Uma Linguagem para Coordenação de Aplicações em Redes Móveis Ad Hoc

Fernando Magno Q. Pereira[1], **Marco Túlio de Oliveira Valente**[2], **Roberto S. Bigonha**[1] **e Mariza A. S. Bigonha**[1]

[1] Departamento de Ciência da Computação,
Universidade Federal de Minas Gerais

[2]Departamento de Ciência da Computação,
Pontifícia Universidade Católica de Minas Gerais

{fernandm, bigonha, mariza}@dcc.ufmg.br, mtov@pucminas.br

Abstract. *This paper presents the language supported by the coordination model called PeerSpaces. PeerSpaces defines an infrastructure that supports process communication, synchronization and resource discovery in mobile ad hoc networks. The model embraces a coordination language based on the notion of tuple spaces, inspired by Linda. PeerSpaces, 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 ad hoc settings.*

Resumo. *Este artigo apresenta a linguagem usada no modelo de coordenação denominado PeerSpaces. PeerSpaces define uma infra-estrutura para comunicação entre processos, sincronização e localização de recursos em redes móveis ad hoc. O modelo de coordenação proposto em PeerSpaces é baseado no conceito de espaços de tuplas, proposto originalmente em Linda. PeerSpaces, no entanto, substitui a arquitetura cliente/servidor, tradicionalmente usada em implementações de Linda, por uma arquitetura peer-to-peer. Argumenta-se no artigo que esta arquitetura é mais apropriada para coordenação de aplicações em cenários de computação móvel e ad hoc.*

1 Introdução

Redes de comunicação móveis passaram por várias transformações nos últimos anos [Varshney and Vetter, 2000]. Estas transformações contribuíram para tornar dispositivos computacionais tais como assistentes pessoais digitais (PDAs) e *laptops* cada vez mais baratos e poderosos. A possibilidade de tornar o acesso à informação independente do local e do momento contribuiu para popularizar o emprego de computadores sem fio em tarefas antes destinadas às redes fixas e permitiu o surgimento de aplicações exclusivas para ambientes móveis, como *m-commerce* e serviços de localização [Robson and Loureiro, 1998].

Embora o número de computadores móveis em utilização e o poder computacional destes dispositivos tenham crescido vertiginosamente, a tecnologia de *software* empregada em redes móveis não sofreu progresso similar. A maioria das linguagens, abstrações e

arquiteturas utilizadas em sistemas sem fio são idênticas àquelas originalmente projetadas para suportar a comunicação em ambientes fixos.

A principal falha detectada nos sistemas projetados originalmente para redes fixas é o tratamento inadequado de eventos típicos de redes móveis como, por exemplo, flutuações na largura de banda e a alta freqüência de desconexões. Ademais, abstrações projetadas para redes locais em geral dão origem a aplicações baseadas no modelo cliente/servidor, que pressupõe, via de regra, um estilo de comunicação síncrono e uma topologia estática, na qual a localização dos elementos comunicantes é bem conhecida. Este modelo não é, entretanto, o mais adequado para redes móveis, uma vez que nodos podem não estar conectados à rede quando requisições forem feitas a eles. A fim de satisfazer as necessidades de uma rede móvel, é necessário um modelo capaz de suportar comunicação assíncrona e de lidar com o contexto de execução dinâmico de uma rede deste tipo.

O desafio de desenvolver sistemas para redes móveis torna-se ainda maior quando estas não possuem qualquer infra-estrutura fixa de apoio. Tal é o caso das chamadas redes *ad hoc* [Perkins, 2000], onde a comunicação acontece apenas via o meio sem fio. Redes *ad hoc* são compostas por nodos autônomos, que desempenham tanto o papel de clientes como de servidores e roteadores. Em uma rede *ad hoc*, as conexões entre os nodos são simétricas e determinadas pela distância.

Uma das questões mais desafiadoras no projeto de sistemas para redes *ad hoc*, diz respeito às políticas de coordenação adotadas nos mesmos. Por coordenação entendem-se os mecanismos disponibilizados por uma linguagem distribuída para comunicação entre processos, para localização de serviços e para sincronização de atividades. Segundo esta definição, computação e coordenação são conceitos ortogonais [Carriero and Gelernter, 2001]. Computação diz respeito à construção de programas individuais, por meio de linguagens tradicionais, como Java, Haskell ou Prolog. Coordenação trata de como estes programas devem interagir para realizar tarefas.

