Marco Túlio de Oliveira Valente

Mobilidade e Coordenação de Aplicações em Redes sem Fio

Tese de Doutorado apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Belo Horizonte

Abril de 2002

Resumo

Avanços recentes nas áreas de *hardware*, telecomunicações e redes de computadores estão transformando em realidade a idéia de computação móvel. Atualmente, dispositivos computacionais móveis, como *laptops*, assistentes pessoais digitais (PDAs), telefones celulares e *pagers*, são cada vez mais populares. Uma vez conectados a redes sem fio, estes dispositivos permitem a seus usuários acessar recursos e informações em qualquer lugar e a qualquer momento. No entanto, se é verdade que as tecnologias de *hardware* e telecomunicações para suportar este novo paradigma de computação encontram-se em fase de consolidação, o mesmo não pode ser afirmado sobre as tecnologias de *software*. Particularmente, linguagens, abstrações, bibliotecas e arquiteturas de *software* usadas atualmente na construção de sistemas computacionais foram projetadas para redes locais e fixas. Em geral, as mesmas não são capazes de tratar de forma adequada eventos típicos de ambientes de computação móvel, como flutuações na largura de banda da rede, desconexões voluntárias e involuntárias e mobilidade física dos dispositivos computacionais.

Assim, tendo em vista as características particulares de cenários de computação móvel, onipresente e sem fio, esta Tese de Doutorado propõe um modelo para programação de aplicações distribuídas para dispositivos computacionais móveis e um modelo para coordenação das mesmas. O objetivo central do modelo de programação proposto é suportar a construção de aplicações distribuídas tolerantes a desconexões. Basicamente, este modelo utiliza mobilidade lógica – ou, mais especificamente, mobilidade de objetos – para tratar desconexões, isto é, para lidar com um problema originado pela mobilidade física de dispositivos computacionais em redes sem fio. Já o modelo de coordenação proposto disponibiliza uma infra-estrutura que, considerando as características inerentes do meio de comunicação sem fio, suporta a realização de tarefas como comunicação entre processos, sincronização e localização de serviços. O modelo de coordenação proposto é baseado no conceito de espaço de tuplas, definido em Linda. No entanto, este modelo substitui a arquitetura cliente/servidor, tradicionalmente usada em implementações de Linda, por uma arquitetura *peer-to-peer*, argumentando que esta é mais adequada para coordenação de sistemas em redes móveis.

o presente trabalho, além de uma descrição detalhada dos modelos de programação e coordenação propostos, apresenta a semântica formal dos mesmos. Descreve-se ainda a implementação dos dois modelos em uma linguagem de programação orientada por objetos de uso geral. Por último, discute-se como os modelos propostos podem ser integrados em um modelo de computação único e apresenta-se um estudo de caso utilizando este modelo integrado.

Abstract

Recent advances in portable hardware technology, telecommunications and wireless networking are enabling the idea of mobile computing. Mobile devices, such as laptops, personal digital assistants and mobile phones are gaining increasing acceptance. Once connected to wireless networks, mobile devices enable their users to access information anytime and anywhere. In the more traditional scenario, these users rely on a base station in the fixed network to route messages to other devices. Recently, with the advent of ad hoc networks, these devices can also detach completely from the fixed infrastructure and establish transient and opportunistic connections with other devices that are in communication range.

Although the hardware and the networks to support this new paradigm of computing are already available, the same can not be said about the software technology. Particularly, programming languages and abstractions, libraries and software architectures widely used nowadays have been designed for local and fixed networks. As result, they do not handle properly events that are typical of mobile computing settings, like bandwidth fluctuations, voluntary and involuntary disconnections and physical mobility of the nodes of the network.

Considering the peculiar characteristics of ubiquitous, embedded and wireless computing, this PhD Thesis presents a programming model and a coordination model for mobile computing systems. The programming model presented supports the construction of distributed applications robust to disconnections. The model relies on the notion of logical mobility – or, more specifically, mobile objects – to handle disconnections raised by the physical mobility of devices in wireless networks. On the other hand, the coordination model proposed in the work defines a infrastructure that supports process communication, synchronization and resource discovery in mobile networks. The coordination model is based on the notion of tuple spaces, proposed in Linda. The model, however, departs from the client/server architecture, traditionally used in Linda implementations, and push towards a peer-to-peer architecture. It is argued that this kind of architecture is more appropriate to distributed applications in mobile settings.

Besides a detailed presentation of the proposed models, the work shows their formal semantics. It is also presented the implementation of the models in a real object oriented language. Finishing the thesis, it is discussed how to integrate the programming and coordination models proposed in the work in a single computation model. A case study using this unified model is also presented.

Para Cynthia,
meus pais, Osvaldo e Vera,
minha madrinha, Maria do Rosário.

Agradecimentos

Ao professor Bigonha, pela orientação, amizade e confiança em mim depositada.

À professora Mariza e ao professor Antônio Alfredo, pela amizade e sugestões dadas durante todo o trabalho.

Ao professor Vitek, pela oportunidade concedida e pelas valiosas contribuições.

Aos professores Roberto Ierusalimschy, Silvio Meira e Wagner Meira, pela leitura criteriosa do texto e pelas sugestões realizadas.

Aos colegas do LLP, Marcelo Maia, Vladimir, Elaine, Fabio, Fernando, Fabíola e Wendell.

Aos colegas do S^3 Lab, especialmente a Bogdan, pela amizade e pelas enriquecedoras discussões sobre o trabalho.

Aos colegas de Purdue, Ronaldo, Cangussu, Hélio e, especialmente, Maurílio e Maria José.

Aos colegas da Telemig (atual Telemar), especialmente a Afonso, Alexandre, Chaim, Evandro, Luiz Henrique, Martinho, Rati, Raul e Rodrigo.

Aos colegas do BDMG, especialmente a Tadeu e Baroni.

A todos os colegas da PUC que, sendo tantos, não me arrisco a relacionar.

À PUC Minas e à CAPES, pelo apoio à realização desta tese.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Avanços recentes nas áreas de *hardware*, telecomunicações e redes de computadores transformaram em realidade a idéia de computação móvel [VV00, RL98]. Atualmente, computadores móveis, como *laptops*, assistentes pessoais digitais (PDAs), telefones celulares e *pagers*, são cada vez mais populares. Esta popularidade é justificada pelo fato de, uma vez conectados a redes sem fio, estes dispositivos permitirem a seus usuários acessar recursos e informações em qualquer lugar e a qualquer momento.

Estima-se, por exemplo, que a partir de 2005 o número de telefones celulares conectados à Internet será maior que o número de computadores pessoais [Sta01]. Por outro lado, observa-se que a Lei de Moore, segundo a qual a capacidade de processamento dos *chips* atuais dobra a cada 18 meses, vem sendo aplicada com todo vigor sobre dispositivos computacionais móveis. Atualmente PDAs, por exemplo, já possuem processadores equivalentes a um Intel 386, cujo uso era comum em micros de mesa há poucos anos atrás. De forma semelhante, celulares vêm ganhando progressivamente maior capacidade de processamento, a ponto de um número cada vez maior dos mesmos embutir uma Máquina Virtual Java e serem capazes, portanto, de executar as mais diversas aplicações desenvolvidas nesta linguagem [Sun00, RTVH01]. Assim, existe atualmente toda uma demanda visando a construção de novos sistemas adaptados a ambientes de computação móvel, onipresente e sem fio.

No entanto, se é verdade que as tecnologias de *hardware* e telecomunicações para suportar estes novos sistemas já se encontram comercialmente disponíveis no mercado, o mesmo não pode ser afirmado sobre as tecnologias de *software* [RPM00, Sat96]. O principal motivo é que a maioria das linguagens, abstrações, bibliotecas e arquiteturas de *software* usadas atualmente na construção de sistemas distribuídos foram projetadas para redes locais [Car99]. Em geral, todas elas têm como objetivo final tornar a existência da rede transparente aos programadores. Sistemas de objetos distribuídos largamente utilizados, como Java RMI [Sun98] e Corba [Obj95], por exemplo, são baseados no conceito de referência de rede, o qual permite que programadores

acessem serviços remotos com um grau de transparência similar ao existente em linguagens convencionais para acesso a serviços locais, conforme ilustrado na Figura 1.1.

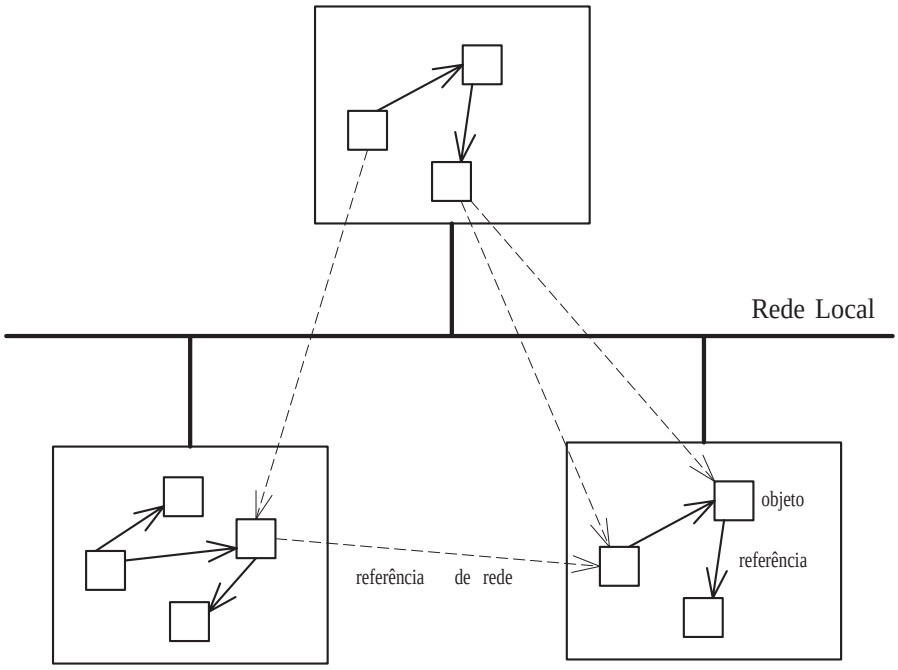


Figura 1.1: Aplicação distribuída em rede local usando referências de rede

A fim de viabilizar transparência no acesso a recursos remotos, abstrações tradicionalmente usadas em programação distribuída assumem a existência de uma rede subjacente com as seguintes características: baixa taxa de erros, largura de banda constante, topologia estática e latência reduzida. Embora todas estas premissas sejam válidas em redes locais, o mesmo não ocorre em ambientes de comunicação sem fio. Como resultado, abstrações projetadas especificamente para redes locais não são capazes de tratar adequadamente eventos típicos de ambientes móveis, como flutuações de largura de banda e desconexões. Referências de rede, por exemplo, requerem o estabelecimento de uma conexão direta entre clientes e provedores de serviço para que estes possam interagir. No entanto, em redes sem fio, nodos clientes e servidores nem sempre estão simultaneamente conectados à rede. Referências de rede também tratam desconexões como exceções e não como eventos normais em ambientes móveis. O resultado geralmente são aplicações pouco tolerantes a falhas de comunicação.

Além disso, abstrações projetadas para uso em redes locais normalmente dão origem a aplicações distribuídas no tradicional modelo cliente/servidor. Embora seja dominante atualmente, inclusive em aplicações Internet, este modelo apresenta diversas limitações quando utilizado em redes sem fio. Basicamente, aplicações cliente/servidor assumem uma associação estática entre cliente e servidor e a disponibilidade constante destes últimos. Assim, o uso deste modelo pode ser inviável em ambientes extremamente dinâmicos e flexíveis, onde nodos podem se conectar

e desconectar da rede a qualquer momento. Referências de rede, por exemplo, pressupõem que nodos clientes conhecem precisamente o endereço dos nodos provedores de serviço. No entanto, esta hipótese não é razoável em redes sem fio, já que um dispositivo móvel pode se encontrar em uma nova rede a qualquer momento.

Assim, necessita-se atualmente projetar novas abstrações para programação distribuída que sejam capazes de lidar com os problemas típicos de ambientes de computação móvel e, se possível, atenuar os efeitos dos mesmos sobre as aplicações. Foi exatamente esta constatação que motivou o desenvolvimento da presente Tese de Doutorado. No restante deste capítulo, caracteriza-se e delimita-se o problema tratado neste Trabalho de Doutorado (Seção 1.3), bem como delineia-se a solução proposta no mesmo (Seção 1.4). Antes disso, no entanto, a próxima seção descreve as características principais que distinguem redes sem fio de redes tradicionais e relaciona algumas aplicações que podem ser desenvolvidas neste tipo de rede.

1.2 Redes sem Fio

Devido a características inerentes do meio de transmissão, redes sem fio são caracterizadas por uma qualidade de comunicação inferior à disponível em redes fixas [FZ94, RL98]. Em geral, redes sem fio possuem uma largura de banda limitada. Enquanto, em redes locais a banda passante já é da ordem de gigabits por segundo, em redes sem fio a mesma alcança, no máximo, alguns megabits por segundo, como nas redes celulares de terceira geração que começam a ser utilizadas comercialmente [Sta01]. Redes sem fio também possuem uma maior taxa de erros do que redes com cabos. Com isso, o número de retransmissões é maior nestes ambientes, o que aumenta a latência observada nas comunicações. Por fim, redes sem fio são sujeitas a desconexões voluntárias e involuntárias. Desconexões voluntárias são comandadas por usuários para reduzir custos de comunicação sem fio ou para economizar bateria de dispositivos computacionais móveis. Já desconexões involuntárias e intermitentes ocorrem devido a interferências nas comunicações e à existência de áreas não cobertas pelo sinal de rádio.

Redes sem fio, também chamadas de redes móveis neste trabalho[1], são classificadas em redes infra-estruturadas ou redes *ad hoc* [Gio02, Per00, CL99].

Redes infra-estruturadas são aquelas em que os nodos móveis se conectam a uma estação de suporte a mobilidade (ou ponto de acesso), localizada na rede fixa tradicional. A Figura 1.2 ilustra uma rede móvel infra-estruturada. Neste tipo de rede, cabe à estação de suporte a mobilidade controlar o fluxo de mensagens de e para dispositivos computacionais móveis. Assim, estas redes tem funcionamento semelhante às redes de telefonia celular, onde toda comunicação é mediada por estações de rádio base conectadas a uma central telefônica. Uma estratégia comum

[1] Em uma definição rigorosa, redes sem fio não necessariamente precisam ser móveis [Tan96]. Por exemplo, uma rede local sem fio pode conectar computadores de mesa em um prédio histórico, com limitações para passagem de cabos. No entanto, na abordagem deste trabalho, considera-se apenas o caso em que redes sem fio são usadas para conectar dispositivos computacionais móveis.

no projeto de sistemas para este tipo de ambiente consiste em mover para a rede fixa parte da lógica da aplicação. Desse modo, a carga de computação nos nodos móveis e os efeitos causados por falhas de comunicação são reduzidos.

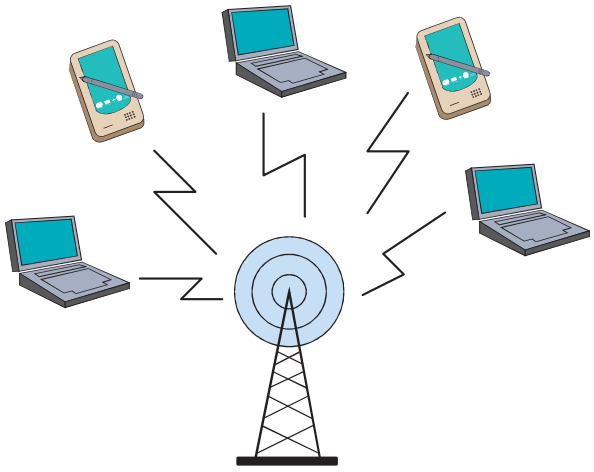


Figura 1.2: Rede móvel infra-estruturada

Redes móveis *ad hoc*[2] são redes que prescindem totalmente de uma infra-estrutura física para funcionarem. Assim, toda comunicação deve utilizar apenas o meio de comunicação sem fio, conforme ilustrado na Figura 1.3. Neste tipo de rede, dispositivos computacionais móveis podem trocar mensagens diretamente entre si, desde que suas interfaces de comunicação estejam uma ao alcance da outra. Além disso, nodos intermediários podem agir como roteadores, propagando mensagens entre nodos que não estejam diretamente conectados e, portanto, ampliando a cobertura da rede. Como *hosts* são autônomos e móveis, os mesmos podem entrar ou sair do alcance da rede aleatoriamente. Logo, redes *ad hoc* constituem um ambiente extremamente dinâmico e sujeito a constantes reconfigurações. Aplicações neste tipo de rede não devem assumir a existência de nenhuma estrutura centralizada, já que o acesso à mesma não pode ser sempre garantido. Além disso, dada a alta probabilidade de desconexões, as aplicações devem estar preparadas para continuarem operando, mesmo que parcialmente, quando a rede se tornar indisponível.

Redes móveis *ad hoc* tem seu uso recomendável quando não for possível ou viável a disponibilização de uma rede fixa. Como exemplo, temos situações de emergência ou desastres naturais, campos de batalha e eventos temporários, como reuniões, conferências e feiras.

1.2.1 Aplicações para Dispositivos Computacionais Móveis

Como exemplos de possíveis áreas de aplicação de sistemas para computadores móveis, podemos citar as seguintes:

[2]Em inglês, *Mobile Ad hoc Network* ou, simplesmente, MANET.

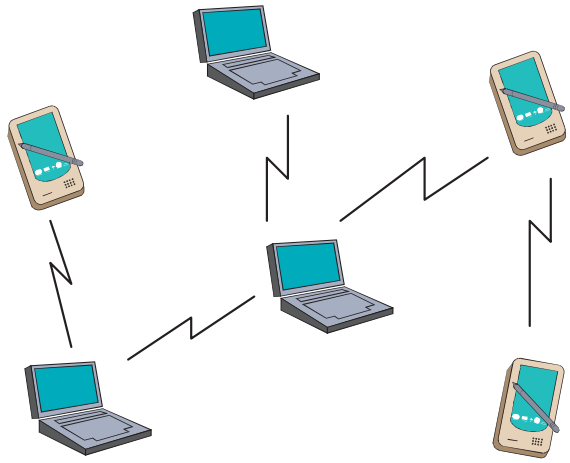


Figura 1.3: Rede móvel *ad hoc*

- Comércio eletrônico móvel, como, por exemplo, uma aplicação para permitir a compra de ingressos de cinemas ou outros eventos a partir de computadores móveis.
- Aplicações bancárias e financeiras, como, por exemplo, uma aplicação para compra e venda de ações negociadas em determinada bolsa de valores.
- Aplicações de *workflow* e *groupware*, como, por exemplo, uma aplicação para permitir que membros do comitê de programa de uma conferência possam revisar artigos utilizando seus *laptops* [Car99]. Um outro exemplo relacionado seria uma aplicação que permitisse a estes mesmos membros compartilharem entre si os formulários de avaliação de artigos durante a reunião final do comitê de programa da conferência.
- Sistemas de informação e localização, como, por exemplo, uma aplicação com objetivo de servir de "guia eletrônico" para um turista em visita a uma determinada cidade. Outro exemplo, seria uma aplicação que fornecesse informações sobre o trânsito em uma grande metrópole.
- Sistemas de entretenimento e diversão, como, por exemplo, jogos para uso em PDAs e telefones celulares.

Evidentemente, a relação acima não tem a pretensão de exaurir todas as possíveis áreas de aplicação de sistemas para computadores móveis. Na verdade, acredita-se que dada à incipiência da área, novas e relevantes aplicações ainda estão para serem propostas. Este é mais um argumento favorável à investigação de novas abstrações para programação em redes sem fio, as quais podem catalisar o surgimento destas novas aplicações.

1.3 Definição do Problema

Pretende-se investigar e defender no presente Trabalho de Doutorado a seguinte tese:

> *À medida que se migra de um ambiente tradicional e estático de programação distribuída para um ambiente de computação móvel, embutida e onipresente, o conjunto de eventos observáveis na arquitetura de rede subjacente altera-se. Passam a ser relevantes, por exemplo, eventos como flutuações na largura de banda da rede, desconexões voluntárias e involuntárias e alterações na localização física dos dispositivos computacionais. Tais eventos justificam a proposição de novos modelos e abstrações para programação distribuída em redes sem fio.*

Conforme enunciado, esta tese centra-se na argumentação de que os fenômenos típicos de ambientes sem fio são relevantes a ponto de justificarem o projeto de novas abstrações para programação distribuída, as quais podem ser incorporadas tanto a *middlewares* para computação móvel [CEM01] como a linguagens de programação [Car99]. Por outro lado, a tese não argumenta que estes fenômenos devem ser visíveis aos usuários finais de aplicações para ambientes móveis. Pelo contrário, espera-se que estes usuários possam continuar utilizando transparentemente suas aplicações ao migrarem para uma nova rede ou durante desconexões ou flutuações na largura de banda disponível. Conforme será argumentado no restante deste capítulo, os sistemas para programação distribuída existentes atualmente não foram projetados visando a construção de aplicações com este grau de mobilidade.

Considera-se que os eventos observáveis apenas em redes sem fio, mencionados na tese proposta, tornam pelo menos três questões fundamentais no projeto de sistemas distribuídos para dispositivos computacionais móveis [RPM00, Sat96, RBP01]:

- **Tolerância a Desconexões:** Em redes sem fio, é importante que usuários possam continuar utilizando suas aplicações durante falhas temporárias de conexão ou mesmo quando eles tenham voluntariamente optado por trabalhar em modo *off-line*. Esta funcionalidade é normalmente conhecida em sistemas distribuídos como operação em modo desconectado [KS92]. Como exemplo de uma aplicação onde operação em modo desconectado é um requisito importante, pode-se citar um sistema para revisão de artigos submetidos a uma conferência. Tal sistema deve permitir aos membros do comitê de programa da conferência avaliar artigos mesmo quando estejam fora do alcance da rede. Um outro exemplo é um sistema de comércio eletrônico móvel, para uso em celulares ou PDAs. Este sistema deve permitir aos usuários de tais equipamentos selecionar o produto ou serviço que desejam comprar mesmo durante desconexões voluntárias.

  Ressalte-se que, via de regra, programas desenvolvidos em linguagens de programação tradicionais não atendem ao requisito de operação em modo desconectado. *Applets* Java, por exemplo, além de dependerem da rede para obter os dados necessários à sua execução, utilizam uma política *lazy* para carregamento de código, isto é, o código de uma *applet*

é carregado dinamicamente de um servidor *Web* à medida que métodos são invocados durante sua execução [LB98]. Evidentemente, esta política não dá origem a aplicações capazes de operar em modo desconectado, já que uma falha de comunicação pode tornar o repositório de código inacessível.

- **Coordenação:** Neste termo único, estão incluídos os mecanismos que uma linguagem distribuída disponibiliza para comunicação entre processos, para localização de recursos, para sincronização de atividades etc. Particularmente em redes sem fio, coordenação é uma atividade desafiadora [RPM00]. Por exemplo, como tais ambientes estão sob constante reconfiguração, é desejável que comunicação entre processos seja possível mesmo que receptores e transmissores não estejam conectados à rede ao mesmo tempo. Localização de recursos é também mais complexa, visto que computadores podem se conectar e desconectar de uma rede a qualquer momento. Quando operando em modo *ad hoc*, esta complexidade é ainda maior, já que a inexistência de uma infra-estrutura fixa de rede inviabiliza a utilização de serviços de diretórios centralizados.

  Conforme afirmado na Seção 1.1, sistemas para construção de aplicações no tradicional modelo cliente/servidor, como CORBA ou Java RMI, não atendem a nenhum dos requisitos acima. Nestes sistemas, comunicação simplesmente falha quando uma das partes não está presente. Além disso, localização de recursos baseia-se em um diretório centralizado e previamente conhecido por todos os clientes.

- **Segurança:** Sendo redes onde novos nodos podem se conectar dinamicamente a qualquer momento, a questão de segurança em redes sem fio adquire relevância bem maior do que em redes locais. A exemplo do que ocorre na Internet tradicional, requisitos como autenticidade, privacidade e integridade nas comunicações são essenciais em alguns tipos de aplicação, como, por exemplo, naquelas de comércio eletrônico. Além disso, a utilização freqüente de tecnologias baseadas em código móvel, como *applets*, torna essencial a existência de mecanismos que impossibilitem que uma aplicação maliciosa cause danos na plataforma de execução ou em outras aplicações [Gon99].

Os problemas relacionados acima afetam diretamente a construção de aplicações distribuídas para dispositivos computacionais móveis. No entanto, quaisquer sistemas para estes dispositivos devem lidar ainda com a limitação de recursos dos mesmos, incluindo velocidade dos processadores, disponibilidade de memória e disco e tamanho das telas e dos mecanismos para entrada de dados. Um outra questão importante é o gerenciamento de energia, já que para funcionarem estes dispositivos dependem de baterias com tempo de vida limitado.

A fim de limitar o escopo do presente Trabalho de Doutorado, serão investigados nesta tese apenas os problemas relacionados com tolerância a desconexões e coordenação.

1.4 Visão Geral da Solução Proposta

A solução a ser proposta nesta Tese de Doutorado para construção de sistemas para dispositivos computacionais móveis inclui um modelo para programação destes sistemas e um modelo para coordenação dos mesmos. O modelo de programação a ser proposto define como sistemas para computadores móveis devem ser estruturados, distribuídos e executados de forma que possam atender ao requisito de operação em modo desconectado, descrito na Seção 1.3. Já o modelo de coordenação define mecanismos para comunicação, sincronização e localização de recursos em ambientes de programação distribuída sujeitos a constantes reconfigurações, como são as redes sem fio. Estes dois modelos são delineados com um pouco mais de detalhes nas subseções seguintes.

1.4.1 Modelo de Programação Proposto

O objetivo do modelo de programação a ser proposto nesta Tese de Doutorado é permitir a construção de sistemas para computadores móveis tolerantes a desconexões. A principal contribuição deste modelo é o fato de *adotar mobilidade de objetos, isto é, mobilidade lógica, para tratar um problema originado pela mobilidade física de dispositivos computacionais em redes sem fio*. Basicamente, o modelo propõe uma abstração, chamada *container*, para construção de sistemas robustos a desconexões. Um *container* é um grupo de objetos e classes que pode ser enviado pró-ativamente para execução em computadores móveis. O modelo propõe ainda uma segunda abstração, chamada *contexto*, para permitir a recepção e execução de *containers* provenientes de outros nodos da rede.

O modelo de programação incentiva, portanto, a construção de aplicações organizadas em *containers*. Como *containers* são constituídos por objetos e classes, eles possuem todo o código e também os dados necessários à sua execução. Assim, uma vez instaladas em computadores móveis, aplicações estruturadas no modelo proposto não mais dependem da rede para sua execução. Além disso, *containers* podem migrar pró-ativamente para estes computadores, isto é, sem que exista uma solicitação explícita por parte dos usuários dos mesmos. Com isso, aplicações construídas no modelo de programação proposto podem tirar vantagem de uma conexão temporária para migrarem para o computador móvel onde deverão ser executadas. Portanto, além de serem capazes de operar em modo desconectado, estas aplicações conseguem se instalar de forma transparente e oportunista nos seus computadores de destino. Esta característica é particularmente interessante em redes sem fio, dado o grau de dinamismo e assincronismo das mesmas.

O presente trabalho, além de uma descrição detalhada das abstrações propostas, inclui uma especificação formal das mesmas. Descreve-se também um sistema que implementa as abstrações propostas em Java.

1.4.2 Modelo de Coordenação Proposto

Apesar de operação em modo desconectado ser um modo de execução importante em redes sem fio, não se pode imaginar que sistemas para computadores móveis sejam totalmente auto-suficientes e adiabáticos. Em determinados momentos, estes sistemas terão que utilizar a rede para, por exemplo, efetuar uma transação em um computador remoto ou localizar um recurso, como uma máquina de fax. Assim, o modelo de coordenação a ser proposto nesta Tese de Doutorado objetiva permitir que sistemas para computadores móveis possam interagir entre si de uma forma que considere as características inerentes do meio de comunicação sem fio.

O modelo proposto baseia-se no conceito de *espaço de tuplas*, proposto pioneiramente em Linda [Gel85]. Um espaço de tuplas é um repositório de dados que pode ser manipulado por meio de primitivas que permitem inserir, ler ou remover tuplas do mesmo. Apesar de ter sido proposto originalmente para coordenação de aplicações paralelas, Linda dá origem a um modelo de coordenação com diversas características interessantes em redes sem fio. Primeiramente, comunicação é associativa, isto é, as operações de leitura e remoção de tuplas especificam um padrão (ou *template*) para a tupla desejada. Caso existam diversas tuplas no espaço que atendam a este padrão, uma delas é aleatoriamente retornada. Caso contrário, a operação permanece bloqueada. Com isso, dois processos não precisam conhecer mutuamente seus identificadores para interagirem. Além disso, comunicação é assíncrona, isto é, a operação de escrita simplesmente deposita uma tupla no espaço e as operações de leitura ficam bloqueadas enquanto não for encontrada uma tupla que atenda ao padrão especificado. Assim, dois processos podem se comunicar sem que seja necessário estabelecer uma conexão direta – como em *sockets* [Tan96] – entre os mesmos.

No entanto, a principal desvantagem de Linda em ambientes móveis é o fato de assumir a existência de um servidor centralizado, conhecido e de fácil acesso, responsável por armazenar o repositório de tuplas. Em redes infra-estruturadas, esta suposição é razoável, já que este servidor pode estar localizado na rede fixa. No entanto, quando operando em modo *ad hoc*, esta suposição passa a restringir a utilização do modelo, já que neste caso a rede fixa simplesmente não existe. Por outro lado, nenhum nodo de uma rede *ad hoc* é um bom candidato para centralizar o armazenamento do espaço de tuplas, já que desconexões podem facilmente isolar qualquer nodo do restante da rede.

Assim, a principal contribuição do modelo de coordenação proposto é o fato de *abandonar o tradicional modelo cliente/servidor, usado em implementações centralizadas de Linda, por um modelo peer-to-peer, o qual é mais adequado para coordenação de sistemas em redes sem fio e, notadamente, em redes ad hoc.* Basicamente, o modelo proposto assume que todo nodo da rede possui seu próprio espaço de tuplas, sendo que este pode ser acessado tanto por aplicações locais a este nodo, como por aplicações remotas. Em relação às primitivas tradicionais de Linda, o modelo proposto adiciona uma nova primitiva para localização de recursos. Apesar de possuir como escopo todos os espaços de tuplas da rede, a implementação desta primitiva não requer

nenhuma forma de sincronização distribuída, já que isso poderia impactar no desempenho e escalabilidade da mesma.

Além de uma descrição detalhada do modelo de coordenação proposto, o trabalho inclui também uma especificação formal do mesmo. Descreve-se ainda a implementação do protótipo de um sistema que emula o funcionamento do modelo em uma rede fixa.

1.5 Organização do Texto

O restante desta Tese de Doutorado encontra-se organizado conforme descrito a seguir:

- O Capítulo 2 contém uma revisão bibliográfica sobre o emprego de mobilidade em diversas linguagens de programação. São descritas linguagens e sistemas que utilizam duas formas distintas de mobilidade: mobilidade de código e mobilidade de agentes. O objetivo é mostrar que nenhuma destas linguagens incorpora um modelo de programação e de coordenação adequados a ambientes de computação móvel. Finalizando o capítulo, são apresentados dois cálculos de processos onde mobilidade desempenha um papel central: o π-Cálculo [MPW92, Mil99] e o Cálculo de Ambientes [Car99, CG98]. Este último será utilizado na formalização dos modelos de programação e de coordenação propostos nos capítulos seguintes.

- O Capítulo 3 propõe um modelo para programação de aplicações móveis tolerantes a desconexões. Inicialmente, apresenta-se informalmente as abstrações e a arquitetura de *software* propostas neste modelo. Em seguida, mostra-se a semântica formal destas abstrações em Cálculo de Ambientes. Descreve-se também um sistema, chamado Jamp, que implementa em Java o modelo de programação proposto. Encerrando o capítulo, compara-se este modelo com trabalhos similares e relacionam-se as principais contribuições do mesmo.

- O Capítulo 4 descreve e analisa os principais modelos de comunicação e de coordenação disponíveis em linguagens de programação distribuídas. São descritos dois tipos de modelos: modelos baseados em objetos distribuídos e modelos baseados em espaços de tuplas. Dentre os modelos baseados em objetos distribuídos, ênfase especial é dada a Jini [Arn00, Wal99], já que este sistema foi originalmente proposto para redes espontâneas, como é o caso de redes sem fio. Já dentre os modelos baseados em espaços de tuplas, analisa-se com mais detalhes o sistema Lime [PMR99, Mur00, MPR01], já que o mesmo foi projetado tendo redes móveis *ad hoc* como alvo. Argumenta-se que o uso de Jini é adequado apenas em redes infra-estruturadas. Já o modelo proposto em Lime possui um custo de operação considerável, o qual impacta o desempenho e a escalabilidade de aplicações construídas usando-se este sistema.

- O Capítulo 5 propõe um modelo baseado em espaços de tuplas para coordenação de aplicações em redes sem fio. O uso deste modelo, chamado PeerSpaces, é particularmente

interessante em redes *ad hoc*, já que o mesmo utiliza uma arquitetura descentralizada e *peer-to-peer*. Inicialmente, são descritos os principais conceitos e primitivas propostos em PeerSpaces. Em seguida, mostra-se a semântica formal do modelo. Descreve-se também o protótipo de uma implementação de PeerSpaces em Java. Encerrando o capítulo, compara-se o modelo proposto com trabalhos semelhantes e relacionam-se as principais contribuições do mesmo.

- O Capítulo 6 descreve como os modelos de programação e de coordenação, propostos nos Capítulos 3 e 5, podem ser integrados em um modelo único. Este modelo integrado permite a construção de aplicações tolerantes a desconexões e cujos padrões de comunicação são adequados a ambientes de computação móvel ou, mais especificamente, a ambientes de redes *ad hoc*. Finalizando o capítulo, apresenta-se um estudo de caso envolvendo o projeto de um sistema para revisão de artigos submetidos a uma conferência.

- O Capítulo 7 conclui a presente Tese de Doutorado. Realiza-se uma análise global dos modelos de programação e coordenação propostos, relacionam-se as principais contribuições da Tese e apontam-se trabalhos futuros.

Capítulo 2

Mobilidade em Linguagens de Programação

Linguagens para programação distribuída sempre se utilizaram de alguma forma de mobilidade. Desde o início, por exemplo, mobilidade de controle esteve presente em abstrações para construção de aplicações no modelo cliente/servidor. Com o surgimento da Internet e, mais recentemente, de redes sem fio, foram propostas linguagens e bibliotecas com suporte a outros estilos de mobilidade, notadamente mobilidade de código e mobilidade de agentes. Neste capítulo, descrevem-se então as principais características destes dois estilos de mobilidade. São apresentadas também as linguagens mais representativas que dão suporte aos mesmos. O objetivo é mostrar que nenhuma destas linguagens atende integralmente a todos os requisitos de uma linguagem para programação distribuída em redes sem fio, descritos na Seção 1.3. O capítulo inclui ainda uma descrição de dois cálculos de processos com suporte à mobilidade: o π-Cálculo e o Cálculo de Ambientes.

O restante do capítulo está organizado como descrito a seguir. A Seção 2.1 trata de mobilidade de código. Inicialmente, descrevem-se as motivações que levaram à proposição deste tipo de mobilidade e, em seguida, descrevem-se resumidamente as principais características de Java, a qual foi indubitavelmente a linguagem responsável por popularizar esta forma de mobilidade. A Seção 2.2 trata de mobilidade de agentes. Inicialmente, relacionam-se alguns trabalhos da área de sistemas distribuídos que inspiraram a proposição deste estilo de mobilidade. Em seguida, descrevem-se resumidamente três linguagens que suportam mobilidade de agentes: Obliq [Car94], Telescript [Whi97] e Aglets [LO98]. A Seção 2.3 inicia descrevendo o π-Cálculo [Mil99]. Este cálculo de processos inspirou o desenvolvimento do Cálculo de Ambientes [CG98], cujos principais conceitos são discutidos em seguida nesta seção. Por último, a Seção 2.4 apresenta algumas considerações finais.

