicas dos grupos registrados no *SqlRepository*.

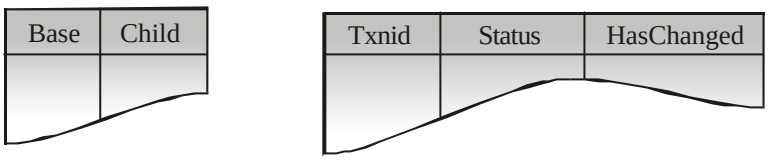

Figura 5.8 – Tabelas gerais de SqlRepository. Na esquerda, Groups. Na direita, Txns.

Groups é a primeira das tabelas gerais. Ela contém a informação de que grupos estão registrados no *SqlRepository* e qual é relação hierárquica entre eles. Para isso, a tabela possui duas colunas. Em ambas as colunas, os grupos aparecem como números; estes números são obtidos aplicando um algoritmo de *hashing* na imagem serializada da classe a que corresponde o grupo. Na primeira coluna, chamada "base", aparece um grupo; na segunda, chamada "child", um "subgrupo" seu, se houver. Se não houver, usa-se um valor sabidamente nulo. Caso um grupo tenha mais de um subgrupo, mais de uma entrada deve ser usadas.

Txns é a segunda tabela geral. Ela contém uma lista de transações ativas no espaço de tuplas. Para cada transação, as informações constantes são: Txnid, Status (cujos valores podem ser ACTIVE, VOTING, PREPARED, NOTCHANGED, COMMITED, ABORTED, conforme a especificação Jini) e um flag (HasChanged) que indica se houve alguma alteração desde o início da transação.

Para cada grupo, existem três tabelas, como mostrado na Figura 5.9. Uma das tabelas armazena as *entries* (a tabela superior na figura). A tabela recebe como nome o resultado da concatenação da letra "E" com o *hash code* do grupo. Cada linha da tabela corresponde a uma *entry*. A tabela possui colunas para: id da *entry* (*id*), horário de inserção da *entry* (*time*) tempo de *lease* (*lease*), id da transação que escreveu (*wtxn*), se for o caso, id da transação que removeu (*ttxn*), se for o caso, e mais uma série de colunas referentes aos campos da *entry*. São duas colunas para cada campo. A primeira coluna, cujo nome é formado a partir da concatenação do nome do campo com o sufixo "field" recebe a chave correspondente ao campo (ex. *foofield*). A segunda coluna contém um arquivo binário referente ao campo propriamente dito no formato serializado. O nome da segunda coluna é formado pelo nome do campo e o sufixo "data" (ex. *foodata*). Tem-se então que o número de colunas da tabela de *entries* varia conforme o grupo (e seu respectivo número de campos).

A tabela de gabaritos (a segunda na figura) tem por nome a letra "T" (de *template*) seguida do *hashcode* do grupo. As colunas são: id do gabarito (*id*), tipo do gabarito (normal ou de notificação) (*type*), horário da inserção do gabarito (*time*), tempo de armazenamento

solicitado (*expiration*), transação sob o qual foi gerado (*txn*), número de notificações que já foram geradas (para o caso dos gabaritos de notificação) (*number*), objeto_1 serializado (para gabaritos de notificação) (*obj1*), objeto_2 serializado (para gabaritos de notificação) (*obj2*) e mais colunas para os campos. Cada campo tem sua coluna correspondente, cuja denominação é o nome do campo mais o sufixo "field" e cujo conteúdo é uma chave gerada a partir do campo do gabarito (ex. *foofield*). Se o campo do gabarito é um *wildcard*, essa coluna assume o valor zero. Novamente, o número de colunas é variável conforme o grupo.

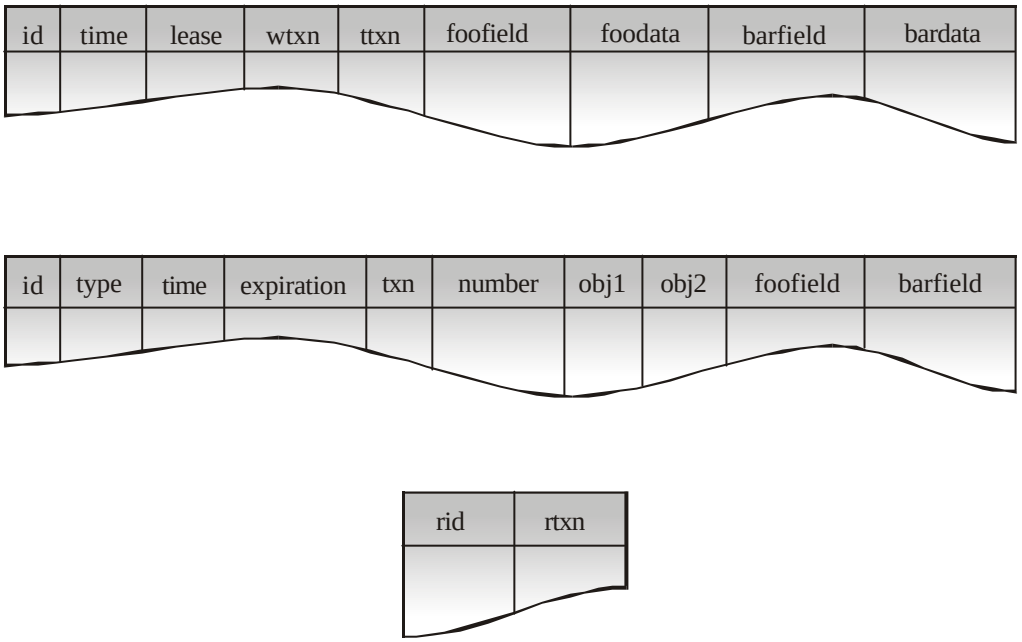

Figura 5.9 – Tabelas específicas de cada grupo no Repository. A classe das entries pertencentes a esse grupo possuem, hipoteticamente, apenas dois campos: foo e bar. A tabela de cima armazena as entries, a do meio, os gabaritos e a de baixo é uma relação de entries lidas por transações.