Sendo a topologia de uma rede móvel *ad hoc* bastante dinâmica, uma linguagem de coordenação projetada para este tipo de ambiente deve permitir que a comunicação entre processos seja possível mesmo que receptores e transmissores não estejam conectados simultaneamente à rede. Tal linguagem também precisa disponibilizar mecanismos para localização de recursos que sejam independentes da topologia de rede, já que nodos podem se conectar e desconectar a qualquer momento.

Neste artigo é descrita uma linguagem para coordenação de aplicações em redes móveis *ad hoc*, a qual é baseada no conceito de *espaço de tuplas* da linguagem Linda [Gelernter, 1985]. Tal linguagem faz parte de um modelo denominado PeerSpaces, desenvolvido para permitir a comunicação entre os dispositivos computacionais que integram uma rede *ad hoc*.

O restante deste artigo está organizado conforme descrito a seguir. A Seção 2 conceitua espaços de tuplas e descreve a linguagem Linda. A Seção 3 apresenta em mais detalhes a linguagem de coordenação proposta neste artigo. A implementação de um sistema capaz de simular a utilização desta linguagem em uma rede fixa é descrita na Seção 4. A Seção 5 contém uma análise a respeito da linguagem de coordenação proposta. Na Seção 6 são discutidas outras linguagens que podem ser utilizadas para programação em ambientes móveis e a Seção 7 apresenta as conclusões do artigo.

2 Linguagens Baseadas em Espaços de Tuplas

A linguagem de coordenação proposta baseia-se em uma estrutura de dados denominada espaço de tuplas. Esta estrutura de armazenamento de dados foi originalmente empregada pela linguagem Linda [Gelernter, 1985] para coordenar a execução de processos paralelos. Em Linda, espaços de tuplas são usados de maneira similar a um quadro de mensagens, onde informações, ou seja, tuplas, podem ser inseridas, removidas ou simplesmente lidas pelos processos comunicantes. Mais formalmente, espaços de tuplas podem ser caracterizados como multiconjuntos cujos elementos são tuplas, as quais são formadas por uma sequência ordenada de campos, cada um deles podendo armazenar valores de determinado tipo de dados. Tuplas são comparadas entre si com base na aridade e no valor armazenado em seus campos, não sendo levado em consideração, para fins de comparação, o momento em que foram inseridas no espaço de tuplas.

2.1 A Linguagem Linda

Proposta por Gelernter, em meados da década de 80, Linda foi a primeira linguagem de coordenação a utilizar o conceito de espaço de tuplas. Em Linda processos se comunicam criando, lendo e removendo tuplas desta estrutura compartilhada. O acesso ao espaço compartilhado se dá via as três primitivas descritas a seguir:

- A primitiva $\mathbf{out}\ t$ insere assincronamente a tupla t no espaço de tuplas.
- A primitiva $\mathbf{in}\ t, x$ 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 é não deterministicamente escolhida; caso nenhuma tupla seja encontrada, a operação fica bloqueada até que uma tupla compatível com t seja inserida.
- A primitiva $\mathbf{rd}\ t, x$ comporta-se como um $\mathbf{in}$, 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 [Gelernter, 1985]. Processos em execução, quando necessitam referir-se a uma tupla, fornecem uma descrição parcial, isto é, um padrão, para a tupla desejada. Compatibilidade entre tuplas e padrões baseia-se na aridade e nos valores ou tipo de dados de seus campos. 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 tupla no espaço persistente, a qual somente será recuperada mais tarde por um processo consumidor. Não é necessária, portanto, a existência de uma conexão temporal entre processos para que eles possam interagir. Já a distribuição no espaço é possível graças ao fato de que tuplas depositadas no repositório compartilhado são visíveis nos diversos nodos de um sistema distribuído.

Em Linda, o uso de padrões para ler e remover tuplas permite que processos se comuniquem sem que conheçam mutuamente seus identificadores. Por exemplo, processos consumidores são capazes de remover tuplas com base no conteúdo das mesmas, sem que necessariamente tenham conhecimento da identidade dos processos produtores. Todas estas características têm motivado o surgimento de novas implementações de Linda, como JavaSpaces [Freeman et al., 1999] da Sun, e TSpaces [Wycko et al., 1998] da IBM.

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 empregada em redes móveis, 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*.