Uma versão um pouco mais extensa do texto apresentado neste capítulo pode ser encontrada em [VBLB99].

2.1 Mobilidade de Código

O termo mobilidade de código é usado para caracterizar programas que trafegam por uma rede de estações heterogêneas, eventualmente cruzando diferentes domínios administrativos, e que são automaticamente executados ao atingirem uma determinada estação de destino [FPV98]. Neste trabalho, considera-se que nesta forma de mobilidade o que trafega pela rede é apenas o código do programa, esteja ele no formato fonte ou em um formato intermediário.

O conceito de mobilidade de código, na verdade, não é novo em linguagens de programação. A linguagem Postscript [Ado85], usada para descrição de serviços de impressão, é um bom exemplo. Postscript é uma linguagem de pilha cujos programas são automaticamente gerados por um *software* de editoração e então enviados para a impressora, onde ocorre a interpretação dos mesmos. Esta interpretação tem como efeito colateral a impressão do documento desejado.

Linguagens e ambientes de programação com suporte a mobilidade de código devem satisfazer particularmente aos seguintes requisitos:

- Portabilidade: devido à diversidade de arquiteturas existentes na Internet, mobilidade de código se tornaria inviável caso fosse necessário a existência de uma versão do código para cada tipo possível de arquitetura. Portanto, linguagens com suporte a código móvel devem possibilitar a geração de aplicações portáveis entre as várias arquiteturas de uma rede.

- Segurança: em um modelo que torna possível a execução de aplicações buscadas livremente na rede, não há dúvida de que segurança é uma questão das mais relevantes. Assim, sistemas com suporte a código móvel devem garantir que as aplicações não irão produzir danos no ambiente de execução, sejam eles acidentais ou maliciosos.

A Subseção 2.1.1 descreve as principais características da linguagem Java, a qual foi responsável por disseminar mobilidade de código na Internet.

2.1.1 Java

Dentre as linguagens que suportam mobilidade de código, certamente Java [GJS96, AGH00], lançada oficialmente pela Sun em 1995, foi a que alcançou maior sucesso. Contribuíram para este fato uma bem articulada estratégia de *marketing* e sua integração com a *Web*, através da possibilidade de se embutir aplicações Java, chamadas *applets*, em páginas HTML. A linguagem é fortemente tipada e possui herança simples, semântica de referência, coleta automática de lixo, tratamento de exceções, herança de interface, recursos para construção de módulos, suporte nativo a programação concorrente e reflexividade.

Programas fonte em Java são compilados para uma linguagem intermediária de pilha chamada *bytecode*. A fim de garantir independência de plataforma, esta linguagem intermediária é então interpretada por uma Máquina Virtual Java (JVM) [LY99], a qual possui implementações

para diversas arquiteturas. A JVM pode ser ainda integrada a um *browser*, viabilizando deste modo o desenvolvimento de aplicações *Web* interativas.

Recentemente, foi lançada pela Sun uma versão simplificada de Java e de suas bibliotecas para uso em dispositivos computacionais móveis, como telefones celulares, *pagers* e PDAs. Esta versão, chamada de *Java 2 Micro Edition* [Sun00, RTVH01], disponibiliza basicamente apenas um subconjunto dos pacotes padrões de Java. Assim, não são incluídos recursos como reflexividade, bibliotecas para construção de interfaces gráficas (como AWT e *Swing*), para chamada remota de métodos, para acesso a banco de dados etc. Além disso, J2ME possui uma biblioteca simplificada de *threads* e de entrada/saída. Em relação à linguagem propriamente dita, as modificações são pontuais. Basicamente, não estão disponíveis tipos de dados em ponto flutuante (como *float* e *double*) e métodos finalizadores. Além dessas simplificações, esta versão do ambiente Java inclui ainda uma máquina virtual customizada para dispositivos móveis, chamada KVM (*Kilobyte Virtual Machine*).

Apesar de não terem alcançado o mesmo sucesso de Java, outras linguagem foram projetadas na última década com suporte a mobilidade de código. Dentre elas, pode-se citar Juice [FK97] e PLAN [HKM+98].

Análise Crítica: Com o crescimento da Internet, mobilidade de código passou a receber maior atenção em linguagens de programação. O motivo é o fato de se tratar de uma solução natural para lidar com a heterogeneidade típica de ambientes abertos de rede. Particularmente no caso de redes sem fio, esta heterogeneidade é ainda maior, dada a diversidade de fabricantes de dispositivos móveis. Assim, não surpreende a iniciativa de se criar uma versão simplificada e customizada do ambiente Java, como a J2ME, para estes tipos de equipamentos. Na verdade, esta iniciativa corresponde a um retorno às origens da linguagem, visto que a mesma foi inicialmente concebida para desenvolvimento de aplicações embutidas em equipamentos eletrônicos.

No entanto, aplicações Java possuem uma série de limitações quando utilizadas em ambientes sem fio. A primeira delas deriva do fato de a linguagem adotar uma política *lazy* para carregamento de classes. *Applets*, por exemplo, buscam seu código no servidor *Web* onde foram instaladas [LB98]. Este código é carregado dinamicamente, à medida que métodos são invocados pela *applet*. No entanto, esta estratégia não é adequada em ambientes sem fio, visto que uma desconexão pode tornar o repositório de código inacessível logo após o início da execução da aplicação. Além disso, *applets* utilizam um estilo passivo de mobilidade, onde cabe aos clientes a iniciativa de requisitar a transferência e execução local da aplicação móvel. No entanto, em redes sem fio um estilo de mobilidade pró-ativo é muitas vezes interessante, pois permite a uma aplicação antecipar-se e migrar para seus clientes antes de existir uma solicitação explícita por parte dos mesmos.

2.2 Mobilidade de Agentes

Em um sentido bastante amplo, um agente é qualquer programa que realiza uma tarefa para a qual recebeu delegação de um usuário [Bra97]. É um termo normalmente empregado em diversas áreas da computação, como Inteligência Artificial, Robótica, Banco de Dados e Recuperação de Informação, dentre outras. Já agentes móveis são agentes cuja execução não necessariamente se restringe a uma única máquina, isto é, um agente móvel é um programa que autonomamente se desloca pelas diversas máquinas de uma rede a fim de melhor realizar a tarefa que lhe foi delegada [Whi97, KT98]. Quando se move de um nodo para outro da rede, o agente móvel carrega consigo não só o seu código mas também o estado corrente de sua execução.

Algumas das características principais de agentes móveis são as seguintes: habilidade para interagir e cooperar com outros agentes, autonomia no sentido de que sua execução procede com nenhuma ou pouca intervenção da entidade que o disparou, execução em diferentes plataformas de *hardware* e *software* (interoperabiliade), capacidade de responder a eventos externos (reatividade) e capacidade de se mover de uma estação para outra da rede (mobilidade) [RL98].

Atualmente, diversas áreas vêm sendo relacionadas como beneficiárias diretas deste novo modelo de comunicação. Dentre elas, podemos citar a construção de sistemas para dispositivos computacionais móveis [LO99]. A principal vantagem de agentes móveis neste tipo de sistema é o fato de propiciarem a construção de aplicações distribuídas dotadas de um grande grau de autonomia e assincronismo. Basicamente, uma vez disparado para a rede fixa, um agente se torna imune aos problemas de comunicação da rede móvel. Além de aplicações para computadores portáteis, argumenta-se que agentes móveis podem ser empregados também em outras áreas, como Comércio Eletrônico, Recuperação de Informação, Gerência de Redes de Telecomunicação, Aplicações de *Workflow* e *Groupware* e Processamento Paralelo [LO99].

O restante desta seção encontra-se organizado como descrito a seguir. Inicialmente, descreve-se na Subseção 2.2.1 os modelos de comunicação para sistemas distribuídos nos quais a idéia de agentes móveis se baseou. Em seguida, nas Subseções 2.2.2, 2.2.3 e 2.2.4 descrevem-se, respectivamente, as linguagens Obliq, Telescript e Aglets. Por último, a Subseção 2.2.5 relaciona outros sistemas para agentes móveis, além dos três descritos nas subseções anteriores.

2.2.1 Trabalhos Relacionados

Para melhor entender as razões que motivaram a proposição do modelo de agentes móveis como uma alternativa para implementação de aplicações distribuídas é importante conhecer os principais modelos de comunicação propostos para o desenvolvimento de sistemas distribuídos.

Dentre esses modelos, o mais utilizado é o Modelo Cliente/Servidor. Nesse modelo, as aplicações são estruturadas como um conjunto de processos cliente e servidor que interagem através de troca de mensagens. Mesmo sendo conceitualmente bastante simples, o Modelo Cliente/Servidor é adequado a uma grande variedade de sistemas. Prova disso é que as aplicações

tradicionais da Internet, como WWW, *mail*, *ftp* e *telnet*, são todas baseadas no mesmo.

As primeiras implementações de sistemas Cliente/Servidor tinham, no entanto, como desvantagem o fato de utilizarem primitivas de comunicação de muito baixo nível, como *send* e *receive*, as quais não fazem parte das construções que um programador tradicional de sistemas centralizados está habituado a utilizar. Para sanar este problema, foi proposto então o recurso de Chamada Remota de Procedimento (RPC) [BN84], com o qual permite-se a um processo cliente chamar um procedimento localizado em uma outra máquina da rede.

Um outro modelo de comunicação proposto para sistemas distribuídos é o de Avaliação Remota (REV) [SG90], o qual pode ser visto como uma generalização do conceito de RPC que permite também a passagem de procedimentos como parâmetros. Estes procedimentos são então avaliados remotamente, vindo daí o nome do modelo. Uma primeira vantagem de REV é sua flexibilidade, pois não se define previamente um conjunto de serviços que podem ser invocados remotamente, como acontece em RPC. Uma outra vantagem é que, para algumas aplicações, REV pode reduzir o montante de comunicação necessário à realização de uma tarefa. Esta redução é obtida graças à possibilidade de se enviar o programa que manipula um conjunto de dados até a máquina onde os mesmos estão localizados, enquanto que em RPC, por exemplo, os dados é que devem ser movidos para a máquina onde encontra-se o programa.

O modelo de agentes móveis pode ser visto como uma evolução dos modelos RPC e REV. Neste modelo, um agente migra de uma máquina a outra da rede carregando consigo seus dados, como em RPC, e o seu código, como em REV. No entanto, ao contrário destes dois modelos, um agente é autônomo, não precisando retornar imediatamente ao nodo de origem após ter sua execução finalizada no primeiro nodo que visita. Ele pode, sempre com o objetivo de executar sua tarefa, migrar por outros nodos da rede antes de retornar ao nodo de origem.

O modelo de agentes móveis, além de ser uma evolução de RPC e REV, é ainda inspirado no modelo de objetos móveis, o qual surgiu a partir da disseminação das idéias de orientação por objetos na década de 80. Nesse modelo, permite-se o envio de objetos de uma máquina para outra da rede. Diversas linguagens foram então propostas para suportar o conceito de objetos móveis, como Emerald, Distributed Smalltalk e Cantor [BST89]. Dentre elas, Emerald [JLHB88] foi uma das que alcançou maior aceitação. No entanto, ao contrário do modelo de agentes móveis, o uso de Emerald é recomendado apenas em redes locais com um número modesto de máquinas, todas elas possuindo a mesma arquitetura.

2.2.2 Obliq

A linguagem Obliq [Car94, Car95], desenvolvida nos laboratórios da DEC no início da década de 90, provê suporte para construção de aplicações distribuídas baseadas no paradigma de avaliação remota. Obliq é uma linguagem interpretada, com escopo léxico, não tipada e que possui apenas conceitos primitivos de orientação por objetos. O seu projeto teve como objetivo o desenvolvimento de uma linguagem onde recursos distribuídos pudessem ser acessados com um

grau de transparência similar ao existente em linguagens convencionais para acesso a recursos locais.

Aplicações desenvolvidas em Obliq são estruturadas como um conjunto de objetos. Não há classes, herança nem chamada dinâmica de métodos. Em Obliq, referências e procedimentos podem ser livremente transmitidos de um nodo para outro da rede. No caso específico de procedimentos, a principal novidade introduzida na linguagem em relação a outros mecanismos de execução remota é o fato de identificadores livres de procedimentos passados como parâmetros permaneceram associados a suas localizações de origem, conforme previsto nas regras de escopo léxico. Na definição da linguagem, argumenta-se que esta abordagem contribui para tornar a semântica de uma computação independente do nodo da rede onde ocorre sua execução.

Migração de objetos não é um conceito primitivo em Obliq. No entanto, pode ser implementada enviando um procedimento para execução remota, o qual cria no *site* de destino uma cópia do objeto local e retorna uma referência para a mesma. Em seguida, no *site* de origem, deve-se executar uma operação de redirecionamento dos atributos do objeto para sua cópia remota.

Análise Crítica: Obliq pode ser considerada a precursora das atuais linguagens para desenvolvimento de agentes móveis. A principal novidade introduzida pela linguagem é o conceito de escopo léxico distribuído, o qual dá origem a um modelo de programação bastante flexível. No entanto, este conceito baseia-se na idéia de *action-at-a-distance* [Car99], isto é, ele supõe que recursos remotos podem ser transparentemente acessados a qualquer momento. Apesar de ser viável em uma rede local, esta característica não é recomendável em redes sem fio, dados os problemas de conectividade típicos das mesmas.

2.2.3 Telescript

O sistema Telescript [Whi97, Gen95], desenvolvido pela General Magic em meados da década de 90, foi pioneiro na proposição do modelo de agentes móveis como uma alternativa para o desenvolvimento de aplicações distribuídas. O seu projeto teve como objetivo disponibilizar comercialmente uma infra-estrutura de *software* que induzisse o desenvolvimento de aplicações para comércio eletrônico baseadas em agentes móveis.

Telescript utiliza uma série de metáforas do mundo real para se referir a uma aplicação distribuída. A rede, em Telescript, é vista como sendo constituída por um conjunto de *places*, os quais constituem localizações virtuais capazes de receber e executar agentes móveis. Estes, por sua vez, são aplicações desenvolvidas na linguagem do sistema e com a capacidade de transferir a sua execução (código e estado) de um *place* para outro da rede. Um *engine* é o *software* responsável por disponibilizar *places* em máquinas físicas da rede e por executar os agentes que encontram-se localizados nestes *places*. A Figura 2.1 mostra a arquitetura de uma aplicação Telescript baseada em agentes móveis.

A linguagem de programação do sistema Telescript é inspirada em C++ e Smalltalk, possuin-

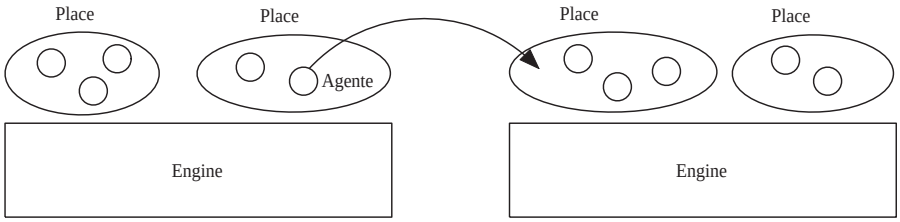


Figura 2.1: Arquitetura de uma aplicação Telescript baseada em agentes móveis

do classes, herança múltipla e tratamento de exceções. Possui ainda uma biblioteca de classes destinadas a programação de agentes móveis. Já um *engine*, além de possuir um interpretador para esta linguagem, é responsável também por gerenciar e controlar o acesso a recursos da máquina hospedeira e pela interação com outros *engines* para enviar e receber agentes móveis.

Para viajar de um *place* a outro, um agente deve executar o comando *go*, o qual tem como parâmetro um *ticket* que especifica o destino da transferência. Dois agentes podem ainda utilizar a instrução *meet* para programarem um encontro em um determinado *place*. Durante o encontro, um agente pode chamar procedimentos do outro e vice-versa.

Análise Crítica: Telescript tem o mérito de ter sido o primeiro sistema proposto para desenvolvimento de agentes móveis, servindo inclusive de inspiração para diversos outros sistemas que surgiram em seguida. O sistema oferece diversos recursos para migração de agentes, para segurança e para controle de acesso a recursos. No entanto, se adotada como uma solução para desenvolvimento de aplicações em ambientes de redes móveis, Telescript apresenta as deficiências descritas a seguir:

- Modelo de Programação: O comando *go* de Telescript implica em mobilidade completa do estado de uma *thread*, incluindo a pilha de execução e o valor do contador de programa. Apesar de bastante poderosa, mobilidade de *threads* não é de fácil implementação nos atuais ambientes de execução. É impossível, por exemplo, compilar um programa em Telescript para o formato de *bytecode*, já que a implementação atual da JVM não permite a captura do estado de *threads*. Daí o fato de Telescript usar um código intermediário próprio, chamado de *Low Telescript* [FPV98].

  Além disso, não existe na linguagem uma abstração que defina precisamente quais objetos, além do agente, são transferidos pelo comando *go*. Como regra geral, transferem-se todos os objetos que foram criados pela *thread* associada ao agente que está sendo migrado. Caso estes objetos possuam referências para objetos não incluídos na migração, as mesmas passam a ter valor *void*. Se o agente retornar posteriormente para seu *place* de origem, o valor original destas referências não é restaurado. Ou seja, a linguagem não oferece recursos para "desconexão" e "reconexão" transparente de um agente de seus possíveis *places* de execução.

- Modelo de Coordenação: Para comunicação local, Telescript oferece o comando *meet*, que permite que dois agentes localizados no mesmo *place* possam interagir por meio de chamada de procedimentos. Já para comunicação remota, existe o comando *connect*, o qual possibilita comunicação irrestrita com qualquer outro agente. Ou seja, este comando é baseado na idéia de *action-at-a-distance* e, portanto, não leva em consideração as restrições de comunicação típicas de redes sem fio.

2.2.4 Aglets

Aglets [LO98] é uma biblioteca de classes Java, desenvolvida pela IBM do Japão, para implementação de agentes móveis. O nome *aglet*, uma fusão das palavras *agent* e *applet*, designa nesta biblioteca um objeto Java capaz de se mover de um nodo para outro da Internet, carregando consigo seus atributos e métodos. A biblioteca é totalmente desenvolvida em Java, não exigindo nenhuma alteração na linguagem ou na JVM.

Os principais conceitos implementados na biblioteca Aglets são os seguintes:

- *Aglet*: objeto Java com capacidade de se deslocar por nodos da Internet. Um *aglet* é também autônomo, pois possui sua própria *thread* de execução, e reativo, já que é capaz de responder a mensagens. Todo *aglet* possui um identificador único no sistema.

- *Proxy*: é o representante local de um *aglet*. É usado para proteger o acesso aos métodos públicos do mesmo e também para prover transparência de localização, visto que é um objeto estático, enquanto que um *aglet* é móvel.

- Contexto: objeto que provê um ambiente para execução de *aglets*. Todo contexto possui um nome, o qual, quando precedido pelo nome da máquina em que executa, constitui um identificador único para o contexto.

Aglets são criados por meio das operações *create* e *clone*. Já para encerrar a execução de um *aglet* existe a operação *dispose*. A migração de um *aglet* para um outro contexto pode ser ativa ou passiva. Na modalidade de migração ativa, suportada pela operação *dispatch*, o próprio *aglet* decide migrar o estado corrente de sua execução para um novo contexto. Já na migração passiva, implementada via operação *retract*, um contexto requisita o retorno de um *aglet* remoto. Além disso, a operação *deactivate* permite que a execução de um *aglet* seja temporariamente interrompida e seu estado salvo em disco. Posteriormente, a operação *activate* possibilita que esta execução seja retomada.

Antes de cada evento ocorrido sobre um *aglet*, é invocado um método do tipo *callback*, o qual permite ao programador especificar uma ação a ser executada neste momento. Por exemplo, antes que um *aglet* seja enviado para um novo contexto, é chamado o método *onDispatch()*. No novo contexto, a execução é retomada executando-se o método *onArrival()*. Assim, grande

parte da programação em Aglets consiste na implementação de métodos *callback*.

Análise Crítica: Aglets foi um dos primeiros sistemas para desenvolvimento de agentes móveis a utilizar Java. Assim, beneficiou-se diretamente da popularidade desta linguagem. Em relação aos modelos de programação e coordenação propostos pela biblioteca podem ser feitos os seguintes comentários:

- Modelo de Programação: Não existe em Aglets nenhuma abstração para delimitar precisamente os objetos que serão transferidos em uma migração. O sistema utiliza a biblioteca de serialização padrão de Java, a qual transfere juntamente com um agente todos os objetos acessíveis a partir do mesmo. Suponha então que, devido a um erro de programação, um *aglet* passe a referenciar uma estrutura de dados de maior tamanho. Assim, ao migrar para um novo nodo, esta estrutura será integralmente transferida junto com o agente, gerando um tráfego considerável na rede.

  No momento da migração de um *aglet*, são transferidos apenas os atributos de seus objetos – e dos objetos acessíveis a partir dos mesmos, conforme já afirmado. Seus métodos são transferidos posteriormente, por demanda, à medida que se utiliza cada uma deles no nodo de destino. Esta última característica torna o sistema menos robusto a desconexões típicas de ambientes móveis, pois requer comunicação com o nodo de origem mesmo após a migração do agente.

- Modelo de Coordenação: Em Aglets, tanto comunicação local como comunicação remota se dá por meio de troca de mensagens, utilizando a operação *sendMessage*. No caso de comunicação remota, o modelo de troca de mensagens de Aglets é baseado em *action-at-a-distance* e, portanto, não leva em consideração as restrições de comunicação inerentes a redes sem fio.

2.2.5 Outras Linguagens

Além de Obliq, Telescript e Aglets, já foram propostos diversos outros sistemas, linguagens e bibliotecas para desenvolvimento de agentes móveis. Como exemplo de sistemas que utilizam Java, temos μ-Code [Pic98], JavaSeal [VB99], D'Agents [GCKR98] e Ajanta [TKV+99].

2.3 Modelos Teóricos

Linguagens de programação são geralmente estudadas por meio de modelos teóricos que capturam apenas os mecanismos essenciais de computação presentes nas mesmas. No caso específico de linguagens distribuídas, os modelos tradicionalmente usados são cálculos de processos, como CSP [Hoa78] e CCS [Mil80]. No entanto, estes dois cálculos não incorporam nenhum tipo

de mobilidade. Logo, não são adequados para formalizar o tipo de linguagem tratado neste capítulo [Car99].

Mais recentemente, no entanto, surgiram alguns cálculos de processos onde mobilidade passou a ser o elemento central das especificações produzidas. Dentre eles, destacam-se o π-Cálculo e o Cálculo de Ambientes, os quais são descritos nesta seção. Como estes dois cálculos serão usados para formalizar os modelos de programação e coordenação a serem propostos nos capítulos seguintes deste trabalho, optou-se por estender um pouco mais a apresentação realizada nesta seção.

Na Subseção 2.3.1, descreve-se o π-Cálculo, o qual foi pioneiro na incorporação de uma noção mais explícita de mobilidade. Em seguida, a Subseção 2.3.2 apresenta o Cálculo de Ambientes, proposto recentemente com o objetivo de servir de fundação para uma nova geração de linguagens para computação distribuída.

2.3.1 π-Cálculo

O π-Cálculo [MPW92, Mil99] é uma extensão do CCS proposta por Milner cuja principal novidade é a introdução da noção de mobilidade na especificação de sistemas concorrentes. No π-Cálculo, a exemplo do CCS, as duas únicas entidades disponíveis são processos e canais. Todas expressões denotam processos e computações são realizadas meramente através de troca de mensagens em canais[1]. Diferentemente do CCS, no entanto, em π-Cálculo um processo pode criar novos canais e enviar canais através de canais, isto é, dispõe-se de mobilidade de canais. Apesar de bastante simples, estes conceitos possuem um poder de expressão suficiente para descrever uma grande variedade de sistemas concorrentes e, ao mesmo tempo, preservam a capacidade de realização de raciocínios e provas formais sobre os mesmos [Pie96].

Processos em π-Cálculo são construídos de acordo com a seguinte sintaxe:

$$P, Q ::= x(y).P \mid \overline{x}y.P \mid P \mid Q \mid (\nu\, x)\, P \mid\ !P \mid \mathbf{0}$$

A Tabela 2.1 descreve o significado das expressões usadas na gramática acima.

A semântica da expressão de saída descrita na Tabela 2.1 é assíncrona. Existem ainda versões do π-Cálculo que assumem uma semântica síncrona para esta operação. Neste caso, esta expressão denota um processo que envia o valor y pelo canal x e, quando este valor for lido por algum outro processo, prossegue como se fosse o processo P.

A semântica operacional do π-Cálculo é definida por meio de reduções. Diz-se que P reduz em Q, isto é, $P \longrightarrow Q$, se P contém um conjunto de subprocessos que *podem* interagir e então se transformar nos subprocessos constituintes de Q. Veja que a semântica é não-determinística, pois especifica-se o que *pode* acontecer durante a evolução de um processo e não o que *deve*

[1] Uma analogia pode ser feita com o λ-cálculo, onde todas expressões são funções e computações são realizadas apenas através da aplicação de funções.

Expressão	Descrição
$x(y).P$	Denota um processo que espera ler um valor y de um canal x para então prosseguir como se fosse o processo P
$\overline{x}y.P$	Denota um processo que envia o valor y pelo canal x e então prossegue como se fosse o processo P
$P \mid Q$	Denota um processo composto por dois subprocessos P e Q, executando em paralelo
$(\nu\, x)\, P$	Denota um processo P com um novo canal de nome x, o qual é diferente de todos os demais nomes no escopo de P
$!P$	Denota um número infinito de cópias de P, todas executando em paralelo
$\mathbf{0}$	Denota o "processo inerte" (processo que não apresenta nenhum comportamento)

Tabela 2.1: Expressões disponíveis em π-Cálculo

acontecer. As reduções possíveis são as seguintes:

$$\begin{aligned}
\overline{x}y.P \mid x(z).Q &\longrightarrow P \mid Q\{y/z\} \\
P \mid R &\longrightarrow Q \mid R \qquad \text{se } P \to Q \\
(\nu\, x)\, P &\longrightarrow (\nu\, x)\, Q \qquad \text{se } P \to Q \\
P &\longrightarrow Q \qquad \text{se } P \equiv P' \to Q' \equiv Q
\end{aligned}$$

Na primeira redução, o termo $Q\{y/z\}$ associa y à variável z no escopo de Q. A última redução requer a existência de uma congruência estrutural entre dois processos. Esta relação, denotada por $\equiv$, define um conjunto de regras para reescrita e/ou simplificação de expressões com o propósito de viabilizar a aplicação de reduções sobre as mesmas. Como são regras que não alteram o comportamento dos processos, elas podem ser aplicadas infinitas vezes. No π-Cálculo, definem-se as seguintes regras de congruência estrutural:

$$\begin{aligned}
P \mid Q &\equiv Q \mid P && && \text{comutatividade da composição} \\
(P \mid Q) \mid R &\equiv P \mid (Q \mid R) && && \text{associatividade da composição} \\
((\nu x)P) \mid Q &\equiv (\nu x)(P \mid Q) && \text{se } x \notin fn(Q) && \text{extrusão de escopo} \\
!P &\equiv P \mid !P && && \text{replicação}
\end{aligned}$$

Na terceira regra acima, o termo $fn(Q)$ denota o conjunto de identificadores livres no processo Q.

Análise Crítica: O π-Cálculo é um modelo parcimonioso, simples e elegante para especificação de diversos tipos de sistemas concorrentes. Recentemente, surgiram algumas propostas defendendo sua utilização na especificação de sistemas móveis. Tais propostas partem do pressuposto de que o conjunto de canais de um processo é um indicativo de sua localização e que,

portanto, mobilidade pode ser descrita como uma mudança neste número de canais. No entanto, essa definição não reflete o conceito de mobilidade de programas tal como existente em agentes móveis. Nestes sistemas, mobilidade relaciona-se com uma mudança no ambiente de execução dos programas e não no número de canais de comunicação dos mesmos. Daí o surgimento de formalismos como o Cálculo de Ambientes, que tornam explícita a localização física dos processos constituintes de uma aplicação distribuída.

2.3.2 Cálculo de Ambientes

O Cálculo de Ambientes [Car99, CG98] é um cálculo de processos inspirado no π-Cálculo que tem como objetivo a especificação de dois tipos distintos de mobilidade: *mobile computation* e *mobile computing*. *Mobile computation* designa o estilo de mobilidade descrito anteriormente nesta seção, isto é, mobilidade de *software*, seja ela mobilidade de código ou de agentes. Já *mobile computing* refere-se ao tipo de mobilidade oferecido por dispositivos computacionais móveis, como *notebooks* e assistentes pessoais.

Um ambiente é um local delimitado em cujo interior acontecem computações. Todo ambiente possui um nome e uma coleção de processos e subambientes. Um ambiente pode ainda se mover para dentro ou para fora de outro ambiente. No Cálculo de Ambientes, portanto, mobilidade está relacionada com a noção de cruzar fronteiras, as quais delimitam ambientes, sendo que estes por sua vez estão organizados de forma hierárquica.

A sintaxe do cálculo, bastante similar à do π-Cálculo, é a seguinte:

$$P, Q \quad ::= \quad (\nu\, n)\, P \mid \mathbf{0} \mid P \mid Q \mid !P \mid n[\,P\,] \mid M.P \mid (x).P \mid \langle M \rangle$$

$$M \quad ::= \quad x \mid n \mid \mathbf{in}\, M \mid \mathbf{out}\, M \mid \mathbf{open}\, M \mid M.M \mid \varepsilon$$

Nesta gramática, P e Q denotam processos e n, x e M referem-se, respectivamente, a nomes, variáveis e capacidades.

Um ambiente é denotado por $n[\,P\,]$, onde n é o nome do ambiente e P é o processo em execução no seu interior. O processo P, que pode ser resultado da composição paralela de diversos outros processos, permanece em execução mesmo durante a movimentação do ambiente a que pertence. O cálculo oferece ainda operações para alterar a estrutura hierárquica dos ambientes, as quais têm sua aplicação restringida por *capacidades* (*capabilities*). A notação $M.P$ denota um processo que executa uma ação regulada pela capacidade M e então prossegue sua execução como se fosse o processo P. Existem três tipos de capacidade: para entrar (**in**), sair (**out**) e abrir (**open**) um ambiente.

A capacidade **in**, usada em uma ação da forma $\mathbf{in}\, m.P$, instrui o ambiente que a cerca a entrar em um ambiente vizinho de nome m, conforme ilustrado na Figura 2.2. Se não existir tal ambiente, a operação fica bloqueada até que o mesmo passe a existir. Se existir mais de um ambiente com este nome, um deles é escolhido aleatoriamente. A redução associada a esta

capacidade é definida pela seguinte regra:

$$n\,[\,\mathbf{in}\,m.P \mid Q\,] \mid m\,[\,R\,] \longrightarrow m\,[\,n\,[\,P \mid Q\,] \mid R\,]$$

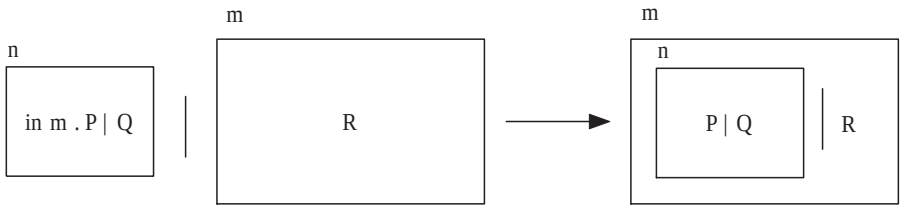


Figura 2.2: Redução *in*

A capacidade **out**, usada em ações da forma $\mathbf{out}\,m.P$, instrui o ambiente que a cerca a sair de seu ambiente pai, cujo nome deve ser m, conforme ilustrado na Figura 2.3. Se tal ambiente não se chama m, a operação fica bloqueada até que o nome do mesmo passe a ser este. A redução associada a esta capacidade é definida pela seguinte regra:

$$m\,[\,n\,[\,\mathbf{out}\,m.P \mid Q\,] \mid R\,] \longrightarrow n\,[\,P \mid Q\,] \mid m\,[\,R\,]$$

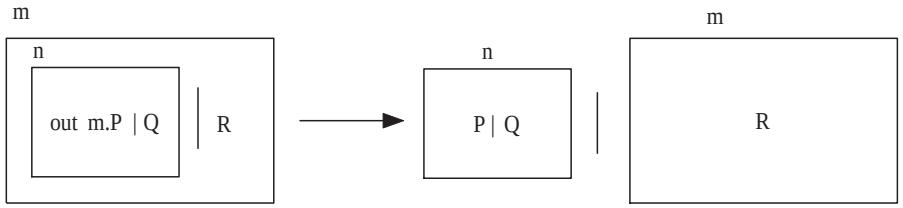


Figura 2.3: Redução *out*

Por fim, a capacidade **open**, usada em uma ação da forma $\mathbf{open}\,m.P$, abre as fronteiras do ambiente vizinho de nome m, conforme ilustrado na Figura 2.4. Como nas ações anteriores, a operação fica bloqueada até que exista um ambiente vizinho com este nome e, no caso de existir diversos ambientes com o mesmo nome, um deles é aleatoriamente escolhido. A redução associada a esta capacidade é definida pela seguinte regra:

$$\mathbf{open}\,n.P \mid n\,[\,Q\,] \longrightarrow P \mid Q$$

O Cálculo de Ambientes oferece apenas primitivas para comunicação no interior de ambientes. A operação de saída, denotada por $\langle M \rangle$, deposita assincronamente a capacidade M no ambiente local. Já a operação de entrada, denotada por $(x).P$, lê uma capacidade qualquer depositada no ambiente local e associa a mesma à variável x de P. A regra de redução associada a esta

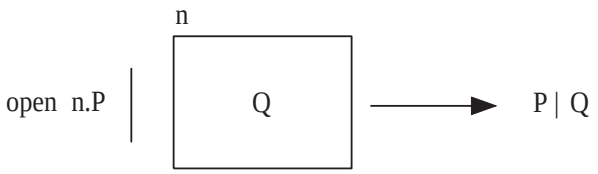


Figura 2.4: Redução *open*

operação é a seguinte:

$$(x).P \mid \langle M \rangle \longrightarrow P\{M/x\}$$

É freqüente estender o Cálculo de Ambientes com primitivas para mobilidade objetiva de ambientes. Estas primitivas possibilitam basicamente mover processos (e não ambientes) para dentro ou fora de ambientes. Mobilidade objetiva é implementada por meio do prefixo **mv**, de acordo com as seguintes regras:

$$\mathbf{mv\ in}\, m.P \mid m\,[\,R\,] \longrightarrow m\,[\,P \mid R\,] \qquad\qquad m\,[\,\mathbf{mv\ out}\, m.P \mid R\,] \longrightarrow P \mid m\,[\,R\,]$$

Existe ainda uma extensão que possibilita a um ambiente "dissolver" suas próprias fronteiras. Esta extensão é implementada por meio da primitiva **acid** e da seguinte redução:

$$n\,[\,\mathbf{acid}.P \mid Q\,] \longrightarrow P \mid Q$$

Por último, é comum o uso das primitivas descritas na Tabela 2.2 para manipular registros. A especificação destas primitivas usando o próprio Cálculo de Ambientes pode ser encontrada em [CG99].