A terceira tabela é uma relação onde cada linha indica uma transação (*rtxn*) que leu uma determinada *entry* (*rid*). Várias transações podem ler uma mesma *entry*, por isso essa informação não pôde ser embutida na tabela de *entries* e uma tabela à parte teve que ser criada.

Esse esquema de tabelas não é o único possível. O mapeamento entre classes de objetos e bases de dados relacionais, especialmente quando se leva em conta a busca em toda a hierarquia de classes, é discutido na literatura, mas esse não foi um objetivo de pesquisa do presente trabalho. No entanto, alguns prós e contras são conhecidos:

- Como uma mesma *entry* está confinada a uma única linha da tabela de *entries* e, eventualmente, a umas poucas linhas na tabela de *entries* lidas por transações, então se espera que o número de *joins* que o banco tenha de realizar numa consulta seja muito baixo. Isso deve repercutir positivamente no desempenho.
- Ainda o fato de a *entry* estar em uma só linha numa tabela facilita a visualização do conteúdo do *Repository* e, portanto, do espaço de tuplas. Nenhum programa especial precisa ser criado, a própria interface texto dos gerenciadores de bancos de dados já serve. Isso favorece positivamente o *debug*.
- Os gerenciadores de banco de dados têm limitações quanto ao número de colunas numa tabela. Isso pode restringir o número de campos das *entries*, o que não é desejado.
- Um mesmo objeto pode aparecer como campo de várias *entries* diferentes. No esquema adotado, ele teria várias cópias serializadas gravadas no banco de dados, o que gera um desperdício de espaço em disco.

5.5.3 Principais operações

As quatro operações de leitura de JuspSpaces (*read*, *take*, *readIfExists*, *takeIfExists*) são implementadas internamente por uma única função chamada *getEntry*. Em função de parâmetros, *getEntry* pode ser bloqueante ou não e pode apagar a *entry* ou não.

Quando o cliente invoca uma das operações de leitura no *Proxy*, este envia uma mensagem para o *Handler* no servidor pedindo para executar a operação *getEntry*. O *Handler*, por sua vez, invoca a operação *getEntry* em *Space*.

Para executar *getEntry*, *Space* interage muitas vezes com *Repository*. A seqüência das operações está listada abaixo.

1. *Space* pede um *lock* para *Repository*, usando a operação *getLock*.
2. Se o parâmetro de *getEntry* referente à transação não for *null*, *Space* confere em *Repository* se a transação está registrada. Em caso negativo, *Space* toma as providências. Por exemplo, se a transação não estiver registrada, o gerenciador de transações deverá ser acionado para filiação, etc.
3. *Space* obtém os grupos em que a *entry* pode potencialmente ser encontrada usando a função *getGroupForClassAndSubclasses* de *Repository*.

4. *Space* busca a *entry* nos grupos, um a um, começando pelo grupo referente à classe do gabarito. O método de *Repository* usado na busca é *findEntry*. Essa busca é pelas *entries* livres, ou seja, aquelas que não estão envolvidas em alguma transação.

5. Se alguma *entry* é encontrada, então seus dados são extraídos do *Repository* com a operação *getFields* para serem repassados como retorno da função. Mas, antes, se a operação de leitura envolvia remoção da *entry*, a operação *removeEntry* é aplicada. No fim de tudo, o *lock* é devolvido usando a operação *releaseLock*.

6. Se nenhuma *entry* livre foi encontrada, repete-se a busca do passo 4, mas dessa vez o alvo são as *entries* envolvidas em transações.

7. Se alguma *entry* envolvida em transação é encontrada, o gabarito é armazenado no *Repository* (usa-se para isso a operação *insertTemplate*), o *lock* é liberado e operação *getEntry* retorna com a opção de espera (essa opção vai chegar ao *Proxy*, que bloqueia).

8. Se nenhuma *entry* envolvida em transação é encontrada na busca do passo 6 e se a leitura em curso não é bloqueante, então libera-se o *lock* e um *null* é retornado.

9. Mas se a operação em curso é bloqueante, o gabarito é armazenado, o *lock* liberado e a operação *getEntry* retorna uma opção de espera.

O *Proxy*, quando recebe uma opção de espera, fecha os *sockets* abertos e abre uma escuta por requisições numa dada porta na máquina cliente. O nome da máquina cliente e a porta de escuta foram passadas para o servidor junto com os dados do gabarito, no início da operação de leitura, através da estrutura de dados *Callback*.