3 O Modelo de Coordenação PeerSpaces

PeerSpaces é o modelo utilizado neste artigo para coordenação de aplicações em redes sem fio. Ele é particularmente interessante para redes *ad hoc* porque não pressupõe a existência de nenhuma estrutura centralizada. A linguagem de coordenação proposta em PeerSpaces difere de implementações tradicionais de Linda por não ser baseada em uma arquitetura cliente/servidor, mas sim em um modelo de comunicação *peer-to-peer*. Em PeerSpaces, assume-se que cada nodo possui seu próprio espaço de tuplas, o qual pode ser utilizado tanto por aplicações locais a este elemento, como por aplicações remotas. Dessa forma, nodos conectados desempenham tanto papéis de clientes como de servidores e podem, eventualmente, comportar-se como roteadores de mensagens.

A linguagem de coordenação do modelo PeerSpaces utiliza primitivas semelhantes àquelas utilizadas por Linda. A maior contribuição de PeerSpaces ao conjunto de operações proposto em Linda é a definição de uma primitiva extra denominada **find**, a qual possui, basicamente, duas utilizações. A primeira delas é localizar serviços disponíveis em uma rede, como por exemplo impressoras ou sistemas de arquivos. A semântica desta primitiva foi definida de tal forma que esta pesquisa não exije qualquer forma de sincronização distribuída ou algum conhecimento prévio da 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.

3.1 Conceitos Principais

O modelo de coordenação proposto em PeerSpaces baseia-se em quatro conceitos principais: nodos, serviços, grupos e rede, os quais são descritos a seguir.

Nodos – Em PeerSpaces, todos os nodos de uma rede são considerados computadores móveis. Cada nodo possui, além de um processo em execução, um espaço de tuplas local. Este espaço de tuplas possui três tipos de usos. Primeiramente, ele é usado para coordenação entre os processos em execução no próprio nodo, como em Linda. Em segundo lugar, o espaço de tuplas é utilizado para troca de mensagens e sincronização entre processos que executam 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, o espaço de tuplas local permite a publicação de serviços disponíveis em um nodo e o armazenamento de resultados de consultas realizadas na rede.

Serviços – Um serviço é qualquer entidade disponível em um nodo que pode ser útil a outros nodos da rede como um dispositivo de *hardware* ou um *software*. Em PeerSpaces

serviços são tornados públicos na rede via a inserção de 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 e sua velocidade de impressão, entre outras características. A localização de um serviço é dada pelo nome do nodo que o disponibiliza. Uma busca pelos serviços disponíveis é uma consulta com intuito de descobrir a localização de um determinado serviço em algum nodo da rede.

Grupos – PeerSpaces assume que os nodos que integram uma rede *ad hoc* são logicamente organizados em grupos. Grupos possuem um nome, um conjunto de nodos e um conjunto de subgrupos e organizam-se hierarquicamente formando uma árvore. O grupo de um nodo é denotado por uma tupla da forma $\langle g_1, \ldots, g_n \rangle$, que 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. O conceito de grupos é útil para restringir o escopo de buscas realizadas na rede via a primitiva **find**.

Rede – Em PeerSpaces, nodos são conectados por meio de uma rede sem fio *ad hoc*. Nestas redes, a 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 os nodos podem funcionar também como roteadores, propagando mensagens entre elementos da rede que não estão diretamente conectados.

3.2 A Linguagem de Coordenação de PeerSpaces

A linguagem de coordenação usada em PeerSpaces define um conjunto de primitivas que permite o desenvolvimento de aplicações utilizando-se os conceitos descritos anteriormente. Estas primitivas podem ser classificadas em primitivas locais, primitivas remotas, primitivas para localização de serviços e primitivas para realização de consultas contínuas.

Primitivas Locais – O espaço de tuplas local de um nodo é manipulado via as três primitivas originalmente empregadas pela linguagem Linda para acesso à memória compartilhada (**in** , **out** e **rd**). Além destas três primitivas, a linguagem PeerSpaces disponibiliza aos desenvolvedores de aplicações uma quarta primitiva considerada de uso local: **chgrp** g, usada para alterar o grupo de um nodo para aquele especificado pela tupla g.

Primitivas Remotas – O projeto de primitivas para realização de operações remotas é crucial para escalabilidade e desempenho de qualquer linguagem de coordenação. Portanto, desde o início deste projeto, decidiu-se que a linguagem de coordenação utilizada em PeerSpaces não ofereceria nenhuma estrutura cujo objetivo fosse prover transparência de localização no acesso aos diversos espaços de tuplas da rede. Em vez disso, para a realização de operações remotas, a linguagem disponibiliza variantes das primitivas tradicionais de Linda que operam no espaço de tuplas de um nodo h previamente conhecido. São elas: **out** h, v; **in** h, v, x e **rd** h, v, x. Nestas primitivas, v denota uma tupla ou um padrão e x uma variável que poderá receber o valor de alguma tupla como resultado de uma operação de inspeção ao espaço de tuplas.