Primitiva	Descrição
record r	Cria um registro vazio de nome r
add $r\,c\,M$	Adiciona uma célula chamada c, contendo valor inicial M, no registro r
get $r\,c\,x.P$	Associa o conteúdo da célula c, do registro r, a x no escopo de P
set $r\,c\,M.P$	Grava o valor M na célula c do registro r e continua como P

Tabela 2.2: Manipulação de registros no Cálculo de Ambientes

Análise Crítica: O Cálculo de Ambientes captura com precisão as noções de *mobile computation* e *mobile computing*. O cálculo parte do pressuposto de que existem diversas localizações na rede, sendo algumas delas virtuais e outras móveis. Localizações virtuais são erguidas, por exemplo, para proteger domínios administrativos. Já localizações móveis correspondem a dispositivos computacionais conectados, por exemplo, a uma rede sem fio. Como estas localizações possuem propriedades e recursos diferentes, é de se esperar que as aplicações sejam capazes de se mover entre elas. Todas estas noções são expressas com fidelidade no Cálculo de Ambientes.

Finalizando esta seção, a Tabela 2.3 mostra uma comparação entre o π-Cálculo e o Cálculo de Ambientes.

	π-Cálculo	**Cálculo de Ambientes**
Entidades	processos e canais	processos e ambientes
Nomes	denotam canais	denotam ambientes
Unidade de Mobilidade	canais	ambientes
Modelo Computacional	comunicação em canais	mobilidade de ambientes
Comunicação	assíncrona ou síncrona	assíncrona e local a ambientes
Localização	número de canais do processo	fronteiras de ambientes

Tabela 2.3: Comparação entre o π-Cálculo e o Cálculo de Ambientes

2.4 Considerações Finais

Este capítulo procurou mostrar que mobilidade tem desempenhado atualmente um papel essencial em linguagens para construção de aplicações distribuídas. Mostrou-se que mobilidade de código, presente em Java, constitui uma solução natural para permitir a execução de uma aplicação nas várias arquiteturas e sistemas operacionais que existem em redes sem fio. No entanto, este estilo de mobilidade por si só não é robusto aos problemas de comunicação que ocorrem nestas redes. Já mobilidade de agentes, disponível em linguagens como Obliq, Telescript e Aglets, permite o desenvolvimento de aplicações distribuídas com um maior grau de autonomia e assincronismo, o que é particularmente interessante em ambientes sem fio. No entanto, via de regra, estas linguagens não possuem mecanismos adequados para particionamento de uma aplicação em diversos agentes, para suportar operação em modo desconectado e para coordenação entre agentes. Assim, nos próximos capítulos, apresenta-se um modelo de programação e, em seguida, um modelo de coordenação que procuram suprir estas deficiências.

Capítulo 3

Mobilidade de Aplicações em Redes sem Fio

3.1 Introdução

Descreve-se neste capítulo um modelo de programação para construção de aplicações destinadas a computadores móveis [VBLB00, VBBL01a, VBBL01b]. Objetiva-se que este modelo permita a construção de aplicações distribuídas que atendam aos seguintes requisitos:

- Operação em modo desconectado: Em ambientes de comunicação sem fio, é importante que usuários possam continuar utilizando suas aplicações durante falhas temporárias de conexão ou mesmo quando eles tenham voluntariamente optado por trabalhar em modo *off-line* [Sat96]. A primeira situação é freqüente em redes sem fio, uma vez que elas são mais sujeitas a falhas de comunicação e flutuações de largura de banda do que ambientes tradicionais de comunicação. Já a segunda situação normalmente ocorre quando se deseja economizar bateria dos dispositivos computacionais móveis. Como exemplo de uma aplicação onde operação em modo desconectado é um requisito importante, pode-se citar um sistema para revisão de artigos submetidos a uma conferência. Tal sistema deve permitir aos membros do comitê de programa da conferência avaliar artigos mesmo quando não estejam conectados a uma infra-estrutura fixa de rede – por exemplo, quando estiverem utilizando um *laptop* em uma sala de espera de um aeroporto. Um outro exemplo é um sistema de comércio eletrônico móvel, para uso em celulares ou PDAs. Este sistema deve permitir aos usuários de tais equipamentos selecionar o produto ou serviço que desejam comprar mesmo durante desconexões.

- Distribuição pró-ativa de aplicações: Sistemas construídos no modelo cliente/servidor normalmente seguem um modelo *pull* para envio de informações. Basicamente, neste modelo de sistema distribuído, cabe aos clientes tomar a iniciativa de solicitar serviços providos por determinado processo servidor. No entanto, em cenários dinâmicos e sujeitos a

conexões temporárias como são as redes sem fio, é interessante a adoção de um modelo *push* para propagação de informação. Neste modelo, processos servidores são entidades ativas, as quais podem enviar dados e código para processos cliente, mesmo não tendo havido uma solicitação explícita por parte destes últimos. Argumenta-se que o modelo *push* para disseminação de informações pode tirar vantagem de conexões temporárias para enviar para processos clientes dados e códigos que serão demandados pelos mesmos futuramente. Suponha novamente como exemplo um sistema para revisão de artigos submetidos a uma conferência. É bastante desejável que este tipo de sistema seja capaz de distribuir pró-ativamente formulários de avaliação de artigos para os membros do comitê de programa desta conferência. De forma semelhante, uma aplicação servidora localizada em um *shopping center* poderia enviar para PDAs dos clientes que visitam este centro informações sobre produtos em oferta nas lojas do mesmo.

O modelo de programação a ser apresentado neste capítulo utiliza o conceito de *mobile computation* para distribuição pró-ativa de aplicações para dispositivos computacionais móveis. Proposta no Cálculo de Ambientes, *mobile computation* designa a capacidade da execução de um programa ocorrer em mais de uma máquina da rede[1] [Car99]. A idéia básica é utilizar aplicações móveis (ou mobilidade lógica) para tratar os problemas originados pela mobilidade física de computadores em um ambiente de rede sem fio. Além de serem distribuídas pró-ativamente, argumenta-se que as aplicações desenvolvidas segundo o modelo de programação proposto são capazes de operar em modo desconectado, já que são enviadas com o código e os dados necessários à sua execução.

O restante deste capítulo está organizado como descrito a seguir. A Seção 3.2 descreve informalmente o modelo de programação proposto, enfatizando seus mecanismos de mobilidade e de comunicação. Na Seção 3.3, formaliza-se as principais construções do modelo. Esta formalização é realizada por meio de uma tradução destas construções para o Cálculo de Ambientes [CG98] . Em seguida, a Seção 3.4 descreve um sistema, chamado Jamp, que implementa as abstrações do modelo proposto. Apresenta-se ainda nesta seção pequenos exemplos de utilização deste sistema. A Seção 3.5 descreve a implementação de Jamp, realizada em Java. Finalizando o capítulo, a Seção 3.6 realiza uma análise crítica do modelo e do sistema propostos, comparando os mesmos com trabalhos relacionados. Ainda na Seção 3.6, são relacionadas as principais contribuições do modelo proposto.

3.2 Modelo de Programação Proposto

O modelo de programação descrito nesta seção assume que aplicações são organizadas de acordo com os princípios básicos de orientação por objetos. Assim, considera-se que programas

[1] Existe ainda a expressão *mobile computing*, usada para designar ambientes onde computadores são dotados de mobilidade. Assim, na abordagem deste trabalho, proõe-se o emprego de *mobile computation* no projeto de sistemas para ambientes de *mobile computing*.

são constituídos por um conjunto de objetos, os quais encapsulam um estado, e por um conjunto de métodos que podem ser chamados sincronamente a partir de clientes. A principal diferença em relação a modelos orientados por objetos tradicionais é a possibilidade de se organizar objetos em grupos, os quais podem ser transferidos autonomamente de uma máquina para outra da rede.

A capacidade de se mover grupos de objetos entre nodos da rede é utilizada no modelo proposto para permitir a implementação de aplicações robustas a desconexões e que, no limite, podem inclusive operar desconectadas da rede. A idéia é disponibilizar construções que permitam mover em uma única operação todo código e os dados que uma aplicação necessita para executar em um dispositivo móvel, mesmo quando este não se encontra conectado à rede.

As subseções seguintes descrevem as construções para mobilidade e comunicação do modelo de programação proposto.

3.2.1 Mobilidade

Além de objetos, o modelo proposto utiliza duas outras construções básicas: *containers* e contextos. Um container é um grupo de objetos que podem ser transferidos autonomamente para uma outra máquina da rede. O modelo possui primitivas para criar *containers*, para inserir e remover objetos de *containers* e para mover *containers* de um nodo para outro da rede. O fato de *containers* serem móveis possibilita a construção de aplicações que podem ser enviadas pró-ativamente para dispositivos computacionais móveis e então serem executadas independentemente de a rede estar disponível ou não. Esta última característica é possível desde que os objetos do *container* contenham todos os serviços (métodos) e dados (atributos) necessários à execução remota da aplicação.

A fim de receber e executar *containers* provenientes de outros nodos da rede, computadores móveis ou fixos devem prover um contexto para esta execução. Assim, contextos são serviços disponibilizados em certos nodos da rede para recepção e execução de *containers*. Na Figura 3.1, são representadas as construções para mobilidade disponíveis no modelo de programação proposto.

Em um ambiente sem fio, no entanto, não é razoável supor que seja sempre possível a um *container* migrar diretamente de qualquer contexto A da rede para um outro contexto B. O motivo é que o contexto de destino pode estar sendo executado em um computador móvel, o qual pode estar desligado e/ou desconectado quando a migração for solicitada. A fim de tratar este problema, o modelo de mobilidade proposto pressupõe que existe uma hierarquia de dois níveis entre os contextos utilizados por uma aplicação móvel. Assim, existem dois tipos de contexto no modelo proposto: *contexto de sistema* e *contexto de usuário*.

Um contexto de sistema é um serviço capaz de receber *containers* provenientes de outros contextos da rede e então proceder de uma das seguintes maneiras: iniciar a execução do *container* recebido, chamando um de seus métodos, ou então armazenar o *container* em memória secundária. O primeiro caso é útil, por exemplo, quando o *container* representa uma tarefa

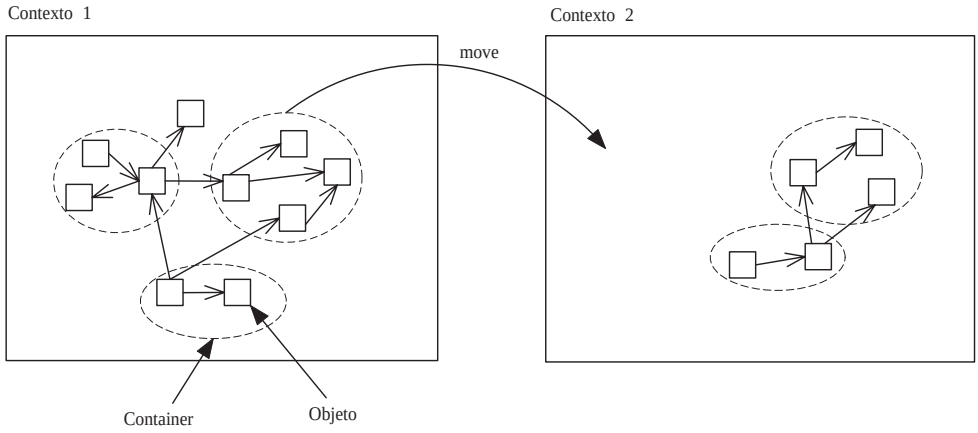


Figura 3.1: Construções para implementação de aplicações móveis

cuja execução deveria ocorrer em um computador móvel, mas que está sendo delegada a um contexto de sistema, já que estes normalmente possuem maior capacidade de processamento. Já o segundo caso é útil quando o destino final do *container* é um computador móvel. No modelo proposto, contextos de sistema foram projetados para execução em estações servidoras, isto é, estações da rede fixa que possuam alta disponibilidade e uma conexão dedicada à Internet.

Um contexto de usuário encontra-se sempre associado a um contexto de sistema e a um usuário específico da aplicação. Periodicamente, um contexto de usuário conecta-se a seu contexto de sistema e transfere do mesmo todos os *containers* que estão ali armazenados em nome de seu usuário. No modelo proposto, um contexto de usuário não necessariamente precisa estar em execução a todo momento, nem dispor de uma conexão ininterrupta à Internet. Assim, este tipo de contexto é adequado para uso em computadores móveis.

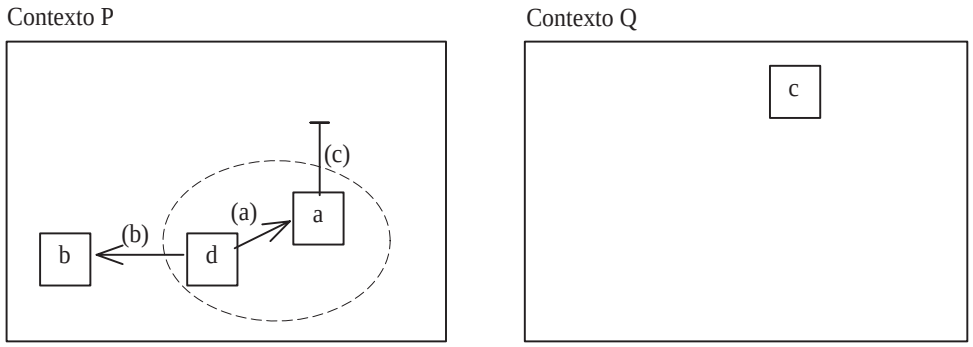
3.2.2 Comunicação Local

No modelo proposto, comunicação entre objetos localizados em um mesmo contexto ocorre por meio de chamadas de métodos, conforme usual em linguagens orientadas por objeto. No entanto, em um sistema de objetos móveis, pode ocorrer de se possuir uma referência para um objeto e o mesmo não se encontrar presente no ambiente de execução corrente. Por isso, no modelo de comunicação proposto, supõe-se que uma referência para um objeto móvel pode se encontrar em dois estados: conectada ou desconectada. O significado de cada um destes estados é descrito a seguir:

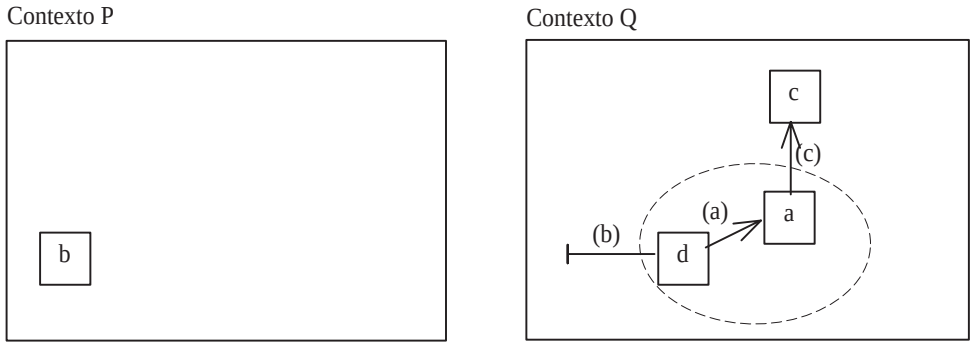
- Uma referência é dita *conectada* quando referencia um objeto de um *container* presente no contexto local.
- Uma referência é dita *desconectada* quando referencia um objeto de um *container* que não se encontra presente no contexto local.

Assim, o modelo proposto utiliza referências para identificar objetos, mas apenas referências conectadas podem ser utilizadas para se chamar métodos do objeto referenciado. Uma tentativa de se chamar um método utilizando-se uma referência desconectada produz uma exceção. Ressalte-se que uma migração pode alterar o estado de uma referência, mas nunca o objeto referenciado, isto é, a identificação do objeto referenciado é preservada mesmo quando ocorre uma migração do objeto de posse da referência. Para isso, objetos no modelo possuem um nome capaz de identificá-los univocamente em qualquer contexto da rede onde se encontrem localizados.

A Figura 3.2 ilustra o funcionamento de referências conectadas e desconectadas. Objetos são denotados nesta figura por quadrados. Na situação 1, mostra-se um *container* em execução no contexto P com três referências a, b e c, estando a e b conectadas e c desconectada. Já na situação 2 da mesma figura, mostra-se a configuração do sistema após a migração do *container* para o contexto Q. Nesta situação, a referência a permanece conectada, a referência b torna-se agora desconectada (devida à ausência do objeto referenciado em Q) e a referência c passa a estar conectada (devido à presença em Q do objeto referenciado). Finalmente, caso o *container* retorne a seu contexto de origem, a situação 1 é restabelecida.



Situação 1: Container localizado no contexto P



Situação 2: Container foi movido para o contexto Q

Figura 3.2: Exemplo de referências conectadas e desconectadas

Como pode ser visto na Figura 3.2, o modelo de comunicação proposto permite que objetos de um *container* referenciem objetos localizados em outros *containers*. Compartilhamento de objetos entre *containers* é importante para não restringir a construção de aplicações móveis. Por outro lado, a existência de referências conectadas e desconectadas, evita que este compartilhamento crie "ligações estáticas" entre *containers*, as quais poderiam vir a restringir a migração dos mesmos. Além disso, a possibilidade de se compartilhar objetos entre *containers* reduz os custos de comunicação entre os mesmos, principalmente quando comparado com uma alternativa baseada exclusivamente em uma semântica de cópia [BR00].

No Capítulo 5 será apresentado um modelo de coordenação que pode ser utilizado para interação entre *containers* localizados em contextos distintos.

3.3 Semântica Formal

Mostra-se a seguir a semântica formal das principais construções do modelo de programação proposto. Para isso, utiliza-se uma linguagem orientada por objetos bastante simples, a qual possui basicamente objetos e chamadas de métodos. Propositalmente, recursos mais sofisticados, como classes e heranças, não estão disponíveis, uma vez que estes conceitos não têm implicações sobre as construções para mobilidade e comunicação a serem formalizadas nesta seção. Uma abordagem semelhantes é adotada em [AC96], porém com o objetivo de definir um formalismo que capture os princípios essenciais de orientação por objetos.

Nesta linguagem baseada em objetos bastante simples são introduzidas as noções de *container* (grupos de objetos) e de contextos (localização física para execução de *containers*). A sintaxe da linguagem a ser formalizada é mostrada na Tabela 3.1.

$Network$	::=	**net** $n\,[SysContext \mid \ldots \mid SysContext \mid UserContext \mid \ldots \mid UserContext], loc$
$SysContext$	::=	**sys_context** $h\,[Container \mid \ldots \mid Container \mid User \mid \ldots \mid User]$
$UserContext$	::=	**user_context** $h\,[Container \mid \ldots \mid Container]$
$User$	::=	**user** u_h
$Container$	::=	**container** $c\,[Object \mid \ldots \mid Object \mid$ **start** $Code]$
$Object$	::=	**obj** $p\,[Attr \mid \ldots \mid Attr \mid Meth \mid \ldots \mid Meth]$
$Attr$	::=	$a => v$
$Meth$	::=	$m =>$ **meth**(x) $Code$ **end**
$Code$	::=	$p.m\langle x\rangle(r).Code \mid$ **return** $x \mid$ **set** $a\ v.Code \mid$ **get** $a\ x.Code \mid$
		$c.$**move**$(h) \mid c.$**move**$(user@h) \mid h.$**retrieve**(u)

Tabela 3.1: Sintaxe

Esta sintaxe permite descrever uma aplicação constituída por um conjunto de contextos, distribuídos ao longo de uma rede. Como afirmado na Seção 3.2, contextos podem ser classificados

em contextos de sistema e de usuário. Além disso, contextos contém um conjunto de *containers*, os quais agrupam um conjunto de objetos. Como usual, objetos são entidades constituídas por atributos e métodos. No corpo de um método, podem ser realizadas as operações descritas na Tabela 3.2.

Operação	Descrição
$p.m\langle x\rangle(r).Code$	Chama método m do objeto p, passando x como parâmetro; finalizada a chamada, o resultado é associado a r e a execução prossegue com $Code$.
return x	Finaliza execução do método corrente, retornando x como resultado
set a $v.Code$	Atualiza o atributo a do objeto corrente com o valor v e prossegue executando $Code$.
get a $x.Code$	Associa a x o valor do atributo a do objeto corrente e prossegue executando $Code$.
c.**move**(h)	Move *container* c para o contexto de sistema h e inicia execução do código inicial do mesmo.
c.**move**$(user@h)$	Move *container* c para o usuário $user$ do contexto de sistema h, onde o mesmo ficará armazenado, aguardando sua transferência para um contexto de usuário.
h.**retrieve**(u)	Move para o contexto de usuário corrente todos os *containers* armazenados no contexto de sistema h em nome do usuário u.

Tabela 3.2: Descrição das operações do modelo de programação

A sintaxe definida anteriormente assume que objetos são estaticamente associados a um *container*, isto é, não é possível acrescentar e remover objetos de *containers* em tempo de execução. Esta limitação contribui para tornar mais simples a especificação do modelo proposto. Como estas duas operações não são fundamentais, argumenta-se que o ganho em legibilidade e clareza na semântica compensa o inevitável prejuízo em completeza causado por esta restrição. Assim, a sintaxe acima assume que é associado a toda aplicação uma função não atualizável *loc*, a qual mapeia objetos para seus respectivos *containers*.

A semântica da linguagem é especificada por meio de uma tradução da mesma para o Cálculo de Ambientes [CG98]. Esta tradução é realizada por meio da função $\ll Code \gg$, a qual traduz um trecho de código para Ambientes. Antes de apresentar a semântica de cada construção, a Figura 3.3 descreve a hierarquia de ambientes utilizada na tradução. Esta hierarquia corresponde fielmente àquela existente na sintaxe da linguagem, o que provavelmente não ocorreria caso se utilizasse um outro cálculo de processos nesta especificação.

A semântica da linguagem proposta é mostrada nas Tabelas 3.3, 3.4, 3.5 e 3.6. Nestas tabelas, para cada construção sintática define-se uma regra de tradução. A fim de facilitar seu entendimento, toda regra é acompanhada de um texto explicativo.

Realiza-se a seguir alguns comentários sobre a aderência desta semântica à descrição do modelo realizada na Seção 3.2. O primeiro deles diz respeito à especificação de referências conectadas e desconectadas. Conforme afirmado nesta seção, a tentativa de se chamar um

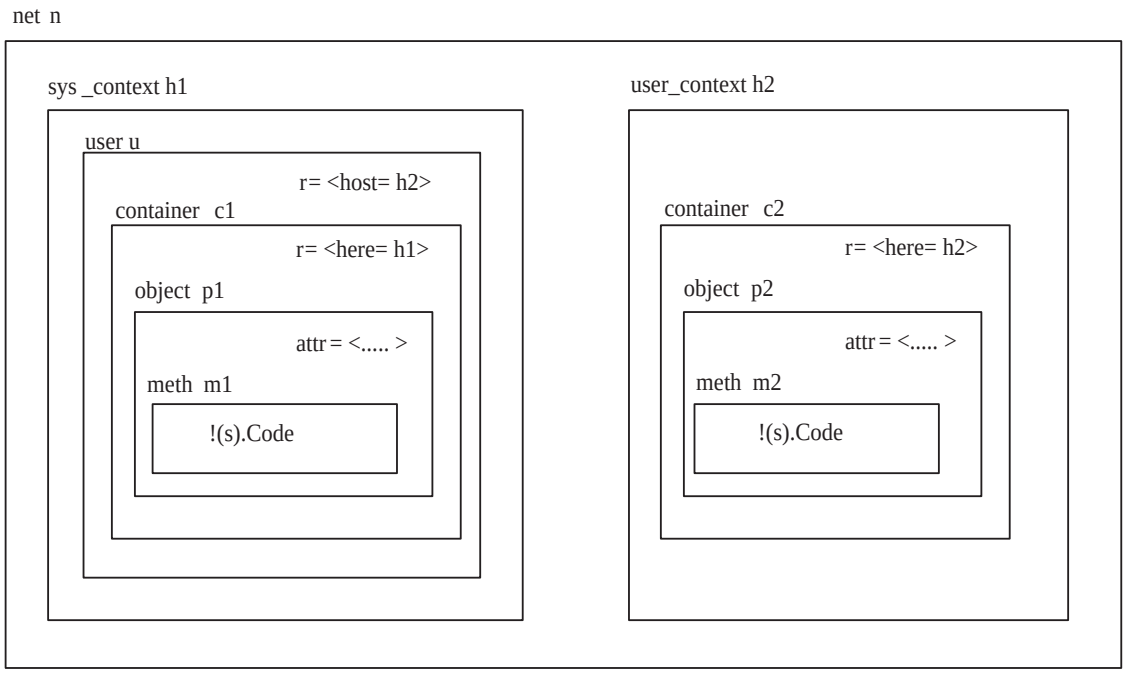


Figura 3.3: Hierarquia utilizada na tradução para o Cálculo de Ambientes

método de um objeto não presente no contexto local produz uma exceção. No entanto, conforme pode ser observado na regra (R9), uma chamada de um método pertencente a um *container* c' não disponível no contexto local é tratada com uma semântica de bloqueio na tradução desta regra. Em outras palavras, esta chamada fica bloqueada até que o *container* c' se torne disponível localmente. Esta solução foi adotada tendo em vista que ela é natural no Cálculo de Ambientes. Prova disso é que a primitiva $\mathbf{in}\, c'$ fica bloqueada neste formalismo em caso de ausência do ambiente c'. Assim, em benefício de legibilidade e clareza, decidiu-se não introduzir o conceito de exceções na semântica apresentada.

A segunda observação diz respeito ao estilo de objetos concorrentes suportado por esta especificação. Como computações são representadas por processos no Cálculo de Ambientes, optou-se por tornar objetos entidades ativas na especificação realizada. Conforme pode ser comprovado analisando-se a regra (R8), todo objeto encapsula um conjunto de processos em contínua execução, os quais são responsáveis por tratar mensagens enviadas ao mesmo. Apesar de natural em cálculos de processos, esta abordagem não é normalmente adotada em linguagens orientadas por objeto.

(*R*1) $\ll \textbf{net}\ n\,[\,SysContext \mid \ldots \mid SysContext \mid UserContext \mid \ldots \mid UserContext\,],\ loc \gg\ \equiv$
$n\,[\,\ll SysContext \gg_{loc}\ \mid \ldots \mid \ll SysContext \gg_{loc}\ \mid$
$\ll UserContext \gg_{loc}\ \mid \ldots \mid \ll UserContext \gg_{loc}\,]$

Uma rede n corresponde a um ambiente de mesmo nome com um conjunto de contextos aninhados. Estes contextos podem representar contextos de sistema ou de usuário. A função *loc* informa o *container* onde está localizado cada objeto da aplicação.

(*R*2) $\ll \textbf{sys\_context}\ h\,[\,Container \mid \ldots \mid Container \mid User \mid \ldots \mid User\,] \gg_{loc}\ \equiv$
$h\,[\,\ll Container \gg_{loc,h}\ \mid \ldots \mid \ll Container \gg_{loc,h}\ \mid \ll User \gg\ \mid \ldots \mid \ll User \gg\,]$

Um contexto de sistema h é traduzido para um ambiente de mesmo nome contendo um conjunto de *containers* e um conjunto de usuários.

(*R*3) $\ll \textbf{user}\ u_h \gg\ \equiv\ \ u\,[\,\textbf{record}\ r.\textbf{add}\ r\ host\ h\,]$

Usuários possem um identificador na forma u_h, onde u é o nome propriamente dito do usuário e h é o nome do contexto de usuário associado a u. Usuários de um contexto de sistema são traduzidos para um ambiente com o mesmo nome do usuário. Este ambiente contém um registro r em cuja célula *host* armazena-se o contexto de usuário h associado a u. Além disso, este ambiente armazenará posteriormente os *containers* enviados para o contexto de sistema em nome do usuário u.

(*R*4) $\ll \textbf{user\_context}\ h\,[\,Container \mid \ldots \mid Container\,] \gg_{loc}\ \equiv$
$h\,[\,\ll Container \gg_{loc,h}\ \mid \ldots \mid \ll Container \gg_{loc,h}\,]$

Um contexto de usuário h é traduzido para um ambiente de mesmo nome contendo um conjunto de *containers*.

(*R*5) $\ll \textbf{container}\ c\,[\,Object \mid \ldots \mid Object \mid \textbf{start}\ Code\,] \gg_{loc,h}\ \equiv$
$c\,[\,\ll Object \gg_{loc,c}\ \mid \ldots \mid \ll Object \gg_{loc,c}\ \mid$
$leader\,[\,start\,[\,!\textbf{open}\ msg.(run).\ \ll Code \gg_{loc,start,leader,c}\,]\,]\ \mid\ \textbf{record}\ r.\textbf{add}\ r\ here\ h\,]$

Um container c é traduzido para um ambiente de mesmo nome contendo um conjunto de objetos. Este ambiente também contém, aninhado nos subambientes *leader* e *start*, múltiplas cópias do código que será disparado quando o *container* for movido para um outro contexto. A execução deste código é iniciada por um ambiente de nome *msg*. O ambiente c contém ainda um registro r, em cuja célula *here* armazena-se o contexto corrente do *container*.

(*R*6) $\ll \textbf{obj}\ p\,[\,Attr \mid \ldots \mid Attr \mid Meth \mid \ldots \mid Meth\,] \gg_{loc,c}\ \equiv$
$p\,[\,\textbf{record}\ attr \mid \ll Attr \gg\ \mid \ldots \mid \ll Attr \gg\ \mid \ll Meth \gg_{loc,p,c}\ \mid \ldots \mid \ll Meth \gg_{loc,p,c}\,]$

Um objeto p corresponde a um ambiente de mesmo nome. Este ambiente possui um registro *attr*, onde serão armazenados os atributos deste objeto. Além disso, o ambiente p contém a definição dos atributos e métodos do objeto.

Tabela 3.3: Semântica do modelo de programação

(*R*7) $\ll x => v \gg \equiv$ **add** *attr* x v

Um atributo é traduzido em um código que adiciona ao registro *attr* uma célula com o nome x do atributo, contendo o valor inicial v do mesmo. O registro *attr* é aquele que foi criado pela regra (R6).

(*R*8) $\ll m => \mathbf{meth}(x)\ Code\ \mathbf{end} \gg_{loc,p,c} \equiv m\,[\,!\mathbf{open}\,msg.(x, res, P^{-1}).\ \ll Code \gg_{loc,m,p,c}\,]$

Um método m é traduzido em um ambiente de mesmo nome. Este ambiente contém múltiplas cópias do código do método. A execução deste código é disparada com o recebimento de uma mensagem (ambiente) de nome *msg*. O ambiente *msg* contém em seu interior o valor $\langle x, res, P^{-1}\rangle$, onde x é o argumento desta chamada do método, *res* é o nome do ambiente que conterá o resultado da execução do método e P^{-1} é o "endereço de retorno" do método, isto é, para onde o ambiente *res* deve ser enviado. Uma vez lidos estes três valores, tem início a execução do código do método.

(*R*9) $\ll p'.m'\langle x\rangle(r).Code \gg_{loc,m,p,c} \equiv$

$(\nu\ res)\ msg\,[\,P.\langle x, res, P^{-1}\rangle\,] \mid \mathbf{open}\,res.(r).\ \ll Code \gg_{loc,m,p,c}$

where $P = \mathbf{out}\,m.\mathbf{out}\,p.\mathbf{out}\,c.\mathbf{in}\,c'.\mathbf{in}\,p'.\mathbf{in}\,m'$, $\quad P^{-1} = \mathbf{out}\,m'.\mathbf{out}\,p'.\mathbf{out}\,c'.\mathbf{in}\,c.\mathbf{in}\,p.\mathbf{in}\,m$

$c' = loc(p')$

A chamada de um método é traduzida em um ambiente de nome *msg*. Inicialmente, este ambiente migra pelo caminho P, o qual transfere *msg* para dentro do ambiente m', associado ao método que está sendo chamado. Em seguida, deposita-se em *msg* o valor $\langle x, res, P^{-1}\rangle$, onde x é o argumento desta chamada, *res* é um novo nome criado especificamente para denotar o ambiente que retornará com a resposta e P^{-1} é o "caminho de retorno" a ser seguido por *res*. A fim de modelar chamadas síncronas, um processo paralelo permanece esperando o retorno de *res*. Quando isto ocorre, o ambiente *res* é "aberto", o resultado x armazenado no mesmo é lido e a continuação *Code* da chamada é executada.

O caminho P é utilizado para migrar sucessivamente o ambiente *msg* para fora do método corrente m, para fora do objeto chamador p, para fora do *container* corrente c e então para dentro dos ambientes c', p' e m', os quais representam, respectivamente, o *container*, o objeto e o método de destino da mensagem. O caminho P^{-1} denota o caminho de retorno.

(*R*10) $\ll \mathbf{return}\ x \gg_{loc,m,p,c} \equiv res\,[\,\langle x\rangle \mid P^{-1}\,]$

O comando **return** cria o ambiente de resposta *res*, cujo nome foi informado na chamada do respectivo método. Este ambiente armazena o resultado $\langle x\rangle$ da execução e o caminho de retorno P^{-1}, também informado quando da chamada do método. Veja que tanto *res* como P^{-1} são identificadores livres no código de um método. Quando da chamada do método (regra R9), ambos identificadores são associados ao nome do ambiente e ao caminho de retorno específicos desta chamada.

Tabela 3.4: Semântica do modelo de programação

(*R*11) $\ll \mathbf{set}\ a\ v.Code \gg_{loc,m,p,c} \equiv \mathbf{mv\ out}\, m.\mathbf{set}\, attr\, a\, v.\mathbf{mv\ in}\, m.\ \ll Code \gg_{loc,m,p,c}$

A atualização do valor de um atributo corresponde a um código que migra para fora do ambiente do método corrente, para ter acesso ao registro *attr* (definido na regra R6). Este código atualiza então a célula a deste registro com seu novo valor v. Em seguida, o código migra novamente para dentro do ambiente do método chamador e prossegue executando a continuação *Code*.

(*R*12) $\ll \mathbf{get}\ a\ x.Code \gg_{loc,m,p,c} \equiv \mathbf{mv\ out}\, m.\mathbf{get}\, attr\, a\, x.\mathbf{mv\ in}\, m.\ \ll Code \gg_{loc,m,p,c}$

A leitura do valor corrente de um atributo corresponde a um código que migra para fora do ambiente corrente, para ter acesso ao registro *attr* (definido na regra R6). Este código lê então o valor corrente da célula a deste registro e associa o mesmo ao identificador x. Em seguida, o código migra novamente para dentro do método chamador e prossegue executando a continuação *Code*.

(*R*13) $\ll c'.\mathbf{move}(h) \gg_{loc,m,p,c} \equiv$

$\mathbf{mv\ out}\, m.\mathbf{mv\ out}\, p.\mathbf{mv\ out}\, c.\mathbf{mv\ in}\, c'.\mathbf{get}\, r\, here\, h'.\mathbf{out}\, h'.\mathbf{in}\, h.\mathbf{set}\, r\, here\, h.$

$msg\,[\,\mathbf{in}\, leader.\mathbf{in}\, start.\langle run\rangle\,]$

A operação **move** é traduzida em um código que sucessivamente migra para fora do método corrente m, do objeto chamador p, do *container* corrente c e então para dentro do *container* a ser movido c'. Inicialmente em c', este código acessa o registro r para obter o contexto corrente h' deste *container*. Feito isso, o *container* c' é movido para fora de seu contexto corrente (**out** h') e, em seguida, para dentro do contexto de destino (**in** h). Atualiza-se então o novo contexto de c' no registro r. Por último, uma mensagem é enviada para disparar a execução do código inicial de c' em seu novo contexto. Esta mensagem corresponde a um ambiente *msg*, que migra para dentro dos ambientes *leader* e *start*, definidos na regra R5.