A operação de escrita (*write*), quando invocada pelo cliente no *Proxy*, dispara uma mensagem para o *Handler* no servidor juntamente com parâmetros. O *Handler* invoca em *Space* uma operação de mesmo nome (*write*). É uma operação mais simples que as de leitura. A *entry* simplesmente é armazenada no *Repository* através da operação *insertEntryAndMark*, depois que o *lock* é obtido, a transação processada e o grupo obtido de *Repository*. Mas, depois que a *entry* é inserida e antes do *lock* ser liberado para encerrar a operação, acontece uma etapa crucial em todo o funcionamento do JuspSpaces – é feita uma pesquisa por gabaritos que estão esperando por uma *entry* como aquela que foi inserida. Esta busca ocorre no grupo de gabaritos que combina com a *entry* e em todos os grupos correspondentes às classes ancestrais dela. Quando a *entry* é comparada com gabaritos de classes ancestrais,

alguns de seus campos precisam ser desconsiderados, pois o gabarito, como pertencente a uma classe ancestral, pode não possuir aqueles campos. Primeiramente, os gabaritos gerados por operações com remoção são procurados. Assim, se um deles for encontrado, a busca se encerra e o *Proxy* bloqueado é acionado por *callback*. O gabarito e a *entry* são apagados. Mas, se nenhum desses existir, busca-se por gabaritos deixados por operações de leitura ou notificação. Para cada gabarito dessa natureza, o *Proxy* bloqueado é acionado por *callback*. Só os gabaritos deixados por operações de leitura são apagados depois do *callback*; os de notificação devem ter o número de evento incrementado.

5.6 Trabalhos Relacionados

O *Outrigger* é a implementação de referência para a especificação JavaSpaces fornecida pela Sun. As técnicas empregadas no *Outrigger* não foram estudadas para efeito do presente trabalho. No entanto, sabe-se que há uma visão geral da implementação do *Outrigger* no WWW[18]. Segue-se uma discussão dos aspectos externos do *Outrigger*.

O *Outrigger* vem em duas versões: uma persistente e outra transiente. A versão persistente depende de um software chamado *PSE Pro*, que é produzido pela *eXcelon*. Embora o *Outrigger* possa ser usado gratuitamente, o PSE Pro, não. O usuário do *Outrigger* deve adquirir o PSE Pro se precisar compilar o código da Sun. Essa é uma das justificativas para existir uma versão transiente do *Outrigger*, além de um melhor desempenho em aplicações que podem dispensar a característica de persistência.

O PSE Pro é uma versão leve e enxuta do *ObjectStore* [Lamb et al., 1991], um sistema gerenciador de bancos de dados baseado em objetos. Uma característica importante do ObjectStore é seu bom desempenho. O *ObjectStore* usa o mecanismo de memória virtual do sistema operacional para colocar os objetos em disco. Com isso, consegue-se persistência com tempos da mesma ordem de grandeza de um tempo de chamada ao sistema operacional. O *ObjectStore* dispensa o uso de tabelas e da linguagem SQL, tornando muito mais natural a persistência de objetos. Essa abordagem influenciou o JDO, um novo padrão para persistência de objetos proposto pela Sun à comunidade Java.

[18] http://www.cdegroot.com/cgi-bin/jini?AboutOutriggerImplementation

Autevo (anteriormente chamado de *IntaSpaces*) da IntaMission é uma implementação comercial da especificação JavaSpaces cujo diferencial é suportar mudanças de versão das classes armazenadas no espaço de tuplas. Um simples objeto pode ser convertido de uma versão para outra dinamicamente. Desta forma, versões diferentes de uma classe podem coexistir no espaço sem problemas.

O *GigaSpaces* (anteriormente conhecido como *J-Spaces*) foi a primeira implementação comercial da implementação JavaSpaces a surgir. Desde o início, seus desenvolvedores se propuseram a oferecer mais benefícios que o *Outrigger*. Várias versões depois, o *GigaSpaces* incorporou dezenas de características novas, nos mais diversos aspectos, e isso o tornou relativamente complexo. Uma das linhas de inovação é a API. O *GigaSpaces* acrescenta operações com coleções de *entries* (*readMultiple, takeMultiple, writeMultiple, count,* etc.) e cria novos eventos que podem gerar notificações (chamados *hook points*). O *GigaSpaces* oferece também muitas ferramentas para administração do espaço de tuplas, como um *browser* para as *entries*, arquivos de configuração em XML, acesso pela linha de comando, etc. Mas a principal característica de interesse no projeto do *GigaSpaces* são os *Storage Adapters*. Com eles, o mecanismo de persistência pode ser escolhido, incluindo bancos de dados relacionais. Recentemente, uma implementação com replicação para *clusters* foi adotada, incluindo mecanismos de balanceamento de carga.

O *RDBSpace* [Arnold e Kapfhammer, 2002] é uma implementação acadêmica da especificação JavaSpace, resultado de um trabalho de graduação, cuja característica mais marcante é o uso de bancos de dados relacionais como mecanismo de persistência. No entanto, *RDBSpace* não suporta transações.

Por fim, existe o *Xtreme Space*, que também usa bancos de dados relacionais para implementar um espaço de tuplas JavaSpaces com persistência. Ele é baseado no Castor[19], que é um *framework* para armazenamento de dados em Java e XML e guarda semelhanças com o *ObjectStore*.

[19] http://castor.exolab.org/

5.6.1 Comparação entre JuspSpaces e outras implementações de JavaSpaces

Há em JuspSpaces, características que nenhuma das demais implementações de JavaSpaces apresenta:

- JuspSpaces não requer o RMI para funcionar, visto que é projetada com base em *sockets*. Para a sua execução, JuspSpaces dispensa o *daemon* rmid.
- JuspSpaces não requer o serviço de *lookup* para funcionar, visto que prevê um mecanismo alternativo de acesso ao *proxy*.

Assim como *GigaSpaces*, *RDBSpace* e *Xtreme Space*, e ao contrário do *Outrigger*, JuspSpaces adota sistemas gerenciadores de bancos de dados relacionais como mecanismo principal para implementar a persistência e a recuperação de dados.