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, 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 $\mathbf{in}\, v_h$. 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 $\mathbf{in}\, h, v, x$ e $\mathbf{rd}\, h, v, x$ são síncronas e, portanto, bloqueiam a execução até que o nodo h esteja conectado e uma tupla compatível com v 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.

Primitivas para 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. Uma vez que não deve existir em PeerSpaces um serviço de localização centralizado em um único nodo, definiu-se, para a realização desta tarefa, a primitiva $\mathbf{find}\ g, p$, que procura em todos os 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: $\mathbf{find}\, \langle \texttt{mall}, \texttt{sellers} \rangle, \langle \texttt{tv}, \texttt{21}, ?, ?, ? \rangle$. Esta operação dispara uma consulta por tuplas compatíveis com o padrão $\langle \texttt{tv}, \texttt{21}, ?, ?, ? \rangle$ nos nodos da rede pertencentes ao grupo `sellers`, o qual é um subgrupo de `mall`. Tuplas compatíveis com este padrão, tais como $\langle \texttt{tv}, \texttt{21}, \texttt{foo}, \texttt{325}, \texttt{h} \rangle$, 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 $\mathbf{find}$ em um instante de tempo qualquer após a ativação da operação. O nodo solicitante pode então recuperar respostas a sua consulta usando a primitiva $\mathbf{in}$ e realizar uma oferta de compra por meio de uma operação $\mathbf{out}\, h, v$.

Embora a semântica de PeerSpaces não predefina um algoritmo específico para propagação de consultas de serviços, exige-se que qualquer implementação real desta operação atenda a dois requisitos: cobertura e ausência de ciclos. Cobertura diz respeito ao fato de o algoritmo 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 proíbe o algoritmo de gerar consultas que fiquem circulando indefinidamente na rede. Além desses requisitos, a propagação de consultas não deve exigir nenhum grau de sincronização distribuída. Basicamente a propagação de consultas deve concorrer e ser executada de forma intercalada com outras operações em espaços de tuplas.

Primitivas para 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. Isto evita 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 falhas 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 $\mathbf{find}\ g, p, t$, 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, as consultas contínuas existentes neste grupo e no nodo recém conectado devem ser sincronizadas. Em PeerSpaces, esta sincronização é realizada usando-se uma estratégia de melhor esforço similar àquela usada para propagar consultas normais de serviços. Por exemplo, suponha a conexão de um novo nodo h em um grupo g da rede. Qualquer consulta contínua localizada em h e que ainda não existe em g é então propagada para um dos nodos deste grupo. Este nodo propaga novamente a consulta para seus vizinhos, que voltam a propagá-la e assim sucessivamente, até que todos os nodos de g tenham recebido uma cópia da mesma. Isto também ocorre com consultas contínuas existentes em g e que não existem em h. Por último, 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.

4 Implementação

A fim de realizar pequenos experimentos com o modelo PeerSpaces, foi desenvolvido e implementado um protótipo capaz de emular o comportamento de uma rede *ad hoc* em um ambiente fixo. Tal protótipo foi codificado em Java e permite testar o comportamento das primitivas providas pela linguagem de coordenação descrita neste artigo.

4.1 Arquitetura Básica

No protótipo, cada nodo é representado por um objeto, cujo estado inclui o espaço de tuplas local e uma lista de outros nodos correntemente conectados à ele. As operações que podem ser invocadas sobre este objeto, por sua vez, implementam as primitivas providas pela linguagem de coordenação do modelo PeerSpaces. A tecnologia utilizada para permitir a comunicação entre os diversos nodos baseia-se em invocação remota de métodos, implementada com o auxílio do pacote Java RMI [Sun Microsystems, 1998].

Este protótipo permite que possam ser especificadas buscas, contínuas ou não, contendo padrões genéricos de tuplas. Tuplas genéricas contêm um ou mais campos preenchidos com indicações de tipos, e não com valores pertencentes a este tipo, como seria usual. Por exemplo, $p = (?, Int)$, especifica uma tupla de dois campos sendo o

segundo deles necessariamente do tipo inteiro, ao passo que o primeiro dos dois campos pode ter qualquer valor.

O algoritmo adotado para a propagação de consultas pesquisa a malha formada pelos nodos comunicantes segundo um padrão de busca em largura em grafos não direcionados. De acordo com esta técnica, também conhecida por *flooding* ou inundação [Tanenbaum, 1996], um nodo, tendo recebido uma consulta, propaga-a para os seus vizinhos, e tal processo repete-se até que todo o 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. Este é, no entanto, um caso extremamente particular, no qual o padrão de conexões se reduz a uma lista encadeada com n nodos. Normalmente cada elemento da rede é atingido após um número de propagações igual a $\log_k n$, onde $k \geq 2$ é a quantidade média de conexões de um nodo.