(*R*14) $\ll c'.\mathbf{move}(user@h) \gg_{loc,m,p,c} \equiv$

$\mathbf{mv\ out}\, m.\mathbf{mv\ out}\, p.\mathbf{mv\ out}\, c.\mathbf{mv\ in}\, c'.\mathbf{get}\, r\, here\, h'.\mathbf{out}\, h'.\mathbf{in}\, h.\mathbf{in}\, user.$

$\mathbf{mv\ out}\, c'.\mathbf{get}\, r\, host\, h''.\mathbf{mv\ in}\, c'.\mathbf{set}\, r\, here\, h''$

Esta versão da operação **move** é similar àquela descrita na regra R13. Algumas diferenças, no entanto, devem ser citadas. Como o *container* é enviado para um contexto de sistema, tendo como destino um dos usuários do mesmo, a primeira diferença consiste no fato de o *container* ser movido para dentro do contexto de destino h e, logo em seguida, para dentro de um ambiente que armazena os *containers* destinados ao usuário *user*. Ressalte-se ainda que, após a migração, a célula *here* do container c' passa a armazenar o nome do contexto de usuário associado a *user*. Este nome é obtido consultanto a célula *host* do registro r existente no ambiente *user*. Por último, deve-se mencionar que o código inicial do *container* c' não é executado, visto que o contexto de sistema é usado apenas para armazenar temporariamente o *container*, enquanto este não é transferido para seu contexto de usuário.

Tabela 3.5: Semântica do modelo de programação

(*R*15) $\ll h'.\mathbf{retrieve}(user) \gg_{loc,m,p,c} \equiv$

$\mathbf{mv\ out}\, m.\mathbf{mv\ out}\, p.\mathbf{get}\, r\, here\, h.\mathbf{mv\ out}\, c.\mathbf{mv\ out}\, h.\mathbf{mv\ in}\, h'.\mathbf{mv\ in}\, user.$

$\mathbf{out}\, h'.\mathbf{in}\, h.\mathbf{acid}.user\, [\mathbf{out}\, h.\mathbf{in}\, h' \mid \mathbf{record}\, r.\mathbf{add}\, r\, host\, h]$

A operação **retrieve** corresponde inicialmente a um código que se move para fora do método corrente m, do objeto chamador p, do *container* corrente c e do contexto corrente h. Logo em seguida, este código se move para dentro do contexto de sistema h' e, no próximo passo, para dentro do ambiente que representa o usuário *user* neste contexto. O código então é usado para mover o ambiente *user* para fora de h' e para dentro de h, onde então este ambiente é aberto, por meio da primitiva **acid**, liberando no contexto de usuário os *containers* contidos em seu interior. A fim de finalizar este protocolo, um novo ambiente *user* é enviado para o contexto de sistema h'.

Tabela 3.6: Semântica do modelo de programação

A última observação diz respeito a migração de *containers*. Na semântica apresentada, ao se migrar um *container* utilizando a operação **move** todos os todos os processos (ou *threads*) em execução neste *container* são migrados junto com o mesmo. Migração de *threads*, no entanto, não é um recurso comum nos atuais ambientes de programação. Assim, como será descrito na Seção 3.4, uma implementação do modelo de programação proposto pode limitar os itens migrados com um *container* ao estado e código de seus objetos.

3.4 Um Sistema para Construção de Aplicações Móveis

Descreve-se nesta seção um sistema, chamado Jamp, que implementa em Java o modelo de programação descrito e formalizado nas seções anteriores [VBBL01b]. O sistema encontra-se estruturado como um pacote com as seguintes classes: `JContainer`, `JSystemContext` e `JUserContext`, as quais são descritas nas subseções seguintes.

3.4.1 Containers

Em Jamp, *containers* são objetos da classe `JContainer`, a qual possui os métodos públicos descritos na Tabela 3.7. Para simplificar a descrição das classes do sistema, as exceções ativadas pelos métodos das mesmas não são relacionadas nas tabelas apresentadas nesta seção.

Em Jamp, existem duas diferenças básicas entre um objeto móvel, isto é, um objeto que pode ser inserido em um *container*, e um objeto comum de Java:

- Objetos móveis são criados usando-se o método `newObject` da classe `JContainer` e não pelo operador `new` de Java.
- Toda classe de um objeto móvel deve implementar pelo menos uma interface. Uma destas interfaces, e não a classe, é que deve ser utilizada para declarar referências para este objeto.

Classe JContainer
`JContainer(String description)` Cria um *container*, com um texto explicativo sobre o mesmo
`void addObj(Object obj)` Insere o objeto especificado no *container*
`void addClass(String className)` Insere o código da classe especificada no *container*
`void removeObj(Object obj)` Remove o objeto especificado do *container*
`void removeClass(String className)` Remove o código da classe especificada do *container*
`void move(String contextName, JStartObject startObj)` Move o *container* para o contexto especificado e informa que sua execução deve se iniciar pelo método `run` do objeto especificado
`static Object newObject(String className)` Cria um objeto móvel da classe especificada

Tabela 3.7: Métodos públicos da classe `JContainer`

Conforme descrito na Seção 3.2, este tipo de referência pode se encontrar em dois estados: conectado ou desconectado.

Como exemplo, o trecho de código a seguir cria um container e dois objetos móveis das classes `A_Impl` e `B_Impl`. Em seguida, os dois objetos são inseridos neste *container*, juntamente com suas respectivas classes e com uma terceira classe `C_Impl`, a qual pode ser necessária, por exemplo, para criar remotamente um objeto. Por fim, envia-se o *container* para um outro contexto, usando para isso o método `move`:

```
JContainer container = new JContainer ("exemplo de container");
A a = (A) JContainer.newObject("A_Impl");  // A_Impl implementa interface A
B b = (B) JContainer.newObject("B_Impl");  // B_Impl implementa interface B
container.addObj(a);
container.addObj(b);
container.addClass("A_Impl");
container.addClass("B_Impl");
container.addClass("C_Impl");
container.move("bob@server.foo.br:5000", a);
```

O método `move` espera que o contexto de sistema de destino seja especificado no seguinte formato: `user@host:port`, onde `user` é nome do usuário para o qual o *container* está sendo enviado, `host` é o nome da estação do contexto de destino e `port` é o número da porta TCP/IP associada a este contexto. Caso não se queira que o *container* seja armazenado no contexto de destino para posterior envio para um contexto de usuário, mas sim que seja executado automaticamente tão logo chegue ao mesmo, deve-se informar um nome de contexto no seguinte formato: `host:port`.

O último argumento do método `move` é o objeto por onde será iniciada a execução do *container* no contexto de destino. Este objeto deve implementar a interface `JStartObject`, cujo único método é o seguinte: `void start()`. Este método será automaticamente chamado por um contexto para iniciar a execução de um *container* recebido pelo mesmo.

O método `move` implementa um estilo de mobilidade conhecido como objetivo, isto é, *containers* no sistema são movidos a partir de um processo externo aos mesmos [Car99]. Além disso, como a implementação atual da JVM não permite migração de *threads*, o método `move` apenas transfere o estado dos objetos e as classes inseridas em um *container*.

3.4.2 Contextos

Em Jamp, contextos são criados usando-se as classes `JSystemContext` e `JUserContext`, conforme os mesmos sejam contextos de sistema ou de usuário. Os métodos públicos destas duas classes são descritos nas Tabelas 3.8 e 3.9, respectivamente.

Classe JSystemContext
`JSystemContext(int port)` Cria um contexto de sistema associado à porta TCP/IP especificada
`addUser(String name, String password)` Adiciona o usuário com nome e senha especificados ao contexto
`void start()` Inicia a execução do contexto

Tabela 3.8: Métodos públicos da classe `JSystemContext`

Classe JUserContext
`JUserContext(int port,String user,String password,String context)` Cria um contexto de usuário associado à porta TCP/IP, ao usuário e ao contexto de sistema especificados
`void start()` Inicia a execução do contexto

Tabela 3.9: Métodos públicos da classe `JUserContext`

Mostra-se a seguir um exemplo de um programa Java que cria um contexto de sistema e, em seguida, inicia a execução do mesmo.

```
public class ContextLauncher  {
  public static void main(String[] args) {
     JSystemContext context= new JSystemContext(Integer.parseInt(args[0]));
     context.addUser("bob", "x2yz");
     context.start();
  }
}
```

No exemplo anterior, ao se chamar o método `start`, o programa entra em um *loop* infinito, onde permanece aguardando a migração de *containers*.

3.5 Implementação

Um protótipo de Jamp foi implementado em Java, usando-se a versão 1.3 do JDK e possuindo aproximadamente 2500 linhas de código. A escolha de Java deve-se aos recursos oferecidos pela linguagem para geração de programas portáveis entre diversas arquiteturas, serialização de objetos, carregamento dinâmico de código, reflexividade, *threads* e programação distribuída usando-se *sockets*. Todos estes recursos são usados em Jamp.

O principal conceito utilizado na implementação do sistema é o de *mediador*. Um mediador é um objeto interno ao sistema, usado para viabilizar a implementação dos conceitos de referências conectadas e desconectadas.

Todo objeto móvel em Jamp possui um objeto mediador. Este mediador é criado juntamente com o objeto móvel quando se chama o método `newObject`. Um mediador possui dois atributos: `guid`, o qual é um valor de 128 bits que identifica unicamente o objeto mediado em qualquer nodo da rede e `ref`, que é uma referência para o objeto mediado. O atributo `guid` é gerado concatenando-se o endereço IP da máquina onde o objeto foi criado com um inteiro cujo processo de geração garante que não é possível obter nesta mesma máquina um valor idêntico ao longo do tempo. A classe `UID` do pacote `java.rmi.server` possui um construtor que retorna um inteiro com estas características.

A referência `ref` é a única referência existente na aplicação para o objeto mediado, já que o método `newObject` não retorna uma referência para o objeto criado e sim para o seu mediador. A Figura 3.4, mostra um *container* na visão do usuário e o mesmo *container* na visão interna do sistema, com os mediadores de cada um de seus objetos representados como círculos. Esta figura ressalta o fato de que mediadores constituem um recurso utilizado internamente na implementação do sistema, não fazendo parte da API descrita na Seção 3.4.

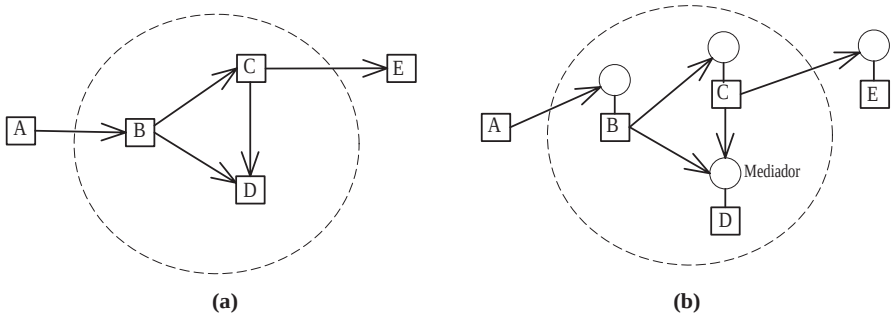


Figura 3.4: Container na visão do usuário e na visão interna do sistema

Suponha que se deseje enviar um *container* $c = \{(m_1, o_1), (m_2, o_2), \ldots, (m_n, o_n)\}$ para um

determinado contexto, onde o_i representa cada um dos n objetos deste *container* e m_i seus respectivos mediadores. Inicialmente, gera-se uma representação serializada do *container* c, utilizando para isso uma versão modificada das rotinas de serialização de objetos de Java. A implementação padrão destas rotinas serializa um objeto e todos os objetos acessíveis a partir do mesmo. Já a rotina de serialização usada na implementação do sistema restringe a serialização a objetos pertencentes ao *container* que está sendo migrado. Para isto, esta rotina utiliza os recursos de reflexividade de Java para tornar `null` atributos de objetos internos ao *container* que referenciem objetos externos ao mesmo. Uma lista encapsulada em cada *container* armazena os objetos internos ao mesmo.

Em seguida, a representação serializada do *container* é transmitida para o contexto de destino. Finalizada a transmissão, atribui-se `null` ao atributo `ref` de cada um dos mediadores $m_1, m_2, \ldots, m_n$ no contexto de origem. Como esta é a única referência mantida no sistema para os objetos $o_1, o_2, ..., o_n$, os mesmos se tornam inacessíveis após esta atribuição e, eventualmente, serão destruídos pelo coletor de lixo. Deste modo, consegue-se transferir um *container* utilizando uma semântica de migração de seus objetos e não de cópia, como ocorre, por exemplo, em sistemas baseados em Java RMI [Sun98].

Como `newObject` retorna uma referência para o mediador de um objeto móvel, quando se chama um método deste objeto está sendo chamado, na verdade, um método do seu mediador. Por este motivo, um mediador deve implementar todos os métodos das interfaces de seu objeto mediado. A implementação destes métodos no mediador verifica se o atributo `ref` é diferente de `null`, isto é, se objeto mediado encontra-se presente no contexto local. Caso esteja, a chamada é repassada para o objeto mediado. Caso contrário, ativa-se uma exceção do tipo `UnavailableObject`. Assim, nesta implementação do sistema, uma referência conectada é, na verdade, uma referência para um mediador com um objeto mediado diferente de `null`. Já uma referência desconectada é uma referência para um mediador com um objeto mediado igual a `null`.

Suponha agora que um determinado contexto tenha recebido para execução o *container* $c = \{(m_1, o_1), (m_2, o_2), \ldots, (m_n, o_n)\}$ mencionado anteriormente. Inicialmente, este contexto "desserializa" a representação do *container* recebida do contexto de origem. Neste processo, é utilizado um carregador de classes diferente do normalmente usado em Java [LB98]. Em Jamp, o carregador de classes usado para obter as classes de um *container* é uma instância da classe `JampClassLoader`. A necessidade de um carregador de classes específico para o sistema deve-se ao fato de que o código das classes dos objetos de um *container* pode não se encontrar armazenado no disco da estação local, como ocorre normalmente em uma aplicação Java. Conforme descrito na Seção 3.4, este código pode ter sido incorporado ao *container* e, portanto, se este for o caso, ele deve ser recuperado da representação serializada do mesmo. Daí a necessidade de um carregador de classes especial capaz de realizar tal tarefa.

Após a "desserialização" de um *container*, a implementação de Jamp verifica se é possível

conectar dois tipos de referências desconectadas:

- Referências desconectadas pertencentes a objetos internos ao *container* e que referenciem objetos presentes no novo contexto do mesmo. Para isso, os recursos de reflexividade de Java são usados para inspecionar atributos de objetos do *container* que referenciem mediadores com campo `ref` igual a `null`. Caso o objeto referenciado esteja presente no contexto corrente, estes atributos passam a referenciar o mediador local do mesmo.

- Referências desconectadas pertencentes a objetos externos ao *container* e que referenciem objetos internos ao mesmo. Para cada um dos objetos internos ao *container*, verifica-se se já existe no contexto local um mediador para o mesmo. Caso exista, este mediador passa a referenciar este objeto.

Para viabilizar os dois tipos de conexões mencionados acima, uma lista interna a todo contexto armazena os mediadores dos objetos localizados no mesmo.

Em Jamp, mediadores são criados usando-se o conceito de classes *proxy* dinâmicas (*dynamic proxy class*) do pacote de reflexividade da versão 1.3 de Java [AGH00]. Uma classe *proxy* dinâmica implementa uma lista de interfaces especificada em tempo de execução, tal como requerido pelo conceito de mediadores. Com isso, dispensou-se, por exemplo, a implementação de ferramentas externas ao ambiente JDK para criação de mediadores.

3.6 Análise Crítica

Inicialmente, na Subseção 3.6.1, apresenta-se uma comparação do modelo e do sistema de programação propostos neste capítulo com outras soluções semelhantes. Em seguida, a Subseção 3.6.2 descreve as principais contribuições do modelo apresentado, bem como relaciona futuros trabalhos.

3.6.1 Comparação com Outros Trabalhos

Como o modelo proposto neste capítulo é baseado na idéia de mobilidade de objetos, existe um grau de semelhança entre o mesmo e sistemas de agentes móveis. Assim, inicialmente nesta seção, compara-se Jamp com quatro sistemas Java para construção de agentes móveis: Aglets [LO98], Ajanta [TKV+99], μCode [Pic98] e JavaSeal [VB99].

Em Aglets e Ajanta, agentes móveis correspondem a subclasses de uma classe pré-definida nestes sistemas. Como Java não suporta herança múltipla, esta solução diminui o grau de reutilização na implementação de agentes. Já em Jamp, apenas *containers* são objetos de uma classe particular. Objetos móveis, isto é, pertencentes a *containers*, podem herdar de qualquer classe. A única restrição imposta pelo sistema é que estes objetos sejam criados por meio do método `newObject` e não via um `new` tradicional.

Tanto em Aglets como em Ajanta, o mecanismo de serialização tradicional de Java é usado para transferir um agente e todos os objetos alcançáveis a partir do mesmo. Já em Jamp, a noção de *containers* possibilita delimitar o conjunto de objetos a serem serializados. Em Ajanta, após a migração, o código de um agente é transferido por demanda de um repositório central. Logo, o sistema não é robusto a desconexões. Em Aglets, o código de um agente pode ser transferido por demanda de um repositório ou então ser armazenado em um arquivo JAR transferido junto com o agente. Esta última alternativa, no entanto, requer que o conjunto de classes a ser migrado seja determinado e fixado em tempo de instalação da aplicação. Já em Jamp, o programador tem controle sobre as classes transmitidas junto com um *container*. Além disso, diferentemente de Ajanta, Jamp não assume a existência de um repositório central de código.

Em μCode, existe uma abstração, chamada grupo, usada para definir o conjunto de objetos e classes a serem transmitidos junto com um agente móvel. No entanto, μCode não implementa nenhum mecanismo para comunicação entre grupos. Não existem, por exemplo, as noções de referências conectadas e desconectadas, disponíveis em Jamp. JavaSeal também disponibiliza uma construção, chamada *seal*, para agrupar um conjunto de objetos e classes. Ao contrário de Jamp, no entanto, o programador não pode remover classes de um *seal*. Além disso, comunicação entre *seals* se dá por meio de troca de mensagens em canais, utilizando primitivas *send* e *receive*. Valores são transmitidos por cópia e não é possível compartilhar objetos entre *seals*. Conforme reconhecido em [BR00], os mecanismos de comunicação de JavaSeal são primtivos e, em alguns casos, ineficientes.

O modelo de mobilidade adotado em Jamp é similar ao usado em sistemas de correio eletrônico na Internet. Um *container*, por exemplo, pode ser comparado a uma mensagem eletrônica, com a vantagem de que o mesmo pode conter dados e código e não apenas um texto estático. Já um contexto de sistema é semelhante a um servidor de mensagens e um contexto de usuário, a um programa leitor das mesmas.

Uma segunda comparação pode ser realizada com o modelo de mobilidade de *applets* de Java [AGH00]. Em certo sentido, um contexto de sistema desempenha papel semelhante ao de um servidor *Web* de onde uma *applet* é transferida. Um contexto de usuário, nesta comparação, seria semelhante a um *browser Web*. Por fim, um *container* seria semelhante a uma *applet*. No entanto, a fim de suportar operação em modo desconectado, *containers* são dotados não só de mobilidade de código, como uma *applet*, mas também de dados. Pelo mesmo motivo, o código de um *container* é transmitido junto com o mesmo, enquanto que *applets* carregam seu código dinamicamente do servidor *Web* onde foram instaladas. Por fim, *applets* são aplicacões móveis passivas, enquanto que *containers* dão origem a aplicações pró-ativas, que podem tirar aproveito de uma conexão temporária de rede para se transferirem para um computador móvel.

Do ponto de vista de Engenharia de Software, existe ainda uma certa similaridade entre *containers* e o modelo de componentes de *software* [Szy98]. A principal diferença entre os dois conceitos, no entanto, se dá em relação a seus propósitos. Um *container* é uma unidade de

mobilidade, proposta neste trabalho para viabilizar a construção de aplicações móveis capazes de lidar com os problemas de comunicação típicos de redes sem fio. Já um componente é basicamente uma unidade de composição, utilizada para viabilizar a construção de aplicações a partir de módulos previamente compilados.

3.6.2 Principais Contribuições

O modelo de programação apresentado, formalizado e implementado por um sistema descrito neste capítulo possibilita a construção de aplicações móveis tolerantes a desconexões. Existem três abstrações no modelo proposto para lidar com desconexões: *containers*, contextos e referências desconectadas.

Um *container* é um grupo de objetos e classes que podem ser enviados para outros nodos da rede. Assim, o modelo, por meio de sua implementaçao no sistema Jamp, permite a construção de aplicações que podem ser pró-ativamente enviadas com código e dados para computadores móveis e então serem executadas em modo desconectado. Além disso, o modelo de mobilidade proposto neste capítulo organiza os contextos de execução da rede em uma hierarquia de dois níveis. Desta maneira, as aplicações podem ser temporariamente armazenadas em nodos da rede fixa antes de serem transmitidas para um computador móvel. Por fim, o modelo propõe os conceitos de referência conectadas e desconectadas para permitir compartilhamento de objetos entre *containers* e, simultaneamente, não restringir a migração dos mesmos.

A versão de Jamp apresentada neste capítulo permite apenas comunicação entre *containers* localizados em um mesmo contexto. Como usual em linguagens orientadas por objetos, esta comunicação se dá por meio de chamadas síncronas de métodos. No capítulo 5, será apresentado um modelo de coordenação que pode ser utilizado para interação entre *containers* localizados em contextos distintos. A fim de se adequar às características do meio de comunicação sem fio, este modelo é baseado em comunicação assíncrona.

O sistema Jamp já foi utilizado para implementar parte de um sistema para revisão de artigos submetidos a uma conferência. Este sistema é citado em [Car99] como exemplo de uma aplicação distribuída que pode se beneficiar do tipo de abstração apresentado neste capítulo. Uma descrição mais detalhada deste sistema é realizada no Capítulo 7. Como trabalho futuro, pretende-se transformar um *container* em um *domínio de proteção* [BR00], isto é, fazer com que toda comunicação entre *containers* seja regulada por uma política de segurança previamente estabelecida em cada contexto da rede. O objetivo é atender aos requisitos de segurança que são colocados quando executa-se uma aplicação proveniente de outro nodo da rede. Além disso, pretende-se implementar uma versão do sistema no ambiente J2ME [Sun00].

Capítulo 4

Modelos de Coordenação

4.1 Introdução

Coordenação consiste na teoria e prática de se construir sistemas que requerem interação entre componentes independentes. Um modelo de coordenação disponibiliza uma infra-estrutura para que esta interação possa ser expressa [Cia96]. Considera-se, neste trabalho, que esta infra-estrutura deve incluir de forma integrada mecanismos que permitam a realização das seguintes tarefas: comunicação entre processos, localização de recursos e sincronização de atividades. Segundo esta definição, computação e coordenação são conceitos ortogonais [GC92, CG01]. Computação diz respeito à construção de programas individuais, por meio de linguagens tradicionais, como Java, Haskell ou Prolog. Já coordenação trata de como estes programas devem interagir entre si para realizar determinada tarefa. Para isso, utilizam-se normalmente *middlewares* para comunicação entre processos, como RPC e Java RMI, ou linguagens de coordenação, como Linda.

Conforme afirmado no Capítulo 1, coordenação de aplicações em redes sem fio é uma atividade bem mais complexa do que em ambientes tradicionais de rede. Basicamente, redes sem fio são caracterizadas por serem ambientes abertos e sujeitos a constantes reconfigurações. Assim, nem sempre duas partes que necessitam de se comunicar estão conectadas à rede ao mesmo tempo. Dada à autonomia e mobilidade dos nodos da rede, localização de recursos é também mais difícil, já que serviços são constantemente adicionados e removidos da rede. Por último, comunicação entre processos deve ser tolerante a desconexões, as quais freqüentemente tornam as aplicações temporariamente inacessíveis.

Neste capítulo são descritos alguns dos principais sistemas que já foram propostos para comunicação e coordenação em ambientes distribuídos. O objetivo é analisar criticamente a utilização dos mesmos em ambientes de redes móveis, incluindo redes infra-estruturadas e redes *ad hoc*. O restante do capítulo está organizado conforme descrito a seguir. A Seção 4.2 descreve sistemas baseados no modelo de objetos distribuídos, como Java RMI, CORBA e Jini. Na Seção 4.3 são descritos sistemas baseados em espaços de tuplas, notadamente Linda e Lime.

Dentre os sistemas mencionados, particular atenção é dada aos sistemas Jini e Lime, já que os mesmos foram projetados tendo redes sem fio como objetivo. A Seção 4.4 descreve um formalismo que adiciona capacidade de comunicação por meio de espaço de tuplas no Cálculo de Ambientes. Por último, a Seção 4.5 encerra o capítulo, apresentando algumas considerações finais.

4.2 Modelos Baseados em Objetos Distribuídos

Estes modelos têm como objetivo transportar para um ambiente distribuído o modelo de comunicação local utilizado em sistemas orientados por objetos. Tradicionalmente, objetos locais de uma aplicação interagem por meio de chamadas de métodos. Assim, em sistemas de objetos distribuídos, esta forma de interação é mantida para objetos remotos, isto é, objetos localizados em nodos diferentes da rede. Em última análise, estes sistemas procuram tornar transparente aos programadores o fato de uma chamada de método ser local ou remota, permitindo assim que eles construam aplicações distribuídas usando os mesmos recursos de comunicação que estão acostumados a utilizar em sistemas centralizados.

No restante desta seção, são descritos três sistemas baseados no modelo de objetos distribuídos: Java RMI, CORBA e Jini. Além de apresentar a arquitetura básica de cada um dos sistemas, analisa-se também a utilização dos mesmos em ambientes de comunicação sem fio.

4.2.1 Java RMI

Java RMI (*Remote Method Invocation*) [Sun98] é a biblioteca padrão Java para construção de aplicações distribuídas no modelo cliente/servidor. Basicamente, esta biblioteca permite que aplicações Java chamem métodos de objetos localizados em JVM remotas, isto é, introduz mobilidade de controle na linguagem. Aplicações distribuídas baseadas em Java RMI são constituídas por dois tipos de programas: servidor e cliente. O servidor tipicamente cria alguns objetos e torna referências para os mesmos disponíveis na rede, geralmente por meio do registro destes objetos em um serviço de nomes centralizado. Feito isso, o servidor passa a aguardar requisições dos clientes. Estes obtêm, por meio de consultas ao serviço de nomes, referências para objetos remotos, as quais podem ser então usadas para invocar métodos dos mesmos.

Java RMI permite ainda que objetos sejam passados como parâmetros, característica esta que o torna mais flexível que sistemas tradicionais de RPC [BN84], onde geralmente permite-se somente a passagem de valores de tipos básicos como parâmetros. Para tanto, Java RMI serializa o estado dos objetos de forma a transmiti-los pela rede. RMI implementa ainda coleta de lixo distribuída, envolvendo objetos remotos que não são mais referenciados por nenhum cliente.

Análise Crítica: Java RMI assume a existência de um serviço de nomes previamente conhecido por servidores e clientes de uma aplicação distribuída. Este requisito torna o sistema demasiada-

mente estático e rígido para uso em ambientes móveis. Neste tipo de ambiente, um computador móvel pode se encontrar a qualquer momento em uma nova rede. Portanto, não é razoável assumir que o mesmo conheça previamente o endereço do servidor de nomes de cada rede a qual pode vir a se conectar. Além disso, devido a desconexões não previstas, servidores podem se desconectar a qualquer momento de uma rede móvel. Neste caso, no entanto, os objetos remotos disponibilizados pelos mesmos permanecem registrados no servidor de nomes utilizado por RMI. Clientes podem então, ao consultar o servidor de nomes, obter uma referência para um objeto remoto não mais disponível na rede.

Por último, de forma a manter aderência ao modelo de comunicação local da linguagem, Java RMI implementa chamadas síncronas de métodos. Este sincronismo não é adequado a redes sem fio, já que nas mesmas o tempo de transmissão de uma mensagem pode variar de milisegundos a minutos, dependendo das condições da rede e da ocorrência de desconexões temporárias. Com isso, torna-se difícil estipular um limite superior para latência, o qual seria utilizado por RMI para sinalizar falhas de comunicação com a JVM do objeto chamado. Observe que a definição de um limite superior típico de redes locais, na ordem de milisegundos, pode dar origem a sistemas pouco tolerantes a falhas. Por outro lado, caso assuma-se o pior caso, e seja definido um limite superior da ordem de segundos ou mesmo minutos, o resultado será uma degradação no tempo de resposta das aplicações.

4.2.2 CORBA

Assim como Java RMI, CORBA (*Common Object Request Broker Architecture*) [Sie96] é um sistema que possibilita a construção de aplicações distribuídas utilizando chamada remota de métodos. No entanto, CORBA é uma solução padronizada, o que permite seu uso em diferentes linguagens e sistemas operacionais. Para garantir interoperabilidade, CORBA utiliza uma linguagem neutra para definição de interfaces, chamada IDL (*Interface Definition Language*) e o conceito de *Object Request Broker* (ORB), o qual constitui uma espécie de "barramento lógico" que conecta os diversos programas de uma aplicação distribuída. As mensagens que trafegam neste barramento são padronizadas por um protocolo chamado IIOP (*Internet Inter-ORB Protocol*). Além disso, para localizar objetos em uma rede, CORBA utiliza um serviço de nomes centralizado, bastante similar ao de Java RMI.

Análise Crítica: Sendo funcionalmente semelhante a Java RMI, CORBA sofre dos mesmos problemas deste sistema quando utilizado em redes sem fio.

4.2.3 Jini

Jini[1] [Arn00, Wal99] é um sistema para construção de aplicações distribuídas em redes sob constantes reconfigurações. O sistema utiliza Java RMI como infra-estrutura básica de comuni-

[1]Recursivamente, Jini é uma sigla para *Jini Is Not Initials*.

cação, possuindo, no entanto, um serviço de nomes próprio, chamado de *lookup service*. Além disso, Jini utiliza um protocolo mais flexível para registro, pesquisa e remoção de informações do serviço de nomes.

Assim como ocorre em Java RMI, Jini assume que uma aplicação distribuída é constituída por servidores e clientes. No entanto, diferentemente de RMI, em Jini servidores e clientes não precisam conhecer o endereço do servidor de nomes da rede à qual estão conectados. A fim de descrever o funcionamento do serviço de nomes de Jini, suponha a conexão de um novo serviço de impressão a uma aplicação distribuída, conforme ilustrado na Figura 4.1. Ao se conectar a esta rede, este servidor de impressão difunde uma mensagem procurando pelo servidor de nomes. Este responde fornecendo uma referência para o serviço de nomes provido pelo mesmo. O servidor de impressora utiliza então esta referência para registrar o novo serviço de impressão na rede. Clientes procurando por um serviço também localizam o serviço de nomes difundindo uma mensagem, a qual retorna uma referência para este serviço. Estes clientes podem utilizar então esta referência para procurar por um determinado serviço, como, por exemplo, o serviço de impressão registrado anteriormente.

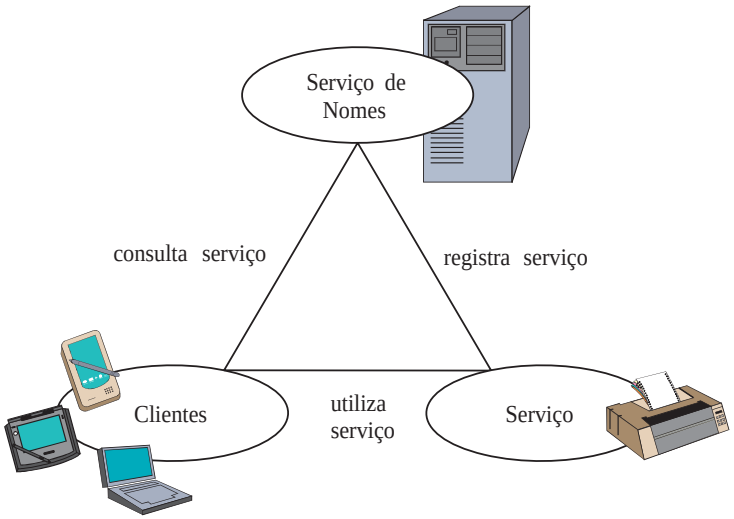


Figura 4.1: Serviço de nomes (*lookup service*) de Jini

Utilizando este protocolo para localização do serviço de nomes, Jini possibilita que novos serviços possam ser acrescentados transparentemente em uma aplicação, sem necessidade de suporte por parte dos administradores da mesma. O sistema, no entanto, dá um passo extra e permite que serviços possam ser também automaticamente desconectados de uma rede. Basicamente, a toda referência de rede registrada no serviço de nomes é associado um *lease*, isto é, um período de tempo durante o qual este registro permanece válido. Servidores podem renovar o *lease* de um serviço, caso pretendam continuar disponibilizando o mesmo. Por outro lado, caso um servidor se desconecte da rede, mesmo que de forma inesperada, seus serviços serão

automaticamente removidos do serviço de nomes assim que expire o *lease* associado aos mesmos.

Além de um serviço de nomes, Jini disponibiliza ainda outros recursos para construção de aplicações distribuídas, como notificação de eventos e transações.

Análise Crítica: Jini pode ser visto como uma extensão de Java RMI especificamente projetada para redes dinâmicas e sujeitas a constantes reconfigurações. Basicamente, o serviço de nomes disponível em Jini permite que serviços sejam acrescentados e removidos de uma rede transparentemente. No entanto, assim como ocorre com Java RMI, o serviço de nomes de Jini é centralizado, o que torna o sistema adequado apenas para programação em redes sem fio infra-estruturadas. Conforme ilustrado na Figura 4.2, reproduzida de [HR01], a arquitetura centralizada usada pelo serviço de nomes de Jini não é aderente a um ambiente de comunicação *ad hoc*.

Além disso, de forma similar a RMI, chamadas remotas em Jini são síncronas e dão origem a uma exceção no caso de falhas de comunicação com o nodo remoto. Por exemplo, na Figura 4.1, uma desconexão do serviço de impressão, mesmo que temporária, é tratada pelo sistema como um evento excepcional e não como um evento normal em ambientes sem fio. Portanto, de forma geral, dado o grau de sincronismo do sistema, aplicações em Jini não são tolerantes a desconexões, nem capazes de operar em modo desconectado.

4.3 Modelos Baseados em Espaços de Tuplas

Nestes modelos, comunicação entre processos é intermediada por um *espaço de tuplas*, o qual representa uma estrutura de dados persistente e compartilhada por todos componentes de uma aplicação distribuída. De forma similar a um quadro de mensagens, processos utilizam o espaço de tuplas para interagirem entre si, depositando, lendo e removendo informações do mesmo. Este estilo de coordenação foi proposto originalmente para uso em aplicações paralelas ou distribuídas. Recentemente, com o surgimento da Internet, o interesse por modelos de coordenação baseados em espaço de tuplas tem sido renovado, devido principalmente ao estilo de comunicação assíncrono proporcionado pelos mesmos.

O restante desta seção está organizado como descrito a seguir. A Subseção 4.3.1 descreve o modelo proposto por Linda. Finalizando, a Subseção 4.3.2 descreve o sistema Lime, proposto recentemente como uma versão de Linda para uso em ambientes de redes móveis.