A preocupação com o gerenciamento de versões de classes das *entries* armazenadas é um destaque de *Autevo*, mas também está presente em *GigaSpaces* e *RDBSpace*. JuspSpaces não foi projetada com essa preocupação em vista.

Outrigger e *RDBSpace* (e eventualmente as demais implementações de JavaSpaces) utilizam mecanismos de *cache* em memória para conseguir melhor desempenho. Com isso, parte do espaço estaria replicado na memória. Costuma-se empregar os termos *front-end* e *back-end* para designar, respectivamente, a réplica parcial dos dados em memória e o o conjunto completo dos dados em disco. JuspSpaces não adota tal otimização – todas as operações são feitas em disco.

JuspSpaces restringe-se estritamente à especificação de JavaSpaces, assim como *Outrigger*, *RDBSpace*, *Autevo* e *Xtreme Space*. Não há funcionalidades adicionais nem ferramentas acessórias como em *GigaSpaces*.

5.7 Protótipo e Testes

Um protótipo para o JuspSpaces foi construído concomitantemente com a elaboração do projeto. O protótipo inclui implementações para os pacotes *JuspSpace*, *AcidChoir* e *MadLeaser* totalizando 4000 linhas de código Java.

O banco de dados utilizado como mecanismo de persistência é o *PostgreSQL*. Ele foi escolhido, primeiramente, pela sua gratuidade. O PostgreSQL é um dos mais populares

bancos de dados para Linux e ele também adota a GNU GPL como modelo de licença. Além disso, o PostgreSQL apresenta vantagens técnicas sobre outros bancos de dados gratuitos, como o popular MySQL. A principal delas é que ele implementa transações. O PostgreSQL também oferece diversas facilidades para sistemas baseados em objetos, sendo enquadrado no que se tem chamado de "bancos de dados objeto-relacional", mas nenhuma dessas facilidades foi empregada em JuspSpaces, pois pretende-se que qualquer outro banco de dados relacional possa ser usado, desde que tenha um *driver* JDBC.

A configuração do PostgreSQL é a única tarefa mais extensa na configuração do JuspSpaces. É preciso criar uma base de dados e configurar as permissões de acesso da mesma. A base deve ser criada vazia. Nenhuma relação (tabela) precisa ser criada.

Um ponto importante é o controle de concorrência do PostgreSQL. Utiliza-se uma técnica chamada de *Controle de Concorrência Multiversão* [POSTGRESQL, 2002]. Ela pode ser resumida pelo seguinte trecho do manual:

> "Ao contrário de outros bancos de dados que usam *locks* para fazer controle de concorrência, o PostgreSQL garante a consistência de dados usando um modelo multiversão. Isso significa que cada transação vê uma fotografia dos dados (uma versão da base de dados) de como ela era há algum tempo atrás, a despeito do estado atual. Isso previne a transação de ver dados tornados inconsistentes pela atualização de outras transações, proporcionando isolação entre as transações.
>
> A principal diferença entre essa técnica e a técnica de *locking*, é que, nessa técnica, os *locks* de leitura não conflitam com os *locks* de escrita, de forma que a leitura nunca bloqueia a escrita e a escrita nunca bloqueia a leitura."

O PostgreSQL nunca bloqueia. Se operações não serializáveis acontecem, ele simplesmente emite uma exceção. Essa característica dificultou de sobremaneira o desenvolvimento do JuspSpaces. Se o banco de dados adotado tivesse um comportamento bloqueante, gerado pelo uso de *locks*, então teria sido mais fácil implementar a semântica de transações definida na especificação JavaSpaces. Uma rápida pesquisa em outro banco de dados – o Oracle – mostrou que ele também adota a mesma técnica do PostgreSQL.

Ao longo do desenvolvimento do protótipo, vários testes para aferir a correção do protótipo foram aplicados. Eram testes básicos, envolvendo poucas *entries*, apenas para atestar se as alterações mais recentes levavam a um comportamento esperado. No entanto, após a última etapa do desenvolvimento, que foi a reestruturação da implementação da semântica de transações em JuspSpaces, ficou pendente a realização de uma completa bateria

de testes para verificação da correção do programa. Essa é uma atividade fundamental para o protótipo, mas não é uma atividade simples. Ela requer um planejamento criterioso para ser efetiva, e foi deixada como trabalho futuro.

Foi realizado um teste de desempenho simples para comparar o desempenho do JuspSpaces com o do *Outrigger*. O teste consiste em repetir, um certo número de vezes, o seguinte procedimento:

1. Escrever uma *entry* de um só campo (operação *write*).
2. Ler a *entry* (operação *read*).
3. Remover a *entry* (operação *take*).

Esse teste foi realizado numa máquina com processador Intel Pentium II, a 266 MHz de freqüência de operação e com 64 Mbytes de memória RAM. A aplicação cliente e o servidor rodavam na mesma máquina. Observou-se o tempo de execução, variando o número de vezes que a operação foi realizada. Várias observações foram feitas para um mesmo número de *entries*. Os valores médios do tempo de execução, medidos em segundos, são mostrados na Tabela 5.1 e na Figura 5.9. Nessas medidas, desconsiderou-se o tempo extra que o *Outrigger* precisa para executar os protocolos para obtenção do *proxy*.

Número de operações	JuspSpaces	Persistent Outrigger	Transient Outrigger
50	35	15	7
200	142	29	28

Tabela 5.1 –Tempo de execução das operações no JuspSpaces e no Outrigger em segundos.