As operações de consulta à rede, realizadas pela primitiva $\mathbf{find}$, possuem identificadores, de modo a evitar ciclos no grafo determinado pelas interconexões entre os nodos comunicantes. Identificadores de consultas são gerados levando-se em conta o nome do nodo que criou a consulta e também o momento no qual ela foi criada. Uma operação de busca encerra-se assim que todos os elementos que compõem o grupo especificado na consulta tenham sido atingidos por ela. Um nodo alcançado por uma operação de busca deve propagá-la para os seus vizinhos na rede, a menos que tal elemento já tenha sido atingido por uma consulta com identificador idêntico.

LighTS [LighTS Home Page, 2002] foi a implementação de espaço de tuplas utilizada neste projeto. Existem outras implementações disponíveis, dentre as quais pode-se citar TSpaces [Wycko et al., 1998] e JavaSpaces [Freeman et al., 1999]. No entanto, elas tendem a consumir muitos recursos computacionais, uma vez que contêm mecanismos extras para garantir, por exemplo, segurança de transações e persistência de dados. Como LighTS implementa somente as operações definidas em Linda, além de algumas primitivas próprias, o desempenho deste sistema é surpreendentemente bom, e o tamanho de todo o pacote de *software* não excede mais que 11 *Kbytes*. LighTS é, portanto, uma biblioteca ideal para dispositivos móveis, como PDA's.

4.2 Implementação de Consultas Contínuas

A implementação do serviço de consultas contínuas foi responsável pela maior parte da complexidade do sistema. A fim de permitir que cada nodo mantivesse um registro de todas as consultas contínuas válidas em um dado momento, foi introduzido nos mesmos um *espaço de tuplas temporário*. Em tal área são inseridas as tuplas pelas quais existe alguma demanda na rede. As tuplas armazenadas nesta estrutura de dados são acrescidas de três campos extras: $(c_1, c_2, \ldots c_n) \Rightarrow (c_1, c_2, \ldots c_n, r, i, t)$. O campo r é uma referência remota para o nodo que iniciou a consulta, ao passo que t determina o instante de tempo no qual a consulta deixará de ser válida, devendo pois, a tupla temporária ser removida. O campo i, por sua vez, identifica a consulta, a fim de que não sejam inseridas consultas repetidas no espaço de tuplas temporário.

A implementação do serviço de consultas contínuas levou à modificação do procedimento para a inserção de uma nova tupla no espaço de tuplas de um nodo. A inserção de uma tupla p, via a primitiva $\mathbf{out}$ local, passou a implicar em uma consulta ao espaço de tuplas temporário, em busca por tuplas p', ali armazenadas, compatíveis com p. A existência

de uma ou mais tuplas compatíveis leva à inserção remota da tupla p nos espaços de tuplas dos nodos que esperam por ela. A referência r, presente nas tuplas p', permite que as inserções remotas sejam efetuadas com facilidade, pois tal referência indica o endereço do nodo que espera por tuplas compatíveis com p'.

Conforme discutido na Seção 3, a remoção de uma consulta contínua não poderia ser deixada a cargo do nodo que a iniciou, pois, como desconexões involuntárias são bastante comuns em redes móveis, tal nodo poderia não mais estar presente quando fosse necessário remover a busca. Assim, a cada consulta contínua é associado um período de tempo t (*time stamp*) durante o qual ela é válida. Expirado este período, ela é removida da rede. O intervalo de tempo t é indicado pelos usuários do sistema, via o comando que inicia a consulta contínua.

A remoção de consultas contínuas cujo período de existência expirou foi implementada com o auxílio de processos concorrentes (*threads*). Um novo processo concorrente é criado para cada consulta recém chegada a um nodo. Tal *thread* tem a única função de retirar a tupla procurada pela busca que a originou do espaço de tuplas temporário, terminando sua execução logo em seguida. Pode acontecer, portanto, de que em um nodo sejam criados diversos fluxos de execução diferentes. Todos os nodos conectados à rede em um dado momento contêm as mesmas consultas contínuas, logo o número de *threads* criadas será igual a $n \times q$, onde q representa o número de consultas contínuas mantidas na rede e n representa o número de nodos que compõem a rede naquele momento. A existência de um grande número de *threads* no sistema não compromete excessivamente a sua eficiência, uma vez que elas são mantidas suspensas pelo tempo que durarem as consultas contínuas que as originaram.