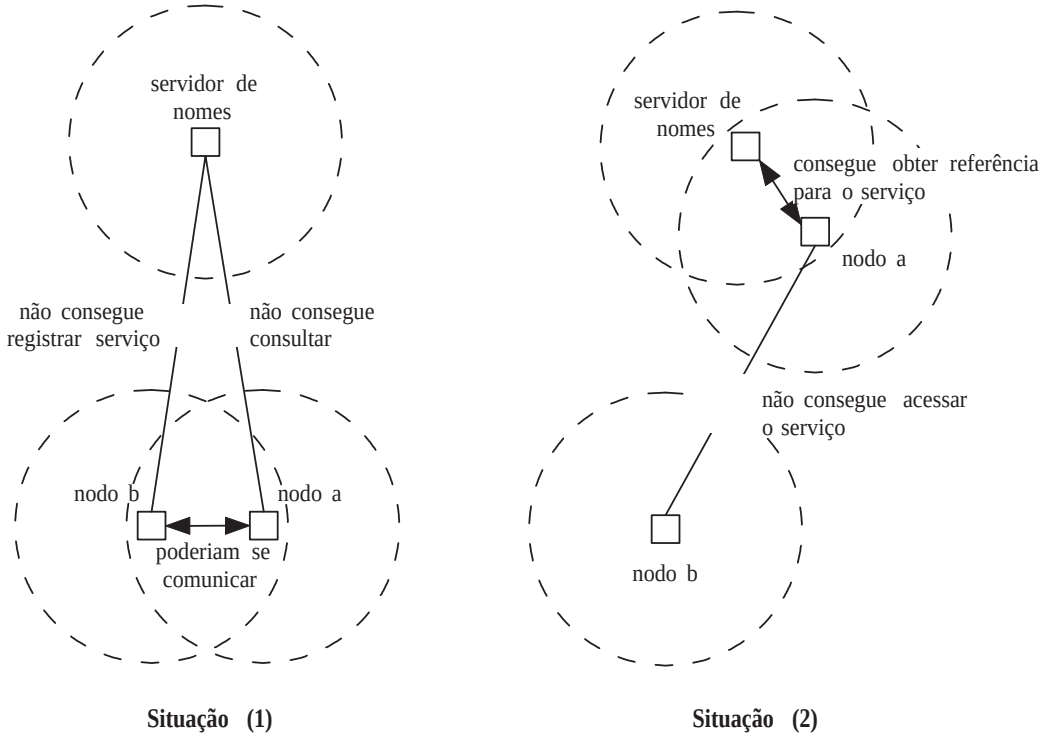


Figura 4.2: Os círculos tracejados representam a área de alcance dos nodos de uma rede móvel *ad hoc*. Na situação (1), os nodos a e b estão um ao alcance do outro, mas o nodo a não consegue descobrir o serviço demandado pelo mesmo no nodo b, já que o servidor de nomes se encontra inacessível. Na situação (2), o nodo a consulta o servidor de nomes e obtém uma referência para o serviço registrado previamente pelo nodo b. No entanto, neste momento, o nodo b encontra-se temporariamente inacessível – ele pode, por exemplo, ter se movido ligeiramente após o registro do serviço. Uma tentativa de a acessar o serviço em b produz então uma exceção.

4.3.1 Linda

Proposto por Gelernter, em meados da década de 80, Linda[2] [Gel85] foi o primeiro modelo de coordenação a utilizar o conceito de espaço de tuplas. Em Linda, processos se comunicam inserindo, lendo e removendo mensagens desta estrutura compartilhada. No modelo, mensagens são estruturadas na forma de tuplas, isto é, como seqüências ordenadas de dados. Processos acessam o espaço compartilhado utilizando apenas três primitivas, conforme descrito a seguir:

- A primitiva **out** t insere assincronamente a tupla t no espaço de tuplas.
- A primitiva **in** t, x remove uma tupla compatível com o padrão t do espaço de tuplas e associa a mesma a x; caso existam várias tuplas compatíveis com este padrão, uma delas

[2]Segundo Ciancarini [Cia01], Linda recursivamente significa *Linda is not Ada*. Veja, portanto, que o nome reflete a separação entre computação e coordenação preconizada pelo modelo. Separação esta que não ocorre em Ada, já que esta linguagem integra um modelo imperativo de computação a um modelo de comunicação baseado em troca de mensagens.

é não-deterministicamente escolhida; caso nenhuma tupla seja encontrada, a operação fica bloqueada até que uma tupla compatível com t seja inserida.

- A primitiva **rd** t, x comporta-se como um **in** , exceto que a tupla não é removida do espaço, mas apenas copiada para x.

O espaço de tuplas de Linda comporta-se como uma memória associativa. Processos leitores e consumidores necessitam apenas fornecer uma descrição parcial, isto é, um padrão, para a tupla desejada. Compatibilidade entre tuplas e padrões baseia-se na aridade e nos valores das tuplas armazenadas. O símbolo **?** em um padrão denota um *wild card*, isto é, compatibilidade com qualquer valor. Por exemplo, a tupla $\langle \text{foo}, 15 \rangle$ é compatível com os padrões $\langle ?, 15 \rangle$, $\langle \text{foo}, ? \rangle$, $\langle \text{foo}, 15 \rangle$ e $\langle ?, ? \rangle$.

Comunicação em Linda é distribuída no espaço e no tempo. Distribuída no tempo porque um processo produtor pode inserir uma mensagem no espaço persistente, a qual somente será recuperada mais tarde por um processo consumidor. Assim, não é necessária a existência de uma conexão temporal entre processos para que eles possam interagir. Já distribuição no espaço é possível graças ao fato de que tuplas depositadas no repositório compartilhado são visíveis aos diversos nodos de um sistema distribuído. Além disso, graças ao uso de padrões para ler e remover tuplas, processos não precisam conhecer mutuamente seus identificadores para trocar mensagens. Por exemplo, processos consumidores são capazes de remover mensagens baseados no conteúdo das mesmas, sem que necessariamente tenham conhecimento da identidade dos processos produtores. A semântica de bloqueio utilizada nas operações **in** e **rd** também possibilita sincronização entre processos. Por exemplo, processos consumidores ficam automaticamente bloqueados caso não existam mensagens disponíveis.

Todas estas características têm motivado recentemente o surgimento de novas implementações de Linda, como JavaSpaces [FHA99], da Sun, e TSpaces [WMLF98], da IBM.

Análise Crítica: Simplicidade e expressividade – ou, em apenas uma palavra, elegância – são características que se sobressaem em Linda. Além disso, o estilo de comunicação assíncrono e associativo inerente ao modelo é particularmente interessante em redes abertas, reconfiguráveis e com latência variável, como é o caso de redes sem fio. A principal limitação de Linda, no entanto, quando empregado nestas redes, é o fato de o repositório de tuplas ser uma estrutura centralizada e compartilhada por todos os nodos da rede. Este fato torna o sistema inadequado para coordenação de aplicações em redes *ad hoc*.

4.3.2 Lime

Proposta na Tese de Doutorado de Murphy, Lime (*Linda in a Mobile Environment*) [PMR99, Mur00, MPR01] é uma versão de Linda projetada para uso em ambientes móveis, mais precisamente em redes *ad hoc*. A principal contribuição de Lime é o fato de abandonar a idéia de

um espaço de tuplas centralizado e global pelo conceito de *espaços de tuplas transientemente compartilhados*, conforme descrito a seguir.

Aplicações em Lime são constituídas por *agentes móveis*. Considera-se, no entanto, que todo agente móvel possui seu próprio espaço de tuplas, o qual é transportado com o mesmo quando ele migra de um nodo para outro da rede[3]. Espaços de tuplas de agentes localizados em um mesmo nodo são transparentemente fundidos pelo sistema, criando uma abstração chamada de espaço de tuplas local. Assim, estes espaços locais são estruturas virtuais e transientes. Virtuais porque são criados pelo sistema a partir dos espaços físicos de cada agente móvel e transientes porque o conteúdo dos mesmos altera-se conforme agentes migrem para o nodo ou abandonem o mesmo.

Lime dá ainda um passo extra e funde os espaços de tuplas locais de cada nodo conectado à rede em um repositório global, porém virtual, chamado de *espaço de tuplas federado*. Este espaço é manipulado usando-se variações das primitivas de Linda que permitem delimitar o espaço de tuplas físico onde a operação será executada, conforme descrito a seguir:

- A primitiva **out** t, a insere a tupla t no espaço de tuplas do agente a. Caso o agente a não esteja presente na rede, a tupla permanece *mal localizada* (*misplaced*) no espaço do agente a' que requisitou a operação até que o agente a se torne disponível. Diz-se, neste caso, que a é o destino da tupla e a' a sua localização. Assim, toda tupla em Lime possui internamente dois campos extras, responsáveis por armazenar sua localização e o seu destino. Uma tupla é dita mal localizada quando estes campos possuem valores diferentes.

  Caso o campo a seja omitido, a tupla é inserida no espaço do agente a', isto é, do agente que solicitou a operação.

- A primitiva **in** t, a, a', x permanece bloqueada até que uma tupla t' compatível com o padrão t e tendo como destino o agente a' esteja localizada no espaço de tuplas do agente a. Se várias tuplas forem encontradas, uma delas é escolhida não-deterministicamente. A operação remove a tupla t' do espaço e associa a mesma a x.

  Como usual em Linda, o símbolo **?** indica qualquer tupla v, qualquer localização a ou qualquer destino a', conforme seja usado, respectivamente, como primeiro, segundo ou terceiro parâmetro da primitiva **in** . Particularmente, **in** t, x é uma abreviação para **in** $t, ?, ?, x$.

- A primitiva **rd** t, a, a', x comporta-se como um **in** , exceto que a tupla não é removida, mas apenas copiada para x.

Além de espaços transientemente compartilhados, Lime introduz ainda o conceito de reações, o qual possibilita que um trecho de código seja executado quando uma determinada tupla for

[3]Na verdade, um agente móvel em Lime pode possuir múltiplos espaços de tuplas, sendo todos eles transportados com o mesmo. No entanto, não sendo fundamental no modelo, omite-se esta característica na descrição realizada na presente seção.

inserida no espaço de tuplas federado. A primitiva **react** t, P é usada para registrar uma reação. Após a execução desta primitiva, quando uma tupla t' compatível com o padrão t for inserida no espaço federado, a rotina P é executada atomicamente. Ou seja, enquanto P estiver em execução, todas as outras primitivas que acessam o espaço de tuplas permanecem bloqueadas.

Implementação de Operações em Espaços de Tuplas Federados: Conforme afirmado anteriormente, um espaço de tuplas federado é uma abstração – ou uma estrutura virtual – criada por Lime a partir dos espaços de tuplas físicos de cada um dos nodos de uma rede. Assim, a fim de se obter um conhecimento maior sobre o modelo, é interessante verificar como as primitivas tradicionais de Linda são implementadas em um espaço de tuplas federado.

Inicialmente, a implementação da primitiva **out** não oferece maior complexidade, já que a determinação do espaço físico onde a tupla será inserida é imediata. Já a primitiva **in** possui uma implementação bem mais complexa, já que nem sempre é especificado o espaço onde a tupla deve ser procurada. Suponha, por exemplo, uma aplicação com um conjunto de agentes $A_1, \ldots, A_n$, todos eles em nodos distintos. Suponha ainda que um agente S solicite uma operação **in** t, x. A execução desta operação em Lime segue o seguinte protocolo:

- O agente S difunde uma operação **rd** t, x, a qual é executada no espaço de tuplas físico de cada um dos agentes A_i. Estes executam a leitura localmente e, quando uma tupla é encontrada, retornam sua localização para o agente S.
- O agente S, ao receber a informação de que a tupla procurada foi encontrada em um determinado agente A_j, envia uma operação **inp** t, x para ser executada no espaço de tuplas deste agente. Esta operação, chamada de *probe*, comporta-se com um **in**, exceto que a mesma retorna imediatamente um erro caso uma tupla compatível não seja encontrada, isto é, a operação não fica bloqueada aguardando esta tupla.
- Caso a operação **inp** consiga remover uma tupla, a mesma é retornada para o agente S. Caso contrário, isto é, se a tupla tiver sido removida por outro agente enquanto o **inp** estava em trânsito, propaga-se o erro retornado pela operação para o agente S.
- Caso o agente S receba como resposta a tupla desejada, a mesma é associada a x. Difunde-se uma operação para remover os **rd** enviados para os demais agentes A e o protocolo é finalizado.
- Caso o agente S receba como resposta um erro, ele envia uma nova operação **rd** t, x para o agente A_j e volta a aguardar a resposta de um dos agentes A sobre a existência de uma tupla em seus espaços locais.

A primitiva **in** requer, portanto, a execução de uma operação em apenas um dos nodos de um espaço federado, cuja localização não é conhecida *a priori*. No melhor caso – quando

o primeiro **inp** enviado consegue remover a tupla – o número de mensagens trocadas por este protocolo é igual a $2n+2$, onde n é o número de nodos. Basicamente, são enviadas n mensagens **in** , uma mensagem com a localização da tupla desejada, uma mensagem **inp**, uma mensagem com a tupla removida e mais $n-1$ mensagens para cancelamento das operações pendentes.

A implementação da primitiva **rd** ocorre difundindo uma operação de leitura para todos os nodos da rede. Esta operação é mais simples que um **in** , já que não constitui um problema o fato de a operação ser bem sucedida em diversos nodos. Neste caso, basta que o agente que solicitou o **rd** escolha uma das múltiplas respostas recebidas.

A última observação sobre a implementação de Lime diz respeito à conexão de novos nodos em um espaço federado. Suponha, por exemplo, a conexão de um novo nodo N a um espaço federado T. Neste caso, no momento da conexão, deve-se migrar para N tuplas *mal localizadas* em T cujo destino final seja este novo nodo. De forma semelhante, tuplas *mal localizadas* em N tendo como destino final um dos nodos de T também devem ser entregues. A fim de manter a consistência do espaço federado, a transferência de tuplas mal localizadas em Lime é executada atomicamente, isto é, demais operações no espaço de tuplas T ficam bloqueadas enquanto tuplas são transferidas de e para o novo nodo N.

A fim de verificar a expressividade e escalabilidade de operações em espaços de tuplas federados, foram desenvolvidos dois pequenos experimentos usando a implementação de Lime disponibilizada por seus projetistas. Esta implementação permite o uso de Lime em uma rede local. Os experimentos descritos a seguir foram executados em uma rede Ethernet, de 10 Mbits, constituída por microcomputadores Dual Pentium II, com *clock* de 400 Mhz e sistema operacional SunOS 5.6. Os programas foram implementados em Java e interpretados usando a JVM que acompanha o ambiente JDK 1.2.2 da Sun. Todas as medidas de tempo foram obtidas calculando-se a média aritmética dos tempos de execução de diversas iterações dos experimentos descritos.

Experimento 1: Neste experimento, dois agentes L_1 e L_2, fixados em um mesmo nodo da rede, encontram-se trocando mensagens. Particularmente, L_1 encontra-se consumindo tuplas produzidas por L_2. Simultaneamente, é ativado um certo número de agentes R_i, localizados cada um deles em nodos diferentes da rede. Estes agentes foram implementados de tal forma que R_1 consome tuplas produzidas por R_2, R_3 consome tuplas produzidas por R_4 e assim sucessivamente. Como os agentes L_1, L_2 e R_j estão conectados na mesma rede, a interação descrita acima ocorre via um espaço de tuplas federado. Além disso, supõe-se que agentes consumidores não conhecem a localização dos agentes produtores. Com isso, consumidores retiram tuplas do espaço federado baseados apenas no conteúdo das mesmas, isto é, sem especificar a localização ou o destino das tuplas que desejam consumir.

A Tabela 4.1 mostra o número de mensagens trocadas por segundo entre L_1 e L_2 quando quantidades diferentes de agentes R estão conectados à rede. Os resultados mostram que a velocidade de comunicação entre L_1 e L_2 diminui à medida que aumenta o número de pares de

agentes R na rede. Por exemplo, com seis agentes R engajados no espaço federado, a velocidade de comunicação diminui para menos da metade de quando não havia nenhum agente R na rede.

Número de Agentes R	Mensagens/seg
0	0.38
1	0.38
2	0.27
3	0.26
4	0.21
5	0.20
6	0.18

Tabela 4.1: Mensagens trocadas por segundo entre os agentes L_1 e L_2

A diminuição do número de mensagens trocadas entre L_1 e L_2 ocorre porque a interação entre estes dois agentes é interrompida para tratar a comunicação entre os agentes R, já que todos eles estão conectados ao mesmo espaço federado. Toda vez que um agente consumidor R_i executa um **in**, uma operação é difundida para todos os nodos da rede, incluindo o nodo onde estão localizados L_1 e L_2. Esta operação concorre, portanto, com as operações para troca de mensagens locais entre estes dois agentes. Assim, o resultado deste experimento sugere que o *overhead* inerente à implementação de um espaço de tuplas federado é dividido por todos os agentes conectados ao mesmo, incluindo aqueles que não estão utilizando diretamente seus recursos, como é o caso dos agentes L_1 e L_2.

Experimento 2: Este segundo experimento calcula um somatório em paralelo. O experimento consiste de um conjunto de agentes, executando em nodos distintos da rede. Inicialmente, cada agente deposita um inteiro no espaço federado. Em seguida, os agentes interagem para calcular o somatório destes inteiros. Dois algoritmos diferentes foram implementados para calcular este somatório.

No primeiro algoritmo, assume-se que os agentes não conhecem a localização uns dos outros. Assim, cada agente sucessivamente remove dois inteiros do espaço de tuplas e insere de volta a soma dos mesmos. A fim de evitar *deadlocks*, um conjunto de *tokens* é inserido previamente no espaço. Basicamente, antes de retirar dois inteiros, um agente deve retirar uma *token* do espaço, a qual garante a disponibilidade dos valores de que necessita. Para fins de identificação, este algoritmo é chamado de algoritmo com transparência de localização[4].

No segundo algoritmo, assume-se que os agentes conhecem a localização uns dos outros. Assim, ao retirar um inteiro, um agente especifica na operação **in** a localização do agente de posse do mesmo. Mais especificamente, um agente a, envolvido no passo p do algoritmo, retira inteiros do seu próprio espaço de tuplas e do espaço de tuplas do agente $a + 2^p$, $0 \leq p < \log n$, onde n é

[4]No contexto de modelos de espaços de tuplas distribuídos, transparência de localização denota a capacidade de consultar ou remover uma tupla sem que seja necessário informar o espaço de tuplas físico da mesma.

o número total de agentes. Um agente a participa do passo p se, e somente se, $a \mod 2^{p+1} = 0$. Este segundo algoritmo é chamado de algoritmo sem transparência de localização.

A Tabela 4.2 mostra os tempos de execução de cada algoritmo. Conforme descrito acima, o número de agentes é igual ao número de nodos da rede, que por sua vez, é igual ao número de inteiros sendo somados. Como esperado, os resultados mostram que o tempo de execução do algoritmo com transparência de localização é sempre maior do que sem transparência de localização. Enquanto a diferença com dois agentes é de 53%, com quatro agentes a mesma atinge 88%. Infelizmente, a implementação de Lime usada nos experimentos abortou com uma exceção quando tentou-se rodar o algoritmo com transparência de localização em uma rede com 8 nodos.

Número de Agentes	Tempo (segundos)	
	Com Transparência de Localização	Sem Transparência de Localização
2	15.2	9.9
4	34.6	18.4
8	-	32.1

Tabela 4.2: Soma paralela em Lime

Mais uma vez, o resultado deste experimento demonstra que o custo de operações cujo escopo é todo espaço de tuplas federado não é desprezível. Mesmo com a impossibilidade de se obter todas as medidas de tempo, verifica-se que a perda de expressividade que ocorre quando abre-se mão de transparência de localização é compensada por um ganho não desprezível em desempenho e escalabilidade.

Análise Crítica: Lime caracteriza-se primeiramente pela ausência de qualquer estrutura centralizada, já que cada componente de uma aplicação possui seu próprio espaço de tuplas. Conforme afirmado ao longo desta seção, este requisito é essencial para suportar operação em modo *ad hoc*. Outra característica essencial do modelo é o conceito de espaços de tuplas transientemente compartilhados. Principalmente quando aplicado a uma rede, dando origem aos chamados espaços de tuplas federados, este conceito é bastante poderoso. Basicamente, ele torna transparente às aplicações a localização física das tuplas na rede, bem como alterações na topologia da mesma. No entanto, espaços de tuplas transientemente compartilhados apresentam desvantagens que não podem ser desconsideradas, conforme descrito a seguir:

- Desempenho e escalabilidade: conforme pode ser verificado nos dois experimentos descritos anteriormente, os custos de operações em espaços de tuplas federados não são desprezíveis. A implementação da primitiva **in** , por exemplo, exige um protocolo específico para garantir unicidade na sua execução, isto é, a remoção de apenas uma tupla dentre os diversos espaços físicos da rede. Além disso, a fim de preservar a consistência do espaço federado, conexões de nodos à rede são serializadas juntamente com as demais operações realizadas

no mesmo. Com isso, no momento da conexão de um novo nodo, uma transação distribuída é executada para entregar tuplas *mal localizadas*. Evidentemente, em ambientes sujeitos a constantes reconfigurações, o impacto de tais transações não pode ser negligenciado.

- Desconexões devem ser anunciadas: nodos que desejem se desconectar da rede, devem anunciar explicitamente tal fato em Lime. Para isso, a implementação do sistema disponibiliza uma operação **disengage**, que deve ser chamada previamente por um nodo antes de sua desconexão da aplicação. Este anúncio é necessário para que o sistema possa remover reações e operações no espaço federado que estejam em andamento neste nodo ou que tenham sido registradas pelo mesmo em nodos remotos. Evidentemente, este requisito não é razoável em redes sem fio, já que desconexões involuntárias são eventos freqüentes neste tipo de ambiente.

Finalmente, espaços de tuplas federados podem ser comparados a sistemas de memória compartilhada distribuída (DSM, ou *Distributed Shared Memory*) [CDK01]. O uso destes sistemas é comum em multicomputadores, isto é, em arquiteturas multiprocessadas onde cada CPU dispõe de sua própria memória física. Veja, no entanto, que multicomputadores são sistemas estáveis, onde o número de processadores é geralmente conhecido *a priori* e mantido constante durante toda execução de uma aplicação. Já redes sem fio são caracterizadas exatamente pela volatilidade de sua topologia. Além disso, as únicas operações oferecidas por um sistema DSM são leitura e escrita em um endereço conhecido de memória. Assim, o mapeamento de um endereço virtual para um endereço físico é relativamente simples. Já em um espaço de tuplas federado, pode-se realizar operações mais complexas, como um **in** , cuja implementação requer considerável grau de sincronização global.

Uma análise mais detalhada de Lime pode ser encontrada em [CVV01b]. Esta análise motivou o desenvolvimento do sistema Corelime [CVV01a], o qual restringe a idéia de espaços transientemente compartilhados ao conjunto de agentes móveis em execução em um nodo. Assim, em nome de escalabilidade e desempenho, Corelime não disponibiliza espaços de tuplas federados. Logo, o modelo é adequado apenas para coordenação de sistemas baseados em agentes móveis, isto é, sistemas dotados de mobilidade lógica.

4.4 Modelo Teórico

Propõe-se nesta seção um formalismo que adiciona ao Cálculo de Ambientes capacidade de comunicação via espaço de tuplas, conforme definido em Linda. Basicamente, o formalismo proposto insere em todo ambiente um espaço de tuplas, o qual pode ser acessado pelos processos em execução neste ambiente usando-se as primitivas tradicionais de Linda. A variante proposta mantém inalteradas as primitivas de Ambientes para mobilidade e para comunicação local. O formalismo resultante será usado para definir a semântica do modelo de coordenação PeerSpaces,

proposto no Capítulo 5 deste trabalho. Ressalte-se que extensões similares à apresentada nesta seção já foram definidas para outros cálculos de processos em [BGZ98, DP96, BOV99].

Processos no formalismo proposto são construídos de acordo com a seguinte sintaxe:

$$
\begin{array}{lcl}
P, Q & ::= & (\nu\, n)\, P \mid \mathbf{0} \mid P \mid Q \mid !P \mid n\,[P, T\,] \mid M.P \mid L.P \mid (x).P \mid \langle M \rangle \\
M & ::= & x \mid n \mid \mathbf{in}\, M \mid \mathbf{out}\, M \mid \mathbf{open}\, M \mid M.M \mid \varepsilon \\
L & ::= & \mathbf{out_L}\ v \mid \mathbf{in_L}\ v, x \mid \mathbf{rd_L}\ v, x
\end{array}
$$

Esta gramática é essencialmente a mesma de Ambientes, apresentada no Capítulo 2, estendida com a regra L, a qual define a sintaxe das primitivas de Linda adicionadas ao formalismo. Como **in** e **out** são palavras reservadas em Ambientes, as primitivas de Linda incorporadas ao modelo são chamadas de $\mathbf{in_L}$, $\mathbf{rd_L}$ e $\mathbf{out_L}$.

No formalismo proposto, um ambiente é denotado por $n\,[P, T\,]$, onde n é o nome do ambiente, P é o conjunto de ambientes aninhados e de processos em execução no interior de n e T é o espaço de tuplas incorporado a cada ambiente. Os processos existentes em P acessam o espaço local de tuplas T por meio das primitivas $\mathbf{in_L}$, $\mathbf{rd_L}$ e $\mathbf{out_L}$. Como usual, a estrutura hierárquica de ambientes é manipulada através das primitivas **in**, **out** e **open**, as quais são usadas, respectivamente, para se entrar, sair e abrir ambientes.

A Tabela 4.3 apresenta a semântica operacional do formalismo proposto. Assim como no Cálculo de Ambientes, a semântica apresentada é baseada em reduções e em uma relação de congruência estrutural entre processos. Existe também um conjunto de regras de compatibilidade de padrões.

As reduções A1 a A4 definem a semântica das primitivas tradicionais de Ambientes. Essencialmente, estas reduções são as mesmas da versão original do cálculo, exceto pelo fato de operarem sobre ambientes equipados com um espaço local de tuplas. Na regra A3, a abertura de um ambiente m acarreta a união do espaço local de tuplas do mesmo com o espaço de tuplas do ambiente n no qual m está incluído. Esta semântica é compatível com a idéia de abertura de ambientes, visto que o mesmo procedimento ocorre em relação às mensagens depositadas em m – as quais, após a abertura, são também incorporadas ao ambiente pai n.

As regras L1 a L3 definem reduções para manipular o espaço de tuplas incorporado a ambientes. A operação de saída, $\mathbf{out_L}\ v$, insere uma tupla assincronamente no espaço de tuplas local de um ambiente (regra L1). As operação de entrada, $\mathbf{in_L}\ v, x.P$, e de leitura, $\mathbf{rd_L}\ v, x.P$, tentam localizar uma tupla v' compatível com o padrão v (regras L2 e L3). Quando uma tupla for encontrada, todas ocorrências livres de x em P são substituídas por v', o que é denotada da seguinte forma: $P\{v'/x\}$. No caso de um $\mathbf{in_L}$, a tupla é removida do espaço de tuplas do ambiente.

O conjunto seguinte de regras define uma relação de congruência estrutural, representada por $\equiv$. Esta relação aplica-se tanto a processos (regras SC1 a SC7) como a ambientes (regras SC8 a SC10). Como usual no Cálculo de Ambientes, estas regras permitem a reorganização sintática

Reduções

Primitivas de Ambientes:

$n\,[\,\mathbf{in}\,m.P \mid Q, T\,] \mid m\,[\,R, T'\,] \;\rightarrow\; m\,[\,n\,[\,P \mid Q, T\,] \mid R, T'\,]$ (A1)

$m\,[\,n\,[\,\mathbf{out}\,m.P \mid Q, T\,] \mid R, T'\,] \;\rightarrow\; n\,[\,P \mid Q, T\,] \mid m\,[\,R, T'\,]$ (A2)

$n\,[\,\mathbf{open}\,m.P \mid m\,[\,Q, T'\,], T\,] \;\rightarrow\; n\,[\,P \mid Q, T \cup T'\,]$ (A3)

$(x).P \mid \langle M \rangle \;\rightarrow\; P\{M/x\}$ (A4)

Primitivas de Linda:

$n\,[\,\mathbf{out_L}\; v \mid P, T\,] \;\rightarrow\; n\,[\,P, v \cup T\,]$ (L1)

$n\,[\,\mathbf{in_L}\; v, x.P \mid Q, v' \cup T\,] \;\rightarrow\; n\,[\,P\{v'/x\} \mid Q, T\,],$ if $v' \leq v$ (L2)

$n\,[\,\mathbf{rd_L}\; v, x.P \mid Q, v' \cup T\,] \;\rightarrow\; n\,[\,P\{v'/x\} \mid Q, v' \cup T\,],$ if $v' \leq v$ (L3)

Congruência Estrutural

$P \mid Q \equiv Q \mid P$	(SC1)	$(\nu\, x)\,(\nu\, y)\,P \equiv (\nu\, y)\,(\nu\, x)\,P$	(SC5)
$!\,P \equiv P \mid !\,P$	(SC2)	$P \equiv Q \Rightarrow (\nu\, x)\,P \equiv (\nu\, x)\,Q$	(SC6)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(SC3)	$(\nu\, x)\,(P \mid Q) \equiv P \mid (\nu\, x)\,Q$	(SC7)
$P \mid \mathbf{0} \equiv P$	(SC4)		

$P \equiv Q \Rightarrow n\,[\,P, T\,] \equiv n\,[\,Q, T\,]$ (SC8)

$(\nu\, x)\,n\,[\,P, T\,] \equiv n\,[\,(\nu\, x)\,P, T\,]$ (SC9)

$(\nu\, x)\,(n\,[\,P, T\,] \mid n'\,[\,P', T'\,]) \equiv (\nu\, x)\,n\,[\,P, T\,] \mid n'\,[\,P', T'\,]$ (SC10)

As regras acima são sujeitas às seguintes condições:
(SC7) if $x \notin fn(P)$ (SC10) if $x \neq n',\; x \notin fn(P')$
(SC9) if $x \neq n$

Compatibilidade de Padrões

$v \leq v \qquad v \leq ? \qquad \dfrac{v'_1 \leq v_1 \;\ldots\; v'_n \leq v_n}{\langle v'_1 \;\ldots\; v'_n \rangle \leq \langle v_1 \;\ldots\; v_n \rangle}$

Tabela 4.3: Semântica

de processos e ambientes a fim de viabilizar a aplicação de reduções. Nas regras mencionadas, $fn(P)$ designa o conjunto de nomes livres em P.

Por último, as regras de compatibilidade de padrões, denotadas por $v' \leq v$, permitem o casamento recursivo de valores com padrões. Padrões são compatíveis com valores se eles são iguais ou se o símbolo ? ocorre do lado direito. Estas regras são usadas pelas primitivas de Linda adicionadas ao formalismo.

4.5 Considerações Finais

Descreveu-se neste capítulo alguns dos principais modelos existentes atualmente para comunicação e coordenação em sistemas distribuídos. Mostrou-se que nenhum deles atende inte-

gralmente aos requisitos de coordenação impostos por um ambiente de rede sem fio, incluindo redes infra-estruturadas e redes *ad hoc*. Em última análise, pode-se argumentar que, à exceção de Lime, os problemas dos demais modelos descritos decorrem da estrita aderência dos mesmos à arquitetura cliente/servidor. Aplicações nesta arquitetura, em geral, assumem servidores centralizados, sempre acessíveis e previamente conhecidos por todos os clientes. Em Jini, por exemplo, apesar da existência de um protocolo bastante dinâmico para localização de recursos, o serviço de nomes ainda é centralizado. O mesmo ocorre com o repositório de tuplas em implementações tradicionais de Linda.

Assim, modelos para coordenação de aplicações em redes sem fio devem oferecer alternativas à arquitetura cliente/servidor. Reconhece-se que Lime constitui um esforço neste sentido. No entanto, conforme mostrado no capítulo, a noção de espaços de tuplas federados possui um custo de implementação considerável, o qual impacta o desempenho e a escalabilidade de aplicações construídas usando-se o sistema. Generalizando, acredita-se que a implementação de estruturas de dados distribuídas em diversos nodos não é recomendável em ambientes dinâmicos como redes sem fio.

No próximo capítulo, apresenta-se um modelo de coordenação baseado em espaço de tuplas que substitui a arquitetura cliente/servidor, usada em implementações tradicionais de Linda, por uma arquitetura *peer-to-peer*. A formalização deste modelo utiliza a versão do Cálculo de Ambientes dotada de comunicação via espaço de tuplas proposta neste capítulo. No modelo de coordenação a ser proposto, cada nodo da rede possui seu próprio espaço de tuplas. Ao contrário de Lime, no entanto, o modelo não oferece nenhuma primitiva cuja implementação requer alguma forma de sincronização distribuída.

Capítulo 5

Coordenação de Aplicações em Redes sem Fio

5.1 Introdução

Descreve-se neste capítulo um modelo baseado em espaços de tuplas para coordenação de aplicações em redes sem fio. O uso do modelo proposto, chamado de PeerSpaces, é particularmente interessante em redes *ad hoc*, já que o mesmo não pressupõe a existência de nenhuma estrutura centralizada. A fim de atender a este requisito, PeerSpaces difere-se de implementações tradicionais de Linda pelo fato de não ser baseado na arquitetura cliente/servidor, mas sim em uma arquitetura *peer-to-peer*. Em PeerSpaces, assume-se que cada nodo de uma rede sem fio possui seu próprio espaço de tuplas, o qual pode ser acessado tanto por aplicações locais a este nodo, como por aplicações remotas. Logo, nodos no sistema desempenham tanto papéis de clientes como de servidores de espaços de tuplas, não existindo diferenças fundamentais entre os mesmos.

Além das três primitivas clássicas de modelos baseados em espaços de tuplas (**out** , **in** e **rd**), PeerSpaces possui uma primitiva extra, chamada de **find**. Esta primitiva possui basicamente duas utilizações. A primeira delas é para localizar serviços disponíveis nos nodos de uma rede, como, por exemplo, uma impressora ou um sistema de arquivos. Esta localização, no entanto, não requer nenhuma forma de sincronização distribuída ou um conhecimento prévio sobre a topologia da rede. O segundo uso da primitiva **find** é para implementação de consultas contínuas, isto é, consultas que permanecem ativas enquanto um determinado serviço não é disponibilizado em um nodo da rede. Consultas contínuas introduzem reatividade em PeerSpaces, isto é, a capacidade de detectar e reagir a mudanças no estado de espaços de tuplas remotos. Por último, PeerSpaces assume que os nodos de uma rede sem fio são logicamente organizados em grupos, de forma a permitir que aplicações possam restringir o escopo de uma consulta realizada via primitiva **find**.

O restante deste capítulo está organizado conforme descrito a seguir. Na Seção 5.2, apresenta-se informalmente o modelo de coordenação proposto, enfatizando seus principais conceitos e

primitivas. A Seção 5.3 apresenta a semântica formal de PeerSpaces, utilizando para isso uma linguagem concorrente baseada no Cálculo de Ambientes. Em seguida, na Seção 5.4, provam-se algumas propriedades do modelo proposto. A Seção 5.5 descreve a utilização de PeerSpaces em redes móveis infra-estruturadas. A Seção 5.6 apresenta um protótipo de um sistema que implementa o modelo de coordenação proposto em PeerSpaces. Finalizando o capítulo, a Seção 5.7 realiza uma análise crítica do modelo proposto, comparando o mesmo com trabalhos semelhantes e relacionando suas principais contribuições.

5.2 Modelo de Coordenação Proposto

PeerSpaces assume uma rede *ad hoc* conectando dispositivos computacionais móveis. Portanto, considera-se que não existe nenhum suporte de uma infra-estrutura física de comunicação e que nodos podem se conectar e desconectar da rede a qualquer momento. Como usual em cenários de comunicação *ad hoc*, dois nodos podem trocar mensagens quando suas interfaces de rede estão uma ao alcance da outra. O modelo também não pressupõe a existência de nenhuma estrutura centralizada ou de uma memória compartilhada distribuída abrangendo o conjunto de nodos da rede. Em vez disso, assume-se um arquitetura de comunicação *peer-to-peer*, onde todos os nodos possuem as mesmas capacidades de comunicação e as mesmas responsabilidades de processamento.

Os principais conceitos usados em PeerSpaces são nodos, serviços, grupos e rede, conforme descrito a seguir:

Nodos: PeerSpaces considera que os nodos de uma rede são computadores móveis. Todo nodo possui um espaço de tuplas local, acessado conforme definido em Linda, e um processo em execução. Um nodo é denotado por $h_g[P, T]$, onde h é o nome do nodo, P é o processo em execução no mesmo, T é seu espaço de tuplas local e g é o grupo ao qual o nodo pertence.