O JuspSpaces mostrou-se significativamente mais lento que o *Outrigger*. Muitas podem ser as causas desse mau desempenho. Primeiramente, o cálculo das chaves pode ter influenciado, visto que ele invoca um processo de serialização seguido do algoritmo MD5 de *hashing*. Num experimento simples, substituiu-se todo o corpo da função que calcula as chaves por uma única linha em que sempre o mesmo número é retornado. O programa assim alterado não funciona devidamente, mas para efeito de medição de desempenho é válido. O tempo obtido para 50 operações foi de 30 segundos. Outra hipótese é que o tempo para geração de conexões JDBC possa ser alto. A adoção de uma estratégia de *pool* de conexões pode favorecer, visto que reduz o número de criação de novas conexões.

Deve-se considerar também que JuspSpaces não adota um mecanismo de *cache* em memória (como faz *Outrigger*), exigino o acesso a disco em toda operação. A adoção de

sockets em vez do RMI não deveria impactar sobre o desempenho sensivelmente, mas isso não foi testado separadamente.

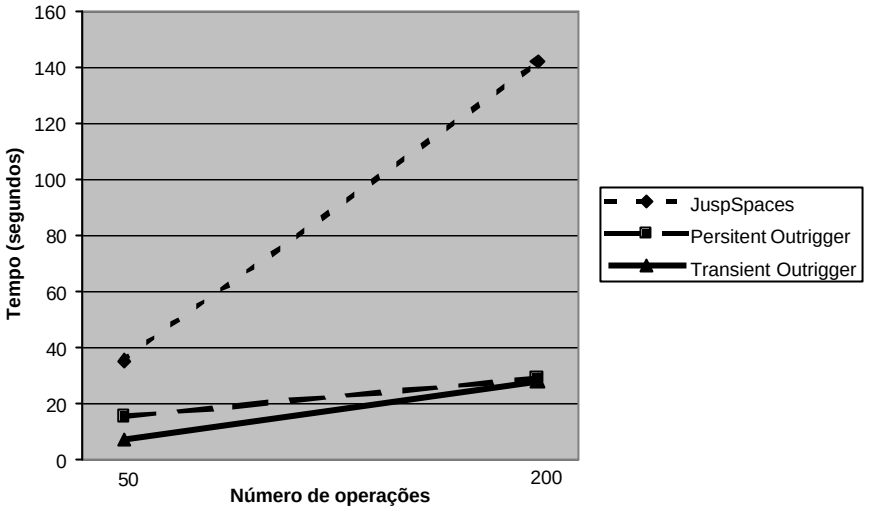

Figura 5.10 – Gráfico comparativo entre *JuspSpaces* e *Outrigger*.

O JuspSpaces foi testado ainda com uma aplicação. Trata-se do *XMLServer* [Martins, 2001], desenvolvido no ICMC-USP. O XMLServer é um servidor de documentos no formato XML que usa a linguagem XQL para receber consultas. Ele usa um JavaSpaces como mecanismo de persistência para os documentos XML que armazena. O protótipo do XMLServer usou a versão transiente do *Outrigger*. No presente trabalho, o XMLServer foi alterado para funcionar com o JuspSpaces. A única alteração necessária foi criar uma nova classe *SpaceAccessor*. Essa classe vem como um utilitário no código do livro de Freeman et al. (1999) e foi empregada na implementação do XMLServer. Ela encapsula todos os protocolos de acesso ao *proxy* do JavaSpaces. A nova classe *SpaceAccessor* invocava o *proxy* do JuspSpaces através do caminho de classe local.

Extremamente importante para o JuspSpaces é a definição de um protocolo alternativo para o *download* do *proxy*, que não seja o protocolo *default* do Jini, pois ele exige a presença de um servidor de *lookup*. No entanto, esse protocolo alternativo não foi definido, nem implementado. Durante todos os testes do protótipo, o cliente obtinha o *proxy* do próprio caminho de classe da máquina cliente, dispensando qualquer *download*. O endereço da máquina servidora deveria ser conhecido *a priori* pela aplicação cliente. Esse item fica como trabalho futuro.

5.8 Considerações finais

O projeto do JuspSpaces contempla amplamente a especificação JavaSpaces e até um pouco mais, visto que um serviço de transações distribuídas também foi implementado. A menos do serviço de *lookup* e seus protocolos, boa parte da especificação Jini foi implementada (*leasing*, eventos remotos, transações distribuídas, *proxies*). Uma característica peculiar desse projeto é que tudo foi construído usando-se apenas *sockets*, o que permite aplicar o JuspSpaces a um conjunto mais amplo de redes, incluindo a própria Internet.

Capítulo 6

Conclusões

"Projetos já nascem grávidos."

Anônimo.

6.1 Considerações iniciais

O tema "espaços de tuplas" é novo no âmbito do Laboratório de Intermídia do ICMC-USP. Sem o amparo de uma experiência anterior no uso de espaço de tuplas, o desafio inicial foi entender precisamente o conceito de espaço de tuplas, em meio a uma literatura em que até autores experientes parecem ter se confundido. Mas, com a leitura extensiva dessa mesma literatura, foi possível encontrar pontos de apoio sólidos e chegar a uma síntese dos conceitos, propriedades, vantagens, modos de uso, etc. ligados ao modelo de espaço de tuplas.

Do ponto de vista do projeto do espaço de tuplas em si, alguns objetivos foram alcançados, outros não. Devido à natureza abrangente do problema, não foi possível enfocar e esmiuçar todas as questões, que ficam como trabalhos futuros. Este capítulo termina com as contribuições (Seção 6.2), os trabalhos futuros (Seção 6.3) e as considerações finais (Seção 6.4).