Na implementação de espaço de tuplas disponível no pacote LighTS não é possível inserir tuplas contendo padrões genéricos. Devido a este fato, a fim de permitir que em buscas contínuas pudessem ser especificadas tuplas com padrões genéricos, foi necessário desenvolver uma implementação própria de espaço de tuplas, para ser utilizada como o espaço temporário. Cuidou-se, contudo, para que esta implementação fosse a mais simples possível, de modo que ela contém apenas as três principais primitivas de Linda: **rd**, **in** e **out**.

Dentre as dificuldades inerentes à inclusão de consultas contínuas à linguagem de coordenação do modelo PeerSpaces, a mais séria diz respeito à manutenção da consistência dos estados da rede. Todos os nodos que compõem a rede devem manter sempre as mesmas buscas contínuas realizadas sobre ela. Os problemas mais complexos para o atendimento deste requisito surgem quando da inclusão de um ou mais nodos a uma rede existente. Após o estabelecimento da conexão é necessário que todos os nodos integrantes da nova rede contenham em seus espaços de tuplas temporários as mesmas consultas contínuas. Esta sincronização pode gerar um tráfego considerável de mensagens entre os elementos recém conectados, dado que estão sendo transferidas várias tuplas simultaneamente.

Outra questão que deve ser levada em consideração é a capacidade computacional de dispositivos móveis e limitações quanto à energia disponível. Tais restrições podem causar a rápida saturação do sistema devido ao excesso de consultas contínuas produzidas. Existem algumas sugestões para minimizar o impacto de pesquisas persistentes, como

a limitação do número de consultas contínuas que podem existir na rede em um dado momento e a restrição ao tempo máximo durante o qual elas são válidas. Pode-se, ainda, adotar políticas de prioridade entre as buscas, a fim de tornar o modelo mais justo.

O conceito de grupos de nodos hierarquicamente organizados descrito na Seção 3 visa restringir o número de elementos da rede alcançados por operações de busca. O protótipo apresentado nesta seção, contudo, não implementa o conceito de grupos.

As consultas contínuas, quando utilizadas, podem impor ao sistema computacional um custo bastante elevado, pois o espaço de tuplas temporário requer memória e a manutenção de estados sempre consistentes na rede pode levar a um grande tráfego de mensagens entre os nodos. Entretanto, caso este serviço não venha a ser utilizado, a sua simples existência não exige do dispositivo móvel qualquer recurso extra.

4.3 Interface com o Usuário

A interface que o sistema disponibiliza para o usuário é de linha de comando. Os diversos comandos disponíveis são listados logo a seguir.

- **connect** <**node**>: estabelece uma conexão bidirecional com o nodo especificado.
- **disconnect** <**node**>: suprime a conexão com o nodo especificado.
- **exit**: remove o nodo da rede simulada, invalidando, antes a conexão entre este nodo e todos os demais.
- **in** <**tup**> **[node]**: insere a tupla especificada no espaço de tuplas local, ou no espaço de tuplas de um nodo remoto, caso o nome deste tenha sido especificado.
- **inp** <**tup**> **[node]**: semelhante à operação **in**, porém não leva ao bloqueio do nodo caso a tupla especificada não possa ser obtida imediatamente.
- **out** <**tup**> **[node]**: insere a tupla especificada no espaço de tuplas local, ou, caso tenha sido fornecido um nome de nodo remoto, insere a tupla remotamente no espaço do nodo especificado.
- **find** <**tup**> **[time]**: inicia uma busca na rede pela tupla especificada. Esta busca será contínua se for especificado algum intervalo de tempo.
- **ngb**: fornece uma lista de todos os nodos conectados ao nodo corrente.
- **ts**: mostra o conteúdo do espaço de tuplas do nodo corrente.
- **fs**: mostra o conteúdo do espaço de tuplas temporário do nodo corrente, ou seja, este comando fornece uma indicação de todas as consultas contínuas que existem na rede em um dado momento.

Como é possível inferir da lista de comandos, a conexão e desconexão entre os nodos é feita explicitamente, via os comandos **connect** e **disconnect**, respectivamente. Em uma implementação real da linguagem PeerSpaces para redes móveis, as conexões serão determinadas pela distância entre os dispositivos comunicantes e acontecerão de forma transparente para o usuário.

5 Análise Crítica da Linguagem de Coordenação Apresentada

Esta seção analisa a linguagem de coordenação proposta neste artigo, segundo requisitos típicos de aplicações distribuídas para redes móveis *ad hoc*.