O espaço de tuplas local de um nodo possui três tipos de usos. Primeiramente, ele é usado para coordenação entre os processos em execução neste nodo, conforme usual em sistemas inspirados em Linda. Além disso, um espaço de tuplas é usado para troca de mensagens e sincronização entre processos executando em nodos distintos. Para isso, PeerSpaces disponibiliza variantes das primitivas tradicionais de Linda que são capazes de operar em espaços de tuplas remotos. Por último, um espaço de tuplas local é usado para se publicar serviços disponíveis em um nodo e para armazenar resultados de consultas de serviços, conforme descrito a seguir.

PeerSpaces requer que o nome de um nodo seja diferente dos nomes dos demais nodos de uma rede. O modelo supõe ainda que nomes de nodos são elementos de um conjunto infinito H.

Serviços: Um serviço é qualquer entidade disponível em um nodo que pode ser útil a outros nodos da rede, como um arquivo, um dispositivo de *hardware*, um *software* etc. Em PeerSpaces, serviços são tornados públicos na rede inserindo uma tupla no espaço local descrevendo seus

atributos e sua localização. Por exemplo, um serviço de impressão pode ter como atributos seu nome, sua tecnologia de impressão (laser, jato de tinta etc), sua velocidade de impressão etc. A localização de um serviço é o nome do nodo que disponibiliza o mesmo. Por último, uma consulta de serviços disponíveis (ou *lookup query*) é uma consulta com intuito de descobrir a localização de um determinado serviço em algum nodo da rede.

Grupos: PeerSpaces assume que os nodos de uma rede *ad hoc* são logicamente organizados em grupos. Grupos possuem um nome, um conjunto de nodos e um conjunto de subgrupos, isto é, grupos são hierarquicamente organizados, formando uma árvore. O grupo de um nodo é denotado por uma tupla da forma $\langle g_1, \ldots, g_n \rangle$, a qual especifica o caminho do grupo raiz g_1 até o grupo folha g_n onde o nodo está localizado. Por exemplo, a tupla $\langle \texttt{ufmg}, \texttt{cs}, \texttt{proglab} \rangle$ denota o conjunto de nodos no grupo `proglab`, o qual é um subgrupo do grupo `cs`, que por sua vez, é um subgrupo do grupo raiz `ufmg`. Dois grupos podem ter o mesmo nome, desde que não sejam subgrupos de um mesmo grupo.

Grupos são um conceito natural em redes *ad hoc*, já que as mesmas são normalmente utilizadas para coordenar equipes de trabalho, como em um campo de batalha, em uma situação de emergência ou em um determinado evento. Particularmente, em PeerSpaces grupos são usados para delimitar o escopo de consultas de serviços. Sempre que possível, o objetivo é restringir a pesquisa de serviços disponíveis aos nodos de um grupo específico da rede.

Rede: Em PeerSpaces, nodos são conectados por meio de uma rede sem fio *ad hoc*. Como usual nestas redes, conectividade entre nodos é transiente, sendo determinada em função das distâncias entre os mesmos. Como nodos são móveis e autônomos, a topologia da rede encontra-se em constante alteração. Além disso, considera-se que determinados nodos podem funcionar como roteadores, propagando mensagens entre nodos que não estão diretamente conectados. Em PeerSpaces, uma rede com nodos $h_1, h_2, \ldots, h_n$ é denotada por:

$$h_{1g_1}[P_1, T_1] \mid h_{2g_2}[P_2, T_2] \mid \quad \ldots \quad \mid h_{ng_n}[P_n, T_n] \ , \ E$$

onde $g_1, g_2, \ldots, g_n$ são os grupos destes nodos e E: $H \times H$ é uma relação representando o mapa de conectividade da rede. A presença de um par ordenado (h_i, h_j) em E, denotada por $h_i \bowtie h_j$, indica que o nodo h_j é alcançável a partir do nodo h_i, isto é, que existe uma rota de h_i para h_j. Como em redes *ad hoc* é possível a existência de canais de comunicação unidirecionais, a relação acima não é simétrica.

PeerSpaces define ainda um conjunto de primitivas para construção de aplicações usando os conceitos descritos anteriormente. Estas primitivas são utilizadas nos seguintes tipos de operações: operações locais, mobilidade de processos, operações remotas, localização de serviços e consultas contínuas. O restante desta seção é dedicado a descrever cada uma destas operações.

Operações locais: O espaço de tuplas local de um nodo é acessado usando as tradicionais primitivas **out**, **in** e **rd** de Linda. Além disso, define-se uma primitiva **chgrp** g, usada para alterar o grupo de um nodo para aquele especificado pela tupla g.

Mobilidade de Processos: Apesar de não ser um conceito central no modelo, PeerSpaces assume que processos são móveis. Basicamente, mobilidade de processos foi adicionada ao modelo para possibilitar a implementação de primitivas para acesso a espaços de tuplas remotos. Evidentemente, este conceito pode ser também usado para implementar aplicações móveis. Assim, a primitiva **move** $h.P$ é usada para mover um processo para o nodo h, onde sua execução prossegue como se fosse o processo P. Se o nodo h não estiver acessível, a operação permanece bloqueada.

Operações Remotas: O projeto de primitivas para realização de operações remotas é crucial para escalabilidade e desempenho de qualquer modelo de coordenação. Portanto, desde o início de seu projeto, decidiu-se que PeerSpaces não ofereceria nenhuma estrutura cujo objetivo fosse prover transparência de localização no acesso aos diversos espaços de tuplas da rede. Em vez disso, o modelo disponibiliza variantes das primitivas tradicionais de Linda que operam no espaço de tuplas de um nodo previamente conhecido h. Assim, as primitivas **out** h, v; **in** h, v, x e **rd** h, v, x são oferecidas para realização de operações remotas. Nestas primitivas, v denota uma tupla ou um padrão e x uma variável.

A operação remota **out** h, v é usada quando um processo deseja transmitir uma informação para ser consumida em um outro nodo. Assim como sua versão local, esta primitiva é assíncrona. A fim de modelar este assincronismo, a operação é executada em dois passos. No primeiro passo, um *tag* é associado à tupla v, a fim de indicar que ela deve ser transferida assim que possível para o host h. A tupla rotulada resultante deste processo, denotada por v_h, é então depositada no espaço de tuplas do host h' que solicitou a operação. O segundo passo consiste em propagar a tupla v_h para o nodo h assim que o mesmo estiver conectado a h', removendo também o *tag* associado à mesma. Como os dois passos não são atômicos, enquanto a tupla não for transferida para o nodo h, ela pode ser removida de h' por meio de uma operação **in** v_h. Esta operação pode ser chamada, por exemplo, por um processo coletor de lixo, responsável por remover tuplas que não foram propagadas para seu destino final após um certo intervalo de tempo.

Como suas correspondentes locais, as primitivas remotas **in** h, v, x e **rd** h, v, x são síncronas e, portanto, bloqueiam a execução até que o o nodo h esteja conectado e uma tupla compatível com p esteja disponível no espaço de tuplas deste nodo. Basicamente, estas primitivas são usadas quando um processo necessita de uma informação localizada em um nodo remoto para prosseguir sua execução.

Localização de Serviços: Sem uma operação para localização de serviços, as primitivas remotas descritas acima possuem pouco uso, já que um nodo pode não conhecer previamente os

provedores de serviços de que necessita ao migrar para uma nova rede. Além disso, localização de serviços não deve ser centralizada em um único nodo, mas distribuída ao longo da rede. A fim de atender a estes requisitos, disponibiliza-se em PeerSpaces a seguinte primitiva: **find** g, p. Esta primitiva procura em nodos do grupo g por tuplas compatíveis com o padrão p. Todas as tuplas encontradas são copiadas assincronamente para o espaço de tuplas local do nodo que solicitou a operação.

Suponha uma aplicação de leilão para uso em dispositivos computacionais móveis. O usuário desta aplicação pode, por exemplo, estar interessado em comprar uma televisão de 21 polegadas, independentemente da marca, preço e do vendedor da mesma. A fim de descobrir quais nodos participantes do leilão possuem uma televisão à venda com estas características, deve-se executar a seguinte operação: **find** $\langle \texttt{mall}, \texttt{sellers} \rangle, \langle \texttt{tv}, 21, ?, ?, ? \rangle$. Esta operação dispara uma consulta por tuplas compatíveis com o padrão $\langle \texttt{tv}, 21, ?, ?, ? \rangle$ nos nodos da rede pertencentes ao grupo `sellers`, o qual é um subgrupo de `mall`. Tuplas compatíveis com este padrão, tais como $\langle \texttt{tv}, 21, \texttt{foo}, 325, \texttt{h} \rangle$, onde `foo`, `325` e `h` são, respectivamente, a marca, o preço e o nodo da rede ofertando a TV, serão inseridas assincronamente no espaço de tuplas do nodo que solicitou o **find** em um instante de tempo qualquer após a ativação do mesmo. O nodo solicitante pode então recuperar respostas a sua consulta usando a primitiva local **in** e realizar uma oferta de compra por meio de uma operação **out** h, v.

A semântica de PeerSpaces não predefine um algoritmo específico para propagação de consultas de serviços. No entanto, exige-se que qualquer algoritmo que venha a ser utilizado em uma implementação real do modelo possua as seguintes propriedades:

- Cobertura: o algoritmo deve ser capaz de propagar uma consulta para qualquer nodo que pertença ao grupo de destino da mesma e que esteja conectado à rede.
- Ausência de ciclos: o algoritmo não deve permitir que consultas fiquem circulando indefinidamente na rede e que sejam executadas mais de uma vez em um mesmo nodo.

Além disso, propagação de consultas não deve exigir nenhum grau de sincronização distribuída. Basicamente, propagação de consultas deve concorrer e ser executada de forma intercalada com outras operações em espaços de tuplas.

Diversos algoritmos atendem aos requisitos acima. O mais simples deles baseia-se na técnica de *flooding* (ou inundação) [Tan96]. Basicamente, neste algoritmo o nodo que originou a consulta propaga a mesma para seus vizinhos, que voltam a propagá-la para seus vizinhos e assim sucessivamente, até que todo o grafo que representa a rede tenha sido coberto. Este algoritmo tem a desvantagem de gerar um tráfego considerável de mensagens. No entanto, o mesmo é bastante robusto a reconfigurações na rede, já que um nodo precisa conhecer apenas seus vizinhos imediatos. Existe ainda um esforço de pesquisa no sentindo de se projetar algoritmos de *multicast* para redes *ad hoc* que gerem um menor número de mensagens na rede [Gio02, LSH+00]. Em geral, estes algoritmos também atendem aos requisitos acima.

Consultas Contínuas: Quando a consulta por um serviço em um determinado nodo falha, pode ser interessante mantê-la ativa a fim de detectar uma possível disponibilização futura do serviço neste mesmo nodo. Assim, evita-se a execução de repetidas consultas a fim de descobrir a inserção de um novo serviço na rede. Em PeerSpaces, consultas que permanecem ativas mesmo em caso de inexistência do serviço demandado são chamadas de consultas contínuas.

Uma primeira questão fundamental no projeto de consultas contínuas diz respeito à forma com que a execução das mesmas é encerrada. A alternativa de se adicionar uma primitiva para explicitamente revogar consultas contínuas não é razoável em cenários de computação móvel, já que reconfigurações na rede podem desconectar o nodo que solicitou a consulta dos nodos responsáveis pela sua execução. Por este motivo, optou-se por permitir a associação de um tempo de vida a uma consulta de serviço. Quando este tempo expira, a consulta é automaticamente cancelada. Consultas contínuas são então submetidas por meio da primitiva **find** g, p, t, onde t é o tempo de vida da mesma.

Uma segunda questão importante no projeto de consultas contínuas é o tratamento de conexões de novos nodos a um grupo da rede. Neste caso, o conjunto de consultas neste grupo e no novo nodo devem ser sincronizados. Por exemplo, suponha a conexão de um novo nodo h em um grupo g da rede. Qualquer consulta localizada em h e que ainda não exista em g deve ser propagada para este grupo. O mesmo ocorre com consultas existentes no grupo g e que não existam em h. Quando uma consulta contínua é propagada de um nodo para outro da rede, o seu tempo de vida restante é transferido junto com a mesma.

5.3 Semântica Formal

Descreve-se nesta seção a semântica do modelo de coordenação proposto por PeerSpaces. A formalização segue um estilo operacional, sendo baseada na versão do Cálculo de Ambientes dotada de comunicação via espaço de tuplas descrita no Capítulo 4. Basicamente, esta versão incorpora em ambientes um espaço local de tuplas, o qual é acessado por meio das tradicionais primitivas de Linda. Com isso, torna-se natural modelar os nodos de uma configuração em PeerSpaces como ambientes.

A Tabela 5.1 apresenta a sintaxe da linguagem de coordenação proposta em PeerSpaces. Assume-se um conjunto infinito de nomes H, usados para identificar ambientes e consultas de serviços. As meta-variáveis h, k e x armazenam elementos de H. Valores básicos, denotados por v e g, são nomes ou tuplas. Tuplas são seqüências ordenadas de valores, denotadas por $\langle v_1, \ldots, v_n \rangle$. Um espaço de tuplas T é um multi-conjunto de tuplas. Utiliza-se o símbolo $? \in H$ para denotar um valor qualquer.

Um programa é composto por uma rede N, a relação E e um conjunto global de nomes X. A relação $E : H \times H$ fornece um mapa de conectividade da rede. Nomes usados em diversos nodos do sistema são armazenados em X, com o intuito de garantir a unicidade dos mesmos. O

$$
\begin{array}{rcl}
Prog & ::= & N\,,\,E\,,\,X \\
N & ::= & \varepsilon \mid H \mid N \\
H & ::= & h_g[P,\,T] \mid (\nu\,x)\,h_g[P,\,T] \\
P,Q & ::= & \mathbf{0} \mid P \mid Q \mid\, !P \mid (\nu\,x)\,P \mid \mathbf{out_L}\ v \mid \mathbf{in_L}\ v, x.P \mid \mathbf{rd_L}\ v, x.P \mid \\
 & & \mathbf{find}\ g, p, t \mid \mathbf{chgrp}\ g \mid \mathbf{move}\ h.P
\end{array}
$$

Tabela 5.1: Sintaxe da linguagem de coordenação de PeerSpaces

ambiente $h_g[P,\,T]$ denota um nodo h que é membro de um grupo g e que possui um processo em execução P e um espaço de tuplas local T. Processos são denotados pelos não-terminais P e Q.

De forma idêntica ao Cálculo de Ambientes, a expressão mais simples desta linguagem é o processo inerte **0**, o qual denota um processo sem nenhum comportamento. O termo $P \mid Q$ representa dois processos executando em paralelo. O termo $!P$ designa infinitas cópias em paralelo do processo P. O operador de restrição $(\nu\,x)\,P$ assegura que x é um nome único no escopo de P. De forma similar a Linda, as primitivas $\mathbf{out_L}$, $\mathbf{in_L}$ e $\mathbf{rd_L}$ permitem acesso ao espaço de tuplas local de um nodo. Conforme definido no Capítulo 4, estas primitivas possuem um *tag* **L** para diferenciá-las de operações sinônimas no Cálculo de Ambientes. Como a operação $\mathbf{out_L}$ é assíncrona, ela não possui uma continuação P. O mesmo ocorre com as primitivas $\mathbf{find}\ g, p, t$ e $\mathbf{chgrp}\ g$. Considera-se que consultas de serviços não-contínuas são simuladas definindo um tempo de vida zero. Por último, a primitiva **move** é usada para alterar a localização da sua continuação P.

A semântica operacional de PeerSpaces é descrita na Tabela 5.2. Como usual no Cálculo de Ambientes, a semântica do modelo é definida em termos de reduções e de regras de congruência estrutural entre processos. A redução $N\,,\,E\,,\,X \;\rightarrow\; N'\,,\,E'\,,\,X'$ define como a configuração $N\,,\,E\,,\,X$ reduz em um único passo de computação em $N'\,,\,E'\,,\,X'$.

Inicialmente, existem três reduções descrevendo as primitivas tradicionais de Linda. Estas reduções são idênticas àquelas definidas no Capítulo 4. Em seguida, define-se a semântica das primitivas próprias de PeerSpaces. A primitiva $\mathbf{find}\,g', p, t$ insere uma tupla representando uma consulta de serviço no espaço local (regra P1). Esta tupla possui o formato $\langle k, g', p, t, h\rangle$, onde k é o identificador da consulta, g' é o grupo de nodos onde a consulta será executada, p é um padrão para o serviço a ser pesquisado, t é o tempo de vida da consulta e h é o nome do nodo que solicitou a operação. A primitiva $\mathbf{chgrp}\ g$ apenas altera o grupo do nodo corrente para aquele especificado pela tupla g (regra P2). Se tal grupo não existir, o mesmo é automaticamente criado. A primitiva $\mathbf{move}\ h'.P$ altera a localização da continuação P para o nodo h', caso o mesmo esteja conectado à rede (regra P3). Caso contrário, a operação permanece bloqueada até a conexão do mesmo.

A redução Q1 define como consultas de serviços são propagadas na rede. Basicamente, qual-

Reduções

Primitivas de Linda

$h_g[\mathbf{out_L}\ v \mid P, T] \mid N, E, X \ \rightarrow\ h_g[P, v \cup T] \mid N, E, X$ (L1)

$h_g[\mathbf{in_L}\ v, x.P \mid Q, v' \cup T] \mid N, E, X \ \rightarrow\ h_g[P\{v'/x\} \mid Q, T] \mid N, E, X$ (L2)

$h_g[\mathbf{rd_L}\ v, x.P \mid Q, v' \cup T] \mid N, E, X \ \rightarrow\ h_g[P\{v'/x\} \mid Q, v' \cup T] \mid N, E, X$ (L3)

Primitivas de PeerSpaces

$h_g[\mathbf{find}\ g', p, t \mid P, T] \mid N, E, X \ \rightarrow\ (\nu\, k)\, h_g[P, \langle k, g', p, t, h\rangle \cup T] \mid N, E, X$ (P1)

$h_g[\mathbf{chgrp}\ g' \mid P, T] \mid N, E, X \ \rightarrow\ h_{g'}[P, T] \mid N, E, X$ (P2)

$h_g[\mathbf{move}\ h'.P \mid Q, T] \mid h'_{g'}[P', T'] \mid N, E, X \ \rightarrow$
$h_g[Q, T] \mid h'_{g'}[P \mid P', T'] \mid N, E, X$ (P3)

Propagação de Consultas

$h_g[P, \langle k, g'', p, t, h\rangle \cup T] \mid h'_{g'}[P', T'] \mid N, E, X \ \rightarrow$
$h_g[P, \langle k, g'', p, t, h\rangle \cup T] \mid h'_{g'}[P' \mid P'', \langle k, g'', p, t, h\rangle \cup T'] \mid N, E, X$ (Q1)

Reconfigurações na Rede

$$\frac{E \ \Rightarrow\ E'}{N, E, X \ \rightarrow\ N, E', X} \quad \text{(N1)}$$

As regras acima são sujeitas às seguintes condições:
(L2) if $v' \leq v$
(L3) if $v' \leq v$
(P3) if $h \bowtie h'$
(Q1) if $(h \bowtie h') \wedge (g'' \preceq g') \wedge (\langle k, g'', p, t, h\rangle \notin T') \wedge P'' = !\,(\mathbf{rd_L}\ p, x.\mathbf{out_L}\ h, x)$

Congruência Estrutural

$P \mid Q \equiv Q \mid P$	(SC1)	$(\nu\, x)\,(\nu\, y)\, P \equiv (\nu\, y)\,(\nu\, x)\, P$	(SC5)
$!\, P \equiv P \mid !\, P$	(SC2)	$P \equiv Q \Rightarrow (\nu\, x)\, P \equiv (\nu\, x)\, Q$	(SC6)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(SC3)	$(\nu\, x)\,(P \mid Q) \equiv P \mid (\nu\, x)\, Q$	(SC7)
$P \mid \mathbf{0} \equiv P$	(SC4)		

$P \equiv Q \Rightarrow n\,[P, T\,] \equiv n\,[Q, T\,]$ (SC8)

$(\nu\, x)\, n\,[P, T\,] \equiv n\,[\,(\nu\, x)\, P, T\,]$ (SC9)

$(\nu\, x)\,(n\,[P, T\,] \mid n'\,[P', T'\,]) \equiv (\nu\, x)\, n\,[P, T\,] \mid n'\,[P', T'\,]$ (SC10)

$(\nu\, x)\, h_g[P, T], E, X \equiv h_g[P, T], E, x \cup X$ (SC11)

As regras acima são sujeitas às seguintes condições:
(SC7) if $x \notin fn(P)$ (SC10) if $x \neq h',\ x \notin fn(P')$
(SC9) if $x \neq h$ (SC11) if $x \notin X$

Compatibilidade de Grupos

$$\frac{g_1 = g'_1\ \ldots\ g_n = g'_n}{\langle g_1\ \ldots\ g_n\rangle \preceq \langle g'_1\ \ldots\ g'_n \ldots\ g'_m\rangle}$$

Tabela 5.2: Semântica Operacional de PeerSpaces

quer nodo que contenha uma consulta $\langle k, g'', p, t, h\rangle$ pode propagá-la para um nodo h' conectado ao mesmo, desde que o grupo g'' de destino da consulta seja compatível com o grupo g' deste nodo. Se esta condição for satisfeita, a consulta é inserida no espaço de tuplas de h' e um processo P'' é adicionado em paralelo aos demais processos em execução neste nodo. O processo P'' continuamente recupera tuplas que sejam compatíveis com o padrão p e então usa uma operação de saída remota para depositar a tupla lida no espaço de tuplas do nodo h que disparou a consulta. Como pode ser verificado na semântica, propagação de consultas pode ser intercalada com qualquer número de reduções representando outras operações do modelo. Além disso, como consultas de serviços são armazenadas nos espaços persistentes de tuplas, as mesmas serão também automaticamente propagadas para novos nodos que se conectem à rede.

A regra N1 introduz uma redução do tipo $\Rightarrow$, a qual é usada para descrever reconfigurações na rede e, conseqüentemente, alterações na relação de conectividade E. Basicamente, esta regra define que atualizações em E devem ser refletidas na configuração corrente do sistema. No entanto, reduções do tipo $\Rightarrow$ não são especificadas na semântica, já que as mesmas dependem de parâmetros tecnológicos da rede, tais como alcance das interfaces de comunicação de cada nodo, protocolos de roteamento empregados etc.

O conjunto seguinte de regras define uma relação de congruência estrutural, representada por $\equiv$. Esta relação aplica-se tanto a processos (regras SC1 a SC7) como a ambientes (regras SC8 a SC11).

Existe ainda um conjunto de regras para compatibilidade de grupos. Os grupos g e g' são compatíveis, denotado por $g \preceq g'$, se todos os subgrupos de g são iguais aos respectivos subgrupos de g'. O grupo g' pode ter ainda um número extra de subgrupos aninhados. Por exemplo, $\langle \texttt{ufmg}, \texttt{cs}\rangle \preceq \langle \texttt{ufmg}, \texttt{cs}, \texttt{proglab}\rangle$, significando que consultas destinadas a $\langle \texttt{ufmg}, \texttt{cs}\rangle$ serão também executadas em nodos localizados no grupo $\langle \texttt{ufmg}, \texttt{cs}, \texttt{proglab}\rangle$. De forma similar, $\langle \texttt{ufmg}\rangle \preceq \langle \texttt{ufmg}, \texttt{cs}, \texttt{proglab}\rangle$, mas $\langle \texttt{ufmg}, \texttt{eng}\rangle \not\preceq \langle \texttt{ufmg}, \texttt{cs}, \texttt{proglab}\rangle$. A flexibilidade extra existente para compatibilidade de grupos torna dispensável o uso do símbolo ? neste tipo de regra.

5.3.1 Operações Remotas

A Tabela 5.3 descreve a semântica de operações remotas em PeerSpaces. Inicialmente, a operação remota $\mathbf{out_L}\ h', v$ simplesmente deposita a tupla v no espaço local com um *tag* h' (regra R1). A regra R2 se encarrega de propagar esta tupla para seu destino final h', quando este se encontrar conectado à rede. Quando de sua inserção no espaço final, o *tag* é removida da tupla. Já a semântica das primitivas $\mathbf{in_L}$ e $\mathbf{rd_L}$ é dada traduzindo as mesmas para um processo que se move para o destino final a fim de realizar a operação solicitada. Por exemplo, na primitiva $\mathbf{in_L}\ h', p, x$, um processo se move para o nodo h', onde executa localmente um $\mathbf{in_L}\ p, y$. Quando uma tupla compatível for encontrada, o processo retorna ao nodo h que solicitou a operação e insere no mesmo uma tupla com uma chave k que identifica a operação e o valor removido y.

Operação $\mathbf{out_L}$ Remota

$h_g[\mathbf{out_L}\ h', v \mid P, T] \mid N, E, X \ \rightarrow\ h_g[P, v_{h'} \cup T] \mid N, E, X$ (R1)

$h_g[P, v_{h'} \cup T] \mid h'_{g'}[P', T'] \mid N, E, X \ \rightarrow\ h_g[P, T] \mid h'_{g'}[P', v \cup T'] \mid N, E, X, \quad \text{if } h \bowtie h'$ (R2)

Operações $\mathbf{in_L}$ e $\mathbf{rd_L}$ Remotas

$h_g[\mathbf{in_L}\ h', p, x.P \mid P', T] \mid N, E, X \ \rightarrow$

$\quad h_g[(\nu\, k)\,(\mathbf{move}\ h'.\mathbf{in_L}\ p, y.\mathbf{move}\ h.\mathbf{out_L}\ \langle k, y\rangle \mid \mathbf{in_L}\ \langle k, p\rangle, x.P) \mid P', T], E, X$ (R3)

$h_g[\mathbf{rd_L}\ h', p, x.P \mid P', T] \mid N, E, X \ \rightarrow$

$\quad h_g[(\nu\, k)\,(\mathbf{move}\ h'.\mathbf{rd_L}\ p, y.\mathbf{move}\ h.\mathbf{out_L}\ \langle k, y\rangle \mid \mathbf{in_L}\ \langle k, p\rangle, x.P) \mid P', T], E, X$ (R4)

Tabela 5.3: Semântica de operações remotas

Um processo paralelo, instalado quando a operação teve início, remove esta tupla e a execução prossegue como P. A semântica de um $\mathbf{rd_L}$ remoto é similar, excetuando-se o fato de que a tupla não é removida.

5.3.2 Coleta de Lixo

Com o intuito de revogar consultas contínuas, deve existir em todo nodo um processo que continuamente decremente o tempo de vida das consultas armazenadas em seu espaço de tuplas. Se o tempo de vida de uma consulta chegar a zero, este processo deve remover a consulta do espaço local e cancelar o processo responsável por executar a mesma (processo P'' da regra Q1, na Tabela 5.2). A fim de preservar a simplicidade e legibilidade da semântica, decidiu-se não especificar o comportamento deste processo coletor de lixo.

5.4 Propriedades do Modelo

Prova-se a seguir duas propriedades fundamentais do modelo de coordenação proposto por PeerSpaces:

Proposição 1 (Cobertura) *Antes de expirar, uma consulta de serviço pode ser propagada para qualquer nodo pertencente a seu grupo de destino que esteja conectado à rede.*

Prova: Diretamente da condição da regra Q1 (Tabela 5.2), a qual garante que uma consulta pode ser propagada para qualquer nodo h' alcançável a partir do nodo de origem h e pertencente ao grupo de destino da consulta.

Proposição 2 (Ausência de Ciclos) *Não há ciclos na propagação de consultas de serviços.*

Prova: Por indução no tamanho dos ciclos.

- Base: O menor ciclo possível em uma rede possui tamanho dois. Assim, considere um ciclo mínimo conectando os nodos h_1 e h_2. Considere ainda uma consulta solicitada pelo nodo h_1. De forma direta, a regra Q1 (Tabela 5.2) pode ser usada para propagar a consulta de h_1 para h_2. No entanto, a condição desta mesma regra impede a propagação da consulta de volta para h_1, já que a mesma já existe no espaço local deste nodo.

- Hipótese Indutiva: Não existem ciclos na propagação de consultas em uma rede cujos ciclos são menores ou iguais a n.

- Passo Indutivo: Suponha uma rede com um ciclo de tamanho $n+1$. A regra Q_1 pode ser usada repetidas vezes para propagar uma consulta para os nodos $h_1, \ldots, h_{n+1}$ deste ciclo. Pode-se ainda construir um ciclo de tamanho n estabelecendo uma conexão entre h_n e h_1. No entanto, pela hipótese indutiva, não se pode propagar a consulta de h_n para h_1. Como a consulta é a mesma, ela também não poderá ser propagada de h_{n+1} para h_1. Logo, não se pode criar um ciclo de propagação de tamanho $n+1$.

A proposição acima é fundamental para assegurar que consultas de serviços não são propagadas indefinidamente na rede. No entanto, a fim de evitar tais ciclos de propagação, a condição da regra Q1 obriga um nodo a manter armazenadas as chaves das consultas já solicitadas pelo mesmo.

5.5 Coordenação em Redes Infra-estruturadas

A arquitetura descentralizada adotada em PeerSpaces é particularmente recomendada para coordenação de aplicações em redes móveis *ad hoc*. No entanto, com algum esforço, o modelo pode ser adaptado para uso em redes móveis com suporte de uma infra-estrutura física de comunicação. Basicamente, esta adaptação implica em uma mudança na arquitetura utilizada para localização de serviços. Quando operando em modo *ad hoc*, localização de serviços em PeerSpaces é totalmente distribuída e baseada em uma arquitetura *peer-to-peer*. Já em redes infra-estruturadas, a tarefa de localização de serviços deve ser transferida para a rede fixa, de forma a reduzir a carga de processamento nos computadores móveis e a utilização dos canais de comunicação sem fio. Assim, no restante desta seção, descrevem-se as alterações que devem ser realizadas em PeerSpaces para que o mesmo suporte também coordenação em redes infra-estruturadas.

Quando operando em redes infra-estruturadas, PeerSpaces assume que nodos anunciam os serviços disponibilizados pelos mesmos não mais em seus espaços de tuplas locais, mas em um espaço de tuplas localizado na rede fixa, o qual é denominado de espaço de serviços. Este espaço centralizado desempenha, portanto, o papel de um repositório dos serviços disponíveis tanto na rede fixa como nos dispositivos móveis da rede. Assim, uma configuração em redes

infra-estruturadas é dada por uma quádrupla N, E, X, h_s, onde N, E e X possuem os mesmos significados que em redes *ad hoc* e h_s é o nodo onde está localizado o espaço de serviços.

Um nodo h pertencente ao grupo g publica a disponibilidade por t unidades de tempo de um serviço descrito pela tupla p executando a seguinte operação no espaço de serviços: **out** $h_s, \langle g, p, t, h\rangle$. Um processo coletor de lixo localizado no espaço de serviços deve ser encarregado de decrementar o tempo de vida desta tupla e removê-la quando o mesmo expirar. Assim, serviços disponibilizados por um nodo serão automaticamente revogados em caso de desconexão não anunciada do mesmo. Além disso, para renovar o tempo de vida de um serviço p, acrescentando k unidades de tempo ao mesmo, basta que o nodo h execute a seguinte operação: **move** h_s.**in** $\langle g, p, ?, h\rangle, x$.**out** $\langle x.1, x.2, x.3+k, x.4\rangle$, onde $x.i$ é o i-ésimo campo da tupla associada à variável x. Esta operação desloca um processo para o espaço de serviços, o qual localmente remove a tupla com a descrição do serviço p e insere uma nova tupla, a qual acrescenta k unidades ao tempo deste serviço.

A primitiva **find** continua sendo usada para localização de serviços. No entanto, consultas realizadas pela mesma serão resolvidas no espaço de serviços da rede fixa, isto é, esta primitiva não mais necessita propagar uma consulta por todos os nodos da rede, como acontece no caso de redes *ad hoc*. Assim, a semântica da primitiva **find** g', p, t em redes infra-estruturadas consiste na seguinte operação de leitura no espaço de serviços: ! **rd** $h_s, \langle g', p, ?, ?\rangle, x$.**out** $\langle x.2, x.4\rangle$, onde $x.2$ e $x.3$ são, respectivamente, o segundo e o quarto campos da tupla associada à variável x. Expirado o tempo de vida t deste **find**, um processo coletor de lixos deve cancelar todos os processos remotos de leitura criados nesta operação. Observe-se ainda que as respostas da consulta iniciada por um **find** são inseridas assincronamente no espaço de tuplas local do nodo que solicitou a operação, da mesma forma que ocorre em redes *ad hoc*.

Em resumo, apesar de ter sido originalmente projetado para coordenação de aplicações em redes móveis *ad hoc*, PeerSpaces pode ser adaptado para operação em redes infra-estruturadas. Conforme descrito anteriormente, esta adaptação inclui a introdução de um espaço centralizado de serviços, localizado na rede fixa, e modificações na semântica das operações para publicação e consulta de serviços. Ressalte-se, no entanto, que a adaptação proposta pode não dispensar a necessidade de manutenções na lógica e nos algoritmos internos de uma aplicação quando a mesma é transportada de um ambiente de rede *ad hoc* para uma rede infra-estruturada, ou vice-versa. Por exemplo, quando se porta uma aplicação de um cenário de rede *ad hoc* para uma arquitetura infra-estruturada, pode ser recomendado ou até mandatório a transferência de algum tipo de processamento para a rede fixa. Por outro lado, no caso de se desejar portar uma aplicação de um ambiente infra-estruturado para um ambiente de rede *ad hoc*, pode ser necessário alterações em alguns dos algoritmos empregados na mesma. O motivo é que no primeiro ambiente admitem-se algoritmos centralizados e, no segundo ambiente, via de regra, é mais recomendável a adoção de algoritmos distribuídos.

5.6 Implementação

Descreve-se resumidamente nesta seção a implementação de um protótipo de um sistema que implementa o modelo de coordenação proposto por PeerSpaces. Nesta implementação, uma rede *ad hoc* é simulada por um conjunto de objetos distribuídos ao longo de uma rede local. Cada um destes objetos representa um nodo da rede; referências entre os mesmos representam conexões entre os nodos. A principal restrição desta implementação é a exigência de que conexões e desconexões sejam anunciadas. Particularmente, quando um nodo (objeto) se conecta à rede, ele deve informar quais são os seus vizinhos. Apesar desta restrição, a topologia simulada por esta implementação é mais próxima de uma rede *ad hoc* real do que aquela disponibilizada, por exemplo, em Lime. Na implementação atual de Lime, todos os nodos compartilham diretamente o mesmo barramento de uma rede local. Portanto, assume-se que um nodo pode acessar diretamente qualquer outro nodo da aplicação, sem utilização de nodos intermediários.

O protótipo foi implementado como um pacote em Java. A principal classe pública deste pacote é a classe `PeerSpaces`, a qual representa o espaço de tuplas de um dispositivo computacional móvel, conforme proposto no modelo. Esta classe possui a seguinte interface:

```
public class PeerSpaces {
    public void out(ITuple tuple);
    public void out(String host, ITuple tuple);
    public ITuple in(ITuple template);
    public ITuple in(String host, ITuple template);
    public ITuple read(ITuple template);
    public ITuple read(String host, ITuple template);
    public void chgrp(ITuple group);
    public void find(ITuple group, ITuple template);
    public void find(ITuple group, ITuple template, int lifetime);
    public void connect(String host);
    public void disconnect(String host);
}
```

A classe `PeerSpaces` inclui métodos para realizar as operações `out`, `in` e `read` em espaços de tuplas locais e remotos. Existem ainda métodos para alterar o grupo de um nodo (**chgrp**) e para realizar consultas de serviços (**find**). Os métodos `connect` e `disconnect` são usados para conectar e desconetar explicitamente um nodo da rede. O método **move** não é implementado pelo protótipo, já que a implementação padrão da JVM não oferece suporte para mobilidade de *threads*.