6.2 Contribuições

Ao fim desse trabalho, conseguiu-se desenvolver um espaço de tuplas compatível com a especificação JavaSpaces, que se distingue de outros trabalhos correlatos por:

- Discutir a possibilidade de se implementar o acesso ao *proxy* de JavaSpaces (ou qualquer outro serviço Jini), com um mecanismo alternativo ao protocolo de *lookup*. Esse mecanismo não fere a especificação Jini.
    - Isso facilita a instalação, implantação e configuração do sistema pois o servidor de *lookup* torna-se opcional.

- Isso propicia a operação sobre redes de amplo alcance como a Internet, pois dispensa o protocolo de descoberta (geralmente associado a um *broadcasting*).

- Ser construído com softwares abertos, dos quais o principal é o banco de dados relacional *PostgreSQL*. (Bancos de dados relacionais são, além de tudo, uma tecnologia bem estabelecida.) O protótipo desenvolvido vai servir de ponto de partida para novas linhas de pesquisa e trabalhos futuros bem como servir de base para o desenvolvimento de aplicativos em geral. Ele poderá ser distribuído como software livre.

- Discutir aspectos importantes para a implementação de um espaço de tuplas persistente em Java com destaque para os aspectos listados abaixo. Cabe ressaltar que nenhuma das técnicas propostas foi particularmente comparada com outras técnicas desenvolvidas para o mesmo fim, de forma que não se pode concluir comparativamente sobre vantagens e desvantagens das técnicas apresentadas. A contribuição deste trabalho no tocante a essas técnicas limita-se ao caráter de estudo de caso.

    - Implementação de persistência de tuplas sobre bancos de dados relacionais;

    - Uso de chaves na recuperação de tuplas, bem como indexação por bases de dados relacionais;

    - Implementação do controle de concorrência e semântica transacional com bancos de dados relacionais;

    - Uso de *sockets* na implementação de um serviço Jini, em contraste com a abordagem padrão de se usar RMI.

6.3 Trabalhos futuros

- *Protocolo de acesso ao proxy.* Especificar, implementar e testar um protocolo simples que permita a uma aplicação requisitar um *proxy* diretamente ao JuspSpace, sem passar por um servidor de *lookup*. Essa idéia foi levantada, mas não implementada.

- *Testes*. Cabe planejar e executar uma suíte de testes completa, contemplando todos os aspectos importantes, para verificar o projeto e o protótipo. Alguns aspectos que merecem atenção: operações bloqueantes, influência das transações nas operações, atomicidade das operações, isolamento das transações, recuperação em caso de falha, ocorrência de *deadlocks*.

- *Validação da técnica de uso de chaves para recuperação de tuplas*. O JuspSpaces introduz o uso de chaves para registro e recuperação de campos de tuplas. É preciso comparar a técnica com outras técnicas desenvolvidas para o mesmo fim e avaliar seu impacto no desempenho da recuperação de tuplas e no tempo de transmissão de tuplas e gabaritos. Perfis de aplicações distintos devem ser considerados, pois deve haver influência de parâmetros como tamanho e quantidade dos campos. Algoritmos para obtenção de chaves a partir de objetos podem ser pesquisados, na intenção de menores tempos de cálculo da chave.

- *Mapeamento de tuplas (*entries*) em bancos de dados relacionais*. As técnicas para mapeamento de tuplas em bancos de dados relacionais empregadas neste trabalho também merecem ser comparadas. A organização geral das tabelas, os mecanismos de indexação e *locking* podem ser revistos.

- *Validação da flexibilidade da arquitetura proposta*. Uma das vantagens apresentadas para a arquitetura proposta para JuspSpaces é a flexibilidade quanto a protocolos e mecanismos de persistência. É importante implementar outros protocolos e mecanismos de persistência, especialmente em cenários em que essa característica seja de maior interesse. Por exemplo, protocolos leves e formatos otimizados de serialização podem ser diferenciais em aplicações para *clusters*.

- *Uso de JavaSpaces*. O objetivo dessa linha é verificar a adequação do modelo de espaço de tuplas e da API do JavaSpaces a problemas de computação distribuída. A metodologia básica pode ser usar o JuspSpaces na implementação de aplicações e observar questões como a capacidade do modelo de expressar algoritmos, a necessidade de novas funcionalidades (inclusive aquelas introduzidas por outros espaços de tuplas) e novas operações (algo correspondente ao *eval* e *tsc* de Linda, por exemplo). Esta linha está na seqüência natural do trabalho, porque a primeira motivação dele foi criar um espaço de tuplas simples para ser usado pelo grupo de pesquisa em agentes do Laboratório de Intermídia do ICMC-USP.

- *Melhoramento do protótipo*. O protótipo, como peça de software e aplicativo servidor, pode melhorar muito. Primeiramente, pela identificação e correção de erros e *bugs*. Em seguida, pela adoção de melhores estratégias para tratamento de exceções, gerenciamento de conexões, acesso a banco de dados, etc. Elas podem conferir ao protótipo maior confiabilidade e desempenho.

6.4 Considerações finais

Apesar de pequenos pontos pendentes, e de um desempenho comparativamente pior, o protótipo desenvolvido surge como uma alternativa real a outras implementações existentes para a implementação de JavaSpaces. Em particular, devido à simplicidade com que pode ser instalado e executado e sua flexibilidade em termos de infra-estrutura de rede.