Aderência às Características de Redes *Ad hoc* – Os conceitos básicos 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. Por exemplo, a primitiva para localização de serviços (**find**) 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 – A linguagem de coordenação proposta em PeerSpaces não faz uso de qualquer estrutura de dados distribuída entre os nodos de uma rede sem fio. Em vez disso a linguagem oferece primitivas para acesso apenas a espaços de tuplas previamente conhecidos. Já a primitiva **find** segue uma estratégia de melhor esforço para propagar consultas de serviços, isto é, a mesma não requer nenhuma forma de sincronização distribuída. Por fim, o conceito de grupos lógicos de nodos fornece um mecanismo para restringir o escopo de buscas. Todas estas características contribuem para a escalabilidade do modelo, principalmente em ambientes sujeitos a constantes reconfigurações.

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 [Gnutella Home Page, 2002]. 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 situações de emergência, como em 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. 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 e flutuações na largura de banda. 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. Embora todas estas hipóteses sejam válidas em redes fixas, o mesmo não ocorre em redes sem fio. Dessa forma, não existem em PeerSpaces estruturas de mais alto nível que tenham como objetivo tornar transparente a existência da rede e dos problemas inerentes 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 em redes locais.

Tolerância a Falhas – Linguagens de programação distribuída tradicionais tratam falhas de comunicação como exceções. No entanto, como estas falhas são eventos comuns em redes sem fio, a linguagem de coordenação de PeerSpaces não segue a mesma estratégia. Esta linguagem suporta formas de comunicação assíncronas a fim de lidar com desconexões. São assíncronas as operações **out** e **find**. Caso não exista uma conexão com o nodo remoto, tais primitivas depositam uma mensagem no espaço local, a qual será

propagada para seu destino quando houver disponibilidade de conexão.

Reatividade – A capacidade de reagir a mudanças é uma característica importante em ambientes dinâmicos como redes *ad hoc*. Assim, a primitiva para consultas contínuas pode ser usada para detectar mudanças no estado de nodos remotos, simulando um modelo de eventos onde todo nodo funciona como originador e como receptor de eventos [Carzaniga et al., 2001]. A principal diferença desta estratégia para modelos tradicionais de tratamento 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. A inexistência do despachante se mostra apropriada pois como este nodo 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. É desejável que linguagens baseadas em espaços de tuplas incluam mecanismos para proteger tais espaços de computações maliciosas e errôneas. A linguagem proposta neste artigo, entretanto não dispõe de nenhum mecanismo de segurança de dados. Acredita-se, porém, que o sistema Secure Spaces [Bryce et al., 1999] 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.

6 Trabalhos Relacionados

Os recentes avanços na tecnologia de comunicação sem fio foram responsáveis por um grande incremento nas pesquisas de modelos, abstrações e linguagens adequadas para este tipo de ambiente. Nesta seção são descritos alguns dos principais modelos e linguagens propostos recentemente para programação em redes móveis.

Java RMI – Java RMI (*Remote Method Invocation*) [Sun Microsystems, 1998] 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.

O pacote Java RMI tem sido largamente utilizado no desenvolvimento de aplicações distribuídas, porém quando transportado para o ambiente móvel, esta biblioteca apresenta diversas limitações. Em primeiro lugar o sistema Java RMI assume a existência de um serviço centralizado de nomes. Esta solução inviabiliza o uso do sistema em redes *ad hoc*, pois, sendo as desconexões freqüentes, pode acontecer do diretório de nomes não estar disponível em algum momento, o que comprometeria o funcionamento de toda a rede. Além disso, Java RMI implementa apenas chamadas síncronas de métodos, não tolerando desconexões, pois invocações remotas ativam uma exceção caso exista qualquer falha de comunicação com a JVM remota.

Jini – Jini [Arnold, 2000] é um sistema para construção de aplicações distribuídas que utiliza Java RMI como infra-estrutura básica de comunicação, mas que possui um serviço de nomes próprio, chamado de *lookup service*. Jini utiliza um protocolo bastante flexível para registro, pesquisa e remoção de informações do diretório de nomes, o que torna este sistema adequado para redes que constantemente são reconfiguradas. Tal protocolo

possibilita que novos serviços possam ser acrescentados e removidos de uma aplicação sem necessidade de suporte por parte dos usuários da mesma.

Embora Jini seja uma extensão da linguagem Java voltada para redes dinâmicas em termos de topologia e recursos disponíveis, o serviço de nomes centralizado torna este sistema adequado apenas para programação em redes sem fio infra-estruturadas. 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.