A implementação da operação **find** neste protótipo realiza um *flooding* na rede. Um nodo, tendo recebido uma consulta, propaga-a para os seus vizinhos e este processo repete-se até que todo grafo que representa a rede tenha sido coberto. Uma vez iniciada uma busca, esta alcançará todos os elementos da rede após, no máximo, $n - 1$ propagações. No entanto, este é

um caso particular, no qual o grafo de conexões se reduz a uma lista com n nodos. Além disso, as consultas geradas por um **find** possuem identificadores globais e únicos na rede, de forma a evitar ciclos na propagação das mesmas. Um nodo simplesmente descarta uma consulta caso já tenha sido recebido uma consulta anterior com mesmo identificador.

A implementação de consultas contínuas foi responsável por grande parte da complexidade do sistema. A fim de armazenar tuplas demandadas por consultas contínuas, foi acrescentado em todo nodo um espaço de tuplas temporário. As tuplas armazenadas nesta estrutura possuem dois campos extras: o nome do nodo que disparou a consulta e o tempo de vida restante das mesmas. A inserção de um tupla p no espaço local de um nodo h, via uma primitiva **out** , verifica inicialmente se existem uma ou mais tuplas p' compatíveis com p no espaço temporário. Caso exista, a tupla p é inserida localmente em h e, assincronamente, nos nodos que dispararam as consultas representadas pelas tuplas p'. A remoção de consultas contínuas cujo tempo de vida expirou é realizada por uma *thread* criada quando a consulta foi inserida no espaço de tuplas temporário.

O protótipo descrito nesta seção foi implementado usando Java RMI [Sun98] para comunicação entre objetos distribuídos e LighTS [Lig] como sistema de espaços de tuplas. Optou-se por esta implementação de Linda em detrimento de implementações mais tradicionais, como JavaSpaces e TSpaces, devido à sua simplicidade e tamanho reduzido. O pacote LighTS completo possui cerca de 11 Kb, o que certamente viabiliza o seu uso em dispositivos como um PDA.

Interface: Foi implementada também uma interface em forma de *shell* que possibilita o uso interativo do sistema. Esta interface possui uma linha de comando que permite que seus usuários executem todas as operações definidas em PeerSpaces. Mostra-se abaixo um exemplo de uma sessão de uso deste *shell*:

```
peer> connect vangogh
peer> connect rembrandt
peer> find cs, <"fax","3rd floor", ?>
......
peer> in <"fax","3rd floor", ?>, x
      x= <"fax", "3rd floor", "cezzane">
```

No exemplo acima, os dois primeiros comandos conectam o nodo corrente aos seus vizinhos `vangogh` e `rembrandt`. O comando seguinte procura por uma máquina de fax localizada no terceiro andar e conectada a um nodo do grupo `cs`. O último comando remove do espaço local uma resposta deste **find**.

Acredita-se que o protótipo descrito nesta seção possa ser usado para realizar pequenos experimentos com o objetivo de verificar a expressividade e escalabilidade do modelo de coordenação proposto em PeerSpaces. Como trabalho futuro, planeja-se portar a presente implementação

para computadores móveis conectados em uma rede *ad hoc* real. Pretende-se usar para isso a versão J2ME de Java [Sun00].

5.7 Análise Crítica

Aplicações distribuídas para dispositivos computacionais móveis conectados por redes sem fio *ad hoc* constituem a principal área de aplicação de PeerSpaces. As primitivas do modelo podem ser usadas em sistemas para computadores móveis, permitindo aos mesmos trocarem informações entre si, sincronizar tarefas e localizar serviços em redes *ad hoc*. Como exemplos de possíveis aplicações que podem se beneficiar dos recursos oferecidos pelo modelo, pode-se mencionar sistemas de arquivos distribuídos, aplicações de *workflow* e *groupware* e sistemas de mensagens instantâneas. No Capítulo 6, apresenta-se como estudo de caso um sistema para revisão de artigos submetidos a uma conferência, isto é, um sistema de *workflow* e *groupware*, que utiliza PeerSpaces como infra-estrutura de coordenação.

O restante desta seção encontra-se organizado conforme descrito a seguir. Na Subseção 5.7.1 avalia-se PeerSpaces segundo requisitos não-funcionais típicos de aplicações distribuídas. Em seguida, a Seção 5.7.2 compara o modelo proposto com outros trabalhos. Finalizando, a Subseção 5.7.3 descreve as principais contribuições do modelo de coordenação apresentado neste capítulo.

5.7.1 Avaliação

Avalia-se a seguir o modelo de coordenação proposto neste capítulo segundo requisitos não funcionais típicos de aplicações distribuídas:

- **Aderência às Características de Redes *Ad hoc*:** Os conceitos e as primitivas de coordenação de PeerSpaces foram projetados de forma a dispensar estruturas centralizadas e conexões estáticas entre computadores móveis. Em vez disso, o modelo propõe uma arquitetura de coordenação descentralizada e *peer-to-peer*. Cada dispositivo móvel possui seu próprio espaço de tuplas, usado para coordenação local e para divulgar serviços para outros nodos. Além disso, localização de serviços não demanda suporte de uma infra-estrutura fixa de comunicação ou conhecimento prévio da topologia da rede. Todas estas características são compatíveis com os princípios de comunicação em redes *ad hoc*.

- **Escalabilidade:** PeerSpaces não disponibiliza nenhuma estrutura de dados distribuída entre os nodos de uma rede sem fio. Em vez disso, o modelo oferece primitivas para acesso apenas a espaços de tuplas previamente conhecidos. Já a primitiva **find** não requer nenhuma forma de sincronização distribuída para propagar consultas de serviços. Por fim, o conceito de grupos lógicos de nodos fornece um mecanismo para restringir o escopo de consultas de serviços. Todas estas características contribuem para a escalabilidade do

modelo, principalmente quando comparado a sistemas baseados no conceito de espaços de tuplas virtuais, como Lime.

Por outro lado, o fato de consultas de serviços em PeerSpaces serem baseadas em comunicação *multicast* ou *broadcast* inviabiliza a utilização do modelo em redes *ad hoc* com grande número de nodos e distribuídas ao longo de uma ampla área geográfica. Prova disso são as críticas freqüentes sobre a quantidade de mensagens geradas em sistemas para compartilhamento de arquivos na Internet, como o Gnutella [Gnu]. Via de regra, estes sistemas são também baseados em *broadcast*. Assim, a utilização de PeerSpaces é recomendada apenas em redes *ad hoc* de pequeno ou médio porte, como aquelas utilizadas em campos de batalhas militares, em situações de emergência ou desastres naturais ou em eventos, como feiras e reuniões.

- **Transparência:** Modelos de programação distribuída para redes fixas tradicionalmente procuram tornar a existência da rede transparente às aplicações. Evidentemente, esta característica aumenta a expressividade destes modelos, já que programadores não precisam tratar problemas típicos de cenários distribuídos, como falhas parciais, latência das comunicações, desconexões, flutuações na largura de banda etc. No entanto, a fim de prover este grau de transparência, estes modelos assumem que o ambiente de rede subjacente é estável, que existe um limite superior para a latência, que existe sempre banda disponível na rede, que a topologia da rede é estática etc. Apesar de todas estas hipóteses serem válidas em redes fixas, o mesmo não ocorre em redes sem fio. Por este motivo, não existe em PeerSpaces nenhuma abstração de mais alto nível que tenha como objetivo tornar transparente a existência da rede e dos problemas típicos à mesma. Em vez disso, o modelo expõe a rede às aplicações e oferece primitivas de comunicação cuja implementação não requer níveis de qualidade de serviço tipicamente disponíveis apenas em redes locais.

- **Tolerância a Falhas:** Modelos de programação distribuída tradicionais tratam falhas de comunicação como exceções. No entanto, como estas falhas são eventos comuns em redes sem fio, PeerSpaces não segue a mesma estratégia. Assim, as operações **out** e **find** são assíncronas no modelo, isto é, as mesmas depositam uma mensagem no espaço local que será propagada para seu destino quando houver disponibilidade de conexão. Veja, no entanto, que, principalmente em se tratando de dispositivos com recursos escassos de processamento, o espaço de tuplas de um nodo possui um tamanho limitado. Assim, pode ocorrer de não se possuir mais espaço local para armazenar temporariamente tuplas geradas por um **out** ou **find**. Neste caso, resta a uma implementação do modelo sinalizar esta condição com uma exceção. Por último, PeerSpaces não cria dependências estáticas entre os nodos da rede, o que torna mais simples o tratamento de desconexões não anunciadas.

- **Reatividade:** A capacidade de reagir a mudanças é uma característica importante em ambientes dinâmicos como redes *ad hoc*. Assim, em PeerSpaces consultas contínuas são

usadas para detectar mudanças no estado de nodos remotos. Mais especificamente, consultas contínuas podem ser usadas para simular um modelo de eventos onde todo nodo funciona como originador e como receptor de eventos [CRW01]. A principal diferença desta estratégia para modelos tradicionais de eventos é a ausência de um nodo despachante (*dispatcher*), responsável por propagar a ocorrência de eventos entre o nodo originador e os nodos interessados nos mesmos. Veja que, como este despachante representa uma autoridade central, seu uso não é recomendável em redes móveis *ad hoc*.

- **Segurança:** Como redes *ad hoc* dão origem a federações espontâneas e abertas de nodos, segurança é uma preocupação constante [HBC01]. Assim, modelos baseados em espaços de tuplas para coordenação de aplicações nestas redes devem incluir mecanismos que permitam proteger tais espaços de computações maliciosas e errôneas. No entanto, como afirmado no Capítulo 1, o projeto de tais mecanismos de segurança ultrapassa o escopo desta Tese de Doutorado. Acredita-se porém que o sistema Secure Spaces [BOV99] constitui um ponto de partida interessante para qualquer trabalho que vise adicionar estes mecanismos em PeerSpaces, já que a principal contribuição do mesmo é a introdução de segurança em implementações centralizadas de Linda.

5.7.2 Comparação com Outros Modelos e Sistemas

Diversas das características de PeerSpaces foram inspiradas em sistemas para compartilhamento de arquivos na Internet, como Napster [Nas], Freenet [CSWH00] e Gnutella [Gnu]. Particularmente, a rede *peer-to-peer* criada pelo sistema Gnutella sobre a infra-estrutura fixa da Internet apresenta diversas propriedades bem vindas em cenários de computação móvel, tais como ausência de controle centralizado, auto-organização e adaptação a falhas. No entanto, no sistema Gnutella, uma consulta é constituída por apenas uma *string*, cuja interpretação pode variar de um nodo para outro da rede. Por outro lado, em PeerSpaces existe uma infra-estrutura de coordenação em cada ponto da rede, organizada na forma de um espaço de tuplas. Esta infra-estrutura permite especificar com maior precisão os atributos de um serviço procurado ao longo da rede. Recentemente, novos algoritmos de roteamento para redes *peer-to-peer* foram propostos em sistemas como Pastry [RD01] e Chord [SMK+01]. Estes algoritmos são mais eficientes do que o algoritmo de roteamento baseado em *broadcast* usado pelo Gnutella.

Outra iniciativa no campo de sistemas *peer-to-peer* que merece ser mencionada é a tecnologia JXTA [Gon01], proposta recentemente pela Sun. JXTA é um conjunto de protocolos cujo objetivo é tornar diferentes sistemas *peer-to-peer* para compartilhamento de arquivos interoperáveis. Por exemplo, permitir que um sistema como o Napster possa fazer uma consulta na rede criada pelo Gnutella e vice-versa. Pode-se, portanto, afirmar que JXTA está para sistemas *peer-to-peer* assim como CORBA está para sistemas baseados em objetos distribuídos. No entanto, é controverso se uma área incipiente, com diversas aplicações ainda para serem projetadas, deve ser objeto de padronização.

Conforme afirmado no Capítulo 3, acredita-se que os sistemas mais próximos de PeerSpaces são Jini [Arn00, Wal99] e Lime [PMR99, Mur00, MPR01].

Diferentemente de Jini, PeerSpaces não assume a existência de um serviço de nomes centralizado. Além disso, PeerSpaces adota um modelo de comunicação assíncrono, o que aumenta o grau de tolerância a desconexões e viabiliza comunicação entre processos que não se encontrem simultaneamente conectados à rede. Suponha por exemplo a Figura 4.2, utilizada anteriormente no Capítulo 4 para demonstrar as limitações de Jini. Em Jini, na situação (1) desta figura, interação entre os nodos a e b não é possível devido à inacessibilidade do servidor de nomes, a qual impede que o nodo a tome conhecimento da existência do serviço requisitado pelo mesmo no nodo vizinho b. Já em PeerSpaces, um **find** disparado pelo nodo a conseguiria localizar o serviço requisitado no nodo b, visto que estes dois nodos encontram-se conectados. Na situação (2), usando-se Jini, o nodo a consegue se comunicar com o servidor de nomes e localizar o serviço de que necessita no nodo b. No entanto, o nodo a não consegue se comunicar com o nodo b, visto que este tornou-se temporariamente inacessível. Em Jini, dada ao sincronismo típico do sistema, a interação entre a e b falha com a ativação de uma exceção. Já em PeerSpaces, uma mensagem enviada de a para b ficaria armazenada em a durante a desconexão temporária de b e seria enviada posteriormente quando voltasse a existir conexão entre estes dois nodos.

Diferentemente de Lime, a implementação dos conceitos e primitivas de PeerSpaces não requer nenhuma forma de sincronização distribuída. Em Lime, existem duas estruturas de dados: uma física, representada pelos espaços de tuplas locais de cada nodo, e outra virtual e global, representada pelos espaços de tuplas federados. O modelo permite que programadores acessem ambas estruturas transparentemente, por meio do mesmo conjunto de primitivas. Sem dúvidas, este modelo de coordenação é poderoso, pois permite a construção de aplicações com alto grau de transparência de localização. No entanto, como mostrado nos experimentos 1 e 2 da Seção 4.3.2 do Capítulo 4, os custos de tal transparência no desempenho e na escalabilidade das aplicações não podem ser neglicenciados.

Ao contrário de Lime, PeerSpaces parte do pressuposto de que banda de rede é um recurso escasso em redes móveis e, portanto, deve ser usado de forma o mais eficiente possível. Assim, incentiva-se um estilo de programação consciente da localização dos serviços remotos a serem manipulados[1]. Em outras palavras, aplicações em PeerSpaces não apresentam transparência de localização em nome de um uso mais eficiente do meio de comunicação sem fio. No experimento 1 da Seção 4.3.2, por exemplo, os agentes R consumidores teriam que descobrir previamente a localização de seus respectivos agentes produtores. Para isso, poderiam utilizar a primitiva **find**. Já no experimento 2, a única implementação possível em PeerSpaces é aquela que não se baseia em transparência de localização. Como mostrado neste experimento, supondo-se quatro agentes na rede, esta implementação realiza a soma em paralelo em quase metade do tempo exigido pela implementação com transparência de localização.

[1] Em inglês, *location-awareness*.

PeerWare [CP01] é um outro trabalho, ainda em andamento, que objetiva resolver as deficiências detectadas em Lime. Assim como Lime, este sistema se baseia no conceito de um estrutura de dados global e virtual (GVDS, ou *Global and Virtual Data Structure*). Cada nodo em PeerWare possui uma estrutura de dados organizada na forma de uma floresta de árvores, onde as folhas são documentos. Esta estrutura lembra a árvore de diretórios de um sistema de arquivos. A GVDS criada por PeerWare é a "superimposição" das florestas locais de cada nodo conectado à rede. O sistema disponibiliza ainda uma operação `execute(Fn,Fd,a)`, a qual executa uma ação arbitrária `a` na projeção da GVDS determinada pela função `Fn` (que filtra nodos) e `Fd` (que filtra documentos). A novidade de PeerWare é o reconhecimento de que atomicidade não é um requisito razoável em cenários móveis. Por esta razão, o sistema oferece uma variante da operação acima que não requer que a execução da ação `a` seja atômica. No entanto, a abstração de uma GVDS somente faz sentido caso sua consistência seja assegurada pelo modelo, o que requer atomicidade. Caso o modelo não assegure a consistência da GVDS, o mesmo se reduz a avaliação remota.

Lana [RBP01] é uma linguagem derivada de Java que está sendo projetada atualmente para construção de aplicações para computadores móveis. A fim de resolver as deficiências apresentadas por Java e Jini em ambientes de comunicação sem fio, Lana e sua máquina virtual suportam a construção de programas móveis que executam em domínios de proteção próprios e que se comunicam via chamada assíncrona de métodos. A linguagem, no entanto, não inclui um serviço descentralizado para localização de recursos, como o existente em PeerSpaces.

SLP (*Service Location Protocol*) [Gut99] é um protocolo para localização de serviços em redes fixas. O protocolo utiliza o termo agente para designar os nodos de uma rede. Na nomenclatura própria de SLP, agentes são classificados em agentes de usuário (UA), agentes de serviço (SA) e agentes diretório (DA). Em outras palavras, UAs correspondem a clientes e SAs a servidores. Apesar de ter sido proposto para redes fixas, SLP suporta localização de serviços mesmo quando não existem DAs na rede. Neste caso, UAs localizam SAs por meio de *multicast*, isto é, de forma semelhante àquela usada pela primitiva **find** de PeerSpaces. No entanto, este modo de operação do protocolo foi proposto para uso em redes locais. Neste tipo de rede, *multicast* é uma operação bem mais simples do que em redes *ad hoc*. Não existe, por exemplo, o risco de criação de ciclos na propagação de mensagens. Por fim, sendo um único protocolo, SLP não integra localização de serviços a um modelo de comunicação entre processos, como ocorre em PeerSpaces.

Actors [Agh86] é outro modelo que merece ser citado. *Actors* são objetos autônomos, que encapsulam um estado, um conjunto de métodos e uma *thread*. Cada *actor* possui uma caixa de mensagens, a qual possui um endereço único no sistema. *Actors* podem enviar mensagens assíncronas para a caixa de mensagens de outros *actors*. Portanto, em certo sentido, Actors e PeerSpaces são similares, visto que ambos modelos são baseados em mensagens assíncronas e não se valem de uma estrutura centralizada de comunicação. No entanto, não existe em Actors uma noção explícita de localização e de conectividade com outros *actors*. Basicamente, basta conhecer

o endereço da caixa de mensagens de um *actor* para se enviar com sucesso uma mensagem ao mesmo – independentemente de existir ou não conectividade com este *actor*. O conceito de espaço de tuplas de PeerSpaces é também mais genérico do que o de caixa de mensagens. O espaço de tuplas de um nodo pode ser usado para armazenar não apenas mensagens cujo destino é este nodo, mas também para armazenar descrições dos serviços disponibilizados no mesmo. Por último, Actors não oferece conceitos similares aos de grupos e de consultas contínuas.

Consultas contínuas (ou *continuous queries*) já foram propostas para sistemas de banco de dados tradicionais [TGNO92]. Um exemplo recente é o sistema NiagaraCQ [CDTW00], o qual têm como objetivo tornar o conceito escalável para grandes bases de dados na Internet.

5.7.3 Principais Contribuições

Neste capítulo, foi apresentado, formalizado e analisado o modelo PeerSpaces para coordenação de aplicações em redes sem fio. O projeto de PeerSpaces teve como objetivo solucionar a principal deficiência apresentada por modelos de coordenação baseados em espaços de tuplas quando utilizados em ambientes de redes móveis *ad hoc*: a estrita observância dos mesmos à arquitetura cliente/servidor tradicional, a qual pressupõe a existência de servidores bem conhecidos, centralizados e globalmente acessíveis. Por outro lado, PeerSpaces preserva o estilo de comunicação distribuída no tempo e no espaço que é característico de modelos de coordenação baseados em espaços de tuplas. Este estilo de comunicação é particularmente útil em sistemas notadamente assíncronos como redes sem fio.

Propõe-se em PeerSpaces uma arquitetura *peer-to-peer*, onde todo computador móvel possui seu próprio espaço de tuplas, o qual é usado para comunicação assíncrona entre processos executando neste dispositivo e também para armazenar descrições dos serviços disponibilizados pelo mesmo. No modelo proposto, nodos descobrem recursos remotos utilizando um serviço de localização descentralizado e cuja implementação não requer nenhuma forma de sincronização distribuída. Resumindo, o projeto do modelo privilegiou aderência aos princípios de redes *ad hoc*, escalabilidade e tolerância a falhas em detrimento de transparência. Basicamente, justifica-se esta opção pelo fato de que abstrações que oferecem um maior grau de transparência e expressividade, via de regra, também demandam um nível de qualidade de serviço não disponível em redes sem fio.

Capítulo 6

Aplicações

6.1 Introdução

Inicialmente, a Seção 6.2 deste capítulo descreve como os modelos de programação e de coordenação, propostos nos Capítulos 3 e 5, podem ser integrados em um modelo único. Este modelo permite a construção de aplicações tolerantes a desconexões e cujos padrões de comunicação são adequados a ambientes de computação móvel ou, mais especificamente, a redes *ad hoc*. Finalizando o capítulo, a Seção 6.3 apresenta como estudo de caso um sistema para revisão de artigos submetidos a uma conferência.

6.2 Integração dos Modelos de Programação e de Coordenação

Considera-se neste trabalho que computação e coordenação são conceitos ortogonais. Daí a proposição de modelos diferentes para ambos conceitos. Computação ocorre conforme definido pelo modelo de programação proposto no Capítulo 3, o qual é implementado no sistema Jamp. Já coordenação ocorre conforme definido no modelo PeerSpaces, descrito no capítulo 5. Argumenta-se que esta abordagem viabiliza o uso independente dos dois modelos. Assim, pode-se usar Jamp em conjunto com um outro sistema de comunicação, como Java RMI ou Jini. Esta alternativa é recomendada em aplicações para redes sem fio infra-estruturadas, que demandem reduzido montante de interação remota e onde operação em modo desconectado é um requisito mandatório. Ortogonalmente, pode-se usar PeerSpaces como infra-estrutura de comunicação de um modelo de programação baseado em objetos estáticos – e não móveis, como em Jamp. Esta segunda alternativa é adequada para aplicações intensivas em comunicação remota, incluindo comunicação em modo *ad hoc*, e que não têm operação em modo desconectado dentre seus requisitos.

Por outro lado, nada impede também que os dois sistemas sejam integrados em um modelo único, o qual será aqui chamado de JampSpaces. O uso deste modelo integrado é útil em aplicações que necessitem aliar tolerância a desconexões a padrões de comunicação adequados a

ambientes de computação móvel, incluindo tanto redes infra-estruturadas como redes *ad hoc*. A arquitetura básica de JampSpaces é descrita no restante desta seção.

Assim como em Jamp, JampSpaces considera que aplicações são organizadas em *containers*, os quais são grupos de objetos e classes que podem ser enviados pró-ativamente para outros nodos da rede. JampSpaces mantém ainda a estruturação dos nodos de uma rede em contextos de sistema e de usuário. Contextos de sistema são serviços disponíveis em nodos da rede fixa para recepção de *containers* provenientes de outros contextos e armazenamento temporário dos mesmos em memória secundária. Já contextos de usuários são serviços executados em nodos da rede sem fio. Um contexto de usuário é associado a um contexto de sistema, do qual recebe *containers* para execução. A fim de manter o sistema aderente ao modelo de coordenação proposto por PeerSpaces, JampSpaces define que todo contexto possuiu um espaço de tuplas local, o qual será acessado pelos *containers* em execução no mesmo e também por *containers* remotos. A Figura 6.1 mostra a arquitetura de *software* utilizada em JampSpaces.

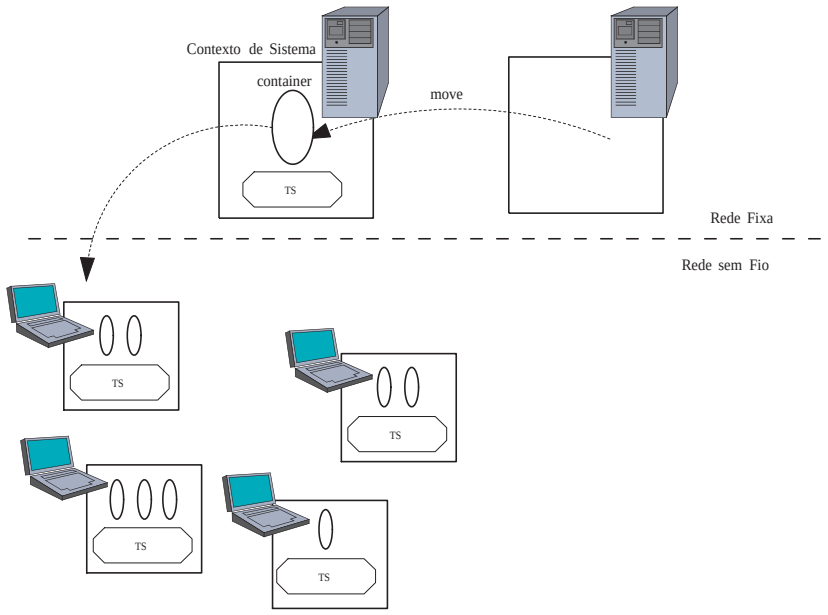


Figura 6.1: Arquitetura de *software* proposta em JampSpaces

Em JampSpaces, objetos contidos em *containers* podem se comunicar de duas formas: sincronamente, por meio de chamadas de métodos, ou assincronamente, por meio de espaços de tuplas. Conforme proposto em Jamp, comunicação síncrona é usada localmente, isto é, entre objetos de *containers* localizados em um mesmo contexto. Já comunicação assíncrona ocorre utilizando as primitivas definidas no modelo PeerSpaces: **out** , **in** , **rd** e **find**. As três primeiras primitivas permitem acesso ao espaço de tuplas local de um contexto, bem como a espaços de tuplas de contextos remotos. Já a primitiva **find** é utilizada para descobrir a localização de serviços em outros contextos da rede. A Figura 6.2 ilustra estas duas formas de comunicação disponíveis em JampSpaces.

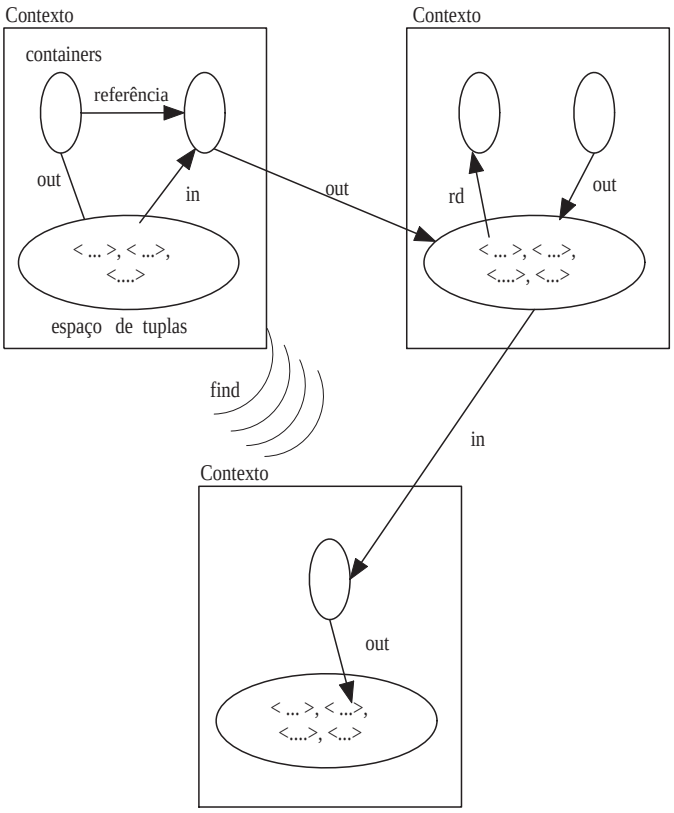


Figura 6.2: Comunicação síncrona e assíncrona em JampSpaces

JampSpaces confina interações síncronas a um único nodo devido à imprevisibilidade típica do meio de comunicação sem fio. Esta imprevisibilidade faz com que o tempo de transmissão de uma mensagem varie de alguns milisegundos a alguns minutos, dependendo das condições da rede e da ocorrência de desconexões temporárias. Com isso, é difícil estipular um limite superior para latência, o qual seria utilizado para sinalizar falhas de comunicação. A definição de um limite superior típico de redes locais, na ordem de milisegundos ou segundos, pode dar origem a sistemas pouco tolerantes a falhas. Assim, usuários podem ser forçados a submeter diversas vezes uma mesma operação remota, como, por exemplo, a compra de um lote de ações em um sistema financeiro. Por outro lado, caso assuma-se o pior caso, e seja definido um limite superior da ordem de segundos ou mesmo minutos, pode-se degradar consideravelmente o tempo de resposta das aplicações. No caso de um sistema financeiro, por exemplo, usuários teriam que aguardar intervalos de tempos demasiadamente longos para realizar uma nova transação, como a compra de um segundo lote de ações.

Pelas razões mencionadas acima, comunicação remota é assíncrona em JampSpaces. No caso do sistema financeiro mencionado anteriormente, a compra de um lote de ações seria implementada por meio de uma operação **out** no espaço de tuplas de um contexto pertencente a uma bolsa de valores ou a uma corretora. Esta operação depositaria neste espaço uma tupla

descrevendo a transação solicitada, isto é, o nome e o número de ações a ser comprado, o valor total da transação, a identificação do comprador etc. Conforme especificado em PeerSpaces, uma operação **out** remota deposita inicialmente sua tupla no próprio espaço do nodo que solicitou a operação. Em um segundo passo, esta tupla é assincronamente transmitida para o seu contexto de destino. Assim, o sistema financeiro do exemplo fica imediatamente liberado para realizar uma nova transação.

Comunicação local em JampSpaces pode ser síncrona ou assíncrona. Comunicação síncrona ocorre por meio de chamadas de métodos e comunicação assíncrona por meio do espaço de tuplas local. Basicamente, comunicação síncrona permite que o emissor tome conhecimento imediato do envio de uma mensagem e que o receptor seja logo notificado da chegada da mesma. Esta forma de comunicação é mandatória em alguns casos, como em sistemas de tempo real. Por outro lado, comunicação local assíncrona pode ser útil quando emissor e receptor não estão em execução ao mesmo tempo. Por exemplo, sempre que for executada, uma aplicação bancária pode depositar no espaço local de um contexto os valores de alguns indicadores financeiros, como a cotação do dólar. Estes indicadores poderão ser utilizados posteriormente em outras aplicações. Portanto, comunicação assíncrona por meio de espaços de tuplas permite a um emissor produzir uma informação no instante de tempo t_1 que somente será consumida por um receptor num instante t_2, onde $t_2 > t_1$.

6.3 Estudo de Caso: Sistema para Revisão de Artigos

Como estudo de caso, descreve-se nesta seção o projeto e a implementação de um sistema para revisão de artigos submetidos a uma determinada conferência [Car99]. Basicamente, o objetivo deste sistema é automatizar o processo de avaliação e seleção dos artigos a serem apresentados nesta conferência. Assim, os seguintes requisitos são importantes neste sistema: operação em modo desconectado e distribuição pró-ativa de formulários de avaliação de artigos. O requisito de operação em modo desconectado possibilita aos membros do comitê de programa realizar avaliações de artigos em seus *laptops*, mesmo quando estes não estiverem conectados a uma rede fixa ou sem fio. Já a distribuição pró-ativa de formulários de avaliação de artigos evita que os membros do comitê de programa tenham que solicitar explicitamente os formulários dos artigos que deverão avaliar.

Mostra-se a seguir como funcionalidades importantes deste sistema são implementadas em JampSpaces. São descritas a implementação das seguintes tarefas: distribuição de formulários de avaliação, impressão de artigos, envio dos formulários de avaliação e seleção dos artigos a serem apresentados.

Distribuição dos Formulários de Avaliação: Como operação em modo desconectado e distribuição pró-ativa de formulários são requisito mandatórios no sistema, considera-se que aplicações para avaliação de artigos são representadas por *containers*. Assim, utilizando-se as

classes descritas na Seção 3.4, um *container* deve ser criado para cada formulário de avaliação, conforme mostrado abaixo:

```
 1: JContainer container= new JContainer ("SBLP 2001 - Paper 00");
 2: Paper p= (Paper) JContainer.newObject("PaperImpl");
 3: ReviewForm r= (ReviewForm) JContainer.newObject("ReviewFormImpl");
 4: ......  // inicializacao de p e r
 5: container.addObj(p);
 6: container.addObj(r);
 7: container.addClass("PaperImpl");
 8: container.addClass("ReviewFormImpl");
 9: container.addClass("ReviewFrameImpl");
10: container.move("valente@jampspaces.dcc.ufmg.br:5000", r);
```

Inicialmente, na linha 1, cria-se um *container* que delimitará a aplicação móvel a ser enviada para um determinado revisor de artigos. Em seguida, nas linhas 2 e 3, criam-se dois objetos móveis, que representarão, respectivamente, um artigo e seu formulário de avaliação. Observe-se que sendo objetos móveis, os mesmos são criados por meio do método `newObject` e não via um construtor normal de Java. Por isso mesmo, na linha 4, deixa-se indicado que estes objetos devem ser inicializados após sua criação. O objeto `p`, por exemplo, deve ser inicializado com o nome do artigo, seus autores, seu texto etc. Nas linhas de 5 a 8, os objetos criados anteriormente são inseridos no *container*, bem com o código de suas respectivas classes. Na linha 9, insere-se ainda no *container* o código da classe `ReviewFrameImpl`, a qual representa um objeto visual utilizado para se entrar com os dados da avaliação de um artigo.

Na linha 10, o *container* é enviado para o revisor `valente@jampspaces.dcc.ufmg.br`, por meio do método `move`. Considera-se, portanto, que existe um contexto de sistema em execução na porta TCP/IP de número 5000 do servidor de nome `jampspaces.dcc.ufmg.br`. Este contexto de sistema deve ainda possuir um usuário de nome `valente`. Veja que o nodo `jampspaces.dcc.ufmg.br` deve estar localizado na Internet tradicional, isto é, não móvel, de forma a garantir que o mesmo possa ser facilmente acessado a partir do domínio de origem do *container*. O método `move` ainda especifica que a execução do *container* terá início pelo método `start` do objeto `r`.

O *container* permanece armazenado no contexto de sistema `jampspaces.dcc.ufmg.br` até que o contexto do usuário `valente` se conecte à rede e solicite a transferência do mesmo. A execução propriamente dita do *container* no contexto de usuário somente ocorre por solicitação do usuário do mesmo. Conforme especificado no método `move`, esta execução inicia-se pelo método `start` do objeto que representa o formulário de avaliação do artigo (objeto `r`). Este método cria um objeto da classe `ReviewFrameImpl`, o qual exibe um formulário para se entrar com os dados da avaliação do artigo associado ao *container* cuja execução foi iniciada. A Figura 6.3 mostra a interface de um contexto de usuário no momento em que este formulário encontra-se aberto.