Referências

[Ahuja et al., 1988]] AHUJA, S., CARRIERO, N, GELERNTER, D. KRISHNASWAMY, V. Matching language and hardware for parallel computation. *IEEE Transactions on Computers*. Vol. 37, n. 8, p. 921-929, 1998.

[Almasi e Gottlieb, 1994] ALMASI, G.S e GOTTLIEB, A. *Highly Parallel Computing*. Benjamin/Cummings, 1994.

[Anderson e Shasha, 1991] ANDERSON, B.G e SHASHA, D. Persistent Linda: Linda + transactions + query processing. Em Banâtre, J.P. e Métayer, D. L., editors, *Workshop on Research Directions in High-Level Parallel Programming Languages*, volume 574 das Lectures Notes in Computer Science, p. 93-109, Springer, 1991

[Arnold e Kapfhammer, 2002] ARNOLD, G., KAPFHAMMER, G.M., ROOS, R.S. Implementation and Analysis of a JavaSpace Supported by a Relational Database. Em *8th International Conference on Parallel and Distributed Techniques and Applications*. Las Vegas, 2002.

[Bakken, 2002] BAKKEN, D.E. Middleware. Capítulo de *Encyclopedia of Distributed Computing*. J. Urban e P. Dasgupta, editores, Kluwer Academic Publishers, 2002.

[Bakken e Schlichting, 1995] BAKKEN, D.E. e SCHLICHTING, R.D. Suporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, n. 3, p. 287-302, 1995.

[Bishop e Warren, 2002] BISHOP, P. e WARREN, N. *JavaSpaces in Practice*. Addison Wesley, 2002.

[Bjornson, 1992] BJORNSON, R.D., *Linda on Distributed Memory Multiprocessors*. Tese de doutorado, Universidade de Yale, 1992.

[Butcher et al. 1994] BUTCHER, P, WOOD, A., ATKINS, M. Global syncronization on Linda. *Concurrency: Practice and Experience*. Vol. 6, n. 6, p. 505-516, 1994.

[Carreira et al, 1994] CARREIRA, J., SILVA, L. SILVA, J.G. On the design of Eilean: A Linda-like library for MPI. In *IEEE 2nd Scalable Parallel Libraries Conference*, 1994.

[Cannon e Dunn, 1994] CANNON, S.R e DUNN, D. Adding fault-tolerant transaction processing to Linda. *Software – Practice and Experience*. V. 24, n. 5, p. 449-466, 1994.

[Carriero e Gelernter, 1986] CARRIERO, N e GELERNTER, D. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*. V. 4, n. 2, p. 110-129, 1986.

[Carriero e Gelernter, 1989a] CARRIERO, N e GELERNTER, D. How to write parallel programs – a guide do the perplexed. *ACM Computing Surveys*. Vol 21, n. 3, p. 323-357, 1989.

[Carriero e Gelernter, 1989b] CARRIERO, N e GELERNTER, D. Linda in context. *Communications of ACM*. Vol 32, n. 4, p. 444-458, 1989.

[Carriero e Gelernter, 1992] CARRIERO, N. e GELERNTER, D. Coordination languages and their significance. *Communications of ACM*. Vol. 35, n. 2, p. 96-107, 1992.

[CORPORATE, 1989] CORPORATE. Technical correspondence. *Communications of ACM*, Vol. 32, n. 10, p. 1241-1258, 1989

[COORDINATION, 1996] COORDINATION. *Proceedings of the First International Conference on Coordination Models, Languages and Applications*, volume 1061 na Lecture Notes in Computer Science. Springer, 1996.

[Faasen, 1991] FAASEN, C. Intermediate uniformly distributed tuple space on transputer meshes. Em Banâtre, J.P. e Métayer, D. L., editores, *Workshop on Research Directions in High-Level Parallel Programming Languages*, volume 574 em Lectures Notes in Computer Science, p. 157-173, Springer, 1991.

[Freeman et al., 1999]] FREEMAN, E., HUPFER, S., ARNOLD, K. *JavaSpaces Principles, Patterns and Practice.* Addison Wesley, 1999.

[Gelernter, 1985] GELERNTER, D. Generative communication in Linda. *ACM Transactions on Programming Language and Systems*. V. 7, n. 1, p. 80-112, 1985.

[Gelernter, 1989] GELERNTER, D. *Multiple tuple spaces in Linda*. Em E. Odijk, M. Rem, e J.-C. Syre, editores, *PARLE '89: Parallel Architectures and Languages Europe. Volume II: Parallel Languages*, volume 366 em Lecture Notes in Computer Science, p. 20-27. Springer, 1989.

[Gelernter e Bernstein, 1982] GELERNTER, D. e BERNSTEIN, A. J. Distributed communication via global buffer. Em *Proceedings of ACM Symposium on Principles of Distributed Computing*, p. 10-18, 1982.

[Java Grande Forum, 1998] JAVA GRANDE FORUM. *Making Java work for high-end computing.* Relatório técnico, 1998.

[Jacob e Wood, 2000] JACOB, J.L. e WOOD, A. A principled semantics for inp. Em A. Porto, GC Roman. *Coordination Languages and Models, 4th International Conference, COORDINATION 2000*, volume 1906 em Lecture Notes in Computer Science, p. 51-65, 2000.

[JSX, 2002] JSX. Java Serialization to XML. *Disponível em* http://freshmeat.net/projects/jsx. Visitado em outubro de 2002.

[Lamb et al., 1991] LAMB, J.C., LANDIS, G., WEINREB, D. The ObjectStore database system. *Communications of the ACM*. Vol. 34, n. 10, 1991.