Lime – A linguagem Lime [Murphy et al., 2001] é uma versão de Linda projetada para coordenação 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*. Aplicações em Lime são constituídas por *agentes móveis* [Valente et al., 1999]. Cada agente móvel possui seu próprio espaço de tuplas, que o acompanha caso este migre entre nodos da rede. Espaços de tuplas de agentes localizados em um mesmo nodo são fundidos pelo sistema em uma abstração chamada espaço de tuplas local, uma estrutura virtual e transiente. Virtual porque é criada pelo sistema a partir dos espaços físicos de cada agente móvel e transiente porque seu conteúdo altera-se conforme agentes migrem para o nodo ou o abandonem. Lime permite ainda que os espaços de tuplas locais de cada nodo conectado à rede sejam fundidos em um repositório global, e também virtual, denominado *espaço de tuplas federado*.

Lime disponibiliza um grande nível de transparência aos desenvolvedores de aplicações, pois estes têm a ilusão de que todas as tuplas distribuídas na rede encontram-se armazenadas em um mesmo local: o espaço de tuplas federado. Tal nível de abstração, entretanto, impõe ao sistema pesados custos em termos de complexidade de implementação e escalabilidade, de modo que a eficiência de aplicações pode cair drasticamente conforme aumente o número de agentes na rede [Carbunar et al., 2001a, Carbunar et al., 2001b].

7 Conclusão

O modelo PeerSpaces foi desenvolvido para coordenar aplicações distribuídas em dispositivos computacionais móveis conectados via uma rede *ad hoc*. O projeto da linguagem de coordenação que integra o modelo não observou os princípios da arquitetura cliente/servidor tradicional, que pressupõem a existência de servidores bem conhecidos, centralizados e globalmente acessíveis, pois considera-se que este modelo não é o mais adequado para uma rede deste tipo.

A fim de satisfazer os requisitos de uma rede *ad hoc*, como escalabilidade e tolerância a falhas, a linguagem apresentada baseia-se no conceito de espaços de tuplas, mantendo com isto um estilo de comunicação distribuída no tempo e no espaço. Neste modelo, cada computador móvel possui seu próprio espaço de tuplas, o qual é usado para prover comunicação assíncrona entre processos e para armazenar descrições dos serviços disponibilizados pelo mesmo. A linguagem descrita disponibiliza uma primitiva que permite que buscas sejam feitas na rede sem que sejam conhecidas informações acerca de sua topologia e cuja implementação não requer nenhuma forma de sincronização distribuída. Assim, embora tenha sacrificado transparência, a linguagem apresentada neste artigo é compatível com os princípios de comunicação de redes móveis *ad hoc*.

Referências

Arnold, K. (2000). *The Jini Specifications*. Addison-Wesley, 2nd edition.

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

Carbunar, B., Valente, M. T., and Vitek, J. (2001a). Corelime: a coordination model for mobile agents. In Picco, G. P., editor, *International Workshop on Concurrency and Coordination*, Electronic Notes in Theoretical Computer Science. Elsevier Science.

Carbunar, B., Valente, M. T., and Vitek, J. (2001b). Lime revisited. In Picco, G. P., editor, *5th IEEE International Conference on Mobile Agents*, volume 2240 of *Lecture Notes in Computer Science*, pages 54–69. Springer-Verlag.

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

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. (2001). Design and evaluation of a wide-area event notification servic e. *ACM Transactions on Computer Systems*, 19(3):332–383.

Freeman, E., Hupfer, S., and Arnold, K. (1999). *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA.

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

Gnutella Home Page (2002). `http://gnutella.wego.com`.

LighTS Home Page (2002). `http://lights.sourceforge.net`.

Murphy, A. L., Picco, G. P., and Roman, G.-C. (2001). Lime: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*.

Perkins, C. (2000). *Ad Hoc Networking*. Addison-Wesley.

Robson, G. and Loureiro, A. A. (1998). *Introdução à Computação Móvel*. Décima Primeira Escola de Computação.

Sun Microsystems (1998). Java Remote Method Invocation Specification.

Tanenbaum, A. S. (1996). *Computer Networks*. Prentice Hall, 3rd edition.

Valente, M. T., Bigonha, R., ureiro, A. A. L., and Bigonha, M. (1999). Linguagens para computação móvel na Internet. *Revista de Informática Teórica e Aplicada*, 6(2):7–47.

Varshney, U. and Vetter, R. (2000). Emerging mobile and wireless networks. *CACM*, 43(6):73–81.

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