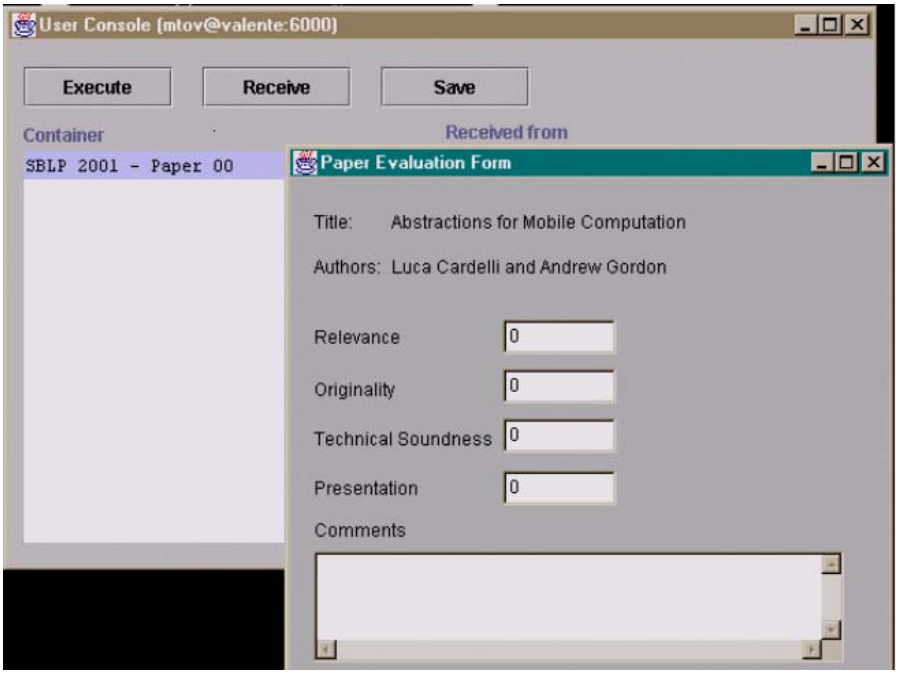


Figura 6.3: Interface de um contexto de usuário

Veja que para iniciar a execução de um *container* na interface mostrada na Figura 6.3 deve-se selecionar o mesmo e, em seguida, clicar no botão `Execute`. Esta interface permite ainda solicitar a transferência de novos *containers* que porventura tenham sido recebidos pelo contexto de sistema associado a este contexto de usuário (botão `Receive`) e salvar o estado corrente dos *containers* localizados no mesmo (botão `Save`).

A aplicação delimitada pelo *container* descrito anteriormente atende ao requisito de operação em modo desconectado, já que a mesma contém todos os dados e todo o código necessários à sua execução. Dentre os dados enviados junto com o *container*, encontra-se, por exemplo, o texto do artigo a ser avaliado. Já dentre o código, encontra-se a classe `ReviewFrameImpl`, a qual foi incluída no *container* para viabilizar a criação de um objeto da mesma quando da execução do *container* no contexto de usuário. Veja que uma aplicação tradicional em Java tentaria carregar esta classe dinamicamente de algum repositório de código no momento da criação de um objeto da mesma. Logo, esta aplicação não seria capaz de operar em modo desconectado, já que a sua execução pressupõe a existência de conectividade com o referido repositório.

Localização de um Serviço de Impressão: A maioria dos seres humanos prefere ler textos em papel do que em um monitor de vídeo, ainda mais em se tratando de monitores de *laptops*. Assim, nada mais razoável do que o desejo de um revisor em imprimir o artigo que lhe caberá revisar. No entanto, ao solicitar esta impressão, o revisor pode se encontrar usando seu *laptop* em sua sala na universidade, ou em sua residência ou ainda em visita a um departamento de

uma outra universidade. Supondo que em todos estes ambientes exista uma rede, possivelmente sem fio, a aplicação para revisão de artigos pode localizar uma impressora por meio do seguinte trecho de código:

```
1: Tuple template= new Tuple();
2: template.addActual("printer");
3: template.addFormal(Integer.class);
4: template.AddFormal(String.class);
5: ts.find(template);
```

Nas linhas de 1 a 4, cria-se um padrão para uma tupla cujos campos são os seguintes: a *string* `printer`, um inteiro qualquer (representando o código da impressora) e uma *string* qualquer (representando o nodo da rede onde a impressora está conectada). As rotinas `AddActual` e `AddFormal` pertencem ao pacote LighTS, usado na implementação de JampSpaces. Em seguida, na linha 5, executa-se um **find** para procurar por um tupla compatível com este padrão nos nodos da rede corrente do revisor.

Conforme definido em PeerSpaces, as respostas deste **find** são depositadas no espaço de tuplas local do nodo que solicitou a operação. Assim, a aplicação móvel para revisão de artigos pode recuperar uma destas respostas da seguinte forma:

```
6: Tuple printer= ts.in(template);
7: Integer printer_id= (Integer) printer.get(1).getValue();
8: String printer_loc= (String) printer.get(2).getValue();
```

Na linha 6, obtém-se uma das respostas do **find**, usando para isso um **in**. O padrão utilizado nesta operação é idêntico àquele empregado anteriormente quando da execução do **find** (linha 5). Em seguida, nas linhas 7 e 8, extraem-se o segundo e terceiro campos da tupla retornada pela operação **in**. Estes campos representam, respectivamente, o identificador e a localização da impressora procurada.

Ressalte-se que operação **in** é sempre síncrona em PeerSpaces, já que sua continuação depende da tupla a ser lida. No exemplo, isto implica que a aplicação fica bloqueada aguardando a localização de uma impressora. Caso não exista impressora na rede atual do revisor, isto implicará em um bloqueio por tempo indeterminado. Em uma implementação mais real do sistema, esta situação pode ser contornada usando uma versão sem bloqueio da primitiva **in**, usualmente chamada de **inp** (*probe in*). Esta primitiva consulta o espaço e, caso não seja encontrada uma tupla compatível, a mesma retorna `null`, em vez de permanecer bloqueada. Para tratar este tipo de situação, a primitiva **inp** foi incorporada à implementação de PeerSpaces. No entanto, como usual em definições formais de Linda, esta primitiva não faz parte da semântica mostrada no Capítulo 5, já que pode ser simulada por meio de um processo em paralelo que "mata" o processo bloqueado pelo **in** após um certo *time-out* [Car99, BOV99].

Impressão de um Artigo: De posse do nome e do nodo da impressora procurada, o seguinte trecho de código envia o artigo para a mesma:

```
 9: Tuple job= new Tuple();
10: job.addActual(printer_id);
11: job.addActual(file);
12: ts.out(printer_loc, job);
```

Nas linhas de 9 a 11, cria-se uma tupla com dois campos: o identificador da impressora e o arquivo a ser impresso. Esta tupla é então enviada para o espaço de tuplas do nodo onde a impressora está conectada (linha 12). Assume-se que neste nodo existe uma *thread* que continuamente remove tuplas no formato acima e imprime o segundo campo das mesmas na impressora informada no primeiro campo.

Envio do Formulário de Avaliação de um Artigo: Após revisar um artigo, o membro do comitê de programa da conferência deve preencher seu formulário de avaliação e enviar o mesmo para o *site* da conferência. Esta operação é realizada pelo seguinte código:

```
1: Tuple t= new Tuple();
2: t.addActual("sblp2001");
3: t.addActual(reviewer_id);
4: t.addActual(paper_id);
5: t.addActual(revelance);
6: t.addActual(originality);
7: t.addActual(soundness);
8: t.addActual(presentation);
9: ts.out("jampspaces.sblp.org.br", t);
```

Nas linhas de 1 a 8, cria-se uma tupla com sete campos: o nome da conferência, o identificador do revisor, o código do artigo que foi avaliado e as notas conferidas ao mesmo. Estas notas correspondem aos critérios de relevância, originalidade, consistência e apresentação, usados no julgamento de artigos. Na linha 9, a tupla `t` criada para armazenar o resultado da avaliação é enviada para o contexto da conferência, o qual possui nome `jampspaces.sblp.org.br`.

Observe que como a operação de saída em espaços de tuplas remotos é assíncrona em JampSpaces, a aplicação em questão fica imediatamente liberada para realizar novas tarefas após executar o **out** da linha 9. Esta característica facilita não só a utilização da aplicação, mas também a sua programação. O programador do sistema em questão não precisa, por exemplo, verificar se o contexto do revisor está conectado ou não à rede no momento em que a operação **out** foi executada.

Seleção dos Artigos a serem Apresentados na Conferência: Normalmente, esta seleção ocorre durante uma reunião do comitê de programa. Imaginando que os membros deste comitê

possuam *laptops* com interfaces de comunicação *ad hoc*, uma rede pode ser criada na sala de reuniões de forma espontânea, isto é, sem que seja necessário instalações físicas ou configurações especiais por parte de administradores de redes. Os participantes da reunião podem usar esta rede para trocar pareceres sobre artigos, para resolver conflitos de avaliação, para descobrir recursos, como uma máquina de fax ou um sistema de arquivos armazenando todos os artigos etc. Participantes podem ainda sair temporariamente da sala para dar um parecer extra sobre um artigo.

No cenário descrito acima, o emprego de um espaço de tuplas centralizado e globalmente acessível não é apropriado. Em vez disso, deve-se assumir que todos os nodos da rede possuem as mesmas capacidades e responsabilidades. Assim, a infra-estrutura de comunicação descentralizada e *peer-to-peer* proposta em JampSpaces pode ser usada na implementação de um sistema de *workgroup* que permita o tipo de interação entre os participantes de uma reunião como a descrita acima. Como este sistema possui um maior número de funcionalidades, a implementação do mesmo não será discutida neste trabalho.

6.3.1 Resultados Experimentais

Analisam-se nesta seção alguns resultados experimentais obtidos a partir da implementação do estudo de caso mencionado anteriormente. Esta implementação foi realizada usando uma versão de JampSpaces obtida integrando-se os protótipos de Jamp e PeerSpaces descritos, respectivamente, nos Capítulos 3 e 5. Os resultados apresentados a seguir foram obtidos em uma rede Ethernet formada por estações de trabalho Pentium 4, com clock de 1.7 Ghz, 128 Mb de memória principal e sistema operacional Windows NT Workstation 4.0. O estudo de caso foi interpretado usando-se a máquina virtual do ambiente JDK 1.4 da Sun. Todas as medidas de tempo foram obtidas calculando-se a média aritmética dos tempos de execução de diversas iterações do experimento descrito.

Descrição do Experimento O experimento realizado teve como objetivo quantificar o *overhead* inerente ao manuseio de objetos móveis. Conforme descrito no Capítulo 3, objetos móveis são aqueles que podem ser inseridos em *containers* e que, portanto, podem migrar de um contexto para outro da rede. Em JampSpaces, todo objeto móvel possui um mediador, o qual implementa as mesmas interfaces de seu objeto mediado. Objetos mediadores são usados para viabilizar a implementação dos conceitos de referências conectadas e desconectadas.

Em JampSpaces, o método `newObject` é usado para criar objetos móveis. Este método, além de criar um objeto, também cria seu respectivo mediador. Assim, mediu-se o tempo necessário para criar um objeto móvel da classe `PaperImpl`, a qual representa artigos a serem avaliados usando-se o Sistema de Revisão de Artigos. Este tempo foi comparado com a criação de um objeto não móvel da mesma classe, isto é, um objeto criado pelo operador `new` tradicional de Java. Os resultados obtidos são mostrados na Tabela 6.1.

Criação de objeto da classe `PaperImpl`	Tempo (ms)	
	Primeiro objeto	Demais objetos
Via método `newObject`	310	10
Via operador `new`	5	5

Tabela 6.1: Tempo de criação de objetos móveis e não móveis

Observa-se nesta tabela que o tempo para se criar o primeiro objeto móvel de uma classe é consideravelmente maior do que o tempo consumido para se criar um objeto não móvel desta mesma classe. Esta diferença deve-se ao custo de criação de mediadores, os quais são gerados em JampSpaces usando-se o recurso de classes *proxy* dinâmicas de Java. Assim, um mediador é um objeto de uma classe criada dinamicamente por meio dos recursos de reflexividade de Java. Logo, como a criação de objetos móveis inclui a criação de seus respectivos mediadores, não é surpresa a grande diferença de tempo observada na primeira coluna da Tabela 6.1. No entanto, uma vez criado o primeiro mediador de um objeto de uma determinada classe, armazena-se em um *cache* a classe gerada dinamicamente para descrever este mediador. Assim, novos mediadores para objetos desta mesma classe reutilizam a classe armazenada anteriormente. Logo, como pode ser conferido na última coluna da referida tabela, o tempo de criação dos demais objetos móveis da classe `PaperImpl` é bastante inferior ao tempo de criação de um primeiro objeto móvel desta mesma classe.

Mediu-se ainda o tempo para se serializar e desserializar o *container* enviado para o revisor de um artigo no estudo de caso realizado. Conforme descrito anteriormente, este *container* possui dois objetos: o artigo a ser avaliado e seu formulário de avaliação. O *container* possui ainda as classes destes dois objetos e mais uma terceira classe, usada para exibir a interface do sistema em um contexto de usuário. Após serializado, este *container* ocupa cerca de 74 Kb. Deste total, 64 Kb foram alocados para simular o texto do artigo. Além disso, este *container* possui uma referência para um objeto de um outro *container*, a qual é desconectada quando o mesmo é enviado para o contexto de seu revisor.

A serialização do *container* descrito demandou cerca de 160 ms e sua desserialização consumiu aproximadamente 210 ms. Como era de se esperar, o tempo de desserialização é maior, visto que inclui o esforço necessário para conectar referências do contexto de destino que denotam objetos do *container* que acabou de ser recebido.

Análise Crítica: Inicialmente, o experimento mostrou que o custo de criação do primeiro objeto móvel de uma classe é significativo. No entanto, a criação dos demais objetos desta classe demanda bem menos tempo. Assim, em programas que manipulam diversos objetos, o custo médio de criação de objetos móveis se assemelha ao de objetos convencionais. Já o custo de serialização de objetos existe em qualquer sistema que tenha operação em modo desconectado como um de seus objetivos. Em JampSpaces, esta serialização baseia-se nos métodos tradicionais de Java. Em termos de desempenho, a desvantagem do sistema é o fato de a desserialização de

containers incluir a conexão de possíveis referências desconectadas. Por outro lado, a principal vantagem de JampSpaces é a limitação dos objetos serializados àqueles previamente agrupados em um *container* – ao passo que as rotinas tradicionais de Java serializam um objeto e todos os demais alcançáveis a partir dele.

Capítulo 7

Conclusões

O desenvolvimento do presente trabalho foi norteado pela constatação de que redes sem fio constituem um ambiente de programação distribuída com diferenças relevantes em relação àquele originado a partir de redes tradicionais. Eventos como flutuações na largura de banda, desconexões e mobilidade física dos dispositivos computacionais são, por exemplo, observáveis apenas em ambientes de comunicação sem fio. Assim, dada à relevância dos mesmos, considera-se que os modelos e abstrações para construção de sistemas distribuídos nestas redes devem ser capazes de lidar com estes eventos e, quando possível, atenuar seus efeitos sobre as aplicações.

Particularmente, nesta Tese de Doutorado foram propostos um modelo para programação de aplicações distribuídas para dispositivos computacionais móveis e um modelo para coordenação destas aplicações. A abordagem adotada no projeto de cada um destes modelos é resumida a seguir.

Modelo de Programação: O objetivo central do modelo de programação que foi descrito no Capítulo 3 é permitir a construção de aplicações distribuídas tolerantes a desconexões. Basicamente, este modelo utiliza mobilidade lógica – ou, mais especificamente, mobilidade de objetos – para tratar desconexões, isto é, para lidar com um problema originado pela mobilidade física de dispositivos computacionais em uma rede sem fio.

O modelo de programação proposto inclui duas abstrações principais: *containers* e contextos. Um *container* é um grupo de objetos e classes que pode ser enviado pró-ativamente para execução em computadores móveis. Como possuem todo código e os dados necessários à sua execução, aplicações delimitadas em *containers* são capazes de operar em modo desconectado. Já contextos são serviços disponibilizados em certos nodos da rede para recepção e execução de *containers*. Contextos são organizados no modelo proposto em uma hierarquia de dois níveis, sendo o primeiro nível localizado em nodos da rede fixa e o segundo nível em nodos da rede móvel. Desta maneira, aplicações são temporariamente armazenadas em nodos da rede fixa antes de serem transmitidas para dispositivos computacionais móveis.

Mostrou-se também no trabalho a semântica formal do modelo de programação proposto.

Para isto, definiu-se uma linguagem baseada em objetos bastante simples, na qual foram introduzidos os conceitos de *containers* e contextos. Em seguida, descreveu-se a semântica desta linguagem por meio de sua tradução para o Cálculo de Ambientes. Descreveu-se ainda no trabalho a implementação do modelo de programação proposto em uma linguagem orientada por objetos de uso geral. Nesta descrição, foi apresentado o sistema Jamp, o qual é um pacote que permite a implementação em Java de aplicações utilizando as abstrações propostas neste trabalho.

Modelo de Coordenação: O objetivo central do modelo de coordenação descrito no Capítulo 5 é disponibilizar uma infra-estrutura que, considerando as características inerentes do meio de comunicação sem fio, suporte a realização de tarefas como comunicação entre processos, sincronização e localização de serviços. O modelo de coordenação proposto, chamado PeerSpaces, é baseado no conceito de espaços de tuplas, proposto originalmente em Linda. No entanto, PeerSpaces substitui o modelo cliente/servidor, tradicionalmente usado em implementações de Linda, por um modelo *peer-to-peer*, o qual é mais adequado para coordenação de sistemas em redes móveis *ad hoc*.

Em PeerSpaces, todo computador móvel possui seu próprio espaço de tuplas, o qual é usado para comunicação entre processos executando neste dispositivo e para publicar descrições dos serviços disponibilizados pelo mesmo. O espaço de tuplas de um nodo pode ser ainda acessado a partir de outros nodos, permitindo que processos remotos possam se comunicar. Assim, nodos no sistema desempenham tanto papéis de clientes como de servidores de espaços de tuplas, não existindo diferenças fundamentais entre os mesmos. Daí se afirmar que o sistema promove uma arquitetura de *software* no estilo *peer-to-peer*. Além disso, computadores móveis descobrem recursos disponíveis em outros nodos utilizando um serviço descentralizado de localização, cuja implementação não requer nenhuma forma de sincronização distribuída.

Mostrou-se também no trabalho a semântica formal do modelo de coordenação proposto em PeerSpaces. Para isso, utilizou-se uma linguagem concorrente bastante simples, baseada no Cálculo de Ambientes, na qual foram introduzidas as primitivas de coordenação definidas no modelo. A semântica desta linguagem foi definida por meio de reduções e de uma relação de congruência estrutural entre processos. Descreveu-se ainda no trabalho o protótipo de um sistema que implementa o modelo de coordenação proposto em PeerSpaces. Neste protótipo, uma rede *ad hoc* é simulada por um conjunto de objetos distribuídos em uma rede fixa.

Os modelos de programação e coordenação descritos acima são ortogonais. No entanto, nada impede também que os mesmos possam ser usados de forma integrada. Assim, finalizando o trabalho, mostrou-se no Capítulo 6 como estes modelos podem ser incorporados em um sistema de computação único, o qual foi denominado de JampSpaces. Argumentou-se que a utilização de um sistema integrado como este é útil quando necessita-se aliar capacidade de operação em modo desconectado – como proposto no sistema Jamp – a padrões de comunicação adequados

a ambientes de comunicação sem fio – como proposto em PeerSpaces. Como estudo de caso, foi descrito o projeto e a implementação de um sistema para revisão de artigos submetidos a uma conferência.

7.1 Contribuições

As principais contribuições deste trabalho são as seguintes:

- Foi proposto o emprego de mobilidade de objetos para tratar desconexões voluntárias e involuntárias em redes sem fio. A utilização de mobilidade lógica como uma ferramenta para se projetar sistemas tolerantes a desconexões diferencia o modelo de programação descrito neste trabalho de outros modelos baseados em objetos móveis, como aqueles existentes em sistemas de agentes móveis. Nestes sistemas, mobilidade lógica é utilizada para construir aplicações autônomas capazes de migrar por diversos nodos da Internet para realizar uma determinada tarefa.

  A fim de viabilizar a construção de aplicações móveis tolerantes a desconexões, foram propostas as abstrações denominadas de *containers* e contextos. *Containers* permitem que se envie pró-ativamente para um computador móvel uma aplicação capaz de ser executada independentemente de a rede estar disponível ou não. Já contextos são classificados em contextos de sistema e de usuário. Os primeiros são localizados na rede fixa e os últimos em nodos da rede móvel. Esta hierarquização de contextos evita que a distribuição de aplicações para dispositivos computacionais dê origem a uma falha de comunicação quando estes não se encontrarem conectados à rede. Os conceitos de *containers* e contextos foram formalizados e, em seguida, implementadas por meio de um conjunto de classes em Java. Esta implementação não exigiu modificações na definição ou na máquina virtual desta linguagem.

- Argumentou-se que sistemas de objetos distribuídos, como Java RMI e Jini, não são totalmente adequados a ambientes sem fio devido ao fato de serem baseados em comunicação síncrona e em serviços de nomes centralizados. Por outro lado, argumentou-se que o conceito de espaços de tuplas, proposto originalmente em Linda, dá origem a um estilo de comunicação assíncrono que é particularmente interessante em aplicações distribuídas para redes sem fio. No entanto, sistemas baseados em Linda assumem a existência de um repositório de tuplas centralizado, o que inviabiliza a utilização dos mesmos em redes móveis *ad hoc*. Argumentou-se ainda que a idéia de espaços de tuplas federados, proposta originalmente em Lime, apesar de expressiva, possui um custo de implementação considerável, o qual impacta negativamente no desempenho e escalabilidade das aplicações.

- Foi proposto um modelo de coordenação baseado em espaço de tuplas e em uma arquitetura de *software* no estilo *peer-to-peer*. O modelo proposto, chamado de PeerSpaces, difere-se

de modelos tradicionais de coordenação baseados em espaços de tuplas por não assumir a existência de servidores centralizados, previamente conhecidos e globalmente acessíveis. PeerSpaces também não utiliza nenhuma estrutura de dados distribuída ao longo dos nodos de uma rede móvel. Em vez disso, o modelo oferece primitivas para acesso apenas a espaços de tuplas de nodos cujos nomes são bem conhecidos. Portanto, abre-se mão de transparência de localização no modelo em nome de um uso mais eficiente do meio de comunicação sem fio. PeerSpaces propõe também que os nodos de uma rede sejam organizados em uma hierarquia de grupos.

Definiu-se ainda em PeerSpaces uma primitiva nova, chamada **find**, a qual não existe em sistemas de coordenação inspirados em Linda. Esta primitiva é utilizada para se localizar serviços de forma descentralizada, independente de uma infra-estrutura física de comunicação e sem exigir nenhuma forma de sincronização distribuída. Além disso, a possibilidade de se definir um tempo de vida para as consultas originadas por um **find** introduz reatividade no modelo de coordenação proposto, permitindo que aplicações possam detectar e reagir a mudanças no estado de espaços de tuplas remotos.

- Mostrou-se que os modelos de programação e de coordenação propostos no trabalho podem ser integrados em um modelo único, o qual permite a construção de aplicações que aliam capacidade de operação em modo desconectado a um modelo de comunicação consistente com os princípios de funcionamento de redes móveis *ad hoc*.

7.2 Trabalhos Futuros

Durante o desenvolvimento desta Tese de Doutorado foram identificadas as seguintes questões que podem ser tratadas em trabalhos futuros:

- Segurança: Conforme afirmado no Capítulo 1, decidiu-se que mecanismos de segurança não estariam incluídos no escopo desta Tese de Doutorado. Assim, resta um trabalho a ser feito para introduzir tais mecanismos tanto no modelo de programação como no modelo de coordenação propostos. Acredita-se que o sistema Object Spaces [BR00] constitui um ponto de partida interessante para qualquer trabalho que vise integrar um modelo de segurança ao modelo de programação proposto no Capítulo 3. Já a introdução de segurança em PeerSpaces pode se basear, por exemplo, nos sistemas Secure Spaces [BOV99] e KLAIM [NFP98].

- Implementação em Computadores Móveis: Os sistemas Jamp e PeerSpaces, descritos nos Capítulos 3 e 5, foram implementados usando a versão tradicional de Java, a qual foi projetada para computadores não móveis e com maior capacidade de processamento. Assim, um trabalho interessante consiste em portar estas implementações para o ambiente J2ME,

isto é, para a versão simplificada e customizada de Java para uso em dispositivos computacionais móveis. Com isso, será possível desenvolver aplicações para computadores móveis que permitam validar com maior precisão os modelos de programação e de coordenação propostos neste trabalho.

- Integração de PeerSpaces e CoreLime: CoreLime [CVV01b, CVV01a] é uma versão simplificada de Lime projetada para coordenação de agentes móveis. Em nome de desempenho e escalabilidade, CoreLime restringe a noção de espaços transientemente compartilhados de Lime aos espaços de tuplas dos agentes móveis em execução em um mesmo nodo da rede. Assim, CoreLime não trata questões relacionadas com a mobilidade física de computadores móveis. O modelo não possui, por exemplo, uma primitiva para localização de serviços no conjunto de dispositivos conectados a uma rede sem fio. Daí a idéia de se integrar futuramente o modelo de coordenação de PeerSpaces e de CoreLime.

- Reconfiguração Dinâmica de Aplicações baseadas em *Containers*: Reconfiguração dinâmica designa a capacidade de estender, evoluir e modificar sistemas distribuídos sem que seja necessário intervenção manual por parte dos usuários dos mesmos [Hof93]. Trata-se, portanto, de um requisito importante em aplicações móveis construídas usando-se o conceito de *containers* proposto neste trabalho. Assim, acredita-se que existe um trabalho futuro a ser realizado no sentido de se incorporar mecanismos de reconfiguração dinâmica ao modelo de programação proposto no Capítulo 3 desta tese. Em linhas gerais, estes mecanismos devem permitir que se envie uma nova versão de um *container* para um determinado contexto a fim de substituir uma versão antiga do mesmo. A transferência de estado do *container* antigo para o novo deverá ser realizada automaticamente e de forma consistente.

- Descrição de Serviços em XML: No modelo de coordenação proposto, serviços são descritos por meio de tuplas. A fim de permitir representações mais complexas e padronizadas de serviços, uma possibilidade seria substituir tuplas por documentos em XML (*Extensible Markup Language*) [W3C98]. XML é uma linguagem para representação de dados semi-estruturados e tipados, que vêm se tornando padrão para pesquisa e troca de documentos na *Web*. Esta alternativa já é inclusive adotada no sistema JXTA [Gon01] e em XMLSpaces [TG01], uma extensão de Linda com suporte a documentos em XML.

Bibliografia

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[Ado85] Adobe Systems. *Postscript Language Reference Manual*. Addison Wesley, 1985.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, 3rd edition, 2000.

[Arn00] Ken Arnold. *The Jini Specifications*. Addison-Wesley, 2nd edition, 2000.

[BGZ98] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, February 1998.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BOV99] C. Bryce, M. Oriol, and J. Vitek. A Coordination Model for Agents Based on Secure Spaces. In P. Ciancarini and A. Wolf, editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594, pages 4–20. Springer-Verlag, Berlin, 1999.

[BR00] Ciarán Bryce and Chrislain Razafimahefa. An approach to safe object sharing. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2000.

[Bra97] Jeffrey Bradshaw. An introduction to software agents. In Jeffrey Bradshaw, editor, *Software Agents*, pages 3–46. AAAI Press/MIT Press, 1997.

[BST89] Henri E. Ball, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[Car94] Luca Cardelli. Obliq: A language with distributed scope. Technical Report 122, DEC Systems Research Center, June 1994.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[Car99] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer-Verlag, 1999.

[CDK01] George Couloris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design*. Addison-Wesley, 3rd edition, 2001.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD Conference on Management of Data*, pages 379–390, 2000.

[CEM01] Licia Capra, Wolfang Emmerich, and Cecilia Mascolo. Middleware for mobile computing. Submitted to publication, 2001.

[CG98] Luca Cardelli and Andrew Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[CG99] Luca Cardelli and Andrew Gordon. Types for mobile ambients. In *26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 79–92. ACM Press, 1999.

[CG01] N. Carriero and D. Gelernter. A computational model of everything. *Communications of the ACM*, 44(11):77–81, November 2001.

[Cia96] Paolo Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 6(28):300–302, June 1996.

[Cia01] Paolo Ciancarini. Coordination models and languages. Lecture Notes, 13th International School for Computer Science Researchers, July 2001.

[CL99] Daniel Câmara and Antônio Alfredo Ferreira Loureiro. Redes de computação móvel ad hoc. Jornada de Atualização em Informática, XV Congresso da SBC, July 1999.

[CP01] Gianpaolo Cugola and Gian Pietro Picco. PeerWare: Core middleware support for peer-to-peer and mobile systems, 2001. Submitted for publication.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[CSWH00] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, International Computer Science Institute, 2000.

[CVV01a] Bogdan Carbunar, Marco Tulio Valente, and Jan Vitek. Corelime a coordination model for mobile agents. In *International Workshop on Concurrency and Coordination*, volume 54 of *Electronic Notes on Theoretical Computer Science*. Elsevier Science, July 2001.

[CVV01b] Bogdan Carbunar, Marco Tulio Valente, and Jan Vitek. Lime revisited. In *5th IEEE International Conference on Mobile Agents*, volume 2240 of *Lecture Notes in Computer Science*, pages 54–69. Springer-Verlag, December 2001.

[DP96] R. DeNicola and R. Pugliese. A process algebra based on Linda. In P. Ciancarini and C. Hankin, editors, *1st International Conference on Coordination Models and Languages*, volume 1061 of *Lecture Notes in Computer Science*, pages 160–178. Springer-Verlag, 1996.

[FHA99] E. Freeman, S. Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[FK97] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, 1997.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[FZ94] George H. Forman and John Zahorjan. The challenges of mobile computing. *IEEE Computer*, 27(6):38–47, April 1994.

[GC92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[GCKR98] Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. D'Agents: Security in a multiple-language, mobile-agent system. In Giovanni Vigna, editor, *Mobile Agents and Security*, Lecture Notes in Computer Science, pages 154–187. Springer-Verlag, 1998.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[Gen95] General Magic. Telescript language reference, 1995.

[Gio02] Silva Giordano. *Mobile Ad-Hoc Networks*, chapter of Handbook of Wireless Networks and Mobile Computing. John Wiley & Sons, 2002.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java language specification*. Addison-Wesley, 1996.

[Gnu] Gnutella Home Page. `http://gnutella.wego.com`.

[Gon99] Li Gong. *Inside Java 2 Platform Security*. Addison Wesley, 1999.

[Gon01] L. Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5(3):88–95, May 2001.

[Gut99] Erik Guttman. Service location protocol: Automatic discovery of IP network services. *IEEE Internet Computing*, 3(4):71–80, July/August 1999.

[HBC01] J. P. Hubaux, L. Buttyan, and S. Capkun. The quest for security in mobile ad hoc networks. In *ACM Symposium on Mobile Ad Hoc Networking and Computing*, October 2001.

[HKM+98] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hof93] C. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, Computer Science Department, University of Maryland, 1993.

[HR01] R. Handorean and G.-C. Roman. Service provision in ad hoc networks. Technical Report WUCS-01-40, Washington University, Department of Computer Science, 2001.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[KS92] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[KT98] Neeran Karnik and Anand Tripathi. Design issues in mobile agent programming systems. *IEEE Concurrency*, pages 52–61, July-September 1998.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java Virtual Machine. *ACM SIGPLAN Notices*, 33(10):36–44, October 1998.

[Lig] LighTS Home Page. `http://lights.sourceforge.net`.

[LO98] Danny Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[LO99] Danny Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[LSH+00] Sung-Ju Lee, William Su, Julian Hsu, Mario Gerla, and Rajive Bagrodia. A performance comparison study of ad hoc wireless multicast protocols. In *Proceedings of IEEE INFOCOM 2000*, pages 565–574, March 2000.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[MPR01] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, May 2001.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[Mur00] Amy T. Murphy. *Enabling the Rapid Development of Dependable Applications in Mobile Environments*. PhD thesis, Washigton University, St. Louis, August 2000.

[Nas] Naspter Home Page. `http://www.napster.com`.

[NFP98] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998.

[Obj95] Object Management Group. CORBA: Architecture and specification, 1995.

[Per00] Charles Perkins. *Ad Hoc Networking*. Addison-Wesley, 2000.

[Pic98] Gian Pietro Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. In *Proceedings of Mobile Agents: Second International Workshop*, volume 1477 of *Lecture Notes on Computer Science*, pages 160–171. Springer-Verlag, September 1998.

[Pie96] Benjamin C. Pierce. Foundational calculi for programming languages. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*, chapter 139. CRC Press, 1996.

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. Lime: Linda meets mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering*, pages 368–377. ACM Press, May 1999.

[RBP01] Chrislain Razafimahefa, Ciarán Bryce, and Michel Pawlak. The Lana approach to wireless computing. In *Proceedings 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. IEEE Computer Society Press, May 2001.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[RL98] Geraldo Robson and Antônio Alfredo Loureiro. *Introdução à Computação Móvel.* Décima Primeira Escola de Computação, 1998.

[RPM00] Gruia-Catalin Roman, Gian Pietro Picco, and Amy L. Murphy. Software Engineering for Mobility: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 241–258. ACM Press, 2000.

[RTVH01] Roger Riggs, Antero Taivalsaari, Mark Vandenbrink, and Jim Holliday. *Programming Wireless Devices with the Java 2 Platform Micro Edition.* Addison-Wesley, 2001.

[Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *ACM Symposium on Principles of Distributed Computing*, May 1996.

[SG90] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[Sie96] J. Siegel. *CORBA: Fundamentals and Programming.* John Wiley & Sons, 1996.

[SMK+01] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM Conference*, pages 149–160, 2001.

[Sta01] Tom Standage. The Internet, untethered. *The Economist*, October 2001.

[Sun98] Sun Microsystems. Java Remote Method Invocation Specification, October 1998.

[Sun00] Sun Microsystems. Java 2 Plataform Micro Edition Technology for Creating Mobile Devices, May 2000.

[Szy98] Clemens Szyperski. *Component Software*. Addison Wesley, 1998.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.

[TG01] Robert Tolksdorf and Dirk Glaubitz. XMLSpaces for coordination in web-based systems, 2001. Submitted for publication.

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *ACM SIGMOD Conference on Management of Data*, pages 321–330, 1992.

[TKV+99] Anand Tripathi, Neeran Karnik, Manish Vora, Tanvir Ahmed, and Ram D. Singh. Ajanta - a mobile agent programming system. Technical Report TR98-016 (revised version), Department of Computer Science, University of Minnesota, 1999.

[VB99] Jan Vitek and Ciarán Bryce. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99) and Third International Symposium on Mobile Agents (MA'99)*, October 1999.

[VBBL01a] Marco Túlio Valente, Roberto Bigonha, Mariza Bigonha, and Antônio Alfredo Loureiro. Mobilidade de grupos de objetos em um sistema para programação distribuída na internet. In *V Simpósio Brasileiro de Linguagens de Programação*, pages 221–235, May 2001.

[VBBL01b] Marco Túlio Valente, Roberto Bigonha, Mariza Bigonha, and Antônio Alfredo Loureiro. Supporting disconnected operation in a mobile object system. In *7th ECOOP Workshop on Mobile Object Systems*, June 2001.

[VBLB99] Marco Túlio Valente, Roberto Bigonha, Antônio Alfredo Loureiro, and Mariza Bigonha. Linguagens para computação móvel na Internet. *Revista de Informática Teórica e Aplicada*, 6(2):7–47, December 1999.

[VBLB00] Marco Túlio Valente, Roberto Bigonha, Antônio Alfredo Loureiro, and Mariza Bigonha. Introduzindo abstrações para computação móvel em linguagens orientadas por objeto. In *IV Simpósio Brasileiro de Linguagens de Programação*, pages 15–28, May 2000.

[VV00] U. Varshney and R. Vetter. Emerging mobile and wireless networks. *Communications of the ACM*, 43(6):73–81, June 2000.

[W3C98] World Wide Web Consortium W3C. Extensible markup language (XML) 1.0, 1998.

[Wal99] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[Whi97] James E. White. Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*, pages 437–472. AAAI Press/MIT Press, 1997.

[WMLF98] P. Wycko, S. W. McLaughry, T. J. Lehman, and D. A. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, August 1998.