[Larsen e Spring, 1999] LARSEN. J.E e SPRING, J.H. *GLOBE: Global Object Exchange – A dynamically fault-tolerant and dynamically scalable distributed tuplespace for heterogeneous, loosely coupled networks*. Tese de doutorado, Universidade de Copenhagen, 1999.

[Malone e Crowston, 1992] MALONE, T.W. e CROWSTON, K. The Interdisciplinary Study of Coordination, *ACM Computing Surveys*, Vol. 26, n. 1, p. 87-120, 1994.

[Martins, 2001] MARTINS, W.R. *Servidor de documentos XML usando Java*. Dissertação de Mestrado. ICMC-USP, 2001.

[Minsky e Leichter, 1995] MINSKY, N. AND LEICHTER, J. Law-governed Linda as a coordination model. Em Ciancarini, P., Nierstrasz, O., Yonezawa, A., editores, *Proc. of the ECOOP'94 Workshop on Object-Based Models and Languages for Concurrent Systems*, volume 924 em Lecture Notes in Computer Science, p. 125-146. Springer, 1995.

[Picco et al., 1998] PICCO, G. P., MURPHY, A. L., ROMAN, G. C., LIME: Linda Meets Mobility, Em *Proceedings of the 21st International Conference of Software Engineering (ICSE'99)*, Los Angeles, 1999.

[Pinakis, 1992] PINAKIS, JAMES. Providing Directed Communication in Linda. *15th Australian Computer Science Conference*, Hobart, 1992.

[POSTGRESQL, 2002] POSTGRESQL. *Disponível em* http://www.postgresql.org Visitado em outubro de 2002.

[Rogue Wave, 2001] ROGUE WAVE. Ruple: A Loosely Coupled Architecture Ideal for the Internet. *Disponível em* http://www.roguewave.com. Visitado em outubro de 2002.

[SCA, 2002] SCIENTIFIC COMPUTING ASSOCIATES. Home Page da SCA. *Disponível em* http://www.lindaspaces.com. Visitado em outubro de 2002.

[Sun Microsystems, 1998] SUN MICROSYSTEMS. *Java Remote Method Invocation Specification*. Disponível em http://java.sun.com Visitado em outubro de 2002.

[Sun Microsystems, 1999a] SUN MICROSYSTEMS. *JavaSpaces Service Specification*. Disponível em http://java.sun.com/products/javaspaces. Visitado em outubro de 2002.

[Sun Microsystems, 1999b] SUN MICROSYSTEMS. *Jini Technology Core Platform Specification*. Disponível em http://www.sun.com/software/jini/specs. Visitado em outubro de 2002.

[Sun Microsystems, 1999c] SUN MICROSYSTEMS. *Object Serialization Specification*. Disponível em http://java.sun.com/products/jdk/1.1/docs/guide/serialization Visitado em outubro de 2002.

[Sun Microsystems, 2002] SUN MICROSYSTEMS. *Java Data Objects*. Disponível em http://access1.sun.com/jdo. Visitado em outubro de 2002.

[Tanenbaum, 1995] TANENBAUM, A.S. *Distributed Operating Systems*. Prentice-Hall, 1995.

[Tanenbaum, 2000] TANENBAUM, A.S. *Modern Operating Systems*. 2a edição, Prentice Hall, 2000.

[Tanenbaum, 2002] TANENBAUM, A.S., MAARTEN, V.S. *Distributed Systems – Principles and Paradigms*. Prentice Hall, 2002.

[Waldo, 1996] WALDO, J. *Distributed Computing and Persistence*. Disponível em http://java.sun.com/javaone/javaone96/pres/DistComp.pdf. Visitado em outubro de 2002.

[Waldo et al, 1994] WALDO, J, KENDALL, S., WOLLRATH, A. e WYANT G. *A note on distributed computing*. Relatório técnico TR-94-29, Sun Microsystems, 1994. Disponível em http://research.sun.com/techrep/1994/abstract-29.html. Visitado em outubro de 2002.

[Waldo, 2000] WALDO, J. The end of protocols. *Disponível em* http://developer.java.sun.com/developer/technicalArticles/jini/protocols.html. Visitado em outubro de 2002.

[Wood, 1999] WOOD, A. Coordination with Attributes. P. Ciancarini e A. Wolf, editores, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594 em Lecture Notes in Computer Science, p. 21-36, Springer, 1999.

[Wyckoff et al., 1998] WYCKOFF, P., McLAUGHRY, S.W., LEHMAN, T.J., FORD, D.A. Tspaces. *IBM Systems Journal*, Vol. 37, n.3, 1998.

Apêndice A

Alterações significativas na presente versão desta dissertação (Versão corrigida)

A Banca Examinadora da presente dissertação, reunida no ICMC-USP em 6 de dezembro de 2002, sugeriu as modificações que resultaram no presente texto:

- (p. 4) Nova redação do terceiro objetivo na seção 1.2, enfatizando o uso de softwares livres.
- (p.9) Nova redação da Seção 2.3, esclarecendo apropriadamente o conceito de Linda.
- (p.33) Inversão na ordem de apresentação dos assuntos do Capítulo 4. Jini (agora Seção 4.2) passa a preceder JavaSpaces (Seção 4.3).
- (p.70) Inclusão da Seção 5.6.1 em que a comparação de JuspSpace com as demais implementações de JavaSpaces é enfatizada.
- (p.73) Consideração sobre a influência do uso de *sockets* no desempenho do protótipo na Seção 5.7.
- (p. 75) Nova redação para a Seção 6.2, reforçando os objetivos propostos na Seção 1.2.